ULTRIX-32 Supplementary Documents
Programmer

Order No. AA-MF07A-TE

ULTRIX-32 Operating System, Version 3.0

Digital Equipment Corporation

Copyright © 1984, 1988 by Digital Equipment Corporation.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

DEC	ULTRIX-32
DECUS	UNIBUS
MASSBUS	VAX
PDP	VMS
ULTRIX	VT
ULTRIX-11	digital™

UNIX is a trademark of AT&T Bell Laboratories.

Information herein is derived from copyrighted material as permitted under a license agreement with AT&T Bell Laboratories.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the Electrical Engineering and Computer Science Departments at the Berkeley Campus of the University of California for their role in its development.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. Digital Equipment Corporation acknowledges the following individuals and institutions for their role in its development:

"The UNIX Time-Sharing System": Copyright © 1974, Association for Computing Machinery, Inc. reprinted by permission. This is a revised version of an article that appeared in Communications of the ACM, 17, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presnted at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973. Acknowledgements: for their help and support, R.H. Canaday, R. Morris, M.D. McIlroy, and J.F. Ossanna.

"Advanced Editing on UNIX" acknowledgement: Ted Dolotta for his ideas and assistance.

"An Introduction to the UNIX Shell" acknowledgements: Dennis Ritchie, John Mashey and Joe Maranzano for their help and support.

"LEARN - Computer-Aided Instruction on UNIX" acknowledgements: for their help and support, M.E. Bittrich, J.L. Blue, S.I. Feldman, P.A. Fox, M.J. McAlpin, E.Z. Rothkopf, Don Jackowski, and Tom Plum.

"A System for Typesetting Mathematics" acknowledgements: J.F. Ossanna, A.V. Aho, and S.C. Johnson, for their ideas and assistance.

"A TROFF Tutorial" acknowledgements: J. F. Ossanna, Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman, for their help and support.

The document "The C Programming Language - Reference Manual" is reprinted, with minor changes, from "The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

"Make - A Program for Maintaining Computer Programs" ackowledgements: S.C. Johnson, and H. Gajewska, for their ideas and assistance.

"YACC: Yet Another Compiler-Compiler" acknowledgements: B.W. Kernighan, P.J. Plauger, S.I. Feldman, C. Imagna, M.E. Lesk, A. Snyder, C.B. Haley, D.M. Ritchie, M.O. Harris and Al Aho, for their ideas and assistance.

"Lex - A Lexical Analyzer Generator" acknowledgements: S.C. Johnson, A.V. Aho, and Eric Schmidt, for their help as originators of much of Lex, as well as debuggers of it.

The document "RATFOR - A Preprocessor for a Rational Fortran" is a revised and expanded version of the one published in Software - Practice and Experience, October 1975. The Ratfor described here is the one in use on UNIX and GCOS at A T & T Bell Laboratories. Acknowledgements: Dennis Ritchie, and Stuart Feldman, for their ideas and assistance.

"The M4 Macro Processor" acknowledgements: Rick Becker, John Chambers, Doug McIlroy, and Jim Weythman, for the help and support.

"BC - An Arbitrary Precision Desk-Calculator Language" acknowledgement: The compiler is written in YACC; its original version was written by S.C. Johnson.

"A Dial-Up Network of UNIX TM Systems" acknowledgements: G.L. Chesson, A.S. Cohen, J. Lions, and P.F. Long, for their suggestions and assistance.

Copyright © 1979, 1980 Regents of the University of California. Permission to copy these documents or any portion thereof as necessary for licensed use of the software is granted to licensees of this software, provided this copyright notice and statement of permission are included.

The document "Writing Tools - The STYLE and DICTION Programs" is copyrighted © 1979 by A T & T Bell Laboratories. Holders of a UNIX TM/32V software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

The document "The Programming Language EFL" is copyrighted © 1979 by A T & T Bell Laboratories. EFL has been approved for general release, so that one may copy it subject only to the restriction of giving proper acknowledgement to A T & T Bell Laboratories.

The documents "A Portable Fortran 77 Compiler" and "Fsck - The UNIX File System Check Program" are modifications of earlier documents which are copyrighted © 1979 by A T & T Bell Laboratories. Holders of a UNIX TM/32V software license are permitted to copy these documents, or any portion of them, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included. This manual reflects system enhancements made at Berkeley and sponsored in part by NSF Grants MCS-7807291, MCS-8005144, and MCS-74-07644-A04; DOE Contract DE-AT03-76SF00034 and Project Agreement DE-AS03-79ER10358; and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, monitored by Naval Electronics Systems Command under Contract No. N00039-80-K-0649.

"Ex Reference Manual" acknowledgements: Chuck Haley contributed greatly to the early development of ex. Bruce Englar encouraged the redesign which led to ex version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and UNIX systems.

"A Guide to the Dungeons of Doom" acknowledgements: Rogue was originally conceived by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added many new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance.

The document "The FRANZ LISP Manual" is copyrighted © 1980, 1981, 1983 by the Regents of the University of California. (exceptions: Chapters 13, 14 (first half), 15 and 16 have separate copyrights, as indicated. These are reproduced by permission of the copyright holders.) Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, and the copyright notice of the Regents, University of California, is given. All rights reserved. Work reported herein was supported in part by the U.S. Department of Energy, Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and the National Science Foundation under Grant No. MCS 7807291. MC68000 is a trademark of Motorola Semiconductor Products, Inc.

"The FRANZ LISP Manual" acknowledgements: Richard Fateman, Mike Curry, John Breedlove, Jeff Levinsky, Bill Rowan, Tom London, Keith Sklower, Kipp Hickman, Charles Koester, Mitch Marcus, Don Cohen, John Foderaro, and Kevin Layer.

The document "Berkeley Pascal User's Manual" is copyrighted © 1977, 1979, 1980, 1983 by W.N. Joy, S.L. Graham, C.B. Haley, M.K. McKusick, P.B. Kessler. The financial support of the first and second authors' work by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the first author's work by an IBM Graduate Fellowship are gratefully acknowledged.

"Introduction to the f77 I/O Library" acknowledgement: Peter J. Weinberger originally wrote the I/O/ Library at A T & T Bell Laboratories.

"Writing Papers with NROFF Using -ME", and "-ME Reference Manual" acknowledgements: Bob Epstein, Bill Joy, Larry Rowe, Ricki Blau, Pamela Humphrey, and Jim Joyce, for their ideas and assistance. UNIX, NROFF, and TROFF are trademarks of A T & T Bell Laboratories.

"Refer - A Bibliography System" acknowledgements: Mike Lesk of A T & T Bell Laboratories wrote the original refer software, including the indexing programs. Al Stanberger of the Forestry Department wrote the first version of addbib, then called bibin. Greg Shenaut of the Linguistics Department wrote the original versions of sortbib and roffbib.

"Screen Updating and Cursor Movement Optimization: A Library Package" acknowledgements: For their help and support, Bill Joy, Doug Merritt, Kurt Shoens, Ken Abrams, Alan Char, Mark Horton, and Joe Kalash.

"Disc Quotas in a UNIX Environment" acknowledgements: Sam Leffler and Kirk McKusick, for their

work on the quota code. The current disc quota system is loosely based on a very early scheme implemented at the University of New South Wales and Syndey University.

The document, "Fsck - The UNIX File System Check Program", is a revision by Marshall Kirk McKusick; T.J. Kowalski wrote the original paper. For their help and support, we thank Bill Joy, Sam Leffler, Robert Elz, Dennis Ritchie, Robert Henry, Larry A. Wehr, and Rick B. Brandt. Our sponsors were the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

"A Fast File System for UNIX" acknowledgements: William N. Joy, Samuel J. Leffler, Robert S. Fabry, Marshall Kirk McKusick, Robert Elz, Michael Powell, Peter Kessler, Rober Henry, and Dennis Ritchie. This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

"4.2BSD Networking Implementation Notes" acknowledgements: The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79]. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

"SENDMAIL - An Internetwork Mail Router" acknowledgements: For their ideas and assistance, Kurt Shoens, Bill Joy, Mark Horton, Erick Schmidt, Kirk McKusick, Marvin Solomon, Mike Stonebraker, and Bob Epstein. A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

BEFORE YOU START

This is the second volume of *ULTRIX-32 Supplementary Documents*, a three volume set that contains articles describing the ULTRIX-32 system. The authors are computer scientists and program developers at Bell Laboratories and the University of California at Berkeley. The articles explain the software tools and utilities available on your ULTRIX-32 system. They constitute most of the lore that enriches this operating system; topics range from getting started to the details of screen updating and cursor movement facilities.

Each volume in this set contains several parts, and each part begins with an introduction. The introduction to each part serves as a map that will help you find your way around in the documentation, allowing you to select articles that relate to your interest. Each introduction gives an overview of the material covered in the part and a description of the articles included. Most readers will not need to read all articles, since many articles cover parallel topics.

These articles provide authoritative and accurate information that is unavailable elsewhere. However, you should be aware that some of the information in some articles is dated. We include those articles because many of the concepts they develop are still current and important.

At the end of each volume in this set, you will find a master index identifying topics in all three volumes.

Topics in Volume II

The articles in this second volume deal with programming and support tools for programmers on the ULTRIX-32 system. Most of the authors assume that readers are familiar with one or more programming languages. For example, the articles on FORTRAN 77 are written for people who already know a standard version of FORTRAN.

"UNIX Programming - Second Edition," in Part 1 of this volume, tells how to write programs that cooperate with the operating system. Many readers will find it useful to read this article before going on to articles on the languages and utilities.

The articles in Part 2 deal with four languages and four preprocessors. The languages are:

- C
- FORTRAN 77
- Franz Lisp
- Pascal

The four preprocessors are:

- RATFOR
- EFL
- FP
- M4

Part 3, Supporting Tools, offers articles on three kinds of utilities:

- Program and library maintenance tools
- Program checking and debugging tools
- Compiler and preprocessor development tools

And the articles in Part 4, System Programming, cover topics such as:

- Inner workings of the ULTRIX-32 system
- System and kernel facilities available to user programs
- Assembly language (*as*)
- Screen manipulation functions
- The ULTRIX-32 line printer spooler

The features described in this volume provide the flexibility and programming power for which UNIX is famous. A good understanding of many of the concepts and procedures presented here is essential for efficient use of your ULTRIX-32 system.

BEFORE YOU START

PART 1: PROGRAMMING CONSIDERATIONS

UNIX PROGRAMMING

INTRODUCTION . . . 1-3
BASICS . . . 1-3
Program Arguments . . . 1-3
The "Standard Input" and "Standard Output" . . . 1-4
THE STANDARD I/O LIBRARY . . . 1-5
File Access . . . 1-5
Error Handling - Stderr and Exit . . . 1-7
Miscellaneous I/O Functions . . . 1-8
LOW-LEVEL I/O . . . 1-8
File Descriptors . . . 1-8
Read and Write . . . 1-9
Open, Creat, Close, Unlink . . . 1-10
Random Access - Seek and Lseek . . . 1-11
Error Processing . . . 1-12
PROCESSES . . . 1-12
The "System" Function . . . 1-12
Low-Level Process Creation - Execl and Execv . . . 1-13
Control of Processes - Fork and Wait . . . 1-14
Pipes . . . 1-14
SIGNALS - INTERRUPTS AND ALL THAT . . . 1-17
APPENDIX: THE STANDARD I/O LIBRARY . . . 1-21
General Usage . . . 1-21
Calls . . . 1-21

PART 2: LANGUAGES

THE C PROGRAMMING LANGUAGE REFERENCE MANUAL

INTRODUCTION . . . 2-5
LEXICAL CONVENTIONS . . . 2-5
Comments . . . 2-5
Identifiers (Names) . . . 2-5
Keywords . . . 2-5
Constants . . . 2-6
Integer Constants . . . 2-6
Explicit Long Constants . . . 2-6
Character Constants . . . 2-6
Floating Constants . . . 2-6
Strings . . . 2-6
Hardware Characteristics . . . 2-6
SYNTAX NOTATION . . . 2-7
WHAT'S IN A NAME? . . . 2-7

THE C PROGRAMMING LANGUAGE REFERENCE MANUAL *(continued)*

OBJECTS AND IVALUES 2-8
CONVERSIONS 2-8
Characters and Integers 2-8
Float and Double 2-8
Floating and Integral 2-8
Pointers and Integers 2-8
Unsigned 2-8
Arithmetic Conversions 2-8
EXPRESSIONS 2-9
Primary Expressions 2-9
Unary Operators 2-10
Multiplicative Operators 2-11
Additive Operators 2-11
Shift Operators 2-12
Relational Operators 2-12
Equality Operators 2-12
Bitwise AND Operator 2-12
Bitwise Exclusive OR Operator 2-12
Bitwise Inclusive OR Operator 2-13
Logical AND Operator 2-13
Logical OR Operator 2-13
Conditional Operator 2-13
Assignment Operators 2-13
Comma Operator 2-14
DECLARATIONS 2-14
Storage Class Specifiers 2-14
Type Specifiers 2-15
Declarators 2-15
Meaning of Declarators 2-15
Structure and Union Declarations 2-16
Initialization 2-18
Type Names 2-19
Typedef 2-20
STATEMENTS 2-20
Expression Statement 2-20
Compound Statements, or Block 2-20
Conditional Statement 2-21
While Statement 2-21
Do Statement 2-21
For Statement 2-21
Switch Statement 2-21
Break Statement 2-22
Continue Statement 2-22
Return Statement 2-22
Goto Statement 2-22
Labeled Statement 2-22
Null Statement 2-23

EXTERNAL DEFINITIONS 2-23
External Function Definitions 2-23
External Data Definitions 2-24
SCOPE RULES 2-24
Lexical Scope 2-24
Scope of Externals 2-24
COMPILER CONTROL LINES 2-25
Token Replacement 2-25
File Inclusion 2-25
Conditional Compilation 2-25
Line Control 2-26
IMPLICIT DECLARATIONS 2-26
TYPES REVISITED 2-26
Structures and Unions 2-26
Functions 2-26
Arrays, Pointers, and Subscripting 2-27
Explicit Pointer Conversions 2-27
CONSTANT EXPRESSIONS 2-28
PORTABILITY CONSIDERATIONS 2-28
ANACHRONISMS 2-29
SYNTAX SUMMARY 2-30
Expressions 2-30
Declarations 2-31
Statements 2-32
External Definitions 2-33
Preprocessor 2-33
RECENT CHANGES TO C 2-35
Structure Assignment 2-35
Enumeration Type 2-35

A TOUR THROUGH THE PORTABLE C COMPILER

INTRODUCTION 2-37
OVERVIEW 2-38
THE SOURCE FILES 2-39
DATA STRUCTURE CONSIDERATIONS 2-40
PASS ONE 2-41
LEXICAL ANALYSIS 2-41
PARSING 2-41
STORAGE CLASSES 2-42
SYMBOL TABLE MAINTENANCE 2-43
TREE BUILDING 2-44
INITIALIZATION 2-45
STATEMENTS 2-46
OPTIMIZATION 2-47
MACHINE DEPENDENT STUFF 2-47
FIRST PASS SUMMARY 2-49
PASS TWO 2-49
OVERVIEW 2-49
THE MACHINE MODEL 2-50

A TOUR THROUGH THE PORTABLE C COMPILER *(continued)*

GENERAL ORGANIZATION 2-50
THE TEMPLATES 2-53
THE TEMPLATE MATCHING ALGORITHM 2-54
REGISTER ALLOCATION 2-55
THE MACHINE DEPENDENT INTERFACE 2-56
THE REWRITING RULES 2-56
THE SETHI-ULLMAN COMPUTATION 2-58
REGISTER ALLOCATION 2-59
COMPILER BUGS 2-59
SUMMARY AND CONCLUSION 2-60

A TOUR THROUGH THE UNIX C COMPILER

THE INTERMEDIATE LANGUAGE 2-63
EXPRESSION OPTIMIZATION. 2-66
CODE GENERATION. 2-68
DELAYING AND REORDERING 2-76

INTRODUCTION TO THE F77 I/O LIBRARY

FORTRAN I/O 2-79
Types of I/O 2-79
Direct Access 2-79
Sequential Access 2-79
List Directed I/O 2-79
Internal I/O 2-80
I/O Execution 2-80
IMPLEMENTATION DETAILS 2-80
Number of Logical Units 2-80
Standard Logical Units 2-80
Vertical Format Control. 2-81
The Open Statement 2-81
Format Interpretation. 2-81
List Directed Output 2-82
I/O Errors 2-82
NON-"ANSI STANDARD" EXTENSIONS. 2-82
Format Specifiers. 2-82
Print Files 2-83
Scratch Files 2-83
List Directed I/O 2-83
RUNNING OLDER PROGRAMS 2-83
Traditional Unit Control Parameters. 2-83
Preattachment of Logical Units 2-84
MAGNETIC TAPE I/O 2-84
CAVEAT PROGRAMMER 2-84
APPENDIX A: I/O LIBRARY ERROR MESSAGES. 2-85
APPENDIX B: EXCEPTIONS TO THE ANSI STANDARD 2-88

A PORTABLE FORTRAN 77 COMPILER

INTRODUCTION 2-89

Usage 2-89
Documentation Conventions 2-90
Implementation Strategy 2-91

LANGUAGE EXTENSIONS 2-91

Double Complex Data Type 2-91
Internal Files 2-91
Implicit Undefined Statement 2-91
Recursion 2-91
Automatic Storage 2-91
Source Input Format 2-92
Include Statement 2-92
Binary Initialization Constants 2-92
Character Strings 2-92
Hollerith 2-93
Equivalence Statements 2-93
One-Trip DO Loops 2-93
Commas in Formatted Input 2-93
Short Integers 2-93
Additional Intrinsic Functions 2-94

VIOLATIONS OF THE STANDARD 2-94

Double Precision Alignment 2-94
Dummy Procedure Arguments 2-94
T and TL Formats 2-94
Carriage Control 2-94
Assigned Goto 2-95

INTER-PROCEDURE INTERFACE 2-95

Procedure Names 2-95
Data Representations 2-95
Return Values 2-95
Argument Lists 2-96

FILE FORMATS 2-96

Structure of Fortran Files 2-96
Portability Considerations 2-97
Pre-Connected Files and File Positions 2-97

A PORTABLE FORTRAN 77 COMPILER *(continued)*

APPENDIX A: DIFFERENCES BETWEEN FORTRAN 66 AND FORTRAN 77 2-98
Features Deleted from Fortran 66 2-98
Hollerith 2-98
Extended Range 2-98
Program Form 2-98
Blank Lines 2-98
Program and Block Data Statements 2-98
ENTRY Statement 2-98
DO Loops 2-99
Alternate Returns 2-99
Declarations 2-99
CHARACTER Data Type 2-99
IMPLICIT Statement 2-99
PARAMETER Statement 2-100
Array Declarations 2-100
SAVE Statement 2-100
INTRINSIC Statement 2-100
Expressions 2-100
Character Constants 2-100
Concatenation 2-101
Character String Assignment 2-101
Substrings 2-101
Exponentiation 2-101
Relaxation of Restrictions 2-101
Executable Statements 2-102
IF-THEN-ELSE 2-102
Alternate Returns 2-102
Input/Output 2-102
Format Variables 2-102
END=, ERR=, and IOSTAT= Clauses 2-103
Formatted I/O 2-103
Character Constants 2-103
Positional Editing Codes 2-103
Colon 2-103
Optional Plus Signs 2-104
Blanks on Input 2-104
Unrepresentable Values 2-104
Iw.m 2-104
Floating Point 2-104
"A" Format Code 2-104
Standard Units 2-104
List-Directed Formatting 2-105
Direct I/O 2-105
Internal Files 2-105

OPEN, CLOSE, and INQUIRE Statements 2-106
OPEN 2-106
CLOSE 2-106
INQUIRE 2-106
APPENDIX B: REFERENCES AND BIBLIOGRAPHY 2-109

RATFOR: A PREPROCESSOR FOR A RATIONAL FORTRAN

INTRODUCTION 2-111
LANGUAGE DESCRIPTION 2-111
Design 2-111
Statement Grouping 2-112
The "Else" Clause 2-112
Nested If's 2-113
If-Else Ambiguity. 2-113
The "Switch" Statement 2-114
The "Do" Statement 2-114
"Break" and "Next" 2-115
The "While" Statement. 2-115
The "For" Statement 2-116
The "Repeat-Until" Statement 2-117
More on Break and Next 2-117
"Return" Statement 2-117
Cosmetics 2-117
Free-Form Input 2-117
Translation Services 2-118
"Define" Statement. 2-118
"Include" Statement 2-118
Pitfalls, Botches, Blemishes and Other Failings. 2-119
IMPLEMENTATION 2-119
EXPERIENCE 2-120
Good Things 2-120
Bad Things. 2-120
CONCLUSIONS 2-121
APPENDIX: USAGE ON UNIX AND GCOS 2-122

THE PROGRAMMING LANGUAGE EFL

INTRODUCTION 2-123
Purpose 2-123
History. 2-123
Notation 2-123
LEXICAL FORM 2-124
Character Set 2-124
Lines 2-124
White Space. 2-124
Comments 2-124
Include Files 2-124
Continuation 2-124
Multiple Statements on a Line 2-125

THE PROGRAMMING LANGUAGE EFL *(continued)*

Tokens 2-125
Identifiers 2-125
Strings 2-125
Integer Constants 2-126
Floating Point Constants 2-126
Punctuation 2-126
Operators 2-126
Macros 2-126
PROGRAM FORM 2-127
Files 2-127
Procedures 2-127
Blocks 2-127
Statements 2-127
Labels 2-128
DATA TYPES AND VARIABLES 2-128
Basic Types 2-128
Constants 2-128
Variables 2-129
Storage Class 2-129
Scope of Names 2-129
Precision 2-129
Arrays 2-129
Structures 2-130
EXPRESSIONS 2-130
Primaries 2-130
Constants 2-131
Variables 2-131
Array Elements 2-131
Structure Members 2-131
Procedure Invocations 2-131
Input/Output Expressions 2-132
Coercions 2-132
Sizes 2-132
Parentheses 2-132
Unary Operators 2-132
Arithmetic 2-133
Logical 2-133
Binary Operators 2-133
Arithmetic 2-133
Logical 2-133
Relational Operators 2-134
Assignment Operators 2-134
Dynamic Structures 2-134
Repetition Operator 2-135
Constant Expressions 2-135

DECLARATIONS 2-135
Syntax 2-135
Attributes 2-135
Basic Types 2-135
Arrays 2-136
Structures 2-136
Precision 2-136
Common 2-136
External 2-137
Variable List 2-137
The Initial Statement 2-137
EXECUTABLE STATEMENTS 2-137
Expression Statements 2-137
Subroutine Call 2-137
Assignment Statements 2-138
Blocks 2-138
Test Statements 2-138
If Statement 2-138
If-Else 2-138
Select Statement 2-139
Loops 2-139
While Statement 2-139
For Statement 2-139
Repeat Statement 2-140
Repeat...Until Statement 2-140
Do Loops 2-140
Branch Statements 2-141
Goto Statement 2-141
Break Statement 2-141
Next Statement 2-142
Return 2-142
Input/Output Statements 2-142
Input/Output Units 2-142
Binary Input/Output 2-143
Formatted Input/Output 2-143
Iolists 2-143
Formats 2-143
Manipulation Statements 2-144
PROCEDURES 2-144
Procedure Statement 2-144
End Statement 2-145
Argument Association 2-145
Execution and Return Values 2-145
Known Functions 2-145
Minimum and Maximum Functions 2-145
Absolute Value 2-145
Elementary Functions 2-145
Other Generic Functions 2-146

THE PROGRAMMING LANGUAGE EFL *(continued)*

ATAVISMS 2-146
Escape Lines 2-146
Call Statement 2-146
Obsolete Keywords 2-146
Numeric Labels 2-146
Implicit Declarations 2-147
Computed Goto 2-147
Go To Statement 2-147
Dot Names 2-147
Complex Constants 2-148
Function Values 2-148
Equivalence 2-148
Minimum and Maximum Functions 2-148
COMPILER OPTIONS 2-148
Default Options 2-149
Input Language Options 2-149
Input/Output Error Handling 2-149
Continuation Conventions 2-149
Default Formats 2-149
Alignments and Sizes 2-149
Default Input/Output Units 2-150
Miscellaneous Output Control Options 2-150
EXAMPLES 2-150
File Copying 2-150
Matrix Multiplication 2-150
Searching a Linked List 2-150
Walking a Tree 2-151
PORTABILITY 2-153
Primitives 2-153
Character String Copying 2-153
Character String Comparisons 2-154
APPENDIX A: RELATION BETWEEN EFL AND RATFOR 2-155
APPENDIX B: COMPILER 2-155
Current Version 2-155
Diagnostics 2-155
Quality of Fortran Produced 2-155
APPENDIX C: CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE 2-156
External Names 2-157
Procedure Interface 2-157
Pointers 2-157
Recursion 2-157
Storage Allocation 2-157

BERKELEY PASCAL USER'S MANUAL

SOURCES OF INFORMATION 2-160

Where To Get Documentation 2-160
Documentation Describing UNIX 2-160
Text Editing Documents 2-161
Pascal Documents: The language 2-161
Pascal Documents: The Berkeley Implementation 2-162
References 2-162

BASIC UNIX PASCAL 2-165

A First Program 2-165
A Larger Program 2-168
Correcting the First Errors 2-169
Executing the Second Example 2-171
Formatting the Program Listing 2-173
Execution Profiling 2-173

ERROR DIAGNOSTICS 2-177

Translator Syntax Errors 2-177
Translator Semantic Eerrors 2-180
Translator Panics, I/O Errors 2-184

INPUT/OUTPUT 2-186

Introduction 2-186
Eof and Eoln 2-187
More about Eoln 2-188
Output Buffering 2-189
Files, Reset, and Rewrite 2-190
Argc and Argv 2-190

DETAILS ON THE COMPONENTS OF THE SYSTEM 2-193

Options 2-193
Options Common to Pi, Pc, and Pix 2-193
Options Available in Pi 2-195
Options Available in Px 2-195
Options Available in Pc 2-195
Options Available in Pxp 2-196
Formatting Programs using Pxp 2-197
Pxref 2-199
Multi-File Programs 2-199
Separate Compilation with Pc 2-199

APPENDIX TO WIRTH'S PASCAL REPORT 2-202

Extensions to the Language Pascal 2-202
Resolution of the Undefined Specifications 2-203
Restrictions and Limitations 2-206
Added Types, Operators, Procedures and Functions 2-207
Remarks on Standard and Portable Pascal 2-208

THE FRANZ LISP MANUAL

INTRODUCTION 2-211
Data Types. 2-211
Lispval 2-212
Symbol 2-212
List. 2-212
Binary 2-213
Fixnum 2-213
Flonum 2-213
Bignum 2-213
String. 2-214
Port. 2-214
Vector 2-214
Array 2-214
Value 2-215
Hunk 2-215
Other 2-215
Documentation 2-215
DATA STRUCTURE ACCESS. 2-217
Lists 2-217
List Creation 2-217
List Predicates 2-219
List Accessing 2-219
List Manipulation 2-221
Predicates 2-223
Symbols and Strings 2-226
Symbol and String Creation 2-226
String and Symbol Predicates 2-228
Symbol and String Accessing. 2-228
Symbol and String Manipulation 2-229
Vectors. 2-231
Vector Creation 2-231
Vector Reference. 2-231
Vector Modification 2-232
Arrays 2-232
Array Creation 2-232
Array Predicate 2-233
Array Accessors 2-233
Array Manipulation 2-234
Hunks 2-235
Hunk Creation 2-235
Hunk Accessor. 2-236
Hunk Manipulators 2-236
Bcds 2-236

Structures 2-237
Assoc List 2-237
Property List 2-238
Tconc Structure 2-240
Fclosures 2-240
Random Functions 2-241
ARITHMETIC FUNCTIONS 2-244
Simple Arithmetic Functions 2-244
Predicates 2-245
Trignometric Functions 2-247
Bignum Functions 2-247
Bit Manipulation 2-248
Other Functions 2-248
SPECIAL FUNCTIONS 2-251
INPUT/OUTPUT 2-266
SYSTEM FUNCTIONS 2-275
THE LISP READER 2-287
Introduction 2-287
Syntax Classes 2-287
Reader Operations 2-288
Character Classes 2-288
Syntax Classes 2-291
Character Macros 2-293
Types 2-293
Normal 2-293
Splicing 2-294
Infix 2-294
Invocations 2-295
Functions 2-296
FUNCTIONS, FCLOSURES, AND MACROS 2-297
Valid Function Objects 2-297
Functions 2-297
Macros 2-297
Macro Forms 2-299
Defmacro 2-299
The Backquote Character Macro 2-299
Sharp Sign Character Macro 2-300
Conditional Inclusion 2-300
Fixnum Character Equivalents 2-301
Read Time Evaluation 2-301
Fclosures 2-302
An Example 2-302
Useful Functions 2-303
Internal Structure 2-304
Foreign Subroutines and Functions 2-304

THE FRANZ LISP MANUAL *(continued)*

ARRAYS AND VECTORS 2-309
General Arrays 2-309
Subparts of an Array Object 2-310
Access Function 2-310
Auxiliary 2-310
Data 2-310
Length 2-310
Delta 2-310
The Maclisp Compatible Array Package 2-310
Vectors. 2-311
Anatomy of Vectors. 2-312
Size. 2-312
Property 2-312
Internal Order. 2-312
Immediate-Vectors 2-312
EXCEPTION HANDLING 2-314
Errset and Error Handler Functions 2-314
The Anatomy of an Error 2-314
Error Handling Algorithm. 2-314
Default Aids 2-315
Autoloading 2-315
Interrupt Processing 2-316
THE JOSEPH LISTER TRACE PACKAGE 2-317
LISZT - THE LISP COMPILER. 2-321
General Strategy of the Compiler 2-321
Running the Compiler 2-321
Special Forms 2-321
Macro Expansion 2-321
Classification 2-322
Using the Compiler 2-323
Compiler Options. 2-324
Autorun 2-326
Pure Literals 2-327
Transfer Tables. 2-327
Fixnum Functions 2-328
THE CMU USER TOPLEVEL AND THE FILE PACKAGE 2-329
User Command Input Top Level. 2-329
The File Package 2-330
THE LISP STEPPER 2-334
Simple Use of Stepping 2-334
Advanced Features 2-335
Selectively Turning On Stepping 2-335
Stepping with Breakpoints 2-336
Overhead of Stepping. 2-336
Evalhook and Funcallhook 2-336
THE FIXIT DEBUGGER 2-338

Introduction . . . 2-338
Interaction with Trace . . . 2-340
Interaction with Step . . . 2-340
Multiple Error Levels . . . 2-340

THE LISP EDITOR . . . 2-341

The Editors . . . 2-341
Scope of Attention . . . 2-341
Pattern Matching Commands . . . 2-342

Commands That Search . . . 2-343

Location Specifications . . . 2-344

The Edit Chain . . . 2-345

Printing Commands . . . 2-345
Structure Modification Commands . . . 2-345
Extraction and Embedding Commands . . . 2-346
Move and Copy Commands . . . 2-347
Parentheses Moving Commands . . . 2-347

Using To and Thru . . . 2-348

Undoing Commands . . . 2-348
Commands That Evaluate . . . 2-349
Commands That Text . . . 2-349
Editor Macros . . . 2-350
Miscellaneous Editor Commands . . . 2-351
Editor Functions . . . 2-351

APPENDIX A: SPECIAL SYMBOLS . . . 2-354
APPENDIX B: SHORT SUBJECTS . . . 2-357

The Garbage Collector . . . 2-357
Debugging . . . 2-357
The Interpreter's Top Level . . . 2-358

BERKELEY FP USER'S MANUAL

BACKGROUND . . . **2-359**
SYSTEM DESCRIPTION . . . **2-361**

Objects . . . **2-361**
Application . . . **2-361**
Functions . . . 2-362

Structural . . . 2-363
Predicate (Test) Functions . . . 2-364
Arithmetic/Logical . . . 2-364
Library Routines . . . 2-365

Functional Forms . . . 2-365
User Defined Functions . . . 2-367

GETTING ON AND OFF THE SYSTEM . . . 2-368

Comments . . . 2-368
Breaks . . . 2-368
Non-Termination . . . 2-368

BERKELEY FP USER'S MANUAL *(continued)*

SYSTEM COMMANDS 2-368
Load 2-368
Save 2-368
Csave and Fsave 2-368
Cload 2-369
Pfn 2-369
Delete 2-369
Fns 2-369
Stats 2-369
On 2-370
Off 2-370
Print 2-370
Reset 2-370
Trace 2-371
Timer 2-371
Script 2-371
Help 2-371
Special System Functions 2-372
Lisp 2-372
Debug 2-372
PROGRAMMING EXAMPLES 2-373
MergeSort 2-373
FP Session 2-375
IMPLEMENTATION NOTES 2-381
The Top Level 2-381
The Scanner 2-381
The Parser 2-381
The Code Generator 2-382
Function Definition and Application 2-383
Function Naming Conventions 2-383
Measurement Impelementation 2-383
Data Structures 2-383
Interpretation of Data Structures 2-384
Times 2-384
Size 2-384
Funargno 2-384
Funargtyp 2-384
Trace Information 2-384
APPENDIX A: LOCAL MODIFICATIONS 2-386
Character Set Changes 2-386
Syntactic Modifications 2-386
While and Conditional 2-386
Function Definitions 2-386
Sequence Construction 2-386
User Interface 2-387
Additions and Ommissions 2-387

APPENDIX B: FP GRAMMAR 2-388
APPENDIX C: COMMAND SYNTAX 2-389
APPENDIX D: TOKEN-NAME CORRESPONDENCES 2-390
APPENDIX E: SYMBOLIC PRIMITIVE FUNCTION NAMES 2-391

THE M4 MACRO PROCESSOR

INTRODUCTION 2-393
USAGE 2-393
DEFINING MACROS 2-393
QUOTING 2-394
ARGUMENTS 2-395
ARITHMETIC BUILT-INS 2-395
FILE MANIPULATION 2-396
SYSTEM COMMAND 2-396
CONDITIONALS 2-397
STRING MANIPULATION 2-397
PRINTING 2-397
SUMMARY OF BUILT-INS 2-398

PART 3: SUPPORTING TOOLS

AWK: A PATTERN SCANNING AND PROCESSING LANGUAGE

INTRODUCTION 3-5
Usage 3-5
Program Structure 3-5
Records and Fields 3-5
Printing 3-6
PATTERNS 3-6
BEGIN and END 3-6
Regular Expressions 3-7
Relational Expressions 3-7
Combinations of Patterns 3-7
Pattern Ranges 3-7
ACTIONS 3-7
Built-In Functions 3-8
Variables, Expressions, and Assignments 3-8
Field Variables 3-8
String Concatenation 3-9
Arrays 3-9
Flow-of-Control Statements 3-9
DESIGN 3-9
IMPLEMENTATION 3-10

MAKE: A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS

INTRODUCTION . . . 3-13
BASIC FEATURES . . . 3-13
DESCRIPTION FILES AND SUBSTITUTIONS . . . 3-15
COMMAND USAGE . . . 3-16
IMPLICIT RULES . . . 3-17
EXAMPLE . . . 3-18
SUGGESTIONS AND WARNINGS . . . 3-20
APPENDIX: SUFFIXES AND TRANSFORMATION RULES . . . 3-21

AN INTRODUCTION TO THE SOURCE CODE CONTROL SYSTEM

INTRODUCTION . . . 3-23
LEARNING THE LINGO . . . 3-23
- S-file . . . 3-23
- Deltas . . . 3-24
- SID's (or, Version Numbers) . . . 3-24
- Id keywords . . . 3-24

CREATING FILES . . . 3-24
GETTING FILES FOR COMPILATION . . . 3-25
CHANGING FILES (OR, CREATING DELTAS) . . . 3-25
- Getting a Copy To Edit . . . 3-25
- Merging the Changes Back into the S-File . . . 3-25
- When To Make Deltas . . . 3-26
- What's Going On: The Info Command . . . 3-26
- ID Keywords . . . 3-26
  - The What Command . . . 3-26
  - Where To Put ID Keywords . . . 3-27
- Keeping SID's Consistent Across Files . . . 3-27
- Creating New Releases . . . 3-27

RESTORING OLD VERSIONS . . . 3-27
- Reverting to Old Versions . . . 3-27
- Selectively Deleting Old Deltas . . . 3-28

AUDITING CHANGES . . . 3-28
- The Prt Command . . . 3-28
- Finding Why Lines Were Inserted . . . 3-29
- Finding What Changes You Have Made . . . 3-29

SHORTHAND NOTATIONS . . . 3-29
- Delget . . . 3-29
- Fix . . . 3-29
- Unedit . . . 3-29
- The -d Flag . . . 3-30

USING SCCS ON A PROJECT . . . 3-30
SAVING YOURSELF . . . 3-30
- Recovering a Munged Edit File . . . 3-30
- Restoring the S-File . . . 3-30

USING THE ADMIN COMMAND . . . 3-31

MAINTAINING DIFFERENT VERSIONS (BRANCHES) 3-31
Creating a Branch 3-31
Merging a Branch Back into the Main Trunk 3-31
A More Detailed Example. 3-32
A Warning 3-32
USING SCCS WITH MAKE. 3-32
To Maintain Single Programs 3-33
To Maintain a Library 3-33
To Maintain a Large Program. 3-34
Further Information. 3-35
QUICK REFERENCE 3-36
Commands. 3-36
Id Keywords 3-37

LINT, A C PROGRAM CHECKER

INTRODUCTION AND USAGE 3-39
A WORD ABOUT PHILOSOPHY 3-39
UNUSED VARIABLES AND FUNCTIONS 3-39
SET/USED INFORMATION. 3-40
FLOW OF CONTROL. 3-40
FUNCTION VALUES 3-41
TYPE CHECKING 3-41
TYPE CASTS. 3-42
NONPORTABLE CHARACTER USE 3-42
ASSIGNMENTS OF LONGS TO INTS 3-42
STRANGE CONSTRUCTIONS 3-43
ANCIENT HISTORY 3-43
POINTER ALIGNMENT 3-44
MULTIPLE USES AND SIDE EFFECTS 3-44
IMPLEMENTATION 3-44
PORTABILITY 3-45
SHUTTING LINT UP. 3-46
LIBRARY DECLARATION FILES 3-47
BUGS, ETC. 3-47
APPENDIX: CURRENT LINT OPTIONS 3-50

A TUTORIAL INTRODUCTION TO ADB

INTRODUCTION 3-51
A QUICK SURVEY 3-51
Invocation 3-51
Current Address 3-51
Formats 3-52
General Request Meanings 3-52
DEBUGGING C PROGRAMS 3-53
Debugging a Core Image 3-53
Multiple Functions 3-54
Setting Breakpoints. 3-55
Advanced Breakpoint Usage. 3-56
Other Breakpoint Facilities 3-58

A TUTORIAL INTRODUCTION TO ADB *(continued)*

MAPS 3-58
ADVANCED USAGE 3-59
Formatted Dump 3-59
Directory Dump 3-61
Ilist Dump 3-61
Converting Values 3-61
PATCHING 3-62
ANOMALIES 3-62
ADB SUMMARY 3-77

YACC: YET ANOTHER COMPILER-COMPILER

INTRODUCTION 3-79
BASIC SPECIFICATIONS 3-81
ACTIONS 3-83
LEXICAL ANALYSIS 3-84
HOW THE PARSER WORKS 3-86
AMBIGUITY AND CONFLICTS 3-89
PRECEDENCE 3-92
ERROR HANDLING 3-94
THE YACC ENVIRONMENT 3-96
HINTS FOR PREPARING SPECIFICATIONS 3-97
Input Style 3-97
Left Recursion 3-97
Lexical Tie-Ins 3-98
Reserved Words 3-98
ADVANCED TOPICS 3-99
Simulating Error and Accept in Actions 3-99
Accessing Values in Enclosing Rules 3-99
Support for Arbitrary Value Types 3-99
APPENDIX A: A SIMPLE EXAMPLE 3-102
APPENDIX B: YACC INPUT SYNTAX 3-104
APPENDIX C: AN ADVANCED EXAMPLE 3-106
APPENDIX D: OLD FEATURES SUPPORTED BUT NOT ENCOURAGED 3-111

LEX: A LEXICAL ANALYZER GENERATOR

INTRODUCTION 3-113
LEX SOURCE 3-115
LEX REGULAR EXPRESSIONS 3-115
Operators 3-115
Character Classes 3-116
Arbitrary Character 3-116
Optional Expression 3-116
Repeated Expressions 3-116
ALTERNATION AND GROUPING 3-116
Context Sensitivity 3-116
Repetitions and Definitions 3-117

LEX ACTIONS 3-117
AMBIGUOUS SOURCE RULES 3-119
LEX SOURCE DEFINITIONS 3-120
USAGE 3-120
UNIX 3-121
GCOS 3-121
TSO 3-121
LEX AND YACC 3-121
EXAMPLES 3-121
LEFT CONTEXT SENSITIVITY 3-123
CHARACTER SET 3-124
SUMMARY OF SOURCE FORMAT 3-124
CAVEATS AND BUGS 3-125

PART 4: System Programming

UNIX IMPLEMENTATION

INTRODUCTION 4-5
PROCESS CONTROL 4-5
Process Creation and Program Execution 4-6
Swapping 4-7
Synchronization and Scheduling 4-7
I/O SYSTEM 4-8
Block I/O System 4-9
Character I/O System 4-9
Disk Drivers 4-9
Character Lists 4-10
Other Character Devices 4-10
THE FILE SYSTEM 4-10
File System Implementation 4-11
Mounted File Systems 4-13
Other System Functions 4-13

4.2BSD System Manual

NOTATION AND TYPES 4-15
KERNEL PRIMITIVES 4-16
Processes and protection 4-17
Host and Process Identifiers 4-17
Process Creation and Termination 4-17
User and Group Ids 4-18
Process Groups 4-19
Memory Management 4-20
Text, Data and Stack 4-20
Mapping Pages 4-20
Page Protection Control 4-21
Giving and Getting Advice 4-21

4.2BSD System Manual

Signals 4–22
Overview 4–22
Signal Types 4–22
Signal Handlers 4–23
Sending Signals 4–23
Protecting Critical Sections 4–24
Signal Stacks 4–24
Timers 4–25
Real Time 4–25
Interval Time 4–25
Descriptors 4–27
The Reference Table 4–27
Descriptor Properties 4–27
Managing Descriptor References 4–27
Multiplexing Requests 4–27
Descriptor Wrapping 4–28
Resource Controls 4–30
Process Priorities 4–30
Resource Utilization 4–30
Resource Limits 4–31
System Operation Support 4–32
Bootstrap Operations 4–32
Shutdown Operations 4–32
Accounting 4–32
SYSTEM FACILITIES 4–33
Generic Operations 4–34
Read and Write 4–34
Input/Output Control 4–34
Nonblocking and Asynchronous Operations 4–35
File System 4–36
Overview 4–36
Naming 4–36
Creation and Removal 4–36
Directory Creation and Removal 4–36
File Creation 4–37
Creating References to Devices 4–37
Portal Creation 4–37
File, Device, and Portal Removal 4–38
Reading and Modifying File Attributes 4–38
Links and Renaming 4–39
Extension and Truncation 4–39
Checking Accessibility 4–41
Locking 4–41
Disk Quotas 4–41

Interprocess Communications 4-42
Interprocess Communication Primitives 4-42
Communication Domains 4-42
Socket Types and Protocols 4-42
Socket Creation, Naming and Service Establishment 4-42
Accepting Connections 4-43
Making Connections 4-43
Sending and Receiving Data. 4-44
Scatter/Gather and Exchanging Access Rights 4-44
Using Read and Write with Sockets 4-45
Shutting Down Halves of Full-Duplex Connections 4-45
Socket and Protocol Options 4-45
UNIX Domain. 4-45
Types of Sockets 4-45
Naming 4-45
Access Rights Transmission 4-46
INTERNET Domain. 4-46
Socket Types and Protocols 4-46
Socket Naming 4-46
Access Rights Transmission 4-46
Raw Access. 4-46
Terminals and Devices 4-47
Terminals 4-47
Terminal Input 4-47
Input Modes 4-47
Interrupt Characters 4-47
Line Editing. 4-47
Terminal Output 4-47
Terminal Control Operations 4-47
Terminal Hardware Support. 4-48
Structured Devices. 4-48
Unstructured Devices 4-48
Process and Kernel Descriptors 4-49
SUMMARY OF FACILITIES 4-50

BERKELEY VAX/UNIX ASSEMBLER REFERENCE MANUAL

INTRODUCTION 4-53
Assembler Revisions Since November 5, 1979. 4-53
Features Supported, But No Longer Encouraged as of February 9, 1983 4-53
USAGE 4-53
LEXICAL CONVENTIONS 4-54
Identifiers 4-54
Constants 4-54
Scalar Constants 4-54
Floating Point Constants. 4-55
String Constants. 4-55

BERKELEY VAX/UNIX ASSEMBLER REFERENCE MANUAL *(continued)*

Operators 4–55
Blanks 4–55
Scratch Mark Comments 4–55
"C" Style Comments 4–56

SEGMENTS AND LOCATION COUNTERS 4–56
STATEMENTS 4–56

Named Global Labels 4–56
Numeric Local Labels 4–56
Null Statements 4–57
Keyword Statements 4–57

EXPRESSIONS 4–57

Expression Operators 4–57
Data Types 4–57

TYPE PROPAGATION IN EXPRESSIONS 4–58
PSEUDO-OPERATIONS (DIRECTIVES) 4–59

Interface to a Previous Pass 4–59
Location Counter Control 4–60
Filled Data 4–60
Symbol Definitions 4–61
Initialized Data 4–61

MACHINE INSTRUCTIONS 4–63

Character Set 4–63
Specifying Displacement Lengths 4–63
Casex Instructions 4–64
Extended Branch Instructions 4–64

DIAGNOSTICS 4–64
LIMITS 4–64
ANNOYANCES AND FUTURE WORK 4–65

THE UNIX I/O SYSTEM

DEVICE CLASSES 4–67
OVERVIEW OF I/O 4–67
CHARACTER DEVICE DRIVERS 4–68
THE BLOCK-DEVICE INTERFACE 4–70
BLOCK DEVICE DRIVERS 4–72
RAW BLOCK-DEVICE I/O 4–73

SCREEN UPDATING AND CURSOR MOVEMENT OPTIMIZATION

OVERVIEW 4–75
TERMINOLOGY (OR, WORDS YOU CAN SAY TO SOUND BRILLIANT) 4–75
COMPILING THINGS 4–75
SCREEN UPDATING 4–76

Naming Conventions 4–76

VARIABLES 4–77

USAGE 4-77
Starting Up 4-77
The Nitty-Gritty 4-78
Output 4-78
Input 4-78
Miscellaneous 4-78
Finishing up 4-78
CURSOR MOTION OPTIMIZATION: STANDING ALONE 4-78
Terminal Information 4-79
Movement Optimizations, or, Getting Over Yonder 4-80
THE FUNCTIONS 4-80
Output Functions 4-80
Input Functions 4-84
Miscellaneous Functions 4-85
Details 4-87
APPENDIX A 4-89
Capabilities from Termcap 4-89
Disclaimer 4-89
Overview 4-89
Variables Set By Setterm() 4-89
Variables Set By Gettmode() 4-90
APPENDIX B 4-91
The WINDOW structure 4-91
APPENDIX C 4-92
Examples 4-92
Screen Updating 4-92
Twinkle 4-92
Life 4-94
Motion Optimization 4-97
Twinkle 4-97

4.2BSD LINE PRINTER SPOOLER MANUAL

OVERVIEW 4-99
COMMANDS 4-99
LPD - Line Printer Dameon 4-99
LPQ - Show Line Printer Queue 4-100
LPRM - Remove Jobs from a Queue 4-100
LPC - Line Printer Control Program 4-100
ACCESS CONTROL 4-100
SETTING UP 4-101
Creating a Printcap File 4-101
Printers on Serial Lines 4-101
Remote Printers 4-101
Output Filters 4-102

4.2BSD LINE PRINTER SPOOLER MANUAL *(continued)*

OUTPUT FILTER SPECIFICATIONS 4–102
LINE PRINTER ADMINISTRATION 4–103
TROUBLESHOOTING 4–103
LPR 4–103
LPQ 4–104
LPRM 4–105
LPD 4–105
LPC 4–105

PART 1: PROGRAMMING CONSIDERATIONS

This part contains one article, "UNIX Programming - Second Edition," by Kernighan and Ritchie. The article gives background information that will help you write programs that make full use of the ULTRIX-32 system. Readers should be familiar with the fundamentals of the ULTRIX-32 system (or the UNIX system). Although the techniques shown in the article apply to programming in any language available on the ULTRIX-32 system, the sample programs are written in the C language.

The authors explain how to:

- Pass arguments to and from a program
- Send program output to a file, to a pipe, or to a terminal
- Use the standard I/O (input/output) library
- Handle I/O errors
- Use low level I/O
- Execute a program from within another
- Handle signals (interrupts)

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function **main** as an argument count **argc** and an array **argv** of pointers to character strings that contain the arguments. By convention, **argv[0]** is the command name itself, so **argc** is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the **echo** command.)

```
main(argc, argv)  /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing **argv[1]** and loops until it has printed them all.

The argument count and the arguments are parameters to **main**. If you want to keep them around so other routines can get at them, you must copy them to external variables.

UNIX is a Trademark of Bell Laboratories

2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function **getchar** returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if **prog** uses **getchar**, then the command line

```
prog <file
```

causes **prog** to read **file** instead of the terminal. **prog** itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the

```
otherprog | prog
```

provides the standard input for **prog** from the standard output of **otherprog**.

getchar returns the value **EOF** when it encounters the end of file (or an error) on whatever you are reading. The value of **EOF** is normally defined to be **-1**, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, **putchar(c)** puts the character **c** on the "standard output," which is also by default the terminal. The output can be captured on a file by using >: if **prog** uses **putchar**,

```
prog >outfile
```

writes the standard output on **outfile** instead of the terminal. **outfile** is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of **prog** into the standard input of **otherprog**.

The function **printf**, which formats output in various ways, uses the same mechanism as **putchar** does, so calls to **printf** and **putchar** may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function **scanf** provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. **scanf** uses the same mechanism as **getchar**, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with **getchar**, **putchar**, **scanf**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of EOF.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called FILE. The only declaration needed for a file pointer is exemplified by

```
FILE  *fp, *fopen();
```

This says that `fp` is a pointer to a FILE, and `fopen` returns a pointer to a FILE. (FILE is a type name, like `int`, not a structure tag.

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("`r`"), write ("w"), or append ("a").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well

(like trying to read a file when you don't have permission). If there is any error, **fopen** will return the null pointer value **NULL** (which is defined as zero in **stdio.h**).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which **getc** and **putc** are the simplest. **getc** returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in **c** the next character from the file referred to by **fp**; it returns **EOF** when it reaches end of file. **putc** is the inverse of **getc**:

```
putc(c, fp)
```

puts the character **c** on the file **fp** and returns **c**. **getc** and **putc** return **EOF** on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called **stdin**, **stdout**, and **stderr**. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. **stdin**, **stdout** and **stderr** are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type **FILE *** can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)  /* wc: count lines, words, chars */
int argc;
char *argv[];
{
   int c, i, inword;
   FILE *fp, *fopen();
   long linect, wordct, charct;
   long tlinect = 0, twordct = 0, tcharct = 0;

   i = 1;
   fp = stdin;
   do {
      if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
         fprintf(stderr, "wc: can't open %s\0, argv[i]);
         continue;
      }
      linect = wordct = charct = inword = 0;
      while ((c = getc(fp)) != EOF) {
         charct++;
         if (c == '\0')
            linect++;
         if (c == ' ' || c == '\t' || c == '\0')
            inword = 0;
         else if (inword == 0) {
            inword = 1;
            wordct++;
         }
      }
      printf("%7ld %7ld %7ld", linect, wordct, charct);
      printf(argc > 1 ? " %s\0 : "\0, argv[i]);
      fclose(fp);
      tlinect += linect;
      twordct += wordct;
      tcharct += charct;
   } while (++i < argc);
   if (argc > 2)
      printf("%7ld %7ld %7ld total\0, tlinect, twordct, tcharct);
   exit(0);
}
```

The function **fprintf** is identical to **printf**, save that the first argument is a file pointer that specifies the file to be written.

The function **fclose** is the inverse of **fopen**; it breaks the connection between the file pointer and the external name that was established by **fopen**, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call **fclose** on an output file — it flushes the buffer in which **putc** is collecting output. (**fclose** is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

stderr is assigned to a program in the same way that **stdin** and **stdout** are. Output written on **stderr** appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on **stderr** instead of **stdout** so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function **exit** to terminate program execution. The argument of **exit** is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls **fclose** for each open output file, to flush out any buffered output, then calls a routine named **exit**. The function **exit** causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with **putc**, etc., is buffered (except to **stderr**); to force it out immediately, use **fflush(fp)**.

fscanf is identical to **scanf**, except that its first argument is a file pointer (as with **fprintf**) that specifies the file from which the input comes; it returns EOF at end of file.

The functions **sscanf** and **sprintf** are identical to **fscanf** and **fprintf**, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for **sscanf** and into it for **sprintf**.

fgets(buf, size, fp) copies the next line from **fp**, up to and including a newline, into **buf**; at most **size-1** characters are copied; it returns NULL at end of file. **fputs(buf, fp)** writes the string in **buf** onto file **fp**.

The function **ungetc(c, fp)** "pushes back" the character **c** onto the input stream **fp**; a subsequent call to **getc**, **fscanf**, etc., will encounter **c**. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a

program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called **read** and **write**. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n read = read(fd, buf, n);

n written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than **n** bytes remained to be read. (When the file is a terminal, **read** normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and **-1** indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define  BUFSIZE  512    /* best size for PDP-11 UNIX */

main()    /* copy input to output */
{
   char  buf[BUFSIZE];
   int   n;

   while ((n = read(0, buf, BUFSIZE)) > 0)
      write(1, buf, n);
   exit(0);
}
```

If the file size is not a multiple of **BUFSIZE**, some **read** will return a smaller number of bytes to be written by **write**; the next call to **read** after that will return zero.

It is instructive to see how **read** and **write** can be used to construct higher level routines like **getchar**, **putchar**, etc. For example, here is a version of **getchar** which does unbuffered input.

```
#define  CMASK 0377   /* for making char's > 0 */

getchar()    /* unbuffered single character input */
{
   char c;

   return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared **char**, because **read** accepts a character pointer. The character being returned must be masked with **0377** to ensure that it is positive; otherwise sign extension may make it negative. (The constant **0377** is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of **getchar** does input in big chunks, and hands out the characters one at a time.

```
#define  CMASK 0377   /* for making char's > 0 */
#define  BUFSIZE  512

getchar()    /* buffered version */
{
   static char buf[BUFSIZE];
   static char *bufp = buf;
   static int  n = 0;

   if (n == 0) {  /* buffer is empty */
      n = read(0, buf, BUFSIZE);
      bufp = buf;
   }
   return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, **open** and **creat** [sic].

open is rather like the **fopen** discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an **int**.

```
int fd;

fd = open(name, rwmode);
```

As with **fopen**, the **name** argument is a character string corresponding to the external file name. The access mode argument is different, however: **rwmode** is 0 for read, 1 for write, and 2 for read and write access. **open** returns **-1** if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to **open** a file that does not exist. The entry point **creat** is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called **name**, and **-1** if not. If the file already exists, **creat** will truncate it to zero length; it is not an error to **creat** a file that already exists.

If the file is brand new, **creat** creates it with the *protection mode* specified by the **pmode** argument. In the UNIX file system, there are nine bits of protection information

associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)  /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
   int   f1, f2, n;
   char  buf[BUFSIZE];

   if (argc != 3)
      error("Usage: cp from to", NULL);
   if ((f1 = open(argv[1], 0)) == -1)
      error("cp: can't open %s", argv[1]);
   if ((f2 = creat(argv[2], PMODE)) == -1)
      error("cp: can't create %s", argv[2]);

   while ((n = read(f1, buf, BUFSIZE)) > 0)
      if (write(f2, buf, n) != n)
         error("cp: write error", NULL);
   exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
   printf(s1, s2);
   printf("0);
   exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`,

which is taken relative to the location specified by **origin**. Subsequent reading or writing will begin at that position. **offset** is a **long**; **fd** and **origin** are **int**'s. **origin** can be 0, 1, or 2 to specify that **offset** is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the 0L argument; it could also be written as **(long) 0**.

With **lseek**, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
   lseek(fd, pos, 0);   /* get to pos */
   return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called **seek**. **seek** is identical to **lseek**, except that its **offset** argument is an **int** rather than a **long**. Accordingly, since PDP-11 integers have only 16 bits, the **offset** specified for **seek** is limited to 65,535; for this reason, **origin** values of 3, 4, 5 cause **seek** to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret **origin** as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has **origin** equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell **errno**. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine **perror** will print a message associated with the value of **errno**; more generally, **sys_errno** is an array of character strings which can be indexed by **errno** and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine **system**. **system** takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of **sprintf** may be useful.

Remember than **getc** and **putc** normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use **fflush**; for input, see **setbuf** in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's **system** routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine **execl**. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to **execl** is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a placeholder. If the command takes arguments, they are strung out after this; the end of the list is marked by a **NULL** argument.

The **execl** call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an **execl** call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where **date** is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'0);
```

A variant of **execl** called **execv** is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where **argp** is an array of pointers to the arguments; the last pointer in the array must be **NULL** so **execv** can tell where the list ends. As with **execl**, **filename** is the file in which the program is found, and **argp[0]** is the name of the program. (This arrangement is identical to the **argv** array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [ ] in the argument list. If you want these, use **execl** to invoke the shell **sh**, which then does all the work. Construct a string **commandline** that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /bin/sh. Its argument -c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl or execv. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork:

```
proc id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc id, the "process id." In one of these processes (the "child"), proc id is zero. In the other (the "parent"), proc id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);/* in child */
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
int status;

if (fork() == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork nor the exec calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of **ls** to the standard input of **pr**. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call **pipe** creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int   fd[2];

stat = pipe(fd);
if (stat == -1)
   /* there was an error ... */
```

fd is an array of two file descriptors, where **fd[0]** is the read side of the pipe and **fd[1]** is for writing. These may be used in **read**, **write** and **close** calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent **read** will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called **popen(cmd, mode)**, which creates a process **cmd** (just as **system** does), and returns a file descriptor that will either read or write that process, according to **mode**. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the **pr** command; subsequent **write** calls using the file descriptor **fout** will send their data to that process through the pipe.

popen first creates the the pipe with a **pipe** system call; it then **forks** to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via **execl**) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define  READ  0
#define  WRITE 1
#define  tst(a, b)    (mode == READ ? (b) : (a))
static   int    popen pid;

popen(cmd, mode)
char  *cmd;
int   mode;
{
   int p[2];

   if (pipe(p) < 0)
      return(NULL);
   if ((popen pid = fork()) == 0) {
      close(tst(p[WRITE], p[READ]));
      close(tst(0, 1));
      dup(tst(p[READ], p[WRITE]));
      close(tst(p[READ], p[WRITE]));
      execl("/bin/sh", "sh", "-c", cmd, 0);
       exit(1);    /* disaster has occurred if we get here */
   }
   if (popen pid == -1)
      return(NULL);
   close(tst(p[READ], p[WRITE]));
   return(tst(p[WRITE], p[READ]));
}
```

The sequence of **close**s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first **close** closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The **close** closes file descriptor 0, that is, the standard input. **dup** is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the **dup** is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function **pclose** to close the pipe created by **popen**. The main reason for using a separate function rather than **close** is that it is desirable to wait for the termination of the child process. First, the return value from **pclose** indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the **wait** lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)  /* close pipe fd */
int fd;
{
   register r, (*hstat)(), (*istat)(), (*qstat)();
   int    status;
   extern int popen pid;

   close(fd);
   istat = signal(SIGINT, SIG IGN);
   qstat = signal(SIGQUIT, SIG IGN);
   hstat = signal(SIGHUP, SIG IGN);
   while ((r = wait(&status)) != popen pid && r != -1);
   if (r == -1)
      status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   signal(SIGHUP, hstat);
   return(status);
}
```

The calls to **signal** make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable **popen pid**; it really should be an array indexed by file descriptor. A **popen** function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called **signal**. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file **signal.h** gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG DFL);
```

restores the default action of process termination. In all cases, **signal** returns the previous value of the signal. The second argument to **signal** may instead be the name of a

function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
   int onintr();

   if (signal(SIGINT, SIG IGN) != SIG IGN)
      signal(SIGINT, onintr);

   /* Process ... */

   exit(0);
}

onintr()
{
   unlink(tempfile);
   exit(1);
}
```

Why the test and the double call to **signal**? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the **onintr** routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that **signal** returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp buf  sjbuf;

main()
{
   int (*istat)(), onintr();

   istat = signal(SIGINT, SIG IGN); /* save original status */
   setjmp(sjbuf); /* save current stack position */
   if (istat != SIG IGN)
      signal(SIGINT, onintr);

   /* main processing loop */
}
```

```
onintr()
{
   printf("0nterrupt0);
   longjmp(sjbuf);    /* return to saved state */
}
```

The include file **setjmp.h** declares the type **jmp buf** an object in which the state can be saved. **sjbuf** is such an object; it is an array of some sort. The **setjmp** routine then saves the state of things. When an interrupt occurs, a call is forced to the **onintr** routine, which can print a message, set flags, or whatever. **longjmp** takes as argument an object stored into by **setjmp**, and restores control to the location after the call to **setjmp**, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling **exit** or **longjmp**, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, **wait**, and **pause**.) A program whose **onintr** program just sets **intflag**, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
   if (intflag)
      /* EOF caused by interrupt */
   else
      /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
   execl(...);
signal(SIGINT, SIG IGN);    /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to

decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function system:

```
#include <signal.h>

system(s)   /* run command string s */
char *s;
{
   int status, pid, w;
   register int (*istat)(), (*qstat)();

   if ((pid = fork()) == 0) {
      execl("/bin/sh", "sh", "-c", s, 0);
       exit(127);
   }
   istat = signal(SIGINT, SIG IGN);
   qstat = signal(SIGQUIT, SIG IGN);
   while ((w = wait(&status)) != pid && w != -1)
      ;
   if (w == -1)
      status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   return(status);
}
```

As an aside on declarations, the function **signal** obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values **SIG IGN** and **SIG DFL** have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define  SIG DFL  (int (*)())0
#define  SIG IGN  (int (*)())1
```

References

[1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF is actually −1, and is the value returned by the read routines on end-of-file or error.

NULL is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE expands to **struct iob** and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See **setbuf**, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names **stdin**, **stdout**, and **stderr** are in effect constants and may not be assigned to.

2. Calls

FILE *fopen(filename, type) char *filename, *type;
opens the file and, if needed, allocates a buffer for it. **filename** is a character string specifying the name. **type** is a character string (not a single character). It may be "r", "w", or "a" to indicate intent to read, write, or append. The value returned is a file pointer. If it is **NULL** the attempt to open failed.

FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;

The stream named by **ioptr** is closed, if necessary, and then reopened as if by **fopen**. If the attempt to open fails, NULL is returned, otherwise **ioptr**, which will now refer to the new file. Often the reopened stream is **stdin** or **stdout**.

int getc(ioptr) FILE *ioptr;
returns the next character from the stream named by **ioptr**, which is a pointer to a file such as returned by **fopen**, or the name **stdin**. The integer EOF is returned on end-of-file or when an error occurs. The null character \x0 is a legal character.

int fgetc(ioptr) FILE *ioptr;
acts like **getc** but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc(c, ioptr) FILE *ioptr;
putc writes the character **c** on the output stream named by **ioptr**, which is a value returned from **fopen** or perhaps **stdout** or **stderr**. The character is returned as value, but EOF is returned on error.

fputc(c, ioptr) FILE *ioptr;
acts like **putc** but is a genuine function, not a macro.

fclose(ioptr) FILE *ioptr;
The file corresponding to **ioptr** is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. **fclose** is automatic on normal termination of the program.

fflush(ioptr) FILE *ioptr;
Any buffered information on the (output) stream named by **ioptr** is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, **stderr** always starts off unbuffered and remains so unless **setbuf** is used, or unless it is reopened.

exit(errcode);
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls **fflush** for each output file. To terminate without flushing, use **exit**.

feof(ioptr) FILE *ioptr;
returns non-zero when end-of-file has occurred on the specified input stream.

ferror(ioptr) FILE *ioptr;
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar();
is identical to **getc(stdin)**.

putchar(c);
is identical to **putc(c, stdout)**.

char *fgets(s, n, ioptr) char *s; FILE *ioptr;
reads up to **n-1** characters from the stream **ioptr** into the character pointer **s**. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. **fgets** returns the first argument, or NULL if error or end-of-file occurred.

fputs(s, ioptr) char *s; FILE *ioptr;
writes the null-terminated string (character array) **s** on the stream **ioptr**. No newline is appended. No value is returned.

ungetc(c, ioptr) FILE *ioptr;
The argument character **c** is pushed back on the input stream named by **ioptr**. Only

one character may be pushed back.

printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;
printf writes on the standard output. **fprintf** writes on the named output stream. **sprintf** puts characters in the character array (string) named by **s**. The specifications are as described in section **printf**(3) of the *UNIX Programmer's Manual.*

scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
scanf reads from the standard input. **fscanf** reads from the named input stream. **sscanf** reads from the character string supplied as **s**. **scanf** reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string **format**, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, **EOF** is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
reads **nitems** of data beginning at **ptr** from file **ioptr**. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the **fopen** call.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
Like **fread**, but in the other direction.

rewind(ioptr) FILE *ioptr;
rewinds the stream named by **ioptr**. It is not very useful except on input, since a rewound output file is still open only for output.

system(string) char *string;
The **string** is executed by the shell as if typed at the terminal.

getw(ioptr) FILE *ioptr;
returns the next word from the input stream named by **ioptr**. **EOF** is returned on end-of-file or error, but since this a perfectly good integer **feof** and **ferror** should be used. A "word" is 16 bits on the PDP-11.

putw(w, ioptr) FILE *ioptr;
writes the integer **w** on the named output stream.

setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf may be used after a stream has been opened but before I/O has started. If **buf** is **NULL**, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char  buf[BUFSIZ];
```

fileno(ioptr) FILE *ioptr;
returns the integer file descriptor associated with the file.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
The location of the next byte in the stream named by **ioptr** is adjusted. **offset** is a long integer. If **ptrname** is 0, the offset is measured from the beginning of the file; if **ptrname** is 1, the offset is measured from the current read or write pointer; if

ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from ftell and the ptrname must be 0).

long ftell(ioptr) FILE *ioptr;
The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to fseek, so as to position the file to the same place it was when ftell was called.)

getpw(uid, buf) char *buf;
The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array buf, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

char *malloc(num);
allocates num bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available.

char *calloc(num, size);
allocates space for num items each of size size. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available .

cfree(ptr) char *ptr;
Space is returned to the pool used by calloc. Disorder can be expected if the pointer was not obtained from calloc.

The following are macros whose definitions may be obtained by including <ctype.h>.

isalpha(c) returns non-zero if the argument is alphabetic.

isupper(c) returns non-zero if the argument is upper-case alphabetic.

islower(c) returns non-zero if the argument is lower-case alphabetic.

isdigit(c) returns non-zero if the argument is a digit.

isspace(c) returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

ispunct(c) returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

isalnum(c) returns non-zero if the argument is a letter or a digit.

isprint(c) returns non-zero if the argument is printable — a letter, digit, or punctuation character.

iscntrl(c) returns non-zero if the argument is a control character.

isascii(c) returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

toupper(c) returns the upper-case character corresponding to the lower-case letter c.

tolower(c) returns the lower-case character corresponding to the upper-case letter c.

PART 2: LANGUAGES

This part includes articles on four of the languages and four of the language preprocessors available on ULTRIX-32:

- C
- FORTRAN 77
- RATFOR
- EFL
- Pascal
- Franz Lisp
- FP
- M4

These articles are authoritative reference materials appropriate for people familiar with programming in the languages described. Each article defines the implementation of a language or preprocessor on the ULTRIX-32 system. With the exception of the articles on Pascal, RATFOR, and M4, these articles are not tutorial, and they are not for beginners.

C Language

The first three articles deal with the C language. "The C Programming Language - Reference Manual" lists in detail the rules, conventions, and concepts that define the implementation of C on the VAX computer. This is reprinted from an appendix in *The C Programming Language* [1], by Kernighan and Ritchie. Before you use this article, you should know how to write programs in C and have read *The C Programming Language*.

The next two articles describe C language compilers. "A Tour Through the Portable C Compiler," by Johnson, explains the Berkeley C compiler available in the ULTRIX-32 system. It tells what happens when you compile a C program on ULTRIX-32 and is meant for people who may support the C compiler. This article gives an excellent overview of the organization, operation, and background of the ULTRIX-32 C compiler. The Ritchie article, "A Tour Through the UNIX C Compiler," describes the Bell UNIX C compiler, not implemented on ULTRIX-32.

FORTRAN

The two articles that follow describe f77 FORTRAN. The "Introduction to the f77 I/O Library," by Wasley, lists specifications and rules for using the f77 I/O library routines. These routines make use of the standard C I/O library routines in ULTRIX-32. The article explains

---

[1] Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, N.J., 1978.

the different methods available for accessing files, rules for use of logical units for I/O, and error and status handling for I/O processing. It tells in detail how the standard FORTRAN commands and concepts are implemented on the ULTRIX-32 system. In addition, the article identifies non-ANSI standard extensions to the library and shows methods you can use to make older FORTRAN programs compatible with this I/O library.

"A Portable FORTRAN 77 Compiler," by Feldman and Weinberger, describes the rules and conventions of FORTRAN 77 as implemented on the ULTRIX-32 system. Familiarity with FORTRAN 66 or another standard FORTRAN is prerequisite to comprehending this article.

RATFOR and EFL

The next two articles deal with FORTRAN preprocessors. *RATFOR* and *EFL* translate input files into FORTRAN source code. They overcome some of the cosmetic and control-flow defects of FORTRAN while retaining desirable FORTRAN features such as universality and efficiency. *RATFOR* and *EFL* programs are compatible with FORTRAN libraries, yet they offer a significant improvement over standard FORTRAN.

The article "RATFOR - A Preprocessor for a Rational FORTRAN," by Kernighan, tells how to write *RATFOR* code that is easier to read and write than FORTRAN code. The article also explains how to:

- Eliminate *goto* statements
- Group statements within a conditional construction
- Include the *else* clause as a part of a conditional construction
- Improve *do*, *while*, *for*, and *repeat until* functions

Readers will find this article easy to read and full of useful examples.

EFL is a descendant of *RATFOR*. *EFL* is more flexible; it allows more general forms for expressions and it provides a more uniform syntax. "The Programming Language EFL," by Feldman, lists concepts and rules and provides some programming examples.

Berkeley Pascal

The "Berkeley Pascal User's Manual" tells what you need to know to write and execute Pascal programs on the ULTRIX-32 system if you are already familiar with Pascal programming. The article is arranged in tutorial format; it lists reference materials, explains how to use an editor to create a Pascal program, and gives various execution options. Berkeley Pascal includes six utilities for translating, compiling, running, and analyzing programs:

pi	Translates the source program into object code and stores the object code
px	Interprets (executes) the object code created by pi
pix	Translates the source program and then executes it
pc	Processes the source program to compile an executable binary file
pxp	Creates an execution profile for a program when used together with pi or pix
pxref	Produces a program listing and a cross-reference identifier from a source program

"The Berkeley Pascal User's Manual" explains how to use these utilities, how to handle piping, input, and output, how to interpret error diagnostics, how to include source text from several files for the translator, and how to compile separate segments of a Pascal program to be linked for running later. An appendix gives a precise definition of Berkeley Pascal.

Franz Lisp

"The Franz Lisp Manual" gives a detailed and extensive description of the Berkeley dialect of Lisp. Franz Lisp is a sophisticated language that provides a complete environment in which you can develop and run programs. In addition, it offers:

- 14 data types
- Both a compiler and an interpreter
- Special functions (such as *apply*)
- System control functions (such as memory allocation)
- Macros and fclosures
- Compatibility with foreign subroutines
- Error handling capabilities
- Powerful debugging tools (trace, stepper, fixit)
- A CMU top-level package that serves as an alternative to the default Franz Lisp top-level package
- A file package that allows you to save functions for use in other sessions
- An editor specially designed for modifying Lisp programs

Because this long article is organized as a reference manual, you may find it useful to read the introductory section in each chapter to gain an overview, before reading the chapters in depth.

FP

FP is a preprocessor that produces Franz Lisp source code. The "Berkeley FP User's Manual" is appropriate reading for sophisticated programmers familiar with Lisp. The article describes, in terse terms, the principles and rules of the language. This description includes definitions of:

- Objects
- Operations
- Functions
- Input and output procedures
- Execution options

You may find the extensive programming examples helpful.

M4

M4 is a macro processor that provides string substitution. It accepts as input source code in any computer language and substitutes a defined text for each occurrence of a macro name. "The M4 Macro Processor," by Kernighan and Ritchie, offers readable explanations and good examples. You can use *M4* to:

- Set up your own macros
- Create and use macros that take several arguments
- Use a set of built-in macros
- Bring in new files with an *include* function
- Call shell functions with a *system* command

The C Programming Language — Reference Manual

Dennis M. Ritchie

Bell Laboratories, Murray Hill, New Jersey

This manual is reprinted, with minor changes, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
DEC VAX-11	8 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int       extern    else
char      register  for
float     typedef   do
double    static    while
struct    goto      switch
union     return    case
long      sizeof    default
short     break     entry
unsigned  continue
auto      if
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some

† UNIX is a Trademark of Bell Laboratories.

implementations also reserve the words `fortran` and `asm`.

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics which affect sizes are summarized in §2.6.

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit zero), decimal otherwise. The digits `8` and `9` have octal value 10 and 11 respectively. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote `'` and the backslash `\`, may be represented according to the following table of escape sequences:

newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
backslash	\	`\\`
single quote	'	`\'`
bit pattern	*ddd*	`\`*ddd*

The escape `\`*ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in `"..."`. A string has type "array of characters" and storage class `static` (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte `\0` at the end of each string so that programs which scan the string can find its end. In a string, the double quote character `"` must be preceded by a `\`; in addition, the same escapes as described for character constants may be used. Finally, a `\` and an immediately following newline are ignored.

2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 bits	9 bits	8 bits	8 bits
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

The VAX-11 is identical to the PDP-11 except that integers have 32 bits.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

4. What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (`char`) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (`float`) and double-precision floating point (`double`) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char` and `int` of all sizes will collectively be called *integral* types. `float` and `double` will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;
functions which return objects of a given type;
pointers to objects of a given type;
structures containing a sequence of objects of various types;
unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage: an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from −128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value −1.

When a longer integer is converted to a shorter or to a `char`, it is truncated on the left; excess bits are simply discarded.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo 2^{wordsize}). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

First, any operands of type `char` or `short` are converted to `int`, and any of type `float` are converted to `double`.

Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.
Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.
Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.
Otherwise, both operands must be `int`, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. All existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(*expression*)
primary-expression [*expression*]
primary-expression (*expression-list*$_{opt}$)
primary-lvalue . *identifier*
primary-expression -> *identifier*

expression-list:
expression
expression-list , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int`; floating constants are `double`.

A string is a primary expression. Its type is originally "array of `char`"; but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a - and a >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
- `*` *expression*
- `&` *lvalue*
- `-` *expression*
- `!` *expression*
- `~` *expression*
- `++` *lvalue*
- `--` *lvalue*
- *lvalue* `++`
- *lvalue* `--`
- `(` *type-name* `)` *expression*
- `sizeof` *expression*
- `sizeof (` *type-name* `)`

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary `-` operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in an `int`. There is no unary `+` operator.

The result of the logical negation operator `!` is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The `~` operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix `++` is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x+=1`. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

The `sizeof` operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be `float`.

7.4 Additive operators

The additive operators + and - group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression - *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same

array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

shift-expression:
expression << *expression*
expression >> *expression*

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; vacated bits are 0-filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0-fill) if `E1` is `unsigned`; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; `a<b<c` does not mean what it seems to.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise AND operator

and-expression:
expression & *expression*

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9 Bitwise exclusive OR operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10 Bitwise inclusive OR operator

inclusive-or-expression:
expression | *expression*

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11 Logical AND operator

logical-and-expression:
expression && *expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.12 Logical OR operator

logical-or-expression:
expression || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.13 Conditional operator

conditional-expression:
expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = *expression*
lvalue += *expression*
lvalue -= *expression*
lvalue *= *expression*
lvalue /= *expression*
lvalue %= *expression*
lvalue >>= *expression*
lvalue <<= *expression*
lvalue &= *expression*
lvalue ^= *expression*
lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left

preparatory to the assignment.

The behavior of an expression of the form `E1` *op*`=` `E2` may be inferred by taking it as equivalent to `E1 = E1` *op* `(E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
`auto`
`static`
`extern`
`register`
`typedef`

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int`, `char`, or pointer. One other restriction applies to register variables: the address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

8.2 Type specifiers

The type-specifiers are

type-specifier:
`char`
`short`
`int`
`long`
`unsigned`
`float`
`double`
struct-or-union-specifier
typedef-name

The words `long`, `short`, and `unsigned` may be thought of as adjectives; the following combinations are acceptable.

```
short int
long int
unsigned int
long float
```

The meaning of the last is the same as `double`. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures and unions are discussed in §8.5; declarations with `typedef` names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , *declarator-list*

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(*declarator*)
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where T is a type-specifier (like int, etc.) and D1 is a declarator. Suppose this declaration makes the identifier have type "... T," where the "..." is empty if D1 is just a plain identifier (so that the type of x in "int x" is just int). Then if D1 has the form

```
*D
```

the type of the contained identifier is "... pointer to T."

If D1 has the form

```
D()
```

then the contained identifier has the type "... function returning T."

If D1 has the form

```
D[constant-expression]
```

or

```
D[]
```

then the contained identifier has type "... array of T." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures, unions or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer i, a pointer ip to an integer, a function f returning an integer, a function fip returning a pointer to an integer, and a pointer pfi to a function which returns an integer. It is especially useful to compare the last two. The binding of *fip() is *(fip()), so that the declaration suggests, and the same construction in an expression requires, the calling of a function fip, and then using indirection through the (pointer) result to yield an integer. In the declarator (*pfi)(), the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, x3d is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions x3d, x3d[i], x3d[i][j], x3d[i][j][k] may reasonably appear in an expression. The first three have type "array," the last has type int.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { *struct-decl-list* }
struct-or-union identifier { *struct-decl-list* }
struct-or-union identifier

struct-or-union:
`struct`
`union`

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , *struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

struct-declarator:
declarator
declarator : *constant-expression*
: *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on the PDP-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

`struct` *identifier* { *struct-decl-list* }
`union` *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

`struct` *identifier*
`union` *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by `=`, and consists of an expression or a list of values nested in braces.

initializer:
- `=` *expression*
- `=` `{` *initializer-list* `}`
- `=` `{` *initializer-list* `,` `}`

initializer-list:
- *expression*
- *initializer-list* `,` *initializer-list*
- `{` *initializer-list* `}`

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
     { 1, 3, 5 },
     { 2, 4, 6 },
     { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = {
     1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but that for `y[0]` does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] = {
     { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of `sizeof`) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (*abstract-declarator*)
    * *abstract-declarator*
    abstract-declarator ( )
    abstract-declarator [$constant\text{-}expression_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of 3 integers," "function returning pointer to integer," and "pointer to function returning an integer."

8.8 Typedef

Declarations whose "storage class" is `typedef` do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
identifier

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein become syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is "pointer to `int`," and that of `z` is the specified structure. `zp` is a pointer to such a structure.

`typedef` does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

9.4 While statement

The `while` statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The `do` statement has the form

```
do statement while ( expression ) ;
```

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The `for` statement has the form

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement
```

This statement is equivalent to

```
expression-1 ;
while (expression-2) {
        statement
        expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch ( expression ) statement
```

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

```
case constant-expression :
```

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`case` and `default` prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see `break`, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {        do {                 for (...) {
  ...                  ...                  ...
contin: ;            contin: ;            contin: ;
}                    } while (...);       }
```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function.

9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as `while`.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

The function-body has the form

function-body:
declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; `{ ... }` is the block giving the code for the statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

10.2 External data definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be `extern` (which is the default) or `static`, but not `auto` or `register`.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
        auto int distance;
        ...
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance`†.

11.2 Scope of externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the `extern` keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

---

†It is agreed that the ice is thin here.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

```
#define identifier token-string
```

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

```
#define identifier( identifier , ... , identifier ) token-string
```

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

```
#undef identifier
```

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

```
#include "filename"
```

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

```
#include <filename>
```

searches only the standard places, and not the directory of the source file.

`#include`'s may be nested.

12.3 Conditional compilation

A compiler control line of the form

```
#if constant-expression
```

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

```
#ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. A control line of the form

```
#ifndef identifier
```

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
#else
```

and then by a control line

```
#endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and an `#else` or, lacking an `#else`, the `#endif`, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

```
#line constant identifier
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning ...", it is implicitly declared to be `extern`.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning `int`".

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the . operator); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or –>) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before ., and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a –> is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of `g` might read

```
g(funcp)
int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that `f` must be declared explicitly in the calling routine since its appearance in `g(f)` was not followed by `(`.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules which apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If `E` is an n-dimensional array of rank $i \times j \times \cdots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `i` is converted to the type of `x`, which involves multiplying `i` by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

`alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is measured in bytes. `char`s have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word just to their right. Thus `char` pointers are measured in units of 2^{16} bytes; everything else is measured in units of 2^{18} machine words. `double` quantities and aggregates containing them must lie on an even word address (0 mod 2^{19}).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to `short` must be 0 mod 2, to `int` and `float` 0 mod 4, and to `double` 0 mod 8. Aggregates are aligned on the strictest boundary required by any of their constituents.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  <  >  <=  >=
```

or by the unary operators

```
-  ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Portability considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11, and VAX-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and VAX-11 and left-to-right on other machines. These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit-fields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

17. Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form *=op* instead of *op=* for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which actually decrements `x` since the `=` and the `-` are adjacent, but which might easily be intended to assign `-1` to `x`.

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
int  x  = 1;
```

one used

```
int  x  1;
```

The change was made because the initialization

```
int  f  (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

18. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

18.1 Expressions

The basic expressions are:

expression:
primary
* *expression*
& *expression*
\- *expression*
! *expression*
~ *expression*
++ *lvalue*
-- *lvalue*
lvalue ++
lvalue --
`sizeof` *expression*
(*type-name*) *expression*
expression binop expression
expression ? *expression* : *expression*
lvalue asgnop expression
expression , *expression*

primary:
identifier
constant
string
(*expression*)
primary (*expression-list*$_{opt}$)
primary [*expression*]
lvalue . *identifier*
primary -> *identifier*

lvalue:
identifier
primary [*expression*]
lvalue . *identifier*
primary -> *identifier*
* *expression*
(*lvalue*)

The primary-expression operators

```
()   []   .   ->
```

have highest priority and group left-to-right. The unary operators

```
*   &   -   !   ~   ++   --   sizeof   ( type-name )
```

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```
*     /     %
+     -
>>    <<
<     >     <=     >=
==    !=
&
^
|
&&
||
?:
```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
=   +=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

The comma operator has the lowest priority, and groups left-to-right.

18.2 Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
`auto`
`static`
`extern`
`register`
`typedef`

type-specifier:
`char`
`short`
`int`
`long`
`unsigned`
`float`
`double`
struct-or-union-specifier
typedef-name

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(*declarator*)
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]

struct-or-union-specifier:
`struct` { *struct-decl-list* }
`struct` *identifier* { *struct-decl-list* }
`struct` *identifier*
`union` { *struct-decl-list* }
`union` *identifier* { *struct-decl-list* }
`union` *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , *struct-declarator-list*

struct-declarator:
declarator
declarator : *constant-expression*
: *constant-expression*

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(*abstract-declarator*)
* *abstract-declarator*
abstract-declarator ()
abstract-declarator [$constant\text{-}expression_{opt}$]

typedef-name:
identifier

18.3 Statements

compound-statement:
{ $declaration\text{-}list_{opt}$ $statement\text{-}list_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*
`while` (*expression*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*
`switch` (*expression*) *statement*
`case` *constant-expression* : *statement*
`default` : *statement*
`break` ;
`continue` ;
`return` ;
`return` *expression* ;
`goto` *identifier* ;
identifier : *statement*
;

18.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
{ *declaration-list*$_{opt}$ *statement-list* }

data-definition:
`extern`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;
`static`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

18.5 Preprocessor

```
#define identifier token-string
#define identifier( identifier , ... , identifier ) token-string
#undef identifier
#include "filename"
#include <filename>
#if constant-expression
#ifdef identifier
#ifndef identifier
#else
#endif
#line constant identifier
```

Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978).

1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

enum-specifier

with syntax

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, winedark };
...
enum color *cp, col;
```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were `int`.

A Tour Through the Portable C Compiler

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.[1]

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.[2]

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.[3] This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these

machines, perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the other hand, a certain amount of the code requres hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.[4] One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.[5]

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the **#define** and **#include** statements, and related features (e.g., **#ifdef**, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined

by the node number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

The Source Files

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*[6] to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

Data Structure Considerations.

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, −=), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *optype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number (=, +=, etc.), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

ARY + (PTR<<2) + (FTN<<4) + DOUBLE

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If *t* is a type word, *BTYPE(t)* gives the basic type. *ISPTR(t)*, *ISARY(t)*, and *ISFTN(t)* ask if an object of this type is a pointer, array, or a function, respectively. *MODTYPE(t,b)* sets the basic type of *t* to *b*. *DECREF(t)* gives the type resulting from removing the first operator from *t*. Thus, if *t* is a pointer to *t'*, a function returning *t'*, or an array of *t'*, then *DECREF(t)* would equal *t'*. *INCREF(t)* gives the type representing a pointer to *t*. Finally, there are operators for dealing with the unsigned types. *ISUNSIGNED(t)* returns true if *t* is one of the four basic unsigned types; in this case, *DEUNSIGN(t)* gives the associated 'signed' type. Similarly, *UNSIGNABLE(t)* returns true if *t* is one of the four basic types that could become unsigned, and *ENUNSIGN(t)* returns the unsigned analogue of *t* in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in *mfile1* and *mfile2*. They are not the same in the two passes; the first pass nodes contain dimension and size information, while the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called *op*, containing the node number, and *type*, containing the type word. A function called *talloc()* returns a pointer to a new tree node. To free a node, its *op* field need merely be set to FREE. The

other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, *left* and *right*, that contain pointers to the left and right descendants. Unary operator nodes have the *left* field, and a value field called *rval*. Leaf nodes, with no descendants, have two value fields: *lval* and *rval*.

At appropriate times, the function *tcheck()* can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function *tcopy(p)* takes a pointer *p* that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function *walkf(p,f)* performs a postorder walk of the tree pointed to by *p*, and applies the function *f* to each node. The function *fwalk(p,f,d)* does a preorder walk of the tree pointed to by *p*. At each node, it calls a function *f*, passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value *d* is the value passed down to the root. *Fwalk* is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

Pass One

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

Lexical Analysis

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.
- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

Parsing

As mentioned above, the parser is generated by Yacc from the grammar on file *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a **sizeof** construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the

recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.c* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values "on the side", independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In this way, when it is time to declare the name, the name and the type information can be quickly brought together. The "$0" mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a **break** or **continue** statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enum declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

Storage Classes

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this transformation is machine dependent, and strongly depends on the storage class.

The classes include EXTERN (for externally declared, but not defined variables), EXTDEF (for external definitions), and similar distinctions for USTATIC and STATIC, UFORTRAN and FORTRAN (for fortran functions) and ULABEL and LABEL. The storage classes REGISTER and AUTO are obvious, as are STNAME, UNAME, and ENAME (for structure, union, and enumeration tags), and the associated MOS, MOU, and MOE (for the members). TYPEDEF is treated as a storage class as well. There are two special storage classes: PARAM and SNULL. SNULL is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, PARAM is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field; a k bit bit field has storage class k plus FIELD. This enables the size to be quickly recovered from the storage class.

Symbol Table Maintenance.

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block; in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if an formal parameter to a function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each *, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enums) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the symbol table index in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to auto.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the

current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry "hidden", and making a new entry, marked "hiding". *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are "unhidden".

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a **goto** into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a "general purpose" field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in dimtab of the list of members. This list contains the symbol table indices for the structure members, terminated by a −1.

Tree Building

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here; many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to

handle issues which are unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *stref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary * and &, structure assignment, and calls. In the case of unary * and &, *buildtree* will cancel a * applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is still a rather strange area of the compiler (and of C!). The recent introduction of type casts has only confounded this situation. Most of the conversion operators are generated by calls to *tymatch* and *ptmatch*, both of which are given a tree, and asked to make the operands agree in type. *Ptmatch* treats the case where one of the operands is a pointer; *tymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makety*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by *buildtree* is given to a machine dependent routine, *clocal*, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to *clocal* as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to *buildtree* may get confused about correct type of the subtrees; thus *clocal* is given the conversion ops only after the entire tree is built. This topic will be dealt with in more detail later.

Initialization

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it is may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure; one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a } has been seen for the current identifier.)

When an initialization begins, the routine *beginit* is called; it handles the alignment restrictions, if any, and calls *instk* to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine *doinit* is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. *gotscal* is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; *endinit* cleans up. If a { or } is encountered in the string of initializers, it is handled by calling *ilbrace* or *irbrace*, respectively.

A central issue is the treatment of the "holes" that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, *inoff*, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine *inforce* which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines *fincode* (for floating point initialization), *incode* (for fields, and other initializations less than an int in size), and *cinit* (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token STRING, but does not do anything with the contents. The parser calls *getstr*, which sets up the appropriate location counters and flags, and calls *lxstr* to read and process the contents of the string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

Statements

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot "fall through" the last statement.

Conditional branches are handled by generating an expression node, CBRANCH, whose left descendant is the conditional expression and the right descendant is an ICON node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, FORCE, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual integer return value register.

Optimization

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional; it must be called for proper operation of the compiler.

Optim is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of * have been eliminated by *buildtree.)* The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true; on some machine, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are currently quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$(x + a) + b$$

where a and b are constants, and attempts to combine a and b at compile time. A number of special cases are also examined; additions of 0 and multiplications by 1 are removed, although the correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion op to keep the types from becoming fouled up. Finally, in cases where a relational operation is being done, and one operand is a constant, the operands are permuted, and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

There are dozens of similar optimizations that can be, and should be, done. It seems likely that this routine will be expanded in the relatively near future.

Machine Dependent Stuff

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcode* and *efcode*.

Clocal has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree*. There are two major areas where this is important; NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer; the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary * node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number; a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different,

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal*, in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte pointer to short or long pointer might require a shift or rotate operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

The other major machine dependent issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.[5] The routine *bfcode* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfcode* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfcode* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfcode* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efcode* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller know and declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called functionis allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is non-reentrant. However, the function can now be called as a subroutine

without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

First Pass Summary

There are may other issues which have been ignored here, partly to justify the title "tour", and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler's first pass the argument

```
-X[flags]
```

Some of the more interesting flags are -Xd for the defining and freeing of symbols, -Xi for initialization comments, and -Xb for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, -Xddd gives more information than -Xd. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

Overview.

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are

free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations, — most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

The Machine Model

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREG|STAREG for scratch AREGS's, and SBREG and SBREG|STBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *istreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

General Organization

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a ')', and making note of the information about stack usage and register allocation contained on lines beginning with ']' and '['. The expression trees, whose beginning is indicated by a line beginning with '.', are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree; it attempts to delay some postfix ++ and −− computations that might reasonably be done after the smoke clears. It also attempts to handle comma (,) operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only; don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG (compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition

codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

Codgen first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *notoff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.[7] It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.[8] These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*; for example, when processing the operators &&, ||, and comma (','), that have a left to right evaluation, it is incorrect for *store* examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the ? : operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to find a match with a different cookie; for example, in order to compute an expression with cookie INTEMP (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a + is a scratch register, the + can be replaced by a += operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

a += *b*

where *a* and *b* are static variables.

To begin with, the whole expression tree is examined with cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the += operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

a = *a* + *b*

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The machine dependent rule for + notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf *a*, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of *a* is rewritten in place to represent the register into which *a* is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

reg + *b*

to consider. Once again, there is no match, but the default rewriting rule rewrites the + as a += operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

reg += *b*

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

a = **reg**

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different; *b* is loaded in to a register, and then an add to storage instruction generated to add this register in to *a*. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

> **If** we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), **then** we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```
ASG PLUS,   INAREG,
            SAREG,      TINT,
            SNAME,      TINT,
                        0,          RLEFT,
                        "           add         AL,AR\n",
```

The top line specifies the operator (+=) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim*. to the output; upper case letters trigger various macro-like expansions. Thus, **AL** would expand into the **A**ddress form of the **L**eft operand — presumably the register number. Similarly, **AR** would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```
ASG OPSIMP, INAREG|FORCC,
            SAREG,      TINT|TUNSIGNED|TPOINT,
            SAREG|SNAME|SOREG|SCON,          TINT|TUNSIGNED|TPOINT,
                        0,          RLEFT|RESCC,
                        "           OI          AL,AR\n",
```

Here, OPSIMP represents the operators +, −, |, &, and ^. The **OI** macro in the output string expands into the appropriate **I**nteger **O**pcode for the operator. The left and right sides can be

integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

The Template Matching Algorithm.

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the op of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the "left" descendant. If a node represents a unary operator or a leaf node (no descendants) the "right" descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees; STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator: these patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries: this may be of some importance on word–addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and −1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

Register Allocation.

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered; the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.[9] The second issue is dealt with rather more successfully; a machine dependent function called *szty(t)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type *t*. If *szty* returns 2, an even/odd pair of A registers is allocated for each request.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to an integer in it, we may load the integer into register 2 itself by saying:

```
L    2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR   3,3
IC   3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye; it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine

usable to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible to have "double indexing" forms of addressing; these are represented by OREGS's with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results; the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error "cannot reclaim" is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

The Machine Dependent Interface

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi–Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here; that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

The Rewriting Rules

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nextcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the

tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds; for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call *order* or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment ops, when rewritten, call the routine *setasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

a += b

to be rewritten as

a = a + b

This is a useful default for certain mixtures of strange types (for example, when *a* is a bit field and *b* an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

x ++

by

((x += 1) − 1)

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of x is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

expr + const

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where A and B are expressions; suppose further that B takes two registers, and A one. It is possible to compute the full expression in two registers by first computing B, and then, using the scratch register used by B, but not containing the answer, compute A. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine

dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

(p + i)

where *p* is a pointer and *i* an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both *p* and *i* into registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

Register Allocation

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ..." , or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the

compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are −e, which prints the expression tree, −r, which gives information about the allocation of registers, −a, which gives information about the performance of *rallo*, and −o, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3
3. A. Snyder, *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, Mass., 1974.
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104, January 1978.
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence*, 1977.
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.
7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.*, vol. 23, no. 3, pp. 488-501, 1975. Also in *Proc. ACM Symp. on Theory of Computing,* pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.*, vol. 17, no. 4, pp. 715-728, October 1970. Reprinted as pp. 229-247 in *Compiler Techniques*, ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages*, pp. 21-28, January 1977.

A Tour Through the UNIX† C Compiler

D. M. Ritchie

Bell Laboratories,
Murray Hill, New Jersey 07974

The Intermediate Language

Communication between the two phases of the compiler proper is carried out by means of a pair of intermediate files. These files are treated as having identical structure, although the second file contains only the code generated for strings. It is convenient to write strings out separately to reduce the need for multiple location counters in a later assembly phase.

The intermediate language is not machine-independent; its structure in a number of ways reflects the fact that C was originally a one-pass compiler chopped in two to reduce the maximum memory requirement. In fact, only the latest version of the compiler has a complete intermediate language at all. Until recently, the first phase of the compiler generated assembly code for those constructions it could deal with, and passed expression parse trees, in absolute binary form, to the second phase for code generation. Now, at least, all inter-phase information is passed in a describable form, and there are no absolute pointers involved, so the coupling between the phases is not so strong.

The areas in which the machine (and system) dependencies are most noticeable are

1. Storage allocation for automatic variables and arguments has already been performed, and nodes for such variables refer to them by offset from a display pointer. Type conversion (for example, from integer to pointer) has already occurred using the assumption of byte addressing and 2-byte words.
2. Data representations suitable to the PDP-11 are assumed; in particular, floating point constants are passed as four words in the machine representation.

As it happens, each intermediate file is represented as a sequence of binary numbers without any explicit demarcations. It consists of a sequence of conceptual lines, each headed by an operator, and possibly containing various operands. The operators are small numbers; to assist in recognizing failure in synchronization, the high-order byte of each operator word is always the octal number 376. Operands are either 16-bit binary numbers or strings of characters representing names. Each name is terminated by a null character. There is no alignment requirement for numerical operands and so there is no padding after a name string.

The binary representation was chosen to avoid the necessity of converting to and from character form and to minimize the size of the files. It would be very easy to make each operator-operand 'line' in the file be a genuine, printable line, with the numbers in octal or decimal; this in fact was the representation originally used.

The operators fall naturally into two classes: those which represent part of an expression, and all others. Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the first phase. Expressions are passed as a whole, with no non-expression operators intervening. The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack; each unary

†UNIX is a Trademark of Bell Laboratories.

operator replaces the top of the stack by a node whose operand is the old top-of-stack; each binary operator replaces the top pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack. Following each expression is a special operator which passes the unique previous expression to the 'optimizer' described below and then to the code generator.

Here is the list of operators not themselves part of expressions.

EOF

marks the end of an input file.

BDATA *flag data ...*

specifies a sequence of bytes to be assembled as static data. It is followed by pairs of words; the first member of the pair is non-zero to indicate that the data continue; a zero flag is not followed by data and terminates the operator. The data bytes occupy the low-order part of a word.

WDATA *flag data ...*

specifies a sequence of words to be assembled as static data; it is identical to the BDATA operator except that entire words, not just bytes, are passed.

PROG

means that subsequent information is to be compiled as program text.

DATA

means that subsequent information is to be compiled as static data.

BSS

means that subsequent information is to be compiled as unitialized static data.

SYMDEF *name*

means that the symbol *name* is an external name defined in the current program. It is produced for each external data or function definition.

CSPACE *name size*

indicates that the name refers to a data area whose size is the specified number of bytes. It is produced for external data definitions without explicit initialization.

SSPACE *size*

indicates that *size* bytes should be set aside for data storage. It is used to pad out short initializations of external data and to reserve space for static (internal) data. It will be preceded by an appropriate label.

EVEN

is produced after each external data definition whose size is not an integral number of words. It is not produced after strings except when they initialize a character array.

NLABEL *name*

is produced just before a BDATA or WDATA initializing external data, and serves as a label for the data.

RLABEL *name*

is produced just before each function definition, and labels its entry point.

SNAME *name number*

is produced at the start of each function for each static variable or label declared therein. Subsequent uses of the variable will be in terms of the given number. The code generator uses this only to produce a debugging symbol table.

ANAME *name number*

Likewise, each automatic variable's name and stack offset is specified by this operator. Arguments count as automatics.

RNAME *name number*

Each register variable is similarly named, with its register number.

SAVE *number*

produces a register-save sequence at the start of each function, just after its label (RLABEL).

SETREG *number*

is used to indicate the number of registers used for register variables. It actually gives the register number of the lowest free register; it is redundant because the RNAME operators could be counted instead.

PROFIL

is produced before the save sequence for functions when the profile option is turned on. It produces code to count the number of times the function is called.

SWIT *deflab line label value ...*

is produced for switches. When control flows into it, the value being switched on is in the register forced by RFORCE (below). The switch statement occurred on the indicated line of the source, and the label number of the default location is *deflab*. Then the operator is followed by a sequence of label-number and value pairs; the list is terminated by a 0 label.

LABEL *number*

generates an internal label. It is referred to elsewhere using the given number.

BRANCH *number*

indicates an unconditional transfer to the internal label number given.

RETRN

produces the return sequence for a function. It occurs only once, at the end of each function.

EXPR *line*

causes the expression just preceding to be compiled. The argument is the line number in the source where the expression occurred.

NAME *class type name*

NAME *class type number*

indicates a name occurring in an expression. The first form is used when the name is external; the second when the name is automatic, static, or a register. Then the number indicates the stack offset, the label number, or the register number as appropriate. Class and type encoding is described elsewhere.

CON *type value*

transmits an integer constant. This and the next two operators occur as part of expressions.

FCON *type 4-word-value*

transmits a floating constant as four words in PDP-11 notation.

SFCON *type value*

transmits a floating-point constant whose value is correctly represented by its high-order word in PDP-11 notation.

NULL

indicates a null argument list of a function call in an expression; call is a binary operator whose second operand is the argument list.

CBRANCH *label cond*

produces a conditional branch. It is an expression operator, and will be followed by an EXPR. The branch to the label number takes place if the expression's truth value is the same as that of *cond*. That is, if *cond=1* and the expression evaluates to true, the branch is taken.

binary-operator *type*

There are binary operators corresponding to each such source-language operator; the type of the result of each is passed as well. Some perhaps-unexpected ones are: COMMA, which is a right-associative operator designed to simplify right-to-left evaluation of function arguments; prefix and postfix ++ and −−, whose second operand is the increment amount, as a CON; QUEST and COLON, to express the conditional expression as 'a?(b:c)'; and a sequence of special operators for expressing relations between pointers, in case pointer comparison is different from integer comparison (e.g. unsigned).

unary-operator *type*

There are also numerous unary operators. These include ITOF, FTOI, FTOL, LTOF, ITOL, LTOI which convert among floating, long, and integer; JUMP which branches indirectly through a label expression; INIT, which compiles the value of a constant expression used as an initializer; RFORCE, which is used before a return sequence or a switch to place a value in an agreed-upon register.

Expression Optimization

Each expression tree, as it is read in, is subjected to a fairly comprehensive analysis. This is performed by the *optim* routine and a number of subroutines; the major things done are

1. Modifications and simplifications of the tree so its value may be computed more efficiently and conveniently by the code generator.

2. Marking each interior node with an estimate of the number of registers required to evaluate it. This register count is needed to guide the code generation algorithm.

One thing that is definitely not done is discovery or exploitation of common subexpressions, nor is this done anywhere in the compiler.

The basic organization is simple: a depth-first scan of the tree. *Optim* does nothing for leaf nodes (except for automatics; see below), and calls *unoptim* to handle unary operators. For binary operators, it calls itself to process the operands, then treats each operator separately. One important case is commutative and associative operators, which are handled by *acommute.*

Here is a brief catalog of the transformations carried out by by *optim* itself. It is not intended to be complete. Some of the transformations are machine-dependent, although they may well be useful on machines other than the PDP-11.

1. As indicated in the discussion of *unoptim* below, the optimizer can create a node type corresponding to the location addressed by a register plus a constant offset. Since this is precisely the implementation of automatic variables and arguments, where the register is fixed by convention, such variables are changed to the new form to simplify later processing.
2. Associative and commutative operators are processed by the special routine *acommute.*
3. After processing by *acommute,* the bitwise & operator is turned into a new *andn* operator; 'a & b' becomes 'a *andn* ~b'. This is done because the PDP-11 provides no *and* operator, but only *andn.* A similar transformation takes place for '=&'.
4. Relationals are turned around so the more complicated expression is on the left. (So that '2 > f(x)' becomes 'f(x) < 2'). This improves code generation since the algorithm prefers to have the right operand require fewer registers than the left.
5. An expression minus a constant is turned into the expression plus the negative constant, and the *acommute* routine is called to take advantage of the properties of addition.
6. Operators with constant operands are evaluated.
7. Right shifts (unless by 1) are turned into left shifts with a negated right operand, since the PDP-11 lacks a general right-shift operator.
8. A number of special cases are simplified, such as division or multiplication by 1, and shifts by 0.

The *unoptim* routine performs the same sort of processing for unary operators.

1. '*&x' and '&*x' are simplified to 'x'.
2. If *r* is a register and *c* is a constant or the address of a static or external variable, the expressions '*(r+c)' and '*r' are turned into a special kind of name node which expresses the name itself and the offset. This simplifies subsequent processing because such constructions can appear as the the address of a PDP-11 instruction.
3. When the unary '&' operator is applied to a name node of the special kind just discussed, it is reworked to make the addition explicit again; this is done because the PDP-11 has no 'load address' instruction.
4. Constructions like '*r++' and '*--r' where *r* is a register are discovered and marked as being implementable using the PDP-11 auto-increment and -decrement modes.
5. If '!' is applied to a relational, the '!' is discarded and the sense of the relational is reversed.
6. Special cases involving reflexive use of negation and complementation are discovered.
7. Operations applying to constants are evaluated.

The *acommute* routine, called for associative and commutative operators, discovers clusters of the same operator at the top levels of the current tree, and arranges them in a list: for 'a+((b+c)+(d+f))' the list would be'a,b,c,d,e,f'. After each subtree is optimized, the list is

sorted in decreasing difficulty of computation; as mentioned above, the code generation algorithm works best when left operands are the difficult ones. The 'degree of difficulty' computed is actually finer than the mere number of registers required; a constant is considered simpler than the address of a static or external, which is simpler than reference to a variable. This makes it easy to fold all the constants together, and also to merge together the sum of a constant and the address of a static or external (since in such nodes there is space for an 'offset' value). There are also special cases, like multiplication by 1 and addition of 0.

A special routine is invoked to handle sums of products. *Distrib* is based on the fact that it is better to compute 'c1*c2*x + c1*y' as 'c1*(c2*x + y)' and makes the divisibility tests required to assure the correctness of the transformation. This transformation is rarely possible with code directly written by the user, but it invariably occurs as a result of the implementation of multi-dimensional arrays.

Finally, *acommute* reconstructs a tree from the list of expressions which result.

Code Generation

The grand plan for code-generation is independent of any particular machine; it depends largely on a set of tables. But this fact does not necessarily make it very easy to modify the compiler to produce code for other machines, both because there is a good deal of machine-dependent structure in the tables, and because in any event such tables are non-trivial to prepare.

The arguments to the basic code generation routine *rcexpr* are a pointer to a tree representing an expression, the name of a code-generation table, and the number of a register in which the value of the expression should be placed. *Rcexpr* returns the number of the register in which the value actually ended up; its caller may need to produce a *mov* instruction if the value really needs to be in the given register. There are four code generation tables.

Regtab is the basic one, which actually does the job described above: namely, compile code which places the value represented by the expression tree in a register.

Cctab is used when the value of the expression is not actually needed, but instead the value of the condition codes resulting from evaluation of the expression. This table is used, for example, to evaluate the expression after *if*. It is clearly silly to calculate the value (0 or 1) of the expression 'a==b' in the context 'if (a==b) ... '

The *sptab* table is used when the value of an expression is to be pushed on the stack, for example when it is an actual argument. For example in the function call 'f(a)' it is a bad idea to load *a* into a register which is then pushed on the stack, when there is a single instruction which does the job.

The *efftab* table is used when an expression is to be evaluated for its side effects, not its value. This occurs mostly for expressions which are statements, which have no value. Thus the code for the statement 'a = b' need produce only the approoriate *mov* instruction, and need not leave the value of *b* in a register, while in the expression 'a + (b = c)' the value of 'b = c' will appear in a register.

All of the tables besides *regtab* are rather small, and handle only a relatively few special cases. If one of these subsidiary tables does not contain an entry applicable to the given expression tree, *rcexpr* uses *regtab* to put the value of the expression into a register and then fixes things up; nothing need be done when the table was *efftab,* but a *tst* instruction is produced when the table called for was *cctab,* and a *mov* instruction, pushing the register on the stack, when the table was *sptab.*

The *rcexpr* routine itself picks off some special cases, then calls *cexpr* to do the real work. *Cexpr* tries to find an entry applicable to the given tree in the given table, and returns −1 if no such entry is found, letting *rcexpr* try again with a different table. A successful match yields a string containing both literal characters which are written out and pseudo-operations, or macros, which are expanded. Before studying the contents of these strings we

will consider how table entries are matched against trees.

Recall that most non-leaf nodes in an expression tree contain the name of the operator, the type of the value represented, and pointers to the subtrees (operands). They also contain an estimate of the number of registers required to evaluate the expression, placed there by the expression-optimizer routines. The register counts are used to guide the code generation process, which is based on the Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator. A subtable consists of a sequence of entries, each with a key describing certain properties of the operands of the operator involved; associated with the key is a code string. Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node. A successful match returns the code string; an unsuccessful search, either for the operator in the main table or a compatble key in the subtable, returns a failure indication.

The tables are all contained in a file which must be processed to obtain an assembly language program. Thus they are written in a special-purpose language. To provided definiteness to the following discussion, here is an example of a subtable entry.

```
%n,aw
    F
    add  A2,R
```

The '%' indicates the key; the information following (up to a blank line) specifies the code string. Very briefly, this entry is in the subtable for '+' of *regtab;* the key specifies that the left operand is any integer, character, or pointer expression, and the right operand is any word quantity which is directly addressible (e.g. a variable or constant). The code string calls for the generation of the code to compile the left (first) operand into the current register ('F') and then to produce an 'add' instruction which adds the second operand ('A2') to the register ('R'). All of the notation will be explained below.

Only three features of the operands are used in deciding whether a match has occurred. They are:

1. Is the type of the operand compatible with that demanded?
2. Is the 'degree of difficulty' (in a sense described below) compatible?
3. The table may demand that the operand have a '*' (indirection operator) as its highest operator.

As suggested above, the key for a subtable entry is indicated by a '%,' and a comma-separated pair of specifications for the operands. (The second specification is ignored for unary operators). A specification indicates a type requirement by including one of the following letters. If no type letter is present, any integer, character, or pointer operand will satisfy the requirement (not float, double, or long).

b A byte (character) operand is required.

w A word (integer or pointer) operand is required.

f A float or double operand is required.

d A double operand is required.

l A long (32-bit integer) operand is required.

Before discussing the 'degree of difficulty' specification, the algorithm has to be explained more completely. *Rcexpr* (and *cexpr)* are called with a register number in which to place their result. Registers 0, 1, ... are used during evaluation of expressions; the maximum register which can be used in this way depends on the number of register variables, but in any event only registers 0 through 4 are available since r5 is used as a stack frame header and r6 (sp) and r7 (pc) have special hardware properties. The code generation routines assume that when called with register *n* as argument, they may use *n+1,* ... (up to the first register

variable) as temporaries. Consider the expression 'X+Y', where both X and Y are expressions. As a first approximation, there are three ways of compiling code to put this expression in register *n*.

1. If Y is an addressible cell, (recursively) put X into register *n* and add Y to it.
2. If Y is an expression that can be calculated in *k* registers, where *k* smaller than the number of registers available, compile X into register *n*, Y into register *n+1*, and add register *n+1* to *n*.
3. Otherwise, compile Y into register *n*, save the result in a temporary (actually, on the stack) compile X into register *n*, then add in the temporary.

The distinction between cases 2 and 3 therefore depends on whether the right operand can be compiled in fewer than *k* registers, where *k* is the number of free registers left after registers 0 through *n* are taken: 0 through *n−1* are presumed to contain already computed temporary results; *n* will, in case 2, contain the value of the left operand while the right is being evaluated.

These considerations should make clear the specification codes for the degree of difficulty, bearing in mind that a number of special cases are also present:

z is satisfied when the operand is zero, so that special code can be produced for expressions like 'x = 0'.

1 is satisfied when the operand is the constant 1, to optimize cases like left and right shift by 1, which can be done efficiently on the PDP-11.

c is satisfied when the operand is a positive (16-bit) constant; this takes care of some special cases in long arithmetic.

a is satisfied when the operand is addressible; this occurs not only for variables and constants, but also for some more complicated constructions, such as indirection through a simple variable, '*p++' where *p* is a register variable (because of the PDP-11's autoincrement address mode), and '*(p+c)' where *p* is a register and *c* is a constant. Precisely, the requirement is that the operand refers to a cell whose address can be written as a source or destination of a PDP-11 instruction.

e is satisfied by an operand whose value can be generated in a register using no more than *k* registers, where *k* is the number of registers left (not counting the current register). The 'e' stands for 'easy.'

n is satisfied by any operand. The 'n' stands for 'anything.'

These degrees of difficulty are considered to lie in a linear ordering and any operand which satisfies an earlier-mentioned requirement will satisfy a later one. Since the subtables are searched linearly, if a '1' specification is included, almost certainly a 'z' must be written first to prevent expressions containing the constant 0 to be compiled as if the 0 were 1.

Finally, a key specification may contain a '*' which requires the operand to have an indirection as its leading operator. Examples below should clarify the utility of this specification.

Now let us consider the contents of the code string associated with each subtable entry. Conventionally, lower-case letters in this string represent literal information which is copied directly to the output. Upper-case letters generally introduce specific macro-operations, some of which may be followed by modifying information. The code strings in the tables are written with tabs and new-lines used freely to suggest instructions which will be generated; the table-compiling program compresses tabs (using the 0200 bit of the next character) and throws away some of the new-lines. For example the macro 'F' is ordinarily written on a line by itself; but since its expansion will end with a new-line, the new-line after 'F' itself is dispensable. This is all to reduce the size of the stored tables.

The first set of macro-operations is concerned with compiling subtrees. Recall that this is done by the *cexpr* routine. In the following discussion the 'current register' is generally the

argument register to *cexpr;* that is, the place where the result is desired. The 'next register' is numbered one higher than the current register. (This explanation isn't fully true because of complications, described below, involving operations which require even-odd register pairs.)

F causes a recursive call to the *rcexpr* routine to compile code which places the value of the first (left) operand of the operator in the current register.

F1 generates code which places the value of the first operand in the next register. It is incorrectly used if there might be no next register; that is, if the degree of difficulty of the first operand is not 'easy;' if not, another register might not be available.

FS generates code which pushes the value of the first operand on the stack, by calling *rcexpr* specifying *sptab* as the table.

Analogously,

S, S1, SS
compile the second (right) operand into the current register, the next register, or onto the stack.

To deal with registers, there are

R which expands into the name of the current register.

R1 which expands into the name of the next register.

R+ which expands into the the name of the current register plus 1. It was suggested above that this is the same as the next register, except for complications; here is one of them. Long integer variables have 32 bits and require 2 registers; in such cases the next register is the current register plus 2. The code would like to talk about both halves of the long quantity, so R refers to the register with the high-order part and R+ to the low-order part.

R− This is another complication, involving division and mod. These operations involve a pair of registers of which the odd-numbered contains the left operand. *Cexpr* arranges that the current register is odd; the R− notation allows the code to refer to the next lower, even-numbered register.

To refer to addressible quantities, there are the notations:

A1 causes generation of the address specified by the first operand. For this to be legal, the operand must be addressible; its key must contain an 'a' or a more restrictive specification.

A2 correspondingly generates the address of the second operand providing it has one.

We now have enough mechanism to show a complete, if suboptimal, table for the + operator on word or byte operands.

```
%n,z
        F

%n,1
        F
        inc   R

%n,aw
        F
        add   A2,R

%n,e
        F
        S1
        add   R1,R

%n,n
        SS
        F
        add   (sp)+,R
```

The first two sequences handle some special cases. Actually it turns out that handling a right operand of 0 is unnecessary since the expression-optimizer throws out adds of 0. Adding 1 by using the 'increment' instruction is done next, and then the case where the right operand is addressible. It must be a word quantity, since the PDP-11 lacks an 'add byte' instruction. Finally the cases where the right operand either can, or cannot, be done in the available registers are treated.

The next macro-instructions are conveniently introduced by noticing that the above table is suitable for subtraction as well as addition, since no use is made of the commutativity of addition. All that is needed is substitution of 'sub' for 'add' and 'dec' for 'inc.' Considerable saving of space is achieved by factoring out several similar operations.

I is replaced by a string from another table indexed by the operator in the node being expanded. This secondary table actually contains two strings per operator.

I' is replaced by the second string in the side table entry for the current operator.

Thus, given that the entries for '+' and '−' in the side table (which is called *instab)* are 'add' and 'inc,' 'sub' and 'dec' respectively, the middle of of the above addition table can be written

```
%n,1
        F
        I'    R

%n,aw
        F
        I     A2,R
```

and it will be suitable for subtraction, and several other operators, as well.

Next, there is the question of character and floating-point operations.

B1 generates the letter 'b' if the first operand is a character, 'f' if it is float or double, and nothing otherwise. It is used in a context like 'movB1' which generates a 'mov', 'movb', or 'movf' instruction according to the type of the operand.

B2 is just like B1 but applies to the second operand.

BE generates 'b' if either operand is a character and null otherwise.

BF generates 'f' if the type of the operator node itself is float or double, otherwise null.

For example, there is an entry in *efftab* for the '=' operator

```
%a,aw
%ab,a
        IBE  A2,A1
```

Note first that two key specifications can be applied to the same code string. Next, observe that when a word is assigned to a byte or to a word, or a word is assigned to a byte, a single instruction, a *mov* or *movb* as appropriate, does the job. However, when a byte is assigned to a word, it must pass through a register to implement the sign-extension rules:

```
%a,n
        S
        IB1  R,A1
```

Next, there is the question of handling indirection properly. Consider the expression 'X + *Y', where X and Y are expressions, Assuming that Y is more complicated than just a variable, but on the other hand qualifies as 'easy' in the context, the expression would be compiled by placing the value of X in a register, that of *Y in the next register, and adding the registers. It is easy to see that a better job can be done by compiling X, then Y (into the next register), and producing the instruction symbolized by 'add (R1),R'. This scheme avoids generating the instruction 'mov (R1),R1' required actually to place the value of *Y in a register. A related situation occurs with the expression 'X + *(p+6)', which exemplifies a construction frequent in structure and array references. The addition table shown above would produce

```
[put X in register R]
mov  p,R1
add  $6,R1
mov  (R1),R1
add  R1,R
```

when the best code is

```
[put X in R]
mov  p,R1
add  6(R1),R
```

As we said above, a key specification for a code table entry may require an operand to have an indirection as its highest operator. To make use of the requirement, the following macros are provided.

F* the first operand must have the form *X. If in particular it has the form *(Y + c), for some constant *c*, then code is produced which places the value of Y in the current register. Otherwise, code is produced which loads X into the current register.

F1* resembles F* except that the next register is loaded.

S* resembles F* except that the second operand is loaded.

S1* resembles S* except that the next register is loaded.

FS* The first operand must have the form '*X'. Push the value of X on the stack.

SS* resembles FS* except that it applies to the second operand.

To capture the constant that may have been skipped over in the above macros, there are

#1 The first operand must have the form *X; if in particular it has the form *(Y + c) for *c* a constant, then the constant is written out, otherwise a null string.

#2 is the same as #1 except that the second operand is used.

Now we can improve the addition table above. Just before the '%n,e' entry, put

```
%n,ew*
        F
        S1*
        add  #2(R1),R
```

and just before the '%n,n' put

```
%n,nw*
        SS*
        F
        add  *(sp)+,R
```

When using the stacking macros there is no place to use the constant as an index word, so that particular special case doesn't occur.

The constant mentioned above can actually be more general than a number. Any quantity acceptable to the assembler as an expression will do, in particular the address of a static cell, perhaps with a numeric offset. If *x* is an external character array, the expression 'x[i+5] = 0' will generate the code

```
mov  i,r0
clrb  x+5(r0)
```

via the table entry (in the '=' part of *efftab)*

```
%e*,z
        F
        I'B1  #1(R)
```

Some machine operations place restrictions on the registers used. The divide instruction, used to implement the divide and mod operations, requires the dividend to be placed in the odd member of an even-odd pair; other peculiarities of multiplication make it simplest to put the multiplicand in an odd-numbered register. There is no theory which optimally accounts for this kind of requirement. *Cexpr* handles it by checking for a multiply, divide, or mod operation; in these cases, its argument register number is incremented by one or two so that it is odd, and if the operation was divide or mod, so that it is a member of a free even-odd pair. The routine which determines the number of registers required estimates, conservatively, that at least two registers are required for a multiplication and three for the other peculiar operators. After the expression is compiled, the register where the result actually ended up is returned. (Divide and mod are actually the same operation except for the location of the result).

These operations are the ones which cause results to end up in unexpected places, and this possibility adds a further level of complexity. The simplest way of handling the problem is always to move the result to the place where the caller expected it, but this will produce unnecessary register moves in many simple cases; 'a = b*c' would generate

```
mov  b,r1
mul  c,r1
mov  r1,r0
mov  r0,a
```

The next thought is used the passed-back information as to where the result landed to change the notion of the current register. While compiling the '=' operation above, which comes from a table entry like

```
%a,e
        S
        mov  R,A1
```

it is sufficient to redefine the meaning of 'R' after processing the 'S' which does the multiply. This technique is in fact used; the tables are written in such a way that correct code is produced. The trouble is that the technique cannot be used in general, because it invalidates the count of the number of registers required for an expression. Consider just 'a*b + X' where X is some expression. The algorithm assumes that the value of a*b, once computed, requires just one register. If there are three registers available, and X requires two registers to compute, then this expression will match a key specifying '%n,e'. If a*b is computed and left in register 1, then there are, contrary to expectations, no longer two registers available to compute X, but only one, and bad code will be produced. To guard against this possibility, *cexpr* checks the result returned by recursive calls which implement F, S and their relatives. If the result is not in the expected register, then the number of registers required by the other operand is checked; if it can be done using those registers which remain even after making unavailable the unexpectedly-occupied register, then the notions of the 'next register' and possibly the 'current register' are redefined. Otherwise a register-copy instruction is produced. A register-copy is also always produced when the current operator is one of those which have odd-even requirements.

Finally, there are a few loose-end macro operations and facts about the tables. The operators:

V is used for long operations. It is written with an address like a machine instruction; it expands into 'adc' (add carry) if the operation is an additive operator, 'sbc' (subtract carry) if the operation is a subtractive operator, and disappears, along with the rest of the line, otherwise. Its purpose is to allow common treatment of logical operations, which have no carries, and additive and subtractive operations, which generate carries.

T generates a 'tst' instruction if the first operand of the tree does not set the condition codes correctly. It is used with divide and mod operations, which require a sign-extended 32-bit operand. The code table for the operations contains an 'sxt' (sign-extend) instruction to generate the high-order part of the dividend.

H is analogous to the 'F' and 'S' macros, except that it calls for the generation of code for the current tree (not one of its operands) using *regtab*. It is used in *cctab* for all the operators which, when executed normally, set the condition codes properly according to the result. It prevents a 'tst' instruction from being generated for constructions like 'if (a+b) ...' since after calculation of the value of 'a+b' a conditional branch can be written immediately.

All of the discussion above is in terms of operators with operands. Leaves of the expression tree (variables and constants), however, are peculiar in that they have no operands. In order to regularize the matching process, *cexpr* examines its operand to determine if it is a leaf; if so, it creates a special 'load' operator whose operand is the leaf, and substitutes it for the argument tree; this allows the table entry for the created operator to use the 'A1' notation to load the leaf into a register.

Purely to save space in the tables, pieces of subtables can be labelled and referred to later. It turns out, for example, that rather large portions of the the *efftab* table for the '=' and '=+' operators are identical. Thus '=' has an entry

```
%[move3:]
%a,aw
%ab,a
        IBE  A2,A1
```

while part of the '=+' table is

```
%aw,aw
%       [move3]
```

Labels are written as '%[... :]', before the key specifications; references are written with '% [...]' after the key. Peculiarities in the implementation make it necessary that labels appear

before references to them.

The example illustrates the utility of allowing separate keys to point to the same code string. The assignment code works properly if either the right operand is a word, or the left operand is a byte; but since there is no 'add byte' instruction the addition code has to be restricted to word operands.

Delaying and reordering

Intertwined with the code generation routines are two other, interrelated processes. The first, implemented by a routine called *delay,* is based on the observation that naive code generation for the expression 'a = b++' would produce

```
mov  b,r0
inc  b
mov  r0,a
```

The point is that the table for postfix ++ has to preserve the value of *b* before incrementing it; the general way to do this is to preserve its value in a register. A cleverer scheme would generate

```
mov  b,a
inc  b
```

Delay is called for each expression input to *rcexpr,* and it searches for postfix ++ and −− operators. If one is found applied to a variable, the tree is patched to bypass the operator and compiled as it stands; then the increment or decrement itself is done. The effect is as if 'a = b; b++' had been written. In this example, of course, the user himself could have done the same job, but more complicated examples are easily constructed, for example 'switch (x++)'. An essential restriction is that the condition codes not be required. It would be incorrect to compile 'if (a++) ...' as

```
tst  a
inc  a
beq  ...
```

because the 'inc' destroys the required setting of the condition codes.

Reordering is a similar sort of optimization. Many cases which it detects are useful mainly with register variables. If *r* is a register variable, the expression 'r = x+y' is best compiled as

```
mov  x,r
add  y,r
```

but the codes tables would produce

```
mov  x,r0
add  y,r0
mov  r0,r
```

which is in fact preferred if *r* is not a register. (If *r* is not a register, the two sequences are the same size, but the second is slightly faster.) The scheme is to compile the expression as if it had been written 'r = x; r =+ y'. The *reorder* routine is called with a pointer to each tree that *rcexpr* is about to compile; if it has the right characteristics, the 'r = x' tree is constructed and passed recursively to *rcexpr;* then the original tree is modified to read 'r =+ y' and the calling instance of *rcexpr* compiles that instead. Of course the whole business is itself recursive so that more extended forms of the same phenomenon are handled, like 'r = x + y | z'.

Care does have to be taken to avoid 'optimizing' an expression like 'r = x + r' into 'r = x; r =+ r'. It is required that the right operand of the expression on the right of the '=' be a ', distinct from the register variable.

The second case that *reorder* handles is expressions of the form 'r = X' used as a subexpression. Again, the code out of the tables for 'x = r = y' would be

```
mov y,r0
mov r0,r
mov r0,x
```

whereas if *r* were a register it would be better to produce

```
mov y,r
mov r,x
```

When *reorder* discovers that a register variable is being assigned to in a subexpression, it calls *rcexpr* recursively to compile the subexpression, then fiddles the tree passed to it so that the register variable itself appears as the operand instead of the whole subexpression. Here care has to be taken to avoid an infinite regress, with *rcexpr* and *reorder* calling each other forever to handle assignments to registers.

A third set of cases treated by *reorder* comes up when any name, not necessarily a register, occurs as a left operand of an assignment operator other than '=' or as an operand of prefix '++' or '--'. Unless condition-code tests are involved, when a subexpression like '(a =+ b)' is seen, the assignment is performed and the argument tree modified so that *a* is its operand; effectively 'x + (y =+ z)' is compiled as 'y =+ z; x + y'. Similarly, prefix increment and decrement are pulled out and performed first, then the remainder of the expression.

Throughout code generation, the expression optimizer is called whenever *delay* or *reorder* change the expression tree. This allows some special cases to be found that otherwise would not be seen.

Introduction to the f77 I/O Library

David L. Wasley

University of California, Berkeley
Berkeley, California 94720

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of FORTRAN input and output. Several enhancements and extensions to FORTRAN I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

1. FORTRAN I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very "expensive" while direct access binary I/O is usually very efficient. Because of the complexity of FORTRAN I/O, some general concepts deserve clarification.

1.1. Types of I/O

There are three forms of I/O: **formatted, unformatted,** and **list-directed.** The last is related to formatted but does not obey all the rules for formatted I/O. There are two modes of access to **external** and **internal** files: **direct** and **sequential.** The definition of a logical record depends upon the combination of I/O form and mode specified by the FORTRAN I/O statement.

1.1.1. Direct access

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

1.1.2. Sequential access

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The "newline" character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with "newline" characters to be read or written.

1.1.3. List directed I/O

Logical record length for **list-directed** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents.

1.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** I/O is not allowed on "internal" files.

1.2. I/O execution

Note that each execution of a FORTRAN **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a FORTRAN **formatted** I/O statement causes one or more logical records to be read or written.

A slash, "/", will terminate assignment of values to the input list during **list-directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

Direct access list-directed I/O is not allowed. **Unformatted internal** I/O is not allowed. Both the above will be caught by the compiler. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement may include an **err=** clause (and **iostat=** clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements may include **end=** to branch on end-of-file. File position and the value of I/O list items is undefined following an error.

2. Implementation details

Some details of the current implementation may be useful in understanding constraints on FORTRAN I/O.

2.1. Number of logical units

The maximum number of logical units that a program may have open at one time is the same as the UNIX† system limit, currently 20. Unit numbers must be in the range 0 – 19 because they are used to index an internal control table.

2.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to "stderr", "stdin", and "stdout" respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name (see § 2.4).

The standard units, 0, 5, and 6, are named internally "stderr", "stdin", and "stdout" respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

---

† UNIX is a trademark of Bell Laboratories.

2.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = 'print'** (see § 3.2). Control codes "0" and "1" are replaced in the output file with "\n" and "\f" respectively. The control character "+" is not implemented and, like any other character in the first position of a record written to a "print" file, is dropped. No vertical format control is recognized for **direct formatted** output or **list directed** output.

2.4. The open statement

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **status = 'scratch'** is specified, a temporary file with a name of the form "tmp.FXXXXXX" will be opened, and, by default, will be deleted when closed or during termination of program execution. Any other **status=** specifier without an associated file name results in opening a file named "fort.N" where N is the specified logical unit number.

It is an error to try to open an existing file with **status = 'new'** . It is an error to try to open a nonexistent file with **status = 'old'** . By default, **status = 'unknown'** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close** , **backspace** , or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.5. Format interpretation

Formats are parsed at the beginning of each execution of a formatted I/O statement. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*). On **E**w.d**E**e output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if "e" is zero (see appendix B).

On output, a real value that is truly zero will display as "0." to distinguish it from a very small non-zero value. This occurs in **F** and **G** format conversions. This was not done for **E** and **D** since the embedded blanks in the external datum causes problems for other input systems.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is not, it will have a different meaning (see § 3.1).

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List-directed output of **complex** values includes an appropriate comma. List-directed output distinguishes between **real** and **double precision** values and formats them differently. Output of a character string that includes "\n" is interpreted reasonably by the output system.

2.7. I/O errors

If I/O errors are not trapped by the user's program an appropriate error message will be written to "stderr" before aborting. An error number will be printed in [ ] along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors, and are described in the introduction to chapter 2 of the UNIX Programmer's Manual. Error numbers ⩾ 100 come from the I/O library, and are described further in the appendix to this writeup. For internal I/O, part of the string will be printed with "|" at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

3. Non-"ANSI Standard" extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

3.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **E**w.d**E**e format specifier has been extended to **D** also. The form **E**w.d.e is allowed but is not standard. The "e" field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the "e" field will be filled with asterisks (*). The default value for "e" is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TR**n, **TL**n, and **T**n are supported where *n* is a positive non-zero number. If **T** or n**T** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P,** the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as [n]**R** where $2 \leqslant n \leqslant 36$. If *n* is omitted, the default decimal radix is restored.

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete. Radix and "unsigned" specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 16R, 8I10.8 )
```

Note: Unsigned integer values greater than (2**30 - 1), i.e. any signed negative value, can not be read by FORTRAN input routines. All internal values will be output correctly.

3.2. Print files

The ANSI standard is ambiguous regarding the definition of a "print" file. Since UNIX has no default "print" file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **form = 'print'** implies **formatted** and enables vertical format control for that logical unit. Vertical format control is interpreted only on sequential formatted writes to a "print" file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as "print" files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a "print" file, use:

```
open (unit=6, form='print')
```

3.3. Scratch files

A **close** statement with **status = 'keep'** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file's real name, using **inquire** , if you want to re-open it later.

3.4. List directed I/O

List directed read has been modified to allow input of a string not enclosed in quotes. The string must not start with a digit, and can not contain a separator (, or /) or blank (space or tab). A newline will terminate the string unless escaped with x Any string not meeting the above restrictions must be enclosed in quotes (" or ').

Internal list-directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the 'end-of-file' is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

4. Running older programs

Traditional FORTRAN environments usually assume carriage control on all logical units, usually interpret blank spaces on input as "0"s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

4.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **−lI66** in the f77 command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

4.2. Preattachment of logical units

The *ioinit* routine also can be used to attach logical units to specific files at run time. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading "0".

Ioinit should prove adequate for most programs as written. However, it is written in FORTRAN–77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by "ar x /usr/lib/libI77.a ioinit.f".

5. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal FORTRAN I/O processing and do not obey FORTRAN I/O rules. See *tapeio*(3f).

6. Caveat Programmer

The I/O library is extremely complex yet we believe there are few bugs left. We've tried to make the system as correct as possible according to the ANSI X3.9–1978 document and keep it compatible with the UNIX file system. Exceptions to the standard are noted in appendix B.

Appendix A

I/O Library Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable if the **err=** return is taken. Error numbers < 100 are generated by the UNIX kernel. See the introduction to chapter 2 of the UNIX Programmers Manual for their description.

/* 100 */ "error in format"
See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested (), or an extremely long format statement.

/* 101 */ "illegal unit number"
It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is system dependent.

/* 102 */ "formatted io not allowed"
The logical unit was opened for unformatted I/O.

/* 103 */ "unformatted io not allowed"
The logical unit was opened for formatted I/O.

/* 104 */ "direct io not allowed"
The logical unit was opened for sequential access, or the logical record length was specified as 0.

/* 105 */ "sequential io not allowed"
The logical unit was opened for direct access I/O.

/* 106 */ "can't backspace file"
The file associated with the logical unit can't seek. May be a device or a pipe.

/* 107 */ "off beginning of record"
The format specified a left tab beyond the beginning of an internal input record.

/* 108 */ "can't stat file"
The system can't return status information about the file. Perhaps the directory is unreadable.

/* 109 */ "no * after repeat count"
Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

/* 110 */ "off end of record"
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

/* 111 */ "truncation failed"
The truncation of an external sequential file on 'close', 'backspace', 'rewind' or 'endfile' failed.

/* 112 */ "incomprehensible list input"
List input has to be just right.

/* 113 */ "out of free space"
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

/* 114 */ "unit not connected"
The logical unit was not open.

/* 115 */ "read unexpected character"
Certain format conversions can't tolerate non-numeric data. Logical data must be T or F.

/* 116 */ "blank logical input field"

/* 117 */ "'new' file exists"
You tried to open an existing file with "status='new'".

/* 118 */ "can't find 'old' file"
You tried to open a non-existent file with "status='old'".

/* 119 */ "unknown system error"
Shouldn't happen, but .....

/* 120 */ "requires seek ability"
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

/* 121 */ "illegal argument"
Certain arguments to 'open', etc. will be checked for legitimacy. Often only non-default forms are looked for.

/* 122 */ "negative repeat count"
The repeat count for list directed input must be a positive integer.

/* 123 */ "illegal operation for unit"
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

Appendix B

Exceptions to the ANSI Standard

A few exceptions to the ANSI standard remain.

1) Vertical format control

The "+" carriage control specifier is not implemented. It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a "FORTRAN output filter" before printing. This filter could recognize a much broader range of carriage control and include terminal dependent processing.

2) Default files

Files created by default use of **rewind** or **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

3) Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

4) Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, 'w', should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more diagnostic.

A Portable Fortran 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The Fortran language has been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by S.I.F., the I/O system by P.J.W. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler [4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is

f77 *flags file* . . .

f77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The **f77** and **cc** commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

.f	Fortran source file
.F	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

Arguments whose names end with **.f** are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with **.o** substituted for **.f**.

Arguments whose names end with **.F** are also taken to be Fortran 77 source programs; these are first processed by the C preprocessor before being compiled by f77.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by f77.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing a **.o** file.

The following flags are understood:

−c	Compile but do not load. Output for **x.f**, **x.F**, **x.e**, **x.r**, **x.c**, or **x.s** is put on file **x.o**.
−g	Have the compiler produce additional symbol table information for *dbx(1)*. This only applies on the Vax UNIX system. Do not use with **−O**.
−i2	On machines which support short integers, make the default integer constants and variables short (see section 2.14). (**−i4** is the standard value of this option). All logical quantities will be short.
−m	Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.
−o *file*	Put executable module on file *file*. (Default is **a.out**).
−onetrip	Compile code that performs every **do** loop at least once (see section 2.12).
−p	Generate code to produce usage profiles.
−pg	Generate code in the manner of **−p**, but invoke a run-time recording mechanism that keeps more extensive statistics.
−w	Suppress all warning messages.
−w66	Suppress warnings about Fortran 66 features used.
−u	Make the default type of a variable **undefined** (see section 2.3).
−C	Compile code that checks that subscripts are within array bounds.
−D*name*=*def*	
−D*name*	Define the *name* to the C preprocessor, as if by '#define'. If no definition is given, the name is defined as "1". (**.F** files only).
−E*str*	Use the string *str* as an EFL option in processing **.e** files.
−F	Ratfor and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed.
−I*dir*	'#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **−I** options, then in directories on a standard list. (**.F** files only).
−O	Invoke the object code optimizer. Do not use with **−g**.
−R*str*	Use the string *str* as a Ratfor option in processing **.r** files.
−U	Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case except within character string constants.
−S	Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f**, **x.F**, **x.e**, **x.r**, or **x.c** is put on file **x.s**.

Other flags, all library names (arguments beginning **−l**), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in Appendix A. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces "internal files" (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in formatted direct reads and writes.

2.3. Implicit Undefined Statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined**. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **−u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic**. These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored.)

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand "&" in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **−U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**; **include** statements may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0**–**7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0**–**9**, **a**–**f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
****	\
*****x*	*x*, where *x* is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe " ' " and the double-quote " " ". If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith "*n***h**" notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **–onetrip** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
–345,.05e–3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2**. (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **–i2** flag, all small integer constants will be of type **integer*2**. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **–i2** command flag is in effect). When the **–i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**).

3. VIOLATIONS OF THE STANDARD

We know only a few ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran Standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed (section 6.3.2 in Appendix A). The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

3.4. Carriage Control

The Standard leaves as implementation dependent which logical unit(s) are treated as "printer" files. In this implementation there is no printer file and thus no carriage control is recognized on formatted output, except by special arrangement [9].

3.5. Assigned Goto

The optional *list* associated with an assigned **goto** statement is not checked against the actual assigned value during execution.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	**C**
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.)

4.3. Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
. . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
. . .
```

and could be invoked in C by

```
char chars[15];
...
g_(chars, 15L, ...);
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on *records*. When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than

being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: the I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist, nor will they be created unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. The I/O system will position the file at the beginning. Therefore a **write** will destroy any data already in the file, but a **read** will work reasonably. To position a file to its end, use a 'read' loop, or the system dependent 'fseek' function. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

APPENDIX A: Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. The best current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute, and the ANSI X3.9-1978 document, the official description of the language. The Standard is written in English rather than a meta-language, but it is forbidding and legalistic. A number of tutorials and textbooks are available (see Appendix B).

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of "Hollerith" (*n***h**) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick

of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use.) The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l+d)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the "alternate returns" is described in section 5.2 of Appendix A.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string.) Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer**; other variables are of type **real**, unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a** ,**b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966.) The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed.) These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, " ' " and " " ". (See section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash "//". The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "*(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments.)

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leqslant n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used.) Also, multiple exponentiation is now defined:

```
a**b**c is equivalent to a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the Fortran operators, except for exponentiation to a floating-point power.) An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common.

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if (. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **else if**, **else**, or **end if** are executed. Otherwise, the next **else if** statement in the group is executed. If none of the **else if** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** block must follow all **else if** blocks in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures.) A case construct may be rendered:

```
if (s .eq. 'ab') then
. . .
else if (s .eq. 'cd') then
. . .
else
. . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in:

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as:

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leqslant k \leqslant n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in:

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=**, **err=**, and **iostat=** clauses, as in:

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. It is not allowed to read into character constants or hollerith fields.

A format may be specified as a character constant within the **read** or **write** statement.

```
write(6,'(i2," isn""t ",i1)') 7, 4
```

produces

```
 7 isn't 4
```

In the example above, the format is the character constant

```
(i2,' isn''t ',i1)
```

and the imbedded character constant

```
isn't
```

is copied into the output.

The example could have been written more legibly by taking advantage of the two types of quote marks.

```
write(6,'(i2," isn''t ",i1)') 7, 4
```

However, the double quote is not standard Fortran 77.

6.3.2. Positional Editing Codes

t, **tl**, **tr**, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.3 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints

```
hello there 12
```

while the second only prints

```
hello
```

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, **ss** and **s** codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks, will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m*. It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w*.1 is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e** or **d.** The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending on the size of the number and of *d*.

6.3.9. "A" Format Code

The **a** code is used for character data. **a***w* uses a field width of *w*, while a plain **a** uses the length of the internal character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output unit is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**.

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file; in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**.) There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,'(a)') x
read(x,'(i3,i4)') n1,n2
```

which reads a character string into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case a record is a single element of an array of character strings.

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 19. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status='scratch'.**

status= one of **'old'**, **'new'**, **'scratch'**, or **'unknown'**. If this parameter is not given, **'unknown'** is assumed. The meaning of **'unknown'** is processor dependent; our system will create the file if it doesn't exist. If **'scratch'** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **'new'** is given, the file must not exist. It will be created for both reading and writing. If **'old'** is given, it is an error for the file not to exist.

access= **'sequential'** or **'direct'**, depending on whether the file is to be opened for sequential or direct I/O.

form= **'formatted'** or **'unformatted'**. On UNIX systems **form='print'** implies **'formatted'** with vertical format control.

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= **'null'** or **'zero'**. This parameter has meaning only for formatted I/O. The default value is **'null'**. **'zero'** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **'keep'** or **'delete'**. For scratch files the default is **'delete'**; otherwise **'keep'** is the default. **'delete'** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit ("inquire by unit") or a file ("inquire by file"). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value '**sequential**' if the connection is for sequential I/O, '**direct**' if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value '**yes**' if the file could be connected for sequential I/O, '**no**' if the file could not be connected for sequential I/O, and '**unknown**' if we can't tell.

direct= a character variable to which is assigned the value '**yes**' if the file could be connected for direct I/O, '**no**' if the file could not be connected for direct I/O, and '**unknown**' if we can't tell.

form= a character variable to which is assigned the value **unformatted**' if the file is connected for unformatted I/O, '**formatted**' if the file is connected for formatted I/O, or '**print**' for formatted I/O with vertical format control.

formatted= a character variable to which is assigned the value '**yes**' if the file could be connected for formatted I/O, '**no**' if the file could not be connected for formatted I/O, and '**unknown**' if we can't tell.

unformatted= a character variable to which is assigned the value '**yes**' if the file could be connected for unformatted I/O, '**no**' if the file could not be connected for unformatted I/O, and '**unknown**' if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value '**null**' if null blank control is in effect for the file connected for formatted I/O, '**zero**' if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the Standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file='/dev/console')
```

On a UNIX system this statement opens the console for formatted sequential I/O. An

inquire statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the FORTRAN environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

For further discussion of the UNIX Fortran I/O system see "Introduction to the f77 I/O Library" [9].

APPENDIX B: References and Bibliography

References

1. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.* New York: American National Standards Institute, 1978.
2. *USA Standard FORTRAN, USAS X3.9-1966.* New York: United States of America Standards Institute, 1966. Clarified in *Comm. ACM* 12:289 (1969) and *Comm. ACM* 14:628 (1971).
3. Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Englewood Cliffs: Prentice-Hall, 1978.
4. Ritchie, D. M. Private communication.
5. Johnson, S. C. "A Portable Compiler: Theory and Practice," *Proceedings of Fifth ACM Symposium on Principles of Programming Languages.* 1978.
6. Feldman, S. I. "An Informal Description of EFL," internal memorandum.
7. Kernighan, B. W. "RATFOR—A Preprocessor for Rational Fortran," *Bell Laboratories Computing Science Technical Report #55.* 1977.
8. Ritchie, D. M. Private communication.
9. Wasley, D. L. "Introduction to the f77 I/O Library", *UNIX Programmer's Manual, Volume 2c.*

Bibliography

The following books or documents describe aspects of Fortran 77. This list cannot pretend to be complete. Certainly no particular endorsement is implied.

1. Brainerd, Walter S., et al. *Fortran 77 Programming.* Harper Row, 1978.
2. Day, A. C. *Compatible Fortran.* Cambridge University Press, 1979.
3. Dock, V. Thomas. *Structured Fortran IV Programming.* West, 1979.
4. Feldman, S. I. "The Programming Language EFL," *Bell Laboratories Technical Report.* June 1979.
5. Hume, J. N., and R. C. Holt. *Programming Fortran 77.* Reston, 1979.
6. Katzan, Harry, Jr. *Fortran 77.* Van Nostrand-Reinhold, 1978.
7. Meissner, Loren P., and Organick, Elliott I. *Fortran 77 Featuring Structured Programming,* Addison-Wesley, 1979.
8. Merchant, Michael J. *ABC's of Fortran Programming.* Wadsworth, 1979.
9. Page, Rex, and Richard Didday. *Fortran 77 for Humans.* West, 1980.
10. Wagener, Jerrold L. *Principles of Fortran 77 Programming.* Wiley, 1980.

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N-1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would

This paper is a revised and expanded version of one published in *Software—Practice and Experience,* October 1975. The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J. UNIX is a Trademark of Bell Laboratories

require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like **"x>100"**), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**'s: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to −1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        ---
else if (...)
        ---
else if (...)
        ---
...
else
        ---
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

switch (*expression*) {

case *expr1* :
statements
case *expr2, expr3* :
statements
...
default:
statements
}

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression,* and there is a **default** section, the statements with it are done; if there is no **default,** nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
        do 10 i = 1, n
                x(i) = 0.0
                y(i) = 0.0
                z(i) = 0.0
10      continue
```

The syntax is:

do *legal-Fortran-DO-text*
Ratfor statement

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ———
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
        # returns sin(x) to accuracy e, by
        # sin(x) = x - x**3/3! + x**5/5! - ...

        sin = x
        term = x

        i = 3
        while (abs(term)>e & i<100) {
                term = -term * x**2 / float(i*(i-1))
                sin = sin + term
                i = i + 2
        }

        return
        end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not

in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
          I = 81 -J
          IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value −1.

```
# equal _compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1) {
                        equal = 1
                        return
                }
        equal = 0
        return
        end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function **F**, **return**(expression) is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1)
                        return(1)
        return(0)
        end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is

needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + _ * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	~	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)

      if (i > ROWS  | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define    YES     1
define    NO      0
define    EOS     -1
define    ARB     100

# equal __compare str1 to str2;
#     return YES if equal, NO if not
      integer function equal(str1, str2)
      integer str1(ARB), str2(ARB)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == EOS)
                  return(YES)
      return(NO)
      end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever

a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog    : stat
        | prog  stat
stat    : if (...) stat
        | if (...) stat else stat
        | while (...) stat
        | for (...; ...; ...) stat
        | do ... stat
        | repeat stat
        | repeat stat until (...)
        | switch (...) { case ...: prog ...
                         default: prog }
        | return
        | break
        | next
        | digits  stat
        | { prog }
        | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else.** Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no

compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v \pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the gen-

erated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience,* October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter,* August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

Appendix: Usage on UNIX and GCOS.

Beware — local customs vary. Check with a native before going into the jungle.

UNIX

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **−6x**, which uses **x** as a continuation character in column 6 (UNIX uses **&** in column 1), and **−C**, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as **cc**. Thus

rc [options] files

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags **−C** and **−6x** described above are recognized, as are

-c	compile only; don	load
-f	save intermediate Fortran .f files
-r	Ratfor only; implies -c and -f
-2	use big Fortran compiler (for large programs)
-U	flag undeclared variables (not universally available)

Other flags are passed on to the loader.

GCOS

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is **&** in column 6. Thus

./ratfor files >output

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

./rc provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

name	Ratfor source or library, depending on type
h=/name	make TSS H* file (runnable version); run as /name
r=/name	update and use random library
a=	compile as ascii (default is bcd)
C=	copy comments into Fortran
f=name	Fortran source file
g=name	gmap source file

Other options are as specified for the **./cc** command described in [4].

TSO, TSS, and other systems

Ratfor exists on various other systems; check with the author for specifics.

The Programming Language EFL

Stuart I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[*item* **]**

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + − * /
	= < > & ~ | $

Letter case (upper or lower) is ignored except within strings, so 'a' and 'A' are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ('!') may be used in place of a tilde ('~'). Square brackets ('[' and ']') may be used in place of braces ('{' and '}').

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

```
include joe
```

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
'hello there'
"ain't misbehavin'"
```

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

```
0
57
123456
```

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+   -   *   /   **
<   <=  >   >=  ==  ~=
&&  ||  &   |
+=  -=      /=  **=
&&= ||=     &=  |=
->  .  $
```

A dot ('.') is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, .lt.).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
. . .
if(x > 2)
        {           # new block
        integer x   # a different variable
        do x = 1,7
                    write(,x)
        . . .
        }           # end of block
end     # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in Section 10. The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
          read(, x)
          if(x < 3) goto error
          . . .
error:    fatal("bad input")
```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A 'real' quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ('double precision') constant is a floating point constant containing an exponent field that begins with the letter **d**. A real ('single precision') constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 (= 7.9×10^{-6})
14e9 (= 1.4×10^{10})

The following are valid **long real** constants

7.9d−6 (= 7.9×10^{-6})
5d3

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in

input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(expression)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=  &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6, -3, 4)
```

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument.* Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a nonzero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3*i*

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary + has no effect. A unary − yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator −− subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or**, are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified.

The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b**. The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise the expression has the value of **b**. The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	$<$	less than
<=	$\leqslant$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geqslant$	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside -> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) -> st.elt

refers to the **elt** member of the st structure starting at the i^{th} element of the array **place**.

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

logical
integer
field(*m*:*n*)
character(*k*)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

6.2.2. Arrays

The dimensionality may be declared by an **array** attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper*−*lower*+1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, −3:0)
array(5, *)
array(0:m−1, m)
```

6.2.3. Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same

block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [*name*]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [*var* = *val*]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements — otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i     # this variable is unknown outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the if statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □ **else** □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-elsed **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select (*expression*) □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case [*constant*] :

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not 'fall through' to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* , □ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

```
initial-statement
while ( logical-expression )
    {
    body-statement
    iteration-statement
    }
```

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
        ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat...Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement*, then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement*. Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until**. In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
    statement

The variable is first given the value *expression-1*. The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2*. If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2*. The sum of the first hundred positive integers could

be computed by

```
n = 0
do i = 1, 100
        n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
   {
   do a computation
   if(finished)
        break
   }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat...until**, or the increment of a **do**. Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a 'unit', identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit*. These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records*. These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file*.

7.7.2. Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* , *binary-output-list*)
readbin(*unit* , *binary-input-list*)

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* , *formatted-output-list*)
read(*unit* , *formatted-input-list*)

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers*. The quantities w, d, and k must be integer constant expressions.

i(*w*)	integer with *w* digits
f(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter e
l(*w*)	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation statements

The three input/output statements

backspace(*unit*)
rewind(*unit*)
endfile(*unit*)

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* ( )
attributes **procedure** *procedurename* ([*name*])

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

end

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real.** Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, -3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real, long real,** or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function** (e^x).
log	**natural (base** *e***) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function** ($\sqrt{x}$).

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x) = \tan^{-1} x$
atan2	$atan2(x,y) = \tan^{-1} \frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; $\mathbf{sign}(x,y) = sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ('%') is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call**.

```
call joe
call work(17)
```

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a−h, o−z) real
implicit (i−n) integer
```

9.6. Computed goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([*label*]), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Go To Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	.lt.
<=	.le.
>	.gt.
>=	.ge.
==	.eq.
~=	.ne.
&	.and.
|	.or.
&&	.andand.
||	.oror.
~	.not.
true	.true.
false	.false.

In this mode, no structure element may be named lt, le, etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

9.10. Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence v_1, v_2, ..., v_n

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option ⟦ *opt* ⟧

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names yes and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos**.

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no**.

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (&) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	Type
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	Size Option	Alignment Option
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per integer variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
            write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
        {
        c(i,j) = 0
        do k = 1,n
                c(i,j) += a(i,k) * b(k,j)
        }
end
```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST         0
define NOTFOUND   -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value, an x, and a y value.
struct
            {
            integer nextindex
            integer x, y
            } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x , p = list(p).nextindex)
            if(list(p).x == x)
                        return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
            print its value
otherwise
            print a left parenthesis
            print the left node
            print the operator
            print the right node
            print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first) # print out an expression tree

integer first                   # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100) # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE         tree(currentnode)
define STACK        stackframe(stackdepth)

#  nextstate values
define DOWN         1
define LEFT         2
define RIGHT        3

#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " ")   # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    {  # a binary operator node
                                outch( "(" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }
                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp
                case RIGHT:
                        outch( ")" )
                        stackdepth -= 1
                }
        }
end
```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

12.1.2. Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb**. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

1. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran", Bell Laboratories Computing Science Technical Report #55

APPENDIX A. Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do  1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk (Section 11.4):

†UNIX is a Trademark of Bell Laboratories.

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real treel(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), treel(1,1))
      data const1(1)/4h    /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
   1  if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
   2        if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(treel(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
   3           call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4        goto  8
   5        call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
   6        call outch(1h))
            stapth = stapth-1
            goto  8
   7        if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
   8     continue
         goto  1
   9  continue
      end
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs

describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Berkeley Pascal User's Manual
Version 3.0 – July 1983

William N. Joy, Susan L. Graham, Charles B. Haley‡,
Marshall Kirk McKusick, and Peter B. Kessler

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

The Berkeley Pascal *User's Manual* consists of five major sections and an appendix. In section 1 we give sources of information about UNIX, about the programming language Pascal, and about the Berkeley implementation of the language. Section 2 introduces the Berkeley implementation and provides a number of tutorial examples. Section 3 discusses the error diagnostics produced by the translators *pc* and *pi*, and the runtime interpreter *px*. Section 4 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX. Section 5 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

History of the implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system were implemented by Charles Haley and William Joy during the latter half of 1976. Earlier versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick with the porting of the interpreter in the spring of 1979, and the implementation of the compiler in the summer of 1980.

1. Sources of information

This section lists the resources available for information about general features of UNIX,† text editing, the Pascal language, and the Berkeley Pascal implementation, concluding with a list of references. The available documents include both so-called standard documents — those distributed with all UNIX system — and documents (such as this one) written at Berkeley.

1.1. Where to get documentation

Current documentation for most of the UNIX system is available "on line" at your terminal. Details on getting such documentation interactively are given in section 1.3.

1.2. Documentation describing UNIX

The following documents are those recommended as tutorial and reference material about the UNIX system. We give the documents with the introductory and tutorial materials first, the reference materials last.

UNIX For Beginners — Second Edition

This document is the basic tutorial for UNIX available with the standard system.

Communicating with UNIX

This is also a basic tutorial on the system and assumes no previous familiarity with computers; it was written at Berkeley.

An introduction to the C shell

This document introduces *csh*, the shell in common use at Berkeley, and provides a good deal of general description about the way in which the system functions. It provides a useful glossary of terms used in discussing the system.

UNIX Programmer's Manual

This manual is the major source of details on the components of the UNIX system. It consists of an Introduction, a permuted index, and eight command sections. Section 1 consists of descriptions of most of the "commands" of UNIX. Most of the other sections have limited relevance to the user of Berkeley Pascal, being of interest mainly to system programmers.

UNIX documentation often refers the reader to sections of the manual. Such a reference consists of a command name and a section number or name. An example of such a reference would be: *ed* (1). Here *ed* is a command name — the standard UNIX text editor, and '(1)' indicates that its documentation is in section 1 of the manual.

The pieces of the Berkeley Pascal system are *pi* (1), *px* (1), the combined Pascal translator and interpretive executor *pix* (1), the Pascal compiler *pc* (1), the Pascal execution profiler *pxp* (1), and the Pascal cross-reference generator *pxref* (1).

It is possible to obtain a copy of a manual section by using the *man* (1) command. To get the Pascal documentation just described one could issue the command:

```
% man pi
```

to the shell. The user input here is shown in **bold face**; the '% ', which was printed by the shell as a prompt, is not. Similarly the command:

```
% man man
```

†UNIX is a Trademark of Bell Laboratories.

asks the *man* command to describe itself.

1.3. Text editing documents

The following documents introduce the various UNIX text editors. Most Berkeley users use a version of the text editor *ex;* either *edit*, which is a version of *ex* for new and casual users, *ex* itself, or *vi* (visual) which focuses on the display editing portion of *ex*.

A Tutorial Introduction to the UNIX Text Editor

This document, written by Brian Kernighan of Bell Laboratories, is a tutorial for the standard UNIX text editor *ed*. It introduces you to the basics of text editing, and provides enough information to meet day-to-day editing needs, for *ed* users.

Edit: A tutorial

This introduces the use of *edit*, an editor similar to *ed* which provides a more hospitable environment for beginning users.

Ex/edit Command Summary

This summarizes the features of the editors *ex* and *edit* in a concise form. If you have used a line oriented editor before this summary alone may be enough to get you started.

Ex Reference Manual — Version 3.5

A complete reference on the features of *ex* and *edit*.

An Introduction to Display Editing with Vi

Vi is a display oriented text editor. It can be used on most any CRT terminal, and uses the screen as a window into the file you are editing. Changes you make to the file are reflected in what you see. This manual serves both as an introduction to editing with *vi* and a reference manual.

Vi Quick Reference

This reference card is a handy quick guide to *vi;* you should get one when you get the introduction to *vi*.

1.4. Pascal documents — The language

This section describes the documents on the Pascal language which are likely to be most useful to the Berkeley Pascal user. Complete references for these documents are given in section 1.7.

Pascal User Manual

By Kathleen Jensen and Niklaus Wirth, the *User Manual* provides a tutorial introduction to the features of the language Pascal, and serves as an excellent quick-reference to the language. The reader with no familiarity with Algol-like languages may prefer one of the Pascal text books listed below, as they provide more examples and explanation. Particularly important here are pages 116-118 which define the syntax of the language. Sections 13 and 14 and Appendix F pertain only to the 6000-3.4 implementation of Pascal.

Pascal Report

By Niklaus Wirth, this document is bound with the *User Manual*. It is the guiding reference for implementors and the fundamental definition of the language. Some programmers find this report too concise to be of practical use, preferring the *User Manual* as a reference.

Books on Pascal

Several good books which teach Pascal or use it as a medium are available. The books by Wirth *Systematic Programming* and *Algorithms + Data Structures = Programs* use Pascal as a vehicle for teaching programming and data structure concepts respectively. They are both recommended. Other books on Pascal are listed in the references below.

1.5. Pascal documents — The Berkeley Implementation

This section describes the documentation which is available describing the Berkeley implementation of Pascal.

User's Manual

The document you are reading is the *User's Manual* for Berkeley Pascal. We often refer the reader to the Jensen-Wirth *User Manual* mentioned above, a different document with a similar name.

Manual sections

The sections relating to Pascal in the *UNIX Programmer's Manual* are *pix* (1), *pi* (1), *pc* (1), *px* (1), *pxp* (1), and *pxref* (1). These sections give a description of each program, summarize the available options, indicate files used by the program, give basic information on the diagnostics produced and include a list of known bugs.

Implementation notes

For those interested in the internal organization of the Berkeley Pascal system there are a series of *Implementation Notes* describing these details. The *Berkeley Pascal PXP Implementation Notes* describe the Pascal interpreter *px*; and the *Berkeley Pascal PX Implementation Notes* describe the structure of the execution profiler *pxp*.

1.6. References

UNIX Documents

Communicating With UNIX
Computer Center
University of California, Berkeley
January, 1978.

Edit: a tutorial
Ricki Blau and James Joyce
Computing Services Division, Computing Affairs
University of California, Berkeley
January, 1978.

Ex/edit Command Summary
Computer Center
University of California, Berkeley
August, 1978.

Ex Reference Manual — Version 3.5
An Introduction to Display Editing with Vi
Vi Quick Reference
William Joy
Computer Science Division

Department of Electrical Engineering and Computer Science
University of California, Berkeley
October, 1980.

An Introduction to the C shell (Revised)
William Joy
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
October, 1980.

Brian W. Kernighan
UNIX for Beginners — Second Edition
Bell Laboratories
Murray Hill, New Jersey.

Brian W. Kernighan
A Tutorial Introduction to the UNIX Text Editor
Bell Laboratories
Murray Hill, New Jersey.

Dennis M. Ritchie and Ken Thompson
The UNIX Time Sharing System
Communications of the ACM
July 1974
365-378.

B. W. Kernighan and M. D. McIlroy
UNIX Programmer's Manual — Seventh Edition
Bell Laboratories
Murray Hill, New Jersey
December, 1978.
(Virtual VAX/11 Version,
U. C. Berkeley
Berkeley, Ca.
November, 1980.)

Pascal Language Documents

Conway, Gries and Zimmerman
A Primer on PASCAL
Winthrop, Cambridge Mass.
1976, 433 pp.

Kathleen Jensen and Niklaus Wirth
Pascal — User Manual and Report
Springer-Verlag, New York.
1975, 167 pp.

C. A. G. Webster
Introduction to Pascal
Heyden and Son, New York
1976, 129pp.

Niklaus Wirth
Algorithms + Data structures = Programs
Prentice-Hall, New York.
1976, 366 pp.

Niklaus Wirth
Systematic Programming
Prentice-Hall, New York.
1973, 169 pp.

Berkeley Pascal documents

The following documents are available from the Computer Center Library at the University of California, Berkeley.

William N. Joy, Susan L. Graham, and Charles B. Haley
Berkeley Pascal User's Manual — Version 2.0
October 1980.

William N. Joy
Berkeley Pascal PX Implementation Notes
Version 1.1, April 1979.
(Vax-11 Version 2.0 By Kirk McKusick, December, 1979)

William N. Joy
Berkeley Pascal PXP Implementation Notes
Version 1.1, April 1979.

2. Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the text editor *ex* (1). Users of the text editor *ed* should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because it allows us to make clearer examples.† The new UNIX† user will find it helpful to read one of the text editor documents described in section 1.4 before continuing with this section.

2.1. A first program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the documents *Communicating with UNIX* and *UNIX for Beginners*.

Once we are logged in we need to choose a name for our program; let us call it 'first' as this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence '.p' so we will call our file 'first.p'.

A sample editing session to create this file would begin:

```
% ex first.p
"first.p" [New file]
:
```

We didn't expect the file to exist, so the error diagnostic doesn't bother us. The editor now knows the name of the file we are creating. The ':' prompt indicates that it is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
program first(output)
begin
    writeln('Hello, world!')
end.
.
:
```

The line containing the single '.' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As the editor operates in a temporary work space we must now store the contents of this work space in the file 'first.p' so we can use the Pascal translator and executor *pix* on it.

```
:write
"first.p" [New file] 4 lines, 59 characters
:quit
%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

---

† Users with CRT terminals should find the editor *vi* more pleasant to use; we do not show its use here because its display oriented nature makes it difficult to illustrate.

†UNIX is a Trademark of Bell Laboratories.

‡ Our examples here assume you are using *csh*.

We are ready to try to translate and execute our program.

```
% pix first.p
Tue Oct 14 21:37 1980  first.p:
    2  begin
e  ----↑--- Inserted ';'
Execution begins...
Hello, world!
Execution terminated.

1 statements executed in 0.02 seconds cpu time.
%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we will see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

```
% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:quit
% pi first.p
%
```

---

†The standard Pascal warnings occur only when the associated s translator option is enabled. The s option is discussed in sections 5.1 and A.6 below. Warning diagnostics are discussed at the end of section 3.2, the associated w option is described in section 5.2.

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

If we now use the UNIX *ls* list files command we can see what files we have:

```
% ls
first.p
obj
%
```

The file 'obj' here contains the Pascal interpreter code. We can execute this by typing:

```
% px obj
Hello, world!

1 statements executed in 0.02 seconds cpu time.
%
```

Alternatively, the command:

```
% obj
```

will have the same effect. Some examples of different ways to execute the program follow.

```
% px
Hello, world!

1 statements executed in 0.02 seconds cpu time.
% pi -p first.p
% px obj
Hello, world!
% pix -p first.p
Hello, world!
%
```

Note that *px* will assume that 'obj' is the file we wish to execute if we don't tell it otherwise. The last two translations use the **-p** no-post-mortem option to eliminate execution statistics and 'Execution begins' and 'Execution terminated' messages. See section 5.2 for more details. If we now look at the files in our directory we will see:

```
% ls
first.p
obj
%
```

We can give our object program a name other than 'obj' by using the move command *mv* (1). Thus to name our program 'hello':

```
% mv obj hello
% hello
Hello, world!
% ls
first.p
hello
%
```

Finally we can get rid of the Pascal object code by using the *rm* (1) remove file command, e.g.:

```
% rm hello
% ls
first.p
%
```

For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

2.2. A larger program

Suppose that we have used the editor to put a larger program in the file 'bigger.p'. We can list this program with line numbers by using the program *cat*-n i.e.:

```
% cat −n bigger.p
%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual*. A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
% pix bigger.p
Tue Oct 14 21:37 1980  bigger.p:
    9        h = 34;       (* Character position of x−axis *)
w ----------------------------↑------ (* in a (* ... *) comment
   16        for i := 0 to lim begin
e ------------------------------------↑------ Inserted keyword do
   18                y := exp(−x9 * sin(i * x);
E ------------------------------------↑------ Undefined variable
e ------------------------------------------------------↑------ Inserted ')'
   19                n := Round(s * y) + h;
E -------------------------↑------ Undefined function
E ---------------------------------------------↑------ Undefined variable
   23                writeln('*')
e --------------------↑------ Inserted ';'
   24  end.
E -----↑------ Expected keyword until
E ----------↑------ Unexpected end−of−file − QUIT
Execution suppressed due to compilation errors
%
```

Since there were fatal 'E' errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

```
% pi −l bigger.p
```

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

```
% pi −l bigger.p | lpr
```

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

2.3. Correcting the first errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was 'malformed'. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in 'standard' Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing '*)' of the comment on line 8. We can begin an editor session to correct this problem by doing:

```
% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ *)
        s = 32;        (* 32 character width for interval [x, x+1] *)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
        for i := 0 to lim begin
:s/begin/do &
        for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper and lower case.† On UNIX lower-case is preferred‡, and all keywords and built-in

†In "standard" Pascal no distinction is made based on case.

‡One good reason for using lower-case is that it is easier to type.

procedure and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
            y := exp(-x) * sin(i * x);
:+s/Round/round
            n := round(s * y) + h;
:/write
                  write(' ');
:/
            writeln('*')
:insert
            until n = 0;
.
:$
end.
:insert
      end
.
:
```

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to *sin*, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

```
:?i ?s//c /
            y := exp(-x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:quit
% pi bigger.p
%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced.

Thus these warning diagnostics may not appear in a program with bad syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
% cat -n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;   (* 1/16, 16 lines for interval [x, x+1] *)
     8          s = 32;       (* 32 character width for interval [x, x+1] *)
     9          h = 34;       (* Character position of x-axis *)
    10          c = 6.28138;  (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim do begin
    17                  x := d / i;
    18                  y := exp(-x) * sin(c * x);
    19                  n := round(s * y) + h;
    20                  repeat
    21                          write(' ');
    22                          n := n - 1
    23                  until n = 0;
    24                  writeln('*')
    25          end
    26  end.
%
```

2.4. Executing the second example

We are now ready to execute the second example. The following output was produced by our first run.

```
% px
Execution begins...


Floating point division error

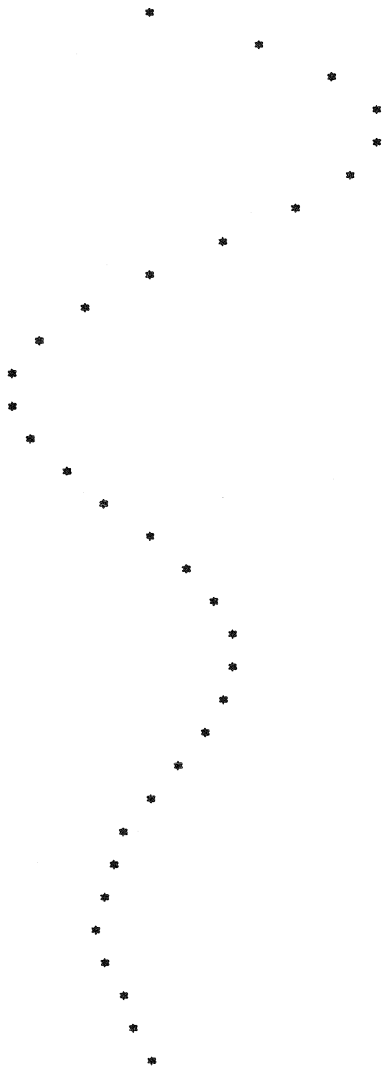
     Error in "graph1"+2 near line 17.
Execution terminated abnormally.

2 statements executed in 0.05 seconds cpu time.
%
```

Here the interpreter is presenting us with a runtime error diagnostic. It detected a 'division by zero' at line 17. Examining line 17, we see that we have written the statement 'x := d / i' instead of 'x := d * i'. We can correct this and rerun the program:

```
% ex bigger.p
```

```
"bigger.p" 26 lines, 538 characters
:17
        x := d / i
:s'/'*
        x := d * i
:write
"bigger.p" 26 lines, 538 characters
:q
% pix bigger.p
Execution begins...
```



```
Execution terminated.

2550 statements executed in 0.30 seconds cpu time.
%
```

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
% px > graph
```

We can use *cat* (1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file, e.g.

```
% px | lpr
Execution begins...
Execution terminated.

2550 statements executed in 0.37 seconds cpu time.
%
```

Note here that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '|&' to the shell rather than '|', i.e.:

```
% px |& lpr
%
```

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

```
% pi -p bigger.p
% px | lpr
%
```

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

2.5. Formatting the program listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. See also section 5.2 for details on the **n** and **i** options of *pi*.

2.6. Execution profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program which was abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a reexamination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source level optimization.

An example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **z** option to *pix*. This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times each statement in the program was executed.† When execution of the program completes,

†The counts are completely accurate only in the absence of runtime errors and nonlocal **goto** statements. This is not generally a problem, however, as in structured programs nonlocal **goto** statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* described below to get a count passes a suspended call point.

either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

```
% pix -l -z primes.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  primes.p

 1  program primes(output);
 2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
 3  var i,k,x,inc,lim,square,l: integer;
 4      prim: boolean;
 5      p,v: array[1..n1] of integer;
 6  begin
 7      write(2:6, 3:6); l := 2;
 8      x := 1; inc := 4; lim := 1; square := 9;
 9      for i := 3 to n do
10      begin (*find next prime*)
11          repeat x := x + inc; inc := 6-inc;
12              if square <= x then
13                  begin lim := lim+1;
14                      v[lim] := square; square := sqr(p[lim+1])
15                  end ;
16              k := 2; prim := true;
17              while prim and (k<lim) do
18              begin k := k+1;
19                  if v[k] < x then v[k] := v[k] + 2*p[k];
20                  prim := x <> v[k]
21              end
22          until prim;
23          if i <= n1 then p[i] := x;
24          write(x:6); l := l+1;
25          if l = 10 then
26              begin writeln; l := 0
27              end
28      end ;
29      writeln;
30  end .
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated.

1404 statements executed in 0.17 seconds cpu time.
%
```

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc* (1). See *prof* (1) for a discussion of the C compiler profiling facilities.

Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to *version stamp* the input program. *Pxp* also indicates the time at which the profile data was gathered.

```
% pxp -z primes.p
Berkeley Pascal PXP --- Version 1.1 (May 7, 1979)

Tue Oct 14 21:38 1980  primes.p

Profiled Tue Oct 21 18:48 1980

     1          1.-----program primes(output):
     2                |const
     2                |   n = 50;
     2                |   n1 = 7; (*n1 = sqrt(n)*)
     3                |var
     3                |   i, k, x, inc, lim, square, l: integer;
     4                |   prim: boolean;
     5                |   p, v: array [1..n1] of integer;
     6                |begin
     7                |   write(2: 6, 3: 6);
     7                |   l := 2;
     8                |   x := 1;
     8                |   inc := 4;
     8                |   lim := 1;
     8                |   square := 9;
     9                |   for i := 3 to n do begin (*find next prime*)
     9             48.-----|   repeat
    11                  76.-----|   x := x + inc;
    11                          |   inc := 6 - inc;
    12                          |   if square <= x then begin
    13                       5.-----|   lim := lim + 1;
    14                          |   v[lim] := square;
    14                          |   square := sqr(p[lim + 1])
    14                       |   end;
    16                       |   k := 2;
    16                       |   prim := true;
    17                       |   while prim and (k < lim) do begin
    18                    157.-----|   k := k + 1;
    19                          |   if v[k] < x then
    19                      42.-----|   v[k] := v[k] + 2 * p[k];
    20                          |   prim := x <> v[k]
    20                       |   end
    20                       |until prim;
    23                |   if i <= n1 then
    23                5.-----|   p[i] := x;
    24                |   write(x: 6);
    24                |   l := l + 1;
    25                |   if l = 10 then begin
    26                5.-----|   writeln;
    26                          |   l := 0
    26                |   end
```

```
      26            |   end;
      29            |   writeln
      29            end.
%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not labelled, we look up in the listing to find the first '|' which directly above the original one which has a count and that is the answer. Thus, in our example, *k* was incremented 157 times on line 18, while the *write* procedure call on line 24 was executed 48 times as given by the count on the **repeat**.

More information on *pxp* can be found in its manual section *pxp* (1) and in sections 5.4, 5.5 and 5.10.

3. Error diagnostics

This section of the *User's Manual* discusses the error diagnostics of the programs *pi*, *pc* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and section 5.2 below for more details. All the diagnostics given by *pi* will also be given by *pc*.

3.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

Illegal characters

Characters such as '$', '!', and '@' are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be 'illegal characters'. This can happen if you leave off an opening string quote "'". Note that the character '"', although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character '?' so that they will show in your listing.

String errors

There is no character string of length 0 in Pascal. Consequently the input "''" is not acceptable. Similarly, encountering an end-of-line after an opening string quote "'" without encountering the matching closing quote yields the diagnostic "Unmatched ' for string". It is permissible to use the character '#' instead of "'" to delimit character and constant strings for portability reasons. For this reason, a spuriously placed '#' sometimes causes the diagnostic about unbalanced quotes. Similarly, a '#' in column one is used when preparing programs which are to be kept in multiple files. See section 5.11 for details.

Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments — those delimited by '{' and '}', and those delimited by '(*' and '*)'. Thus consider:

```
{ This is a comment enclosing a piece of program
a := functioncall;      (* comment within comment *)
procedurecall;
lhs := rhs;             (* another comment *)
}
```

By using one kind of comment exclusively in your program you can use the other delimiters when you need to "comment out" parts of your program†. In this way you will also allow the translator to help by detecting statements accidentally placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of "QUIT" below.

---

†If you wish to transport your program, especially to the 6000-3.4 implementation, you should use the character sequence '(*' to delimit comments. For transportation over the *rcslink* to Pascal 6000-3.4, the character '#' should be used to delimit characters and constant strings.

Digits in numbers

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

```
Tue Oct 14 21:37 1980  digits.p:
    4  r := 0.;
e  ------------↑------ Digits required after decimal point
    5  r := .0;
e  ----------↑------ Digits required before decimal point
    6  r := 1.e10;
e  ------------↑------ Digits required after decimal point
    7  r := .05e-10;
e  ----------↑------ Digits required before decimal point
```

These same constructs are also illegal as input to the Pascal interpreter *px*.

Replacements, insertions, and deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
% pix -l synerr.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 21 23:51 1980  synerr.p

    1  program syn(output);
    2  var i, j are integer;
e  ------------↑--- Replaced identifier with a ':'
    3  begin
    4          for j := 1 to 20 begin
e  -------------------↑--- Replaced '*' with a '='
e  -----------------------------↑--- Inserted keyword do
    5                  write(j);
    6                  i = 2 ** j;
e  ---------------------↑--- Inserted ':'
E  -------------------------↑--- Inserted identifier
    7                  writeln(i))
E  -------------------------------↑--- Deleted ')'
    8          end
    9  end.
%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that, if you assume that the language has a feature which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be summarized at the end of the containing **procedure** or **function** or at the end of the **program** if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, e.g. if a **type** identifier is used in an assignment statement, or if a simple variable is used where a **record** variable is required, a diagnostic will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print a diagnostic which indicates that the given symbol was 'Expected'.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates that the input is "malformed". If necessary, the translator may then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
% pix -l synerr2.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  synerr2.p

    1  program synerr2(input,outpu);
    2  integer a(10)
E   ----↑------ Malformed declaration
    3  begin
    4          read(b);
E   ------------------↑------ Undefined variable
    5          for c := 1 to 10 do
E   ----------------↑------ Undefined variable
    6                  a(c) := b * c;
E   --------------------↑------ Undefined procedure
E   ------------------------↑------ Malformed statement
    7  end.
E 1 - File outpu listed in program statement but not declared
e 1 - The file output must appear in the program statement file list
In program synerr2:
  E - a undefined on line 6
  E - b undefined on line 4
  E - c undefined on lines 5 6
Execution suppressed due to compilation errors
%
```

Here we misspelled *output* and gave a FORTRAN style variable declaration which the translator diagnosed as a 'Malformed declaration'. When, on line 6, we used '(' and ')' for subscripting (as in FORTRAN) rather than the '[' and ']' which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

Expected and unexpected end-of-file, "QUIT"

If the translator finds a complete program, but there is more non-comment text in the input file, then it will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **end**s are present in the input. The message may appear after the recovery says that it "Expected '.'" since '.' is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by "QUIT". One example of this was given above — a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
% pix -l mism.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  mism.p

    1  program mismatch(output)
    2  begin
e  ----↑------ Inserted ';'
    3          writeln('***');
    4          { The next line is the last line in the file }
    5          writeln
E  --------------------↑------ Unexpected end-of-file - QUIT
%
```

3.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line in the text of the program as well as the text of the error message. While this number is most often the line where the error occurred, it is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the **end** or **until**. to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary to some users. This is the only notable exception, however, to reasonable associations.

Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called 'type clashes' by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type 'classes':

array	Boolean	char	file	integer
pointer	real	record	scalar	string

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

```
Tue Oct 14 21:37 1980  clash.p:
E 7 – Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two line error message. If the same error occurs more than once, the same explanatory diagnostic will be given each time.

Scalar

The only class whose meaning is not self-explanatory is 'scalar'. Scalar has a precise meaning in the Jensen-Wirth *User Manual* where, in fact, it refers to *char*, *integer*, *real*, and *Boolean* types as well as the enumerated types. For the purposes of the Pascal translator, scalar in an error message refers to a user-defined, enumerated type, such as *ops* in the example above or *color* in

```
type color = (red, green, blue)
```

For integers, the more explicit denotation *integer* is used. Although it would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.

Function and procedure type errors

For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message similar to:

```
Tue Oct 14 21:38 1980  sin1.p:
E 12 – sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
Tue Oct 14 21:38 1980  sin2.p:
E 12 – sin's argument must be integer or real, not char
```

is produced. A few functions and procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.

Can't read and write scalars, etc.

The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that if you define

```
type color = (red, green, blue)
```

"standard" Pascal does not associate these constants with the strings 'red', 'green', and 'blue' in any way. An extension has been added which allows enumerated types to be read and written, however if the program is to be portable, you will have to write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real numbers from text files. You cannot read strings or Booleans. It is possible to make a

```
file of color
```

but the representation is binary rather than string.

Expression diagnostics

The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

```
% pi -l expr.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:37 1980  expr.p

    1  program x(output);
    2  var
    3          a: set of char;
    4          b: Boolean;
    5          c: (red, green, blue);
    6          p: ↑ integer;
    7          A: alfa;
    8          B: packed array [1..5] of char;
    9  begin
   10          b := true;
   11          c := red;
   12          new(p);
   13          a := [];
   14          A := 'Hello, yellow';
   15          b := a and b;
   16          a := a * 3;
   17          if input < 2 then writeln('boo');
   18          if p <= 2 then writeln('sure nuff');
   19          if A = B then writeln('same');
   20          if c = true then writeln('hue''s and color''s')
   21  end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
In program x:
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
%
```

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

Type equivalence

Several diagnostics produced by the Pascal translator complain about 'non-equivalent types'. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same type identifier. Thus, the variables *x* and *y* declared as

```
var
   x: ↑ integer;
   y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

```
Tue Oct 14 21:38 1980  typequ.p:
E 7 – Type clash: non–identical pointer types
  ... Type of expression clashed with type of variable in assignment
```

Thus it is always necessary to declare a type such as

```
type intptr = ↑ integer;
```

and use it to declare

```
var x: intptr; y: intptr;
```

Note that if we had initially declared

```
var x, y: ↑ integer;
```

then the assignment statement would have worked. The statement

```
x↑ := y↑
```

is allowed in either case. Since the parameter to a **procedure** or **function** must be declared with a type identifier rather than a constructed type, it is always necessary, in practice, to declare any type which will be used in this way.

Unreachable statements

Berkeley Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement:

```
if false then
  writeln('impossible!')
```

Goto's into structured statements

The translator detects and complains about **goto** statements which transfer control into structured statements (**for**, **while**, etc.) It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. Such checks are made only within the body of a single procedure or function.

Unused variables, never set variables

Although *pi* always clears variables to 0 at **procedure** and **function** entry, *pc* does not unless runtime checking is enabled using the C option. It is **not** good programming practice to rely on this initialization. To discourage this practice, and to help detect errors in program logic, *pi* flags as a 'w' warning error:

1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.

In fact, these diagnostics are applied to all declared items. Thus a **const** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see sections 5.1 and 5.2.

3.3. Translator panics, i/o errors

Panics

One class of error which rarely occurs, but which causes termination of all processing when it does is a panic. A panic indicates a translator-detected internal inconsistency. A typical panic message is:

```
snark (rvalue) line=110 yyline=109
Snark in pi
```

If you receive such a message, the translation will be quickly and perhaps ungracefully terminated. You should contact a teaching assistant or a member of the system staff, after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. A small number of panics are possible in *px*. All panics should be reported to a teaching assistant or systems staff so that they can be fixed.

Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up the full available process space of 64000 bytes on the PDP-11. On the VAX-11, table sizes are extremely generous and very large (25000) line programs have been easily accommodated. For the PDP-11, it is generally true that the size of the largest translatable program is directly related to **procedure** and **function** size. A number of non-trivial Pascal programs, including some with more than 2000 lines and 2500 statements have been translated and interpreted using Berkeley Pascal on PDP-11's. Notable among these are the Pascal-S interpreter, a large set of programs for automated generation of code generators, and a general context-free parsing program which has been used to parse sentences with a grammar for a superset of English. In general, very large programs should be translated using *pc* and the separate compilation facility.

If you receive an out of space message from the translator during translation of a large **procedure** or **function** or one containing a large number of string constants you may yet be able to translate your program if you break this one **procedure** or **function** into several routines.

I/O errors

Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.

3.4. Run-time errors

We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (1).

Start-up errors

These errors occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or being inaccessible to the user.

Program execution errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite loop, or running out of

memory‡. The interpreter will produce a backtrace after the error occurs, showing all the active routine calls, unless the **p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.*

As an example of such an error, assume that we have accidentally declared the constant *nl* to be 6, instead of 7 on line 2 of the program primes as given in section 2.6 above. If we run this program we get the following response.

```
% pix primes.p
Execution begins...
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167

Subscript out of range

        Error in "primes"+8 near line 14.
Execution terminated abnormally.

941 statements executed in 0.50 seconds cpu time.
%
```

Here the interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program primes.

Interrupts

If the program is interrupted while executing and the **p** option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the program was translated with the **z** option enabled to *pi* or *pix*.

I/O interaction errors

The final class of interpreter errors results from inappropriate interactions with files, including the user's terminal. Included here are bad formats for integer and real numbers (such as no digits after the decimal point) when reading.

---

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image when it runs out of memory. This situation occurs very rarely.

* On the VAX-11, each variable is restricted to allocate at most 65000 bytes of storage (this is a PDP-11ism that has survived to the VAX.)

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a **procedure** or **function** entry or exit. In this case, the backtrace will only contain the current line. A reverse call order list of procedures will not be given.

4. Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

4.1. Introduction

Our first sample programs, in section 2, used the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the following Pascal program which is similar to the program *cat* (1).

```
% pix -l kat.p <primes
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:38 1980  kat.p

    1  program kat(input, output);
    2  var
    3      ch: char;
    4  begin
    5      while not eof do begin
    6          while not eoln do begin
    7              read(ch);
    8              write(ch)
    9          end;
   10          readln;
   11          writeln
   12      end
   13  end { kat }.
Execution begins...
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229

Execution terminated.

925 statements executed in 0.15 seconds cpu time.
%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our prime number program of section 2.6. It is also possible to 'pipe' input to this program much as we piped input to the line printer daemon *lpr* (1) before. Thus, the same output as above would be produced by

```
% cat primes | pix -l kat.p
```

All of these examples use the shell to control the input and output from files. One very simple way to associate Pascal files with named UNIX† files is to place the file name in the **program** statement. For example, suppose we have previously created the file *data*. We then use it as input to another version of a listing program.

---

†UNIX is a Trademark of Bell Laboratories.

```
% cat data
line one.
line two.
line three is the end.
% pix -l copydata.p
Berkeley Pascal PI -- Version 2.0 (Sat Oct 18 21:01:54 1980)

Tue Oct 14 21:37 1980  copydata.p

    1  program copydata(data, output);
    2  var
    3      ch: char;
    4      data: text;
    5  begin
    6      reset(data);
    7      while not eof(data) do begin
    8          while not eoln(data) do begin
    9              read(data, ch);
   10              write(ch)
   11          end;
   12          readln(data);
   13          writeln
   14      end
   15  end { copydata }.
Execution begins...
line one.
line two.
line three is the end.
Execution terminated.

134 statements executed in 0.08 seconds cpu time.
%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we 'reset(data)', the Pascal system opens our file 'data' for reading. More sophisticated, but less portable, examples of using UNIX files will be given in sections 4.5 and 4.6. There is a portability problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

4.2. Eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of *eof* and *eoln*. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is "used up". In fact, there is no way the system can know whether the input is at the end-of-file or the end-of-line unless it attempts to read a line from it. If the input is from a previously created file, then this reading can take place without run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were a terminal, the user would have to type some input before execution could begin. This would make it impossible

†It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe*, the output of a program which is reading from the terminal.

for the program to begin by prompting for input or printing a herald.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file or end-of-line conditions are true. Thus, internally, these functions can have three values — true, false, and "I don't know yet; if you ask me I'll have to find out". All files remain in this last, indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, e.g. in a call to *read*. The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it will be necessary for it to attempt to read from the input.

Thus consider the following example code

```
while not eof do begin
    write('number, please? ');
    read(i);
    writeln('that was a ', i: 2)
end
```

At first glance, this may be appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait for the user to type a line. By producing the desired prompting before testing *eof*, the following code avoids this problem:

```
write('number, please ?');
while not eof do begin
    read(i);
    writeln('that was a ', i:2);
    write('number, please ?')
end
```

The user must still type a line before the **while** test is completed, but the prompt will ask for it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

4.3. More about eoln

To have a good understanding of when *eoln* will be true it is necessary to know that in any file there is a special character indicating end-of-line, and that, in effect, the Pascal system always reads one character ahead of the Pascal *read* commands.† For instance, in response to 'read(ch)', the system sets *ch* to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with

†In Pascal terms, 'read(ch)' corresponds to 'ch := input^; get(input)'

a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this will almost surely have the effect of ignoring the last character in the line. The 'read(ch)' belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a *readln* call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

4.4. Output buffering

A final point about Pascal input-output must be noted here. This concerns the buffering of the file *output*. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program such as the line printer daemon *lpr* (1). To gain efficiency, the Pascal system "buffers" the output characters (i.e. it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction.) However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) The procedure *message* or *flush* is called.

Thus, in the code sequence

```
for i := 1 to 5 do begin
    write(i: 2);
    Compute a lot with no output
end;
writeln
```

the output integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the **b** option to 0 before the **program** statement by inserting a comment of the form

```
(*$b0*)
```

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in section 5.

4.5. Files, reset, and rewrite

It is possible to use extended forms of the built-in functions *reset* and *rewrite* to get more general associations of UNIX file names with Pascal file variables. When a file other than *input* or *output* is to be read or written, then the reading or writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. As we saw in section 2.9, by mentioning the file in the **program** statement, we could cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the **program** statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *removed* by the Pascal system as soon as they become inaccessible. They are not removed, however, if a run-time error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in section 3.1 by doing:

```
reset(data, 'primes')
```

instead of a simple

```
reset(data)
```

In this case it is not essential to mention 'data' in the program statement, but it is still a good idea because is serves as an aid to program documentation. The second parameter to *reset* and *rewrite* may be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at run time. Full details on file name/file variable associations are given in section A.3.

4.6. Argc and argv

Each UNIX process receives a variable length sequence of arguments each of which is a variable length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*−1, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
% obj /etc/motd /usr/dict/words hello
```

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be 'obj', the first '/etc/motd', etc.

Pascal does not provide variable size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure and has the syntax

```
argv(i, a)
```

where *i* is an integer and *a* is a string variable. This procedure call assigns the (possibly truncated or blank padded) *i*'th argument of the current process to the string variable *a*. The file manipulation routines *reset* and *rewrite* will strip trailing blanks from their optional second

arguments so that this blank padding is not a problem in the usual case where the arguments are file names.

We are now ready to give a Berkeley Pascal program 'kat', based on that given in section 3.1 above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
% cat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
        end
    until i >= argc
end { kat }.
%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed on other systems. As this program deals mostly with *argc* and *argv* and UNIX system dependent considerations, portability is of little concern.

If this program is in the file 'kat.p', then we can do

```
% pi kat.p
% mv obj kat
% kat primes
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229


930 statements executed in 0.18 seconds cpu time.
% kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)

287 statements executed in 0.03 seconds cpu time.
%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat*, copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
% kat < primes
```

now equivalent to

```
% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
% kat xxxxqqq


Could not open xxxxqqq: No such file or directory

    Error in "kat"+5 near line 11.

4 statements executed in 0.02 seconds cpu time.
%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
% pi -pb kat.p
% mv obj kat
```

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the full traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the **t** option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
% kat xxxxqqq


Could not open xxxxqqq: No such file or directory

    Error in "kat"
% kat primes
  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229

%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

5. Details on the components of the system

5.1. Options

The programs *pi*, *pc*, and *pxp* take a number of options.† There is a standard UNIX† convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted of the character '−' followed by a single character option name.

Except for the **b** option which takes a single digit value, each option may be set on (enabled) or off (disabled.) When an on/off valued option appears on the command line of *pi* or it inverts the default setting of that option. Thus

```
% pi −l foo.p
```

enables the listing option **l**, since it defaults off, while

```
% pi −t foo.p
```

disables the run time tests option **t**, since it defaults on.

In additon to inverting the default settings of *pi* options on the command line, it is also possible to control the *pi* options within the body of the program by using comments of a special form illustrated by

```
{$l−}
```

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters‡. The most basic actions for options are to set them, thus

```
{$l+ Enable listing}
```

or to clear them

```
{$t−,p− No run−time tests, no post mortem analysis}
```

Notice that '+' always enables an option and '−' always disables it, no matter what the default is. Thus '−' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

5.2. Options common to Pi, Pc, and Pix

The following options are common to both the compiler and the interpreter. With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

---

†As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

†UNIX is a Trademark of Bell Laboratories.

‡This format was chosen because it is used by Pascal 6000-3.4. In general the options common to both implementations are controlled in the same way so that comment control in options is mostly portable. It is recommended, however, that only one control be put per comment for maximum portability, as the Pascal 6000-3.4 implementation will ignore controls after the first one which it does not recognize.

Buffering of the file output — b

The **b** option controls the buffering of the file *output*. The default is line buffering, with flushing at each reference to the file *input* and under certain other circumstances detailed in section 5 below. Mentioning **b** on the command line, e.g.

```
% pi -b assembler.p
```

causes standard output to be block buffered, where a block is some system—defined number of characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, e.g.

```
{$b0}
```

causes the file *output* to be unbuffered. Any value 2 or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **b** must precede the **program** statement.

Include file listing — i

The **i** option takes the name of an **include** file, **procedure** or **function** name and causes it to be listed while translating†. Typical uses would be

```
% pix -i scanner.i compiler.p
```

to make a listing of the routines in the file scanner.i, and

```
% pix -i scanner compiler.p
```

to make a listing of only the routine *scanner*. This option is especially useful for conservation-minded programmers making partial program listings.

Make a listing — l

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The **l** option is pushed and popped by the **i** option at appropriate points in the program.

Standard Pascal only — s

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as 's' warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedures** and **functions**, extensions to the **procedure** *write*, and the padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters so that the case of keywords and identifiers is effectively ignored. The **s** option is most useful when a program is to be transported, thus

```
% pi -s isitstd.p
```

will produce warnings unless the program meets the standard.

Runtime tests — t and C

These options control the generation of tests that subrange variable values are within bounds at run time. *pi* defaults to generating tests and uses the option **t** to disable them. *pc* defaults to not generating tests, and uses the option **C** to enable them. Disabling runtime tests also causes **assert** statements to be treated as comments.‡

---

†**Include** files are discussed in section 5.9.

‡See section A.1 for a description of **assert** statements.

Suppress warning diagnostics − w

The w option, which defaults on, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

```
{Sw−}
```

or on the command line

```
% pi −w tryme.p
```

suppresses these usually useful diagnostics.

Generate counters for a pxp execution profile − z

The z option, which defaults off, enables the production of execution profiles. By specifying z on the command line, i.e.

```
% pi −z foo.p
```

or by enabling it in a comment before the **program** statement causes *pi* and *pc* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in section 2.6; its options are described in section 5.6. Note that the z option cannot be used on separately compiled programs.

5.3. Options available in Pi

Post-mortem dump − p

The p option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. It also cause *px* to count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying p on the command line disables these checks and the ability to give this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the p option in comments. To prevent the post-mortem backtrace on error, p must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

```
% pi −pbt pxref.p
```

5.4. Options available in Px

The first argument to *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc* and *argv* as described in section 4.6.

Px may also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

```
% obj primes
```

will be converted to read

```
% px obj primes
```

5.5. Options available in Pc

Generate assembly language − S

The program is compiled and the assembly language output is left in file appended .s. Thus

```
% pc −S foo.p
```

creates a file *foo.s*. No executable file is created.

Symbolic Debugger Information — g

The **g** option causes the compiler to generate information needed by *sdb*(1) the symbolic debugger. For a complete description of *sdb* see Volume 2c of the UNIX Reference Manual.

Redirect the output file — o

The *name* argument after the **-o** is used as the name of the output file instead of *a.out*. Its typical use is to name the compiled program using the root of the file name. Thus:

```
% pc -o myprog myprog.p
```

causes the compiled program to be called *myprog*.

Generate counters for a *prof* execution profile — p

The compiler produces code which counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system rather than by inline counters used by *pxp*. This results in less degradation in execution, at somewhat of a loss in accuracy. See *prof*(1) for a more complete description.

Run the object code optimizer — O

The output of the compiler is run through the object code optimizer. This provides an increase in compile time in exchange for a decrease in compiled code size and execution time.

5.6. Options available in Pxp

Pxp takes, on its command line, a list of options followed by the program file name, which must end in '.p' as it must for *pi*, *pc*, and *pix*. *Pxp* will produce an execution profile if any of the **z**, **t** or **c** options is specified on the command line. If none of these options is specified, then *pxp* functions as a program reformatter.

It is important to note that only the **z** and **w** options of *pxp*, which are common to *pi*, *pc*, and *pxp* can be controlled in comments. All other options must be specified on the command line to have any effect.

The following options are relevant to profiling with *pxp*.

Include the bodies of all routines in the profile — a

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

Suppress declaration parts from a profile — d

Normally a profile includes declaration parts. Specifying **d** on the command line suppresses declaration parts.

Eliminate include directives — e

Normally, *pxp* preserves **include** directives to the output when reformatting a program, as though they were comments. Specifying **-e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate **include** directives, e.g. before transporting a program.

Fully parenthesize expressions — f

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

```
d := a + b mod c / e
```

with minimal parenthesization, the default, will print as

```
d := a + ((b mod c) / e)
```

with the f option specified on the command line.

Left justify all procedures and functions – j

Normally, each **procedure** and **function** body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

Print a table summarizing procedure and function calls – t

The t option causes *pxp* to print a table summarizing the number of calls to each **procedure** and **function** in the program. It may be specified in combination with the z option, or separately.

Enable and control the profile – z

The z profile option is very similar to the i listing control option of *pi*. If z is specified on the command line, then all arguments up to the source file argument which ends in '.p' are taken to be the names of **procedures** and **functions** or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

```
% pxp -z test parser.i compiler.p
```

This specifies that profiles of the routines in the file *parser.i* and the routine *test* are to be made.

5.7. Formatting programs using pxp

The program *pxp* can be used to reformat programs, by using a command of the form

```
% pxp dirty.p > clean.p
```

Note that since the shell creates the output file 'clean.p' before *pxp* executes, so 'clean.p' and 'dirty.p' must not be the same file.

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

1) Left marginal comments, which begin in the first column of the input line and are placed in the first column of an output line.
2) Aligned comments, which are preceded by no input tokens on the input line. These are aligned in the output with the running program text.
3) Trailing comments, which are preceded in the input line by a token with no more than two spaces separating the token from the comment.
4) Right marginal comments, which are preceded in the input line by a token from which they are separated by at least three spaces or a tab. These are aligned down the right margin of the output, currently to the first tab stop after the 40th column from the current "left margin".

Consider the following program.

```
% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
```

```
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;        {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced.

```
% pxp comments.p
{ This is a left marginal comment. }

program hello(output);
var
   i: integer; {This is a trailing comment}
   j: integer;                                    {This is a right marginal comment}
   k: array [1..10] of array [1..10] of integer;  {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
   i := 1; {Trailing i comment}
{A left marginal comment}
   {An aligned comment}
   j := 1;                                        {Right marginal comment}
   k[1] := 1;
   writeln(i, j, k[1])
end.
%
```

The following formatting related options are currently available in *pxp*. The options f and j described in the previous section may also be of interest.

Strip comments −s

The s option causes *pxp* to remove all comments from the input text.

Underline keywords − _

A command line argument of the form −_ as in

```
% pxp -_ dirty.p
```

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

Specify indenting unit – [23456789]

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form $-d$ with d a digit, $2 \leqslant d \leqslant 9$ you can specify that d spaces are to be used per level instead.

5.8. Pxref

The cross-reference program *pxref* may be used to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file 'foo.p' one can execute the command:

```
% pxref foo.p
```

The cross-reference is, unfortunately, not block structured. Full details on *pxref* are given in its manual section *pxref* (1).

5.9. Multi-file programs

A text inclusion facility is available with Berkeley Pascal. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character '#' in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word 'include' followed by a filename enclosed in single '' or double "" quotation marks. The file name may be followed by a semicolon ';' if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '.i'. An example of the use of included files in a main program would be:

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"

begin
    { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translation and runtime diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. Nested includes are possible up to 10 deep.

See the descriptions of the **i** option of *pi* in section 5.2 above; this can be used to control listing when **include** files are present.

When a non-trivial line is encountered in the source text after an **include** finishes, the 'popped' filename is printed, in the same manner as above.

For the purposes of error diagnostics when not making a listing, the filename will be printed before each diagnostic if the current filename has changed since the last filename was printed.

5.10. Separate Compilation with Pc

A separate compilation facility is provided with the Berkeley Pascal compiler, *pc*. This facility allows programs to be divided into a number of files and the pieces to be compiled individually, to be linked together at some later time. This is especially useful for large programs, where small changes would otherwise require time-consuming re-compilation of the entire

program.

Normally, *pc* expects to be given entire Pascal programs. However, if given the −c option on the command line, it will accept a sequence of definitions and declarations, and compile them into a **.o** file, to be linked with a Pascal program at a later time. In order that procedures and functions be available across separately compiled files, they must be declared with the directive **external**. This directive is similar to the directive **forward** in that it must precede the resolution of the function or procedure, and formal parameters and function result types must be specified at the **external** declaration and may not be specified at the resolution.

Type checking is performed across separately compiled files. Since Pascal type defintions define unique types, any types which are shared between separately compiled files must be the same definition. This seemingly impossible problem is solved using a facility similar to the **include** facility discussed above. Definitions may be placed in files with the extension **.h** and the files included by separately compiled files. Each definition from a **.h** file defines a unique type, and all uses of a definition from the same **.h** file define the same type. Similarly, the facility is extended to allow the definition of **consts** and the declaration of **labels**, **vars**, and **external functions** and **procedures**. Thus **procedures** and **functions** which are used between separately compiled files must be declared **external**, and must be so declared in a **.h** file included by any file which calls or resolves the **function** or **procedure**. Conversely, **functions** and **procedures** declared **external** may only be so declared in **.h** files. These files may be included only at the outermost level, and thus define or declare global objects. Note that since only **external function** and **procedure** declarations (and not resolutions) are allowed in **.h** files, statically nested **functions** and **procedures** can not be declared **external**.

An example of the use of included **.h** files in a program would be:

```
program compiler(input, output, obj);

#include "globals.h"
#include "scanner.h"
#include "parser.h"
#include "semantics.h"

begin
    { main program }
end.
```

This might include in the main program the definitions and declarations of all the global **labels**, **consts**, **types** **vars** from the file globals.h, and the **external function** and **procedure** declarations for each of the separately compiled files for the scanner, parser and semantics. The header file *scanner.h* would contain declarations of the form:

```
type
    token = record
        { token fields }
    end;

function scan(var inputfile: text): token;
    external;
```

Then the scanner might be in a separately compiled file containing:

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
    { scanner code }
end;
```

which includes the same global definitions and declarations and resolves the scanner functions and procedures declared **external** in the file scanner.h.

A. Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available. It concludes with a list of differences with the commonly available Pascal 6000−3.4 implementation, and some comments on standard and portable Pascal.

A.1. Extensions to the language Pascal

This section defines non-standard language constructs available in Berkeley Pascal. The s standard Pascal option of the translators *pi* and *pc* can be used to detect these extensions in programs which are to be transported.

String padding

Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make them as long as is required. The following is a legal Berkeley Pascal program:

```
program x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
   z := 'red';
   writeln(z)
end;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red
```

which is standard Pascal.

Octal constants, octal and hexadecimal write

Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

```
write(a:n hex)
```

cause the internal representation of expression *a*, which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal respectively.

Assert statement

An **assert** statement causes a *Boolean* expression to be evaluated each time the statement is executed. A runtime error results if any of the expressions evaluates to be *false*. The **assert** statement is treated as a comment if run-time tests are disabled. The syntax for **assert** is:

```
assert <expr>
```

Enumerated type input-output

Enumerated types may be read and written. On output the string name associated with the enumerated value is output. If the value is out of range, a runtime error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table a runtime error occurs.

Structure returning functions

An extension has been added which allows functions to return arbitrary sized structures rather than just scalars as in the standard.

Separate compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level may be compiled separately. Type checking of calls to separately compiled routines is performed at load time to insure that the program as a whole is consistent. See section 5.10 for details.

A.2. Resolution of the undefined specifications

File name – file variable associations

Each Pascal file variable is associated with a named UNIX† file. Except for *input* and *output*, which are exceptions to some of the rules, a name can become associated with a file in any of three ways:

1) If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.

2) If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.

3) If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form 'tmp.x' is associated with the file. Temporary names start with 'tmp.1' and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

The program statement

The syntax of the **program** statement is:

program <id> (<file id> { , <file id > }) ;

The file identifiers (other than *input* and *output*) must be declared as variables of **file** type in the global declaration part.

The files input and output

The formal parameters *input* and *output* are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:

1) The program heading **must** contains the formal parameter *output*. If *input* is used, explicitly or implicitly, then it must also be declared here.

2) Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

var input, output: text

†UNIX is a Trademark of Bell Laboratories.

3) The procedure *reset* may be used on *input*. If no UNIX file name has ever been associated with *input*, and no file name is given, then an attempt will be made to 'rewind' *input*. If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

```
rewrite(output)
```

associates a temporary name with *output*.

Details for files

If a file other than *input* is to be read, then reading must be initiated by a call to the procedure *reset* which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

Buffering

The buffering for *output* is determined by the value of the **b** option at the end of the **program** statement. If it has its default value 1, then *output* is buffered in blocks of up to 512 characters, flushed whenever a writeln occurs and at each reference to the file *input*. If it has the value 0, *output* is unbuffered. Any value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are always buffered in blocks of 512 characters. All output buffers are flushed when the files are closed at scope exit, whenever the procedure *message* is called, and can be flushed using the built-in procedure *flush*.

An important point for an interactive implementation is the definition of 'input↑'. If *input* is a teletype, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt could be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, reading from a file occurring only when necessary.

The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols 'and', 'or', 'not', '<=', '>=', '<>', and the uparrow '↑' (for pointer qualification) are recognized.† Less portable are the synonyms tilde '~' for **not**, '&' for **and**, and '|' for **or**.

Upper and lower case are considered to be distinct. Keywords and built-in **procedure** and **function** names are composed of all lower case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword **goto**. The standard type 'boolean' is also available as 'Boolean'.

Character strings and constants may be delimited by the character "'" or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line — see *Multi-file programs* below.

The standard types

The standard type *integer* is conceptually defined as

```
type integer = minint .. maxint;
```

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

---

†On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes.
The proposed standard for Pascal considers them to be the same.

```
const maxint = 2147483647; minint = -2147483648;
```

The standard type *char* is conceptually defined as

```
type char = minchar .. maxchar;
```

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0, ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 17 digits of precision with numbers as small as 10 to the negative 38th power and as large as 10 to the 38th power.

Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

Option control

Options of the translators may be controlled in two distinct ways. A number of options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '-' and cause the default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

```
% pi -l -s foo.p
```

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. The format for option control in comments is identical to that used in Pascal 6000-3.4. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

```
{$l+,s+ listing on, standard Pascal}
```

as the first line of the program. The 'l' option is more appropriately specified on the command line, since it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

Directives consist of a letter designating the option, followed either by a '+' to turn the option on, or by a '-' to turn the option off. The b option takes a single digit instead of a '+' or '-'.

Notes on the listings

The first page of a listing includes a banner line indicating the version and date of generation of *pi* or *pc*. It also includes the UNIX path name supplied for the source file and the date of last modification of that file.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed character, control-l, which causes a page eject in the listing, and a line with no characters which causes the line number to be suppressed in the listing, creating a truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many assemblers. Non-printing characters are printed as the character '?' in the listing.†

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

The standard procedure write

If no minimum field length parameter is specified for a *write*, the following default values are assumed:

integer	10
real	22
Boolean	length of 'true' or 'false'
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be explicitly indicated by 'writeln(f)', where 'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

Files

Files cannot be members of files or members of dynamically allocated structures.

Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to −32768, and the upper bound less than 32768. In particular, strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements.

Line and symbol length

There is no intrinsic limit on the length of identifiers. Identifiers are considered to be distinct if they differ in any single position over their entire length. There is a limit, however, on the maximum input line length. This limit is quite generous however, currently exceeding 160 characters.

Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the hardware and thus, on the PDP-11, by the 16 bit address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

Additional predefined types

The type *alfa* is predefined as:

```
type alfa = packed array [ 1..10 ] of char
```

The type *intset* is predefined as:

```
type intset = set of 0..127
```

In most cases the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer†. In the latter case the type defaults to the current binding of *intset*, which must be "type set of (a subrange of) integer" at that point.

Note that if *intset* is redefined via:

```
type intset = set of 0..58;
```

then the default integer set is the implicit *intset* of Pascal 6000−3.4

Additional predefined operators

The relationals '<' and '>' of proper set inclusion are available. With *a* and *b* sets, note that

(not (*a* < *b*)) <> (*a* >= *b*)

As an example consider the sets *a* = [0,2] and *b* = [1]. The only relation true between these sets is '<>'.

Non-standard procedures

argv(i,a)	where *i* is an integer and *a* is a string variable assigns the (possibly truncated or blank padded) *i*'th argument of the invocation of the current UNIX process to the variable *a*. The range of valid *i* is *0* to *argc−1*.
date(a)	assigns the current date to the alfa variable *a* in the format 'dd mmm yy ', where 'mmm' is the first three characters of the month, i.e. 'Apr'.
flush(f)	writes the output buffered for Pascal file *f* into the associated UNIX file.
halt	terminates the execution of the program with a control flow back-trace.
linelimit(f,x)‡	with *f* a textfile and *x* an integer expression causes the program to be abnormally terminated if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in **procedure** *write*, to be written unbuffered on the diagnostic unit 2, almost always the user's terminal.

†The current translator makes a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16 bit arithmetic given above in this case.
‡Currently ignored by pdp-11 *px*.

null	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
remove(a)	where *a* is a string causes the UNIX file whose name is *a*, with trailing blanks eliminated, to be removed.
reset(f,a)	where *a* is a string causes the file whose name is *a* (with blanks trimmed) to be associated with *f* in addition to the normal function of *reset*.
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where *i* is an integer sets the statement limit to be *i* statements. Specifying the **p** option to *pc* disables statement limit counting.
time(a)	causes the current time in the form ' hh:mm:ss ' to be assigned to the alfa variable *a*.

Non-standard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x*, i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).
random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 536870912. The initial seed is 7774755.
seed(i)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i*.
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.

A.5. Remarks on standard and portable Pascal

It is occasionally desirable to prepare Pascal programs which will be acceptable at other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the non-portable usages. Wirth's *Pascal Report* concludes with a standard for implementation and program exchange.

In particular, the following differences may cause trouble when attempting to transport programs between this implementation and Pascal 6000−3.4. Using the s translator option may serve to indicate many problem areas.†

†The s option does not, however, check that identifiers differ in the first 8 characters. *Pi* and *pc* also do not check the semantics of **packed**.

Features not available in Berkeley Pascal

Segmented files and associated functions and procedures.

The function *trunc* with two arguments.

Arrays whose indices exceed the capacity of 16 bit arithmetic.

Features available in Berkeley Pascal but not in Pascal 6000-3.4

The procedures *reset* and *rewrite* with file names.

The functions *argc, seed, sysclock,* and *wallclock.*

The procedures *argv, flush,* and *remove.*

Message with arguments other than character strings.

Write with keyword **hex.**

The **assert** statement.

Reading and writing of enumerated types.

Allowing functions to return structures.

Separate compilation of programs.

Comparison of records.

Other problem areas

Sets and strings are more general in Berkeley Pascal; see the restrictions given in the Jensen-Wirth *User Manual* for details on the 6000−3.4 restrictions.

The character set differences may cause problems, especially the use of the function *chr,* characters as arguments to *ord,* and comparisons of characters, since the character set ordering differs between the two machines.

The Pascal 6000−3.4 compiler uses a less strict notion of type equivalence. In Berkeley Pascal, types are considered identical only if they are represented by the same type identifier. Thus, in particular, unnamed types are unique to the variables/fields declared with them.

Pascal 6000−3.4 doesn't recognize our option flags, so it is wise to put the control of Berkeley Pascal options to the end of option lists or, better yet, restrict the option list length to one.

For Pascal 6000−3.4 the ordering of files in the program statement has significance. It is desirable to place *input* and *output* as the first two files in the **program** statement.

CHAPTER 1

FRANZ LISP

1.1. FRANZ LISP[†] was created as a tool to further research in symbolic and algebraic manipulation, artificial intelligence, and programming languages at the University of California at Berkeley. Its roots are in a PDP-11 Lisp system which originally came from Harvard. As it grew it adopted features of Maclisp and Lisp Machine Lisp which enables our work to be shared with colleagues at the Laboratory for Computer Science at M.I.T. Substantial compatibility with other Lisp dialects (Interlisp, UCILisp, CMULisp) is achieved by means of support packages and compiler switches. The heart of FRANZ LISP is written almost entirely in the programming language C. Of course, it has been greatly extended by additions written in Lisp. A small part is written in the assembly language for the current host machines, VAXen and a couple of flavors of 68000. Because FRANZ LISP is written in C, it is relatively portable and easy to comprehend.

FRANZ LISP is capable of running large lisp programs in a timesharing environment, has facilities for arrays and user defined structures, has a user controlled reader with character and word macro capabilities, and can interact directly with compiled Lisp, C, Fortran, and Pascal code.

This document is a reference manual for the FRANZ LISP system. It is not a Lisp primer or introduction to the language. Some parts will be of interest only to those maintaining FRANZ LISP at their computer site. This document is divided into four Movements. In the first one we will attempt to describe the language of FRANZ LISP precisely and completely as it now stands (Opus 38.69, June 1983). In the second Movement we will look at the reader, function types, arrays and exception handling. In the third Movement we will look at several large support packages written to help the FRANZ LISP user, namely the trace package, compiler, fixit and stepping package. Finally the fourth movement contains an index into the other movements. In the rest of this chapter we shall examine the data types of FRANZ LISP. The conventions used in the description of the FRANZ LISP functions will be given in §1.3 -- it is very important that these conventions are understood.

1.2. Data Types FRANZ LISP has fourteen data types. In this section we shall look in detail at each type and if a type is divisible we shall look inside it. There is a Lisp function *type* which will return the type name of a lisp object. This is the official FRANZ LISP name for that type and we will use this name and this name only in the manual to avoid confusing the reader. The types are listed in terms of importance rather than alphabetically.

[†]It is rumored that this name has something to do with Franz Liszt [Frants List] (1811-1886) a Hungarian composer and keyboard virtuoso. These allegations have never been proven.

1.2.0. lispval This is the name we use to describe any lisp object. The function *type* will never return 'lispval'.

1.2.1. symbol This object corresponds to a variable in most other programming languages. It may have a value or may be 'unbound'. A symbol may be *lambda bound* meaning that its current value is stored away somewhere and the symbol is given a new value for the duration of a certain context. When the Lisp processor leaves that context, the symbol's current value is thrown away and its old value is restored.

A symbol may also have a *function binding.* This function binding is static; it cannot be lambda bound. Whenever the symbol is used in the functional position of a Lisp expression the function binding of the symbol is examined (see Chapter 4 for more details on evaluation).

A symbol may also have a *property list*, another static data structure. The property list consists of a list of an even number of elements, considered to be grouped as pairs. The first element of the pair is the *indicator* the second the *value* of that indicator.

Each symbol has a print name *(pname)* which is how this symbol is accessed from input and referred to on (printed) output.

A symbol also has a hashlink used to link symbols together in the oblist -- this field is inaccessible to the lisp user.

Symbols are created by the reader and by the functions *concat*, *maknam* and their derivatives. Most symbols live on FRANZ LISP's sole *oblist*, and therefore two symbols with the same print name are usually the exact same object (they are *eq*). Symbols which are not on the oblist are said to be *uninterned.* The function *maknam* creates uninterned symbols while *concat* creates *interned* ones.

Subpart name	Get value	Set value	Type
value	eval	set setq	lispval
property list	plist get	setplist putprop defprop	list or nil
function binding	getd	putd def	array, binary, list or nil
print name	get_pname		string
hash link			

1.2.2. list A list cell has two parts, called the car and cdr. List cells are created by the function *cons.*

Subpart name	Get value	Set value	Type
car	car	rplaca	lispval
cdr	cdr	rplacd	lispval

1.2.3. binary This type acts as a function header for machine coded functions. It has two parts, a pointer to the start of the function and a symbol whose print name describes the argument *discipline.* The discipline (if *lambda, macro* or *nlambda*) determines whether the arguments to this function will be evaluated by the caller before this function is called. If the discipline is a string (specifically "*subroutine*", "*function*", "*integer-function*", "*real-function*", "*c-function*", "*double-c-function*", or "*vector-c-function*") then this function is a foreign subroutine or function (see §8.5 for more details on this). Although the type of the *entry* field of a binary type object is usually **string** or **other**, the object pointed to is actually a sequence of machine instructions.
Objects of type binary are created by *mfunction, cfasl,* and *getaddress.*

Subpart name	Get value	Set value	Type
entry	getentry		string or fixnum
discipline	getdisc	putdisc	symbol or fixnum

1.2.4. fixnum A fixnum is an integer constant in the range -2^{31} to $2^{31}-1$. Small fixnums (-1024 to 1023) are stored in a special table so they needn't be allocated each time one is needed.

1.2.5. flonum A flonum is a double precision real number in the range $\pm 2.9\times10^{-37}$ to $\pm 1.7\times10^{38}$. There are approximately sixteen decimal digits of precision.

1.2.6. bignum A bignum is an integer of potentially unbounded size. When integer arithmetic exceeds the limits of fixnums mentioned above, the calculation is automatically done with bignums. Should calculation with bignums give a result which can be represented as a fixnum, then the fixnum representation will be used†. This contraction is known as *integer normalization.* Many Lisp functions assume that integers are normalized. Bignums are composed of a sequence of **list** cells and a cell known as an **sdot.** The user should consider a **bignum** structure indivisible and use functions such as *haipart,* and *bignum-leftshift* to extract parts of it.

†The current algorithms for integer arithmetic operations will return (in certain cases) a result between $\pm 2^{30}$ and 2^{31} as a bignum although this could be represented as a fixnum.

1.2.7. string A string is a null terminated sequence of characters. Most functions of symbols which operate on the symbol's print name will also work on strings. The default reader syntax is set so that a sequence of characters surrounded by double quotes is a string.

1.2.8. port A port is a structure which the system I/O routines can reference to transfer data between the Lisp system and external media. Unlike other Lisp objects there are a very limited number of ports (20). Ports are allocated by *infile* and *outfile* and deallocated by *close* and *resetio*. The *print* function prints a port as a percent sign followed by the name of the file it is connected to (if the port was opened by *fileopen, infile, or outfile*). During initialization, FRANZ LISP binds the symbol **piport** to a port attached to the standard input stream. This port prints as %$stdin. There are ports connected to the standard output and error streams, which print as %$stdout and %$stderr. This is discussed in more detail at the beginning of Chapter 5.

1.2.9. vector Vectors are indexed sequences of data. They can be used to implement a notion of user-defined types, via their associated property list. They make **hunks** (see below) logically unnecessary, although hunks are very efficiently garbage collected. There is a second kind of vector, called an immediate-vector, which stores binary data. The name that the function *type* returns for immediate-vectors is **vectori**. Immediate-vectors could be used to implement strings and block-flonum arrays, for example. Vectors are discussed in chapter 9. The functions *new-vector*, and *vector*, can be used to create vectors.

Subpart name	Get value	Set value	Type
datum[*i*]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	–	fixnum

1.2.10. array Arrays are rather complicated types and are fully described in Chapter 9. An array consists of a block of contiguous data, a function to access that data and auxiliary fields for use by the accessing function. Since an array's accessing function is created by the user, an array can have any form the user chooses (e.g. n-dimensional, triangular, or hash table).
Arrays are created by the function *marray*.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

1.2.11. value A value cell contains a pointer to a lispval. This type is used mainly by arrays of general lisp objects. Value cells are created with the *ptr* function. A value cell containing a pointer to the symbol 'foo' is printed as '(ptr to)foo'

1.2.12. hunk A hunk is a vector of from 1 to 128 lispvals. Once a hunk is created (by *hunk* or *makhunk*) it cannot grow or shrink. The access time for an element of a hunk is slower than a list cell element but faster than an array. Hunks are really only allocated in sizes which are powers of two, but can appear to the user to be any size in the 1 to 128 range. Users of hunks must realize that *(not (atom 'lispval))* will return true if *lispval* is a hunk. Most lisp systems do not have a direct test for a list cell and instead use the above test and assume that a true result means *lispval* is a list cell. In FRANZ LISP you can use *dtpr* to check for a list cell. Although hunks are not list cells, you can still access the first two hunk elements with *cdr* and *car* and you can access any hunk element with *cxr*[†]. You can set the value of the first two elements of a hunk with *rplacd* and *rplaca* and you can set the value of any element of the hunk with *rplacx*. A hunk is printed by printing its contents surrounded by { and }. However a hunk cannot be read in in this way in the standard lisp system. It is easy to write a reader macro to do this if desired.

1.2.13. other Occasionally, you can obtain a pointer to storage not allocated by the lisp system. One example of this is the entry field of those FRANZ LISP functions written in C. Such objects are classified as of type **other**. Foreign functions which call malloc to allocate their own space, may also inadvertantly create such objects. The garbage collector is supposed to ignore such objects.

1.3. Documentation The conventions used in the following chapters were designed to give a great deal of information in a brief space. The first line of a function description contains the function name in **bold face** and then lists the arguments, if any. The arguments all have names which begin with a letter or letters and an underscore. The letter(s) gives the allowable type(s) for that argument according to this table.

[†]In a hunk, the function *cdr* references the first element and *car* the second.

Letter	Allowable type(s)
g	any type
s	symbol (although nil may not be allowed)
t	string
l	list (although nil may be allowed)
n	number (fixnum, flonum, bignum)
i	integer (fixnum, bignum)
x	fixnum
b	bignum
f	flonum
u	function type (either binary or lambda body)
y	binary
v	vector
V	vectori
a	array
e	value
p	port (or nil)
h	hunk

In the first line of a function description, those arguments preceded by a quote mark are evaluated (usually before the function is called). The quoting convention is used so that we can give a name to the result of evaluating the argument and we can describe the allowable types. If an argument is not quoted it does not mean that that argument will not be evaluated, but rather that if it is evaluated, the time at which it is evaluated will be specifically mentioned in the function description. Optional arguments are surrounded by square brackets. An ellipsis (...) means zero or more occurrences of an argument of the directly preceding type.

CHAPTER 2

Data Structure Access

The following functions allow one to create and manipulate the various types of lisp data structures. Refer to §1.2 for details of the data structures known to FRANZ LISP.

2.1. Lists

The following functions exist for the creation and manipulating of lists. Lists are composed of a linked list of objects called either 'list cells', 'cons cells' or 'dtpr cells'. Lists are normally terminated with the special symbol **nil**. **nil** is both a symbol and a representation for the empty list ().

2.1.1. list creation

(cons 'g_arg1 'g_arg2**)**

RETURNS: a new list cell whose car is g_arg1 and whose cdr is g_arg2.

(xcons 'g_arg1 'g_arg2**)**

EQUIVALENT TO: *(cons 'g_arg2 'g_arg1)*

(ncons 'g_arg**)**

EQUIVALENT TO: *(cons 'g_arg nil)*

(list ['g_arg1 ...]**)**

RETURNS: a list whose elements are the g_arg*i*.

(append 'l_arg1 'l_arg2**)**

RETURNS: a list containing the elements of l_arg1 followed by l_arg2.

NOTE: To generate the result, the top level list cells of l_arg1 are duplicated and the cdr of the last list cell is set to point to l_arg2. Thus this is an expensive operation if l_arg1 is large. See the descriptions of *nconc* and *tconc* for cheaper ways of doing the *append* if the list l_arg1 can be altered.

(append1 'l_arg1 'g_arg2**)**

RETURNS: a list like l_arg1 with g_arg2 as the last element.

NOTE: this is equivalent to (append 'l_arg1 (list 'g_arg2)).

```
; A common mistake is using append to add one element to the end of a list
-> (append '(a b c d) 'e)
(a b c d . e)
; The user intended to say:
-> (append '(a b c d) '(e))
(a b c d e)
; better is append1
-> (append1 '(a b c d) 'e)
(a b c d e)
```

(quote! [g_qform*i*] ...[! 'g_eform*i*] ... [!! 'l_form*i*] ...**)**

RETURNS: The list resulting from the splicing and insertion process described below.

NOTE: *quote!* is the complement of the *list* function. *list* forms a list by evaluating each for in the argument list; evaluation is suppressed if the form is *quote*ed. In *quote!*, each form is implicitly *quote*ed. To be evaluated, a form must be preceded by one of the evaluate operations ! and !!. ! g_eform evaluates g_form and the value is inserted in the place of the call; !! l_form evaluates l_form and the value is spliced into the place of the call.

'Splicing in' means that the parentheses surrounding the list are removed as the example below shows. Use of the evaluate operators can occur at any level in a form argument.

Another way to get the effect of the *quote!* function is to use the backquote character macro (see § 8.3.3).

```
(quote! cons ! (cons 1 2) 3) = (cons (1 . 2) 3)
(quote! 1 !! (list 2 3 4) 5) = (1 2 3 4 5)
(setq quoted 'evaled)(quote! ! ((I am ! quoted))) = ((I am evaled))
(quote! try ! '(this ! one)) = (try (this ! one))
```

(bignum-to-list 'b_arg**)**

RETURNS: A list of the fixnums which are used to represent the bignum.

NOTE: the inverse of this function is *list-to-bignum.*

(list-to-bignum 'l_ints**)**

WHERE: l_ints is a list of fixnums.

RETURNS: a bignum constructed of the given fixnums.

NOTE: the inverse of this function is *bignum-to-list.*

2.1.2. list predicates

(dtpr 'g_arg**)**

RETURNS: t iff g_arg is a list cell.

NOTE: that (dtpr '()) is nil.

(listp 'g_arg**)**

RETURNS: t iff g_arg is a list object or nil.

(tailp 'l_x 'l_y**)**

RETURNS: l_x, if a list cell *eq* to l_x is found by *cdr*ing down l_y zero or more times, nil otherwise.

```
-> (setq x '(a b c d) y (cddr x))
(c d)
-> (and (dtpr x) (listp x))     ; x and y are dtprs and lists
t
-> (dtpr '())                   ; () is the same as nil and is not a dtpr
nil
-> (listp '())                  ; however it is a list
t
-> (tailp y x)
(c d)
```

(length 'l_arg**)**

RETURNS: the number of elements in the top level of list l_arg.

2.1.3. list accessing

(car 'l_arg**)**
(cdr 'l_arg**)**

RETURNS: *cons* cell. (*car* (*cons* x y)) is always x, (*cdr* (*cons* x y)) is always y. In FRANZ LISP, the cdr portion is located first in memory. This is hardly noticeable, and seems to bother few.

(c..r 'lh_arg**)**

WHERE: the .. represents any positive number of **a**'s and **d**'s.

RETURNS: the result of accessing the list structure in the way determined by the function name. The **a**'s and **d**'s are read from right to left, a **d** directing the access down the cdr part of the list cell and an **a** down the car part.

NOTE: lh_arg may also be nil, and it is guaranteed that the car and cdr of nil is nil. If lh_arg is a hunk, then *(car 'lh_arg)* is the same as *(cxr 1 'lh_arg)* and *(cdr 'lh_arg)* is the same as *(cxr 0 'lh_arg)*.
It is generally hard to read and understand the context of functions with large strings of **a**'s and **d**'s, but these functions are supported by rapid accessing and open-compiling (see Chapter 12).

(nth 'x_index 'l_list**)**

RETURNS: the nth element of l_list, assuming zero-based index. Thus (nth 0 l_list) is the same as (car l_list). *nth* is both a function, and a compiler macro, so that more efficient code might be generated than for *nthelem* (described below).

NOTE: If x_arg1 is non-positive or greater than the length of the list, nil is returned.

(nthcdr 'x_index 'l_list**)**

RETURNS: the result of *cdr*ing down the list l_list x_index times.

NOTE: If x_index is less than 0, then *(cons nil 'l_list)* is returned.

(nthelem 'x_arg1 'l_arg2**)**

RETURNS: The x_arg1'*st* element of the list l_arg2.

NOTE: This function comes from the PDP-11 lisp system.

(last 'l_arg**)**

RETURNS: the last list cell in the list l_arg.

EXAMPLE: *last* does NOT return the last element of a list!
(last '(a b)) = (b)

(ldiff 'l_x 'l_y**)**

RETURNS: a list of all elements in l_x but not in l_y , i.e., the list difference of l_x and l_y.

NOTE: l_y must be a tail of l_x, i.e., *eq* to the result of applying some number of *cdr*'s to l_x. Note that the value of *ldiff* is always new list structure unless l_y is nil, in which case *(ldiff l_x nil)* is l_x itself. If l_y is not a tail of l_x, *ldiff* generates an error.

EXAMPLE: *(ldiff 'l_x (member 'g_foo 'l_x))* gives all elements in l_x up to the first g_foo.

2.1.4. list manipulation

(rplaca 'lh_arg1 'g_arg2)

RETURNS: the modified lh_arg1.

SIDE EFFECT: the car of lh_arg1 is set to g_arg2. If lh_arg1 is a hunk then the second element of the hunk is set to g_arg2.

(rplacd 'lh_arg1 'g_arg2)

RETURNS: the modified lh_arg1.

SIDE EFFECT: the cdr of lh_arg2 is set to g_arg2. If lh_arg1 is a hunk then the first element of the hunk is set to g_arg2.

(attach 'g_x 'l_l)

RETURNS: l_l whose *car* is now g_x, whose *cadr* is the original *(car l_l)*, and whose *cddr* is the original *(cdr l_l)*.

NOTE: what happens is that g_x is added to the beginning of list l_l yet maintaining the same list cell at the beginning of the list.

(delete 'g_val 'l_list ['x_count])

RETURNS: the result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: x_count defaults to a very large number, thus if x_count is not given, all occurrences of g_val are removed from the top level of l_list. g_val is compared with successive *car*'s of l_list using the function *equal*.

SIDE EFFECT: l_list is modified using rplacd, no new list cells are used.

(delq 'g_val 'l_list ['x_count])
(dremove 'g_val 'l_list ['x_count])

RETURNS: the result of splicing g_val from the top level of l_list no more than x_count times.

NOTE: *delq* (and *dremove*) are the same as *delete* except that *eq* is used for comparison instead of *equal*.

```
; note that you should use the value returned by delete or delq
; and not assume that g_val will always show the deletions.
; For example

-> (setq test '(a b c a d e))
(a b c a d e)
-> (delete 'a test)
(b c d e)       ; the value returned is what we would expect
-> test
(a b c d e)     ; but test still has the first a in the list!
```

(remq 'g_x 'l_l ['x_count])
(remove 'g_x 'l_l)

RETURNS: a *copy* of l_l with all top level elements *equal* to g_x removed. *remq* uses *eq* instead of *equal* for comparisons.

NOTE: remove does not modify its arguments like *delete*, and *delq* do.

(insert 'g_object 'l_list 'u_comparefn 'g_nodups)

RETURNS: a list consisting of l_list with g_object destructively inserted in a place determined by the ordering function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g_x can precede g_y in sorted order, nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used. If g_nodups is non-nil, an element will not be inserted if an equal element is already in the list. *insert* does binary search to determine where to insert the new element.

(merge 'l_data1 'l_data2 'u_comparefn)

RETURNS: the merged list of the two input sorted lists l_data1 and l_data1 using binary comparison function u_comparefn.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g_x can precede g_y in sorted order, nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used. u_comparefn should be thought of as "less than or equal". *merge* changes both of its data arguments.

(subst 'g_x 'g_y 'l_s)
(dsubst 'g_x 'g_y 'l_s)

RETURNS: the result of substituting g_x for all *equal* occurrences of g_y at all levels in l_s.

NOTE: If g_y is a symbol, *eq* will be used for comparisons. The function *subst* does not modify l_s but the function *dsubst* (destructive substitution) does.

(lsubst 'l_x 'g_y 'l_s)

RETURNS: a copy of l_s with l_x spliced in for every occurrence of of g_y at all levels. Splicing in means that the parentheses surrounding the list l_x are removed as the example below shows.

```
-> (subst '(a b c) 'x '(x y z (x y z) (x y z)))
((a b c) y z ((a b c) y z) ((a b c) y z))
-> (lsubst '(a b c) 'x '(x y z (x y z) (x y z)))
(a b c y z (a b c y z) (a b c y z))
```

(subpair 'l_old 'l_new 'l_expr**)**

WHERE: there are the same number of elements in l_old as l_new.

RETURNS: the list l_expr with all occurrences of a object in l_old replaced by the corresponding one in l_new. When a substitution is made, a copy of the value to substitute in is not made.

EXAMPLE: *(subpair '(a c)' (x y) '(a b c d)) = (x b y d)*

(nconc 'l_arg1 'l_arg2 ['l_arg3 ...]**)**

RETURNS: A list consisting of the elements of l_arg1 followed by the elements of l_arg2 followed by l_arg3 and so on.

NOTE: The *cdr* of the last list cell of l_arg*i* is changed to point to l_arg*i*+*1*.

```
; nconc is faster than append because it doesn't allocate new list cells.
-> (setq lis1 '(a b c))
(a b c)
-> (setq lis2 '(d e f))
(d e f)
-> (append lis1 lis2)
(a b c d e f)
-> lis1
(a b c)      ; note that lis1 has not been changed by append
-> (nconc lis1 lis2)
(a b c d e f) ; nconc returns the same value as append
-> lis1
(a b c d e f) ; but in doing so alters lis1
```

(reverse 'l_arg**)**
(nreverse 'l_arg**)**

RETURNS: the list l_arg with the elements at the top level in reverse order.

NOTE: The function *nreverse* does the reversal in place, that is the list structure is modified.

(nreconc 'l_arg 'g_arg**)**

EQUIVALENT TO: *(nconc (nreverse 'l_arg) 'g_arg)*

2.2. Predicates

The following functions test for properties of data objects. When the result of the test is either 'false' or 'true', then **nil** will be returned for 'false' and something other than **nil** (often **t**) will be returned for 'true'.

(arrayp 'g_arg**)**

RETURNS: t iff g_arg is of type array.

(atom 'g_arg**)**

RETURNS: t iff g_arg is not a list or hunk object.

NOTE: *(atom '())* returns t.

(bcdp 'g_arg**)**

RETURNS: t iff g_arg is a data object of type binary.

NOTE: the name of this function is a throwback to the PDP-11 Lisp system.

(bigp 'g_arg**)**

RETURNS: t iff g_arg is a bignum.

(dtpr 'g_arg**)**

RETURNS: t iff g_arg is a list cell.

NOTE: that (dtpr '()) is nil.

(hunkp 'g_arg**)**

RETURNS: t iff g_arg is a hunk.

(listp 'g_arg**)**

RETURNS: t iff g_arg is a list object or nil.

(stringp 'g_arg**)**

RETURNS: t iff g_arg is a string.

(symbolp 'g_arg**)**

RETURNS: t iff g_arg is a symbol.

(valuep 'g_arg**)**

RETURNS: t iff g_arg is a value cell

(vectorp 'v_vector**)**

RETURNS: **t** iff the argument is a vector.

(vectorip 'v_vector**)**

RETURNS: **t** iff the argument is an immediate-vector.

(type 'g_arg)
(typep 'g_arg)

RETURNS: a symbol whose pname describes the type of g_arg.

(signp s_test 'g_val)

RETURNS: t iff g_val is a number and the given test s_test on g_val returns true.

NOTE: The fact that *signp* simply returns nil if g_val is not a number is probably the most important reason that *signp* is used. The permitted values for s_test and what they mean are given in this table.

s_test	tested
l	g_val $<$ 0
le	g_val $\leqslant$ 0
e	g_val $=$ 0
n	g_val $\neq$ 0
ge	g_val $\geqslant$ 0
g	g_val $>$ 0

(eq 'g_arg1 'g_arg2)

RETURNS: t if g_arg1 and g_arg2 are the exact same lisp object.

NOTE: *Eq* simply tests if g_arg1 and g_arg2 are located in the exact same place in memory. Lisp objects which print the same are not necessarily *eq*. The only objects guaranteed to be *eq* are interned symbols with the same print name. [Unless a symbol is created in a special way (such as with *uconcat* or *maknam*) it will be interned.]

(neq 'g_x 'g_y)

RETURNS: t if g_x is not *eq* to g_y, otherwise nil.

(equal 'g_arg1 'g_arg2)
(eqstr 'g_arg1 'g_arg2)

RETURNS: t iff g_arg1 and g_arg2 have the same structure as described below.

NOTE: g_arg and g_arg2 are *equal* if

(1) they are *eq*.
(2) they are both fixnums with the same value
(3) they are both flonums with the same value
(4) they are both bignums with the same value
(5) they are both strings and are identical.
(6) they are both lists and their cars and cdrs are *equal*.

```
; eq is much faster than equal, especially in compiled code,
; however you cannot use eq to test for equality of numbers outside
; of the range -1024 to 1023.  equal will always work.
-> (eq 1023 1023)
t
-> (eq 1024 1024)
nil
-> (equal 1024 1024)
t
```

(not 'g_arg**)**
(null 'g_arg**)**

RETURNS: t iff g_arg is nil.

(member 'g_arg1 'l_arg2**)**
(memq 'g_arg1 'l_arg2**)**

RETURNS: that part of the l_arg2 beginning with the first occurrence of g_arg1. If g_arg1 is not in the top level of l_arg2, nil is returned.

NOTE: *member* tests for equality with *equal*, *memq* tests for equality with *eq*.

2.3. Symbols and Strings

In many of the following functions the distinction between symbols and strings is somewhat blurred. To remind ourselves of the difference, a string is a null terminated sequence of characters, stored as compactly as possible. Strings are used as constants in FRANZ LISP. They *eval* to themselves. A symbol has additional structure: a value, property list, function binding, as well as its external representation (or print-name). If a symbol is given to one of the string manipulation functions below, its print name will be used.

Another popular way to represent strings in Lisp is as a list of fixnums which represent characters. The suffix 'n' to a string manipulation function indicates that it returns a string in this form.

2.3.1. symbol and string creation

(concat ['stn_arg1 ...])
(uconcat ['stn_arg1 ...])

RETURNS: a symbol whose print name is the result of concatenating the print names, string characters or numerical representations of the sn_arg*i*.

NOTE: If no arguments are given, a symbol with a null pname is returned. *concat* places the symbol created on the oblist, the function *uconcat* does the same thing but does not place the new symbol on the oblist.

EXAMPLE: *(concat 'abc (add 3 4) "def")* = abc7def

(concatl 'l_arg)

EQUIVALENT TO: *(apply 'concat 'l_arg)*

(implode 'l_arg)
(maknam 'l_arg)

WHERE: l_arg is a list of symbols, strings and small fixnums.

RETURNS: The symbol whose print name is the result of concatenating the first characters of the print names of the symbols and strings in the list. Any fixnums are converted to the equivalent ascii character. In order to concatenate entire strings or print names, use the function *concat*.

NOTE: *implode* interns the symbol it creates, *maknam* does not.

(gensym ['s_leader])

RETURNS: a new uninterned atom beginning with the first character of s_leader's pname, or beginning with g if s_leader is not given.

NOTE: The symbol looks like x0nnnnn where x is s_leader's first character and nnnnn is the number of times you have called gensym.

(copysymbol 's_arg 'g_pred)

RETURNS: an uninterned symbol with the same print name as s_arg. If g_pred is non nil, then the value, function binding and property list of the new symbol are made *eq* to those of s_arg.

(ascii 'x_charnum)

WHERE: x_charnum is between 0 and 255.

RETURNS: a symbol whose print name is the single character whose fixnum representation is x_charnum.

(intern 's_arg**)**

RETURNS: s_arg

SIDE EFFECT: s_arg is put on the oblist if it is not already there.

(remob 's_symbol**)**

RETURNS: s_symbol

SIDE EFFECT: s_symbol is removed from the oblist.

(rematom 's_arg**)**

RETURNS: t if s_arg is indeed an atom.

SIDE EFFECT: s_arg is put on the free atoms list, effectively reclaiming an atom cell.

NOTE: This function does *not* check to see if s_arg is on the oblist or is referenced anywhere. Thus calling *rematom* on an atom in the oblist may result in disaster when that atom cell is reused!

2.3.2. string and symbol predicates

(boundp 's_name**)**

RETURNS: nil if s_name is unbound, that is it has never be given a value. If x_name has the value g_val, then (nil . g_val) is returned.

(alphalessp 'st_arg1 'st_arg2**)**

RETURNS: t iff the 'name' of st_arg1 is alphabetically less than the name of st_arg2. If st_arg is a symbol then its 'name' is its print name. If st_arg is a string, then its 'name' is the string itself.

2.3.3. symbol and string accessing

(symeval 's_arg**)**

RETURNS: the value of symbol s_arg.

NOTE: It is illegal to ask for the value of an unbound symbol. This function has the same effect as *eval*, but compiles into much more efficient code.

(get_pname 's_arg**)**

RETURNS: the string which is the print name of s_arg.

(plist 's_arg**)**

RETURNS: the property list of s_arg.

(getd 's_arg**)**

RETURNS: the function definition of s_arg or nil if there is no function definition.

NOTE: the function definition may turn out to be an array header.

(getchar 's_arg 'x_index**)**
(nthchar 's_arg 'x_index**)**
(getcharn 's_arg 'x_index**)**

RETURNS: the x_index*th* character of the print name of s_arg or nil if x_index is less than 1 or greater than the length of s_arg's print name.

NOTE: *getchar* and *nthchar* return a symbol with a single character print name, *getcharn* returns the fixnum representation of the character.

(substring 'st_string 'x_index ['x_length]**)**
(substringn 'st_string 'x_index ['x_length]**)**

RETURNS: a string of length at most x_length starting at x_index*th* character in the string.

NOTE: If x_length is not given, all of the characters for x_index to the end of the string are returned. If x_index is negative the string begins at the x_index*th* character from the end. If x_index is out of bounds, nil is returned.

NOTE: *substring* returns a list of symbols, *substringn* returns a list of fixnums. If *substringn* is given a 0 x_length argument then a single fixnum which is the x_index*th* character is returned.

2.3.4. symbol and string manipulation

(set 's_arg1 'g_arg2**)**

RETURNS: g_arg2.

SIDE EFFECT: the value of s_arg1 is set to g_arg2.

(setq s_atm1 'g_val1 [s_atm2 'g_val2 ... ...]**)**

WHERE: the arguments are pairs of atom names and expressions.

RETURNS: the last g_val*i*.

SIDE EFFECT: each s_atm*i* is set to have the value g_val*i*.

NOTE: *set* evaluates all of its arguments, *setq* does not evaluate the s_atm*i*.

(desetq sl_pattern1 'g_exp1 [... ...])

RETURNS: g_expn

SIDE EFFECT: This acts just like *setq* if all the sl_pattern*i* are symbols. If sl_pattern*i* is a list then it is a template which should have the same structure as g_exp*i* The symbols in sl_pattern are assigned to the corresponding parts of g_exp.

EXAMPLE: *(desetq (a b (c . d)) '(1 2 (3 4 5)))*
sets a to 1, b to 2, c to 3, and d to (4 5).

(setplist 's_atm 'l_plist)

RETURNS: l_plist.

SIDE EFFECT: the property list of s_atm is set to l_plist.

(makunbound 's_arg)

RETURNS: s_arg

SIDE EFFECT: the value of s_arg is made 'unbound'. If the interpreter attempts to evaluate s_arg before it is again given a value, an unbound variable error will occur.

(aexplode 's_arg)
(explode 'g_arg)
(aexplodec 's_arg)
(explodec 'g_arg)
(aexploden 's_arg)
(exploden 'g_arg)

RETURNS: a list of the characters used to print out s_arg or g_arg.

NOTE: The functions beginning with 'a' are internal functions which are limited to symbol arguments. The functions *aexplode* and *explode* return a list of characters which *print* would use to print the argument. These characters include all necessary escape characters. Functions *aexplodec* and *explodec* return a list of characters which *patom* would use to print the argument (i.e. no escape characters). Functions *aexploden* and *exploden* are similar to *aexplodec* and *explodec* except that a list of fixnum equivalents of characters are returned.

```
-> (setq x |quote this \| ok?|)
|quote this \| ok?|
-> (explode x)
(q u o t e |\| | t h i s |\| | |\| |\| |\| | o k ?)
; note that |\| just means the single character: backslash.
; and |\| just means the single character: vertical bar
; and | | means the single character: space

-> (explodec x)
(q u o t e | | t h i s | | |\| | | o k ?)
-> (exploden x)
(113 117 111 116 101 32 116 104 105 115 32 124 32 111 107 63)
```

2.4. Vectors

See Chapter 9 for a discussion of vectors. They are intermediate in efficiency between arrays and hunks.

2.4.1. vector creation

(new-vector 'x_size ['g_fill ['g_prop]])

RETURNS: A **vector** of length x_size. Each data entry is initialized to g_fill, or to nil, if the argument g_fill is not present. The vector's property is set to g_prop, or to nil, by default.

(new-vectori-byte 'x_size ['g_fill ['g_prop]])
(new-vectori-word 'x_size ['g_fill ['g_prop]])
(new-vectori-long 'x_size ['g_fill ['g_prop]])

RETURNS: A **vectori** with x_size elements in it. The actual memory requirement is two long words + x_size*(n bytes), where n is 1 for new-vector-byte, 2 for new-vector-word, or 4 for new-vectori-long. Each data entry is initialized to g_fill, or to zero, if the argument g_fill is not present. The vector's property is set to g_prop, or nil, by default.

Vectors may be created by specifying multiple initial values:

(vector ['g_val0 'g_val1 ...])

RETURNS: a **vector**, with as many data elements as there are arguments. It is quite possible to have a vector with no data elements. The vector's property will be null.

(vectori-byte ['x_val0 'x_val2 ...])
(vectori-word ['x_val0 'x_val2 ...])
(vectori-long ['x_val0 'x_val2 ...])

RETURNS: a **vectori**, with as many data elements as there are arguments. The arguments are required to be fixnums. Only the low order byte or word is used in the case of vectori-byte and vectori-word. The vector's property will be null.

2.4.2. vector reference

(vref 'v_vect 'x_index)
(vrefi-byte 'V_vect 'x_bindex)
(vrefi-word 'V_vect 'x_windex)
(vrefi-long 'V_vect 'x_lindex)

RETURNS: the desired data element from a vector. The indices must be fixnums. Indexing is zero-based. The vrefi functions sign extend the data.

(vprop 'Vv_vect**)**

RETURNS: The Lisp property associated with a vector.

(vget 'Vv_vect**)**

RETURNS: The value stored under g_ind if the Lisp property associated with 'Vv_vect is a disembodied property list.

(vsize 'Vv_vect**)**
(vsize-byte 'V_vect**)**
(vsize-word 'V_vect**)**

RETURNS: the number of data elements in the vector. For immediate-vectors, the functions vsize-byte and vsize-word return the number of data elements, if one thinks of the binary data as being comprised of bytes or words.

2.4.3. vector modfication

(vset 'v_vect 'x_index 'g_val**)**
(vseti-byte 'V_vect 'x_bindex 'x_val**)**
(vseti-word 'V_vect 'x_windex 'x_val**)**
(vseti-long 'V_vect 'x_lindex 'x_val**)**

RETURNS: the datum.

SIDE EFFECT: The indexed element of the vector is set to the value. As noted above, for vseti-word and vseti-byte, the index is construed as the number of the data element within the vector. It is not a byte address. Also, for those two functions, the low order byte or word of x_val is what is stored.

(vsetprop 'Vv_vect 'g_value**)**

RETURNS: g_value. This should be either a symbol or a disembodied property list whose *car* is a symbol identifying the type of the vector.

SIDE EFFECT: the property list of Vv_vect is set to g_value.

(vputprop 'Vv_vect 'g_value 'g_ind**)**

RETURNS: g_value.

SIDE EFFECT: If the vector property of Vv_vect is a disembodied property list, then vputprop adds the value g_value under the indicator g_ind. Otherwise, the old vector property is made the first element of the list.

2.5. Arrays

See Chapter 9 for a complete description of arrays. Some of these functions are part of a Maclisp array compatibility package, which represents only one simple way of using the array structure of FRANZ LISP.

2.5.1. array creation

(marray 'g_data 's_access 'g_aux 'x_length 'x_delta**)**

RETURNS: an array type with the fields set up from the above arguments in the obvious way (see § 1.2.10).

(*array 's_name 's_type 'x_dim1 ... 'x_dim*n***)**
(array s_name s_type x_dim1 ... x_dim*n***)**

WHERE: s_type may be one of t, nil, fixnum, flonum, fixnum-block and flonum-block.

RETURNS: an array of type s_type with n dimensions of extents given by the x_dim*i*.

SIDE EFFECT: If s_name is non nil, the function definition of s_name is set to the array structure returned.

NOTE: These functions create a Maclisp compatible array. In FRANZ LISP arrays of type t, nil, fixnum and flonum are equivalent and the elements of these arrays can be any type of lisp object. Fixnum-block and flonum-block arrays are restricted to fixnums and flonums respectively and are used mainly to communicate with foreign functions (see §8.5).

NOTE: **array* evaluates its arguments, *array* does not.

2.5.2. array predicate

(arrayp 'g_arg**)**

RETURNS: t iff g_arg is of type array.

2.5.3. array accessors

(getaccess 'a_array**)**
(getaux 'a_array**)**
(getdelta 'a_array**)**
(getdata 'a_array**)**
(getlength 'a_array**)**

RETURNS: the field of the array object a_array given by the function name.

(arrayref 'a_name 'x_ind**)**

RETURNS: the x_ind*th* element of the array object a_name. x_ind of zero accesses the first element.

NOTE: *arrayref* uses the data, length and delta fields of a_name to determine which object to return.

(arraycall s_type 'as_array 'x_ind1 ... **)**

RETURNS: the element selected by the indicies from the array a_array of type s_type.

NOTE: If as_array is a symbol then the function binding of this symbol should contain an array object.
s_type is ignored by *arraycall* but is included for compatibility with Maclisp.

(arraydims 's_name**)**

RETURNS: a list of the type and bounds of the array s_name.

(listarray 'sa_array ['x_elements]**)**

RETURNS: a list of all of the elements in array sa_array. If x_elements is given, then only the first x_elements are returned.

```
; We will create a 3 by 4 array of general lisp objects
-> (array ernie t 3 4)
array[12]

; the array header is stored in the function definition slot of the
; symbol ernie
-> (arrayp (getd 'ernie))
t
-> (arraydims (getd 'ernie))
(t 3 4)

; store in ernie[2][2] the list (test list)
-> (store (ernie 2 2) '(test list))
(test list)

; check to see if it is there
-> (ernie 2 2)
(test list)

; now use the low level function arrayref to find the same element
; arrays are 0 based and row-major (the last subscript varies the fastest)
; thus element [2][2] is the 10th element , (starting at 0).
-> (arrayref (getd 'ernie) 10)
(ptr to)(test list)    ; the result is a value cell (thus the (ptr to))
```

2.5.4. array manipulation

(putaccess 'a_array 'su_func)
(putaux 'a_array 'g_aux)
(putdata 'a_array 'g_arg)
(putdelta 'a_array 'x_delta)
(putlength 'a_array 'x_length)

RETURNS: the second argument to the function.

SIDE EFFECT: The field of the array object given by the function name is replaced by the second argument to the function.

(store 'l_arexp 'g_val)

WHERE: l_arexp is an expression which references an array element.

RETURNS: g_val

SIDE EFFECT: the array location which contains the element which l_arexp references is changed to contain g_val.

(fillarray 's_array 'l_itms)

RETURNS: s_array

SIDE EFFECT: the array s_array is filled with elements from l_itms. If there are not enough elements in l_itms to fill the entire array, then the last element of l_itms is used to fill the remaining parts of the array.

2.6. Hunks

Hunks are vector-like objects whose size can range from 1 to 128 elements. Internally hunks are allocated in sizes which are powers of 2. In order to create hunks of a given size, a hunk with at least that many elements is allocated and a distinguished symbol EMPTY is placed in those elements not requested. Most hunk functions respect those distinguished symbols, but there are two *(*makhunk* and **rplacx)* which will overwrite the distinguished symbol.

2.6.1. hunk creation

(hunk 'g_val1 ['g_val2 ... 'g_val*n*])

RETURNS: a hunk of length n whose elements are initialized to the g_val*i*.

NOTE: the maximum size of a hunk is 128.

EXAMPLE: *(hunk 4 'sharp 'keys)* = {4 sharp keys}

(makhunk 'xl_arg)

RETURNS: a hunk of length xl_arg initialized to all nils if xl_arg is a fixnum. If xl_arg is a list, then we return a hunk of size *(length 'xl_arg)* initialized to the elements in xl_arg.

NOTE: *(makhunk '(a b c))* is equivalent to *(hunk 'a 'b 'c).*

EXAMPLE: *(makhunk 4)* = *{nil nil nil nil}*

(*makhunk 'x_arg)

RETURNS: a hunk of size 2^{x_arg} initialized to EMPTY.

NOTE: This is only to be used by such functions as *hunk* and *makhunk* which create and initialize hunks for users.

2.6.2. hunk accessor

(cxr 'x_ind 'h_hunk)

RETURNS: element x_ind (starting at 0) of hunk h_hunk.

(hunk-to-list 'h_hunk)

RETURNS: a list consisting of the elements of h_hunk.

2.6.3. hunk manipulators

(rplacx 'x_ind 'h_hunk 'g_val)
(*rplacx 'x_ind 'h_hunk 'g_val)

RETURNS: h_hunk

SIDE EFFECT: Element x_ind (starting at 0) of h_hunk is set to g_val.

NOTE: *rplacx* will not modify one of the distinguished (EMPTY) elements whereas **rplacx* will.

hunksize 'h_arg)

RETURNS: the size of the hunk h_arg.

EXAMPLE: *(hunksize (hunk 1 2 3))* = 3

2.7. Bcds

A bcd object contains a pointer to compiled code and to the type of function object the compiled code represents.

(getdisc 'y_bcd**)**
(getentry 'y_bcd**)**

RETURNS: the field of the bcd object given by the function name.

(putdisc 'y_func 's_discipline**)**

RETURNS: s_discipline

SIDE EFFECT: Sets the discipline field of y_func to s_discipline.

2.8. Structures

There are three common structures constructed out of list cells: the assoc list, the property list and the tconc list. The functions below manipulate these structures.

2.8.1. assoc list

An 'assoc list' (or alist) is a common lisp data structure. It has the form
((key1 . value1) (key2 . value2) (key3 . value3) ... (keyn . valuen))

(assoc 'g_arg1 'l_arg2**)**
(assq 'g_arg1 'l_arg2**)**

RETURNS: the first top level element of l_arg2 whose *car* is *equal* (with *assoc*) or *eq* (with *assq*) to g_arg1.

NOTE: Usually l_arg2 has an *a-list* structure and g_arg1 acts as key.

(sassoc 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: the result of *(cond ((assoc 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassoc is written as a macro.

(sassq 'g_arg1 'l_arg2 'sl_func**)**

RETURNS: the result of *(cond ((assq 'g_arg 'l_arg2) (apply 'sl_func nil)))*

NOTE: sassq is written as a macro.

```
; assoc or assq is given a key and an assoc list and returns
; the key and value item if it exists, they differ only in how they test
; for equality of the keys.

-> (setq alist '((alpha . a) ( (complex key) . b) (junk . x)))
((alpha . a) ((complex key) . b) (junk . x))

; we should use assq when the key is an atom
-> (assq 'alpha alist)
(alpha . a)

; but it may not work when the key is a list
-> (assq '(complex key) alist)
nil

; however assoc will always work
-> (assoc '(complex key) alist)
((complex key) . b)
```

(sublis 'l_alst 'l_exp**)**

WHERE: l_alst is an *a-list.*

RETURNS: the list l_exp with every occurrence of key*i* replaced by val*i.*

NOTE: new list structure is returned to prevent modification of l_exp. When a substitution is made, a copy of the value to substitute in is not made.

2.8.2. property list

A property list consists of an alternating sequence of keys and values. Normally a property list is stored on a symbol. A list is a 'disembodied' property list if it contains an odd number of elements, the first of which is ignored.

(plist 's_name**)**

RETURNS: the property list of s_name.

(setplist 's_atm 'l_plist**)**

RETURNS: l_plist.

SIDE EFFECT: the property list of s_atm is set to l_plist.

(get 'ls_name 'g_ind**)**

RETURNS: the value under indicator g_ind in ls_name's property list if ls_name is a symbol.

NOTE: If there is no indicator g_ind in ls_name's property list nil is returned. If ls_name is a list of an odd number of elements then it is a disembodied property list. *get* searches a disembodied property list by starting at its *cdr*, and comparing every other element with g_ind, using *eq*.

(getl 'ls_name 'l_indicators**)**

RETURNS: the property list ls_name beginning at the first indicator which is a member of the list l_indicators, or nil if none of the indicators in l_indicators are on ls_name's property list.

NOTE: If ls_name is a list, then it is assumed to be a disembodied property list.

(putprop 'ls_name 'g_val 'g_ind**)**
(defprop ls_name g_val g_ind**)**

RETURNS: g_val.

SIDE EFFECT: Adds to the property list of ls_name the value g_val under the indicator g_ind.

NOTE: *putprop* evaluates it arguments, *defprop* does not. ls_name may be a disembodied property list, see *get*.

(remprop 'ls_name 'g_ind**)**

RETURNS: the portion of ls_name's property list beginning with the property under the indicator g_ind. If there is no g_ind indicator in ls_name's plist, nil is returned.

SIDE EFFECT: the value under indicator g_ind and g_ind itself is removed from the property list of ls_name.

NOTE: ls_name may be a disembodied property list, see *get*.

```
-> (putprop 'xlate 'a 'alpha)
a
-> (putprop 'xlate 'b 'beta)
b
-> (plist 'xlate)
(alpha a beta b)
-> (get 'xlate 'alpha)
a
; use of a disembodied property list:
-> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

2.8.3. tconc structure

A tconc structure is a special type of list designed to make it easy to add objects to the end. It consists of a list cell whose *car* points to a list of the elements added with *tconc* or *lconc* and whose *cdr* points to the last list cell of the list pointed to by the *car*.

(tconc 'l_ptr 'g_x**)**

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with g_x added to the end.

(lconc 'l_ptr 'l_x**)**

WHERE: l_ptr is a tconc structure.

RETURNS: l_ptr with the list l_x spliced in at the end.

```
; A tconc structure can be initialized in two  ways.
; nil can be given to tconc in which case tconc will generate
; a tconc structure.

->(setq foo (tconc nil 1))
((1) 1)

; Since tconc destructively adds to
; the list, you can now add to foo without using setq again.

->(tconc foo 2)
((1 2) 2)
->foo
((1 2) 2)

; Another way to create a null  tconc structure
; is to use (ncons nil).

->(setq foo (ncons nil))
(nil)
->(tconc foo 1)
((1) 1)

; now see what lconc can do
-> (lconc foo nil)
((1) 1)                    ; no change
-> (lconc foo '(2 3 4))
((1 2 3 4) 4)
```

2.8.4. fclosures

An fclosure is a functional object which admits some data manipulations. They are discussed in §8.4. Internally, they are constructed from vectors.

(fclosure 'l_vars 'g_funobj**)**

WHERE: l_vars is a list of variables, g_funobj is any object that can be funcalled (including, fclosures).

RETURNS: A vector which is the fclosure.

(fclosure-alist 'v_fclosure**)**

RETURNS: An association list representing the variables in the fclosure. This is a snapshot of the current state of the fclosure. If the bindings in the fclosure are changed, any previously calculated results of *fclosure-alist* will not change.

(fclosure-function 'v_fclosure**)**

RETURNS: the functional object part of the fclosure.

(fclosurep 'v_fclosure**)**

RETURNS: t iff the argument is an fclosure.

(symeval-in-fclosure 'v_fclosure 's_symbol**)**

RETURNS: the current binding of a particular symbol in an fclosure.

(set-in-fclosure 'v_fclosure 's_symbol 'g_newvalue**)**

RETURNS: g_newvalue.

SIDE EFFECT: The variable s_symbol is bound in the fclosure to g_newvalue.

2.9. Random functions

The following functions don't fall into any of the classifications above.

(bcdad 's_funcname**)**

RETURNS: a fixnum which is the address in memory where the function s_funcname begins. If s_funcname is not a machine coded function (binary) then *bcdad* returns nil.

(copy 'g_arg**)**

RETURNS: A structure *equal* to g_arg but with new list cells.

(copyint* 'x_arg**)**

RETURNS: a fixnum with the same value as x_arg but in a freshly allocated cell.

(cpy1 'xvt_arg**)**

RETURNS: a new cell of the same type as xvt_arg with the same value as xvt_arg.

(getaddress 's_entry1 's_binder1 'st_discipline1 [... ... ...])

RETURNS: the binary object which s_binder1's function field is set to.

NOTE: This looks in the running lisp's symbol table for a symbol with the same name as s_entry*i*. It then creates a binary object whose entry field points to s_entry*i* and whose discipline is st_discipline*i*. This binary object is stored in the function field of s_binder*i*. If st_discipline*i* is nil, then "subroutine" is used by default. This is especially useful for *cfasl* users.

(macroexpand 'g_form)

RETURNS: g_form after all macros in it are expanded.

NOTE: This function will only macroexpand expressions which could be evaluated and it does not know about the special nlambdas such as *cond* and *do*, thus it misses many macro expansions.

(ptr 'g_arg)

RETURNS: a value cell initialized to point to g_arg.

(quote g_arg)

RETURNS: g_arg.

NOTE: the reader allows you to abbreviate (quote foo) as 'foo.

(kwote 'g_arg)

RETURNS: *(list (quote quote) g_arg)*.

(replace 'g_arg1 'g_arg2)

WHERE: g_arg1 and g_arg2 must be the same type of lispval and not symbols or hunks.

RETURNS: g_arg2.

SIDE EFFECT: The effect of *replace* is dependent on the type of the g_arg*i* although one will notice a similarity in the effects. To understand what *replace* does to fixnum and flonum arguments, you must first understand that such numbers are 'boxed' in FRANZ LISP. What this means is that if the symbol x has a value 32412, then in memory the value element of x's symbol structure contains the address of another word of memory (called a box) with 32412 in it.

Thus, there are two ways of changing the value of x: the first is to change the value element of x's symbol structure to point to a word of memory with a different value. The second way is to change the value in the box which x points to. The former method is used almost all of the time, the latter is used very rarely and has the potential to cause great confusion. The function *replace* allows you to do the latter, i.e., to actually change the value in the box.

You should watch out for these situations. If you do *(setq y x)*, then both x and y will point to the same box. If you now *(replace x 12345)*, then y will also have the value 12345. And, in fact, there may be many other pointers to that box.

Another problem with replacing fixnums is that some boxes are read-only. The fixnums between -1024 and 1023 are stored in a read-only area and attempts to replace them will result in an "Illegal memory reference" error

(see the description of *copyint** for a way around this problem).

For the other valid types, the effect of *replace* is easy to understand. The fields of g_val1's structure are made eq to the corresponding fields of g_val2's structure. For example, if x and y have lists as values then the effect of *(replace x y)* is the same as *(rplaca x (car y))* and *(rplacd x (cdr y))*.

(scons 'x_arg 'bs_rest**)**

WHERE: bs_rest is a bignum or nil.

RETURNS: a bignum whose first bigit is x_arg and whose higher order bigits are bs_rest.

(setf g_refexpr 'g_value**)**

NOTE: *setf* is a generalization of setq. Information may be stored by binding variables, replacing entries of arrays, and vectors, or being put on property lists, among others. Setf will allow the user to store data into some location, by mentioning the operation used to refer to the location. Thus, the first argument may be partially evaluated, but only to the extent needed to calculate a reference. *setf* returns g_value.

```
(setf x 3)       = (setq x 3)
(setf (car x) 3) = (rplaca x 3)
(setf (get foo 'bar) 3) = (putprop foo 3 'bar)
(setf (vref vector index) value) = (vset vector index value)
```

(sort 'l_data 'u_comparefn**)**

RETURNS: a list of the elements of l_data ordered by the comparison function u_comparefn

SIDE EFFECT: the list l_data is modified rather than allocate new storage.

NOTE: *(comparefn 'g_x 'g_y)* should return something non-nil if g-x can precede g_y in sorted order; nil if g_y must precede g_x. If u_comparefn is nil, alphabetical order will be used.

(sortcar 'l_list 'u_comparefn**)**

RETURNS: a list of the elements of l_list with the *car*'s ordered by the sort function u_comparefn.

SIDE EFFECT: the list l_list is modified rather than allocating new storage.

NOTE: Like *sort*, if u_comparefn is nil, alphabetical order will be used.

CHAPTER 3

Arithmetic Functions

This chapter describes FRANZ LISP's functions for doing arithmetic. Often the same function is known by many names, such as *add* which is also *plus*, *sum*, and +. This is due to our desire to be compatible with other Lisps. The FRANZ LISP user is advised to avoid using functions with names such as + and * unless their arguments are fixnums. The Lisp compiler takes advantage of the fact that their arguments are fixnums.

An attempt to divide or to generate a floating point result outside of the range of floating point numbers will cause a floating exception signal from the UNIX operating system. The user can catch and process this interrupt if desired (see the description of the *signal* function).

3.1. simple arithmetic functions

(add ['n_arg1 ...])
(plus ['n_arg1 ...])
(sum ['n_arg1 ...])
(+ ['x_arg1 ...])

RETURNS: the sum of the arguments. If no arguments are given, 0 is returned.

NOTE: if the size of the partial sum exceeds the limit of a fixnum, the partial sum will be converted to a bignum. If any of the arguments are flonums, the partial sum will be converted to a flonum when that argument is processed and the result will thus be a flonum. Currently, if in the process of doing the addition a bignum must be converted into a flonum an error message will result.

(add1 'n_arg)
(1+ 'x_arg)

RETURNS: its argument plus 1.

(diff ['n_arg1 ...])
(difference ['n_arg1 ...])
(– ['x_arg1 ...])

RETURNS: the result of subtracting from n_arg1 all subsequent arguments. If no arguments are given, 0 is returned.

NOTE: See the description of add for details on data type conversions and restrictions.

(sub1 'n_arg)
(1− 'x_arg)

RETURNS: its argument minus 1.

(minus 'n_arg)

RETURNS: zero minus n_arg.

(product ['n_arg1 ...])
(times ['n_arg1 ...])
(* ['x_arg1 ...])

RETURNS: the product of all of its arguments. It returns 1 if there are no arguments.

NOTE: See the description of the function *add* for details and restrictions to the automatic data type coercion.

(quotient ['n_arg1 ...])
(/ ['x_arg1 ...])

RETURNS: the result of dividing the first argument by succeeding ones.

NOTE: If there are no arguments, 1 is returned. See the description of the function *add* for details and restrictions of data type coercion. A divide by zero will cause a floating exception interrupt -- see the description of the *signal* function.

(*quo 'i_x 'i_y)

RETURNS: the integer part of i_x / i_y.

(Divide 'i_dividend 'i_divisor)

RETURNS: a list whose car is the quotient and whose cadr is the remainder of the division of i_dividend by i_divisor.

NOTE: this is restricted to integer division.

(Emuldiv 'x_fact1 'x_fact2 'x_addn 'x_divisor)

RETURNS: a list of the quotient and remainder of this operation: ((x_fact1 * x_fact2) + (sign extended) x_addn) / x_divisor.

NOTE: this is useful for creating a bignum arithmetic package in Lisp.

3.2. predicates

(numberp 'g_arg)

(numbp 'g_arg**)**

RETURNS: t iff g_arg is a number (fixnum, flonum or bignum).

(fixp 'g_arg**)**

RETURNS: t iff g_arg is a fixnum or bignum.

(floatp 'g_arg**)**

RETURNS: t iff g_arg is a flonum.

(evenp 'x_arg**)**

RETURNS: t iff x_arg is even.

(oddp 'x_arg**)**

RETURNS: t iff x_arg is odd.

(zerop 'g_arg**)**

RETURNS: t iff g_arg is a number equal to 0.

(onep 'g_arg**)**

RETURNS: t iff g_arg is a number equal to 1.

(plusp 'n_arg**)**

RETURNS: t iff n_arg is greater than zero.

(minusp 'g_arg**)**

RETURNS: t iff g_arg is a negative number.

(greaterp ['n_arg1 ...])
(> 'fx_arg1 'fx_arg2**)**
(>& 'x_arg1 'x_arg2**)**

RETURNS: t iff the arguments are in a strictly decreasing order.

NOTE: In functions *greaterp* and > the function *difference* is used to compare adjacent values. If any of the arguments are non-numbers, the error message will come from the *difference* function. The arguments to > must be fixnums or both flonums. The arguments to >& must both be fixnums.

(lessp ['n_arg1 ...])
(< 'fx_arg1 'fx_arg2**)**
(<& 'x_arg1 'x_arg2**)**

RETURNS: t iff the arguments are in a strictly increasing order.

NOTE: In functions *lessp* and < the function *difference* is used to compare adjacent values. If any of the arguments are non numbers, the error message will come from the *difference* function. The arguments to < may be either fixnums or flonums but must be the same type. The arguments to <& must be fixnums.

(= 'fx_arg1 'fx_arg2)

(=& 'x_arg1 'x_arg2)

RETURNS: t iff the arguments have the same value. The arguments to = must be the either both fixnums or both flonums. The arguments to =& must be fixnums.

3.3. trignometric functions

(cos 'fx_angle)

RETURNS: the (flonum) cosine of fx_angle (which is assumed to be in radians).

(sin 'fx_angle)

RETURNS: the sine of fx_angle (which is assumed to be in radians).

(acos 'fx_arg)

RETURNS: the (flonum) arc cosine of fx_arg in the range 0 to π.

(asin 'fx_arg)

RETURNS: the (flonum) arc sine of fx_arg in the range $-\pi/2$ to $\pi/2$.

(atan 'fx_arg1 'fx_arg2)

RETURNS: the (flonum) arc tangent of fx_arg1/fx_arg2 in the range $-\pi$ to π.

3.4. bignum functions

(haipart bx_number x_bits**)**

RETURNS: a fixnum (or bignum) which contains the x_bits high bits of *(abs bx_number)* if x_bits is positive, otherwise it returns the *(abs x_bits)* low bits of *(abs bx_number).*

(haulong bx_number**)**

RETURNS: the number of significant bits in bx_number.

NOTE: the result is equal to the least integer greater to or equal to the base two logarithm of one plus the absolute value of bx_number.

(bignum-leftshift bx_arg x_amount**)**

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg will be shifted right by the magnitude of x_amount.

NOTE: If bx_arg is shifted right, it will be rounded to the nearest even number.

(sticky-bignum-leftshift 'bx_arg 'x_amount**)**

RETURNS: bx_arg shifted left by x_amount. If x_amount is negative, bx_arg will be shifted right by the magnitude of x_amount and rounded.

NOTE: sticky rounding is done this way: after shifting, the low order bit is changed to 1 if any 1's were shifted off to the right.

3.5. bit manipulation

(boole 'x_key 'x_v1 'x_v2 ...**)**

RETURNS: the result of the bitwise boolean operation as described in the following table.

NOTE: If there are more than 3 arguments, then evaluation proceeds left to right with each partial result becoming the new value of x_v1. That is,
(boole 'key 'v1 'v2 'v3) ≡ *(boole 'key (boole 'key 'v1 'v2) 'v3).*
In the following table, * represents bitwise and, + represents bitwise or, ⊕ represents bitwise xor and ¬ represents bitwise negation and is the highest precedence operator.

(boole 'key 'x 'y)								
key	0	1	2	3	4	5	6	7
result	0	x * y	¬ x * y	y	x * ¬ y	x	x ⊕ y	x + y
common names		and			bitclear		xor	or
key	8	9	10	11	12	13	14	15
result	¬ (x + y)	¬(x ⊕ y)	¬ x	¬ x + y	¬ y	x + ¬ y	¬ x + ¬ y	-1
common names	nor	equiv		implies			nand	

(lsh 'x_val 'x_amt**)**

RETURNS: x_val shifted left by x_amt if x_amt is positive. If x_amt is negative, then *lsh* returns x_val shifted right by the magnitude if x_amt.

NOTE: This always returns a fixnum even for those numbers whose magnitude is so large that they would normally be represented as a bignum, i.e. shifter bits are lost. For more general bit shifters, see *bignum-leftshift* and *sticky-bignum-leftshift.*

(rot 'x_val 'x_amt**)**

RETURNS: x_val rotated left by x_amt if x_amt is positive. If x_amt is negative, then x_val is rotated right by the magnitude of x_amt.

3.6. other functions

(abs 'n_arg**)**
(absval 'n_arg**)**

RETURNS: the absolute value of n_arg.

(exp 'fx_arg**)**

RETURNS: *e* raised to the fx_arg power (flonum) .

(expt 'n_base 'n_power**)**

RETURNS: n_base raised to the n_power power.

NOTE: if either of the arguments are flonums, the calculation will be done using *log* and *exp*.

(fact 'x_arg**)**

RETURNS: x_arg factorial. (fixnum or bignum)

(fix 'n_arg**)**

RETURNS: a fixnum as close as we can get to n_arg.

NOTE: *fix* will round down. Currently, if n_arg is a flonum larger than the size of a fixnum, this will fail.

(float 'n_arg**)**

RETURNS: a flonum as close as we can get to n_arg.

NOTE: if n_arg is a bignum larger than the maximum size of a flonum, then a floating exception will occur.

(log 'fx_arg**)**

RETURNS: the natural logarithm of fx_arg.

(max 'n_arg1 ... **)**

RETURNS: the maximum value in the list of arguments.

(min 'n_arg1 ... **)**

RETURNS: the minimum value in the list of arguments.

(mod 'i_dividend 'i_divisor**)**
(remainder 'i_dividend 'i_divisor**)**

RETURNS: the remainder when i_dividend is divided by i_divisor.

NOTE: The sign of the result will have the same sign as i_dividend.

(*mod 'x_dividend 'x_divisor**)**

RETURNS: the balanced representation of x_dividend modulo x_divisor.

NOTE: the range of the balanced representation is abs(x_divisor)/2 to (abs(x_divisor)/2) − x_divisor + 1.

(random ['x_limit]**)**

RETURNS: a fixnum between 0 and x_limit − 1 if x_limit is given. If x_limit is not given, any fixnum, positive or negative, might be returned.

(sqrt 'fx_arg**)**

RETURNS: the square root of fx_arg.

CHAPTER 4

Special Functions

(and [g_arg1 ...])

RETURNS: the value of the last argument if all arguments evaluate to a non-nil value, otherwise *and* returns nil. It returns t if there are no arguments.

NOTE: the arguments are evaluated left to right and evaluation will cease with the first nil encountered

(apply 'u_func 'l_args)

RETURNS: the result of applying function u_func to the arguments in the list l_args.

NOTE: If u_func is a lambda, then the *(length l_args)* should equal the number of formal parameters for the u_func. If u_func is a nlambda or macro, then l_args is bound to the single formal parameter.

```
; add1 is a lambda of 1 argument
-> (apply 'add1 '(3))
4

; we will define plus1 as a macro which will be equivalent to add1
-> (def plus1 (macro (arg) (list 'add1 (cadr arg))))
plus1
-> (plus1 3)
4

; now if we apply a macro we obtain the form it changes to.
-> (apply 'plus1 '(plus1 3))
(add1 3)

; if we funcall a macro however, the result of the macro is evaled
; before it is returned.
-> (funcall 'plus1 '(plus1 3))
4

; for this particular macro, the car of the arg is not checked
; so that this too will work
-> (apply 'plus1 '(foo 3))
(add1 3)
```

(arg ['x_numb])

RETURNS: if x_numb is specified then the x_numb'*th* argument to the enclosing lexpr If x_numb is not specified then this returns the number of arguments to the enclosing lexpr.

NOTE: it is an error to the interpreter if x_numb is given and out of range.

(break [g_message ['g_pred]])

WHERE: if g_message is not given it is assumed to be the null string, and if g_pred is not given it is assumed to be t.

RETURNS: the value of *(*break 'g_pred 'g_message)*

(*break 'g_pred 'g_message)

RETURNS: nil immediately if g_pred is nil, else the value of the next (return 'value) expression typed in at top level.

SIDE EFFECT: If the predicate, g_pred, evaluates to non-null, the lisp system stops and prints out 'Break ' followed by g_message. It then enters a break loop which allows one to interactively debug a program. To continue execution from a break you can use the *return* function. to return to top level or another break level, you can use *retbrk* or *reset.*

(caseq 'g_key-form l_clause1 ...)

WHERE: l_clause*i* is a list of the form (g_comparator ['g_form*i* ...]). The comparators may be symbols, small fixnums, a list of small fixnums or symbols.

NOTE: The way caseq works is that it evaluates g_key-form, yielding a value we will call the selector. Each clause is examined until the selector is found consistent with the comparator. For a symbol, or a fixnum, this means the two must be *eq.* For a list, this means that the selector must be *eq* to some element of the list.

The symbol t has special semantics: it matches anything, and consequently, should be the last comparator. Then, having chosen a clause, *caseq* evaluates each form within that clause and

RETURNS: the value of the last form. If no comparators are matched, *caseq* returns nil.

Here are two ways of defining the same function:

```
-> (defun fate (personna)
       (caseq personna
         (cow '(jumped over the moon))
         (cat '(played nero))
         ((dish spoon) '(ran away together))
         (t '(lived happily ever after))))
fate
-> (defun fate (personna)
       (cond
           ((eq personna 'cow) '(jumped over the moon))
           ((eq personna 'cat) '(played nero))
           ((memq personna '(dish spoon)) '(ran away together))
           (t '(lived happily ever after))))
fate
```

(catch g_exp [ls_tag])

WHERE: if ls_tag is not given, it is assumed to be nil.

RETURNS: the result of *(*catch 'ls_tag g_exp)*

NOTE: catch is defined as a macro.

(*catch 'ls_tag g_exp)

WHERE: ls_tag is either a symbol or a list of symbols.

RETURNS: the result of evaluating g_exp or the value thrown during the evaluation of g_exp.

SIDE EFFECT: this first sets up a 'catch frame' on the lisp runtime stack. Then it begins to evaluate g_exp. If g_exp evaluates normally, its value is returned. If, however, a value is thrown during the evaluation of g_exp then this *catch will return with that value if one of these cases is true:

(1) the tag thrown to is ls_tag

(2) ls_tag is a list and the tag thrown to is a member of this list

(3) ls_tag is nil.

NOTE: Errors are implemented as a special kind of throw. A catch with no tag will not catch an error but a catch whose tag is the error type will catch that type of error. See Chapter 10 for more information.

(comment [g_arg ...])

RETURNS: the symbol comment.

NOTE: This does absolutely nothing.

(cond [l_clause1 ...])

RETURNS: the last value evaluated in the first clause satisfied. If no clauses are satisfied then nil is returned.

NOTE: This is the basic conditional 'statement' in lisp. The clauses are processed from left to right. The first element of a clause is evaluated. If it evaluated to a non-null value then that clause is satisfied and all following elements of that clause are evaluated. The last value computed is returned as the value of the cond. If there is just one element in the clause then its value is returned. If the first element of a clause evaluates to nil, then the other elements of that clause are not evaluated and the system moves to the next clause.

(cvttointlisp)

SIDE EFFECT: The reader is modified to conform with the Interlisp syntax. The character % is made the escape character and special meanings for comma, backquote and backslash are removed. Also the reader is told to convert upper case to lower case.

(cvttofranzlisp)

SIDE EFFECT: The reader is modified to conform with franz's default syntax. One would run this function after having run cvttomaclisp, only. Backslash is made the escape character, and super-brackets are reinstated. The reader is reminded to distinguish between upper and lower case.

(cvttomaclisp)

SIDE EFFECT: The reader is modified to conform with Maclisp syntax. The character / is made the escape character and the special meanings for backslash, left and right bracket are removed. The reader is made case-insensitive.

(cvttoucilisp)

SIDE EFFECT: The reader is modified to conform with UCI Lisp syntax. The character / is made the escape character, tilde is made the comment character, exclamation point takes on the unquote function normally held by comma, and backslash, comma, semicolon become normal characters. Here too, the reader is made case-insensitive.

(debug s_msg**)**

SIDE EFFECT: Enter the Fixit package described in Chapter 15. This package allows you to examine the evaluation stack in detail. To leave the Fixit package type 'ok'.

(debugging 'g_arg**)**

SIDE EFFECT: If g_arg is non-null, Franz unlinks the transfer tables, does a *(*rset t)* to turn on evaluation monitoring and sets the all-error catcher (ER%all) to be *debug-err-handler.* If g_arg is nil, all of the above changes are undone.

(declare [g_arg ...]**)**

RETURNS: nil

NOTE: this is a no-op to the evaluator. It has special meaning to the compiler (see Chapter 12).

(def s_name (s_type l_argl g_expl ...))

WHERE: s_type is one of lambda, nlambda, macro or lexpr.

RETURNS: s_name

SIDE EFFECT: This defines the function s_name to the lisp system. If s_type is nlambda or macro then the argument list l_argl must contain exactly one non-nil symbol.

(defmacro s_name l_arg g_expl ...)
(defcmacro s_name l_arg g_expl ...)

RETURNS: s_name

SIDE EFFECT: This defines the macro s_name. *defmacro* makes it easy to write macros since it makes the syntax just like *defun.* Further information on *defmacro* is in §8.3.2. *defcmacro* defines compiler-only macros, or cmacros. A cmacro is stored on the property list of a symbol under the indicator **cmacro**. Thus a function can have a normal definition and a cmacro definition. For an example of the use of cmacros, see the definitions of nthcdr and nth in /usr/lib/lisp/common2.l

(defun s_name [s_mtype] ls_arg1 g_exp1 ...)

WHERE: s_mtype is one of fexpr, expr, args or macro.

RETURNS: s_name

SIDE EFFECT: This defines the function s_name.

NOTE: this exists for Maclisp compatibility, it is just a macro which changes the defun form to the def form. An s_mtype of fexpr is converted to nlambda and of expr to lambda. Macro remains the same. If ls_arg1 is a non-nil symbol, then the type is assumed to be lexpr and ls_arg1 is the symbol which is bound to the number of args when the function is entered.
For compatability with the Lisp Machine lisp, there are three types of optional parameters that can occur in ls_argl: *&optional* declares that the following symbols are optional, and may or may not appear in the argument list to the function, *&rest symbol* declares that all forms in the function call that are not accounted for by previous lambda bindings are to be assigned to *symbol*, and *&aux form1 ... formn* declares that the *formi* are either symbols, in which case they are lambda bound to **nil**, or lists, in which case the first element of the list is lambda bound to the second, evaluated element.

```
; def and defun here are used to define identical functions
; you can decide for yourself which is easier to use.
-> (def append1 (lambda (lis extra) (append lis (list extra))))
append1

-> (defun append1 (lis extra) (append lis (list extra)))
append1

; Using the & forms...
-> (defun test (a b &optional c &aux (retval 0) &rest z)
        (if c then (msg "Optional arg present" N
                        "c is " c N))
        (msg "rest is " z N
             "retval is " retval N))
test
-> (test 1 2 3 4)
Optional arg present
c is 3
rest is (4)
retval is 0
```

(defvar s_variable ['g_init])

RETURNS: s_variable.

NOTE: This form is put at the top level in files, like *defun*.

SIDE EFFECT: This declares s_variable to be special. If g_init is present, and s_variable is unbound when the file is read in, s_variable will be set to the value of g_init. An advantage of '(defvar foo)' over '(declare (special foo))' is that if a file containing defvars is loaded (or fasl'ed) in during compilation, the variables mentioned in the defvar's will be declared special. The only way to have that effect with '(declare (special foo))' is to *include* the file.

(do l_vrbs l_test g_exp1 ...**)**

RETURNS: the last form in the cdr of l_test evaluated, or a value explicitly given by a return evaluated within the do body.

NOTE: This is the basic iteration form for FRANZ LISP. l_vrbs is a list of zero or more var-init-repeat forms. A var-init-repeat form looks like:

(s_name [g_init [g_repeat]])

There are three cases depending on what is present in the form. If just s_name is present, this means that when the do is entered, s_name is lambda-bound to nil and is never modified by the system (though the program is certainly free to modify its value). If the form is (s_name 'g_init) then the only difference is that s_name is lambda-bound to the value of g_init instead of nil. If g_repeat is also present then s_name is lambda-bound to g_init when the loop is entered and after each pass through the do body s_name is bound to the value of g_repeat.

l_test is either nil or has the form of a cond clause. If it is nil then the do body will be evaluated only once and the do will return nil. Otherwise, before the do body is evaluated the car of l_test is evaluated and if the result is non-null, this signals an end to the looping. Then the rest of the forms in l_test are evaluated and the value of the last one is returned as the value of the do. If the cdr of l_test is nil, then nil is returned -- thus this is not exactly like a cond clause.

g_exp1 and those forms which follow constitute the do body. A do body is like a prog body and thus may have labels and one may use the functions go and return.

The sequence of evaluations is this:

(1) the init forms are evaluated left to right and stored in temporary locations.

(2) Simultaneously all do variables are lambda bound to the value of their init forms or nil.

(3) If l_test is non-null, then the car is evaluated and if it is non-null, the rest of the forms in l_test are evaluated and the last value is returned as the value of the do.

(4) The forms in the do body are evaluated left to right.

(5) If l_test is nil the do function returns with the value nil.

(6) The repeat forms are evaluated and saved in temporary locations.

(7) The variables with repeat forms are simultaneously bound to the values of those forms.

(8) Go to step 3.

NOTE: there is an alternate form of do which can be used when there is only one do variable. It is described next.

```
; this is  a simple function which numbers the elements of a list.
; It uses a do function with two local variables.
-> (defun printem (lis)
            (do ((xx lis (cdr xx))
                 (i 1 (1+ i)))
                ((null xx) (patom "all done") (terpr))
                (print i)
                (patom ": ")
                (print (car xx))
                (terpr)))
printem
-> (printem '(a b c d))
1: a
2: b
3: c
4: d
all done
nil
->
```

(do s_name g_init g_repeat g_test g_exp1 ...**)**

NOTE: this is another, less general, form of do. It is evaluated by:

(1) evaluating g_init

(2) lambda binding s_name to value of g_init

(3) g_test is evaluated and if it is not nil the do function returns with nil.

(4) the do body is evaluated beginning at g_exp1.

(5) the repeat form is evaluated and stored in s_name.

(6) go to step 3.

RETURNS: nil

(environment [l_when1 l_what1 l_when2 l_what2 ...]**)**
(environment-maclisp [l_when1 l_what1 l_when2 l_what2 ...]**)**
(environment-lmlisp [l_when1 l_what1 l_when2 l_what2 ...]**)**

WHERE: the when's are a subset of (eval compile load), and the symbols have the same meaning as they do in 'eval-when'.

The what's may be
(files file1 file2 ... fileN),
which insure that the named files are loaded. To see if file*i* is loaded, it looks for a 'version' property under file*i*'s property list. Thus to prevent multiple loading, you should put
(putprop 'myfile t 'version),
at the end of myfile.l.

Another acceptible form for a what is
(syntax type)
Where type is either maclisp, intlisp, ucilisp, franzlisp. This sets the syntax correctly.

environment-maclisp sets the environment to that which 'liszt -m' would generate. *environment-lmlisp* sets up the lisp machine environment. This is like maclisp but it has additional macros. For these specialized environments, only the **files** clauses are useful. (environment-maclisp (compile eval) (files foo bar))

(err ['s_value [nil]])

RETURNS: nothing (it never returns).

SIDE EFFECT: This causes an error and if this error is caught by an *errset* then that *errset* will return s_value instead of nil. If the second arg is given, then it must be nil (MACLisp compatibility).

(error ['s_message1 ['s_message2]])

RETURNS: nothing (it never returns).

SIDE EFFECT: s_message1 and s_message2 are *patom*ed if they are given and then *err* is called (with no arguments), which causes an error.

(errset g_expr [s_flag])

RETURNS: a list of one element, which is the value resulting from evaluating g_expr. If an error occurs during the evaluation of g_expr, then the locus of control will return to the *errset* which will then return nil (unless the error was caused by a call to *err*, with a non-null argument).

SIDE EFFECT: S_flag is evaluated before g_expr is evaluated. If s_flag is not given, then it is assumed to be t. If an error occurs during the evaluation of g_expr, and s_flag evaluated to a non-null value, then the error message associated with the error is printed before control returns to the errset.

(eval 'g_val ['x_bind-pointer])

RETURNS: the result of evaluating g_val.

NOTE: The evaluator evaluates g_val in this way:
If g_val is a symbol, then the evaluator returns its value. If g_val had never been assigned a value, then this causes an 'Unbound Variable' error. If x_bind-pointer is given, then the variable is evaluated with respect to that pointer (see *evalframe* for details on bind-pointers).

If g_val is of type value, then its value is returned. If g_val is of any other type than list, g_val is returned.

If g_val is a list object then g_val is either a function call or array reference. Let g_car be the first element of g_val. We continually evaluate g_car until we end up with a symbol with a non-null function binding or a non-symbol. Call what we end up with: g_func.

G_func must be one of three types: list, binary or array. If it is a list then the first element of the list, which we shall call g_functype, must be either lambda, nlambda, macro or lexpr. If g_func is a binary, then its discipline, which we shall call g_functype, is either lambda, nlambda, macro or a string. If g_func is an array then this form is evaluated specially, see Chapter 9 on arrays. If g_func is a list or binary, then g_functype will determine how the arguments to this function, the cdr of g_val, are processed. If g_functype is a string, then this is a foreign function call (see §8.5 for more details).

If g_functype is lambda or lexpr, the arguments are evaluated (by calling *eval* recursively) and stacked. If g_functype is nlambda then the argument list is stacked. If g_functype is macro then the entire form, g_val is stacked.

Next, the formal variables are lambda bound. The formal variables are the cadr of g_func. If g_functype is nlambda, lexpr or macro, there should only be one formal variable. The values on the stack are lambda bound to the formal variables except in the case of a lexpr, where the number of actual arguments is bound to the formal variable.

After the binding is done, the function is invoked, either by jumping to the entry point in the case of a binary or by evaluating the list of forms beginning at cddr g_func. The result of this function invocation is returned as the value of the call to eval.

(evalframe 'x_pdlpointer**)**

RETURNS: an evalframe descriptor for the evaluation frame just before x_pdlpointer. If x_pdlpointer is nil, it returns the evaluation frame of the frame just before the current call to *evalframe.*

NOTE: An evalframe descriptor describes a call to *eval, apply* or *funcall.* The form of the descriptor is
(type pdl-pointer expression bind-pointer np-index lbot-index)
where type is 'eval' if this describes a call to *eval* or 'apply' if this is a call to *apply* or *funcall.* pdl-pointer is a number which describes this context. It can be passed to *evalframe* to obtain the next descriptor and can be passed to *freturn* to cause a return from this context. bind-pointer is the size of variable binding stack when this evaluation began. The bind-pointer can be given as a second argument to *eval* to order to evaluate variables in the same context as this evaluation. If type is 'eval' then expression will have the form *(function-name arg1 ...).* If type is 'apply' then expression will have the form *(function-name (arg1 ...)).* np-index and lbot-index are pointers into the argument stack (also known as the *namestack* array) at the time of call. lbot-index points to the first argument, np-index points one beyond the last argument.
In order for there to be enough information for *evalframe* to return, you must call *(*rset t).*

EXAMPLE: *(progn (evalframe nil))*
returns *(eval 2147478600 (progn (evalframe nil)) 1 8 7)*

(evalhook 'g_form 'su_evalfunc ['su_funcallfunc]**)**

RETURNS: the result of evaluating g_form after lambda binding 'evalhook' to su_evalfunc and, if it is given, lambda binding 'funcallhook' to su_funcallhook.

NOTE: As explained in §14.4, the function *eval* may pass the job of evaluating a form to a user 'hook' function when various switches are set. The hook function normally prints the form to be evaluated on the terminal and then evaluates it by calling *evalhook. Evalhook* does the lambda binding mentioned above and then calls *eval* to evaluate the form after setting an internal switch to tell *eval* not to call the user's hook function just this one time. This allows the evaluation process to advance one step and yet insure that further calls to *eval* will cause traps to the hook function (if su_evalfunc is non-null).
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously.

(exec s_arg1 ...**)**

RETURNS: the result of forking and executing the command named by concatenating the s_arg*i* together with spaces in between.

(exece 's_fname ['l_args ['l_envir]]**)**

RETURNS: the error code from the system if it was unable to execute the command s_fname with arguments l_args and with the environment set up as specified in l_envir. If this function is successful, it will not return, instead the lisp system will be overlaid by the new command.

(freturn 'x_pdl-pointer 'g_retval**)**

RETURNS: g_retval from the context given by x_pdl-pointer.

NOTE: A pdl-pointer denotes a certain expression currently being evaluated. The pdl-pointer for a given expression can be obtained from *evalframe.*

(frexp 'f_arg**)**

RETURNS: a list cell *(exponent . mantissa)* which represents the given flonum

NOTE: The exponent will be a fixnum, the mantissa a 56 bit bignum. If you think of the the binary point occurring right after the high order bit of mantissa, then f_arg = 2^{exponent} * mantissa.

(funcall 'u_func ['g_arg1 ...]**)**

RETURNS: the value of applying function u_func to the arguments g_arg*i* and then evaluating that result if u_func is a macro.

NOTE: If u_func is a macro or nlambda then there should be only one g_arg. *funcall* is the function which the evaluator uses to evaluate lists. If *foo* is a lambda or lexpr or array, then *(funcall 'foo 'a 'b 'c)* is equivalent to *(foo 'a 'b 'c).* If *foo* is a nlambda then *(funcall 'foo '(a b c))* is equivalent to *(foo a b c).* Finally, if *foo* is a macro then *(funcall 'foo '(foo a b c))* is equivalent to *(foo a b c).*

(funcallhook 'l_form 'su_funcallfunc ['su_evalfunc]**)**

RETURNS: the result of *funcall*ing the *(car l_form)* on the already evaluated arguments in the *(cdr l_form)* after lambda binding 'funcallhook' to su_funcallfunc and, if it is given, lambda binding 'evalhook' to su_evalhook.

NOTE: This function is designed to continue the evaluation process with as little work as possible after a funcallhook trap has occurred. It is for this reason that the form of l_form is unorthodox: its *car* is the name of the function to call and its *cdr* are a list of arguments to stack (without evaluating again) before calling the given function. After stacking the arguments but before calling *funcall* an internal switch is set to prevent *funcall* from passing the job of funcalling to su_funcallfunc. If *funcall* is called recursively in funcalling l_form and if su_funcallfunc is non-null, then the arguments to *funcall* will actually be given to su_funcallfunc (a lexpr) to be funcalled.
In order for *evalhook* to work, *(*rset t)* and *(sstatus evalhook t)* must have been done previously. A more detailed description of *evalhook* and *funcallhook* is given in Chapter 14.

(function u_func**)**

RETURNS: the function binding of u_func if it is an symbol with a function binding otherwise u_func is returned.

(getdisc 'y_func**)**

RETURNS: the discipline of the machine coded function (either lambda, nlambda or macro).

(go g_labexp**)**

WHERE: g_labexp is either a symbol or an expression.

SIDE EFFECT: If g_labexp is an expression, that expression is evaluated and should result in a symbol. The locus of control moves to just following the symbol g_labexp in the current prog or do body.

NOTE: this is only valid in the context of a prog or do body. The interpreter and compiler will allow non-local *go*'s although the compiler won't allow a *go* to leave a function body. The compiler will not allow g_labexp to be an expression.

(if 'g_a 'g_b**)**
(if 'g_a 'g_b 'g_c ...**)**
(if 'g_a **then** 'g_b [...] [**elseif** 'g_c **then** 'g_d ...] [**else** 'g_e [...])
(if 'g_a **then** 'g_b [...] [**elseif** 'g_c **thenret**] [**else** 'g_d [...])

NOTE: The various forms of *if* are intended to be a more readable conditional statement, to be used in place of *cond*. There are two varieties of *if*, with keywords, and without. The keyword-less variety is inherited from common Maclisp usage. A keyword-less, two argument *if* is equivalent to a one-clause *cond*, i.e. (*cond* (a b)). Any other keyword-less *if* must have at least three arguments. The first two arguments are the first clause of the equivalent *cond*, and all remaining arguments are shoved into a second clause beginning with t. Thus, the second form of *if* is equivalent to

(*cond* (a b) (t c ...)).

The keyword variety has the following grouping of arguments: a predicate, a then-clause, and optional else-clause. The predicate is evaluated, and if the result is non-nil, the then-clause will be performed, in the sense described below. Otherwise, (i.e. the result of the predicate evaluation was precisely nil), the else-clause will be performed.

Then-clauses will either consist entirely of the single keyword **thenret**, or will start with the keyword **then**, and be followed by at least one general expression. (These general expressions must not be one of the keywords.) To actuate a **thenret** means to cease further evaluation of the *if*, and to return the value of the predicate just calculated. The performance of the longer clause means to evaluate each general expression in turn, and then return the last value calculated.

The else-clause may begin with the keyword **else** and be followed by at least one general expression. The rendition of this clause is just like that of a then-clause. An else-clause may begin alternatively with the keyword **elseif**, and be followed (recursively) by a predicate, then-clause, and optional else-clause. Evaluation of this clause, is just evaluation of an *if*-form, with the same predicate, then- and else-clauses.

(I-throw-err 'l_token**)**

WHERE: l_token is the *cdr* of the value returned from a **catch* with the tag ER%unwind-protect.

RETURNS: nothing (never returns in the current context)

SIDE EFFECT: The error or throw denoted by l_token is continued.

NOTE: This function is used to implement *unwind-protect* which allows the processing of a transfer of control though a certain context to be interrupted, a user function to be executed and than the transfer of control to continue. The form of l_token is either
(t tag value) for a throw or
(nil type message valret contuab uniqueid [arg ...]) for an error.
This function is not to be used for implementing throws or errors and is only documented here for completeness.

(let l_args g_exp1 ... g_exprn**)**

RETURNS: the result of evaluating g_exprn within the bindings given by l_args.

NOTE: l_args is either nil (in which case *let* is just like *progn*) or it is a list of binding objects. A binding object is a list *(symbol expression)*. When a *let* is entered all of the expressions are evaluated and then simultaneously lambda bound to the corresponding symbols. In effect, a *let* expression is just like a lambda expression except the symbols and their initial values are next to each other which makes the expression easier to understand. There are some added features to the *let* expression: A binding object can just be a symbol, in which case the expression corresponding to that symbol is 'nil'. If a binding object is a list and the first element of that list is another list, then that list is assumed to be a binding template and *let* will do a *desetq* on it.

(let* l_args g_exp1 ... g_expn**)**

RETURNS: the result of evaluating g_exprn within the bindings given by l_args.

NOTE: This is identical to *let* except the expressions in the binding list l_args are evaluated and bound sequentially instead of in parallel.

(lexpr-funcall 'g_function ['g_arg1 ...] 'l_argn**)**

NOTE: This is a cross between funcall and apply. The last argument, must be a list (possibly empty). The element of list arg are stack and then the function is funcalled.

EXAMPLE: (lexpr-funcall 'list 'a '(b c d)) is the same as
(funcall 'list 'a 'b 'c 'd)

(listify 'x_count**)**

RETURNS: a list of x_count of the arguments to the current function (which must be a lexpr).

NOTE: normally arguments 1 through x_count are returned. If x_count is negative then a list of last abs(x_count) arguments are returned.

(map 'u_func 'l_arg1 ...**)**

RETURNS: l_arg1

NOTE: The function u_func is applied to successive sublists of the l_arg*i*. All sublists should have the same length.

(mapc 'u_func 'l_arg1 ...**)**

RETURNS: l_arg1.

NOTE: The function u_func is applied to successive elements of the argument lists. All of the lists should have the same length.

(mapcan 'u_func 'l_arg1 ...**)**

RETURNS: nconc applied to the results of the functional evaluations.

NOTE: The function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcar 'u_func 'l_arg1 ...**)**

RETURNS: a list of the values returned from the functional application.

NOTE: the function u_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcon 'u_func 'l_arg1 ...**)**

RETURNS: nconc applied to the results of the functional evaluation.

NOTE: the function u_func is applied to successive sublists of the argument lists. All sublists should have the same length.

(maplist 'u_func 'l_arg1 ...**)**

RETURNS: a list of the results of the functional evaluations.

NOTE: the function u_func is applied to successive sublists of the arguments lists. All sublists should have the same length.

Readers may find the following summary table useful in remembering the differences between the six mapping functions:

	Value returned is		
Argument to functional is	l_arg1	list of results	*nconc* of results
elements of list	mapc	mapcar	mapcan
sublists	map	maplist	mapcon

(mfunction t_entry 's_disc**)**

RETURNS: a lisp object of type binary composed of t_entry and s_disc.

NOTE: t_entry is a pointer to the machine code for a function, and s_disc is the discipline (e.g. lambda).

(oblist)

RETURNS: a list of all symbols on the oblist.

(or [g_arg1 ...]**)**

RETURNS: the value of the first non-null argument or nil if all arguments evaluate to nil.

NOTE: Evaluation proceeds left to right and stops as soon as one of the arguments evaluates to a non-null value.

(prog l_vrbls g_exp1 ...**)**

RETURNS: the value explicitly given in a return form or else nil if no return is done by the time the last g_exp*i* is evaluated.

NOTE: the local variables are lambda bound to nil then the g_exp are evaluated from left to right. This is a prog body (obviously) and this means than any symbols seen are not evaluated, instead they are treated as labels. This also means that return's and go's are allowed.

(prog1 'g_exp1 ['g_exp2 ...]**)**

RETURNS: g_exp1

(prog2 'g_exp1 'g_exp2 ['g_exp3 ...]**)**

RETURNS: g_exp2

NOTE: the forms are evaluated from left to right and the value of g_exp2 is returned.

(progn 'g_exp1 ['g_exp2 ...]**)**

RETURNS: the last g_exp*i*.

(progv 'l_locv 'l_initv g_exp1 ...**)**

WHERE: l_locv is a list of symbols and l_initv is a list of expressions.

RETURNS: the value of the last g_exp*i* evaluated.

NOTE: The expressions in l_initv are evaluated from left to right and then lambda-bound to the symbols in l_locv. If there are too few expressions in l_initv then the missing values are assumed to be nil. If there are too many expressions in l_initv then the extra ones are ignored (although they are evaluated). Then the g_exp*i* are evaluated left to right. The body of a progv is like the body of a progn, it is *not* a prog body. (C.f. *let*)

(purcopy 'g_exp**)**

RETURNS: a copy of g_exp with new pure cells allocated wherever possible.

NOTE: pure space is never swept up by the garbage collector, so this should only be done on expressions which are not likely to become garbage in the future. In certain cases, data objects in pure space become read-only after a *dumplisp* and then an attempt to modify the object will result in an illegal memory reference.

(purep 'g_exp**)**

RETURNS: t iff the object g_exp is in pure space.

(putd 's_name 'u_func**)**

RETURNS: u_func

SIDE EFFECT: this sets the function binding of symbol s_name to u_func.

(return ['g_val]**)**

RETURNS: g_val (or nil if g_val is not present) from the enclosing prog or do body.

NOTE: this form is only valid in the context of a prog or do body.

(selectq 'g_key-form [l_clause1 ...]**)**

NOTE: This function is just like *caseq* (see above), except that the symbol **otherwise** has the same semantics as the symbol **t**, when used as a comparator.

(setarg 'x_argnum 'g_val**)**

WHERE: x_argnum is greater than zero and less than or equal to the number of arguments to the lexpr.

RETURNS: g_val

SIDE EFFECT: the lexpr's x_argnum'th argument is set to g-val.

NOTE: this can only be used within the body of a lexpr.

(throw 'g_val [s_tag]**)**

WHERE: if s_tag is not given, it is assumed to be nil.

RETURNS: the value of *(*throw 's_tag 'g_val)*.

(*throw 's_tag 'g_val**)**

RETURNS: g_val from the first enclosing catch with the tag s_tag or with no tag at all.

NOTE: this is used in conjunction with **catch* to cause a clean jump to an enclosing context.

(unwind-protect g_protected [g_cleanup1 ...]**)**

RETURNS: the result of evaluating g_protected.

NOTE: Normally g_protected is evaluated and its value remembered, then the g_cleanup*i* are evaluated and finally the saved value of g_protected is returned. If something should happen when evaluating g_protected which causes control to pass through g_protected and thus through the call to the unwind-protect, then the g_cleanup*i* will still be evaluated. This is useful if g_protected does something sensitive which must be cleaned up whether or not g_protected completes.

CHAPTER 5

Input/Output

The following functions are used to read from and write to external devices (e.g. files) and programs (through pipes). All I/O goes through the lisp data type called the port. A port may be open for either reading or writing, but usually not both simultaneously (see *fileopen*). There are only a limited number of ports (20) and they will not be reclaimed unless they are *close*d. All ports are reclaimed by a *resetio* call, but this drastic step won't be necessary if the program closes what it uses.

If a port argument is not supplied to a function which requires one or if a bad port argument (such as nil) is given, then FRANZ LISP will use the default port according to this scheme: If input is being done then the default port is the value of the symbol **piport** and if output is being done then the default port is the value of the symbol **poport**. Furthermore, if the value of piport or poport is not a valid port, then the standard input or standard output will be used, respectively.

The standard input and standard output are usually the keyboard and terminal display unless your job is running in the background and its input or output is connected to a pipe. All output which goes to the standard output will also go to the port **ptport** if it is a valid port. Output destined for the standard output will not reach the standard output if the symbol **^w** is non nil (although it will still go to **ptport** if **ptport** is a valid port).

Some of the functions listed below reference files directly. FRANZ LISP has borrowed a convenient shorthand notation from */bin/csh*, concerning naming files. If a file name begins with ~ (tilde), and the symbol **tilde-expansion**

is bound to something other than nil, then FRANZ LISP expands the file name. It takes the string of characters between the leading tilde, and the first slash as a user-name. Then, that initial segment of the filename is replaced by the home directory of the user. The null username is taken to be the current user.

Having gone to the effort of searching the password file, FRANZ LISP remembers the user directory, in case it gets asked to do so again. Tilde-expansion is performed in the following functions: *cfasl, chdir, fasl, ffasl, fileopen, infile, load, outfile, probef, sys:access, sys:unlink.*

(cfasl 'st_file 'st_entry 'st_funcname ['st_disc ['st_library]])

RETURNS: t

SIDE EFFECT: This is used to load in a foreign function (see §8.4). The object file st_file is loaded into the lisp system. St_entry should be an entry point in the file just loaded. The function binding of the symbol s_funcname will be set to point to st_entry, so that when the lisp function s_funcname is called, st_entry will be run. st_disc is the discipline to be given to s_funcname. st_disc defaults to "subroutine" if it is not given or if it is given as nil. If st_library is non-null, then after st_file is loaded, the libraries given in st_library will be searched to resolve external references. The form of st_library should be something like "-lS -lm". The C library (" -lc ") is always searched so when loading in a C file you probably won't need to specify a library. For Fortran files, you should specify "-lF77" and if you are doing any I/O, the library entry should be "-lI77 -lF77". For Pascal files "-lpc" is required.

NOTE: This function may be used to load the output of the assembler, C compiler, Fortran compiler, and Pascal compiler but NOT the lisp compiler (use *fasl* for that). If a file has more than one entry point, then use *getaddress* to locate and setup other foreign functions.
It is an error to load in a file which has a global entry point of the same name as a global entry point in the running lisp. As soon as you load in a file with *cfasl*, its global entry points become part of the lisp's entry points. Thus you cannot *cfasl* in the same file twice unless you use *removeaddress* to change certain global entry points to local entry points.

(close 'p_port**)**

RETURNS: t

SIDE EFFECT: the specified port is drained and closed, releasing the port.

NOTE: The standard defaults are not used in this case since you probably never want to close the standard output or standard input.

(cprintf 'st_format 'xfst_val ['p_port]**)**

RETURNS: xfst_val

SIDE EFFECT: The UNIX formatted output function printf is called with arguments st_format and xfst_val. If xfst_val is a symbol then its print name is passed to printf. The format string may contain characters which are just printed literally and it may contain special formatting commands preceded by a percent sign. The complete set of formatting characters is described in the UNIX manual. Some useful ones are %d for printing a fixnum in decimal, %f or %e for printing a flonum, and %s for printing a character string (or print name of a symbol).

EXAMPLE: *(cprintf "Pi equals %f" 3.14159)* prints 'Pi equals 3.14159'

(drain ['p_port]**)**

RETURNS: nil

SIDE EFFECT: If this is an output port then the characters in the output buffer are all sent to the device. If this is an input port then all pending characters are flushed. The default port for this function is the default output port.

(ex [s_filename]**)**
(vi [s_filename]**)**
(exl [s_filename]**)**
(vil [s_filename]**)**

RETURNS:

SIDE EFFECT: The lisp system starts up an editor on the file named as the argument. It will try appending .l to the file if it can't find it. The functions *exl* and *vil* will load the file after you finish editing it. These functions will also remember the name of the file so that on subsequent invocations, you don't need to provide the argument.

NOTE: These functions do not evaluate their argument.

(fasl 'st_name ['st_mapf ['g_warn]])

WHERE: st_mapf and g_warn default to nil.

RETURNS: t if the function succeeded, nil otherwise.

SIDE EFFECT: this function is designed to load in an object file generated by the lisp compiler Liszt. File names for object files usually end in '.o', so *fasl* will append '.o' to st_name (if it is not already present). If st_mapf is non nil, then it is the name of the map file to create. *Fasl* writes in the map file the names and addresses of the functions it loads and defines. Normally the map file is created (i.e. truncated if it exists), but if *(sstatus appendmap t)* is done then the map file will be appended. If g_warn is non nil and if a function is loaded from the file which is already defined, then a warning message will be printed.

NOTE: *fasl* only looks in the current directory for the file to load. The function *load* looks through a user-supplied search path and will call *fasl* if it finds a file with the same root name and a '.o' extension. In most cases the user would be better off using the function *load* rather than calling *fasl* directly.

(ffasl 'st_file 'st_entry 'st_funcname ['st_discipline ['st_library]])

RETURNS: the binary object created.

SIDE EFFECT: the Fortran object file st_file is loaded into the lisp system. St_entry should be an entry point in the file just loaded. A binary object will be created and its entry field will be set to point to st_entry. The discipline field of the binary will be set to st_discipline or "subroutine" by default. If st_library is present and non-null, then after st_file is loaded, the libraries given in st_library will be searched to resolve external references. The form of st_library should be something like "-lS -ltermcap". In any case, the standard Fortran libraries will be searched also to resolve external references.

NOTE: in F77 on Unix, the entry point for the fortran function foo is named '_foo_'.

(filepos 'p_port ['x_pos])

RETURNS: the current position in the file if x_pos is not given or else x_pos if x_pos is given.

SIDE EFFECT: If x_pos is given, the next byte to be read or written to the port will be at position x_pos.

(filestat 'st_filename)

RETURNS: a vector containing various numbers which the UNIX operating system assigns to files. if the file doesn't exist, an error is invoked. Use *probef* to determine if the file exists.

NOTE: The individual entries can be accesed by mnemonic functions of the form filestat:*field*, where field may be any of atime, ctime, dev, gid, ino, mode,mtime, nlink, rdev, size, type, uid. See the UNIX programmers manual for a more detailed description of these quantities.

(flatc 'g_form ['x_max])

RETURNS: the number of characters required to print g_form using *patom*. If x_max is given, and if *flatc* determines that it will return a value greater than x_max, then it gives up and returns the current value it has computed. This is useful if you just want to see if an expression is larger than a certain size.

(flatsize 'g_form ['x_max])

RETURNS: the number of characters required to print g_form using *print*. The meaning of x_max is the same as for flatc.

NOTE: Currently this just *explode*'s g_form and checks its length.

(fileopen 'st_filename 'st_mode)

RETURNS: a port for reading or writing (depending on st_mode) the file st_name.

SIDE EFFECT: the given file is opened (or created if opened for writing and it doesn't yet exist).

NOTE: this function call provides a direct interface to the operating system's fopen function. The mode may be more than just "r" for read, "w" for write or "a" for append. The modes "r+", "w+" and "a+" permit both reading and writing on a port provided that *fseek* is done between changes in direction. See the UNIX manual description of fopen for more details. This routine does not look through a search path for a given file.

(fseek 'p_port 'x_offset 'x_flag)

RETURNS: the position in the file after the function is performed.

SIDE EFFECT: this function positions the read/write pointer before a certain byte in the file. If x_flag is 0 then the pointer is set to x_offset bytes from the beginning of the file. If x_flag is 1 then the pointer is set to x_offset bytes from the current location in the file. If x_flag is 2 then the pointer is set to x_offset bytes from the end of the file.

(infile 's_filename)

RETURNS: a port ready to read s_filename.

SIDE EFFECT: this tries to open s_filename and if it cannot or if there are no ports available it gives an error message.

NOTE: to allow your program to continue on a file-not-found error, you can use something like:

```
(cond ((null (setq myport (car (errset (infile name) nil))))
       (patom "couldn't open the file")))
```

which will set myport to the port to read from if the file exists or will print a message if it couldn't open it and also set myport to nil. To simply determine if a file exists, there is a function named *probef*.

(load 's_filename ['st_map ['g_warn]])

RETURNS: t

NOTE: The function of *load* has changed since previous releases of FRANZ LISP and the following description should be read carefully.

SIDE EFFECT: *load* now serves the function of both *fasl* and the old *load*. *Load* will search a user defined search path for a lisp source or object file with the filename s_filename (with the extension .l or .o added as appropriate). The search path which *load* uses is the value of *(status load-search-path)*. The default is (|| /usr/lib/lisp) which means look in the current directory first and then /usr/lib/lisp. The file which *load* looks for depends on the last two characters of s_filename. If s_filename ends with ".l" then *load* will only look for a file name s_filename and will assume that this is a FRANZ LISP source file. If s_filename ends with ".o" then *load* will only look for a file named s_filename and will assume that this is a FRANZ LISP object file to be *fasl*ed in. Otherwise, *load* will first look for s_filename.o, then s_filename.l and finally s_filename itself. If it finds s_filename.o it will assume that this is an object file, otherwise it will assume that it is a source file. An object file is loaded using *fasl* and a source file is loaded by reading and evaluating each form in the file. The optional arguments st_map and g_warn are passed to *fasl* should *fasl* be called.

NOTE: *load* requires a port to open the file s_filename. It then lambda binds the symbol piport to this port and reads and evaluates the forms.

(makereadtable ['s_flag])

WHERE: if s_flag is not present it is assumed to be nil.

RETURNS: a readtable equal to the original readtable if s_flag is non-null, or else equal to the current readtable. See chapter 7 for a description of readtables and their uses.

(msg [l_option ...] ['g_msg ...])

NOTE: This function is intended for printing short messages. Any of the arguments or options presented can be used any number of times, in any order. The messages themselves (g_msg) are evaluated, and then they are transmitted to *patom*. Typically, they are strings, which evaluate to themselves. The options are interpreted specially:

msg Option Summary

(P p_portname)	causes subsequent output to go to the port p_portname (port should be opened previously)
B	print a single blank.
(B 'n_b)	evaluate n_b and print that many blanks.
N	print a single by calling *terpr*.
(N 'n_n)	evaluate n_n and transmit that many newlines to the stream.
D	*drain* the current port.

(nwritn ['p_port])

RETURNS: the number of characters in the buffer of the given port but not yet written out to the file or device. The buffer is flushed automatically when filled, or when *terpr* is called.

(outfile 's_filename ['st_type])

RETURNS: a port or nil

SIDE EFFECT: this opens a port to write s_filename. If st_type is given and if it is a symbol or string whose name begins with 'a', then the file will be opened in append mode, that is the current contents will not be lost and the next data will be written at the end of the file. Otherwise, the file opened is truncated by *outfile* if it existed beforehand. If there are no free ports, outfile returns nil. If one cannot write on s_filename, an error is signalled.

(patom 'g_exp ['p_port])

RETURNS: g_exp

SIDE EFFECT: g_exp is printed to the given port or the default port. If g_exp is a symbol or string, the print name is printed without any escape characters around special characters in the print name. If g_exp is a list then *patom* has the same effect as *print*.

(pntlen 'xfs_arg)

RETURNS: the number of characters needed to print xfs_arg.

(portp 'g_arg**)**

RETURNS: t iff g_arg is a port.

(pp [l_option] s_name1 ...**)**

RETURNS: t

SIDE EFFECT: If s_name*i* has a function binding, it is pretty-printed, otherwise if s_name*i* has a value then that is pretty-printed. Normally the output of the pretty-printer goes to the standard output port poport. The options allow you to redirect it.

PP Option Summary

(F s_filename)	direct future printing to s_filename
(P p_portname)	causes output to go to the port p_portname (port should be opened previously)
(E g_expression)	evaluate g_expression and don't print

(princ 'g_arg ['p_port]**)**

EQUIVALENT TO: patom.

(print 'g_arg ['p_port]**)**

RETURNS: nil

SIDE EFFECT: prints g_arg on the port p_port or the default port.

(probef 'st_file**)**

RETURNS: t iff the file st_file exists.

NOTE: Just because it exists doesn't mean you can read it.

(pp-form 'g_form ['p_port]**)**

RETURNS: t

SIDE EFFECT: g_form is pretty-printed to the port p_port (or poport if p_port is not given). This is the function which *pp* uses. *pp-form* does not look for function definitions or values of variables, it just prints out the form it is given.

NOTE: This is useful as a top-level-printer, c.f. *top-level* in Chapter 6.

(ratom ['p_port ['g_eof]])

RETURNS: the next atom read from the given or default port. On end of file, g_eof (default nil) is returned.

(read ['p_port ['g_eof]])

RETURNS: the next lisp expression read from the given or default port. On end of file, g_eof (default nil) is returned.

NOTE: An error will occur if the reader is given an ill formed expression. The most common error is too many right parentheses (note that this is not considered an error in Maclisp).

(readc ['p_port ['g_eof]])

RETURNS: the next character read from the given or default port. On end of file, g_eof (default nil) is returned.

(readlist 'l_arg)

RETURNS: the lisp expression read from the list of characters in l_arg.

(removeaddress 's_name1 ['s_name2 ...])

RETURNS: nil

SIDE EFFECT: the entries for the s_name*i* in the Lisp symbol table are removed. This is useful if you wish to *cfasl* or *ffasl* in a file twice, since it is illegal for a symbol in the file you are loading to already exist in the lisp symbol table.

(resetio)

RETURNS: nil

SIDE EFFECT: all ports except the standard input, output and error are closed.

(setsyntax 's_symbol 's_synclass ['ls_func])

RETURNS: t

SIDE EFFECT: this sets the code for s_symbol to sx_code in the current readtable. If s_synclass is *macro* or *splicing* then ls_func is the associated function. See Chapter 7 on the reader for more details.

(sload 's_file)

SIDE EFFECT: the file s_file (in the current directory) is opened for reading and each form is read, printed and evaluated. If the form is recognizable as a function definition, only its name will be printed, otherwise the whole form is printed.

NOTE: This function is useful when a file refuses to load because of a syntax error and you would like to narrow down where the error is.

(tab 'x_col ['p_port])

SIDE EFFECT: enough spaces are printed to put the cursor on column x_col. If the cursor is beyond x_col to start with, a *terpr* is done first.

(terpr ['p_port])

RETURNS: nil

SIDE EFFECT: a terminate line character sequence is sent to the given port or the default port. This will also drain the port.

(terpri ['p_port])

EQUIVALENT TO: terpr.

(tilde-expand 'st_filename)

RETURNS: a symbol whose pname is the tilde-expansion of the argument, (as discussed at the beginning of this chapter). If the argument does not begin with a tilde, the argument itself is returned.

(tyi ['p_port])

RETURNS: the fixnum representation of the next character read. On end of file, -1 is returned.

(tyipeek ['p_port])

RETURNS: the fixnum representation of the next character to be read.

NOTE: This does not actually read the character, it just peeks at it.

(tyo 'x_char ['p_port])

RETURNS: x_char.

SIDE EFFECT: the character whose fixnum representation is x_code, is printed as a on the given output port or the default output port.

(untyi 'x_char ['p_port])

SIDE EFFECT: x_char is put back in the input buffer so a subsequent *tyi* or *read* will read it first.

NOTE: a maximum of one character may be put back.

(username-to-dir 'st_name)

RETURNS: the home directory of the given user. The result is stored, to avoid unnecessarily searching the password file.

(zapline)

RETURNS: nil

SIDE EFFECT: all characters up to and including the line termination character are read and discarded from the last port used for input.

NOTE: this is used as the macro function for the semicolon character when it acts as a comment character.

System Functions

This chapter describes the functions used to interact with internal components of the Lisp system and operating system.

(allocate 's_type 'x_pages**)**

WHERE: s_type is one of the FRANZ LISP data types described in §1.3.

RETURNS: x_pages.

SIDE EFFECT: FRANZ LISP attempts to allocate x_pages of type s_type. If there aren't x_pages of memory left, no space will be allocated and an error will occur. The storage that is allocated is not given to the caller, instead it is added to the free storage list of s_type. The functions *segment* and *small-segment* allocate blocks of storage and return it to the caller.

(argv 'x_argnumb**)**

RETURNS: a symbol whose pname is the x_argnumb*th* argument (starting at 0) on the command line which invoked the current lisp.

NOTE: if x_argnumb is less than zero, a fixnum whose value is the number of arguments on the command line is returned. *(argv 0)* returns the name of the lisp you are running.

(baktrace)

RETURNS: nil

SIDE EFFECT: the lisp runtime stack is examined and the name of (most) of the functions currently in execution are printed, most active first.

NOTE: this will occasionally miss the names of compiled lisp functions due to incomplete information on the stack. If you are tracing compiled code, then *baktrace* won't be able to interpret the stack unless *(sstatus translink nil)* was done. See the function *showstack* for another way of printing the lisp runtime stack.

(boundp 's_name**)**

RETURNS: nil if s_name is unbound, that is it has never be given a value. If x_name has the value g_val, then (nil . g_val) is returned.

(chdir 's_path**)**

RETURNS: t iff the system call succeeds.

SIDE EFFECT: the current directory set to s_path. Among other things, this will affect the default location where the input/output functions look for and create files.

NOTE: *chdir* follows the standard UNIX conventions, if s_path does not begin with a slash, the default path is changed to the current path with s_path appended. *Chdir* employs tilde-expansion (discussed in Chapter 5).

(command-line-args)

RETURNS: a list of the arguments typed on the command line, either to the lisp interpreter, or saved lisp dump, or application compiled with the autorun option (liszt -r).

(deref 'x_addr**)**

RETURNS: The contents of x_addr, when thought of as a longword memory location.

NOTE: This may be useful in constructing arguments to C functions out of 'dangerous' areas of memory.

(dumplisp s_name**)**

RETURNS: nil

SIDE EFFECT: the current lisp is dumped to the named file. When s_name is executed, you will be in a lisp in the same state as when the dumplisp was done.

NOTE: dumplisp will fail if one tries to write over the current running file. UNIX does not allow you to modify the file you are running.

(eval-when l_time g_exp1 ...**)**

SIDE EFFECT: l_time may contain any combination of the symbols *load*, *eval*, and *compile*. The effects of load and compile is discussed in §12.3.2.1 compiler. If eval is present however, this simply means that the expressions g_exp1 and so on are evaluated from left to right. If eval is not present, the forms are not evaluated.

(exit ['x_code]**)**

RETURNS: nothing (it never returns).

SIDE EFFECT: the lisp system dies with exit code x_code or 0 if x_code is not specified.

(fake 'x_addr**)**

RETURNS: the lisp object at address x_addr.

NOTE: This is intended to be used by people debugging the lisp system.

(fork)

RETURNS: nil to the child process and the process number of the child to the parent.

SIDE EFFECT: A copy of the current lisp system is made in memory and both lisp systems now begin to run. This function can be used interactively to temporarily save the state of Lisp (as shown below), but you must be careful that only one of the lisp's interacts with the terminal after the fork. The *wait* function is useful for this.

```
-> (setq foo 'bar)          ;; set a variable
bar
-> (cond ((fork)(wait)))    ;; duplicate the lisp system and
nil                                 ;; make the parent wait
-> foo                             ;; check the value of the variable
bar
-> (setq foo 'baz)          ;; give it a new value
baz
-> foo                             ;; make sure it worked
baz
-> (exit)                          ;; exit the child
(5274 . 0)                         ;; the wait function returns this
-> foo                             ;; we check to make sure parent was
bar                                ;; not modified.
```

(gc)

RETURNS: nil

SIDE EFFECT: this causes a garbage collection.

NOTE: The function *gcafter* is not called automatically after this function finishes. Normally the user doesn't have to call *gc* since garbage collection occurs automatically whenever internal free lists are exhausted.

(gcafter s_type**)**

WHERE: s_type is one of the FRANZ LISP data types listed in §1.3.

NOTE: this function is called by the garbage collector after a garbage collection which was caused by running out of data type s_type. This function should determine if more space need be allocated and if so should allocate it. There is a default gcafter function but users who want control over space allocation can define their own -- but note that it must be an nlambda.

(getenv 's_name**)**

RETURNS: a symbol whose pname is the value of s_name in the current UNIX environment. If s_name doesn't exist in the current environment, a symbol with a null pname is returned.

(hashtabstat)

RETURNS: a list of fixnums representing the number of symbols in each bucket of the oblist.

NOTE: the oblist is stored a hash table of buckets. Ideally there would be the same number of symbols in each bucket.

(help [sx_arg])

SIDE EFFECT: If sx_arg is a symbol then the portion of this manual beginning with the description of sx_arg is printed on the terminal. If sx_arg is a fixnum or the name of one of the appendicies, that chapter or appendix is printed on the terminal. If no argument is provided, *help* prints the options that it recognizes. The program 'more' is used to print the manual on the terminal; it will stop after each page and will continue after the space key is pressed.

(include s_filename)

RETURNS: nil

SIDE EFFECT: The given filename is *load*ed into the lisp.

NOTE: this is similar to load except the argument is not evaluated. Include means something special to the compiler.

(include-if 'g_predicate s_filename)

RETURNS: nil

SIDE EFFECT: This has the same effect as include, but is only actuated if the predicate is non-nil.

(includef 's_filename)

RETURNS: nil

SIDE EFFECT: this is the same as *include* except the argument is evaluated.

(includef-if 'g_predicate s_filename)

RETURNS: nil

SIDE EFFECT: This has the same effect as includef, but is only actuated if the predicate is non-nil.

(maknum 'g_arg)

RETURNS: the address of its argument converted into a fixnum.

(monitor ['xs_maxaddr])

RETURNS: t

SIDE EFFECT: If xs_maxaddr is t then profiling of the entire lisp system is begun. If xs_maxaddr is a fixnum then profiling is done only up to address xs_maxaddr. If xs_maxaddr is not given, then profiling is stopped and the data obtained is written to the file 'mon.out' where it can be analyzed with the UNIX 'prof' program.

NOTE: this function only works if the lisp system has been compiled in a special way, otherwise, an error is invoked.

(opval 's_arg ['g_newval])

RETURNS: the value associated with s_arg before the call.

SIDE EFFECT: If g_newval is specified, the value associated with s_arg is changed to g_newval.

NOTE: *opval* keeps track of storage allocation. If s_arg is one of the data types then *opval* will return a list of three fixnums representing the number of items of that type in use, the number of pages allocated and the number of items of that type per page. You should never try to change the value *opval* associates with a data type using *opval*.
If s_arg is *pagelimit* then *opval* will return (and set if g_newval is given) the maximum amount of lisp data pages it will allocate. This limit should remain small unless you know your program requires lots of space as this limit will catch programs in infinite loops which gobble up memory.

(*process 'st_command ['g_readp ['g_writep]])

RETURNS: either a fixnum if one argument is given, or a list of two ports and a fixnum if two or three arguments are given.

NOTE: **process* starts another process by passing st_command to the shell (it first tries /bin/csh, then it tries /bin/sh if /bin/csh doesn't exist). If only one argument is given to **process*, **process* waits for the new process to die and then returns the exit code of the new process. If more two or three arguments are given, **process* starts the process and then returns a list which, depending on the value of g_readp and g_writep, may contain i/o ports for communcating with the new process. If g_writep is non-null, then a port will be created which the lisp program can use to send characters to the new process. If g_readp is non-null, then a port will be created which the lisp program can use to read characters from the new process. The value returned by **process* is (readport writeport pid) where readport and writeport are either nil or a port based on the value of g_readp and g_writep. Pid is the process id of the new process. Since it is hard to remember the order of g_readp and g_writep, the functions **process-send* and **process-receive* were written to perform the common functions.

(*process-receive 'st_command)

RETURNS: a port which can be read.

SIDE EFFECT: The command st_command is given to the shell and it is started running in the background. The output of that command is available for reading via the port returned. The input of the command process is set to /dev/null.

(*process-send 'st_command)

RETURNS: a port which can be written to.

SIDE EFFECT: The command st_command is given to the shell and it is started runing in the background. The lisp program can provide input for that command by sending characters to the port returned by this function. The output of the command process is set to /dev/null.

(process s_pgrm [s_frompipe s_topipe])

RETURNS: if the optional arguments are not present a fixnum which is the exit code when s_prgm dies. If the optional arguments are present, it returns a fixnum which is the process id of the child.

NOTE: This command is obsolete. New programs should use one of the **process* commands given above.

SIDE EFFECT: If s_frompipe and s_topipe are given, they are bound to ports which are pipes which direct characters from FRANZ LISP to the new process and to FRANZ LISP from the new process respectively. *Process* forks a process named s_prgm and waits for it to die iff there are no pipe arguments given.

(ptime)

RETURNS: a list of two elements, the first is the amount of processor time used by the lisp system so far, the second is the amount of time used by the garbage collector so far.

NOTE: the time is measured in those units used by the *times*(2) system call, usually 60*ths* of a second. The first number includes the second number. The amount of time used by garbage collection is not recorded until the first call to ptime. This is done to prevent overhead when the user is not interested in garbage collection times.

(reset)

SIDE EFFECT: the lisp runtime stack is cleared and the system restarts at the top level by executing a *(funcall top-level nil)*

(restorelisp 's_name)

SIDE EFFECT: this reads in file s_name (which was created by *savelisp*) and then does a *(reset)*.

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(retbrk ['x_level])

WHERE: x_level is a small integer of either sign.

SIDE EFFECT: The default error handler keeps a notion of the current level of the error caught. If x_level is negative, control is thrown to this default error handler whose level is that many less than the present, or to *top-level* if there aren't enough. If x_level is non-negative, control is passed to the handler at that level. If x_level is not present, the value -1 is taken by default.

(*rset 'g_flag)

RETURNS: g_flag

SIDE EFFECT: If g_flag is non nil then the lisp system will maintain extra information about calls to *eval* and *funcall*. This record keeping slows down the evaluation but this is required for the functions *evalhook*, *funcallhook*, and *evalframe* to work. To debug compiled lisp code the transfer tables should be unlinked: *(sstatus translink nil)*

(savelisp 's_name**)**

RETURNS: t

SIDE EFFECT: the state of the Lisp system is saved in the file s_name. It can be read in by *restorelisp.*

NOTE: This is only used on VMS systems where *dumplisp* cannot be used.

(segment 's_type 'x_size**)**

WHERE: s_type is one of the data types given in §1.3

RETURNS: a segment of contiguous lispvals of type s_type.

NOTE: In reality, *segment* returns a new data cell of type s_type and allocates space for x_size − 1 more s_type's beyond the one returned. *Segment* always allocates new space and does so in 512 byte chunks. If you ask for 2 fixnums, segment will actually allocate 128 of them thus wasting 126 fixnums. The function *small-segment* is a smarter space allocator and should be used whenever possible.

(shell)

RETURNS: the exit code of the shell when it dies.

SIDE EFFECT: this forks a new shell and returns when the shell dies.

(showstack)

RETURNS: nil

SIDE EFFECT: all forms currently in evaluation are printed, beginning with the most recent. For compiled code the most that showstack will show is the function name and it may miss some functions.

(signal 'x_signum 's_name**)**

RETURNS: nil if no previous call to signal has been made, or the previously installed s_name.

SIDE EFFECT: this declares that the function named s_name will handle the signal number x_signum. If s_name is nil, the signal is ignored. Presently only four UNIX signals are caught, they and their numbers are: Interrupt(2), Floating exception(8), Alarm(14), and Hang-up(1).

(sizeof 'g_arg**)**

RETURNS: the number of bytes required to store one object of type g_arg, encoded as a fixnum.

(small-segment 's_type 'x_cells**)**

WHERE: s_type is one of fixnum, flonum and value.

RETURNS: a segment of x_cells data objects of type s_type.

SIDE EFFECT: This may call *segment* to allocate new space or it may be able to fill the request on a page already allocated. The value returned by *small-segment* is usually stored in the data subpart of an array object.

(sstatus g_type g_val)

RETURNS: g_val

SIDE EFFECT: If g_type is not one of the special sstatus codes described in the next few pages this simply sets g_val as the value of status type g_type in the system status property list.

(sstatus appendmap g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when *fasl* is told to create a load map, it will append to the file name given in the *fasl* command, rather than creating a new map file. The initial value is nil.

(sstatus automatic-reset g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when an error occurs which no one wants to handle, a *reset* will be done instead of entering a primitive internal break loop. The initial value is t.

(sstatus chainatom g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is non nil and a *car* or *cdr* of a symbol is done, then nil will be returned instead of an error being signaled. This only affects the interpreter, not the compiler. The initial value is nil.

(sstatus dumpcore g_val)

RETURNS: g_val

SIDE EFFECT: If g_val is nil, FRANZ LISP tells UNIX that a segmentation violation or bus error should cause a core dump. If g_val is non nil then FRANZ LISP will catch those errors and print a message advising the user to reset.

NOTE: The initial value for this flag is nil, and only those knowledgeable of the innards of the lisp system should ever set this flag non nil.

(sstatus dumpmode x_val)

RETURNS: x_val

SIDE EFFECT: All subsequent *dumplisp*'s will be done in mode x_val. x_val may be either 413 or 410 (decimal).

NOTE: the advantage of mode 413 is that the dumped Lisp can be demand paged in when first started, which will make it start faster and disrupt other users less. The initial value is 413.

(sstatus evalhook g_val**)**

RETURNS: g_val

SIDE EFFECT: When g_val is non nil, this enables the evalhook and funcallhook traps in the evaluator. See §14.4 for more details.

(sstatus feature g_val**)**

RETURNS: g_val

SIDE EFFECT: g_val is added to the *(status features)* list,

(sstatus gcstrings g_val**)**

RETURNS: g_val

SIDE EFFECT: if g_val is non-null, and if string garbage collection was enabled when the lisp system was compiled, string space will be garbage collected.

NOTE: the default value for this is nil since in most applications garbage collecting strings is a waste of time.

(sstatus ignoreeof g_val**)**

RETURNS: g_val

SIDE EFFECT: If g_val is non-null when an end of file (CNTL-D on UNIX) is typed to the standard top-level interpreter, it will be ignored rather then cause the lisp system to exit. If the the standard input is a file or pipe then this has no effect, an EOF will always cause lisp to exit. The initial value is nil.

(sstatus nofeature g_val**)**

RETURNS: g_val

SIDE EFFECT: g_val is removed from the status features list if it was present.

(sstatus translink g_val**)**

RETURNS: g_val

SIDE EFFECT: If g_val is nil then all transfer tables are cleared and further calls through the transfer table will not cause the fast links to be set up. If g_val is the symbol *on* then all possible transfer table entries will be linked and the flag will be set to cause fast links to be set up dynamically. Otherwise all that is done is to set the flag to cause fast links to be set up dynamically. The initial value is nil.

NOTE: For a discussion of transfer tables, see §12.8.

(sstatus uctolc g_val**)**

RETURNS: g_val

SIDE EFFECT: If g_val is not nil then all unescaped capital letters in symbols read by the reader will be converted to lower case.

NOTE: This allows FRANZ LISP to be compatible with single case lisp systems (e.g. Maclisp, Interlisp and UCILisp).

(status g_code)

RETURNS: the value associated with the status code g_code if g_code is not one of the special cases given below

(status ctime)

RETURNS: a symbol whose print name is the current time and date.

EXAMPLE: *(status ctime)* = |Sun Jun 29 16:51:26 1980|

NOTE: This has been made obsolete by *time-string*, described below.

(status feature g_val)

RETURNS: t iff g_val is in the status features list.

(status features)

RETURNS: the value of the features code, which is a list of features which are present in this system. You add to this list with *(sstatus feature 'g_val)* and test if feature g_feat is present with *(status feature 'g_feat)*.

(status isatty)

RETURNS: t iff the standard input is a terminal.

(status localtime)

RETURNS: a list of fixnums representing the current time.

EXAMPLE: *(status localtime)* = (3 51 13 31 6 81 5 211 1)
means 3*rd* second, 51*st* minute, 13*th* hour (1 p.m), 31*st* day, month 6 (0 = January), year 81 (0 = 1900), day of the week 5 (0 = Sunday), 211*th* day of the year and daylight savings time is in effect.

(status syntax s_char)

NOTE: This function should not be used. See the description of *getsyntax* (in Chapter 7) for a replacement.

(status undeffunc)

RETURNS: a list of all functions which transfer table entries point to but which are not defined at this point.

NOTE: Some of the undefined functions listed could be arrays which have yet to be created.

(status version)

RETURNS: a string which is the current lisp version name.

EXAMPLE: *(status version)* = "Franz Lisp, Opus 38.61"

(syscall 'x_index ['xst_arg1 ...])

RETURNS: the result of issuing the UNIX system call number x_index with arguments xst_arg*i*.

NOTE: The UNIX system calls are described in section 2 of the UNIX Programmer's manual. If xst_arg*i* is a fixnum, then its value is passed as an argument, if it is a symbol then its pname is passed and finally if it is a string then the string itself is passed as an argument. Some useful syscalls are:
(syscall 20) returns process id.
(syscall 13) returns the number of seconds since Jan 1, 1970.
(syscall 10 'foo) will unlink (delete) the file foo.

(sys:access 'st_filename 'x_mode)
(sys:chmod 'st_filename 'x_mode)
(sys:gethostname)
(sys:getpid)
(sys:getpwnam 'st_username)
(sys:link 'st_oldfilename 'st_newfilename)
(sys:time)
(sys:unlink 'st_filename)

NOTE: We have been warned that the actual system call numbers may vary among different UNIX systems. Users concerned about portability may wish to use this group of functions. Another advantage is that tilde-expansion is performed on all filename arguments. These functions do what is described in the system call section of your UNIX manual.

sys:getpwname returns a vector of four entries from the password file, being the user name, user id, group id, and home directory.

(time-string ['x_seconds])

RETURNS: an ascii string, giving the time and date which was x_seconds after UNIX's idea of creation (Midnight, Jan 1, 1970 GMT). If no argument is given, time-string returns the current date. This supplants *(status ctime)*, and may be used to make the results of *filestat* more intelligible.

(top-level)

RETURNS: nothing (it never returns)

NOTE: This function is the top-level read-eval-print loop. It never returns any value. Its main utility is that if you redefine it, and do a (reset) then the redefined (top-level) is then invoked. The default top-level for Franz, allow one to specify his own printer or reader, by binding the symbols **top-level-printer** and **top-level-reader**. One can let the default top-level do most of the drudgery in catching *reset*'s, and reading in .lisprc files, by binding the symbol **user-top-level**, to a routine that concerns itself only with the read-eval-print loop.

(wait)

RETURNS: a dotted pair *(processid . status)* when the next child process dies.

CHAPTER 7

The Lisp Reader

7.1. Introduction

The *read* function is responsible for converting a stream of characters into a Lisp expression. *Read* is table driven and the table it uses is called a *readtable.* The *print* function does the inverse of *read*; it converts a Lisp expression into a stream of characters. Typically the conversion is done in such a way that if that stream of characters were read by *read*, the result would be an expression equal to the one *print* was given. *Print* must also refer to the readtable in order to determine how to format its output. The *explode* function, which returns a list of characters rather than printing them, must also refer to the readtable.

A readtable is created with the *makereadtable* function, modified with the *setsyntax* function and interrogated with the *getsyntax* function. The structure of a readtable is hidden from the user - a readtable should only be manipulated with the three functions mentioned above.

There is one distinguished readtable called the *current readtable* whose value determines what *read*, *print* and *explode* do. The current readtable is the value of the symbol *readtable.* Thus it is possible to rapidly change the current syntax by lambda binding a different readtable to the symbol *readtable.* When the binding is undone, the syntax reverts to its old form.

7.2. Syntax Classes

The readtable describes how each of the 128 ascii characters should be treated by the reader and printer. Each character belongs to a *syntax class* which has three properties:

character class -
: Tells what the reader should do when it sees this character. There are a large number of character classes. They are described below.

separator -
: Most types of tokens the reader constructs are one character long. Four token types have an arbitrary length: number (1234), symbol print name (franz), escaped symbol print name (\|franz\|), and string ("franz"). The reader can easily determine when it has come to the end of one of the last two types: it just looks for the matching delimiter (\| or "). When the reader is reading a number or symbol print name, it stops reading when it comes to a character with the *separator* property. The separator character is pushed back into the input stream and will be the first character read when the reader is called again.

escape -
: Tells the printer when to put escapes in front of, or around, a symbol whose print name contains this character. There are three possibilities: always escape a symbol with this character in it, only escape a symbol if this is the only character in the

symbol, and only escape a symbol if this is the first character in the symbol. [note: The printer will always escape a symbol which, if printed out, would look like a valid number.]

When the Lisp system is built, Lisp code is added to a C-coded kernel and the result becomes the standard lisp system. The readtable present in the C-coded kernel, called the *raw readtable*, contains the bare necessities for reading in Lisp code. During the construction of the complete Lisp system, a copy is made of the raw readtable and then the copy is modified by adding macro characters. The result is what is called the *standard readtable.* When a new readtable is created with *makereadtable,* a copy is made of either the raw readtable or the current readtable (which is likely to be the standard readtable).

7.3. Reader operations

The reader has a very simple algorithm. It is either *scanning* for a token, *collecting* a token, or *processing* a token. Scanning involves reading characters and throwing away those which don't start tokens (such as blanks and tabs). Collecting means gathering the characters which make up a token into a buffer. Processing may involve creating symbols, strings, lists, fixnums, bignums or flonums or calling a user written function called a character macro.

The components of the syntax class determine when the reader switches between the scanning, collecting and processing states. The reader will continue scanning as long as the character class of the characters it reads is *cseparator.* When it reads a character whose character class is not *cseparator* it stores that character in its buffer and begins the collecting phase.

If the character class of that first character is *ccharacter*, *cnumber*, *cperiod*, or *csign.* then it will continue collecting until it runs into a character whose syntax class has the *separator* property. (That last character will be pushed back into the input buffer and will be the first character read next time.) Now the reader goes into the processing phase, checking to see if the token it read is a number or symbol. It is important to note that after the first character is collected the component of the syntax class which tells the reader to stop collecting is the *separator* property, not the character class.

If the character class of the character which stopped the scanning is not *ccharacter*, *cnumber*, *cperiod*, or *csign.* then the reader processes that character immediately. The character classes *csingle-macro*, *csingle-splicing-macro*, and *csingle-infix-macro* will act like *ccharacter* if the following token is not a *separator.* The processing which is done for a given character class is described in detail in the next section.

7.4. Character classes

ccharacter — raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_`{}~
standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

A normal character.

cnumber — raw readtable:0-9
standard readtable:0-9

This type is a digit. The syntax for an integer (fixnum or bignum) is a string of *cnumber* characters optionally followed by a *cperiod.* If the digits are not followed by a *cperiod*,

then they are interpreted in base *ibase* which must be eight or ten. The syntax for a floating point number is either zero or more *cnumber*'s followed by a *cperiod* and then followed by one or more *cnumber*'s. A floating point number may also be an integer or floating point number followed by 'e' or 'd', an optional '+' or '−' and then zero or more *cnumber*'s.

csign

raw readtable:+ −
standard readtable:+ −

A leading sign for a number. No other characters should be given this class.

cleft-paren

raw readtable:(
standard readtable:(

A left parenthesis. Tells the reader to begin forming a list.

cright-paren

raw readtable:)
standard readtable:)

A right parenthesis. Tells the reader that it has reached the end of a list.

cleft-bracket

raw readtable:[
standard readtable:[

A left bracket. Tells the reader that it should begin forming a list. See the description of *cright-bracket* for the difference between cleft-bracket and cleft-paren.

cright-bracket

raw readtable:]
standard readtable:]

A right bracket. A *cright-bracket* finishes the formation of the current list and all enclosing lists until it finds one which begins with a *cleft-bracket* or until it reaches the top level list.

cperiod

raw readtable:.
standard readtable:.

The period is used to separate element of a cons cell [e.g. (a . (b . nil)) is the same as (a b)]. *cperiod* is also used in numbers as described above.

cseparator

raw readtable:^I-^M esc space
standard readtable:^I-^M esc space

Separates tokens. When the reader is scanning, these character are passed over. Note: there is a difference between the *cseparator* character class and the *separator* property of a syntax class.

csingle-quote

raw readtable:'
standard readtable:'

This causes *read* to be called recursively and the list (quote <value read>) to be returned.

csymbol-delimiter

raw readtable:

standard readtable:|

This causes the reader to begin collecting characters and to stop only when another identical *csymbol-delimiter* is seen. The only way to escape a *csymbol-delimiter* within a symbol name is with a *cescape* character. The collected characters are converted into a string which becomes the print name of a symbol. If a symbol with an identical print name already exists, then the allocation is not done, rather the existing symbol is used.

cescape

raw readtable:\
standard readtable:\

This causes the next character to read in to be treated as a **vcharacter**. A character whose syntax class is **vcharacter** has a character class *ccharacter* and does not have the *separator* property so it will not separate symbols.

cstring-delimiter

raw readtable:"
standard readtable:"

This is the same as *csymbol-delimiter* except the result is returned as a string instead of a symbol.

csingle-character-symbol

raw readtable:none
standard readtable:none

This returns a symbol whose print name is the the single character which has been collected.

cmacro

raw readtable:none
standard readtable:‘,

The reader calls the macro function associated with this character and the current readtable, passing it no arguments. The result of the macro is added to the structure the reader is building, just as if that form were directly read by the reader. More details on macros are provided below.

csplicing-macro

raw readtable:none
standard readtable:#;

A *csplicing-macro* differs from a *cmacro* in the way the result is incorporated in the structure the reader is building. A *csplicing-macro* must return a list of forms (possibly empty). The reader acts as if it read each element of the list itself without the surrounding parenthesis.

csingle-macro

raw readtable:none
standard readtable:none

This causes to reader to check the next character. If it is a *cseparator* then this acts like a *cmacro*. Otherwise, it acts like a *ccharacter*.

csingle-splicing-macro

raw readtable:none
standard readtable:none

This is triggered like a *csingle-macro* however the result is spliced in like a *csplicing-macro*.

cinfix-macro

raw readtable:none

standard readtable:none

This is differs from a *cmacro* in that the macro function is passed a form representing what the reader has read so far. The result of the macro replaces what the reader had read so far.

csingle-infix-macro

raw readtable:none
standard readtable:none

This differs from the *cinfix-macro* in that the macro will only be triggered if the character following the *csingle-infix-macro* character is a *cseparator.*

cillegal

raw readtable:^@-^G^N-^Z^\-^_rubout
standard readtable:^@-^G^N-^Z^\-^_rubout

The characters cause the reader to signal an error if read.

7.5. Syntax classes

The readtable maps each character into a syntax class. The syntax class contains three pieces of information: the character class, whether this is a separator, and the escape properties. The first two properties are used by the reader, the last by the printer (and *explode*). The initial lisp system has the following syntax classes defined. The user may add syntax classes with *add-syntax-class.* For each syntax class, we list the properties of the class and which characters have this syntax class by default. More information about each syntax class can be found under the description of the syntax class's character class.

vcharacter
ccharacter

raw readtable:A-Z a-z ^H !#$%&*,/:;<=>?@^_`{}~
standard readtable:A-Z a-z ^H !$%&*/:;<=>?@^_{}~

vnumber
cnumber

raw readtable:0-9
standard readtable:0-9

vsign
csign

raw readtable:+-
standard readtable:+-

vleft-paren
cleft-paren
escape-always
separator

raw readtable:(
standard readtable:(

vright-paren
cright-paren
escape-always
separator

raw readtable:)
standard readtable:)

vleft-bracket
cleft-bracket

raw readtable:[
standard readtable:[

escape-always
separator

vright-bracket raw readtable:]
cright-bracket standard readtable:]
escape-always
separator

vperiod raw readtable:.
cperiod standard readtable:.
escape-when-unique

vseparator raw readtable:^I-^M esc space
cseparator standard readtable:^I-^M esc space
escape-always
separator

vsingle-quote raw readtable:'
csingle-quote standard readtable:'
escape-always
separator

vsymbol-delimiter raw readtable:|
csingle-delimiter standard readtable:|
escape-always

vescape raw readtable:\
cescape standard readtable:\
escape-always

vstring-delimiter raw readtable:"
cstring-delimiter standard readtable:"
escape-always

vsingle-character-symbol raw readtable:none
csingle-character-symbol standard readtable:none
separator

vmacro raw readtable:none
cmacro standard readtable:`,
escape-always
separator

vsplicing-macro raw readtable:none
csplicing-macro standard readtable:#;
escape-always
separator

vsingle-macro raw readtable:none
csingle-macro standard readtable:none

escape-when-unique

vsingle-splicing-macro raw readtable:none
csingle-splicing-macro standard readtable:none
escape-when-unique

vinfix-macro raw readtable:none
cinfix-macro standard readtable:none
escape-always
separator

vsingle-infix-macro raw readtable:none
csingle-infix-macro standard readtable:none
escape-when-unique

villegal raw readtable:^@-^G^N-^Z^\-^_rubout
cillegal standard readtable:^@-^G^N-^Z^\-^_rubout
escape-always
separator

7.6. Character Macros

Character macros are user written functions which are executed during the reading process. The value returned by a character macro may or may not be used by the reader, depending on the type of macro and the value returned. Character macros are always attached to a single character with the *setsyntax* function.

7.6.1. Types There are three types of character macros: normal, splicing and infix. These types differ in the arguments they are given or in what is done with the result they return.

7.6.1.1. Normal

A normal macro is passed no arguments. The value returned by a normal macro is simply used by the reader as if it had read the value itself. Here is an example of a macro which returns the abbreviation for a given state.

```
-> (defun stateabbrev nil
       (cdr (assq (read) '((california . ca) (pennsylvania . pa)))))
stateabbrev
-> (setsyntax \! 'vmacro 'stateabbrev)
t
-> '( ! california ! wyoming ! pennsylvania)
(ca nil pa)
```

Notice what happened to
! wyoming. Since it wasn't in the table, the macro probably didn't want to return anything at all, but it had to return something, and whatever it returned was put in the list. The splicing macro, described next, allows a character macro function to return a value that is ignored.

7.6.1.2. Splicing

The value returned from a splicing macro must be a list or nil. If the value is nil, then the value is ignored, otherwise the reader acts as if it read each object in the list. Usually the list only contains one element. If the reader is reading at the top level (i.e. not collecting elements of list), then it is illegal for a splicing macro to return more then one element in the list. The major advantage of a splicing macro over a normal macro is the ability of the splicing macro to return nothing. The comment character (usually ;) is a splicing macro bound to a function which reads to the end of the line and always returns nil. Here is the previous example written as a splicing macro

```
-> (defun stateabbrev nil
       ((lambda (value)
            (cond (value (list value))
                  (t nil)))
        (cdr (assq (read) '((california . ca) (pennsylvania . pa))))))
-> (setsyntax '! 'vsplicing-macro 'stateabbrev)
-> '(!pennsylvania ! foo !california)
(pa ca)
-> '!foo !bar !pennsylvania
pa
->
```

7.6.1.3. Infix

Infix macros are passed a *conc* structure representing what has been read so far. Briefly, a tconc structure is a single list cell whose car points to a list and whose cdr points to the last list cell in that list. The interpretation by the reader of the value returned by an infix macro depends on whether the macro is called while the reader is constructing a list or whether it is called at the top level of the

reader. If the macro is called while a list is being constructed, then the value returned should be a tconc structure. The car of that structure replaces the list of elements that the reader has been collecting. If the macro is called at top level, then it will be passed the value nil, and the value it returns should either be nil or a tconc structure. If the macro returns nil, then the value is ignored and the reader continues to read. If the macro returns a tconc structure of one element (i.e. whose car is a list of one element), then that single element is returned as the value of *read*. If the macro returns a tconc structure of more than one element, then that list of elements is returned as the value of read.

```
-> (defun plusop (x)
       (cond ((null x) (tconc nil \+))
             (t (lconc nil (list 'plus (caar x) (read))))))

plusop
-> (setsyntax \+ 'vinfix-macro 'plusop)
t
-> '(a + b)
(plus a b)
-> '+
|+|
->
```

7.6.2. Invocations

There are three different circumstances in which you would like a macro function to be triggered.

Always -
Whenever the macro character is seen, the macro should be invoked. This is accomplished by using the character classes *cmacro*, *csplicing-macro*, or *cinfix-macro*, and by using the *separator* property. The syntax classes **vmacro**, **vsplicing-macro**, and **vsingle-macro** are defined this way.

When first -
The macro should only be triggered when the macro character is the first character found after the scanning process. A syntax class for a *when first* macro would be defined using *cmacro*, *csplicing-macro*, or *cinfix-macro* and not including the *separator* property.

When unique -
The macro should only be triggered when the macro character is the only character collected in the token collection phase of the reader, i.e the macro character is preceeded by zero or more *cseparators* and followed by a *separator*. A syntax class for a *when unique* macro would be defined using *csingle-macro*, *csingle-splicing-macro*, or *csingle-infix-macro* and not including the *separator* property. The syntax classes so defined are **vsingle-macro**, **vsingle-splicing-macro**, and **vsingle-infix-macro**.

7.7. Functions

(setsyntax 's_symbol 's_synclass ['ls_func])

WHERE: ls_func is the name of a function or a lambda body.

RETURNS: t

SIDE EFFECT: S_symbol should be a symbol whose print name is only one character. The syntax class for that character is set to s_synclass in the current readtable. If s_synclass is a class that requires a character macro, then ls_func must be supplied.

NOTE: The symbolic syntax codes are new to Opus 38. For compatibility, s_synclass can be one of the fixnum syntax codes which appeared in older versions of the FRANZ LISP Manual. This compatibility is only temporary: existing code which uses the fixnum syntax codes should be converted.

(getsyntax 's_symbol)

RETURNS: the syntax class of the first character of s_symbol's print name. s_symbol's print name must be exactly one character long.

NOTE: This function is new to Opus 38. It supercedes *(status syntax)* which no longer exists.

(add-syntax-class 's_synclass 'l_properties)

RETURNS: s_synclass

SIDE EFFECT: Defines the syntax class s_synclass to have properties l_properties. The list l_properties should contain a character classes mentioned above. l_properties may contain one of the escape properties: *escape-always*, *escape-when-unique*, or *escape-when-first*. l_properties may contain the *separator* property. After a syntax class has been defined with *add-syntax-class*, the *setsyntax* function can be used to give characters that syntax class.

```
; Define a non-separating macro character.
; This type of macro character is used in UCI-Lisp, and
; it corresponds to a  FIRST MACRO in Interlisp
-> (add-syntax-class 'vuci-macro '(cmacro escape-when-first))
vuci-macro
->
```

CHAPTER 8

Functions, Fclosures, and Macros

8.1. valid function objects

There are many different objects which can occupy the function field of a symbol object. Table 8.1, on the following page, shows all of the possibilities, how to recognize them, and where to look for documentation.

8.2. functions

The basic Lisp function is the lambda function. When a lambda function is called, the actual arguments are evaluated from left to right and are lambda-bound to the formal parameters of the lambda function.

An nlambda function is usually used for functions which are invoked by the user at top level. Some built-in functions which evaluate their arguments in special ways are also nlambdas (e.g *cond*, *do*, *or*). When an nlambda function is called, the list of unevaluated arguments is lambda bound to the single formal parameter of the nlambda function.

Some programmers will use an nlambda function when they are not sure how many arguments will be passed. Then, the first thing the nlambda function does is map *eval* over the list of unevaluated arguments it has been passed. This is usually the wrong thing to do, as it will not work compiled if any of the arguments are local variables. The solution is to use a lexpr. When a lexpr function is called, the arguments are evaluated and a fixnum whose value is the number of arguments is lambda-bound to the single formal parameter of the lexpr function. The lexpr can then access the arguments using the *arg* function.

When a function is compiled, *special* declarations may be needed to preserve its behavior. An argument is not lambda-bound to the name of the corresponding formal parameter unless that formal parameter has been declared *special* (see §12.3.2.2).

Lambda and lexpr functions both compile into a binary object with a discipline of lambda. However, a compiled lexpr still acts like an interpreted lexpr.

8.3. macros

An important feature of Lisp is its ability to manipulate programs as data. As a result of this, most Lisp implementations have very powerful macro facilities. The Lisp language's macro facility can be used to incorporate popular features of the other languages into Lisp. For example, there are macro packages which allow one to create records (ala Pascal) and refer to elements of those records by the field names. The *struct* package imported from Maclisp does this. Another popular use for macros is to create more readable control structures which expand into *cond*, *or* and *and*. One such example is the If macro. It allows you to write

informal name	object type	documentation
interpreted lambda function	list with *car* *eq* to lambda	8.2
interpreted nlambda function	list with *car* *eq* to nlambda	8.2
interpreted lexpr function	list with *car* *eq* to lexpr	8.2
interpreted macro	list with *car* *eq* to macro	8.3
fclosure	vector with *vprop* *eq* to fclosure	8.4
compiled lambda or lexpr function	binary with discipline *eq* to lambda	8.2
compiled nlambda function	binary with discipline *eq* to nlambda	8.2
compiled macro	binary with discipline *eq* to macro	8.3
foreign subroutine	binary with discipline of "subroutine"†	8.5
foreign function	binary with discipline of "function"†	8.5
foreign integer function	binary with discipline of "integer-function"†	8.5
foreign real function	binary with discipline of "real-function"†	8.5
foreign C function	binary with discipline of "c-function"†	8.5
foreign double function	binary with discipline of "double-c-function"†	8.5
foreign structure function	binary with discipline of "vector-c-function"†	8.5
array	array object	9

Table 8.1

```
(If (equal numb 0) then (print 'zero) (terpr)
 elseif (equal numb 1) then (print 'one) (terpr)
 else (print |I give up|))
```

which expands to

```
(cond
   ((equal numb 0) (print 'zero) (terpr))
   ((equal numb 1) (print 'one) (terpr))
   (t (print |I give up|)))
```

†**Only the first character of the string is significant (i.e "s" is ok for "subroutine")**

8.3.1. macro forms

A macro is a function which accepts a Lisp expression as input and returns another Lisp expression. The action the macro takes is called macro expansion. Here is a simple example:

```
-> (def first (macro (x) (cons 'car (cdr x))))
first
-> (first '(a b c))
a
-> (apply 'first '(first '(a b c)))
(car '(a b c))
```

The first input line defines a macro called *first*. Notice that the macro has one formal parameter, *x*. On the second input line, we ask the interpreter to evaluate *(first '(a b c))*. *Eval* sees that *first* has a function definition of type macro, so it evaluates *first*'s definition, passing to *first*, as an argument, the form *eval* itself was trying to evaluate: *(first '(a b c))*. The *first* macro chops off the car of the argument with *cdr*, cons' a *car* at the beginning of the list and returns *(car '(a b c))*, which *eval* evaluates. The value *a* is returned as the value of *(first '(a b c))*. Thus whenever *eval* tries to evaluate a list whose car has a macro definition it ends up doing (at least) two operations, the first of which is a call to the macro to let it macro expand the form, and the other is the evaluation of the result of the macro. The result of the macro may be yet another call to a macro, so *eval* may have to do even more evaluations until it can finally determine the value of an expression. One way to see how a macro will expand is to use *apply* as shown on the third input line above.

8.3.2. defmacro

The macro *defmacro* makes it easier to define macros because it allows you to name the arguments to the macro call. For example, suppose we find ourselves often writing code like *(setq stack (cons newelt stack))*. We could define a macro named *push* to do this for us. One way to define it is:

```
-> (def push
       (macro (x) (list 'setq (caddr x) (list 'cons (cadr x) (caddr x)))))
push
```

then *(push newelt stack)* will expand to the form mentioned above. The same macro written using defmacro would be:

```
-> (defmacro push (value stack)
     (list 'setq ,stack (list 'cons ,value ,stack)))
push
```

Defmacro allows you to name the arguments of the macro call, and makes the macro definition look more like a function definition.

8.3.3. the backquote character macro

The default syntax for FRANZ LISP has four characters with associated character macros. One is semicolon for comments. Two others are the backquote and comma which are used by the backquote character macro. The fourth is the sharp sign macro

described in the next section.

The backquote macro is used to create lists where many of the elements are fixed (quoted). This makes it very useful for creating macro definitions. In the simplest case, a backquote acts just like a single quote:

```
-> '(a b c d e)
(a b c d e)
```

If a comma precedes an element of a backquoted list then that element is evaluated and its value is put in the list.

```
-> (setq d '(x y z))
(x y z)
-> '(a b c ,d e)
(a b c (x y z) e)
```

If a comma followed by an at sign precedes an element in a backquoted list, then that element is evaluated and spliced into the list with *append*.

```
-> '(a b c ,@d e)
(a b c x y z e)
```

Once a list begins with a backquote, the commas may appear anywhere in the list as this example shows:

```
-> '(a b (c d ,(cdr d)) (e f (g h ,@(cddr d) ,@d)))
(a b (c d (y z)) (e f (g h z x y z)))
```

It is also possible and sometimes even useful to use the backquote macro within itself. As a final demonstration of the backquote macro, we shall define the first and push macros using all the power at our disposal: defmacro and the backquote macro.

```
-> (defmacro first (list) '(car ,list))
first
-> (defmacro push (value stack) '(setq ,stack (cons ,value ,stack)))
stack
```

8.3.4. sharp sign character macro

The sharp sign macro can perform a number of different functions at read time. The character directly following the sharp sign determines which function will be done, and following Lisp s-expressions may serve as arguments.

8.3.4.1. conditional inclusion

If you plan to run one source file in more than one environment then you may want to some pieces of code to be included or not included depending on the environment. The C language uses "#ifdef" and "#ifndef" for this purpose, and Lisp uses "#+" and "#-". The environment that the sharp sign macro checks is the *(status features)* list which is initialized when the Lisp system is built and which may be altered by *(sstatus feature foo)* and *(sstatus nofeature bar)* The form of conditional inclusion is

#+when what

where *when* is either a symbol or an expression involving symbols and the functions *and*, *or*, and *not*. The meaning is that *what* will only be read in if *when* is true. A symbol in *when* is true only if it appears in the *(status features)* list.

```
; suppose we want to write a program which references a file
; and which can run at ucb, ucsd and cmu where the file naming conventions
; are different.
;
-> (defun howold (name)
       (terpr)
       (load #+(or ucb ucsd) "/usr/lib/lisp/ages.l"
             #+cmu "/usr/lisp/doc/ages.l")
       (patom name)
       (patom " is ")
       (print (cdr (assoc name agefile)))
       (patom "years old")
       (terpr))
```

The form

#-when what

is equivalent to

#+(not when) what

8.3.4.2. fixnum character equivalents

When working with fixnum equivalents of characters, it is often hard to remember the number corresponding to a character. The form

#/c

is equivalent to the fixnum representation of character c.

```
; a function which returns t if the user types y else it returns nil.
;
-> (defun yesorno nil
       (progn (ans)
              (setq ans (tyi))
              (cond ((equal ans #/y) t)
                    (t nil))))
```

8.3.4.3. read time evaluation

Occasionally you want to express a constant as a Lisp expression, yet you don't want to pay the penalty of evaluating this expression each time it is referenced. The form

#.expression

evaluates the expression at read time and returns its value.

```
; a function to test if any of bits 1 3 or 12 are set in a fixnum.
;
-> (defun testit (num)
       (cond ((zerop (boole 1 num #.(+ (lsh 1 1) (lsh 1 3) (lsh 1 12))))
              nil)
             (t t)))
```

8.4. fclosures

Fclosures are a type of functional object. The purpose is to remember the values of some variables between invocations of the functional object and to protect this data from being inadvertently overwritten by other Lisp functions. Fortran programs usually exhibit this behavior for their variables. (In fact, some versions of Fortran would require the variables to be in COMMON). Thus it is easy to write a linear congruent random number generator in Fortran, merely by keeping the seed as a variable in the function. It is much more risky to do so in Lisp, since any special variable you picked, might be used by some other function. Fclosures are an attempt to provide most of the same functionality as closures in Lisp Machine Lisp, to users of FRANZ LISP. Fclosures are related to closures in this way:
(fclosure '(a b) 'foo) <==>
(let ((a a) (b b)) (closure '(a b) 'foo))

8.4.1. an example

```
% lisp
Franz Lisp, Opus 38.60
->(defun code (me count)
  (print (list 'in x))
  (setq x (+ 1 x))
  (cond ((greaterp count 1) (funcall me me (sub1 count))))
  (print (list 'out x)))
code
->(defun tester (object count)
  (funcall object object count) (terpri))
tester
->(setq x 0)
0
->(setq z (fclosure '(x) 'code))
fclosure[8]
-> (tester z 3)
(in 0)(in 1)(in 2)(out 3)(out 3)(out 3)
nil
->x
0
```

The function *fclosure* creates a new object that we will call an fclosure, (although it is actually a vector). The fclosure contains a functional object, and a set of symbols and values for the symbols. In the above example, the fclosure functional object is the function code. The set of symbols and values just contains the symbol 'x' and zero, the value of 'x' when the fclosure was created.

When an fclosure is funcall'ed:

1) The Lisp system lambda binds the symbols in the fclosure to their values in the fclosure.
2) It continues the funcall on the functional object of the fclosure.
3) Finally, it un-lambda binds the symbols in the fclosure and at the same time stores the current values of the symbols in the fclosure.

Notice that the fclosure is saving the value of the symbol 'x'. Each time a fclosure is created, new space is allocated for saving the values of the symbols. Thus if we execute fclosure again, over the same function, we can have two independent counters:

```
-> (setq zz (fclosure '(x) 'code))
fclosure[1]
-> (tester zz 2)
(in 0)(in 1)(out 2)(out 2)
-> (tester zz 2)
(in 2)(in 3)(out 4)(out 4)
-> (tester z 3)
(in 3)(in 4)(in 5)(out 6)(out 6)(out 6)
```

8.4.2. useful functions

Here are some quick some summaries of functions dealing with closures. They are more formally defined in §2.8.4. To recap, fclosures are made by *(fclosure 'l_vars 'g_funcobj)*. l_vars is a list of symbols (not containing nil), g_funcobj is any object that can be funcalled. (Objects which can be funcalled, include compiled Lisp functions, lambda expressions, symbols, foreign functions, etc.) In general, if you want a compiled function to be closed over a variable, you must declare the variable to be special within the function. Another example would be:

```
(fclosure '(a b) #'(lambda (x) (plus x a)))
```

Here, the #' construction will make the compiler compile the lambda expression.

There are times when you want to share variables between fclosures. This can be done if the fclosures are created at the same time using *fclosure-list*. The function *fclosure-alist* returns an assoc list giving the symbols and values in the fclosure. The predicate *fclosurep* returns t iff its argument is a fclosure. Other functions imported from Lisp Machine Lisp are *symeval-in-fclosure, let-fclosed,* and *set-in-fclosure*. Lastly, the function *fclosure-function* returns the function argument.

8.4.3. internal structure

Currently, closures are implemented as vectors, with property being the symbol fclosure. The functional object is the first entry. The remaining entries are structures which point to the symbols and values for the closure, (with a reference count to determine if a recursive closure is active).

8.5. foreign subroutines and functions

FRANZ LISP has the ability to dynamically load object files produced by other compilers and to call functions defined in those files. These functions are called *foreign* functions.* There are seven types of foreign functions. They are characterized by the type of result they return, and by differences in the interpretation of their arguments. They come from two families: a group suited for languages which pass arguments by reference (e.g. Fortran), and a group suited for languages which pass arguments by value (e.g. C).

There are four types in the first group:

subroutine
This does not return anything. The Lisp system always returns t after calling a subroutine.

function
This returns whatever the function returns. This must be a valid Lisp object or it may cause the Lisp system to fail.

integer-function
This returns an integer which the Lisp system makes into a fixnum and returns.

real-function
This returns a double precision real number which the Lisp system makes into a flonum and returns.

There are three types in the second group:

c-function
This is like an integer function, except for its different interpretation of arguments.

double-c-function
This is like a real-function.

vector-c-function
This is for C functions which return a structure. The first argument to such functions must be a vector (of type vectori), into which the result is stored. The second Lisp argument becomes the first argument to the C function, and so on

A foreign function is accessed through a binary object just like a compiled Lisp function. The difference is that the discipline field of a binary object for a foreign function is a string whose first character is given in the following table:

*This topic is also discussed in Report PAM-124 of the Center for Pure and Applied Mathematics, UCB, entitled "Parlez-Vous Franz? An Informal Introduction to Interfacing Foreign Functions to Franz LISP", by James R. Larus

letter	type
s	subroutine
f	function
i	integer-function
r	real-function.
c	c-function
v	vector-c-function
d	double-c-function

Two functions are provided for setting-up foreign functions. *Cfasl* loads an object file into the Lisp system and sets up one foreign function binary object. If there are more than one function in an object file, *getaddress* can be used to set up additional foreign function objects.

Foreign functions are called just like other functions, e.g *(funname arg1 arg2)*. When a function in the Fortran group is called, the arguments are evaluated and then examined. List, hunk and symbol arguments are passed unchanged to the foreign function. Fixnum and flonum arguments are copied into a temporary location and a pointer to the value is passed (this is because Fortran uses call by reference and it is dangerous to modify the contents of a fixnum or flonum which something else might point to). If the argument is an array object, the data field of the array object is passed to the foreign function (This is the easiest way to send large amounts of data to and receive large amounts of data from a foreign function). If a binary object is an argument, the entry field of that object is passed to the foreign function (the entry field is the address of a function, so this amounts to passing a function as an argument).

When a function in the C group is called, fixnum and flownum arguments are passed by value. For almost all other arguments, the address is merely provided to the C routine. The only exception arises when you want to invoke a C routine which expects a "structure" argument. Recall that a (rarely used) feature of the C language is the ability to pass structures by value. This copies the structure onto the stack. Since the Franz's nearest equivalent to a C structure is a vector, we provide an escape clause to copy the contents of an immediate-type vector by value. If the property field of a vectori argument, is the symbol "value-structure-argument", then the binary data of this immediate-type vector is copied into the argument list of the C routine.

The method a foreign function uses to access the arguments provided by Lisp is dependent on the language of the foreign function. The following scripts demonstrate how how Lisp can interact with three languages: C, Pascal and Fortran. C and Pascal have pointer types and the first script shows how to use pointers to extract information from Lisp objects. There are two functions defined for each language. The first (cfoo in C, pfoo in Pascal) is given four arguments, a fixnum, a flonum-block array, a hunk of at least two fixnums and a list of at least two fixnums. To demonstrate that the values were passed, each ?foo function prints its arguments (or parts of them). The ?foo function then modifies the second element of the flonum-block array and returns a 3 to Lisp. The second function (cmemq in C, pmemq in Pascal) acts just like the Lisp *memq* function (except it won't work for fixnums whereas the lisp *memq* will work for small fixnums). In the script, typed input is in **bold**, computer output is in roman and comments are in *italic.*

These are the C coded functions
% **cat ch8auxc.c**
/* demonstration of c coded foreign integer-function */

/* the following will be used to extract fixnums out of a list of fixnums */

```
struct listoffixnumscell
{   struct listoffixnumscell *cdr;
    int *fixnum;
};

struct listcell
{           struct listcell *cdr;
            int car;
};

cfoo(a,b,c,d)
int *a;
double b[];
int *c[];
struct listoffixnumscell *d;
{
   printf("a: %d, b[0]: %f, b[1]: %f0, *a, b[0], b[1]);
   printf(" c (first): %d   c (second): %d0,
               *c[0],*c[1]);
   printf(" ( %d %d ... ) ", *(d->fixnum), *(d->cdr->fixnum));
   b[1] = 3.1415926;
   return(3);
}

struct listcell *
cmemq(element,list)
int element;
struct listcell *list;
{
  for( ; list && element != list->car ;  list = list->cdr);
  return(list);
}
```

These are the Pascal coded functions

% **cat ch8auxp.p**

```
type        pinteger = ^integer;
            realarray = array[0..10] of real;
            pintarray = array[0..10] of pinteger;
            listoffixnumscell = record
                                            cdr  : ^listoffixnumscell;
                                            fixnum : pinteger;
                                end;
            plistcell = ^listcell;
            listcell = record
                            cdr : plistcell;
                            car : integer;
                       end;

function pfoo ( var a : integer ;
                    var b : realarray;
                    var c : pintarray;
                    var d : listoffixnumscell) : integer;
begin
  writeln(' a:',a, ' b[0]:', b[0], ' b[1]:', b[1]);
  writeln(' c (first):', c[0]^,' c (second):', c[1]^);
  writeln(' ( ', d.fixnum^, d.cdr^.fixnum^, ' ...) ');
  b[1] := 3.1415926;
  pfoo := 3
end ;

{ the function pmemq looks for the Lisp pointer given as the first argument
  in the list pointed to by the second argument.
  Note that we declare " a : integer " instead of " var a : integer " since
  we are interested in the pointer value instead of what it points to (which
  could be any Lisp object)
}
```

```
function pmemq( a : integer; list : plistcell) : plistcell;
begin
 while (list <> nil) and (list^.car <> a) do list := list^.cdr;
 pmemq := list;
end ;
```

The files are compiled

```
% cc -c ch8auxc.c
1.0u 1.2s 0:15 14% 30+39k 33+20io 147pf+0w
% pc -c ch8auxp.p
3.0u 1.7s 0:37 12% 27+32k 53+32io 143pf+0w
```

```
% lisp
Franz Lisp, Opus 38.60
```

First the files are loaded and we set up one foreign function binary. We have two functions in each file so we must choose one to tell cfasl about. The choice is arbitrary.

```
-> (cfasl 'ch8auxc.o '_cfoo 'cfoo "integer-function")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxc.o -e _cfoo -o /tmp/Li7055.0  -lc
#63000-"integer-function"
-> (cfasl 'ch8auxp.o '_pfoo 'pfoo "integer-function" "-lpc")
/usr/lib/lisp/nld -N -A /tmp/Li7055.0 -T 63200 ch8auxp.o -e _pfoo -o /tmp/Li7055.1 -lpc -lc
#63200-"integer-function"
```

Here we set up the other foreign function binary objects

```
-> (getaddress '_cmemq 'cmemq "function" '_pmemq 'pmemq "function")
#6306c-"function"
```

We want to create and initialize an array to pass to the cfoo function. In this case we create an unnamed array and store it in the value cell of testarr. When we create an array to pass to the Pascal program we will use a named array just to demonstrate the different way that named and unnamed arrays are created and accessed.

```
-> (setq testarr (array nil flonum-block 2))
array[2]
-> (store (funcall testarr 0) 1.234)
1.234
-> (store (funcall testarr 1) 5.678)
5.678
-> (cfoo 385 testarr (hunk 10 11 13 14) '(15 16 17))
a: 385, b[0]: 1.234000, b[1]: 5.678000
 c (first): 10   c (second): 11
 ( 15 16 ... )
 3
```

Note that cfoo has returned 3 as it should. It also had the side effect of changing the second value of the array to 3.1415926 which check next.

```
-> (funcall testarr 1)
3.1415926
```

In preparation for calling pfoo we create an array.

```
-> (array test flonum-block 2)
array[2]
-> (store (test 0) 1.234)
1.234
-> (store (test 1) 5.678)
5.678
-> (pfoo 385 (getd 'test) (hunk 10 11 13 14) '(15 16 17))
 a:      385 b[0]:  1.23400000000000E+00 b[1]:  5.67800000000000E+00
 c (first):       10 c (second):       11
 (       15       16 ...)
3
-> (test 1)
3.1415926
```

Now to test out the memq's

```
-> (cmemq 'a '(b c a d e f))
(a d e f)
-> (pmemq 'e '(a d f g a x))
nil
```

The Fortran example will be much shorter since in Fortran you can't follow pointers as you can in other languages. The Fortran function ffoo is given three arguments: a fixnum, a fixnum-block array and a flonum. These arguments are printed out to verify that they made it and then the first value of the array is modified. The function returns a double precision value which is converted to a flonum by lisp and printed. Note that the entry point corresponding to the Fortran function ffoo is _ffoo_ as opposed to the C and Pascal convention of preceding the name with an underscore.

```
% cat ch8auxf.f
          double precision function ffoo(a,b,c)
          integer a,b(10)
          double precision c
          print 2,a,b(1),b(2),c
2         format(' a=',i4,', b(1)=',i5,', b(2)=',i5,' c=',f6.4)
          b(1) = 22
          ffoo = 1.23456
          return
          end
% f77 -c ch8auxf.f
ch8auxf.f:
  ffoo:
0.9u 1.8s 0:12 22% 20+22k 54+48io 158pf+0w
% lisp
Franz Lisp, Opus 38.60
-> (cfasl 'ch8auxf.o '_ffoo_ 'ffoo "real-function" "-lF77 -lI77")
/usr/lib/lisp/nld -N -A /usr/local/lisp -T 63000 ch8auxf.o -e _ffoo_
-o /tmp/Li11066.0 -lF77 -lI77 -lc
#6307c-"real-function"

-> (array test fixnum-block 2)
array[2]
-> (store (test 0) 10)
10
-> (store (test 1) 11)
11
-> (ffoo 385 (getd 'test) 5.678)
 a= 385, b(1)=   10, b(2)=   11 c=5.6780
1.234559893608093
-> (test 0)
22
```

CHAPTER 9

Arrays and Vectors

Arrays and vectors are two means of expressing aggregate data objects in FRANZ LISP. Vectors may be thought of as sequences of data. They are intended as a vehicle for user-defined data types. This use of vectors is still experimental and subject to revision. As a simple data structure, they are similar to hunks and strings. Vectors are used to implement closures, and are useful to communicate with foreign functions. Both of these topics were discussed in Chapter 8. Later in this chapter, we describe the current implementation of vectors, and will advise the user what is most likely to change.

Arrays in FRANZ LISP provide a programmable data structure access mechanism. One possible use for FRANZ LISP arrays is to implement Maclisp style arrays which are simple vectors of fixnums, flonums or general lisp values. This is described in more detail in §9.3 but first we will describe how array references are handled by the lisp system.

The structure of an array object is given in §1.3.10 and reproduced here for your convenience.

Subpart name	Get value	Set value	Type
access function	getaccess	putaccess	binary, list or symbol
auxiliary	getaux	putaux	lispval
data	arrayref	replace set	block of contiguous lispval
length	getlength	putlength	fixnum
delta	getdelta	putdelta	fixnum

9.1. general arrays Suppose the evaluator is told to evaluate *(foo a b)* and the function cell of the symbol foo contains an array object (which we will call foo_arr_obj). First the evaluator will evaluate and stack the values of *a* and *b*. Next it will stack the array object foo_arr_obj. Finally it will call the access function of foo_arr_obj. The access function should be a lexpr[†] or a symbol whose function cell contains a lexpr. The access function is responsible for locating and returning a value from the array. The array access function is free to interpret the arguments as it wishes. The Maclisp compatible array access function which is provided in the standard FRANZ LISP system interprets the arguments as subscripts in the same way as languages like Fortran and Pascal.

The array access function will also be called upon to store elements in the array. For example, *(store (foo a b) c)* will automatically expand to (foo c a b) and when the evaluator is called to evaluate this, it will evaluate the arguments *c*, *b* and *a*. Then it will stack the array object (which is stored in the function cell of foo) and call the array access function with (now) four arguments. The array access function must be able to tell this is a store operation, which it can do by checking the number of arguments it has

[†]A lexpr is a function which accepts any number of arguments which are evaluated before the function is called.

been given (a lexpr can do this very easily).

9.2. subparts of an array object An array is created by allocating an array object with *marray* and filling in the fields. Certain lisp functions interpret the values of the subparts of the array object in special ways as described in the following text. Placing illegal values in these subparts may cause the lisp system to fail.

9.2.1. access function The purpose of the access function has been described above. The contents of the access function should be a lexpr, either a binary (compiled function) or a list (interpreted function). It may also be a symbol whose function cell contains a function definition. This subpart is used by *eval*, *funcall*, and *apply* when evaluating array references.

9.2.2. auxiliary This can be used for any purpose. If it is a list and the first element of that list is the symbol unmarked_array then the data subpart will not be marked by the garbage collector (this is used in the Maclisp compatible array package and has the potential for causing strange errors if used incorrectly).

9.2.3. data This is either nil or points to a block of data space allocated by *segment* or *small-segment.*

9.2.4. length This is a fixnum whose value is the number of elements in the data block. This is used by the garbage collector and by *arrayref* to determine if your index is in bounds.

9.2.5. delta This is a fixnum whose value is the number of bytes in each element of the data block. This will be four for an array of fixnums or value cells, and eight for an array of flonums. This is used by the garbage collector and *arrayref* as well.

9.3. The Maclisp compatible array package

A Maclisp style array is similar to what is known as arrays in other languages: a block of homogeneous data elements which is indexed by one or more integers called subscripts. The data elements can be all fixnums, flonums or general lisp objects. An array is created by a call to the function *array* or **array*. The only difference is that **array* evaluates its arguments. This call: *(array foo t 3 5)* sets up an array called foo of dimensions 3 by 5. The subscripts are zero based. The first element is *(foo 0 0)*, the next is *(foo 0 1)* and so on up to *(foo 2 4)*. The t indicates a general lisp object array which means each element of foo can be any type. Each element can be any type since all that is stored in the array is a pointer to a lisp object, not the object itself. *Array* does this by allocating an array object with *marray* and then allocating a segment of 15

consecutive value cells with *small-segment* and storing a pointer to that segment in the data subpart of the array object. The length and delta subpart of the array object are filled in (with 15 and 4 respectively) and the access function subpart is set to point to the appropriate array access function. In this case there is a special access function for two dimensional value cell arrays called arrac-twoD, and this access function is used. The auxiliary subpart is set to (t 3 5) which describes the type of array and the bounds of the subscripts. Finally this array object is placed in the function cell of the symbol foo. Now when *(foo 1 3)* is evaluated, the array access function is invoked with three arguments: 1, 3 and the array object. From the auxiliary field of the array object it gets a description of the particular array. It then determines which element *(foo 1 3)* refers to and uses arrayref to extract that element. Since this is an array of value cells, what arrayref returns is a value cell whose value is what we want, so we evaluate the value cell and return it as the value of *(foo 1 3)*.

In Maclisp the call *(array foo fixnum 25)* returns an array whose data object is a block of 25 memory words. When fixnums are stored in this array, the actual numbers are stored instead of pointers to the numbers as is done in general lisp object arrays. This is efficient under Maclisp but inefficient in FRANZ LISP since every time a value was referenced from an array it had to be copied and a pointer to the copy returned to prevent aliasing†. Thus t, fixnum and flonum arrays are all implemented in the same manner. This should not affect the compatibility of Maclisp and FRANZ LISP. If there is an application where a block of fixnums or flonums is required, then the exact same effect of fixnum and flonum arrays in Maclisp can be achieved by using fixnum-block and flonum-block arrays. Such arrays are required if you want to pass a large number of arguments to a Fortran or C coded function and then get answers back.

The Maclisp compatible array package is just one example of how a general array scheme can be implemented. Another type of array you could implement would be hashed arrays. The subscript could be anything, not just a number. The access function would hash the subscript and use the result to select an array element. With the generality of arrays also comes extra cost; if you just want a simple aggregate of (less than 128) general lisp objects you would be wise to look into using hunks.

9.4. vectors Vectors were invented to fix two shortcommings with hunks. They can be longer than 128 elements. They also have a tag associated with them, which is intended to say, for example, "Think of me as an *Blobit*." Thus a **vector** is an arbitrary sized hunk with a property list.

Continuing the example, the lisp kernel may not know how to print out or evaluate *blobits*, but this is information which will be common to all *blobits*. On the other hand, for each individual blobits there are particulars which are likely to change, (height, weight, eye-color). This is the part that would previously have been stored in the individual entries in the hunk, and are stored in the data slots of the vector. Once again we summarize the structure of a vector in tabular form:

---

†Aliasing is when two variables are share the same storage location. For example if the copying mentioned weren't done then after *(setq x (foo 2))* was done, the value of x and (foo 2) would share the same location. Then should the value of (foo 2) change, x's value would change as well. This is considered dangerous and as a result pointers are never returned into the data space of arrays.

Subpart name	Get value	Set value	Type
datum[*i*]	vref	vset	lispval
property	vprop	vsetprop vputprop	lispval
size	vsize	—	fixnum

Vectors are created specifying size and optional fill value using the function (*new-vector* 'x_size ['g_fill ['g_prop]]), or by initial values: (*vector* ['g_val ...]).

9.5. anatomy of vectors There are some technical details about vectors, that the user should know:

9.5.1. size The user is not free to alter this. It is noted when the vector is created, and is used by the garbage collector. The garbage collector will coallesce two free vectors, which are neighbors in the heap. Internally, this is kept as the number of bytes of data. Thus, a vector created by (*vector* 'foo), has a size of 4.

9.5.2. property Currently, we expect the property to be either a symbol, or a list whose first entry is a symbol. The symbols **fclosure** and **structure-value-argument** are magic, and their effect is described in Chapter 8. If the property is a (non-null) symbol, the vector will be printed out as <symbol>[<size>]. Another case is if the property is actually a (disembodied) property-list, which contains a value for the indicator **print**. The value is taken to be a Lisp function, which the printer will invoke with two arguments: the vector and the current output port. Otherwise, the vector will be printed as vector[<size>]. We have vague (as yet unimplemented) ideas about similar mechanisms for evaluation properties. Users are cautioned against putting anything other than nil in the property entry of a vector.

9.5.3. internal order In memory, vectors start with a longword containing the size (which is immediate data within the vector). The next cell contains a pointer to the property. Any remaining cells (if any) are for data. Vectors are handled differently from any other object in FRANZ LISP, in that a pointer to a vector is pointer to the first data cell, i.e. a pointer to the *third* longword of the structure. This was done for efficiency in compiled code and for uniformity in referencing immediate-vectors (described below). The user should never return a pointer to any other part of a vector, as this may cause the garbage collector to follow an invalid pointer.

9.6. immediate-vectors Immediate-vectors are similar to vectors. They differ, in that binary data are stored in space directly within the vector. Thus the garbage collector will preserve the vector itself (if used), and will only traverse the property cell. The data may be referenced as longwords, shortwords, or even bytes. Shorts and bytes are returned sign-extended. The compiler open-codes such references, and will avoid boxing the resulting integer data, where possible. Thus, immediate vectors may be used for efficiently processing character data. They are also useful in storing results from

functions written in other languages.

Subpart name	Get value	Set value	Type
datum[*i*]	vrefi-byte vrefi-word vrefi-long	vseti-byte vseti-word vseti-long	fixnum fixnum fixnum
property	vprop	vsetprop vputprop	lispval
size	vsize vsize-byte vsize-word	–	fixnum fixnum fixnum

To create immediate vectors specifying size and fill data, you can use the functions *new-vectori-byte*, *new-vectori-word*, or *new-vectori-long*. You can also use the functions *vectori-byte*, *vectori-word*, or *vectori-long*. All of these functions are described in chapter 2.

CHAPTER 10

Exception Handling

10.1. Errset and Error Handler Functions

FRANZ LISP allows the user to handle in a number of ways the errors which arise during computation. One way is through the use of the *errset* function. If an error occurs during the evaluation of the *errset*'s first argument, then the locus of control will return to the errset which will return nil (except in special cases, such as *err*). The other method of error handling is through an error handler function. When an error occurs, the error handler is called and is given as an argument a description of the error which just occurred. The error handler may take one of the following actions:

(1) it could take some drastic action like a *reset* or a *throw*.

(2) it could, assuming that the error is continuable, return to the function which noticed the error. The error handler indicates that it wants to return a value from the error by returning a list whose *car* is the value it wants to return.

(3) it could decide not to handle the error and return a non-list to indicate this fact.

10.2. The Anatomy of an error

Each error is described by a list of these items:

(1) error type - This is a symbol which indicates the general classification of the error. This classification may determine which function handles this error.

(2) unique id - This is a fixnum unique to this error.

(3) continuable - If this is non-nil then this error is continuable. There are some who feel that every error should be continuable and the reason that some (in fact most) errors in FRANZ LISP are not continuable is due to the laziness of the programmers.

(4) message string - This is a symbol whose print name is a message describing the error.

(5) data - There may be from zero to three lisp values which help describe this particular error. For example, the unbound variable error contains one datum value, the symbol whose value is unbound. The list describing that error might look like:

(ER%misc 0 t |Unbound Variable:| foobar)

10.3. Error handling algorithm

This is the sequence of operations which is done when an error occurs:

(1) If the symbol **ER%all** has a non nil value then this value is the name of an error handler function. That function is called with a description of the error. If that function returns (and of course it may choose not to) and the value is a list and this error is continuable, then we return the *car* of the list to the function which

called the error. Presumably the function will use this value to retry the operation. On the other hand, if the error handler returns a non list, then it has chosen not to handle this error, so we go on to step (2). Something special happens before we call the **ER%all** error handler which does not happen in any of the other cases we will describe below. To help insure that we don't get infinitely recursive errors if **ER%all** is set to a bad value, the value of **ER%all** is set to nil before the handler is called. Thus it is the responsibility of the **ER%all** handler to 'reenable' itself by storing its name in **ER%all.**

(2) Next the specific error handler for the type of error which just occurred is called (if one exists) to see if it wants to handle the error. The names of the handlers for the specific types of errors are stored as the values of the symbols whose names are the types. For example the handler for miscellaneous errors is stored as the value of **ER%misc.** Of course, if **ER%misc** has a value of nil, then there is no error handler for this type of error. Appendix B contains list of all error types. The process of classifying the errors is not complete and thus most errors are lumped into the **ER%misc** category. Just as in step (1), the error handler function may choose not to handle the error by returning a non-list, and then we go to step (3).

(3) Next a check is made to see if there is an *errset* surrounding this error. If so the second argument to the *errset* call is examined. If the second argument was not given or is non nil then the error message associated with this error is printed. Finally the stack is popped to the context of the *errset* and then the *errset* returns nil. If there was no *errset* we go to step (4).

(4) If the symbol **ER%tpl** has a value then it is the name of an error handler which is called in a manner similar to that discussed above. If it chooses not to handle the error, we go to step (5).

(5) At this point it has been determined that the user doesn't want to handle this error. Thus the error message is printed out and a *reset* is done to send the flow of control to the top-level.

To summarize the error handling system: When an error occurs, you have two chances to handle it before the search for an *errset* is done. Then, if there is no *errset*, you have one more chance to handle the error before control jumps to the top level. Every error handler works in the same way: It is given a description of the error (as described in the previous section). It may or may not return. If it returns, then it returns either a list or a non-list. If it returns a list and the error is continuable, then the *car* of the list is returned to the function which noticed the error. Otherwise the error handler has decided not to handle the error and we go on to something else.

10.4. Default aids

There are two standard error handlers which will probably handle the needs of most users. One of these is the lisp coded function *break-err-handler* which is the default value of **ER%tpl.** Thus when all other handlers have ignored an error, *break-err-handler* will take over. It will print out the error message and go into a read-eval-print loop. The other standard error handler is *debug-err-handler.* This handler is designed to be connected to **ER%all**and is useful if your program uses *errset* and you want to look at the error before it is thrown up to the *errset.*

10.5. Autoloading

When *eval, apply* or *funcall* are told to call an undefined function, an **ER%undef** error is signaled. The default handler for this error is *undef-func-handler.* This function

checks the property list of the undefined function for the indicator autoload. If present, the value of that indicator should be the name of the file which contains the definition of the undefined function. *Undef-func-handler* will load the file and check if it has defined the function which caused the error. If it has, the error handler will return and the computation will continue as if the error did not occur. This provides a way for the user to tell the lisp system about the location of commonly used functions. The trace package sets up an autoload property to point to /usr/lib/lisp/trace.

10.6. Interrupt processing

The UNIX operating system provides one user interrupt character which defaults to ^C.[†] The user may select a lisp function to run when an interrupt occurs. Since this interrupt could occur at any time, and in particular could occur at a time when the internal stack pointers were in an inconsistent state, the processing of the interrupt may be delayed until a safe time. When the first ^C is typed, the lisp system sets a flag that an interrupt has been requested. This flag is checked at safe places within the interpreter and in the *qlinker* function. If the lisp system doesn't respond to the first ^C, another ^C should be typed. This will cause all of the transfer tables to be cleared forcing all calls from compiled code to go through the *qlinker* function where the interrupt flag will be checked. If the lisp system still doesn't respond, a third ^C will cause an immediate interrupt. This interrupt will not necessarily be in a safe place so the user should *reset* the lisp system as soon as possible.

[†]Actually there are two but the lisp system does not allow you to catch the QUIT interrupt.

CHAPTER 11

The Joseph Lister Trace Package

The Joseph Lister† Trace package is an important tool for the interactive debugging of a Lisp program. It allows you to examine selected calls to a function or functions, and optionally to stop execution of the Lisp program to examine the values of variables.

The trace package is a set of Lisp programs located in the Lisp program library (usually in the file /usr/lib/lisp/trace.l). Although not normally loaded in the Lisp system, the package will be loaded in when the first call to *trace* is made.

(trace [ls_arg1 ...])

WHERE: the form of the ls_arg*i* is described below.

RETURNS: a list of the function sucessfully modified for tracing. If no arguments are given to *trace*, a list of all functions currently being traced is returned.

SIDE EFFECT: The function definitions of the functions to trace are modified.

The ls_arg*i* can have one of the following forms:

foo - when foo is entered and exited, the trace information will be printed.

(foo break) - when foo is entered and exited the trace information will be printed. Also, just after the trace information for foo is printed upon entry, you will be put in a special break loop. The prompt is 'T>' and you may type any Lisp expression, and see its value printed. The *i*th argument to the function just called can be accessed as (arg *i*). To leave the trace loop, just type ^D or (tracereturn) and execution will continue. Note that ^D will work only on UNIX systems.

(foo if expression) - when foo is entered and the expression evaluates to non-nil, then the trace information will be printed for both exit and entry. If expression evaluates to nil, then no trace information will be printed.

(foo ifnot expression) - when foo is entered and the expression evaluates to nil, then the trace information will be printed for both entry and exit. If both **if** and **ifnot** are specified, then the **if** expression must evaluate to non nil AND the **ifnot** expression must evaluate to nil for the trace information to be printed out.

(foo evalin expression) - when foo is entered and after the entry trace information is printed, expression will be evaluated. Exit trace information will be printed when foo exits.

---

†*Lister, Joseph* 1st Baron Lister of Lyme Regis, 1827-1912; English surgeon: introduced antiseptic surgery.

(foo evalout expression) - when foo is entered, entry trace information will be printed. When foo exits, and before the exit trace information is printed, expression will be evaluated.

(foo evalinout expression) - this has the same effect as (trace (foo evalin expression evalout expression)).

(foo lprint) - this tells *trace* to use the level printer when printing the arguments to and the result of a call to foo. The level printer prints only the top levels of list structure. Any structure below three levels is printed as a &. This allows you to trace functions with massive arguments or results.

The following trace options permit one to have greater control over each action which takes place when a function is traced. These options are only meant to be used by people who need special hooks into the trace package. Most people should skip reading this section.

(foo traceenter tefunc) - this tells *trace* that the function to be called when foo is entered is tefunc. tefunc should be a lambda of two arguments, the first argument will be bound to the name of the function being traced, foo in this case. The second argument will be bound to the list of arguments to which foo should be applied. The function tefunc should print some sort of "entering foo" message. It should not apply foo to the arguments, however. That is done later on.

(foo traceexit txfunc) - this tells *trace* that the function to be called when foo is exited is txfunc. txfunc should be a lambda of two arguments, the first argument will be bound to the name of the function being traced, foo in this case. The second argument will be bound to the result of the call to foo. The function txfunc should print some sort of "exiting foo" message.

(foo evfcn evfunc) - this tells *trace* that the form evfunc should be evaluated to get the value of foo applied to its arguments. This option is a bit different from the other special options since evfunc will usually be an expression, not just the name of a function, and that expression will be specific to the evaluation of function foo. The argument list to be applied will be available as T-arglist.

(foo printargs prfunc) - this tells *trace* to used prfunc to print the arguments to be applied to the function foo. prfunc should be a lambda of one argument. You might want to use this option if you wanted a print function which could handle circular lists. This option will work only if you do not specify your own **traceenter** function. Specifying the option **lprint** is just a simple way of changing the printargs function to the level printer.

(foo printres prfunc) - this tells *trace* to use prfunc to print the result of evaluating foo. prfunc should be a lambda of one argument. This option will work only if you do not specify your own **traceexit** function. Specifying the option **lprint** changes printres to the level printer.

You may specify more than one option for each function traced. For example:

(trace (foo if (eq 3 (arg 1)) break lprint) (bar evalin (print xyzzy)))

This tells *trace* to trace two more functions, foo and bar. Should foo be called with the first argument *eq* to 3, then the entering foo message will be printed with the level printer. Next it will enter a trace break loop, allowing you to evaluate any lisp expressions. When you exit the trace break loop, foo will be applied to its arguments and the resulting value will be printed, again using the level printer. Bar is also traced, and each time bar is entered, an entering bar message will be printed and then the value of xyzzy will be printed. Next bar will be applied to its arguments and the result will be printed. If you tell *trace* to trace a function which is already traced, it will first *untrace* it. Thus if you want to specify more than one trace option for a function, you must do it all at once. The following is *not* equivalent to the preceding call to *trace* for foo:

(trace (foo if (eq 3 (arg 1))) (foo break) (foo lprint))

In this example, only the last option, lprint, will be in effect.

If the symbol $tracemute is given a non nil value, printing of the function name and arguments on entry and exit will be surpressed. This is particularly useful if the function you are tracing fails after many calls to it. In this case you would tell *trace* to trace the function, set $tracemute to t, and begin the computation. When an error occurs you can use *tracedump* to print out the current trace frames.

Generally the trace package has its own internal names for the the lisp functions it uses, so that you can feel free to trace system functions like *cond* and not worry about adverse interaction with the actions of the trace package. You can trace any type of function: lambda, nlambda, lexpr or macro whether compiled or interpreted and you can even trace array references (however you should not attempt to store in an array which has been traced).

When tracing compiled code keep in mind that many function calls are translated directly to machine language or other equivalent function calls. A full list of open coded functions is listed at the beginning of the liszt compiler source. *Trace* will do a *(sstatus translink nil)* to insure that the new traced definitions it defines are called instead of the old untraced ones. You may notice that compiled code will run slower after this is done.

(traceargs s_func [x_level])

WHERE: if x_level is missing it is assumed to be 1.

RETURNS: the arguments to the x_level*th* call to traced function s_func are returned.

(tracedump)

SIDE EFFECT: the currently active trace frames are printed on the terminal. returns a list of functions untraced.

(untrace [s_arg1 ...])

RETURNS: a list of the functions which were untraced.

NOTE: if no arguments are given, all functions are untraced.

SIDE EFFECT: the old function definitions of all traced functions are restored except in the case where it appears that the current definition of a function was not created by trace.

CHAPTER 12

Liszt - the lisp compiler

12.1. General strategy of the compiler

The purpose of the lisp compiler, Liszt, is to create an object module which when brought into the lisp system using *fasl* will have the same effect as bringing in the corresponding lisp coded source module with *load* with one important exception, functions will be defined as sequences of machine language instructions, instead of lisp S-expressions. Liszt is not a function compiler, it is a *file* compiler. Such a file can contain more than function definitions; it can contain other lisp S-expressions which are evaluated at load time. These other S-expressions will also be stored in the object module produced by Liszt and will be evaluated at fasl time.

As is almost universally true of Lisp compilers, the main pass of Liszt is written in Lisp. A subsequent pass is the assembler, for which we use the standard UNIX assembler.

12.2. Running the compiler

The compiler is normally run in this manner:

% **liszt foo**

will compile the file foo.l or foo (the preferred way to indicate a lisp source file is to end the file name with '.l'). The result of the compilation will be placed in the file foo.o if no fatal errors were detected. All messages which Liszt generates go to the standard output. Normally each function name is printed before it is compiled (the −q option suppresses this).

12.3. Special forms

Liszt makes one pass over the source file. It processes each form in this way:

12.3.1. macro expansion

If the form is a macro invocation (i.e it is a list whose car is a symbol whose function binding is a macro), then that macro invocation is expanded. This is repeated until the top level form is not a macro invocation. When Liszt begins, there are already some macros defined, in fact some functions (such as defun) are actually macros. The user may define his own macros as well. For a macro to be used it must be defined in the Lisp system in which Liszt runs.

12.3.2. classification

After all macro expansion is done, the form is classified according to its *car* (if the form is not a list, then it is classified as an *other*).

12.3.2.1. eval-when

The form of eval-when is *(eval-when (time1 time2 ...) form1 form2 ...)* where the time*i* are one of *eval*, *compile*, or *load*. The compiler examines the form*i* in sequence and the action taken depends on what is in the time list. If *compile* is in the list then the compiler will invoke *eval* on each form*i* as it examines it. If *load* is in the list then the compile will recursively call itself to compile each form*i* as it examines it. Note that if *compile* and *load* are in the time list, then the compiler will both evaluate and compile each form. This is useful if you need a function to be defined in the compiler at both compile time (perhaps to aid macro expansion) and at run time (after the file is *fasl*ed in).

12.3.2.2. declare

Declare is used to provide information about functions and variables to the compiler. It is (almost) equivalent to *(eval-when (compile) ...)*. You may declare functions to be one of three types: lambda (*expr), nlambda (*fexpr), lexpr (*lexpr). The names in parenthesis are the Maclisp names and are accepted by the compiler as well (and not just when the compiler is in Maclisp mode). Functions are assumed to be lambdas until they are declared otherwise or are defined differently. The compiler treats calls to lambdas and lexprs equivalently, so you needn't worry about declaring lexprs either. It is important to declare nlambdas or define them before calling them. Another attribute you can declare for a function is localf which makes the function 'local'. A local function's name is known only to the functions defined within the file itself. The advantage of a local function is that is can be entered and exited very quickly and it can have the same name as a function in another file and there will be no name conflict.

Variables may be declared special or unspecial. When a special variable is lambda bound (either in a lambda, prog or do expression), its old value is stored away on a stack for the duration of the lambda, prog or do expression. This takes time and is often not necessary. Therefore the default classification for variables is unspecial. Space for unspecial variables is dynamically allocated on a stack. An unspecial variable can only be accessed from within the function where it is created by its presence in a lambda, prog or do expression variable list. It is possible to declare that all variables are special as will be shown below.

You may declare any number of things in each declare statement. A sample declaration is

```
(declare
    (lambda func1 func2)
    (*fexpr func3)
    (*lexpr func4)
    (localf func5)
    (special var1 var2 var3)
    (unspecial var4))
```

You may also declare all variables to be special with *(declare (specials t))*. You may declare that macro definitions should be compiled as well as evaluated at compile time by *(declare (macros t))*. In fact, as was mentioned above, declare is

much like *(eval-when (compile) ...)*. Thus if the compiler sees *(declare (foo bar))* and foo is defined, then it will evaluate *(foo bar)*. If foo is not defined then an undefined declare attribute warning will be issued.

12.3.2.3. (progn 'compile form1 form2 ... formn)

When the compiler sees this it simply compiles form1 through formn as if they too were seen at top level. One use for this is to allow a macro at top-level to expand into more than one function definition for the compiler to compile.

12.3.2.4. include/includef

Include and *includef* cause another file to be read and compiled by the compiler. The result is the same as if the included file were textually inserted into the original file. The only difference between *include* and *includef* is that include doesn't evaluate its argument and includef does. Nested includes are allowed.

12.3.2.5. def

A def form is used to define a function. The macros *defun* and *defmacro* expand to a def form. If the function being defined is a lambda, nlambda or lexpr then the compiler converts the lisp definition to a sequence of machine language instructions. If the function being defined is a macro, then the compiler will evaluate the definition, thus defining the macro withing the running Lisp compiler. Furthermore, if the variable *macros* is set to a non nil value, then the macro definition will also be translated to machine language and thus will be defined when the object file is fasled in. The variable *macros* is set to t by *(declare (macros t))*.

When a function or macro definition is compiled, macro expansion is done whenever possible. If the compiler can determine that a form would be evaluated if this function were interpreted then it will macro expand it. It will not macro expand arguments to a nlambda unless the characteristics of the nlambda is known (as is the case with *cond)*. The map functions (*map*, *mapc*, *mapcar*, and so on) are expanded to a *do* statement. This allows the first argument to the map function to be a lambda expression which references local variables of the function being defined.

12.3.2.6. other forms

All other forms are simply stored in the object file and are evaluated when the file is *fasl*ed in.

12.4. Using the compiler

The previous section describes exactly what the compiler does with its input. Generally you won't have to worry about all that detail as files which work interpreted will work compiled. Following is a list of steps you should follow to insure that a file will compile correctly.

[1] Make sure all macro definitions precede their use in functions or other macro definitions. If you want the macros to be around when you *fasl* in the object file you should include this statement at the beginning of the file: *(declare (macros t))*

[2] Make sure all nlambdas are defined or declared before they are used. If the compiler comes across a call to a function which has not been defined in the current file, which does not currently have a function binding, and whose type has not been declared then it will assume that the function needs its arguments evaluated (i.e. it is a lambda or lexpr) and will generate code accordingly. This means that you do not have to declare nlambda functions like *status* since they have an nlambda function binding.

[3] Locate all variables which are used for communicating values between functions. These variables must be declared special at the beginning of a file. In most cases there won't be many special declarations but if you fail to declare a variable special that should be, the compiled code could fail in mysterious ways. Let's look at a common problem, assume that a file contains just these three lines:

```
(def aaa (lambda (glob loc) (bbb loc)))
(def bbb (lambda (myloc) (add glob myloc)))
(def ccc (lambda (glob loc) (bbb loc)))
```

We can see that if we load in these two definitions then (aaa 3 4) is the same as (add 3 4) and will give us 7. Suppose we compile the file containing these definitions. When Liszt compiles aaa, it will assume that both glob and loc are local variables and will allocate space on the temporary stack for their values when aaa is called. Thus the values of the local variables glob and loc will not affect the values of the symbols glob and loc in the Lisp system. Now Liszt moves on to function bbb. Myloc is assumed to be local. When it sees the add statement, it find a reference to a variable called glob. This variable is not a local variable to this function and therefore glob must refer to the value of the symbol glob. Liszt will automatically declare glob to be special and it will print a warning to that effect. Thus subsequent uses of glob will always refer to the symbol glob. Next Liszt compiles ccc and treats glob as a special and loc as a local. When the object file is *fasl*'ed in, and (ccc 3 4) is evaluated, the symbol glob will be lambda bound to 3 bbb will be called and will return 7. However (aaa 3 4) will fail since when bbb is called, glob will be unbound. What should be done here is to put *(declare (special glob)* at the beginning of the file.

[4] Make sure that all calls to *arg* are within the lexpr whose arguments they reference. If *foo* is a compiled lexpr and it calls *bar* then *bar* cannot use *arg* to get at *foo*'s arguments. If both *foo* and *bar* are interpreted this will work however. The macro *listify* can be used to put all of some of a lexprs arguments in a list which then can be passed to other functions.

12.5. Compiler options

The compiler recognizes a number of options which are described below. The options are typed anywhere on the command line preceded by a minus sign. The entire command line is scanned and all options recorded before any action is taken. Thus

```
% liszt -mx foo
% liszt -m -x foo
% liszt foo -mx
```

are all equivalent. Before scanning the command line for options, liszt looks for in the

environment for the variable LISZT, and if found scans its value as if it was a string of options. The meaning of the options are:

C The assembler language output of the compiler is commented. This is useful when debugging the compiler and is not normally done since it slows down compilation.

I The next command line argument is taken as a filename, and loaded prior to compilation.

e Evaluate the next argument on the command line before starting compilation. For example
% liszt -e '(setq foobar "foo string")' foo
will evaluate the above s-expression. Note that the shell requires that the arguments be surrounded by single quotes.

i Compile this program in interlisp compatibility mode. This is not implemented yet.

m Compile this program in Maclisp mode. The reader syntax will be changed to the Maclisp syntax and a file of macro definitions will be loaded in (usually named /usr/lib/lisp/machacks). This switch brings us sufficiently close to Maclisp to allow us to compile Macsyma, a large Maclisp program. However Maclisp is a moving target and we can't guarantee that this switch will allow you to compile any given program.

o Select a different object or assembler language file name. For example
% liszt foo -o xxx.o
will compile foo and into xxx.o instead of the default foo.o, and
% liszt bar -S -o xxx.s
will compile to assembler language into xxx.s instead of bar.s.

p place profiling code at the beginning of each non-local function. If the lisp system is also created with profiling in it, this allows function calling frequency to be determined (see *prof(1)*)

q Run in quiet mode. The names of functions being compiled and various "Note"'s are not printed.

Q print compilation statistics and warn of strange constructs. This is the inverse of the **q** switch and is the default.

r place bootstrap code at the beginning of the object file, which when the object file is executed will cause a lisp system to be invoked and the object file *fasl*ed in. This is known as 'autorun' and is described below.

S Create an assembler language file only.
% liszt -S foo
will create the file assembler language file foo.s and will not attempt to assemble it. If this option is not specified, the assembler language file will be put in the temporary disk area under a automatically generated name based on the lisp compiler's process id. Then if there are no compilation errors, the assembler will be invoked to assemble the file.

T Print the assembler language output on the standard output file. This is useful when debugging the compiler.

u Run in UCI-Lisp mode. The character syntax is changed to that of UCI-Lisp and a UCI-Lisp compatibility package of macros is read in.

w Suppress warning messages.

x Create an cross reference file.
% liszt -x foo
not only compiles foo into foo.o but also generates the file foo.x . The file foo.x is lisp readable and lists for each function all functions which that function could call. The program lxref reads one or more of these ".x" files and produces a human

readable cross reference listing.

12.6. autorun

The object file which liszt writes does not contain all the functions necessary to run the lisp program which was compiled. In order to use the object file, a lisp system must be started and the object file *fasl*ed in. When the -r switch is given to liszt, the object file created will contain a small piece of bootstrap code at the beginning, and the object file will be made executable. Now, when the name of the object file is given to the UNIX command interpreter (shell) to run, the bootstrap code at the beginning of the object file will cause a lisp system to be started and the first action the lisp system will take is to *fasl* in the object file which started it. In effect the object file has created an environment in which it can run.

Autorun is an alternative to *dumplisp*. The advantage of autorun is that the object file which starts the whole process is typically small, whereas the minimum *dumplisp*ed file is very large (one half megabyte). The disadvantage of autorun is that the file must be *fasl*ed into a lisp each time it is used whereas the file which *dumplisp* creates can be run as is. liszt itself is a *dumplisp*ed file since it is used so often and is large enough that too much time would be wasted *fasl*ing it in each time it was used. The lisp cross reference program, lxref, uses *autorun* since it is a small and rarely used program.

In order to have the program *fasl*ed in begin execution (rather than starting a lisp top level), the value of the symbol user-top-level should be set to the name of the function to get control. An example of this is shown next.

---

we want to replace the unix date program with one written in lisp.

```
% cat lispdate.l
(defun mydate nil
   (patom "The date is ")
   (patom (status ctime))
   (terpr)
   (exit 0))
(setq user-top-level 'mydate)

% liszt -r lispdate
Compilation begins with Lisp Compiler 5.2
source: lispdate.l, result: lispdate.o
mydate
%Note: lispdate.l: Compilation complete
%Note: lispdate.l:  Time: Real: 0:3, CPU: 0:0.28, GC: 0:0.00 for 0 gcs
%Note: lispdate.l: Assembly begins
%Note: lispdate.l: Assembly completed successfully
3.0u 2.0s 0:17 29%

 We change the name to remove the ".o", (this isn't necessary)
% mv lispdate.o lispdate

 Now we test it out
% lispdate
The date is Sat Aug  1 16:58:33 1981
%
```

12.7. pure literals

Normally the quoted lisp objects (literals) which appear in functions are treated as constants. Consider this function:

```
(def foo
  (lambda nil (cond ((not (eq 'a (car (setq x '(a b)))))
                     (print 'impossible!!))
                    (t (rplaca x 'd)))))
```

At first glance it seems that the first cond clause will never be true, since the *car* of *(a b)* should always be *a*. However if you run this function twice, it will print 'impossible!!' the second time. This is because the following clause modifies the 'constant' list *(a b)* with the *rplaca* function. Such modification of literal lisp objects can cause programs to behave strangely as the above example shows, but more importantly it can cause garbage collection problems if done to compiled code. When a file is *fasl*ed in, if the symbol $purcopylits is non nil, the literal lisp data is put in 'pure' space, that is it put in space which needn't be looked at by the garabage collector. This reduces the work the garbage collector must do but it is dangerous since if the literals are modified to point to non pure objects, the marker may not mark the non pure objects. If the symbol $purcopylits is nil then the literal lisp data is put in impure space and the compiled code will act like the interpreted code when literal data is modified. The default value for $purcopylits is t.

12.8. transfer tables

A transfer table is setup by *fasl* when the object file is loaded in. There is one entry in the transfer table for each function which is called in that object file. The entry for a call to the function *foo* has two parts whose contents are:

[1] function address — This will initially point to the internal function *qlinker*. It may some time in the future point to the function *foo* if certain conditions are satisfied (more on this below).

[2] function name — This is a pointer to the symbol *foo*. This will be used by *qlinker*.

When a call is made to the function *foo* the call will actually be made to the address in the transfer table entry and will end up in the *qlinker* function. *Qlinker* will determine that *foo* was the function being called by locating the function name entry in the transfer table[†]. If the function being called is not compiled then *qlinker* just calls *funcall* to perform the function call. If *foo* is compiled and if *(status translink)* is non nil, then *qlinker* will modify the function address part of the transfer table to point directly to the function *foo*. Finally *qlinker* will call *foo* directly . The next time a call is made to *foo* the call will go directly to *foo* and not through *qlinker*. This will result in a substantial speedup in compiled code to compiled code transfers. A disadvantage is that no debugging information is left on the stack, so *showstack* and *baktrace* are useless. Another disadvantage is that if you redefine a compiled function either through loading in a new version or interactively defining it, then the old version may still be called from compiled code if the fast linking described above has already been done. The solution to these problems is to use *(sstatus translink value)*. If value is

nil All transfer tables will be cleared, i.e. all function addresses will be set to point to *qlinker*. This means that the next time a function is called *qlinker* will be called and

[†]*Qlinker* does this by tracing back the call stack until it finds the *calls* machine instruction which called it. The address field of the *calls* contains the address of the transfer table entry.

will look at the current definition. Also, no fast links will be set up since *(status translink)* will be nil. The end result is that *showstack* and *baktrace* will work and the function definition at the time of call will always be used.

on This causes the lisp system to go through all transfer tables and set up fast links wherever possible. This is normally used after you have *fasl*ed in all of your files. Furthermore since *(status translink)* is not nil, *qlinker* will make new fast links if the situation arises (which isn't likely unless you *fasl* in another file).

t This or any other value not previously mentioned will just make *(status translink)* be non nil, and as a result fast links will be made by *qlinker* if the called function is compiled.

12.9. Fixnum functions

The compiler will generate inline arithmetic code for fixnum only functions. Such functions include +, −, *, /, \, 1+ and 1−. The code generated will be much faster than using *add*, *difference*, etc. However it will only work if the arguments to and results of the functions are fixnums. No type checking is done.

CHAPTER 13

The CMU User Toplevel and the File Package

This documentation was written by Don Cohen, and the functions described below were imported from PDP-10 CMULisp.

Non CMU users note: this is not the default top level for your Lisp system. In order to start up this top level, you should type *(load 'cmuenv)*.

13.1. User Command Input Top Level

The top-level is the function that reads what you type, evaluates it and prints the result. The *newlisp* top-level was inspired by the CMULisp top-level (which was inspired by interlisp) but is much simpler. The top-level is a function (of zero arguments) that can be called by your program. If you prefer another top-level, just redefine the top-level function and type "(reset)" to start running it. The current top-level simply calls the functions tlread, tleval and tlprint to read, evaluate and print. These are supposed to be replaceable by the user. The only one that would make sense to replace is tlprint, which currently uses a function that refuses to go below a certain level and prints "...]" when it finds itself printing a circular list. One might want to prettyprint the results instead. The current top-level numbers the lines that you type to it, and remembers the last n "events" (where n can be set but is defaulted to 25). One can refer to these events in the following "top-level commands":

TOPLEVEL COMMAND SUMMARY

??	prints events - both the input and the result. If you just type "??" you will see all of the recorded events. "?? 3" will show only event 3, and "?? 3 6" will show events 3 through 6.
redo	pretends that you typed the same thing that was typed before. If you type "redo 3" event number 3 is redone. "redo -3" redoes the thing 3 events ago. "redo" is the same as "redo -1".
ed	calls the editor and then does whatever the editor returns. Thus if you want to do event 5 again except for some small change, you can type "ed 5", make the change and leave the editor. "ed -3" and "ed" are analogous to redo.

Finally, you can get the value of event 7 with the function (valueof 7). The other interesting feature of the top-level is that it makes outermost parentheses superfluous for the most part. This works the same way as in CMULisp, so you can use the help for an explanation. If you're not sure and don't want to risk it you can always just include the parentheses.

(top-level)

SIDE EFFECT: *top-level* is the LISP top level function. As well as being the top level function with which the user interacts, it can be called recursively by the user or any function. Thus, the top level can be invoked from inside the editor, break package, or a user function to make its commands available to the user.

NOTE: The CMU FRANZ LISP top-level uses *lineread* rather than *read.* The difference will not usually be noticeable. The principal thing to be careful about is that input to the function or system being called cannot appear on the same line as the top-level call. For example, typing *(editf foo)fP on one line will edit foo and evaluate P, not edit foo and execute the p command in the editor. top-level specially recognizes the following commands:*

(valueof *'g_eventspec***)**

RETURNS: the value(s) of the event(s) specified by g_eventspec. If a single event is specified, its value will be returned. If more than one event is specified, or an event has more than one subevent (as for *redo,* etc), a list of vlaues will be returned.

13.2. The File Package

Users typically define functions in lisp and then want to save them for the next session. If you do *(changes)*, a list of the functions that are newly defined or changed will be printed. When you type *(dskouts)*, the functions associated with files will be saved in the new versions of those files. In order to associate functions with files you can either add them to the *filefns* list of an existing file or create a new file to hold them. This is done with the *file* function. If you type *(file new)* the system will create a variable called *newfns.* You may add the names of the functions to go into that file to *newfns.* After you do *(changes)*, the functions which are in no other file are stored in the value of the atom *changes.* To put these all in the new file, *(setq newfns (append newfns changes)).* Now if you do *(changes)*, all of the changed functions should be associated with files. In order to save the changes on the files, do *(dskouts).* All of the changed files (such as NEW) will be written. To recover the new functions the next time you run FRANZ LISP, do *(dskin new).*

```
Script started on Sat Mar 14 11:50:32 1981
$ newlisp
Welcome to newlisp...
1.(defun square (x) (* x x))            ; define a new function
square
2.(changes)                                     ; See, this function is associated
                                                ; with no file.
<no-file>       (square)nil
3.(file 'new)                                   ; So let's declare file NEW.
new
4.newfns                                ; It doesn't have anything on it yet.
nil
5.(setq newfns '(square))               ; Add the function associated
(square)                                ; with no file to file NEW.
6.(changes)                                     ; CHANGES magically notices this fact.

new             (square)nil
7.(dskouts)                                     ; We write the file.
creating new
(new)
8.(dskin new)                                   ; We read it in!
(new)
14.Bye
$
script done on Sat Mar 14 11:51:48 1981
```

(changes s_flag**)**

RETURNS: Changes computes a list containing an entry for each file which defines atoms that have been marked changed. The entry contains the file name and the changed atoms defined therein. There is also a special entry for changes to atoms which are not defined in any known file. The global variable *filelst* contains the list of "known" files. If no flag is passed this result is printed in human readable form and the value returned is t if there were any changes and nil if not. Otherwise nothing is printed and the computer list is returned. The global variable *changes* contains the atoms which are marked changed but not yet associated with any file. The *changes* function attempts to associate these names with files, and any that are not found are considered to belong to no file. The *changes* property is the means by which changed functions are associated with files. When a file is read in or written out its *changes* property is removed.

(dc s_word s_id [g_descriptor1 ...] <text> <esc>**)**

RETURNS: *dc* defines comments. It is exceptional in that its behavior is very context dependent. When *dc* is executed from *dskin* it simply records the fact that the comment exists. It is expected that in interactive mode comments will be found via *getdef* - this allows large comments which do not take up space in your core image. When *dc* is executed from the terminal it expects you to type a comment. *dskout* will write out the comments that you define and also copy the comments on the old version of the file, so that the new version will keep the old comments even though they were never actually brought into core. The optional id is a mechanism for distinguishing among several comments associated with the same word. It defaults to nil. However if you define two comments with the same id, the second is considered to be a replacement for the first. The behavior of *dc* is determined by the value of the global variable *def-comment. def-comment* contains the name of a function that is run. Its arguments are the word, id and attribute list. *def-comment* is initially *dc-define.* Other functions rebind it to *dc-help, dc-userhelp,* and the value of *dskin-comment.* The comment property of an atom is a list of entries, each representing one comment. Atomic entries are assumed to be identifiers of comments on a file but not in core. In-core comments are represented by a list of the id, the attribute list and the comment text. The comment text is an uninterned atom. Comments may be deleted or reordered by editing the comment property.

(dskin l_filenames**)**

SIDE EFFECT: READ-EVAL-PRINTs the contents of the given files. This is the function to use to read files created by *dskout. dskin* also declares the files that it reads (if a *file-fns* list is defined and the file is otherwise declarable by *file*), so that changes to it can be recorded.

(dskout s_file1 ...**)**

SIDE EFFECT: For each file specified, *dskout* assumes the list named filenameFNS (i.e., the file name, excluding extension, concatenated with *fns*) contains a list of function names, etc., to be loaded Any previous version of the file will be renamed to have extension ".back".

(dskouts s_file1 ...**)**

SIDE EFFECT: applies *dskout* to and prints the name of each s_filei (with no additional arguments, assuming filenameFNS to be a list to be loaded) for which s_file*i* is either not in *filelst* (meaning it is a new file not previously declared by *file* or given as an argument to *dskin, dskouts,* or *dskouts)* or is in *filelst* and has some recorded changes to definitions of atoms in filenameFNS, as recorded by *mark!changed* and noted by changes. If *filei* is not specified, *filelst* will be used. This is the most common way of using dskouts. Typing *(dskouts)* will save every file reported by *(changes)* to have changed definitions.

(dv s_atom g_value**)**

EQUIVALENT TO: *(setq atom 'value). dv* calls *mark!changed.*

(file 's_file**)**

SIDE EFFECT: declares its argument to be a file to be used for reporting and saving changes to functions by adding the file name to a list of files, *filelst. file* is called for each file argument of *dskin, dskout,* and *dskouts.*

(file-fns 's_file**)**

RETURNS: the name of the fileFNS list for its file argument s_file.

(getdef 's_file ['s_i1 ...]**)**

SIDE EFFECT: selectively executes definitions for atoms s_i1 ... from the specified file. Any of the words to be defined which end with "@" will be treated as patterns in which the @ matchs any suffix (just like the editor). *getdef* is driven by *getdeftable* (and thus may be programmed). It looks for lines in the file that start with a word in the table. The first character must be a "(" or "[" followed by the word, followed by a space, return or something else that will not be considered as part of the identifier by *read,* e.g., "(" is unacceptable. When one is found the next word is read. If it matches one of the identifiers in the call to *getdef* then the table entry is executed. The table entry is a function of the expression starting in this line. Output from *dskout* is in acceptable format for *getdef. getdef*

RETURNS: a list of the words which match the ones it looked for, for which it found (but, depending on the table, perhaps did not execute) in the file.

NOTE: *getdeftable* is the table that drives *getdef.* It is in the form of an association list. Each element is a dotted pair consisting of the name of a function for which *getdef* searches and a function of one argument to be executed when it is found.

(mark!changed 's_f**)**

SIDE EFFECT: records the fact that the definition of s_f has been changed. It is automatically called by *def, defun, de, df, defprop, dm, dv,* and the editor when a definition is altered.

CHAPTER 14

The LISP Stepper

14.1. Simple Use Of Stepping

(step s_arg1...**)**

NOTE: The LISP "stepping" package is intended to give the LISP programmer a facility analogous to the Instruction Step mode of running a machine language program. The user interface is through the function (fexpr) step, which sets switches to put the LISP interpreter in and out of "stepping" mode. The most common *step* invocations follow. These invocations are usually typed at the top-level, and will take effect immediately (i.e. the next S-expression typed in will be evaluated in stepping mode).

```
(step t)                    ; Turn on stepping mode.
(step nil)                  ; Turn off stepping mode.
```

SIDE EFFECT: In stepping mode, the LISP evaluator will print out each S-exp to be evaluated before evaluation, and the returned value after evaluation, calling itself recursively to display the stepped evaluation of each argument, if the S-exp is a function call. In stepping mode, the evaluator will wait after displaying each S-exp before evaluation for a command character from the console.

STEP COMMAND SUMMARY

<return>	Continue stepping recursively.
c	Show returned value from this level only, and continue stepping upward.
e	Only step interpreted code.
g	Turn off stepping mode. (but continue evaluation without stepping).
n <number>	Step through <number> evaluations without stopping
p	Redisplay current form in full (i.e. rebind prinlevel and prinlength to nil)
b	Get breakpoint
q	Quit
d	Call debug

14.2. Advanced Features

14.2.1. Selectively Turning On Stepping.

If

(step foo1 foo2 ...)

is typed at top level, stepping will not commence immediately, but rather when the evaluator first encounters an S-expression whose car is one of *foo1, foo2*, etc. This form will then display at the console, and the evaluator will be in stepping mode waiting for a command character.

Normally the stepper intercepts calls to *funcall* and *eval*. When *funcall* is intercepted, the arguments to the function have already been evaluated but when *eval* is intercepted, the arguments have not been evaluated. To differentiate the two cases, when printing the form in evaluation, the stepper preceded intercepted calls to *funcall* with "f:". Calls to *funcall* are normally caused by compiled lisp code calling other functions, whereas calls to *eval* usually occur when lisp code is interpreted. To step only calls to eval use: *(step e)*

14.2.2. Stepping With Breakpoints.

For the moment, step is turned off inside of error breaks, but not by the break function. Upon exiting the error, step is reenabled. However, executing *(step nil)* inside a error loop will turn off stepping globally, i.e. within the error loop, and after return has be made from the loop.

14.3. Overhead of Stepping.

If stepping mode has been turned off by *(step nil)*, the execution overhead of having the stepping packing in your LISP is identically nil. If one stops stepping by typing "g", every call to eval incurs a small overhead--several machine instructions, corresponding to the compiled code for a simple cond and one function pushdown. Running with *(step foo1 foo2 ...)* can be more expensive, since a member of the car of the current form into the list *(foo1 foo2 ...)* is required at each call to eval.

14.4. Evalhook and Funcallhook

There are hooks in the FRANZ LISP interpreter to permit a user written function to gain control of the evaluation process. These hooks are used by the Step package just described. There are two hooks and they have been strategically placed in the two key functions in the interpreter: *eval* (which all interpreted code goes through) and *funcall* (which all compiled code goes through if *(sstatus translink nil)* has been done). The hook in *eval* is compatible with Maclisp, but there is no Maclisp equivalent of the hook in *funcall.*

To arm the hooks two forms must be evaluated: *(*rset t)* and *(sstatus evalhook t).* Once that is done, *eval* and *funcall* do a special check when they enter.

If *eval* is given a form to evaluate, say *(foo bar)*, and the symbol 'evalhook' is non nil, say its value is 'ehook', then *eval* will lambda bind the symbols 'evalhook' and 'funcallhook' to nil and will call ehook passing *(foo bar)* as the argument. It is ehook's responsibility to evaluate *(foo bar)* and return its value. Typically ehook will call the function 'evalhook' to evaluate *(foo bar).* Note that 'evalhook' is a symbol whose function binding is a system function described in Chapter 4, and whose value binding, if non nil, is the name of a user written function (or a lambda expression, or a binary object) which will gain control whenever eval is called. 'evalhook' is also the name of the *status* tag which must be set for all of this to work.

If *funcall* is given a function, say foo, and a set of already evaluated arguments, say barv and bazv, and if the symbol 'funcallhook' has a non nil value, say 'fhook', then *funcall* will lambda bind 'evalhook' and 'funcallhook' to nil and will call fhook with arguments barv, bazv and foo. Thus fhook must be a lexpr since it may be given any number of arguments. The function to call, foo in this case, will be the *last* of the arguments given to fhook. It is fhooks responsibility to do the function call and return the value. Typically fhook will call the function *funcallhook* to do the funcall. This is an example of a funcallhook function which just prints the arguments on each entry to funcall and the return value.

```
-> (defun fhook n (let ((form (cons (arg n) (listify (1- n))))
                                (retval))
                           (patom "calling ")(print form)(terpr)
                           (setq retval (funcallhook form 'fhook))
                           (patom "returns ")(print retval)(terpr)
                           retval))
fhook
-> (*rset t) (sstatus evalhook t) (sstatus translink nil)
-> (setq funcallhook 'fhook)
calling (print fhook)                    ;; now all compiled code is traced
fhookreturns nil
calling (terpr)

returns nil
calling (patom "-> ")
-> returns "-> "
calling (read nil Q00000)
(array foo t 10)                 ;; to test it, we see what happens when
returns (array foo t 10)         ;; we make an array
calling (eval (array foo t 10))
calling (append (10) nil)
returns (10)
calling (lessp 1 1)
returns nil
calling (apply times (10))
returns 10
calling (small-segment value 10)
calling (boole 4 137 127)
returns 128
 ... there is plenty more ...
```

CHAPTER 15

The FIXIT Debugger

15.1. Introduction FIXIT is a debugging environment for FRANZ LISP users doing program development. This documentation and FIXIT were written by David S. Touretzky of Carnegie-Mellon University for MACLisp, and adapted to FRANZ LISP by Mitch Marcus of Bell Labs. One of FIXIT's goals is to get the program running again as quickly as possible. The user is assisted in making changes to his functions "on the fly", i.e. in the midst of execution, and then computation is resumed.

To enter the debugger type *(debug)*. The debugger goes into its own read-eval-print loop. Like the top-level, the debugger understands certain special commands. One of these is help, which prints a list of the available commands. The basic idea is that you are somewhere in a stack of calls to eval. The command "bka" is probably the most appropriate for looking at the stack. There are commands to move up and down. If you want to know the value of "x" as of some place in the stack, move to that place and type "x" (or (cdr x) or anything else that you might want to evaluate). All evaluation is done as of the current stack position. You can fix the problem by changing the values of variables, editing functions or expressions in the stack etc. Then you can continue from the current stack position (or anywhere else) with the "redo" command. Or you can simply return the right answer with the "return" command.

When it is not immediately obvious why an error has occurred or how the program got itself into its current state, FIXIT comes to the rescue by providing a powerful debugging loop in which the user can:

- examine the stack
- evaluate expressions in context
- enter stepping mode
- restart the computation at any point

The result is that program errors can be located and fixed extremely rapidly, and with a minimum of frustration.

The debugger can only work effectively when extra information is kept about forms in evaluation by the lisp system. Evaluating *(*rset t)* tells the lisp system to maintain this information. If you are debugging compiled code you should also be sure that the compiled code to compiled code linkage tables are unlinked, i.e do *(sstatus translink nil)*.

(debug [s_msg])

NOTE: Within a program, you may enter a debug loop directly by putting in a call to *debug* where you would normally put a call to *break*. Also, within a break loop you may enter FIXIT by typing *debug*. If an argument is given to DEBUG, it is treated as a message to be printed before the debug loop is entered. Thus you can put *(debug |just before loop|)* into a program to indicate what part of the program is being debugged.

FIXIT Command Summary

TOP go to top of stack (latest expression)
BOT go to bottom of stack (first expression)
P show current expression (with ellipsis)
PP show current expression in full
WHERE give current stack position
HELP types the abbreviated command summary found
in /usr/lisp/doc/fixit.help. H and ? work too.
U go up one stack frame
U n go up n stack frames
U f go up to the next occurrence of function f
U n f go up n occurrences of function f
UP go up to the next user-written function
UP n go up n user-written functions
...the DN and DNFN commands are similar, but go down
...instead of up.
OK resume processing; continue after an error or debug loop
REDO restart the computation with the current stack frame.
The OK command is equivalent to TOP followed by REDO.
REDO f restart the computation with the last call to function f.
(The stack is searched downward from the current position.)
STEP restart the computation at the current stack frame,
but first turn on stepping mode. (Assumes Rich stepper is loaded.)
RETURN e return from the current position in the computation
with the value of expression e.
BK.. print a backtrace. There are many backtrace commands,
formed by adding suffixes to the BK command. "BK" gives
a backtrace showing only user-written functions, and uses
ellipsis. The BK command may be suffixed by one or more
of the following modifiers:
..F.. show function names instead of expressions
..A.. show all functions/expressions, not just user-written ones
..V.. show variable bindings as well as functions/expressions
..E.. show everything in the expression, i.e. don't use ellipsis
..C.. go no further than the current position on the stack
Some of the more useful combinations are BKFV, BKFA,
and BKFAV.
BK.. n show only n levels of the stack (starting at the top).
(BK n counts only user functions; BKA n counts all functions.)
BK.. f show stack down to first call of function f
BK.. n f show stack down to nth call of function f

15.2. Interaction with *trace* FIXIT knows about the standard Franz trace package, and tries to make tracing invisible while in the debug loop. However, because of the way *trace* works, it may sometimes be the case that the functions on the stack are really un*interned* atoms that have the same name as a traced function. (This only happens when a function is traced WHEREIN another one.) FIXIT will call attention to *trace's* hackery by printing an appropriate tag next to these stack entries.

15.3. Interaction with *step* The *step* function may be invoked from within FIXIT via the STEP command. FIXIT initially turns off stepping when the debug loop is entered. If you step through a function and get an error, FIXIT will still be invoked normally. At any time during stepping, you may explicitly enter FIXIT via the "D" (debug) command.

15.4. Multiple error levels FIXIT will evaluate arbitrary LISP expressions in its debug loop. The evaluation is not done within an *errset,* so, if an error occurs, another invocation of the debugger can be made. When there are multiple errors on the stack, FIXIT displays a barrier symbol between each level that looks something like <----------UDF-->. The UDF in this case stands for UnDefined Function. Thus, the upper level debug loop was invoked by an undefined function error that occurred while in the lower loop.

CHAPTER 16

The LISP Editor

16.1. The Editors

It is quite possible to use VI, Emacs or other standard editors to edit your lisp programs, and many people do just that. However there is a lisp structure editor which is particularly good for the editing of lisp programs, and operates in a rather different fashion, namely within a lisp environment. application. It is handy to know how to use it for fixing problems without exiting from the lisp system (e.g. from the debugger so you can continue to execute rather than having to start over.) The editor is not quite like the top-level and debugger, in that it expects you to type editor commands to it. It will not evaluate whatever you happen to type. (There is an editor command to evaluate things, though.)

The editor is available (assuming your system is set up correctly with a lisp library) by typing (load 'cmufncs) and (load 'cmuedit).

The most frequent use of the editor is to change function definitions by starting the editor with one of the commands described in section 16.14. (see *editf*), values (*editv*), properties (*editp*), and expressions (*edite*). The beginner is advised to start with the following (very basic) commands: *ok, undo, p, #*, under which are explained two different basic commands which start with numbers, and f.

This documentation, and the editor, were imported from PDP-10 CMULisp by Don Cohen. PDP-10 CMULisp is based on UCILisp, and the editor itself was derived from an early version of Interlisp. Lars Ericson, the author of this section, has provided this very concise summary. Tutorial examples and implementation details may be found in the Interlisp Reference Manual, where a similar editor is described.

16.2. Scope of Attention

Attention-changing commands allow you to look at a different part of a Lisp expression you are editing. The sub-structure upon which the editor's attention is centered is called "the current expression". Changing the current expression means shifting attention and not actually modifying any structure.

SCOPE OF ATTENTION COMMAND SUMMARY

n (n> 0) . Makes the nth element of the current expression be the new current expression.

-n (n> 0). Makes the nth element from the end of the current expression be the new current expression.

0. Makes the next higher expression be the new correct expression. If the intention is to go back to the next higher left parenthesis, use the command !0.

up . If a p command would cause the editor to type ... before typing the current expression, (the current expression is a tail of the next higher expression) then has no effect; else, up makes the old current expression the first element in the new current expression.

!0 . Goes back to the next higher left parenthesis.

^ . Makes the top level expression be the current expression.

nx . Makes the current expression be the next expression.

(nx n) equivalent to n nx commands.

!nx . Makes current expression be the next expression at a higher level. Goes through any number of right parentheses to get to the next expression.

bk . Makes the current expression be the previous expression in the next higher expression.

(nth n) n> 0 . Makes the list starting with the nth element of the current expression be the current expression.

(nth $) - generalized nth command. nth locates $, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation.

:: . (pattern :: . $) e.g., (cond :: return). finds a cond that contains a return, at any depth.

(below com x) . The below command is useful for locating a substructure by specifying something it contains. (below cond) will cause the cond clause containing the current expression to become the new current expression. Suppose you are editing a list of lists, and want to find a sublist that contains a foo (at any depth). Then simply executes f foo (below).

(nex x) . same as *(below x)* followed by nx. For example, if you are deep inside of a selectq clause, you can advance to the next clause with *(nex selectq).*

nex. The atomic form of *nex* is useful if you will be performing repeated executions of *(nex x).* By simply marking the chain corresponding to x, you can use *nex* to step through the sublists.

16.3. Pattern Matching Commands

Many editor commands that search take patterns. A pattern *pat* matches with x if:

PATTERN SPECIFICATION SUMMARY

- *pat* is *eq* to x.

- *pat* is &.

- *pat* is a number and equal to x.

- if (car *pat*) is the atom *any*, (cdr *pat*) is a list of patterns, and *pat* matches x if and only if one of the patterns on (cdr *pat*) matches x.

- if *pat* is a literal atom or string, and (nthchar *pat* -1) is @, then *pat* matches with any literal atom or string which has the same initial characters as *pat*, e.g. ver@ matches with verylongatom, as well as "verylongstring".

- if (car *pat*) is the atom --, *pat* matches x if (a) (cdr *pat*) =nil, i.e. *pat*=(--), e.g., (a --) matches (a) (a b c) and (a . b) in other words, -- can match any tail of a list. (b) (cdr *pat*) matches with some tail of x, e.g. (a -- (&)) will match with (a b c (d)), but not (a b c d), or (a b c (d) e). however, note that (a -- (&) --) will match with (a b c (d) e). in other words, -- will match any interior segment of a list.

- if (car *pat*) is the atom ==, *pat* matches x if and only if (cdr *pat*) is *eq* to x. (this pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command typed in by the user obviously cannot be *eq* to existing structure.) - otherwise if x is a list, *pat* matches x if (car *pat*) matches (car x), and (cdr *pat*) matches (cdr x).

- when searching, the pattern matching routine is called only to match with elements in the structure, unless the pattern begins with :::, in which case cdr of the pattern is matched against tails in the structure. (in this case, the tail does not have to be a proper tail, e.g. (::: a --) will match with the element (a b c) as well as with cdr of (x a b c), since (a b c) is a tail of (a b c).)

16.3.1. Commands That Search

SEARCH COMMAND SUMMARY

f pattern . f informs the editor that the next command is to be interpreted as a pattern. If no pattern is given on the same line as the f then the last pattern is used. f pattern means find the next instance of pattern.

(f pattern n). Finds the next instance of pattern.

(f pattern t). similar to f pattern, except, for example, if the current expression is (cond ..), f cond will look for the next cond, but (f cond t) will 'stay here'.

(f pattern n) n>0. Finds the nth place that pattern matches. If the current expression is (foo1 foo2 foo3), (f f00@ 3) will find foo3.

(f pattern) or (f pattern nil). only matches with elements at the top level of the current expression. If the current expression is *(prog nil (setq x (cond & &)) (cond &) ...)* f (cond --) will find the cond inside the setq, whereas (f (cond --)) will find the top level cond, i.e., the second one.

(second . $) . same as (lc . $) followed by another (lc . $) except that if the first succeeds and second fails, no change is made to the edit chain.

(third . $) . Similar to second.

(fs pattern1 ... patternn) . equivalent to f pattern1 followed by f pattern2 ... followed by f pattern n, so that if f pattern m fails, edit chain is left at place pattern m-1 matched.

(f= expression x) . Searches for a structure eq to expression.

(orf pattern1 ... patternn) . Searches for an expression that is matched by either pattern1 or ... patternn.

bf pattern . backwards find. If the current expression is *(prog nil (setq x (setq y (list z))) (cond ((setq w --) --)) --)* f list followed by bf setq will leave the current expression as (setq y (list z)), as will f cond followed by bf setq

(bf pattern t). backwards find. Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

16.3.1.1. Location Specifications Many editor commands use a method of specifying position called a location specification. The meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic, in which case it is interpreted as (list $). a location specification is a list of edit commands that are executed in the normal fashion with two exceptions. first, all commands not recognized by the editor are interpreted as though they had been preceded by f. The location specification (cond 2 3) specifies the 3rd element in the first clause of the next cond.

the if command and the ## function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression.

In insert, delete, replace and change, if $ is nil (empty), the corresponding operation is performed on the current edit chain, i.e. (replace with (car x)) is equivalent to (:(car x)). for added readability, here is also permitted, e.g., (insert (print x) before here) will insert (print x) before the current expression (but not change the edit chain). It is perfectly legal to ascend to insert, replace, or delete. for example (insert (*return*) after ^ prog -1) will go to the top, find the first prog, and insert a (*return*) at its end, and not change the current edit chain.

The a, b, and : commands all make special checks in e1 thru em for expressions of the form (## . coms). In this case, the expression used for inserting or replacing is a copy of the current expression after executing coms, a list of edit commands. (insert (## f cond -1 -1) after3) will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

$. In descriptions of the editor, the meta-symbol $ is used to denote a location specification. $ is a list of commands interpreted as described above. $ can also be atomic.

LOCATION COMMAND SUMMARY

(lc . $) . Provides a way of explicitly invoking the location operation. (lc cond 2 3) will perform search.

(lcl . $) . Same as lc except search is confined to current expression. To find a cond containing a *return*, one might use the location specification (cond (lcl *return*)) where the would reverse the effects of the lcl command, and make the final current expression be the cond.

16.3.2. The Edit Chain The edit-chain is a list of which the first element is the the one you are now editing ("current expression"), the next element is what would become the current expression if you were to do a 0, etc., until the last element which is the expression that was passed to the editor.

EDIT CHAIN COMMAND SUMMARY

mark . Adds the current edit chain to the front of the list marklst.

_ . Makes the new edit chain be (car marklst).

(_ pattern) . Ascends the edit chain looking for a link which matches pattern. for example:

__ . Similar to _ but also erases the mark.

\ . Makes the edit chain be the value of unfind. unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ^, _, __, !nx, all commands that involve a search, e.g., f, lc, ::, below, et al and and themselves. if the user types f cond, and then f car, would take him back to the cond. another would take him back to the car, etc.

\p . Restores the edit chain to its state as of the last print operation. If the edit chain has not changed since the last printing, \p restores it to its state as of the printing before that one. If the user types p followed by 3 2 1 p, \p will return to the first p, i.e., would be equivalent to 0 0 0. Another \p would then take him back to the second p.

16.4. Printing Commands

PRINTING COMMAND SUMMARY

p Prints current expression in abbreviated form. (p m) prints mth element of current expression in abbreviated form. (p m n) prints mth element of current expression as though printlev were given a depth of n. (p 0 n) prints current expression as though printlev were given a depth of n. (p cond 3) will work.

? . prints the current expression as though printlev were given a depth of 100.

pp . pretty-prints the current expression.

*pp** . is like pp, but forces comments to be shown.

16.5. Structure Modification Commands

All structure modification commands are undoable. See *undo*.

STRUCTURE MODIFICATION COMMAND SUMMARY

 # [editor commands] (n) n>1 deletes the corresponding element from the current expression.

(n e1 ... em) n,m> 1 replaces the nth element in the current expression with e1 ... em.

(-n e1 ... em) n,m> 1 inserts e1 ... em before the n element in the current expression.

(n e1 ... em) (the letter "n" for "next" or "nconc", not a number) m>1 attaches e1 ... em at the end of the current expression.

(a e1 ... em) . inserts e1 ... em after the current expression (or after its first element if it is a tail).

(b e1 ... em) . inserts e1 ... em before the current expression. to insert foo before the last element in the current expression, perform -1 and then (b foo).

(: e1 ... em) . replaces the current expression by e1 ... em. If the current expression is a tail then replace its first element.

delete or (:) . deletes the current expression, or if the current expression is a tail, deletes its first element.

(delete . $). does a (lc . $) followed by delete. current edit chain is not changed.

(insert e1 ... em before . $) . similar to (lc. $) followed by (b e1 ... em).

(insert e1 ... em after . $). similar to insert before except uses a instead of b.

(insert e1 ... em for . $). similar to insert before except uses : for b.

(replace $ with e1 ... em) . here $ is the segment of the command between replace and with.

(change $ to e1 ... em) . same as replace with.

16.6. Extraction and Embedding Commands

EXTRACTION AND EMBEDDING COMMAND SUMMARY

(xtr . $) . replaces the original current expression with the expression that is current after performing (lcl . $).

(mbd x) . x is a list, substitutes the current expression for all instances of the atom * in x, and replaces the current expression with the result of that substitution. (mbd x) : x atomic, same as (mbd (x *)).

(extract $1 from $2) . extract is an editor command which replaces the current expression with one of its subexpressions (from any depth). ($1 is the segment between extract and from.) example: if the current expression is (print (cond ((null x) y) (t z))) then following (extract y from cond), the current expression will be (print y). (extract 2 -1 from cond), (extract y from 2), (extract 2 -1 from 2) will all produce the same result.

(embed $ in . x) . embed replaces the current expression with a new expression which contains it as a subexpression. ($ is the segment between embed and in.) example: (embed print in setq x), (embed 3 2 in *return*), (embed cond 3 1 in (or * (null x))).

16.7. Move and Copy Commands

MOVE AND COPY COMMAND SUMMARY

(move $1 to com . $2) . ($1 is the segment between move and to.) where com is before, after, or the name of a list command, e.g., :, n, etc. If $2 is nil, or (here), the current position specifies where the operation is to take place. If $1 is nil, the move command allows the user to specify some place the current expression is to be moved to. if the current expression is (a b d c), (move 2 to after 4) will make the new current expression be (a c d b).

(mv com . $) . is the same as (move here to com . $).

(copy $1 to com . $2) is like move except that the source expression is not deleted.

(cp com . $). is like mv except that the source expression is not deleted.

16.8. Parentheses Moving Commands The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses.

PARENTHESES MOVING COMMAND SUMMARY

(bi n m) . both in. inserts parentheses before the nth element and after the mth element in the current expression. example: if the current expression is (a b (c d e) f g), then (bi 2 4) will modify it to be (a (b (c d e) f) g). (bi n) : same as (bi n n). example: if the current expression is (a b (c d e) f g), then (bi -2) will modify it to be (a b (c d e) (f) g).

(bo n) . both out. removes both parentheses from the nth element. example: if the current expression is (a b (c d e) f g), then (bo d) will modify it to be (a b c d e f g).

(li n) . left in. inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression). example: if the current expression is (a b (c d e) f g), then (li 2) will modify it to be (a (b (c d e) f g)).

(lo n) . left out. removes a left parenthesis from the nth element. all elements following the nth element are deleted. example: if the current expression is (a b (c d e) f g), then (lo 3) will modify it to be (a b c d e).

(ri n m) . right in. move the right parenthesis at the end of the nth element in to after the mth element. inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression. example: if the current expression is (a (b c d e) f g), (ri 2 2) will modify it to be (a (b c) d e f g).

(ro n) . right out. move the right parenthesis at the end of the nth element out to the end of the current expression. removes the right parenthesis from the nth element, moving it to the end of the current expression. all elements following the nth element are moved inside of the nth element. example: if the current expression is (a b (c d e) f g), (ro 3) will modify it to be (a b (c d e f g)).

(r x y) replaces all instances of x by y in the current expression, e.g., (r caadr cadar). x can be the s-expression (or atom) to be substituted for, or can be a pattern which specifies that s-expression (or atom).

(sw n m) switches the nth and mth elements of the current expression. for example, if the current expression is (list (cons (car x) (car y)) (cons (cdr y))), (sw 2 3) will modify it to be (list (cons (cdr x) (cdr y)) (cons (car x) (car y))). (sw car cdr) would produce the same result.

16.8.1. Using to and thru

to, thru, extract, embed, delete, replace, and move can be made to operate on several contiguous elements, i.e., a segment of a list, by using the to or thru command in their respective location specifications. thru and to are intended to be used in conjunction with extract, embed, delete, replace, and move. to and thru can also be used directly with xtr (which takes after a location specification), as in (xtr (2 thru 4)) (from the current expression).

TO AND THRU COMMAND SUMMARY

($1 to $2) . same as thru except last element not included.

($1 to). same as ($1 thru -1)

($1 thru $2) . If the current expression is (a (b (c d) (e) (f g h) i) j k), following (c thru g), the current expression will be ((c d) (e) (f g h)). If both $1 and $2 are numbers, and $2 is greater than $1, then $2 counts from the beginning of the current expression, the same as $1. in other words, if the current expression is (a b c d e f g), (3 thru 4) means (c thru d), not (c thru f). in this case, the corresponding bi command is (bi 1 $2-$1+1).

($1 thru). same as *($1 thru -1).*

16.9. Undoing Commands each command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command. The undo command undoes the last, i.e., most recent such command.

UNDO COMMAND SUMMARY

undo . the undo command undoes most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., mbd undone. The edit chain is then exactly what it was before the 'undone' command had been performed.

!undo . undoes all modifications performed during this editing session, i.e., this call to the editor.

unblock . removes an undo-block. If executed at a non-blocked state, i.e., if undo or !undo could operate, types not blocked.

test . adds an undo-block at the front of undolst. note that test together with !undo provide a 'tentative' mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !undo command.

undolst [value]. each editor command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command.

?? prints the entries on undolst. The entries are listed most recent entry first.

16.10. Commands that Evaluate

EVALUATION COMMAND SUMMARY

e . only when typed in, (i.e., (insert d before e) will treat e as a pattern) causes the editor to call the lisp interpreter giving it the next input as argument.

(e x) evaluates x, and prints the result. (e x t) same as (e x) but does not print.

(i c x1 ... xn) same as (c y1 ... yn) where yi=(eval xi). example: (i 3 (cdr foo)) will replace the 3rd element of the current expression with the cdr of the value of foo. (i n foo (car fie)) will attach the value of foo and car of the value of fie to the end of the current expression. (i f= foo t) will search for an expression eq to the value of foo. If c is not an atom, it is evaluated as well.

(coms x1 ... xn) . each xi is evaluated and its value executed as a command. The i command is not very convenient for computing an entire edit command for execution, since it computes the command name and its arguments separately. also, the i command cannot be used to compute an atomic command. The coms and comsq commands provide more general ways of computing commands. (coms (cond (x (list 1 x)))) will replace the first element of the current expression with the value of x if non-nil, otherwise do nothing. (nil as a command is a nop.)

(comsq com1 ... comn) . executes com1 ... comn. comsq is mainly useful in conjunction with the coms command. for example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the coms command. he would then write (coms (cons (quote comsq) x)) where x computed the list of commands, e.g., (coms (cons (quote comsq) (get foo (quote commands))))

16.11. Commands that Test

TESTING COMMAND SUMMARY

(if x) generates an error unless the value of (eval x) is non-nil, i.e., if (eval x) causes an error or (eval x)=nil, if will cause an error. (if x coms1 coms2) if (eval x) is non-nil, execute coms1; if (eval x) causes an error or is equal to nil, execute coms2. (if x coms1) if (eval x) is non-nil, execute coms1; otherwise generate an error.

(lp . coms) . repeatedly executes coms, a list of commands, until an error occurs. (lp f print (n t)) will attach a t at the end of every print expression. (lp f print (if (## 3) nil ((n t)))) will attach a t at the end of each print expression which does not already have a second argument. (i.e. the form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((n t)) as the list of commands to be executed. The if could also be written as (if (cddr (##)) nil ((n t))).).

(lpq . coms) same as lp but does not print n occurrences.

(orr coms1 ... comsn) . orr begins by executing coms1, a list of commands. If no error occurs, orr is finished. otherwise, orr restores the edit chain to its original value, and continues by executing coms2, etc. If none of the command lists execute without errors, i.e., the orr "drops off the end", orr generates an error. otherwise, the edit chain is left as of the completion of the first command list which executes without error.

16.12. Editor Macros

Many of the more sophisticated branching commands in the editor, such as orr, if, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire. (however, built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not modified.) macros are defined by using the m command.

(m c . coms) for c an atom, m defines c as an atomic command. (if a macro is redefined, its new definition replaces its old.) executing c is then the same as executing the list of commands coms. macros can also define list commands, i.e., commands that take arguments. (m (c) (arg[1] ... arg[n]) . coms) c an atom. m defines c as a list command. executing (c e1 ... en) is then performed by substituting e1 for arg[1], ... en for arg[n] throughout coms, and then executing coms. a list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments'. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. if the of arguments. (m (c) args . coms) c, args both atoms, defines c as a list command. executing (c e1 ... en) is performed by substituting (e1 ... en), i.e., cdr of the command, for args throughout coms, and then executing coms.

(m bp bk up p) will define bp as an atomic command which does three things, a bk, an up, and a p. note that macros can use commands defined by macros as well as built in commands in their definitions. for example, suppose z is defined by (m z -1 (if (null (##)) nil (p))), i.e. z does a -1, and then if the current expression is not nil, a p. now we can define zz by (m zz -1 z), and zzz by (m zzz -1 -1 z) or (m zzz -1 zz). we could define a more general bp by (m (bp) (n) (bk n) up p). (bp 3) would perform (bk 3), followed by an up, followed by a p. The command second can be defined as a macro by (m (2nd) x (orr ((lc . x) (lc . x)))).

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are completely independent. in other words, the existence of an atomic definition for c in no way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in no way affects the treatment of c when it appears as an atom. in particular, c can be used as the name of either an atomic command, or a list command, or both. in the latter case, two entirely different definitions can be used. note also that once c is defined as an atomic command via a macro definition, it will not be searched for when used in a location specification, unless c is preceded by an f. (insert -- before bp) would not search for bp, but instead perform a bk, an up, and a p, and then do the insertion. The corresponding also holds true for list commands.

(bind . coms) bind is an edit command which is useful mainly in macros. it binds three dummy variables #1, #2, #3, (initialized to nil), and then executes the edit commands coms. note that these bindings are only in effect while the commands are being executed, and that bind can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.

usermacros [value]. this variable contains the users editing macros . if you want to save your macros then you should save usermacros. you should probably also save editcomsl.

editcomsl [value]. editcomsl is the list of "list commands" recognized by the editor. (these are the ones of the form (command arg1 arg2 ...).)

16.13. Miscellaneous Editor Commands

MISCELLANEOUS EDITOR COMMAND SUMMARY

ok . Exits from the editor.

nil . Unless preceded by f or bf, is always a null operation.

tty: . Calls the editor recursively. The user can then type in commands, and have them executed. The tty: command is completed when the user exits from the lower editor (with ok or stop). the tty: command is extremely useful. it enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. for example the command (move 3 to after cond 3 p tty:) allows the user to interact, in effect, within the move command. he can verify for himself that the correct location has been found, or complete the specification "by hand". in effect, tty: says "I'll tell you what you should do when you get there."

stop . exits from the editor with an error. mainly for use in conjunction with tty: commands that the user wants to abort. since all of the commands in the editor are errset protected, the user must exit from the editor via a command. stop provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session.

tl . tl calls (top-level). to return to the editor just use the *return* top-level command.

repack . permits the 'editing' of an atom or string.

(repack $) does (lc . $) followed by repack, e.g. (repack this@).

(makefn form args n m) . makes (car form) an expr with the nth through mth elements of the current expression with each occurrence of an element of (cdr form) replaced by the corresponding element of args. The nth through mth elements are replaced by form.

(makefn form args n). same as (makefn form args n n).

(s var . $) . sets var (using setq) to the current expression after performing (lc . $). (s foo) will set foo to the current expression, (s foo -1 1) will set foo to the first element in the last element of the current expression

---

16.14. Editor Functions

(editf s_x1 ...**)**

SIDE EFFECT: edits a function. s_x1 is the name of the function, any additional arguments are an optional list of commands.

RETURNS: s_x1.

NOTE: if s_x1 is not an editable function, editf generates an fn not editable error.

(edite l_expr l_coms s_atm))
edits an expression. its value is the last element of (editl (list l_expr) l_coms s_atm nil nil).

(editracefn s_com)
is available to help the user debug complex edit macros, or subroutine calls to the editor. editracefn is to be defined by the user. whenever the value of editracefn is non-nil, the editor calls the function editracefn before executing each command (at any level), giving it that command as its argument. editracefn is initially equal to nil, and undefined.

(editv s_var [g_com1 ...])

SIDE EFFECT: similar to editf, for editing values. editv sets the variable to the value returned.

RETURNS: the name of the variable whose value was edited.

(editp s_x)

SIDE EFFECT: similar to editf for editing property lists. used if x is nil.

RETURNS: the atom whose property list was edited.

(editl coms atm marklst mess)

SIDE EFFECT: editl is the editor. its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited. (l is a special variable, and so can be examined or set by edit commands. ^ is equivalent to (e (setq l(last l)) t).) coms is an optional list of commands. for interactive editing, coms is nil. in this case, editl types edit and then waits for input from the teletype. (if mess is not nil editl types it instead of edit. for example, the tty: command is essentially (setq l (editl l nil nil nil (quote tty:))).) exit occurs only via an ok, stop, or save command. If coms is not nil, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed , the rest of the commands are ignored, and editl exits with an error, i.e., the effect is the same as though a stop command had been executed. If all commands execute successfully, editl returns the current value of l. marklst is the list of marks. on calls from editf, atm is the name of the function being edited; on calls from editv, the name of the variable, and calls from editp, the atom of which some property of its property list is being edited. The property list of atm is used by the save command for saving the state of the edit. save will not save anything if atm=nil i.e., when editing arbitrary expressions via edite or editl directly.

(editfns s_x [g_coms1 ...])
fsubr function, used to perform the same editing operations on several functions. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.

EXAMPLE: editfns foofns (r fie fum)) will change every fie to fum in each of the functions on foofns.

NOTE: the call to the editor is errset protected, so that if the editing of one function causes an error, editfns will proceed to the next function. in the above example, if one of the functions did not contain a fie, the r command would cause an error, but editing would continue with the next function. The value of editfns is nil.

(edit4e pat y)

SIDE EFFECT: is the pattern match routine.

RETURNS: t if pat matches y. see edit-match for definition of 'match'.

NOTE: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings that end in at-signs. These are replaced by patterns of the form (cons (quote /@) (explodec atom)). from the standpoint of edit4e, pattern type 5, atoms or strings ending in at-signs, is really "if car[pat] is the atom @ (at-sign), pat will match with any literal atom or string whose initial character codes (up to the @) are the same as those in cdr[pat]." if the user wishes to call edit4e directly, he must therefore convert any patterns which contain atoms or strings ending in at-signs to the form recognized by edit4e. this can be done via the function editfpat.

(editfpat pat flg)
makes a copy of pat with all patterns of type 5 (see edit-match) converted to the form expected by edit4e. flg should be passed as nil (flg=t is for internal use by the editor).

(editfindp x pat flg)

NOTE: Allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is t if the command f pat would succeed, nil otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=t. if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=t.

(## g_com1 ...)

RETURNS: what the current expression would be after executing the edit commands com1 ... starting from the present edit chain. generates an error if any of comi cause errors. The current edit chain is never changed. example: (i r (quote x) (## (cons ..z))) replaces all x's in the current expression by the first cons containing a z.

APPENDIX A

Special Symbols

The values of these symbols have a predefined meaning. Some values are counters while others are simply flags whose value the user can change to affect the operation of lisp system. In all cases, only the value cell of the symbol is important, the function cell is not. The value of some of the symbols (like **ER%misc**) are functions - what this means is that the value cell of those symbols either contains a lambda expression, a binary object, or symbol with a function binding.

The values of the special symbols are:

$gccount$ — The number of garbage collections which have occurred.

$gcprint — If bound to a non nil value, then after each garbage collection and subsequent storage allocation a summary of storage allocation will be printed.

$ldprint — If bound to a non nil value, then during each *fasl* or *cfasl* a diagnostic message will be printed.

ER%all — The function which is the error handler for all errors (see §10)

ER%brk — The function which is the handler for the error signal generated by the evaluation of the *break* function (see §10).

ER%err — The function which is the handler for the error signal generated by the evaluation of the *err* function (see §10).

ER%misc — The function which is the handler of the error signal generated by one of the unclassified errors (see §10). Most errors are unclassified at this point.

ER%tpl — The function which is the handler to be called when an error has occurred which has not been handled (see §10).

ER%undef — The function which is the handler for the error signal generated when a call to an undefined function is made.

^w — When bound to a non nil value this will prevent output to the standard output port (poport) from reaching the standard output (usually a terminal). Note that ^w is a two character symbol and should not be confused with ^W which is how we would denote control-w. The value of ^w is checked when the standard output buffer is flushed which occurs after a *terpr*, *drain* or when the buffer overflows. This is most useful in conjunction with ptport described below. System error handlers rebind ^w to nil when they are invoked to assure that error messages are not lost. (This was introduced for Maclisp compatibility).

defmacro-for-compiling — The has an effect during compilation. If non-nil it causes macros defined by defmacro to be compiled and included in the object file.

environment — The UNIX environment in assoc list form.

errlist – When a *reset* is done, the value of errlist is saved away and control is thrown to the top level. *Eval* is then mapped over the saved away value of this list.

errport – This port is initially bound to the standard error file.

evalhook – The value of this symbol, if bound, is the name of a function to handle evalhook traps (see §14.4)

float-format – The value of this symbol is a string which is the format to be used by print to print flonums. See the documentation on the UNIX function printf for a list of allowable formats.

funcallhook – The value of this symbol, if bound, is the name of a function to handle funcallhook traps (see §14.4).

gcdisable – If non nil, then garbage collections will not be done automatically when a collectable data type runs out.

ibase – This is the input radix used by the lisp reader. It may be either eight or ten. Numbers followed by a decimal point are assumed to be decimal regardless of what ibase is.

linel – The line length used by the pretty printer, pp. This should be used by *print* but it is not at this time.

nil – This symbol represents the null list and thus can be written (). Its value is always nil. Any attempt to change the value will result in an error.

piport – Initially bound to the standard input (usually the keyboard). A read with no arguments reads from piport.

poport – Initially bound to the standard output (usually the terminal console). A print with no second argument writes to poport. See also: ^w and ptport.

prinlength – If this is a positive fixnum, then the *print* function will print no more than prinlength elements of a list or hunk and further elements abbreviated as '...'. The initial value of prinlength is nil.

prinlevel – If this is a positive fixnum, then the *print* function will print only prinlevel levels of nested lists or hunks. Lists below this level will be abbreviated by '&' and hunks below this level will be abbreviated by a '%'. The initial value of prinlevel is nil.

ptport – Initially bound to nil. If bound to a port, then all output sent to the standard output will also be sent to this port as long as this port is not also the standard output (as this would cause a loop). Note that ptport will not get a copy of whatever is sent to poport if poport is not bound to the standard output.

readtable – The value of this is the current readtable. It is an array but you should NOT try to change the value of the elements of the array using the array functions. This is because the readtable is an array of bytes and the smallest unit the array functions work with is a full word (4 bytes). You can use *setsyntax* to change the values and *(status syntax ...)* to read the values.

t – This symbol always has the value t. It is possible to change the value of this symbol for short periods of time but you are strongly advised against it.

top-level — In a lisp system without /usr/lib/lisp/toplevel.l loaded, after a *reset* is done, the lisp system will *funcall* the value of top-level if it is non nil. This provides a way for the user to introduce his own top level interpreter. When /usr/lib/lisp/toplevel.l is loaded, it sets top-level to franz-top-level and changes the *reset* function so that once franz-top-level starts, it cannot be replaced by changing top-level. Franz-top-level does provide a way of changing the top level however, and that is through user-top-level.

user-top-level — If this is bound then after a *reset*, the top level function will *funcall* the value of this symbol rather than go through a read eval print loop.

APPENDIX B

Short Subjects.

The Garbage Collector

The garbage collector is invoked automatically whenever a collectable data type runs out. All data types are collectable except strings and atoms are not. After a garbage collection finishes, the collector will call the function *gcafter* which should be a lambda of one argument. The argument passed to *gcafter* is the name of the data type which ran out and caused the garbage collection. It is *gcafter*'s responsibility to allocate more pages of free space. The default *gcafter* makes its decision based on the percentage of space still in use after the garbage collection. If there is a large percentage of space still in use, *gcafter* allocates a larger amount of free space than if only a small percentage of space is still in use. The default *gcafter* will also print a summary of the space in use if the variable *$gcprint* is non nil. The summary always includes the state of the list and fixnum space and will include another type if it caused the garbage collection. The type which caused the garbage collection is preceded by an asterisk.

Debugging

There are two simple functions to help you debug your programs: *baktrace* and *showstack*. When an error occurs (or when you type the interrupt character), you will be left at a break level with the state of the computation frozen in the stack. At this point, calling the function *showstack* will cause the contents of the lisp evaluation stack to be printed in reverse chronological order (most recent first). When the programs you are running are interpreted or traced, the output of *showstack* can be very verbose. The function *baktrace* prints a summary of what *showstack* prints. That is, if showstack would print a list, *baktrace* would only print the first element of the list. If you are running compiled code with the *(status translink)* non nil, then fast links are being made. In this case, there is not enough information on the stack for *showstack* and *baktrace*. Thus, if you are debugging compiled code you should probably do *(sstatus translink nil)*.

If the contents of the stack don't tell you enough about your problem, the next thing you may want to try is to run your program with certain functions traced. You can direct the trace package to stop program execution when it enters a function, allowing you to examine the contents of variables or call other functions. The trace package is documented in Chapter 11.

It is also possible to single step the evaluator and to look at stack frames within lisp. The programs which perform these actions are described in Chapters 14 and 15.

The Interpreter's Top Level

The default top level interpreter for Franz, named *franz-top-level* is defined in /usr/lib/lisp/toplevel.l It is given control when the lisp system starts up because the variable top-level is bound to the symbol *franz-top-level.* The first action *franz-top-level* takes is to print out the name of the current version of the lisp system. Then it loads the file .lisprc from the HOME directory of the person invoking the lisp system if that file exists. The .lisprc file allows you to set up your own defaults, read in files, set up autoloading or anything else you might want to do to personalize the lisp system. Next, the top level goes into a prompt-read-eval-print loop. Each time around the loop, before printing the prompt it checks if the variable user-top-level is bound. If so, then the value of user-top-level will be *funcall*ed. This provides a convenient way for a user to introduce his own top level (Liszt, the lisp compiler, is an example of a program which uses this). If the user types a ^D (which is the end of file character), and the standard input is not from a keyboard, the lisp system will exit. If the standard input is a keyboard and if the value of *(status ignoreeof)* is nil, the lisp system will also exit. Otherwise the end of file will be ignored. When a *reset* is done the current value of *errlist* is saved away and control is thrown back up to the top level where *eval* is mapped over the saved value of *errlist.*

1. Background

FP stands for a *Functional Programming* language. Functional programs deal with *functions* instead of *values*. There is no explicit representation of state, there are no assignment statments, and hence, no variables. Owing to the lack of state, FP functions are free from side-effects; so we say the FP is *applicative*.

All functions take one argument and they are evaluated using the single FP operation, *application* (the colon ':' is the apply operator). For example, we read +:<3 4> as "apply the function '+' to its argument <3 4>".

Functional programs express a functional-level combination of their components instead of describing state changes using value-oriented expressions. For example, we write the function returning the *sin* of the *cos* of its input, *i.e.*, *sin*(*cos*(*x*)), as: sin @ cos. This is a *functional expression*, consisting of the single *combining form* called *compose* ('@' is the compose operator) and its *functional arguments sin* and *cos*.

All combining forms take functions as arguments and return functions as results; functions may either be applied, *e.g.*, sin@cos:3, or used as a functional argument in another functional expression, *e.g.*, *tan @ sin @ cos*.

As we have seen, FP's combining forms allow us to express control abstractions without the use of variables. The *apply to all* functional form (&) is another case in point. The function '& exp' exponentiates all the elements of its argument:

$$\&exp : \langle 1.0\ 2.0\rangle \equiv \langle 2.718\ 7.389\rangle \tag{1.1}$$

In (1.1) there are no induction variables, nor a loop bounds specification. Moreover, the code is useful for any size argument, so long as the sub-elements of its argument conform to the domain of the *exp* function.

We must change our view of the programming process to adapt to the functional style. Instead of writing down a set of steps that manipulate and assign values, we compose functional expressions using the higher-level functional forms. For example, the function that adds a scalar to all elements of a vector will be written in two steps. First, the function that distributes the scalar amongst each element of the vector:

$$distl : \langle 3\ \langle 4\ 6\rangle\rangle \equiv \langle\langle 3\ 4\rangle\ \langle 3\ 6\rangle\rangle \tag{1.2}$$

Next, the function that adds the pairs of elements that make up a sequence:

$$\&+ : \langle\langle 3\ 4\rangle\ \langle 3\ 6\rangle\rangle \equiv \langle 7\ 9\rangle \tag{1.3}$$

In a value-oriented programming language the computation would be expressed as:

$$\&+ : distl : \langle 3\ \langle 4\ 6\rangle\rangle, \tag{1.4}$$

which means to apply 'distl' to the input and then to apply '+' to every element of the result. In FP we write (1.4) as:

$$\&+ \ @\ distl : \langle 3\ \langle 4\ 6\rangle\rangle. \tag{1.5}$$

The functional expression of (1.5) replaces the two step value expression of (1.4).

Often, functional expressions are built from the inside out, as in LISP. In the next example we derive a function that scales then shifts a vector, *i.e.*, for scalars a, b and a vector $\overline{v}$, compute $a + b\overline{v}$. This FP function will have three arguments, namely a, b and $\overline{v}$. Of course, FP does not use formal parameter names, so they will be designated by the function symbols 1, 2, 3. The first code segment scales $\overline{v}$ by b (defintions are delimited with curly braces '{}'):

$$\{scaleVec\ \&*\ @\ distl\ @\ [2,3]\} \tag{1.6}$$

The code segment in (1.5) shifts the vector. The completed function is:

$$\{changeVec\ \&+\ @\ distl\ @\ [1\ ,\ scaleVec]\} \tag{1.7}$$

In the derivation of the program we wrote from right to left, first doing the *distl*'s and then composing with the *apply-to-all* functional form. Using an imperative language, such as Pascal, we would write the program from the outside in, writing the loop before inserting the arithmetic operators.

Although FP encourages a recursive programming style, it provides combining forms to avoid explicit recursion. For example, the right insert combining form (!) can be used to write a function that adds up a list of numbers:

$$!+ : <1\ 2\ 3> \equiv 6 \tag{1.8}$$

The equivalent, recursive function is much longer:

$$\{addNumbers\ (null\ ->\ \%0\ ;\ +\ @\ [1,\ addNumbers\ @\ tl])\} \tag{1.9}$$

The generality of the combining forms encourages hierarchical program development. Unlike APL, which restricts the use of combining forms to certain builtin functions, FP allows combining forms to take any functional expression as an argument.

2. System Description

2.1. Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, *i.e.*, 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fod80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom **?**, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, *e.g.*, division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	"*CD*"	<1,<2,3>>
<>	**T**	<*a*,<>>

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, *e.g.*, $<1,?> \equiv ?$. This property is the so-called "bottom preserving property" [Ba78].

2.2. Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (*i.e.*, evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

+:<7,8>	≡	15	**tl**:<1,2,3>	≡	<2,3>
1 :<*a*,*b*,*c*,*d*>	≡	*a*	**2** :<*a*,*b*,*c*,*d*>	≡	*b*

2.3. Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma : ? \equiv ?, \quad \forall\ \sigma \in \mathbf{F} \tag{2.1}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n ; e_{n+1} \tag{2.2}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that $x, x_i, y, y_i, z_i \in \Omega$.

2.3.1. Structural

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; 0 < \mu \leqslant k \rightarrow x_\mu;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; -k \leqslant \mu < 0 \rightarrow x_{k+\mu+1};\ ?$

pick : $\langle n,x\rangle \equiv$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; 0 < n \leqslant k \rightarrow x_n;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; -k \leqslant n < 0 \rightarrow x_{k+n+1};\ ?$

The user should note that the function symbols **1,2,3,...** are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

$x=\langle\rangle \rightarrow \langle\rangle\ ;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; k \geqslant 1 \rightarrow x_k;\ ?$

first : $x \equiv$

$x=\langle\rangle \rightarrow \langle\rangle\ ;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; k \geqslant 1 \rightarrow x_1;\ ?$

Tail Functions

tl : $x \equiv$

$x=\langle x_1\rangle \rightarrow \langle\rangle\ ;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; k \geqslant 2 \rightarrow \langle x_2, \ldots, x_k\rangle\ ;\ ?$

tlr : $x \equiv$

$x=\langle x_1\rangle \rightarrow \langle\rangle\ ;$

$x=\langle x_1, x_2, \ldots, x_k\rangle \;\wedge\; k \geqslant 2 \rightarrow \langle x_1, \ldots, x_{k-1}\rangle\ ;\ ?$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

$x=\langle y,\langle\rangle\rangle \rightarrow \langle\rangle;$

$x=\langle y,\langle z_1, z_2, \ldots, z_k\rangle\rangle \rightarrow \langle\langle y,z_1\rangle,\ldots,\langle y,z_k\rangle\rangle;\ ?$

distr : $x \equiv$

$x=<<>,y> \rightarrow <>;$

$x=<<y_1, y_2, \ldots, y_k>,z> \rightarrow <<y_1,z>,\ldots,<y_k,z>>;$ *?*

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to **id**. Like **id** it returns its argument as the result, unlike **id** it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

$x=<y,<>> \rightarrow <y>;$

$x=<y,<z_1, z_2, \ldots, z_k>> \rightarrow <y,z_1, z_2, \ldots, z_k>;$ *?*

apndr : $x \equiv$

$x=<<>,z> \rightarrow <z>;$

$x=<<y_1, y_2, \ldots, y_k>,z> \rightarrow <y_1, y_2, \ldots, y_k, z>;$ *?*

Transpose

trans : $x \equiv$

$x=<<>,\ldots,<>> \rightarrow <>;$

$x=<x_1, x_2, \ldots, x_k> \rightarrow <y_1, \ldots, y_m>;$ *?*

where $x_i = <x_{i1}, \ldots, x_{im}> \; \backslash \; y_j = <x_{1j}, \ldots, x_{kj}>,$

$1 \leqslant i \leqslant k \; , \; 1 \leqslant j \leqslant m.$

reverse : $x \equiv$

$x=<> \rightarrow;$

$x=<x_1, x_2, \ldots, x_k> \rightarrow <x_k, \ldots, x_1>;$ *?*

Rotate Left and Right

rotl : $x \equiv$

$x=<> \rightarrow <>; \; x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \; \backslash \; k \geqslant 2 \rightarrow <x_2, \ldots, x_k, x_1>;$ *?*

rotr :$x \equiv$

$x=<> \rightarrow <>; \; x=<x_1> \rightarrow <x_1>;$

$x=<x_1, x_2, \ldots, x_k> \; \backslash \; k \geqslant 2 \rightarrow <x_k, x_1, \ldots, x_{k-2}, x_{k-1}>;$ *?*

concat : $x \equiv$

$x=<<x_{11}, \ldots, x_{1k}>, <x_{21}, \ldots, x_{2n}>, \ldots, <x_{m1}, \ldots, x_{mp}>> \wedge k, m, n, p > 0 \rightarrow$
$<x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}>; ?$

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5> (2.3)

pair : $x \equiv$

$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is even* $\rightarrow <<x_1,x_2>, \ldots, <x_{k-1},x_k>>;$
$x=<x_1, x_2, \ldots, x_k> \wedge k>0 \wedge k$ *is odd* $\rightarrow <<x_1,x_2>, \ldots, <x_k>>; ?$

split : $x \equiv$

$x=<x_1> \rightarrow <<x_1>,<>>;$
$x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow <<x_1, \ldots, x_{\lceil k/2 \rceil}>, <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; ?$

iota : $x \equiv$

$x=0 \rightarrow <>;$
$x \in \mathbf{N}^+ \rightarrow <1,2,\ldots,x>; ?$

2.3.2. Predicate (Test) Functions

atom : $x \equiv x \in$ *atoms* $\rightarrow$ **T**; $x \neq ? \rightarrow$ **F**; *?*

eq : $x \equiv x = <y,z> \wedge y=z \rightarrow$ **T**; $x=<y,z> \wedge y \neq z \rightarrow$ **F**; *?*

Also less than (**<**), greater than (**>**), greater than or equal (**>=**), less than or equal (**<=**), not equal (**~=**); '**=**' is a synonym for **eq**.

null : $x \equiv x=<> \rightarrow$ **T**; $x \neq ? \rightarrow$ **F**; *?*

length : $x \equiv x = <x_1, x_2, \ldots, x_k> \rightarrow k$; $x=<> \rightarrow 0$; *?*

2.3.3. Arithmetic/Logical

+ : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y+z$; *?*
− : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y-z$; *?*
*** : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\rightarrow y \times z$; *?* **/** : $x \equiv x=<y,z> \wedge y,z$ *are numbers* $\wedge z \neq 0 \rightarrow y \div z$; *?*

And, or, not, xor

and : $<x,y> \equiv x=\mathbf{T} \rightarrow y$; $x=\mathbf{F} \rightarrow$ **F**; *?*

or : $<x,y> \equiv x=\mathbf{F} \rightarrow y$; $x=\mathbf{T} \rightarrow$ **T**; *?*

xor : $<x,y> \equiv$
$x=\mathbf{T} \wedge y=\mathbf{T} \rightarrow \mathbf{F}$; $x=\mathbf{F} \wedge y=\mathbf{F} \rightarrow \mathbf{F}$;
$x=\mathbf{T} \wedge y=\mathbf{F} \rightarrow \mathbf{T}$; $x=\mathbf{F} \wedge y=\mathbf{T} \rightarrow \mathbf{T}$; *?*

not : $x \equiv x=\mathbf{T} \rightarrow F$; $x=\mathbf{F} \rightarrow \mathbf{T}$; *?*

2.3.4. Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x)$; *?*

asin : $x \equiv x$ is a number $\wedge\ |x| \leqslant 1 \rightarrow \sin^{-1}(x)$; *?*

cos : $x \equiv x$ is a number $\rightarrow cos(x)$; *?*

acos : $x \equiv x$ is a number $\wedge\ |x| \leqslant 1 \rightarrow \cos^{-1}(x)$; *?*

exp : $x \equiv x$ is a number $\rightarrow e^{x}$; *?*

log : $x \equiv x$ is a positive number $\rightarrow ln(x)$; *?*

mod : $<x,y> \equiv x$ **and** y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor$; *?*

2.4. Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition.* For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi \ @\ \psi) : x \equiv \phi:(\psi:x), \quad \forall \quad x \in \Omega \tag{2.4}$$

The *constant* function takes an object parameter:

$$\%x:y \equiv y=? \rightarrow ?;\ x, \quad \forall \quad x,y \in \Omega \tag{2.5}$$

The function $\%?$ always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma \ @\ \tau):x \equiv \sigma:(\tau:x)$

Construction

$[\sigma_1, \ldots, \sigma_n]:x \equiv <\sigma_1:x,\ldots,\sigma_n:x>$

Note that construction is also bottom-preserving, *e.g.*,

$$[+,/]:<3,0> = <3,?> = ? \tag{2.6}$$

Condition

$$(\xi \text{ -> } \sigma;\ \tau):x \equiv$$
$$(\xi:x)=\mathbf{T} \rightarrow \sigma:x;$$
$$(\xi:x)=\mathbf{F} \rightarrow \tau:x;\ ?$$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional form *must* be enclosed in parenthesis, *e.g.*,

```
(isNegative -> - @ [%0,id] ; id)
```
(2.7)

Constant

$$\%x:y \equiv y=? \rightarrow ?;\ x, \qquad \forall\ x \in \Omega$$

This function returns its object parameter as its result.

Right Insert

$$!\sigma :x \equiv$$
$$x=<> \rightarrow e_f:x;$$
$$x=<x_1> \rightarrow x_1;$$
$$x=<x_1, x_2, \ldots, x_k> \wedge k>2 \rightarrow \sigma:<x_1, !\sigma:<x_2, \ldots, x_k>>;\ ?$$

e.g., $!+:<4,5,6>=15$.

If σ has a right identity element e_f, then $!\sigma:<> = e_f$, *e.g.*,

$$!+:<>=0 \text{ and } !*:<>=1 \tag{2.8}$$

Currently, identity functions are defined for + (0), − (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$$
\begin{aligned}
&|\sigma : x \equiv \\
&\quad x=<> \rightarrow e_f:x; \\
&\quad x=<x_1> \rightarrow x_1; \\
&\quad x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow \\
&\quad \sigma : <|\sigma : <x_1, \ldots, x_{\lceil k/2 \rceil}> , |\sigma : <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; ?
\end{aligned}
$$

e.g.,

$$\vdash:<4,5,6,7> \equiv +:<+:<4,5>,+:<6,7>> \equiv 15 \qquad (2.9)$$

Tree insert uses the same identity functions as right insert.

Apply to All

$$
\begin{aligned}
&\&\sigma: x \equiv \\
&\quad x=<> \rightarrow <>; \\
&\quad x=<x_1, x_2, \ldots, x_k> \rightarrow <\sigma:x_1, \ldots, \sigma:x_k>; ?
\end{aligned}
$$

While

$$
\begin{aligned}
&(\textbf{while } \xi\ \sigma):x \equiv \\
&\quad \xi:x=\mathbf{T} \rightarrow (\textbf{while } \xi\ \sigma):(\sigma:x); \\
&\quad \xi:x=\mathbf{F} \rightarrow x; ?
\end{aligned}
$$

2.5. User Defined Functions

An FP definition is entered as follows:

$$\{\textit{fn-name fn-form}\}, \qquad (2.10)$$

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

$$\{\textit{factorial } !^* @ \textit{ iota}\} \qquad (2.11)$$

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x$, $\forall\ x \in \Omega$.

References to undefined functions bottom out:

$$f:x \equiv ?\forall\ x \in \Omega, f \not\in \mathbf{F} \qquad (2.12)$$

3. Getting on and off the System

Startup FP from the shell by entering the command:

/usr/local/fp.

The system will prompt you for input by indenting over six character positions. Exit from FP (back to the shell) with a control/D (^D).

3.1. Comments

A user may end any line (including a command) with a comment; the comment character is '#'. The interpreter will ignore any character after the '#' until it encounters a newline character or end-of-file, whichever comes first.

3.2. Breaks

Breaks interrupt any work in progress causing the system to do a FRANZ reset before returning control back to the user.

3.3. Non-Termination

LISP's namestack may, on occasion, overflow. FP responds by printing "non-terminating" and returning bottom as the result of the application. It does a FRANZ reset before returning control to the user.

4. System Commands

System commands start with a right parenthesis and they are followed by the command-name and possibly one or more arguments. All this information *must be typed on a single line*, and any number of spaces or tabs may be used to separate the components.

4.1. Load

Redirect the standard input to the file named by the command's argument. If the file doesn't exist then FP appends '.fp' to the file-name and retries the open (error if the file doesn't exist). This command allows the user to read in FP function definitions from a file. The user can also read in applications, but such operation is of little utility since none of the input is echoed at the terminal. Normally, FP returns control to the user on an end-of-file. It will also do so whenever it does a FRANZ reset, e.g., whenever the user issues a break, or whenever the system encounters a non-terminating application.

4.2. Save

Output the source text for all user-defined functions to the file named by the argument.

4.3. Csave and Fsave

These commands output the lisp code for all the user-defined functions, including the original source-code, to the file named by the argument. Csave pretty prints the code, Fsave does not. Unless the user wishes to examine the code, he should use 'fsave'; it is about ten times faster than 'csave', and the resulting file will be about three times smaller.

These commands are intended to be used with the liszt compiler and the 'cload' command, as explained below.

4.4. Cload

This command loads or fasls in the file shown by the argument. First, FP appends a '.o' to the file-name, and attempts a load. Failing that, it tries to load the file named by the argument. If the user outputs his function definitions using fsave or csave, and then compiles them using liszt, then he may fasl in the compiled code and speed up the execution of his defined functions by a factor of 5 to 10.

4.5. Pfn

Print the source text(s) (at the terminal) for the user-defined function(s) named by the argument(s) (error if the function doesn't exist).

4.6. Delete

Delete the user-defined function(s) named by the argument (error if the function doesn't exist).

4.7. Fns

List the names of all user-defined functions in alphabetical order. Traced functions are labeled by a trailing '@' (see § 4.7 for sample output).

4.8. Stats

The "stats" command has several options that help the user manage the collection of dynamic statistics for functions[1] and functional forms. Option names follow the keyword "stats", e.g., ")stats reset".

The statistic package records the frequency of usage for each function and functional form; also the size[2] of all the arguments for all functions and functional expressions. These two measures allow the user to derive the average argument size per call. For functional forms the package tallies the frequency of each functional argument. Construction has an additional statistic that tells the number of functional arguments involved in the construction.

Statistics are gathered whenever the mode is on, except for applications that "bottom out" (*i.e.*, return bottom − ?). Statistic collection slows the system down by ×2 to ×4. The following printout illustrates the use of the statistic package (user input is emboldened):

---

[1] Measurement of user-defined functions is done with the aid of the trace package, discussed in § 4.9.

[2] "Size" is the top-level length of the argument, for most functions. Exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

```
)stats on
```

Stats collection turned on.

```
+:<3 4>
7
!* @ iota :3
6
)stats print
plus:       times 1
times:      times 2
iota:       times 1
insert:     times 1     size          3
                    Functional Args
                Name                          Times
                times                 1
compos:     times 1     size          1
                    Functional Args
                Name                          Times
                insert                1
                iota                          1
```

4.8.1. On

Enable statistics collection.

4.8.2. Off

Disable statistics collection. The user may selectively collect statistics using the on and off commands.

4.8.3. Print

Print the dynamic statistics at the terminal, or, output them to a file. The latter option requires an additional argument, *e.g.*, "")stats print fooBar" prints the stats to the file "fooBar".

4.8.4. Reset

Reset the dynamic statistics counters. To prevent accidental loss of collected statistics, the system will query the user if he tries to reset the counters without first outputting the data (the system will also query the user if he tries to log out without outputting the data).

4.9. Trace

Enable or disable the tracing and the dynamic measurement of the user defined functions named by the argument(s). The first argument tells whether to turn tracing off or on and the others give the name of the functions affected. The tracing and untracing commands are independent of the dynamic statistics commands. This command is cumulative *e.g.,* to ')trace on f1 f2'.

FP tracer output is similar to the FRANZ tracer output: function entries and exits, call level, the functional argument (remember that FP functions have only one argument!), and the result, are printed at the terminal:

```
                )pfn fact

{fact (eq0 -> %1 ; * @ [id, fact @ s1])}
                )fns

eq0             fact  s1

                )trace on fact
                )fns

eq0             fact@      s1

                fact : 2

1 >Enter> fact [2]
| 2 >Enter> fact [1]
| | 3 >Enter> fact [0]
| | 3 <EXIT<  fact  1
| 2 <EXIT<  fact  1
1 <EXIT<  fact  2

2
```

4.10. Timer

FP provides a simple timing facility to time top-level applications. The command "')timer on" puts the system in timing mode, "')timer off" turns the mode off (the mode is initially off). While in timing mode, the system reports CPU time, garbage collection time, and elapsed time, in seconds. The timing output follows the printout of the result of the application.

4.11. Script

Open or close a script file. The first argument gives the option, the second the optional script file-name. The "open" option causes a new script-file to be opened and any currently open script file to be closed. If the file cannot be opened, FP sends and error message and, if a script file was already opened, it remains open. The command "')script close" closes an open script file. The user may elect to append script output to the script-file with the append mode.

4.12. Help

Print a short summary of all the system commands:

```
)help
Commands are:

load <file>                        Redirect input from <file>
save <file>                        Save defined fns in <file>
pfn <fn1> ...                      Print source text of <fn1> ...
delete <fn1> ...                   Delete <fn1> ...
fns                                List all functions
stats on/off/reset/print [file]    Collect and print dynamic stats
trace on/off <fn1> ...             Start/Stop exec trace of <fn1> ...
timer on/of                        Turn timer on/off
script open/close/append           Open or close a script-file
lisp                               Exit to the lisp system (return with '^D')
debug on/off                       Turn debugger output on/off
csave <file>                       Output Lisp code for all user-defined fns
cload <file>                       Load Lisp code from a file (may be compiled)
fsave <file>                       Same as csave except without pretty-printing
```

4.13. Special System Functions

There are two system functions that are not generally meant to be used by average users.

4.13.1. Lisp

This exits to the lisp system. Use "^D" to return to FP.

4.13.2. Debug

Turns the 'debug' flag on or off. The command ")debug on" turns the flag on, ")debug off" turns the flag off. The main purpose of the command is to print out the parse tree.

5. Programming Examples

We will start off by developing a larger FP program, *mergeSort*. We measure *mergeSort* using the trace package, and then we comment on the measurements. Following *mergeSort* we show an actual session at the terminal.

5.1. MergeSort

The source code for *mergeSort* is:

```
# Use a divide and conquer strategy
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [[], fixLists]}

# Must convert atomic arguments into sequences
# Atomic arguments occur at the leaves of the execution tree
{fixLists &(atom -> [id] ; id)}

# Merge until one or both input lists are empty
{mergeHelp (while and @ &(not@null) @ 2
                    (firstIsSmaller -> takeFirst ;
                                       takeSecond))}

# Find the list with the smaller first element
{firstIsSmaller < @ [1@1@2, 1@2@2]}

# Take the first element of the first list
{takeFirst [apndr@[1,1@1@2], [tl@1@2, 2@2]]}

# Take the first element of the second list
{takeSecond [apndr@[1,1@2@2], [1@2, tl@2@2]]}

# If one list isn't null, then append it to the
# end of the merged list
{atEnd (firstIsNull -> concat@[1,2@2] ;
                       concat@[1,1@2])}

{firstIsNull null@1@2}
```

The merge sort algorithm uses a divide and conquer strategy; it splits the input in half, recursively sorts each half, and then merges the sorted lists. Of course, all these sub-sorts can execute in parallel, and the tree-insert (|) functional form expresses this concurrency. *Merge* removes successively larger elements from the heads of the two lists (either *takeFirst* or *takeSecond*) and appends these elements to the end of the merged sequence. *Merge* terminates when one sequence is empty, and then *atEnd* appends any remaining non-empty sequence to the end of the merged one.

On the next page we give the trace of the function *merge*, which information we can use to determine the structure of *merge*'s execution tree. Since the tree is well-balanced, many of the *merge*'s could be executed in parallel. Using this trace we can also calculate the average length of the arguments passed to *merge*, or a distribution of argument lengths. This information is useful for determining communication costs.

```
)trace on merge

mergeSort : <0 3 -2 1 11 8 -22 -33>
|3 >Enter> merge [<0 3>]
|3 <EXIT<  merge  <0 3>
|3 >Enter> merge [<-2 1>]
|3 <EXIT<  merge  <-2 1>
2 >Enter> merge [<<0 3> <-2 1>>]
2 <EXIT<  merge  <-2 0 1 3>
|3 >Enter> merge [<11 8>]
|3 <EXIT<  merge  <8 11>
|3 >Enter> merge [<-22 -33>]
|3 <EXIT<  merge  <-33 -22>
2 >Enter> merge [<<8 11> <-33 -22>>]
2 <EXIT<  merge  <-33 -22 8 11>
1 >Enter> merge [<<-2 0 1 3> <-33 -22 8 11>>]
1 <EXIT<  merge  <-33 -22 -2 0 1 3 8 11>

<-33 -22 -2 0 1 3 8 11>
```

5.2. FP Session

User input is **emboldened**, terminal output in Roman script.

```
fp

FP, v. 4.1 11/31/82
        )load ex_man
{all_le}
{sort}
{abs_val}
{find}
{ip}
{mm}
{eq0}
{fact}
{sub1}
{alt_fnd}
{alt_fact}
        )fns

abs_val   all_le   alt_fact   alt_fnd   eq0   fact   find
ip        mm       sort       sub1

        abs_val : 3

3

        abs_val : -3

3

        abs_val : 0

0

        abs_val : <-5 0 66>

?

        &abs_val : <-5 0 66>

<5 0 66>

        )pfn abs_val

{abs_val ((> @ [id,%0]) -> id ; (- @ [%0,id]))}

        [id,%0] : -3

<-3 0>

        [%0,id] : -3
```

```
<0 -3>
        - @ [%0,id] : -3
3
        all_le : <1 3 5 7>
T
        all_le : <1 0 5 7>
F
        )pfn all_le
{all_le ! and @ &<= @ distl @ [1,tl]}
        distl @ [1,tl] : <1 2 3 4>
<<1 2> <1 3> <1 4>>
        &<= @ distl @ [1,tl] : <1 2 3 4>
<T T T>
        ! and : <F T T>
F
        ! and : <T T T>
T
        sort : <3 1 2 4>
<1 2 3 4>
        sort : <1>
<1>
        sort : <>
?
        sort : 4
?
        )pfn sort
{sort (null @ tl -> [1] ; (all_le -> apndl @ [1,sort@tl]; sort@rotl))}
        fact : 3
6
```

```
        )pfn fact sub1 eq0
{fact (eq0 -> %1 ; *@[id , fact@sub1])}
{sub1 -@[id,%1]}
{eq0 = @ [id,%0]}
        &fact : <1 2 3 4 5>
<1 2 6 24 120>
        eq0 : 3
F
        eq0 : <>
F
        eq0 : 0
T
        sub1 : 3
2
        %1 : 3
1
        alt_fact : 3
6
        )pfn alt_fact
{alt_fact !* @ iota}
        iota : 3
<1 2 3>
        !* @ iota : 3
6
        !+ : <1 2 3>
6
        find : <3 <3 4 5>>
T
        find : <<> <3 4 <>>>
```

```
T
        find : <3 <4 5>>
F
        )pfn find
{find (null@2 -> %F ; (=@[1,1@2] -> %T ; find@[1,tl@2]))}
        [1,tl@2] : <3 <3 4 5>>
<3 <4 5>>
        [1,1@2] : <3 <3 4 5>>
<3 3>
        alt_fnd : <3 <3 4 5>>
T
        )pfn alt_fnd
{alt_fnd ! or @ &eq @ distl }
        distl : <3 <3 4 5>>
<<3 3> <3 4> <3 5>>
        &eq @ distl : <3 <3 4 5>>
<T F F>
        !or : <T F T>
T
        !or : <F F F>
F
        )delete alt_fnd
        )fns
abs_val   all_le   alt_fact   eq0   fact   find   ip
mm        sort     subl
        alt_fnd : <3 <3 4 5>>
alt_fnd not defined
?
        {g g}
```

```
{g}
        g : 3

non-terminating

?

[Return to top level]

FP, v. 4.0 10/8/82
        [+,*] : <3 4>

<7 12>

            [+,* : <3 4>

    syntax error:

    [+,* : <3 4>
           ^

        ip : <<3 4 5> <5 6 7>>

74

        )pfn ip

{ip !+ @ &* @ trans}

        trans : <<3 4 5> <5 6 7>>

<<3 5> <4 6> <5 7>>

        &* @ trans : <<3 4 5> <5 6 7>>

<15 24 35>

        mm : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<3 4> <5 6>>

        )pfn mm

{mm &&ip @ &distl @ distr @[1,trans@2]}

        [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <0 1>> <<3 4> <5 6>>>

        distr : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

        &distl : <<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

<<<<1 0> <3 4>> <<1 0> <5 6>>> <<<0 1> <3 4>> <<0 1> <5 6>>>>
```

&ip @ &dist & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

syntax error:

[+,* : <3 4>
  ^

&ip @ &distl & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
             ^

&ip @ &distl @ distr @ [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

?

6. Implementation Notes

FP was written in 3000 lines of FRANZ LISP [Fod 80]. Table 1 breaks down the distribution of the code by functionality.

Functionality	% (bytes)
compiler	34
user interface	32
dynamic stats	16
primitives	14
miscellaneous	3

Table 1

6.1. The Top Level

The top-level function *runFp* starts up the subsystem by calling the routine *fpMain*, that takes three arguments:

(1) A boolean argument that says whether debugging output will be enabled.

(2) A Font identifier. Currently the only one is supported **'asc** (ASCII).

(3) A boolean argument that identifies whether the interpreter was invoked from the shell. If so then all exits from FP return the user back to the shell.

The compiler converts the FP functions into LISP equivalents in two stages: first it forms the parse tree, and then it does the code generation.

6.2. The Scanner

The scanner consists of a main routine, *get_tkn*, and a set of action functions. There exists one set of action functions for each character font (currently only ASCII is supported). All the action functions are named *scan$<font>*, where *<font>* is the specified font, and each is keyed on a particular character (or sometimes a particular character-type – *e.g.*, a letter or a number). *get_tkn* returns the token type, and any ancillary information, *e.g.*, for the token "name" the name itself will also be provided. (See Appendix C for the font-token name correspondences). When a character has been read the scanner finds the action function by doing a

(get 'scan$ <font> <char>)

A syntax error message will be generated if no action exists for the particular character read.

6.3. The Parser

The main parsing function, *parse*, accepts a single argument, that identifies the parsing context, or type of construct being handled. Table 2 shows the valid parsing contexts.

id	construct
top_lev	initial call
constr*$$*	construction
compos*$$*	composition
alpha*$$*	apply-to-all
insert*$$*	insert
ti*$$*	tree insert
arrow*$$*	affirmative clause of conditional
semi*$$*	negative clause of conditional
lparen*$$*	parenthetic expr.
while*$$*	while

Table 2, Valid Parsing Contexts

For each type of token there exists a set of parse action functions, of the name *p$<tkn-name>*. Each parse-action function is keyed on a valid context, and it is looked up in the same manner as scan action functions are looked up. If an action function cannot be found, then there is a syntax error in the source code. Parsing proceeds as follows: initially *parse* is called from the top-level, with the context argument set to *"top_lev"*. Certain tokens cause parse to be recursively invoked using that token as a context. The result is the parse tree.

6.4. The Code Generator

The system compiles FP source into LISP source. Normally, this code is interpreted by the FRANZ LISP system. To speed up the implementation, there is an option to compile into machine code using the *liszt* compiler [Joy 79]. This feature improves performance tenfold, for some programs.

The compiler expands all functional forms into their LISP equivalents instead of inserting calls to functions that generate the code at run-time. Otherwise, *liszt* would be ineffective in speeding up execution since all the functional forms would be executed interpretively. Although the amount of code generated by an expanding compiler is 3 or 4 times greater than would be generated by a non-expanding compiler, even in interpreted mode the code runs twice as quickly as unexpanded code. With *liszt* compilation this performance advantage increases to more than tenfold.

A parse tree is either an atom or a hunk of parse trees. An atomic parse-tree identifies either an fp built-in function or a user defined function. The hunk-type parse tree represents functional forms, *e.g.*, compose or construct. The first element identifies the functional form and the other elements are its functional parameters (they may in turn be functional forms). Table 3 shows the parse-tree formats.

Form	Format
user-defined	<atom>
fp builtin	<atom>
apply-to-all	{*alpha$$* Φ}
insert	{*insert $$* Φ}
tree insert	{*ti $$* Φ}
select	{*select $$* μ}
constant	{*constant $$* μ}
conditional	{*condit $$* Φ_1 Φ_2 Φ_3}
while	{*while $$* Φ_1 Φ_2}
compose	{*compos $$* Φ_1 Φ_2}
construct	{*constr $$* Φ_1 Φ_2 , . . . , Φ_n *nil*}

Note: Φ and the Φ_k are parse-trees and μ is an optionally signed integer constant.

Table 3, Parse-Tree Formats

6.5. Function Definition and Application

Once the code has been generated, then the system defines the function via *putd*. The source code is placed onto a property list, *'sources*, to permit later access by the system commands.

For an application, the indicated function is compiled and then defined, only temporarily, as *tmp $$*. The single argument is read and *tmp $$* is applied to it.

6.6. Function Naming Conventions

When the parser detects a named primitive function, it returns the name *<name>$fp*, where *<name>* is the name that was parsed (all primitive function-names end in *$fp*). See Appendix D for the symbolic (*e.g.*, compose, +) function names.

Any name that isn't found in the list of builtin functions is assumed to represent a user-defined function; hence, it isn't possible to redefine FP primitive functions. FP protects itself from accidental or malicious internal destruction by appending the suffix "*_fp*" to all user-defined function names, before they are defined.

6.7. Measurement Impelementation

This work was done by Dorab Patel at UCLA. Most of the measurement code is in the file 'fpMeasures.l'. Many of the remaining changes were effected in 'primFp.l', to add calls on the measurement package at run-time; to 'codeGen.l', to add tracing of user defined functions; to 'utils.l', to add the new system commands; and to 'fpMain.l', to protect the user from forgetting to output statistics when he leaves FP.

6.7.1. Data Structures

All the statistics are in the property list of the global symbol *Measures*. Associated with each each function (primitive or user-defined, or functional form) is an indicator; the statistics gathered for each function are the corresponding values. The names corresponding to primitive functions and functional forms end in '$fp' and the names corresponding to user-defined functions end in '_fp'. Each of the property values is an association list:

```
(get 'Measures 'rotl$fp) ==> ((times . 0) (size . 0))
```

The car of the pair is the name of the statistic (*i.e.*, times, size) and the cdr is the value. There is one exception. Functional forms have a statistic called funargtyp. Instead of being a dotted pair, it is a list of two elements:

```
(get 'Measures 'compose$fp) ==>
((times . 2) (size . 4) (funargtyp ((select$fp . 2) (sub$fp . 2))))
```

The car is the atom 'funargtyp' and the cdr is an alist. Each element of the alist consists of a functional argument-frequency dotted pair.

The statistic packages uses two other global symbols. The symbol DynTraceFlg is non-nil if dynamic statistics are being collected and is nil otherwise. The symbol TracedFns is a list (initially nil) of the names of the user functions being traced.

6.7.2. Interpretation of Data Structures

6.7.2.1. Times

The number of times this function has been called. All functions and functional forms have this statistic.

6.7.2.2. Size

The sum of the sizes of the arguments passed to this function. This could be divided by the times statistic to give the average size of argument this function was passed. With few exceptions, the size of an object is its top-level length (note: version 4.0 defined the size as the total number of *atoms* in the object); the empty sequence, "<>", has a size of 0 and all other atoms have size of one. The exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

This statistic is not collected for some primitive functions (mainly binary operators like +,-,*).

6.7.2.3. Funargno

The number of functional arguments supplied to a functional form.

Currently this statistic is gatherered only for the construction functional form.

6.7.2.4. Funargtyp

How many times the named function was used as a functional parameter to the particular functional form.

6.8. Trace Information

The level number of a call shows the number of steps required to execute the function on an ideal machine (*i.e.*, one with unbounded resources). The level number is calculated under an assumption of infinite resources, and the system evaluates the condition of a conditional before evaluating either of its clauses. The number of functions executed at each level can give an idea of the distribution of parallelism in the given FP program.

7. Acknowledgements

Steve Muchnick proposed the initial construction of this system. Bob Ballance added some of is own insights, and John Foderaro provided helpful hints regarding effective use of the FRANZ LISP system [Fod80]. Dorab Patel [Pat81] wrote the dynamic trace and statistics package and made general improvements to the user interface. Portions of this manual were

excerpted from the *COMPCON-83 Digest of Papers*[3].

8. References

[Bac78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, Turing Award Lecture, 21, 8 (August 1978), 613-641.

[Fod80]
John K. Foderaro, "The FRANZ LISP Manual," University of California, Berkeley, California, 1980.

[Joy79]
W.N. Joy, O. Babaoglu, "UNIX Programmer's Manual," November 7, 1979, Computer Science Division, University of California, Berkeley, California.

[Mc60]
J. McCarthy, "Recursive Functions of Symbolic expressions and their Computation by Machine," Part I, *CACM* 3, 4 (April 1960), 184-195.

[Pat80]
Dorab Ratan Patel, "A System Organization for Applicative Programming," M.S Thesis, University of California, Los Angeles, California, 1980.

[Pat81]
Dorab Patel, "Functional Language Interpreter User Manual," University of California, Los Angeles, California, 1981.

---

[3] Scott B. Baden and Dorab R. Patel, "Berkeley FP – Experiences With a Functional Programming Language", © 1982, IEEE.

Appendix A: Local Modifications

1. Character Set Changes

Backus [Ba78] used some characters that do not appear on our ASCII terminals, so we have made the following substitutions:

constant	$\bar{x}$	%x
insert	/	!
apply-to-all	α	&
composition	$\circ$	@
arrow	$\rightarrow$	->
empty set	ϕ	<>
bottom	$\perp$	?
divide	$\div$	/
multiply	$\times$	*

2. Syntactic Modifications

2.1. While and Conditional

While and conditional functional expressions *must* be enclosed in parenthesis, *e.g.,*

(**while** *f g*)

(*p* -> *f*; *g*)

2.2. Function Definitions

Function definitions are enclosed by curly braces; they consist of a name-definition pair, separated by blanks. For example:

{**fact** !* @ iota}

defines the function **fact** (the reader should recognize this as the non-recursive factorial function).

2.3. Sequence Construction

It is not necessary to separate elements of a sequences with a comma; a blank will suffice:

<1,2,3> ≡ <1 2 3>

For nested sequences, the terminating right angle bracket acts as the delimiter:

<<1,2,3>,<4,5,6>> ≡ <<1 2 3><4 5 6>>

3. User Interface

We have provided a rich set of commands that allow the user to catalog, print, and delete functions, to load them from a file and to save them away. The user may generate script files, dynamically trace and measure functional expression execution, generate debugging output, and, temporarily exit to the FRANZ LISP system. A command must begin with a right parenthesis. Consult Appendix C for a complete description of the command syntax.

Debugging in FP is difficult; all undefined results map to a single atom — *bottom* ("?"). To pinpoint the cause of an error the user can use the special debugging output function, **out**, or the tracer.

4. Additions and Ommissions

Many relational functions have been added: $<$, $>$, $=$, $\neq$, $\leqslant$, $\geqslant$; their syntax is: $<$, $>$, $=$, $\tilde{}=$, $<=$, $>=$. Also added are the **iota** function (This is the APL iota function an n-element sequence of natural numbers) and the exclusive OR ($\oplus$) function.

Several new structural functions have been added: **pair** pairs up successive elements of a sequence, **split** splits a sequence into two (roughly) equal halves, **last** returns the last element of the sequence ($<>$ if the sequence is empty), **first** returns the first element of the sequence ($<>$ if it is empty), and **concat** concatenates all subsequences of a sequence, squeezing out null sequences ($<>$). **Front** is equivalent to **tlr**. **Pick** is a parameterized form of the selector function; the first component of the argument selects a single element from the second component. **Out** is the only side-effect function; it is equivalent to the **id** function but it also prints its argument out at the terminal. This function is intended to be used only for debugging.

One new functional form has been added, tree insert. This functional form breaks up the the argument into two roughly equal pieces applying itself recursively to the two halves. The functional parameter is applied to the result.

The binary-to-unary functions ('**bu**') has been omitted.

Seven mathematical library functions have been added: sin, cos, asin ($\sin^{-1}$), acos ($\cos^{-1}$), log, exp, and mod (the remainder function)

Appendix B: FP Grammar

I. BNF Syntax

fpInput →	(fnDef | application | fpCmd[a])* | '^D'
fnDef →	'{' name funForm '}'
application →	funForm ':' object
name →	letter (letter | digit | '_')*
nameList →	(name)*
object →	atom | fpSequence | '?'
fpSequence →	'<' (ϵ | object ((',' | ' ') object)*) '>'
atom →	'T' | 'F' | '<>' | '"' (ascii-char)* '"' | (letter | digit)* | number
funForm →	simpFn | composition | construction | conditional | constantFn | insertion | alpha | while | '(' funForm ')'
simpFn →	fpDefined | fpBuiltin
fpDefined →	name
fpBuiltin →	selectFn | 'tl' | 'id' | 'atom' | 'not' | 'eq' | relFn | 'null' | 'reverse' | 'distl' | 'distr' | 'length' | binaryFn | 'trans' | 'apndl' | 'apndr' | 'tlr' | 'rotl' | 'rotr' | 'iota' | 'pair' | 'split' | 'concat' | 'last' | 'libFn'
selectFn →	(ϵ | '+' | '-') unsignedInteger
relFn →	'<=' | '<' | '=' | '~=' | '>' | '>='
binaryFn →	'+' | '-' | '*' | '/' | 'or' | 'and' | 'xor'
libFn →	'sin' | 'cos' | 'asin' | 'acos' | 'log' | 'exp' | 'mod'
composition →	funForm '@' funForm
construction →	'[' formList ']'
formList →	ϵ | funForm (',' funForm)*
conditional →	'(' funForm '->' funForm ';' funForm ')'
constantFn →	'%' object
insertion →	'!' funForm | '/' funForm
alpha →	'&' funForm
while →	'(' 'while' funForm funForm ')'

II. Precedences

1.	%, !, &	(highest)
2.	@	
3.	[· · ·]	
4.	-> · · · ; · · ·	
5.	while	(least)

[a] Command Syntax is listed in Appendix C.

Appendix C: Command Syntax

All commands begin with a right parenthesis (")").

)fns
)pfn <nameList>
)load <UNIX file name>
)cload <UNIX file name>
)save <UNIX file name>
)csave <UNIX file name>
)fsave <UNIX file name>
)delete <nameList>
)stats on
)stats off
)stats reset
)stats print [UNIX file name]
)trace on <nameList>
)trace off <nameList>
)timer on
)timer off
)debug on
)debug off
)script open <UNIX file name>
)script close
)script append <UNIX file name>
)help
)lisp

Appendix D: Token-Name Correspondences

Token	Name
[	lbrack$$
]	rbrack$$
{	lbrace$$
}	rbrace$$
(	lparen$$
)	rparen$$
@	compos$$
!	insert$$
|	ti$$
&	alpha$$
;	semi$$
:	colon$$
,	comma$$
+	builtin$$
+ μ [a]	select$$
*	builtin$$
/	builtin$$
=	builtin$$
-	builtin$$
->	arrow$$
- μ	select$$
>	builtin$$
>=	builtin$$
<	builtin$$
<=	builtin$$
~=	builtin$$
%o [b]	constant$$

[a] μ is an optionally signed integer constant.

[b] o is any FP object.

Appendix E: Symbolic Primitive Function Names

The scanner assigns names to the alphabetic primitive functions by appending the string "$fp" to the end of the function name. The following table designates the naming assignments to the non-alphabetic primitive function names.

Function	Name
+	plus$fp
-	minus$fp
*	times$fp
/	div$fp
=	eq$fp
>	gt$fp
>=	ge$fp
<	lt$fp
<=	le$fp
~=	ne$fp

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is `-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple

---

UNIX is a Trademark of Bell Laboratories

lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines **N** to be 100, and uses this "symbolic constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true — **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N´)
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define´ = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N´, 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ´ are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N´)
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define´)
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix´, `define(wordsize,16)´ )
ifdef(`gcos´, `define(wordsize,36)´ )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix´, on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,  b  c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)´)
```

Then **N1** is defined as one more than the

current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^      (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!           (not)
& or &&     (logical and)
|or ||            (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2******N+1**. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(a, b, c, d)

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

define(compare, `ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr(`now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or −1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
    define(...)
    ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

errprint(`fatal error')

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get

everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name´, `name´, ...)
5 errprint(s, s, ...)
4 eval(numeric expression)
3 ifdef(`name´, this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name´)
4 undivert(number,number,...)

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Inc., 1976.

PART 3: SUPPORTING TOOLS

The seven articles in this part deal with ULTRIX-32 software tools that support program development and maintenance. The articles and the tools they describe range in sophistication from moderate to complex. *Lint*, for example, will be useful to anybody writing C programs, and the article is correspondingly simple. The articles on *yacc* and *lex*, on the other hand, assume a knowledge of the inner workings of compilers and a familiarity with compiler terminology.

Program and Library Maintenance Tools

Large scale software development projects involve manipulation of many programs. Any change in one file may require corresponding changes in many other files. ULTRIX-32 provides two tools that can help you automate portions of this development process: *awk* and *make*.

Awk lets you write programs to retrieve information and manipulate text in a set of files; this is a powerful tool. The article on *awk*, by Aho, Weinberger, and Kernighan, explains how to select text patterns and how to process the patterns selected. You can select patterns by:

- Literal text
- Relational expressions
- Pattern combinations
- Ranges of lines between patterns

Actions you can specify on the selected patterns include:

- Printing lines and fields from lines
- Formatting printed output
- Sending output to one or more files
- Piping output to other processes
- Determining string length
- Performing arithmetic functions
- Manipulating arrays

In addition, you can make your *awk* programs respond appropriately to diverse situations by using the flow control statements *if*, *else*, *while*, and *for*.

The *make* utility automates many activities related to program development and maintenance, and it saves processor time by ensuring that only required processing occurs. *Make* is useful when you compile a program made of many parts. When you make changes to one or more of the source files, you must recompile them to produce an up-to-date target file. If several people are involved in the development, keeping track of the changes could be a complex process without help from *make*.

"Make - A Program for Maintaining Computer Programs," by Feldman, tells how to use the *make* utility and gives helpful examples. You supply a description file to *make* showing relationships of source files to target files. If one of the source files has changed, *make* will recompile it, because the corresponding object file will be obsolete. If a source file has not changed, the corresponding object file will still be valid, and *make* will not recompile it. Useful *make* features include:

- Macros
- Command line options
- A set of implicit rules that supply dependency information

The source control code system (*sccs*) is a file maintenance utility; it is not available in the first ULTRIX-32 implementation. The article on *sccs*, by Allman, tells how you can use *sccs* to:

- Create a library of source files
- Store multiple versions of files
- Remove a stored file for editing
- Replace a file after editing
- Get a copy of a stored file for printing or compiling
- Get old versions of a stored file for editing or copying
- Comment on each version of a file
- Maintain separate variations of a file in parallel
- Coordinate the *make* utility with *sccs* to extract appropriate source files from the library automatically

Sccs protects files from conflicting access when they have been removed for editing. Only the person who has removed the file for editing can make changes. In addition, the software lets you coordinate groups of files according to versions, so that all files associated with a particular stage of a project may be accessed as a set.

Checking and Debugging Programs

ULTRIX-32 offers two tools for program checking and debugging. *Lint* detects programming errors in C programs. *Adb* is a versatile debugger that lets you look at programs as they run and look at programs that have crashed.

The first article, "Lint, a C Program Checker," by Johnson, explains command line options that allow you to specify features you want checked or ignored. Johnson also lists *lint* limitations. *Lint* checks for:

- Correspondence between combinations of programs
- Inefficiencies such as unused variables
- Problems that may interfere with portability
- Legal but strange constructions

The "Tutorial Introduction to adb," by Maranzano and Bourne, gives extensive examples that include sample C programs, corresponding *adb* debugging sessions, and detailed explanations. *Adb* is useful for debugging all kinds of programs, but it is especially appropriate for debugging system programs, because it opens a window on the memory (core) image of a program. Using *adb* you can:

- Set breakpoints
- Single step through the program from any point
- Examine memory at any point
- Choose the format for dumping a core image
- Use a script to produce a dump
- Convert numbers from one base to another using octal, decimal, and hexadecimal
- Change values and instructions

Compiler Writing Tools

Two articles in this part deal with compiler development tools: *yacc* and *lex*. Both articles are addressed to sophisticated users.

Yacc is a utility that allows you to build parsing routines. The output will be a C source file; and it may be used as part of a compiler. *Yacc* was used in the development of some other utilities on the ULTRIX-32 system including *lint*, the portable C compiler, and *troff*. The article "Yacc: Yet Another Compiler-Compiler," by Johnson, explains how to use *yacc* to:

- Specify grammar rules
- Specify actions to be associated with the rules
- Prepare lexical analyzers
- Detect and recover from errors

"Lex - A Lexical Analyzer Generator," by Lesk, tells how to use *lex* to produce lexical analyzing routines. The output of *lex* is a C program that separates an input stream of characters into strings that match given general expressions. When the lexical analyzer recognizes a string, it executes a sequence of instructions.

The lexical analyzer you produce with *lex* can be designed to cooperate with a parser produced using *yacc*. In this case, the lexical analyzer will identify words and strings; and the parser will identify structures built with the words and strings identified by the lexical analyzer. When they are used together, the parser and the lexical analyzer you create with *yacc* and *lex* can form the front end of a compiler.

Awk — A Pattern Scanning and Processing Language

(Second Edition)

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep*[1] will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev    { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk program [files]
```

executes the *awk* commands in the string **program** on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

```
awk -f pfile [files]
```

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern   { action }
pattern   { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as

† UNIX is a trademark of Bell Laboratories.

described below. Fields are referred to as **$1, $2,** and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument **−F***c* may also be used to set **FS** to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, **$1**, on the file **foo1**, and the second field on file **foo2**. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to **bwk**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

Awk also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints **$1** as a floating point number 8 digits wide, with two after the decimal point, and **$2** as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C.[2]

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN  { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

END { print NR }

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

/smith/

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

blacksmithing

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*[1] and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and **?** for "zero or one", all as in *lex*. Character classes may be abbreviated: **[a–zA–Z0–9]** is the set of all letters and digits. As an example, the *awk* program

/[Aa]ho|[Ww]einberger|[Kk]ernighan/

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

/\/.*\//

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

$1 ~ /[jJ]ohn/

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

$1 ~ /^[jJ]ohn$/

The caret ^ refers to the beginning of a line or field; the dollar sign **$** refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

$2 > $1 + 100

which selects lines where the second field is at least 100 greater than the first field. Similarly,

NF % 2 == 0

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

$1 >= "s"

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

$1 > $2

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), **&&** (and), and **!** (not). For example,

$1 >= "s" && $1 < "t" && $1 != "smith"

selects lines where the first field begins with "s", but is not "smith". **&&** and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

pat1, pat2 { ... }

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

/start/, /stop/

prints all lines between **start** and **stop**, while

NR == 100, NR == 200 { ... }

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipu-

lating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

```
      { s1 += $1; s2 += $2 }
END   { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). The C increment ++ and decrement −− operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
    $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string **s** into **array[1]**, ..., **array[n]**. The number of elements found is

returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the **NR**-th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
      { x[NR] = $0 }
END   { ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/   { x["apple"]++ }
/orange/  { x["orange"]++ }
END       { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like <, <=, >, >=, == ("is equal to"), and **!=** ("not equal to"); regular expression matches with the match operators ~ and **!~**; the logical operators **||**, **&&**, and **!**; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y     # this is a comment
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with

a particularly fast algorithm. *Sed*[1] provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;[4] the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls −l*; each line has the form

−rw−rw−rw− 1 ava 123 Oct 15 17:05 xxx

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. Sixth Edition

2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

4. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

Program	Task 1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. `END {print NR}`

2. `/doug/`

3. `/ken|doug|dmr/`

4. `{print $3}`

5. `{print $3, $2}`

6.
```
/ken/    {print >"jken"}
/doug/   {print >"jdoug"}
/dmr/    {print >"jdmr"}
```

7. `{print NR ": " $0}`

8.
```
     {sum = sum + $4}
END  {print sum}
```

SED:

1. `$=`

2. `/doug/p`

3.
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4. `/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p`

5. `/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p`

6.
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1.
```
%{
int i;
%}
%%
\n    i++;
.     ;
%%
yywrap() {
     printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$      printf("%s\n", yytext);
.     ;
\n    ;
```

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that

the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
     cc x.o y.o z.o  -lS  -o prog

x.o y.o :  defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc −c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog : x.o y.o z.o
     cc x.o y.o z.o -lS  -o prog
x.o : x.c defs
     cc -c x.c
y.o : y.c defs
     cc -c y.c
z.o : z.c
     cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed.

This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
      cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex ("-ll") and the Standard ("-lS") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has

† UNIX is a trademark of Bell Laboratories.

the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[*(tab)* commands] [# . . .]
. . .

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "−i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "−" denotes the standard input. If there are no "−f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

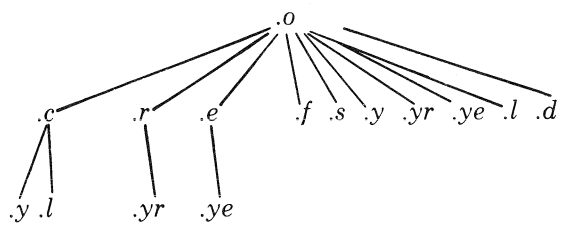
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS= -O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command
P = und -3 | opr -r2        # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O
make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make
$(OBJECTS):  defs
gram.o: lex.c
cleanup:
        -rm *.o gram.c
        -du
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make
print:  $(FILES)        # print recently changed files
        pr $? | $P
        touch print
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
lint :  dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  -c version.c
cc  -c main.c
cc  -c doname.c
cc  -c misc.c
cc  -c files.c
cc  -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was

suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
      or
make print "P=  cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "−n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "−t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("−d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, "Lex — A Lexical Analyzer Generator", Computing Science Technical Report #39, October 1975.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "−r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

An Introduction to the Source Code Control System

Eric Allman
Project Ingres
University of California at Berkeley

This document gives a quick introduction to using the Source Code Control System (SCCS). The presentation is geared to programmers who are more concerned with what to do to get a task done rather than how it works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section on "Further Information".

> This is a working document. Please send any comments or suggestions to csvax:eric.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file". There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes – equivalent to removing your changes later.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form %*x*%, where *x* is an upper case letter. For example, %**I**% is the SID of the latest delta applied, %**W**% includes the module name, SID, and a mark that makes it findable by a program, and %**G**% is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is "%W%" or whatever).

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin -i$i $i
        mv $i save/$i
end
```

This will put the named files into s-files in the subdirectory "SCCS" The files will be removed from the current directory and hidden away in the directory "save", so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory "save".

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print "No Id Keywords (cm7)", which is a warning message only.

---

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for "comments?" before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

---

[2]Actually, the SID of the final delta applied was 1.1.

[3]The "edit" command is equivalent to using the −e flag to *get*, as:

```
sccs get −e prog.c
```

Keep this in mind when reading other documentation.

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an "install" entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **−b** flag to ignore "branches" (alternate versions, described later) and the **−u** flag to only give files being edited by you. The **−u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info −ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c     1.2       08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with "@(#)". This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c  1.2     08/29/80
/usr/bin/prog:
        prog.c  1.1     02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%      %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W%      %G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because "SccsId" is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID's consistent across files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit -r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit -r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get -r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the **-c** (cutoff) flag. For example,

```
sccs get -c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get -c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit -x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit -x1.3-1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit -x1.3-1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using **-x** (or **-i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of "a set of changes") can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prt command

When you created a delta, you presumably gave a reason for the delta to the "comments?" prompt. To print out these comments later, use:

```
sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2   80/08/29 12:35:31      bill    2       1       00005/00003/00084
removed "-q" option
D 1.1   79/02/05 00:19:31      eric    1       0       00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get -m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get -m -p prog.c c grep '^1.3'
```

The **-p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the "diff" flags can be used. To pass the **-c** flag, use **-C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The "deledit" command is equivalent to "delget" except that the "edit" command is used instead of the "get" command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix -r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The -r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The −d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs −d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file "/usr/src/cmd/SCCS/who.c". The file "who.c" will always be created in your current directory regardless of the value of the −d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[6]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get −k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin −z prog.c
```

---

[6]Or given up and decided to start over.

11. Using the Admin Command

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the **−f** flag. For example:

```
sccs admin −fd1 prog.c
```

sets the "d" flag to the value "1". This flag can be deleted by using:

```
sccs admin −dd prog.c
```

The most useful flags are:

b Allow branches to be made using the −b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.

y The "type" of the module. Actually, the value of this flag is unused by SCCS except that it replaces the %**Y**% keyword.

The **−t***file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **−t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use "prt −t".

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

The ability to create branches must be enabled in advance using:

```
sccs admin −fb prog.c
```

The **−fb** flag can be specified when the SCCS file is first created.

12.1. Creating a branch

To create a branch, use:

```
sccs edit −b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get −r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
sccs edit -i1.5.1.1-1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs -d../xyz edit prog.c
```

When using the old version, be sure to use the **-b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info -b
```

when in the directory "xyz".

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the **-i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all cruft from the directory.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= -i -s

prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h

example: example.o
	$(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
	sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS=prog.c prog.h example.c

LDFLAGS= -i -s

prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h

example: example.o
	$(CC) $(LDFLAGS) -o example example.o

sources: $(SRCS)
$(SRCS):
	sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say "make sources", and (3) the makefile is less likely to do confusing things since it won't try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.c d.s x.h y.h z.h
TARG=       /usr/lib

# programs
GET= sccs get
REL=
AR=   -ar
RANLIB=     ranlib

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a

install: lib.a
        sccs check
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@

print: sources
        pr *.h *.[cs]
clean:
        rm -f *.o
        rm -f core a.out $(LIB)
```

The "$(REL)" in the get can be used to get old versions easily; for example:

```
make b.o REL=-r1.3
```

The *install* entry includes the line "sccs check" before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.y d.s x.h y.h z.h

GET= sccs get
REL=

a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with "sccs". This list is not exhaustive; for more options see *Further Information.*

get — Gets files for compilation (not for editing). Id keywords are expanded.

- −r*SID* Version to get.
- −p Send to standard output rather than to the actual file.
- −k Don't expand id keywords.
- −i*list* List of deltas to include.
- −x*list* List of deltas to exclude.
- −m Precede each line with SID of creating delta.
- −c*date* Don't apply any deltas created after *date.*

edit — Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

- −r*SID* Same as *get.* If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID.*
- −b Create a branch.
- −i*list* Same as *get.*
- −x*list* Same as *get.*

delta — Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.

unedit — Remove a file that has been edited previously without merging the changes into the s-file.

prt — Produce a report of changes.

- −t Print the descriptive text.
- −e Print (nearly) everything.

info — Give a list of all files being edited.

- −b Ignore branches.
- −u[*user*] Ignore files not being edited by *user.*

check — Same as *info*, except that nothing is printed if nothing is being edited and exit status is returned.

tell — Same as *info*, except that one line is produced per file being edited containing only the file name.

clean — Remove all files that can be regenerated from the s-file.

what — Find and print id keywords.

admin — Create or set parameters on s-files.

- −i*file* Create, using *file* as the initial contents.
- −z Rebuild the checksum in case the file has been trashed.

–f*flag* Turn on the *flag*.

–d*flag* Turn off (delete) the *flag*.

–t*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.

Useful flags are:

b Allow branches to be made using the –b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*.

i Cause "No Id Keywords" error message to be a fatal error rather than a warning.

t The module "type"; the value of this flag replaces the %**Y**% keyword.

fix Remove a delta and reedit it.

delget Do a *delta* followed by a *get*.

deledit Do a *delta* followed by an *edit*.

2. Id Keywords

%Z% Expands to "@(#)" for the *what* command to find.

%M%
The current module name, *e.g.*, "prog.c".

%I% The highest SID applied.

%W%
A shorthand for "%Z%%M% <tab> %I%".

%G% The date of the delta corresponding to the "%I%" keyword.

%R% The current release number, *i.e.*, the first component of the "%I%" keyword.

%Y% Replaced by the value of the **t** flag (set by *admin*).

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction and Usage

Suppose there are two C[1] source files, *file1*.c and *file2*.c, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit extern statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the −x flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The −v option is available to suppress the printing of complaints about unused arguments. When −v is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The −u flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The −O flag in the C

compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the −b option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in "working" programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (? :), and relational operators have this property; the argument of a return statement, and expressions used in initialization also suffer similar conversions. In these operations, char, short, int, long, unsigned, float, and double types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the −> be a pointer to structure, the left operand of the . be a structure, and the right operand of

these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar()) < 0 ) ....
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use typedefs. When a typedef variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **int**s, losing accuracy. Since there are a number of legitimate reasons for assigning **long**s to **int**s, the detection of these assignments is enabled by the **-a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The −h flag is used to enable these checks. For example, in the statement

```
*p++;
```

the * does nothing; this provokes the message "null effect" from *lint.* The program fragment

```
unsigned x;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the −h flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =−, . . .) could cause ambiguous expressions, such as

```
a =-1;
```

which could be taken as either

```
a =- 1;
```

or

```
a = -1;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, −=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned

operators.

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

```
int x (-1);
```

looks somewhat like the beginning of a function declaration:

```
int x (y) { ...
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **-p** or **-h** flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will draw the complaint:

```
warning: i evaluation order undefined
```

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[4, 5] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file

which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the −p flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* −p causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

†UNIX is a Trademark of Bell Laboratories.

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed unsigned. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The −v flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the typedef is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are

pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.* 57(6) pp. 2021-2048 (1978).
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).

Appendix: Current Lint Options

The command currently has the form

lint [−options] files... library-descriptors...

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable **break** statements.

x Report unused external declarations

a Report assignments of **long** to **int** or shorter.

c Complain about questionable casts

n No library checking is done

s Same as **h** (for historical reasons)

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX† with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (−) means ignore this argument as in:

adb − core

ADB has requests for examining locations in either file. The ? request examines the contents of *objfile*, the / request examines the *corefile*. The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

†UNIX is a Trademark of Bell Laboratories.

sets dot to octal 126 and prints the instruction at that address. The request:

```
.,10/d
```

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, −, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name:* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, 'D' for long decimal, and 'F' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	Meaning
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**
;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request Sq or SQ (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp*. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable: 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by ? whereas the map for *core* file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp*. This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main*.

The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main*. Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.-10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f*, *g*, and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f*, *g*, and *h*. Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f*. Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h*, type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with $C or the occurrence of a variable in the most recent call of a function. It is possible with the $C request, however, to print the stack frame starting at some address as **address$C**.

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
```

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count* -1 times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

```
settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the ? command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen*), ADB requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8o
```

to print three lines of 8 locations each from the array called *tabs*. By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs*.

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

```
:c
```

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

```
ibuf+6/20c
```

When we continue the program with:

```
:c
```

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

```
tabpos+4:d
```

If the program is continued with:

```
:c
```

it resumes normal execution after ADB prints the message

```
a.out:running
```

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b settab,5?ia *
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only

---

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

```
settab+4:b      settab,5?ia;0
getc+4,3:b      main.c?C;0
settab+4:b      settab,5?ia; ptab/o;0
```

Note that ;0 will set dot to zero and stop at the breakpoint.

stop after the third occurrence by typing:

```
getc+4,3:b  main.c?C
```

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b  .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia; ptab/o
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
$b
```

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f*, *g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
adb ex3 -
```

Suppose we enter the following breakpoints:

```
h+4:b     hcnt/d; h.hi/; h.hr/
g+4:b     gcnt/d; g.gi/; g.gr/
f+4:b     fcnt/d; f.fi/; f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b     fcnt/d; f.a/; f.b/; f.fi/
g+4:b     gcnt/d; g.p/; g.q/; g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b      fcnt/d; f.a/"a="d; f.b/"b="d; f.fi/"fi="d
```

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  ```
  :r arg1 arg2 ... <infile >outfile
  ```

  This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

  ```
  :s
  ```

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

  ```
  address:r
  ```

- The count field can be used to skip the first *n* breakpoints as:

  ```
  ,n:r
  ```

  The request:

  ```
  ,n:c
  ```

  may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  ```
  address:c
  ```

- The program being debugged runs as a separate process and can be killed by:

  ```
  :k
  ```

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

```
$m
```

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution

of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the ?* operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

$$b1 \leqslant A \leqslant e1 \Rightarrow \text{file address} = (A - b1) + f1$$
$$b2 \leqslant A \leqslant e2 \Rightarrow \text{file address} = (A - b2) + f2$$

A user can access locations by using the ADB defined variables. The $v request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

```
<b
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets b to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.

<b,-1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, *dump*, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero

thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the 8t means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072        58          #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a          0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write*, w or W, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate*, l or L request. In general, the request syntax for l and w are similar as follows:

```
?l value
```

The request l is used to match on two bytes, L is used for four bytes. The request w is used to write two bytes, whereas W writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request ?l starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of ? to write to the *a.out* file. The form ?* would have been used for a 411 file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the w request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if ? is used for text (instructions) and / for data.

3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System." CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;

        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = 'T';
printf("debug 1 %s\n",charp);
        while(cc= *charp++)
                putc(cc,obuf);
        fflush(obuf);
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
_main(02,0177762)
$C
_main(02,0177762)
        argc:       02
        argv:       0177762
        cc:         02124
$r
ps      0170010
pc      0204    _main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
_main+0152:     mov     _obuf,(sp)
$e
savr5:      0
_obuf:      0
_charp:     0124
_errno:     0
_fout:      0
$m
text map    'ex1'
b1 = 0              e1   = 02360            f1 = 020
b2 = 0              e2   = 02360            f2 = 020
data map    'core1'
b1 = 0              e1   = 03500            f1 = 02000
b2 = 0175400        e2   = 0200000          f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx      Nh@x&_
=
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756:        2
*main.argv/3o
0177762:        0177770 0177776 0177777
0177770/s
0177770:        a.out
*main.argv/3o
0177762:        0177770 0177776 0177777
*=/s
0177770:        a.out
 .=o
                0177770
 .-10/d
0177756:        2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
*h(04452,04451)
*g(04453,011124)
*f(02,04451)
*h(04450,04447)
*g(04451,011120)
*f(02,04447)
*h(04446,04445)
*g(04447,011114)
*f(02,04445)
*h(04444,04443)
HIT DEL KEY
adb
.,5$C
*h(04452,04451)
        x:          04452
        y:          04451
        hi:         ?
*g(04453,011124)
        p:          04453
        q:          011124
        gi:         04451
        gr:         ?
*f(02,04451)
        a:          02
        b:          04451
        fi:         011124
        fr:         04453
*h(04450,04447)
        x:          04450
        y:          04447
        hi:         04451
        hr:         02
*g(04451,011120)
        p:          04451
        q:          011120
        gi:         04447
        gr:         04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE     80
#define YES         1
#define NO          0
#define TABSP       8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++;
                                }
                                break;
                        case '\n':/*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count    bkpt              command
1        _tabpos+04
1        _getc+04
1        _fopen+04
1        _settab+04
settab,5?ia
_settab:        jsr     r5,csv
_settab+04:     tst     -(sp)
_settab+06:     clr     0177770(r5)
_settab+012:    cmp     $0120,0177770(r5)
_settab+020:    bit     _settab+076
_settab+022:
settab,5?i
_settab:        jsr     r5,csv
                tst     -(sp)
                clr     0177770(r5)
                cmp     $0120,0177770(r5)
                bit     _settab+076
:r
a.out: running
breakpoint      _settab+04:     tst     -(sp)
settab+4:d
:c
a.out: running
breakpoint      _fopen+04:      mov     04(r5),nulstr+012
$C
_fopen(02302,02472)
_main(01,0177770)
        col:    01
        c:      0
        ptab:   03500
tabs,3/8o
03500:          01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

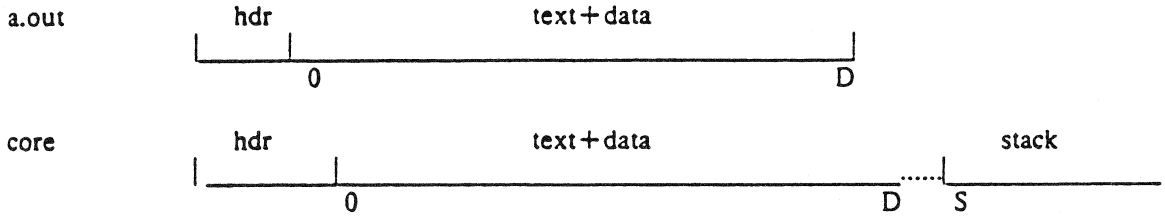
```
:c
a.out: running
breakpoint      _getc+04:       mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:          This    is      a test  of
:c
a.out: running
breakpoint      ~tabpos+04:     cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt            command
1       ~tabpos+04
3       _getc+04        main.c?C;0
1       _fopen+04
1       ~settab+04      settab,5?ia;ptab?o;0
~settab:        jsr     r5,csv
~settab+04:     bpt
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
0177766:        0177770
0177744:        @`
T0177744:       T
h0177744:       h
i0177744:       i
s0177744:       s
```

Figure 7: ADB output for C program with breakpoints

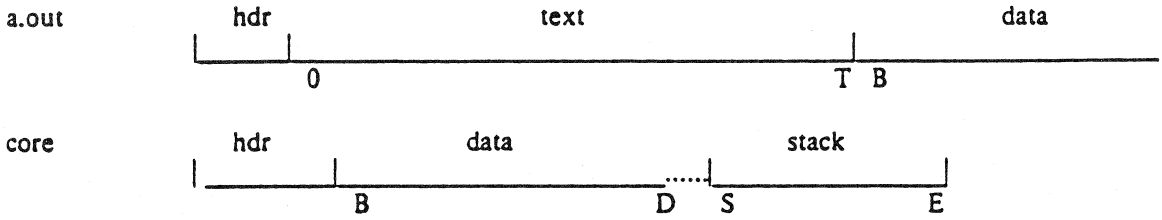
```
adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:          0
0177732:        214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:          0
0177746:        1
0177750:        1
0177732:        214
_gcnt:          0
0177726:        2
0177730:        3
0177712:        214
_hcnt:          0
0177706:        2
0177710:        1
0177672:        214
_fcnt:          1
0177666:        2
0177670:        3
0177652:        214
_gcnt:          1
0177646:        5
0177650:        8
0177632:        214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:          0
0177746:        a = 1
0177750:        b = 1
0177732:        fi = 214
_gcnt:          0
0177726:        p = 2
0177730:        q = 3
0177712:        gi = 214
_hcnt:          0
0177706:        x = 2
0177710:        y = 1
0177672:        hi = 214
_fcnt:          1
0177666:        a = 2
0177670:        b = 3
0177652:        fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

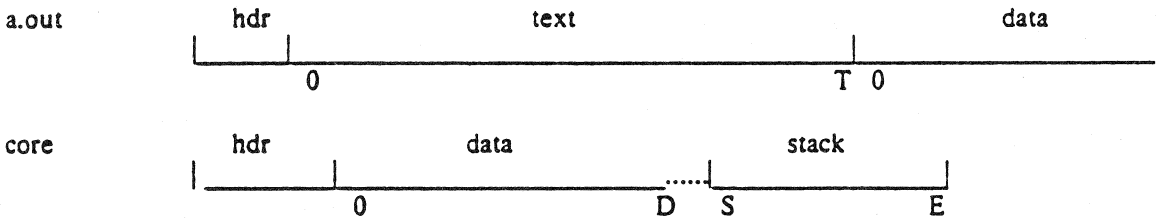
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D−B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    map407'
b1 = 0              e1     = 0256          f1 = 020
b2 = 0              e2     = 0256          f2 = 020
data map    core407'
b1 = 0              e1     = 0300          f1 = 02000
b2 = 0175400        e2     = 0200000       f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410'
b1 = 0              e1     = 0200          f1 = 020
b2 = 020000         e2     = 020116   f2 = 0220
data map    `core410'
b1 = 020000         e1     = 020200   f1 = 02000
b2 = 0175400        e2     = 0200000       f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411'
b1 = 0              e1     = 0200          f1 = 020
b2 = 0              e2     = 0116          f2 = 0220
data map    `core411'
b1 = 0              e1     = 0200          f1 = 02000
b2 = 0175400        e2     = 0200000       f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]   "This is a character string";
int     one      1;
int     number 456;
long    lnum     1234;
float   fpt      1.25;
char    str2[]   "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,-1/8ona
020000:          0     064124     071551     064440     020163     020141     064143     071141
_str1+016: 061541      062564     020162     072163     064562     063556     0     02
_number:
_number:   0710 0      02322040240      0     064124     071551     064440
_str2+06:  020163      064164     020145     062563     067543     062156     061440     060550
_str2+026: 060562      072143     071145     071440     071164     067151     0147 0
savr5+02:  0     0     0     0     0     0     0     0
<b,20/4o4^8Cn
020000:          0     064124     071551     064440     @ @This i
           020163      020141     064143     071141     s a char
           061541      062564     020162     072163     acter st
           064562      063556     0     02   ring@'@'@b@
_number:   0710 0      02322040240      H@a@'@'R@d @@
           0     064124     071551     064440     @'@'This i
           020163      064164     020145     062563     s the se
           067543      062156     061440     060550     cond cha
           060562      072143     071145     071440     racter s
           071164      067151     0147 0     tring@'@'@'
           0     0     0     0     @'@'@'@'@'@'@'@'
           0     0     0     0     @'@'@'@'@'@'@'@'
data address not found
<b,20/4o4^8t8cna
020000:          0     064124     071551     064440          This i
_str1+06:  020163      020141     064143     071141          s a char
_str1+016: 061541      062564     020162     072163          acter st
_str1+026: 064562      063556     0     02          ring
_number:
_number:   0710 0      02322040240          HR
_fpt+02:   0     064124     071551     064440          This i
_str2+06:  020163      064164     020145     062563          s the se
_str2+016: 067543      062156     061440     060550          cond cha
_str2+026: 060562      072143     071145     071440          racter s
_str2+036: 071164      067151     0147 0          tring
savr5+02:  0     0     0     0
savr5+012: 0     0     0     0
data address not found
<b,10/2b8t^2cn
020000:          0     0
_str1:     0124 0150     Th
           0151 0163     is
           040  0151      i
           0163 040      s
           0141 040      a
           0143 0150     ch
           0141 0162     ar
           0141 0143     ac
           0164 0145     te
SQ
```

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?ut14cn

          Inode      Name
0:        652  .
          82   ..
          5971 cap.c
          5323 cap
          0    pp



adb /dev/src -
02000>b
?m<b
new map     '/dev/src'
b1 = 02000        e1     = 0100000000   f1 = 0
b2 = 0            e2     = 0            f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:          flags 073145
          links,uid,gid     0163 0164 0141
          size  0162 10356
          addr  28770     8236 25956     27766     25455     8236 25956     25206
          times 1976 Feb 5 08:34:56  1975 Dec 28 10:55:15

02040:          flags 024555
          links,uid,gid     012  0163 0164
          size  0162 25461
          addr  8308 30050     8294 25130     15216     26890     29806     10784
          times 1976 Aug 17 12:16:51 1976 Aug 17 12:16:51

02100:          flags 05173
          links,uid,gid     011  0162 0145
          size  0147 29545
          addr  25972     8306 28265     8308 25642     15216     2314 25970
          times 1977 Apr 2 08:58:01  1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a). formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

> *name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
***n*t**	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
-	subtract
*****	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*****	contents of location
-	integer negate

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

---

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

```
month_name : 'J' 'a' 'n'  ;
month_name : 'F' 'e' 'b'  ;

    . . .

month_name : 'D' 'e' 'c'  ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year  ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

---

[2] A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys*, vol. 6, no. 2, pp. 99-124, June 1974.

[3] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 441-452, August 1975.

[4] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

[5] S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3

[6] S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104, January 1978.

[7] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp.*

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

---

Mach., vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975.

'\n'	newline
'\r'	return
'\''	single quote "'"
'\\'	backslash "\"
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A   :   B C D   ;
A   :   E F   ;
A   :   G   ;
```

can be given to Yacc as

```
A   :   B C D
    |   E F
    |   G
    ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1   name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A    :    '(' B ')'
              {     hello( 1, "abc" ); }
```

and

```
XXX      :    YYY ZZZ
              {     printf("a message\n");
                    flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A    :    B C D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr :    '(' expr ')'  ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr :    '(' expr ')'          { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A    :    B  ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A    :    B
               { $$ = 1; }
          C
               {  x = $2;  y = $3; }
     ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT     :     /* empty */
               { $$ = 1; }
     ;

A    :     B $ACT C
               {  x = $2;  y = $3; }
     ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr :     expr '+' expr
               { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has

been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

---

[8] M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF    shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.     reduce 18
```

refers to *grammar rule* 18, while the action

```
IF    shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A   :   x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of

a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A    goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme    :    sound place
         ;
sound    :    DING DONG
         ;
place:   DELL
         ;
```

When Yacc is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_ (1)

        . reduce 1

state 5
        place : DELL_ (3)

        . reduce 3

state 6
        sound : DING DONG_ (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next

token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound    goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr :    expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat :    IF '(' cond ')' stat
     |    IF '(' cond ')' stat ELSE stat
     ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule

will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
    IF ( C2 ) S1
    }
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
    }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (**−v**) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_        (18)
        stat : IF ( cond ) stat_ELSE stat

      ELSE    shift 45
              reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[9,10,11] might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

---

[9] A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys*, vol. 6, no. 2, pp. 99-124, June 1974.

[10] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 441-452, August 1975.

[11] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr :    expr '=' expr
     |    expr '+' expr
     |    expr '-' expr
     |    expr '*' expr
     |    expr '/' expr
     |    NAME
     ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr :    expr '+' expr
      |   expr '-' expr
      |   expr '*' expr
      |   expr '/' expr
      |   '-' expr      %prec '*'
      |   NAME
      ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery

might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat :    error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input:    error '\n' { printf( "Reenter last line: " ); } input
               {     $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input:    error '\n'
               {     yyerrok;
                     printf( "Reenter last line: " );  }
          input
               {     $$ = $4; }
     ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat :    error
              {    resynch();
                   yyerrok ;
                   yyclearin ;  }
     ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse* ; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **−ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
     return( yyparse() );
     }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
     fprintf( stderr, "%s\n", s );
     }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name:     name  rest of rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list  :     item
      |     list  ','  item
      ;
```

and

```
seq   :     item
      |     seq  item
      ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq   :     item
      |     item  seq
      ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq  :     /* empty */
     |     seq  item
     ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
     int dflag;
%}
 ... other declarations ...

%%

prog :     decls  stats
     ;

decls:     /* empty */
                { dflag = 1; }
     |     decls  declaration
     ;

stats :    /* empty */
                { dflag = 0; }
     |     stats  statement
     ;

  ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent :    adj noun verb adj noun
               { look at the sentence . . . }
     ;

adj  :    THE      {    $$ = THE; }
     |    YOUNG    {    $$ = YOUNG; }
     . . .
     ;

noun :    DOG
               {    $$ = DOG; }
     |    CRONE
               {    if( $0 == YOUNG ){
                         printf( "what?\n" );
                         }
                    $$ = CRONE;
                    }
     ;
     . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[12] will be far more silent.

---

[12] S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. updated version TM 78-1273-3

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
    body of union ...
    }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
    body of union ...
    } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '−'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 − see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule :    aaa { $<intval>$ = 3; } bbb
              {    fun( $<intval>2, $<other>0 ); }
    ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS      /* supplies precedence for unary minus */

%%     /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
```

```
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
          /* returns LETTER for a lower case letter, yylval = 0 through 25 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers  and literals */
%token  C_IDENTIFIER    /*   identifier (but not literal) followed by colon   */
%token  NUMBER                  /*   [0-9]+   */

        /* reserved words:   %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
```

```
        |       NONASSOC
        |       TYPE
        ,

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
      double lo, hi;
      } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start   lines

%union   {
      int ival;
      double dval;
      INTERVAL vval;
      }

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST       /* floating point constant */

%type <dval> dexp      /* expression */

%type <vval> vexp      /* interval expression */

      /* precedence information about the operators */

%left      '+' '-'
%left      '*' '/'
%left      UMINUS      /* precedence for unary minus */

%%

lines :     /* empty */
      |     lines line
      ;

line  :     dexp '\n'
                  {     printf( "%15.8f\n", $1 ); }
      |     vexp '\n'
                  {     printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
      |     DREG '=' dexp '\n'
                  {     dreg[$1] = $3; }
      |     VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        |   error '\n'
                { yyerrok; }
        ;

dexp :  CONST
        |   DREG
                { $$ = dreg[$1]; }
        |   dexp '+' dexp
                { $$ = $1 + $3; }
        |   dexp '-' dexp
                { $$ = $1 - $3; }
        |   dexp '*' dexp
                { $$ = $1 * $3; }
        |   dexp '/' dexp
                { $$ = $1 / $3; }
        |   '-' dexp %prec UMINUS
                { $$ = - $2; }
        |   '(' dexp ')'
                { $$ = $2; }
        ;

vexp :  dexp
                { $$.hi = $$.lo = $1; }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                    printf( "interval out of order\n" );
                    YYERROR;
                    }
                }
        |   VREG
                { $$ = vreg[$1]; }
        |   vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        |   dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        |   vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        |   dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        |   vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        |   dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        |   vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
        |   dexp '/' vexp
```

```
            {   if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1, $1, $3 ); }
    |   '-' vexp   %prec UMINUS
            {   $$.hi = -$2.lo;   $$.lo = -$2.hi;   }
    |   '(' vexp ')'
            {   $$ = $2; }
    ;

%%

# define BSZ 50      /* buffer size for floating point numbers */

    /* lexical analysis */

yylex(){
    register c;

    while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

    if( isupper( c ) ){
        yylval.ival = c - 'A';
        return( VREG );
        }
    if( islower( c ) ){
        yylval.ival = c - 'a';
        return( DREG );
        }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if( isdigit( c ) ) continue;
            if( c == '.' ){
                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                continue;
                }

            if( c == 'e' ){
                if( exp++ ) return( 'e' );   /* will cause syntax error */
                continue;
                }

            /* end of number */
            break;
            }
        *cp = '\0';
        if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
        else ungetc( c, stdin );  /* push back last char read */
        yylval.dval = atof( buf );
```

```
        return( CONST );
        }
    return( c );
    }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a;   v.lo = b; }
    else { v.hi = b;   v.lo = a; }

    if( c>d ) {
        if( c>v.hi ) v.hi = c;
        if( d<v.lo ) v.lo = d;
        }
    else {
        if( d>v.hi ) v.hi = d;
        if( c<v.lo ) v.lo = c;
        }
    return( v );
    }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
    }

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return( 1 );
        }
    return( 0 );
    }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
    }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or_, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt
Bell Laboratories
Murray Hill, New Jersey 07974

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

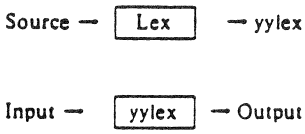
Table of Contents

1. Introduction. 1
2. Lex Source. 3
3. Lex Regular Expressions. 3
4. Lex Actions. 5
5. Ambiguous Source Rules. 7
6. Lex Source Definitions. 8
7. Usage. 8
8. Lex and Yacc. 9
9. Examples. 10
10. Left Context Sensitivity. 11
11. Character Set. 12
12. Summary of Source Format. 12
13. Caveats and Bugs. 13
14. Acknowledgments. 13
15. References. 13

1 Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to



An overview of Lex

Figure 1

write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$    ;
```

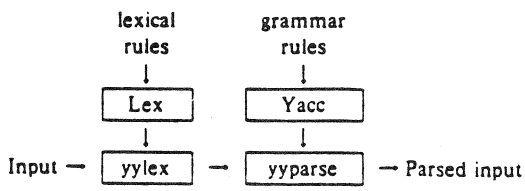
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time



Lex with Yacc

Figure 2

taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch (in C) or branches of a computed GOTO (in Ratfor). The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd*. . . Such backup is more costly than the processing of simpler languages.

2 Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second **%%** is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described later.

3 Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above

expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction *[ab]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ − and ^. The − character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using − between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character − in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

is one or more instances of *a*. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for the next rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+    {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

in C or

```
yytext(yyleng)
```

in Ratfor.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng-1] == '\\')
               yymore();
           else
               ... normal user processing
           }
```

which will, when faced with a string such as "*abc*\"*def*" first match the five characters "*abc*\; then the call to *yymore()* will cause the next part of the string, "*def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "= -a" but print a message. A rule might be

```
=-[a-zA-Z]    {
              printf("Operator (=-) ambiguous\n");
              yyless(yyleng-1);
              ... action for =- ...
              }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]    {
              printf("Operator (=-) ambiguous\n");
              yyless(yyleng-2);
              ... action for = ...
              }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. There is another important routine in Ratfor, named *lexshf*, which is described below under "Character Set". These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, *input*,

output, ***unput***, ***yywrap***, and ***lexshf***, are defined as integer functions. This requires ***input*** and ***yywrap*** to be called with arguments. One dummy argument is supplied and ignored.

5 Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

to be given in that order. If the input is ***integers***, it is taken as an identifier, because *[a-z]+* matches 8 characters while ***integer*** matches only 7. If the input is ***integer***, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. ***int***) will not match the expression ***integer*** and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like *[.\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally ***not*** recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+     { ... ; REJECT;}
a[cd]+     { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string ***accb*** matches the first rule for four characters and then the second rule for three characters. In contrast, the input ***accd*** agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a-z][a-z]     {digram[yytext[0]][yytext[1]]++; REJECT;}
\n             ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [TEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file *%R*.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file *%C*.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll*. So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text.* The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a. Edit the Ratfor program.
1. Remove all tabs.
2. Change all lower case letters to upper case letters.
3. Convert the file to an 80-column card image file.
b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.
c. Compile the Fortran.
d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. *Warning:* Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll -lS
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9 Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
            int k;
[0-9]+      {
            scanf(-1, yytext, "%d", &k);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

to do just that. The rule [0-9]+ recognizes strings of digits; *scanf* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
-?[0-9]+                {
                        scanf(-1, yytext, "%d", &k);
                        printf("%d", k%7 == 0 ? k+3 : k);
                        }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}  {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]      ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+       |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+     |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+     {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' | *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'-'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
                  yytext[0] =+ 'a' - 'd';
                  ECHO;
                  }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
                        ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10 Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of ***start conditions*** on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word ***magic*** to ***first*** on every line which began with the letter ***a***, changing ***magic*** to ***second*** on every line which began with the letter ***b***, and changing ***magic*** to ***third*** on every line which began with the letter ***c***. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
            int flag;
%%
^a          {flag = 'a'; ECHO;}
^b          {flag = 'b'; ECHO;}
^c          {flag = 'c'; ECHO;}
\n          {flag =  0 ; ECHO;}
magic       {
            switch (flag)
            {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
            }
            }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start     name1 name2 ...
```

where the conditions may be named in any order. The word ***Start*** may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the < > brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition ***name1***. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to ***name1***. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11 Character Set.

The programs generated by Lex handle character I/O only through the routines ***input, output,*** and ***unput.*** Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *1Ha* while in C the letter *a* is represented as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the

```
%T
 1    Aa
 2    Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

Sample character table.

rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input, output, unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

```
%S name1 name2 ...
```

5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

6) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) Changes to internal array sizes, in the form

%x nnn

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13 Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15 References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software – Practice and Experience, **5**, pp. 395-496 (1975).
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Comm. ACM **18**, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor,* Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library,* Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

PART 4: SYSTEM PROGRAMMING

The six articles in this part provide a background for understanding the operation of the ULTRIX-32 system. While these articles are conceptually valid, some details are obsolete. The file system description in particular does not reflect the current implementation.

Fundamentals

"UNIX Implementation," by Thompson, summarizes the inner workings of the ULTRIX-32 system: what parts make up the software and how they cooperate to make the system function. Topics include features of the kernel code, process control, the I/O system, and the file system.

Users are prohibited from changing the kernel code, but they can write utilities that use the kernel as a tool.

The description of process control provides explanations of:

- User and system processes
- Use of read-only segments of user processes
- Use of primary and secondary memory
- User address space
- User and system data segments
- Process tables
- *Exec*'ing a file
- Use of forks
- Swapping files
- Process synchronization
- Possible race conditions
- Process priorities

The I/O system supports two kinds of input/output processing: structured and unstructured. Structured I/O handles data in 512-byte blocks, while unstructured (character) I/O handles data in all other formats.

"UNIX Implementation" serves as a primer for the remaining articles in this part, and it should be read first.

ULTRIX-32 Virtual Machine

The environment in which a user process runs on the ULTRIX-32 system is called the ULTRIX-32 virtual machine. The "4.2BSD System Manual" defines this environment. It tells what the computer looks like to user processes and it identifies the kernel and system facilities available to user processes. The article defines the commands and calls related to

many of the concepts and generalities presented in the first article in this part, "UNIX Implementation."

Available kernel facilities include:

- Processes and protection
- Memory management
- Signals
- Timers
- Descriptors
- Resource controls
- Mounting and unmounting devices
- Accounting

Available system facilities include:

- Read, write, and I/O control calls
- File control
- Management of disk quotas
- Interprocess and interprocessor communication
- Terminal control

The "4.2BSD System Manual" is the best general reference article for kernel and system calls on the ULTRIX-32 system; it supplements all the other system programming articles in this part.

Assembly Language

The assembly language for the ULTRIX-32 system is called *as*. The C compiler produces *as* code, making the assembly language an intermediate stage in the process of translating high level language programs into executable code. *As* includes the VAX-11 instruction set.

The "Berkeley VAX/UNIX Assembler Reference Manual," by Reiser and Henry, specifies the rules and conventions of *as*. Explanations are terse and to the point. The article is written for compiler writers (at least one stage of every compiler must include *as* code) and for people who maintain the *as* assembler.

Device Drivers

The ULTRIX-32 software simplifies the process of writing device drivers. If your computer system includes a peripheral device that is not supported by the standard software, you will need to write your own device driver to control the operation of the device and the flow of data to and from it. The ULTRIX-32 system includes a set of routines that handle I/O for character and block devices. The article entitled "The UNIX I/O System," by Ritchie, explains how to use those routines to build a device driver. However, many of the details in this article refer to previous implementations.

Screen Manipulation

The ULTRIX-32 system provides a library of screen updating and cursor control routines (not all are foolproof) that serve as building blocks for any program that controls terminal screen displays at a basic level. The article entitled "Screen Updating and Cursor Movement Optimization: A Library Package," by Arnold, describes this set of routines and explains how to use them. There are output, input, initializing, and miscellaneous routines. An appendix provides two sample screen manipulation programs.

Suitable applications for this screen control library include text editors, terminal drivers, and video games.

Spooler

The ULTRIX-32 system provides a line printer spooler utility that supports standard printer devices and multiple spooling queues. The "4.2BSD Line Printer Spooler Manual," by Campbell, describes the installation, components, and functions of the spooler. The article should be particularly useful to users who want to modify the ULTRIX-32 system to accommodate a nonstandard printer. It tells how to alter the printer data base (the *printcap* file) for devices that do not conform to the default printer description. In addition, the article:

- Tells how to use the spooler with output filters
- Defines commands you can make to the line printer daemon and the line printer administration program
- Lists spooler error messages with explanations

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The UNIX† kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of

† UNIX is a trademark of Bell Laboratories.

processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

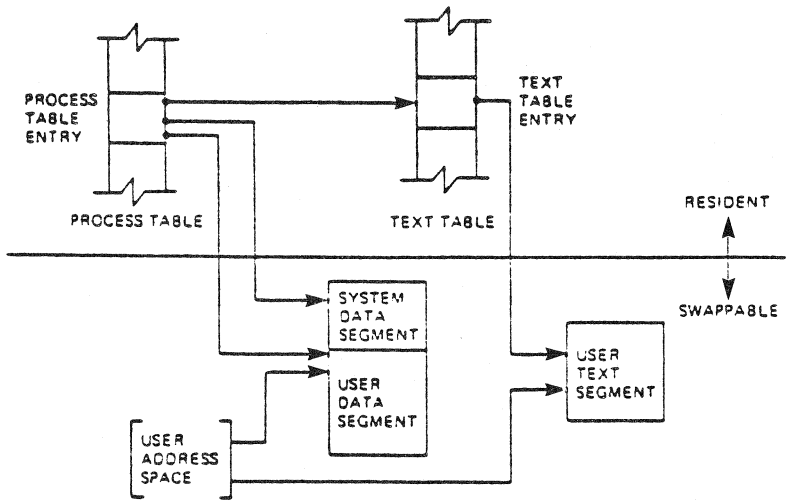


Fig. 1 Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of

its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry.

Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular

device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver.

The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further

discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and

reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

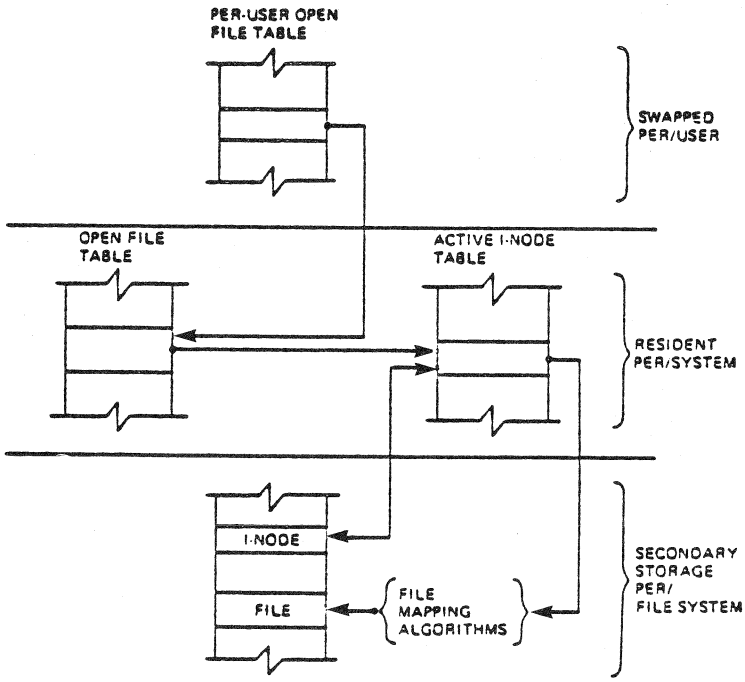


Fig. 2 File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **fork**s) share a common open file table entry. A separate open of the same file will only share

the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipe**s consists of implied **seek**s before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better interprocess communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[4] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[5]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices*, vol. 12, no. 4, pp. 40-50, April 1977.
2. E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, pp. 43-112, Academic Press, New York, 1968.
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal., 1975.
4. This issue, D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.
5. E. I. Organick, *The MULTICS System,* M.I.T. Press, Cambridge, Mass., 1972.

4.2BSD System Manual

Revised July, 1983

William Joy, Eric Cooper, Robert Fabry,

Samuel Leffler, Kirk McKusick and David Mosher

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

0. Notation and types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 2.1:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

The first line shows how the *read* routine is called, with three parameters. As shown on the second line *cc* is an integer and *read* also returns information in the parameter *buf*.

Description of all error conditions arising from each system call is not provided here; they appear in the programmer's manual. In particular, when accessed from the C language, many calls return a characteristic −1 value when an error occurs, returning the error code in the global variable *errno*. Other languages may present errors in different ways.

A number of system standard types are defined in the include file <sys/types.h> and used in the specifications here and in many C programs. These include **caddr_t** giving a memory address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

UNIX is a trademark of Bell Laboratories

1. Kernel primitives

The facilities available to a UNIX user process are logically divided into two parts: kernel facilities directly implemented by UNIX code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*. These kernel facilities are described in this section 1.

The facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
sethostid(hostid)
long hostid;

hostid = gethostid();
result long hostid;

sethostname(name, len)
char *name; int len;

len = gethostname(buf, buflen)
result int len; result char *buf; int buflen;
```

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the *getpid* routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

1.1.2. Process creation and termination

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result int pid;
```

The *fork* call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an *exit* call:

```
exit(status)
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

1.1.3. User and group ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
result int ruid;

euid = geteuid();
result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
result int rgid;

egid = getegid();
result int egid;
```

and the access group id set is returned by a *getgroups* call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups; int gidsetsize; result int gidset[gidsetsize];
```

The user and group id's are assigned at login time using the *setreuid*, *setregid*, and *setgroups* calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(gidsetsize, gidset)
int gidsetsize; int gidset[gidsetsize];
```

The *setreuid* call sets both the real and effective user-id's, while the *setregid* call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The *setgroups* call is restricted to the super-user.

1.1.4. Process groups

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, a system terminal has a process group and only processes which are in the process group of the terminal may read from the terminal, allowing arbitration of terminals among several different jobs.

The process group associated with a process may be changed by the *setpgrp* call:

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

1.2. Memory management†

1.2.1. Text, data and stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

```
addr = sstk(incr);
result caddr_t addr; int incr;
```

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

1.2.2. Mapping pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

```
/* protections are chosen from these bits, or-ed together */
#define  PROT_READ        0x4     /* pages can be read */
#define  PROT_WRITE       0x2     /* pages can be written */
#define  PROT_EXEC        0x1     /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define  MAP_SHARED       1       /* share changes */
#define  MAP_PRIVATE      2       /* changes are private */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for *len* bytes to be mapped from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A process can move pages within its own memory by using the *mremap* call:

```
mremap(addr, len, prot, share, fromaddr);
caddr_t addr; int len, prot, share; caddr_t fromaddr;
```

This call maps the pages starting at *fromaddr* to the address specified by *addr*.

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.2BSD.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

This causes further references to these pages to refer to private pages initialized to zero.

1.2.3. Page protection control

A process can control the protection of pages using the call

```
mprotect(addr, len, prot);
caddr_t addr; int len, prot;
```

This call changes the specified pages to have protection *prot*.

1.2.4. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

Behav describes expected behavior, as given in <mman.h>:

```
#define  MADV_NORMAL      0     /* no further special treatment */
#define  MADV_RANDOM      1     /* expect random page references */
#define  MADV_SEQUENTIAL  2     /* expect sequential references */
#define  MADV_WILLNEED    3     /* will need these pages */
#define  MADV_DONTNEED    4     /* don't need these pages */
```

Finally, a process may obtain information about whether pages are core resident by using the call

```
mincore(addr, len, vec)
caddr_t addr; int len; result char *vec;
```

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

1.3. Signals

1.3.1. Overview

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

1.3.2. Signal types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (SIGIOT, SIGEMT, and SIGTRAP).

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

1.3.3. Signal handlers

A process has a handler associated with each signal that controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
        int         (*sv_handler)();
        int         sv_mask;
        int         sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv mask* and *sv onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for signals. It prevents signals from being delivered much as a raised hardware interrupt priority level prevents hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sv handler* is called by a C call of the form

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

1.3.4. Sending signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo)
int pid, signo;

killpgrp(pgrp, signo)
int pgrp, signo;
```

Unless the process sending the signal is privileged, it and the process receiving the signal must have the same effective user id.

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

1.3.5. Protecting critical sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
long mask;
```

1.3.6. Signal stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;
```

to provide the system with a stack based at *ss sp* for delivery of signals. The value *ss onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current Greenwich time and the current time zone is set and returned by the calls:

```
#include <sys/time.h>

settimeofday(tvp, tzp);
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in <sys/time.h> as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvsec)
result long *tvsec;
```

returning only the tv sec field from the *gettimeofday* call.

1.4.2. Interval time

The system provides each process with three interval timers, defined in <sys/time.h>:

```
#define ITIMER_REAL     0       /* real time intervals */
#define ITIMER_VIRTUAL  1       /* virtual time intervals */
#define ITIMER_PROF     2       /* user and system virtual time */
```

The ITIMER_REAL timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A SIGPROF signal is delivered when it expires.

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval; /* timer interval */
        struct  timeval it_value;    /* current value */
};
```

and a timer is set or read by the call:

```
getitimer(which, value);
int which; result struct itimerval *value;

setitimer(which, value, ovalue);
int which; struct itimerval *value; result struct itimerval *ovalue;
```

The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

1.5. Descriptors

1.5.1. The reference table

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the *getdtablesize* call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

1.5.2. Descriptor properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. The generic operations applying to many of these types are described in section 2.1. Naming contexts, files and directories are described in section 2.2. Section 2.3 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 2.4.

1.5.3. Managing descriptor references

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The *dup2* call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

1.5.4. Multiplexing requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the *select* call:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result *in, *out, *except;
struct timeval *tvp;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the

specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bytes, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as *read* or *receive* is possible, or if a connection request may be accepted (see section 2.3.1.4).
- A descriptor selects for output if an output oriented operation such as *write* or *send* is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 2.1.3).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 1.3.2).

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting i/o on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg)
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define  F_SETFL        3       /* set descriptor options */
#define  F_GETFL        4       /* get descriptor options */
#define  F_SETOWN       5       /* set descriptor owner (pid/pgrp) */
#define  F_GETOWN       6       /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. F_SETOWN may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The *select* facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

1.5.5. Descriptor wrapping.†

A user process may build descriptors of a specified type by *wrapping* a communications channel with a system supplied protocol translator:

```
new = wrap(old, proto)
result int new; int old; struct dprop *proto;
```

Operations on the descriptor *old* are then translated by the system provided protocol translator into requests on the underyling object *old* in a way defined by the protocol. The protocols supported by the kernel may vary from system to system and are described in the programmers manual.

† The facilities described in this section are not included in 4.2BSD.

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file abstraction may or may not include locally generated "read-ahead" requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

Another example is the terminal driving facilities. Normally a terminal is associated with a communications line and the terminal type and standard terminal access protocol is wrapped around a synchronous communications line and given to the user. If a virtual terminal is required, the terminal driver can be wrapped around a communications link, the other end of which is held by a virtual terminal protocol interpreter.

1.6. Resource controls

1.6.1. Process priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define  PRIO_PROCESS     0      /* process */
#define  PRIO_PGRP        1      /* process group */
#define  PRIO_USER        2      /* user id */

prio = getpriority(which, who);
result int prio; int which, who;

setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range −20 to 20. The default priority is 0; lower priorities cause more favorable execution. The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

1.6.2. Resource utilization

The resources used by a process are returned by a *getrusage* call, returning information in a structure defined in <sys/resource.h>:

```
#define  RUSAGE_SELF      0      /* usage by this process */
#define  RUSAGE_CHILDREN -1      /* usage by all children */

getrusage(who, rusage)
int who; result struct rusage *rusage;

struct rusage {
        struct    timeval ru_utime;   /* user time used */
        struct    timeval ru_stime;   /* system time used */
        int       ru_maxrss;          /* maximum core resident set size: kbytes */
        int       ru_ixrss;           /* integral shared memory size (kbytes*sec) */
        int       ru_idrss;           /* unshared data " */
        int       ru_isrss;           /* unshared stack " */
        int       ru_minflt;          /* page-reclaims */
        int       ru_majflt;          /* page faults */
        int       ru_nswap;           /* swaps */
        int       ru_inblock;         /* block input operations */
        int       ru_oublock;         /* block output " */
        int       ru_msgsnd;          /* messages sent */
        int       ru_msgrcv;          /* messages received */
        int       ru_nsignals;        /* signals received */
        int       ru_nvcsw;           /* voluntary context switches */
        int       ru_nivcsw;          /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

1.6.3. Resource limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the *getrlimit* and *setrlimit* calls:

```
#define  RLIMIT_CPU        0       /* cpu time in milliseconds */
#define  RLIMIT_FSIZE      1       /* maximum file size */
#define  RLIMIT_DATA       2       /* maximum data segment size */
#define  RLIMIT_STACK      3       /* maximum stack segment size */
#define  RLIMIT_CORE       4       /* maximum core file size */
#define  RLIMIT_RSS        5       /* maximum resident set size */

#define  RLIM_NLIMITS      6

#define  RLIM_INFINITY     0x7fffffff

struct rlimit {
        int         rlim_cur;          /* current (soft) limit */
        int         rlim_max;          /* hard limit */
};

getrlimit(resource, rlp)
int resource; result struct rlimit *rlp;

setrlimit(resource, rlp)
int resource; struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

1.7. System operation support

Unless noted otherwise, the calls in this section are permitted only to a privileged user.

1.7.1. Bootstrap operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Dir* is normally a name in the root directory.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

specifies a device to be made available for paging and swapping.

1.7.2. Shutdown operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
sync();
```

schedules input/output to clean all system buffer caches. (This call does not require priveleged status.)

The call

```
reboot(how)
int how;
```

causes a machine halt or reboot. The call may request a reboot by specifying *how* as RB AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

1.7.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

2. System facilities

This section discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

Directory contexts
: A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files may also be mapped into process address space. A directory may be read as a file†.

Communications domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
: A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
: Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctl*s.

Processes
: Process descriptors provide facilities for control and debugging of other processes.

† Support for mapping files is not included in the 4.2 release.

2.1. Generic operations

Many system abstractions support the operations *read*, *write* and *ioctl*. We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

2.1.1. Read and write

The *read* and *write* system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The *read* call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is −1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The *write* call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in <sys/uio.h> as:

```
struct iovec {
        caddr_t     iov msg;        /* base of a component */
        int         iov len;        /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;
```

Here *iovlen* is the count of elements in the *iov* array.

2.1.2. Input/output control

Control operations on an object are performed by the *ioctl* operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct *ioctl* requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in <sys/ioctl.h>.

2.1.3. Non-blocking and asynchronous operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 1.5.4. Thereafter the *read* call will return a specific EWOULDBLOCK error indication if there is no data to be *read*. The process may *dselect* the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a *select* call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be *select*ed for *write* to find out when the operation can be retried. When *select* indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

2.2. File system

2.2.1. Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other sub-directories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

2.2.2. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by "/" characters, where each file name is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a "/", it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a "/" it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name ".." in each directory refers to the parent directory of that directory. The parent directory of a file system is always the systems root directory.

The calls

```
chdir(path);
char *path;

chroot(path)
char *path;
```

change the current working directory and root directory context of a process. Only the superuser can change the root directory context of a process.

2.2.3. Creation and removal

The file system allows directories, files, special devices, and "portals" to be created and removed from the file system.

2.2.3.1. Directory creation and removal

A directory is created with the *mkdir* system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the *rmdir* system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

2.2.3.2. File creation

Files are created with the *open* system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include O_CREAT from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in <sys/file.h>:

```
#define  O_RDONLY     000    /* open for reading */
#define  O_WRONLY     001    /* open for writing */
#define  O_RDWR       002    /* open for read & write */
#define  O_NDELAY     004    /* non-blocking open */
#define  O_APPEND     010    /* append on each write */
#define  O_CREAT      01000  /* open with file create */
#define  O_TRUNC      02000  /* open with truncation */
#define  O_EXCL       04000  /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility.

2.2.3.3. Creating references to devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their "major" and "minor" device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special *ioctl* operations, and may have input and output performed in large units. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

2.2.3.4. Portal creation†

The call

```
fd = portal(name, server, param, dtype, protocol, domain, socktype)
result int fd; char *name, *server, *param; int dtype, protocol;
int domain, socktype;
```

places a *name* in the file system name space that causes connection to a server process when the name is used. The portal call returns an active portal in *fd* as though an access had occurred to activate an inactive portal, as now described.

† The *portal* call is not implemented in 4.2BSD.

When an inactive portal is accesseed, the system sets up a socket of the specified *socktype* in the specified communications *domain* (see section 2.3), and creates the *server* process, giving it the specified *param* as argument to help it identify the portal, and also giving it the newly created socket as descriptor number 0. The accessor of the portal will create a socket in the same *domain* and *connect* to the server. The user will then *wrap* the socket in the specified *protocol* to create an object of the required descriptor type *dtype* and proceed with the operation which was in progress before the portal was encountered.

While the server process holds the socket (which it received as *fd* from the *portal* call on descriptor 0 at activation) further references will result in connections being made to the same socket.

2.2.3.5. File, device, and portal removal

A reference to a file, special device or portal may be removed with the *unlink* call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

2.2.4. Reading and modifying file attributes

Detailed information about the attributes of a file may be obtained with the calls:

```
#include <sys/stat.h>

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The *stat* structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the *lstat* call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp)
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

2.2.5. Links and renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The *unlink* primitive may be used to remove either type of link.

If a file is a symbolic link, the "value" of the link may be read with the *readlink* call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the *rename* call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

2.2.6. Extension and truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about

in the file in a random access fashion. To set the current offset into a file, the *lseek* call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in <sys/file.h> as one of,

```
#define  L_SET      0    /* set absolute file offset */
#define  L_INCR     1    /* set file offset relative to current position */
#define  L_XTND     2    /* set offset relative to end-of-file */
```

The call "lseek(fd, 0, L_INCR)" returns the current offset into the file.

Files may have "holes" in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path; int length;

ftruncate(fd, length);
int fd, length;
```

reducing the size of the specified file to *length* bytes.

2.2.7. Checking accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the *access* call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in <sys/file.h>:

```
#define  F_OK       0    /* file exists */
#define  X_OK       1    /* file is executable */
#define  W_OK       2    /* file is writable */
#define  R_OK       4    /* file is readable */
```

The presence or absence of advisory locks does not affect the result of *access*.

2.2.8. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory *read* or *write* lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an *open* call by applying the *flock* primitive,

```
flock(fd, how);
int fd, how;
```

where the *how* parameter is formed from bits defined in <sys/file.h>:

```
#define LOCK_SH     1    /* shared lock */
#define LOCK_EX     2    /* exclusive lock */
#define LOCK_NB     4    /* don't block when locking */
#define LOCK_UN     8    /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a *flock* call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

2.2.9. Disk quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file)
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

```
#include <sys/quota.h>

quota(cmd, uid, arg, addr)
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <sys/quota.h> contains definitions pertinent to the use of this call.

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file <sys/socket.h>. Important standard domains supported by the system are the "unix" domain, AF_UNIX, for communication within the system, and the "internet" domain for communication in the DARPA internet, AF_INET. Other domains can be added to the system.

2.3.1.2. Socket types and protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in <sys/socket.h>:

```
/* Standard socket types */
#define SOCK_DGRAM      1       /* datagram */
#define SOCK_STREAM     2       /* virtual circuit */
#define SOCK_RAW        3       /* raw socket */
#define SOCK_RDM        4       /* reliably-delivered message */
#define SOCK_SEQPACKET  5       /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The *send* and *receive* operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.†

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

2.3.1.3. Socket creation, naming and service establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the *socket* call:

† 4.2BSD does not support the SOCK RDM and SOCK SEQPACKET types.

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a *bind* call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a *getsockname* call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with *getpeername*:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made, it is possible to *listen* for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An *accept* call:

```
t = accept(s, name, anamelen);
result int t; int s; result caddr_t name; result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

2.3.1.5. Making connections

An active connection to a named socket is made by the *connect* call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the *socketpair* call†:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with *accept* and *connect*.

The call

```
pipe(pv)
result int pv[2];
```

creates a pair of SOCK_STREAM sockets in the UNIX domain, with pv[0] only writeable and

† 4.2BSD supports *socketpair* creation only in the "unix" communication domain.

pv[1] only readable.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result caddr_t buf; int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result caddr_t buf; int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define MSG_PEEK        0x1     /* peek at incoming message */
#define MSG_OOB         0x2     /* process out-of-band data */
```

2.3.1.7. Scatter/gather and exchanging access rights

It is possible scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in <sys/socket.h>, which can be used to conveniently contain the parameters to the calls:

```
struct msghdr {
        caddr_t     msg_name;           /* optional address */
        int         msg_namelen;        /* size of address */
        struct      iov *msg_iov;       /* scatter/gather array */
        int         msg_iovlen;         /* # elements in msg_iov */
        caddr_t     msg_accrights;      /* access rights sent/received */
        int         msg_accrightslen;   /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 2.1.3. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations *sendmsg* and *recvmsg*:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flags;
```

2.3.1.8. Using read and write with sockets

The normal UNIX *read* and *write* calls may be applied to connected sockets and translated into *send* and *receive* calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

2.3.1.9. Shutting down halves of full-duplex connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

2.3.1.10. Socket and protocol options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The *getsockopt* and *setsockopt* calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname; result caddr_t optval; result int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

2.3.2. UNIX domain

This section describes briefly the properties of the UNIX communications domain.

2.3.2.1. Types of sockets

In the UNIX domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides (usually) reliable message-style communications.

2.3.2.2. Naming

Socket names are strings and may appear in the UNIX file system name space through portals†.

† The 4.2BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

2.3.2.3. Access rights transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

2.3.3. INTERNET domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the document describing the network implementation in 4.2BSD.

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the INTERNET TCP protocol; SOCK_DGRAM by the UDP protocol. The SOCK_SEQPACKET has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

2.3.3.2. Socket naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional address for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

2.3.3.3. Access rights transmission

No access rights transmission facilities are provided in the INTERNET domain.

2.3.3.4. Raw access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as SOCK_RAW sockets.

2.4. Terminals and Devices

2.4.1. Terminals

Terminals support *read* and *write* i/o operations, as well as a collection of terminal specific *ioctl* operations, to control input character editing, and output delays.

2.4.1.1. Terminal input

Terminals are handled according to the underlying communication characteristics such as baud rate and required delays, and a set of software parameters.

2.4.1.1.1. Input modes

A terminal is in one of three possible modes: *raw*, *cbreak*, or *cooked*. In raw mode all input is passed through to the reading process immediately and without interpretation. In cbreak mode, the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available as in raw mode. In cooked mode, input is processed to provide standard line-oriented local editing functions, and input is presented on a line-by-line basis.

2.4.1.1.2. Interrupt characters

Interrupt characters are interpreted by the terminal handler only in cbreak and cooked modes, and cause a software interrupt to be sent to all processes in the process group associated with the terminal. Interrupt characters exist to send SIGINT and SIGQUIT signals, and to stop a process group with the SIGTSTP signal either immediately, or when all input up to the stop character has been read.

2.4.1.1.3. Line editing

When the terminal is in cooked mode, editing of an input line is performed. Editing facilities allow deletion of the previous character or word, or deletion of the current input line. In addition, a special character may be used to reprint the current input line after some number of editing operations have been applied.

Certain other characters are interpreted specially when a process is in cooked mode. The *end of line* character determines the end of an input record. The *end of file* character simulates an end of file occurrence on terminal input. Flow control is provided by *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and a *literal character* may be used to force literal input of the immediately following character in the input line.

2.4.1.2. Terminal output

On output, the terminal handler provides some simple formatting services. These include converting the carriage return character to the two character return-linefeed sequence, displaying non-graphic ASCII characters as "^character", inserting delays after certain standard control characters, expanding tabs, and providing translations for upper-case only terminals.

2.4.1.3. Terminal control operations

When a terminal is first opened it is initialized to a standard state and configured with a set of standard control, editing, and interrupt characters. A process may alter this configuration with certain control operations, specifying parameters in a standard structure:

```
struct ttymode {
        short       tt_ispeed;      /* input speed */
        int         tt_iflags;      /* input flags */
        short       tt_ospeed;      /* output speed */
        int         tt_oflags;      /* output flags */
};
```

and "special characters" are specified with the *ttychars* structure,

```
struct ttychars {
        char        tc_erasec;      /* erase char */
        char        tc_killc;       /* erase line */
        char        tc_intrc;       /* interrupt */
        char        tc_quitc;       /* quit */
        char        tc_startc;      /* start output */
        char        tc_stopc;       /* stop output */
        char        tc_eofc;        /* end-of-file */
        char        tc_brkc;        /* input delimiter (like nl) */
        char        tc_suspc;       /* stop process signal */
        char        tc_dsuspc;      /* delayed stop process signal */
        char        tc_rprntc;      /* reprint line */
        char        tc_flushc;      /* flush output (toggles) */
        char        tc_werasc;      /* word erase */
        char        tc_lnextc;      /* literal next character */
};
```

2.4.1.4. Terminal hardware support

The terminal handler allows a user to access basic hardware related functions; e.g. line speed, modem control, parity, and stop bits. A special signal, SIGHUP, is automatically sent to processes in a terminal's process group when a carrier transition is detected. This is normally associated with a user hanging up on a modem controlled terminal line.

2.4.2. Structured devices

Structures devices are typified by disks and magnetic tapes, but may represent any random-access device. The system performs read-modify-write type buffering actions on block devices to allow them to be read and written in a totally random access fashion like ordinary files. File systems are normally created in block devices.

2.4.3. Unstructured devices

Unstructured devices are those devices which do not support block structure. Familiar unstructured devices are raw communications lines (with no terminal handler), raster plotters, magnetic tape and disks unfettered by buffering and permitting large block input/output and positioning and formatting commands.

2.5. Process and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of 4.1BSD is provided in 4.2BSD. Planned enhancements would allow a descriptor based process control facility.

I. Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

sethostid	set UNIX host id
gethostid	get UNIX host id
sethostname	set UNIX host name
gethostname	get UNIX host name
getpid	get process id
fork	create new process
exit	terminate a process
execve	execute a different process
getuid	get user id
geteuid	get effective user id
setreuid	set real and effective user id's
getgid	get accounting group id
getegid	get effective accounting group id
getgroups	get access group set
setregid	set real and effective group id's
setgroups	set access group set
getpgrp	get process group
setpgrp	set process group

1.2 Memory management

<mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
mremap†	remap pages in memory
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages

1.3 Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone
getitimer	read an interval timer
setitimer	get and set an interval timer

† Not supported in 4.2BSD.

profil	profile process

1.5 Descriptors

getdtablesize	descriptor reference table size
dup	duplicate descriptor
dup2	duplicate to specified index
close	close descriptor
select	multiplex input/output
fcntl	control descriptor options
wrap†	wrap descriptor with protocol

1.6 Resource controls

<sys/resource.h>	resource-related definitions
getpriority	get process priority
setpriority	set process priority
getrusage	get resource usage
getrlimit	get resource limitations
setrlimit	set resource limitations

1.7 System operation support

mount	mount a device file system
swapon	add a swap device
umount	umount a file system
sync	flush system caches
reboot	reboot a machine
acct	specify accounting file

2. System facilities

2.1 Generic operations

read	read data
write	write data
<sys/uio.h>	scatter-gather related definitions
readv	scattered data input
writev	gathered data output
<sys/ioctl.h>	standard control operations
ioctl	device control operation

2.2 File system

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a "f".

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
portal†	make a portal entry
unlink	remove a link
stat*	return status for a file

† Not supported in 4.2BSD.

lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

2.3 Communications

<sys/socket.h>	standard definitions
socket	create socket
bind	bind socket to name
getsockname	get socket name
listen	allow queueing of connections
accept	accept a connection
connect	connect to peer socket
socketpair	create pair of connected sockets
sendto	send data to named socket
send	send data to connected socket
recvfrom	receive data on unconnected socket
recv	receive data on connected socket
sendmsg	send gathered data and/or rights
recvmsg	receive scattered data and/or rights
shutdown	partially close full-duplex connection
getsockopt	get socket option
setsockopt	set socket option

2.5 Terminals, block and character devices

2.4 Processes and kernel hooks

Berkeley VAX/UNIX Assembler Reference Manual

John F. Reiser
Bell Laboratories,
Holmdel, NJ

and

Robert R. Henry[1]
Electronics Research Laboratory
University of California
Berkeley, CA 94720

November 5, 1979

Revised
February 9, 1983

1. Introduction

This document describes the usage and input syntax of the UNIX VAX-11 assembler *as*. *As* is designed for assembling the code produced by the "C" compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended. This document is intended only for the writer of a compiler or a maintainer of the assembler.

1.1. Assembler Revisions since November 5, 1979

There has been one major change to *as* since the last release. *As* has been updated to assemble the new instructions and data formats for "G" and "H" floating point numbers, as well as the new queue instructions.

1.2. Features Supported, but No Longer Encouraged as of February 9, 1983

These feature(s) in *as* are supported, but no longer encouraged.

- The colon operator for field initialization is likely to disappear.

2. Usage

As is invoked with these command arguments:

as [**−LVWJR**] [**−d***n*] [**−DTS**] [**−t** *directory*] [**−o** *output*] [$name_1$] ··· [$name_n$]

The **−L** flag instructs the assembler to save labels beginning with a "L" in the symbol table portion of the *output* file. Labels are not saved by default, as the default action of the link editor *ld* is to discard them anyway.

The **−V** flag tells the assembler to place its interpass temporary file into virtual memory. In normal circumstances, the system manager will decide where the temporary file should lie. Our experiments with very large temporary files show that placing the temporary file into virtual memory will save about 13% of the assembly time, where the size of the temporary file is about 350K bytes. Most assembler sources will not be this long.

The **−W** turns of all warning error reporting.

The **−J** flag forces UNIX style pseudo−branch instructions with destinations further away than a byte displacement to be turned into jump instructions with 4 byte offsets. The

[1]Preparation of this paper supported in part by the National Science Foundation under grant MCS #78-07291.

−J flag buys you nothing if **−d2** is set. (See §8.4, and future work described in §11)

The **−R** flag effectively turns "**.data** *n*" directives into "**.text** *n*" directives. This obviates the need to run editor scripts on assembler source to "read−only" fix initialized data segments. Uninitialized data (via **.lcomm** and **.comm** directives) is still assembled into the data or bss segments.

The **−d** flag specifies the number of bytes which the assembler should allow for a displacement when the value of the displacement expression is undefined in the first pass. The possible values of *n* are 1, 2, or 4; the assembler uses 4 bytes if **-d** is not specified. See §8.2.

Provided the **−V** flag is not set, the **−t** flag causes the assembler to place its single temporary file in the *directory* instead of in */tmp*.

The **−o** flag causes the output to be placed on the file *output*. By default, the output of the assembler is placed in the file *a.out* in the current directory.

The input to the assembler is normally taken from the standard input. If file arguments occur, then the input is taken sequentially from the files $name_1$, $name_2$ ··· $name_n$ This is not to say that the files are assembled separately; $name_1$ is effectively concatenated to $name_2$, so multiple definitions cannot occur amongst the input sources.

The **−D** (debug), **−T** (token trace), and the **−S** (symbol table) flags enable assembler trace information, provided that the assembler has been compiled with the debugging code enabled. The information printed is long and boring, but useful when debugging the assembler.

3. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), constants, and operators.

3.1. Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and dollar "$"). The first character may not be numeric. Identifiers may be (practically) arbitrary long; all characters are significant.

3.2. Constants

3.2.1. Scalar constants

All scalar (non floating point) constants are (potentially) 128 bits wide. Such constants are interpreted as two's complement numbers. Note that 64 bit (quad words) and 128 bit (octal word) integers are only partially supported by the VAX hardware. In addition, 128 bit integers are only supported by the extended VAX architecture. *As* supports 64 and 128 bit integers only so they can be used as immediate constants or to fill initialized data space. *As* can not perform arithmetic on constants larger than 32 bits.

Scalar constants are initially evaluated to a full 128 bits, but are pared down by discarding high order copies of the sign bit and categorizing the number as a long, quad or octal integer. Numbers with less precision than 32 bits are treated as 32 bit quantities.

The digits are "0123456789abcdefABCDEF" with the obvious values.

An octal constant consists of a sequence of digits with a leading zero.

A decimal constant consists of a sequence of digits without a leading zero.

A hexadecimal constant consists of the characters "0x" (or "0X") followed by a sequence of digits.

A single-character constant consists of a single quote "'" followed by an ASCII character, including ASCII newline. The constant's value is the code for the given character.

3.2.2. Floating Point Constants

Floating point constants are internally represented in the VAX floating point format that is specified by the lexical form of the constant. Using the meta notation that [dec] is a decimal digit ("0123456789"), [expt] is a type specification character ("fFdDhHgG"), [expe] is a exponent delimiter and type specification character ("eEfFdDhHgG"), x^* means 0 or more occurences of x, x^+ means 1 or more occurences of x, then the general lexical form of a floating point number is:

0[expe]([+-])[dec]$^+$(.)([dec]*)([expt]([+-])(dec]$^+$))

The standard semantic interpretation is used for the signed integer, fraction and signed power of 10 exponent. If the exponent delimiter is specified, it must be either an "e" or "E", or must agree with the initial type specification character that is used. The type specification character specifies the type and representation of the constructed number, as follows:

type character	floating representation	size (bits)
f, F	F format floating	32
d, D	D format floating	64
g, G	G format floating	64
h, H	H format floating	128

Note that "G" and "H" format floating point numbers are not supported by all implementations of the VAX architecture. *As* does not require the augmented architecture in order to run.

The assembler uses the library routine *atof()* to convert "F" and "D" numbers, and uses its own conversion routine (derived from *atof*, and believed to be numerically accurate) to convert "G" and "H" floating point numbers.

Collectively, all floating point numbers, together with quad and octal scalars are called *Bignums*. When *as* requires a Bignum, a 32 bit scalar quantity may also be used.

3.2.3. String Constants

A string constant is defined using the same syntax and semantics as the "C" language uses. Strings begin and end with a """ (double quote). The DEC MACRO-32 assembler conventions for flexible string quoting is not implemented. All "C" backslash conventions are observed; the backslash conventions peculiar to the PDP-11 assembler are not observed. Strings are known by their value and their length; the assembler does not implicitly end strings with a null byte.

3.3. Operators

There are several single-character operators; see §6.1.

3.4. Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5. Scratch Mark Comments

The character "#" introduces a comment, which extends through the end of the line on which it appears. Comments starting in column 1, having the format "# *expression string*", are interpreted as an indication that the assembler is now assembling file *string* at line *expression*. Thus, one can use the "C" preprocessor on an assembly language source file, and use the *#include* and *#define* preprocessor directives. (Note that there may not be an assembler comment starting in column 1 if the assembler source is given to the "C" preprocessor, as it will be interpreted by the preprocessor in a way not intended.)

Comments are otherwise ignored by the assembler.

3.6. "C" Style Comments

The assembler will recognize "C" style comments, introduced with the prologue **/*** and ending with the epilogue ***/**. "C" style comments may extend across multiple lines, and are the preferred comment style to use if one chooses to use the "C" preprocessor.

4. Segments and Location Counters

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The UNIX operating system makes some assumptions about the content of these segments; the assembler does not. Within the text and data segments there are a number of sub-segments, distinguished by number ("**text** 0", "**text** 1", ··· "**data** 0", "**data** 1", ···). Currently there are four subsegments each in text and data. The subsegments are for programming convenience only.

Before writing the output file, the assembler zero-pads each text subsegment to a multiple of four bytes and then concatenates the subsegments in order to form the text segment; an analogous operation is done for the data segment. Requesting that the loader define symbols and storage regions is the only action allowed by the assembler with respect to the bss segment. Assembly begins in "**text** 0".

Associated with each (sub)segment is an implicit location counter which begins at zero and is incremented by 1 for each byte assembled into the (sub)segment. There is no way to explicitly reference a location counter. Note that the location counters of subsegments other than "**text** 0" and "**data** 0" behave peculiarly due to the concatenation used to form the text and data segments.

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are two kinds of statements: null statements and keyword statements. Either kind of statement may be preceded by one or more labels.

5.1. Named Global Labels

A global label consists of a name followed by a colon. The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

A global label is referenced by its name.

Global labels beginning with a "L" are discarded unless the **−L** option is in effect.

5.2. Numeric Local Labels

A numeric label consists of a digit 0 to 9 followed by a colon. Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of the location counter to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References to symbols of the form "*n*b" refer to the first numeric label "*n*:" *b*ackwards from the reference; "*n*f" symbols refer to the first numeric label "*n*:" *f*orwards from the reference. Such numeric labels conserve the inventive powers of the human programmer.

For various reasons, *as* turns local labels into labels of the form L*n*.$*m*. Although unlikely, these generated labels may conflict with programmer defined labels.

5.3. Null statements

A null statement is an empty statement ignored by the assembler. A null statement may be labeled, however.

5.4. Keyword statements

A keyword statement begins with one of the many predefined keywords known to *as*; the syntax of the remainder of the statement depends on the keyword. All instruction opcodes are keywords. The remaining keywords are assembler pseudo-operations, also called *directives*. The pseudo-operations are listed in §8, together with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, operators, and parentheses. Each expression has a type.

All operators in expressions are fundamentally binary in nature. Arithmetic is two's complement and has 32 bits of precision. *As* can not do arithmetic on floating point numbers, quad or octal precision scalar numbers. There are four levels of precedence, listed here from lowest precedence level to highest:

precedence	operators
binary	+, −
binary	|, &, ^, !
binary	*, /, %,
unary	−, ~

All operators of the same precedence are evaluated strictly left to right, except for the evaluation order enforced by parenthesis.

6.1. Expression Operators

The operators are:

operator	meaning
+	addition
−	(binary) subtraction
*	multiplication
/	division
%	modulo
−	(unary) 2's complement
&	bitwise and
|	bitwise or
^	bitwise exclusive or
!	bitwise or not
~	bitwise 1's complement
>	logical right shift
>>	logical right shift
<	logical left shift
<<	logical left shift

Expressions may be grouped by use of parentheses, "(" and ")".

6.2. Data Types

The assembler manipulates several different types of expressions. The types likely to be met explicitly are:

undefined Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external
A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of "." is "text 0".

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of "." is "data 0".

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments.

external absolute, text, data, or bss
Symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register The symbols

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 ap fp sp pc

are predefined as register symbols. In addition, the "%" operator converts the following absolute expression whose value is between 0 and 15 into a register reference.

other types
Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3. Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then

(1) If one of the operands is undefined, the result is undefined.

(2) If both operands are absolute, the result is absolute.

(3) If an absolute is combined with one of the "other types" mentioned above, the result has the other type. An "other type" combined with an explicitly discussed type other than absolute it acts like an absolute.

Further rules applying to particular operators are:

\+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

− If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others
It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations (Directives)

The keywords listed below introduce directives or instructions, and influence the later behavior of the assembler for this statement. The metanotation

[stuff]

means that 0 or more instances of the given "stuff" may appear.

Boldface tokens must appear literally; words in *italic* words are substitutable.

The pseudo−operations listed below are grouped into functional categories.

7.1. Interface to a Previous Pass

.ABORT

As soon as the assembler sees this directive, it ignores all further input (but it does read to the end of file), and aborts the assembly. No files are created. It is anticipated that this would be used in a pipe interconnected version of a compiler, where the first major syntax error would cause the compiler to issue this directive, saving unnecessary work in assembling code that would have to be discarded anyway.

.file *string*

This directive causes the assembler to think it is in file *string*, so error messages reflect the proper source file.

.line *expression*

This directive causes the assembler to think it is on line *expression* so error messages reflect the proper source file.

The only effect of assembling multiple files specified in the command string is to insert the *file* and *line* directives, with the appropriate values, at the beginning of the source from each file.

*expression string*
*expression*

This is the only instance where a comment is meaningful to the assembler. The "#" *must* be in the first column. This meta comment causes the assembler to believe it is on line *expression*. The second argument, if included, causes the assembler to believe it is in file *string*, otherwise the current file name does not change.

7.2. Location Counter Control

.data [*expression*]
.text [*expression*]

These two pseudo-operations cause the assembler to begin assembling into the indicated text or data subsegment. If specified, the *expression* must be defined and absolute; an omitted *expression* is treated as zero. The effect of a **.data** directive is treated as a **.text** directive if the **−R** assembly flag is set. Assembly starts in the **.text** 0 subsegment.

The directives **.align** and **.org** also control the placement of the location counter.

7.3. Filled Data

.align *align_expr* [, *fill_expr*]

The location counter is adjusted so that the *expression* lowest bits of the location counter become zero. This is done by assembling from 0 to 2^{align_expr} bytes, taken from the low order byte of *fill_expr*. If present, *fill_expr* must be absolute; otherwise it defaults to 0. Thus ".align 2" pads by null bytes to make the location counter evenly divisible by 4. The *align_expr* must be defined, absolute, nonnegative, and less than 16.

Warning: the subsegment concatenation convention and the current loader conventions may not preserve attempts at aligning to more than 2 low-order zero bits.

.org *org_expr* [, *fill_expr*]

The location counter is set equal to the value of *org_expr*, which must be defined and absolute. The value of the *org_expr* must be greater than the current value of the location counter. Space between the current value of the location counter and the desired value are filled with bytes taken from the low order byte of *fill_expr*, which must be absolute and defaults to 0.

.space *space_expr* [, *fill_expr*]

The location counter is advanced by *space_expr* bytes. *Space_expr* must be defined and absolute. The space is filled in with bytes taken from the low order byte of *fill_expr*, which must be defined and absolute. *Fill_expr* defaults to 0. The **.fill** directive is a more general way to accomplish the **.space** directive.

.fill *rep_expr*, *size_expr*, *fill_expr*

All three expressions must be absolute. *fill_expr*, treated as an expression of size *size_expr* bytes, is assembled and replicated *rep_expr* times. The effect is to advance the current location counter *rep_expr* * *size_expr* bytes. *size_expr* must be between 1 and 8.

7.4. Symbol Definitions

7.5. Initialized Data

.byte *expr* [, *expr*]
.word *expr* [, *expr*]
.int *expr* [, *expr*]
.long *expr* [, *expr*]

The *expressions* in the comma-separated list are truncated to the size indicated by the key word:

keyword	length (bits)
.byte	8
.word	16
.int	32
.long	32

and assembled in successive locations. The *expressions* must be absolute.

Each *expression* may optionally be of the form:

$$expression_1 : expression_2$$

In this case, the value of $expression_2$ is truncated to $expression_1$ bits, and assembled in the next $expression_1$ bit field which fits in the natural data size being assembled. Bits which are skipped because a field does not fit are filled with zeros. Thus, "**.byte** 123" is equivalent to "**.byte** 8:123", and "**.byte** 3:1,2:1,5:1" assembles two bytes, containing the values 9 and 1.

NB: Bit field initialization with the colon operator is likely to disappear in future releases of the assembler.

.quad *number* [, *number*]
.octa *number* [, *number*]
.float *number* [, *number*]
.double *number* [, *number*]
.ffloat *number* [, *number*]
.dfloat *number* [, *number*]
.gfloat *number* [, *number*]
.hfloat *number* [, *number*]

These initialize Bignums (see §3.2.2) in successive locations whose size is a function on the key word. The type of the Bignums (determined by the exponent field, or lack thereof) may not agree with type implied by the key word. The following table shows the key words, their size, and the data types for the Bignums they expect.

keyword	format	length (bits)	valid *number*(s)
.quad	quad scalar	64	scalar
.octa	octal scalar	128	scalar
.float	F float	32	F, D and scalar
.ffloat	F float	32	F, D and scalar
.double	D float	64	F, D and scalar
.dfloat	D float	64	F, D and scalar
.gfloat	G float	64	G scalar
.hfloat	H float	128	H scalar

As will correctly perform other floating point conversions while initializing, but issues a warning message. *As* performs all floating point initializations and conversions using only the

facilities defined in the original (native) architecture.

.ascii *string* [, *string*]
.asciz *string* [, *string*]

Each *string* in the list is assembled into successive locations, with the first letter in the string being placed into the first location, etc. The **.ascii** directive will not null pad the string; the **.asciz** directive will null pad the string. (Recall that strings are known by their length, and need not be terminated with a null, and that the "C" conventions for escaping are understood.) The **.ascii** directive is identical to:
.byte $string_0$, $string_1$, · · ·

.comm *name*, *expression*

Provided the *name* is not defined elsewhere, its type is made "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes.

.lcomm *name*, *expression*

expression bytes will be allocated in the bss segment and *name* assigned the location of the first byte, but the *name* is not declared as global and hence will be unknown to the link editor.

.globl *name*

This statement makes the *name* external. If it is otherwise defined (by **.set** or by appearance as a label) it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.

Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbol. The assembler makes all otherwise undefined symbols external.

.set *name*, *expression*

The (*name*, *expression*) pair is entered into the symbol table. Multiple **.set** statements with the same name are legal; the most recent value replaces all previous values.

.lsym *name*, *expression*

A unique and otherwise unreferencable instance of the (*name*, *expression*) pair is created in the symbol table. The Fortran 77 compiler uses this mechanism to pass local symbol definitions to the link editor and debugger.

.stabs *string*, $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabn $expr_1$, $expr_2$, $expr_3$, $expr_4$
.stabd $expr_1$, $expr_2$, $expr_3$

The *stab* directives place symbols in the symbol table for the symbolic debugger, *sdb*[2]. A "stab" is a symbol *tab*le entry. The **.stabs** is a string stab, the **.stabn** is a stab not having a string, and the **.stabd** is a "dot" stab that implicitly references "dot", the current location

[2]Katseff, H.P. *Sdb: A Symbol Debugger.* Bell Laboratories, Holmdel, NJ. April 12, 1979.
Katseff, H.P. *Symbol Table Format for Sdb*, File 39394, Bell Laboratories, Holmdel, NJ. March 14, 1979.

counter.

The *string* in the **.stabs** directive is the name of a symbol. If the symbol name is zero, the **.stabn** directive may be used instead.

The other expressions are stored in the name list structure of the symbol table and preserved by the loader for reference by *sdb*; the value of the expressions are peculiar to formats required by *sdb*.

$expr_1$ is used as a symbol table tag (nlist field *n_type*).

$expr_2$ seems to always be zero (nlist field *n_other*).

$expr_3$ is used for either the source line number, or for a nesting level (nlist field *n_desc*).

$expr_4$ is used as tag specific information (nlist field *n_value*). In the case of the **.stabd** directive, this expression is nonexistent, and is taken to be the value of the location counter at the following instruction. Since there is no associated name for a **.stabd** directive, it can only be used in circumstances where the name is zero. The effect of a **.stabd** directive can be achieved by one of the other **.stab**x directives in the following manner:
.stabn $expr_1$, $expr_2$, $expr_3$, LL*n*
LL*n*:

The **.stabd** directive is preferred, because it does not clog the symbol table with labels used only for the stab symbol entries.

8. Machine instructions

The syntax of machine instruction statements accepted by *as* is generally similar to the syntax of DEC MACRO-32. There are differences, however.

8.1. Character set

As uses the character "$" instead of "#" for immediate constants, and the character "*" instead of "@" for indirection. Opcodes and register names are spelled with lower-case rather than upper-case letters.

8.2. Specifying Displacement Lengths

Under certain circumstances, the following constructs are (optionally) recognized by *as* to indicate the number of bytes to allocate for the displacement used when constructing displacement and displacement deferred addressing modes:

primary	alternate	length
B`	**B^**	byte (1 byte)
W`	**W^**	word (2 bytes)
L`	**L^**	long word (4 bytes)

One can also use lower case **b**, **w** or **l** instead of the upper case letters. There must be no space between the size specifier letter and the "^" or "`". The constructs **S^** and **G^** are not recognized by *as*, as they are by the DEC MACRO-32 assembler. It is preferred to use the "`"displacement so that the "^" is not misinterpreted as the **xor** operator.

Literal values (including floating-point literals used where the hardware expects a floating-point operand) are assembled as short literals if possible, hence not needing the **S^** DEC MACRO-32 directive.

If the displacement length modifier is present, then the displacement is **always** assembled with that displacement, even if it will fit into a smaller field, or if significance is lost. If the length modifier is not present, and if the value of the displacement is known exactly in *as*'s first pass, then *as* determines the length automatically, assembling it in the shortest possible way, Otherwise, *as* will use the value specified by the **−d** argument, which defaults to 4

bytes.

8.3. case*x* Instructions

As considers the instructions **caseb**, **casel**, **casew** to have three operands. The displacements must be explicitly computed by *as*, using one or more **.word** statements.

8.4. Extended branch instructions

These opcodes (formed in general by substituting a "j" for the initial "b" of the standard opcodes) take as branch destinations the name of a label in the current subsegment. It is an error if the destination is known to be in a different subsegment, and it is a warning if the destination is not defined within the object module being assembled.

If the branch destination is close enough, then the corresponding short branch "b" instruction is assembled. Otherwise the assembler choses a sequence of one or more instructions which together have the same effect as if the "b" instruction had a larger span. In general, *as* chooses the inverse branch followed by a **brw**, but a **brw** is sometimes pooled among several "j" instructions with the same destination.

As is unable to perform the same long/short branch generation for other instructions with a fixed byte displacement, such as the **sob**, **aob** families, or for the **acbx** family of instructions which has a fixed word displacement. This would be desirable, but is prohibitive because of the complexity of these instructions.

If the **−J** assembler option is given, a **jmp** instruction is used instead of a **brw** instruction for **ALL** "j" instructions with distant destinations. This makes assembly of large (>32K bytes) programs (inefficiently) possible. *As* does not try to use clever combinations of **brb**, **brw** and **jmp** instructions. The **jmp** instructions use PC relative addressing, with the length of the offset given by the **−d** assembler option.

These are the extended branch instructions *as* recognizes:

jeql	**jeqlu**	**jneq**
jgeq	**jgequ**	**jgtr**
jleq	**jlequ**	**jlss**
jbcc	**jbsc**	**jbcs**
jlbc	**jlbs**	
jcc	**jcs**	
jvc	**jvs**	
jbc	**jbs**	
jbr		

Note that **jbr** turns into **brb** if its target is close enough; otherwise a **brw** is used.

9. Diagnostics

Diagnostics are intended to be self explanatory and appear on the standard output. Diagnostics either report an *error* or a *warning*. Error diagnostics complain about lexical, syntactic and some semantic errors, and abort the assembly.

The majority of the warnings complain about the use of VAX features not supported by all implementations of the architecture. *As* will warn if new opcodes are used, if "G" or "H" floating point numbers are used and will complain about mixed floating conversions.

10. Limits

limit	what
Arbitrary[3]	Files to assemble
BUFSIZ	Significant characters per name
Arbitrary	Characters per input line
Arbitrary	Characters per string
Arbitrary	Symbols
4	Text segments
4	Data segments

11. Annoyances and Future Work

Most of the annoyances deal with restrictions on the extended branch instructions.

As only uses a two level algorithm for resolving extended branch instructions into short or long displacements. What is really needed is a general mechanism to turn a short conditional jump into a reverse conditional jump over one of **two** possible unconditional branches, either a **brw** or a **jmp** instruction. Currently, the **−J** forces the **jmp** instruction to *always* be used, instead of the shorter **brw** instruction when needed.

The assembler should also recognize extended branch instructions for **sob**, **aob**, and **acbx** instructions. **Sob** instructions will be easy, **aob** will be harder because the synthesized instruction uses the index operand twice, so one must be careful of side effects, and the **acbx** family will be much harder (in the general case) because the comparison depends on the sign of the addend operand, and two operands are used more than once. Augmenting *as* with these extended loop instructions will allow the peephole optimizer to produce much better loop optimizations, since it currently assumes the worst case about the size of the loop body.

The string temporary file is not put in memory when the -V flag is set. The string table in the generated a.out contains some strings and names that are never referenced from the symbol table; the loader removes these unreferenced strings, however.

[3]Although the number of characters available to the *argv* line is restricted by UNIX to 10240.

The UNIX I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result

†UNIX is a Trademark of Bell Laboratories.

from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver

to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on*, that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns –1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count*.

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns –1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
        int   c_cc; /* character count */
        char *c_cf;/* first character */
        char *c_cl;/* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns –1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns

either the (non-negative) character or -1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup,* (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg).* Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers

that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread,* but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that

the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTED
This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRIThis bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

1. Overview

In making available the generalized terminal descriptions in **/etc/termcap**, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*:

screen: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file <**curses.h**> needs to include <**sgtty.h**>, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [*flags*] file ... **−lcurses −ltermlib**

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does* ***not*** *change the terminal.* Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

---

[1] The screen package also uses the Standard I/O library, so <**curses.h**> includes <**stdio.h**>. It is redundant (but harmless) for the programmer to do it, too.

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (e.g., *reg int i;*)
bool	boolean type, actually a "char" (e.g., *bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must ***always*** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *crmode()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid

of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *getttmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions

are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **eye** and **vi**[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing **cat**(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **/etc/termcap** database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by **gtty**(2) and **stty**(2) The second, *setterm()*, a larger task by reading in the descriptions from the **/etc/termcap** database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
        gettmode();
        if (sp=getenv("TERM"))
                setterm(sp);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a **gtty**(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **/etc/termcap** associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see **termcap**(3)) to put out a string). It is these things which *endwin()* undoes.

---

[3] **Eye** actually uses these functions, **vi** does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as **rocket** and **gun**.

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the **/etc/termcap** database to describe each capability. See Appendix A for a complete list of those read, and **termcap**(5) for a full description.

[7] *isatty()* is defined in the default C library function routines. It does a **gtty**(2) on the descriptor and checks the return value.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the **termlib**(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS−1, LINES−1, 0)
```

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "**w**" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

addch(ch) †
char ch;

waddch(win, ch)
*WINDOW *win;*
char ch;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str) †
*char *str;*

---

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you

waddstr(win, str)
WINDOW **win;*
char **str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
WINDOW **win;*
char *vert, hor;*

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
WINDOW **win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
WINDOW **scr;*
bool *boolf;*

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol() †

---

have the social life of a kumquat.

wclrtoeol(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)
*WINDOW *win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
*WINDOW *win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged. This returns ERR if it would cause the screen to scroll illegally.

move(y, x) †
int *y, x;*

wmove(win, y, x)
WINDOW **win;*
int *y, x;*

Change the current (y, x) co-ordinates of the window to (*y, x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
WINDOW **win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
WINDOW **win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
char **fmt;*

wprintw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
WINDOW **win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen

to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
*char *str;*

wgetstr(win, str)
WINDOW **win;*
char **str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

delwin(win)
WINDOW **win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

getyx(win, y, x) †
WINDOW **win;*
int *y, x;*

---

[9] UNIX is a trademark of Bell Laboratories.

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
*WINDOW *win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

leaveok(win, boolf) †
*WINDOW *win;*
bool boolf;

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
*char *termbuf, *name;*

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent()*.

mvwin(win, y, x)
*WINDOW *win;*
int y, x;

Move the home position of the window *win* from its current starting coordinates to (*y*, *x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything.

*WINDOW **
newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES − begin_y*) or (*COLS − begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <**RETURN**> to <**LINE-FEED**>. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
WINDOW **win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW *
subwin(win, lines, cols, begin_y, begin_x)
WINDOW **win;*
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES − begin_y*) or (*COLS − begin_x*) respectively.

unctrl(ch) †
char *ch;*

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their uppercase equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

tstp()

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode

variables set by *setterm()*

Type	Name	Pad	Description
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence[10]
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with *X* are reserved for severely nauseous glitches

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

[10] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm()*

1.
The WINDOW structure

The WINDOW structure is defined as follows:

```
# define          WINDOW struct _win_st

struct _win_st {
        short     _cury, _curx;
        short     _maxy, _maxx;
        short     _begy, _begx;
        short     _flags;
        bool      _clear;
        bool      _leave;
        bool      _scroll;
        char      ** _y;
        short     * _firstch;
        short     * _lastch;
};

# define          _SUBWIN         01
# define          _ENDLINE        02
# define          _FULLWIN        04
# define          _SCROLLWIN      010
# define          _STANDOUT       0200
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury, _curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy, _begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line.

_flags can have one or more values or'd into it. **_SUBWIN** means that the window is a subwindow, which indicates to *delwin()* that the space for the lines is not to be freed. **_ENDLINE** says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_STANDOUT** says that all characters added to the screen are in standout mode.

---

[11] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

struct locs {
        char    y, x;
};

typedef struct locs             LOCS;

LOCS    Layout[NCOLS * NLINES];     /* current board layout */

int     Pattern,                    /* current pattern number */
        Numstars;                   /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
```

```
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);

        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int     y, x; {

        switch (Pattern) {
```

```
            case 0:             /* alternating lines */
                return !(y & 01);
            case 1:             /* box */
                if (x >= LINES && y >= NCOLS)
                    return FALSE;
                if (y < 3 | y >= NLINES - 3)
                    return TRUE;
                return (x < 3 | x >= NCOLS - 3);
            case 2:             /* holy pattern! */
                return ((x + y) & 01);
            case 3:             /* bar across center */
                return (y >= 9 && y <= 15);
        }
        /* NOTREACHED */
}

puton(ch)
reg char            ch; {

        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include           <curses.h>
# include           <signal.h>

/*
 *          Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */
```

```
struct lst_st {                         /* linked list element */
        int             y, x;           /* (y, x) position of piece */
        struct lst_st   *next, *last;   /* doubly linked */
};

typedef struct lst_st   LIST;

LIST    *Head;                          /* head of linked list */

main(ac, av)
int     ac;
char    *av[]; {

        int     die();

        evalargs(ac, av);               /* evaluate arguments */

        initscr();                      /* initialize screen package */
        signal(SIGINT, die);            /* set to restore tty stats */
        crmode();                       /* set for char-by-char */
        noecho();                       /* input */
        nonl();                         /* for optimization */

        getstart();                     /* get starting position */
        for (;;) {
                prboard();              /* print out current board */
                update();               /* update board position */
        }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die() {

        signal(SIGINT, SIG_IGN);        /* ignore rubouts'*/
        mvcur(0, COLS-1, LINES-1, 0);   /* go to bottom of screen */
        endwin();                       /* set terminal to initial state */
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart() {
```

```
        reg char        c;
        reg int         x, y;

        box(stdscr, '|', '_');                  /* box in the screen */
        move(1, 1);                             /* move to upper left corner */

        do {
                refresh();                      /* print current position */
                if ((c=getch()) == 'q')
                        break;
                switch (c) {
                 case 'u':
                 case 'i':
                 case 'o':
                 case 'j':
                 case 'l':
                 case 'm':
                 case ',':
                 case '.':
                        adjustyx(c);
                        break;
                 case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                 case 'x':
                        addch('X');
                        break;
                 case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                       /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}
```

```
/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;

        erase();                                /* clear out last position */
        box(stdscr, '|', '_');                  /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

        reg char        *sp;
        char            *getenv();
        int             _putchar(), die();

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
            gettmode();
            if (sp=getenv("TERM"))
                    setterm(sp);
            signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
```

```
        tputs(CL, NLINES, _putchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * _putchar defined for tputs() (and _puts())
 */
_putchar(c)
reg char        c; {

        putchar(c);
}

puton(ch)
char    ch; {

        static int      lasty, lastx;
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
                        /* prevent scrolling */
                if (!AM | (lp->y < NLINES - 1 | lp->x < NCOLS - 1)) {
                        mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                        lasty = lp->y;
                        if ((lastx = lp->x + 1) >= NCOLS)
                                if (AM) {
                                        lastx = 0;
                                        lasty++;
                                }
                                else
                                        lastx = NCOLS - 1;
                }
}
```

4.2BSD Line Printer Spooler Manual

Revised July 27, 1983

Ralph Campbell

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

1. Overview

The line printer system supports:

- multiple printers,
- multiple spooling queues,
- both local and remote printers, and
- printers attached via serial lines which require line initialization such as the baud rate.

Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

The line printer system consists mainly of the following files and commands:

/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, does all the real work
/usr/ucb/lpr	program to enter a job in a printer queue
/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/etc/lpc	program to administer printers and spooling queues
/dev/printer	socket on which lpd listens

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page entry *printcap*(5) provides the ultimate definition of the format of this data base, as well as indicating default values for important items such as the directory in which spooling is performed. This document highlights the important information which may be placed *printcap*.

2. Commands

2.1. lpd – line printer dameon

The program *lpd*(8), usually invoked at boot time from the /etc/rc file, acts as a master server for coordinating and controlling the spooling queues configured in the printcap file. When *lpd* is started it makes a single pass through the *printcap* database restarting any printers which have jobs. In normal operation *lpd* listens for service requests on multiple

UNIX is a trademark of Bell Laboratories.

sockets, one in the UNIX domain (named "/dev/printer") for local requests, and one in the Internet domain (under the "printer" service specification) for requests for printer access from off machine; see *socket*(2) and *services*(5) for more information on sockets and service specifications, respectively. *Lpd* spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with *lpd* using a simple transaction oriented protocol. Authentication of remote clients is done based on the "privilege port" scheme employed by *rshd*(8C) and *rcmd*(3X). The following table shows the requests understood by *lpd*. In each request the first byte indicates the "meaning" of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

Request	Interpretation
^Aprinter \n	check the queue for jobs and print any found
^Bprinter \n	receive and queue a job from another machine
^Cprinter [users ...] [jobs ...] \n	return short list of current queue state
^Dprinter [users ...] [jobs ...] \n	return long list of current queue state
^Eprinter person [users ...] [jobs ...] \n	remove jobs from a queue

The *lpr*(1) command is used by users to enter a print job in a local queue and to notify the local *lpd* that there are new jobs in the spooling area. *Lpd* either schedules the job to be printed locally, or in the case of remote printing, attempts to forward the job to the appropriate machine. If the printer cannot be opened or the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

2.2. lpq – show line printer queue

The *lpq*(1) program works recursively backwards displaying the queue of the machine with the printer and then the queue(s) of the machine(s) that lead to it. *Lpq* has two forms of output: in the default, short, format it gives a single line of output per queued job; in the long format it shows the list of files, and their sizes, which comprise a job.

2.3. lprm – remove jobs from a queue

The *lprm*(1) command deletes jobs from a spooling queue. If necessary, *lprm* will first kill off a running daemon which is servicing the queue, restarting it after the required files are removed. When removing jobs destined for a remote printer, *lprm* acts similarly to *lpq* except it first checks locally for jobs to remove and then tries to remove files in queues off-machine.

2.4. lpc – line printer control program

The *lpc*(8) program is used by the system administrator to control the operation of the line printer system. For each line printer configured in /etc/printcap, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer dameons.

3. Access control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The strategy used to maintain protected spooling areas is as follows:

- The spooling area is writable only by a *daemon* user and *spooling* group.
- The *lpr* program runs setuid *root* and setgid *spooling*. The *root* access is used to read any file required, verifying accessibility with an *access*(2) call. The group ID is used in setting

up proper ownership of files in the spooling area for *lprm*.

- Control files in a spooling area are made with *daemon* ownership and group ownership *spooling*. Their mode is 0660. This insures control files are not modified by a user and that no user can remove files except through *lprm*.
- The spooling programs, *lpd*, *lpq*, and *lprm* run setuid *root* and setgid *spooling* to access spool files and printers.
- The printer server, *lpd*, uses the same verification procedures as *rshd*(8C) in authenticating remote clients. The host on which a client resides must be present in the file /etc/hosts.equiv, used to create clusters of machines under a single administration.

In practice, none of *lpd*, *lpq*, or *lprm* would have to run as user *root* if remote spooling were not supported. In previous incarnations of the printer system *lpd* ran setuid *daemon*, setgid *spooling*, and *lpq* and *lprm* ran setgid *spooling*.

4. Setting up

The 4.2BSD release comes with the necessary programs installed and with the default line printer queue created. If the system must be modified, the makefile in the directory /usr/src/usr.lib/lpr should be used in recompiling and reinstalling the necessary programs.

The real work in setting up is to create the *printcap* file and any printer filters for printers not supported in the distribution system.

4.1. Creating a printcap file

The *printcap* database contains one or more entries per printer. A printer should have a separate spooling directory; otherwise, jobs will be printed on different printers depending on which printer daemon starts first. This section describes how to create entries for printers which do not conform to the default printer description (an LP-11 style interface to a standard, band printer).

4.1.1. Printers on serial lines

When a printer is connected via a serial communication line it must have the proper baud rate and terminal modes set. The following example is for a DecWriter III printer connected locally via a 1200 baud serial line.

```
lp|LA-180 DecWriter III:\
        :lp=/dev/lp:br#1200:fs#06320:\
        :tr=\f:of=/usr/lib/lpf:lf=/usr/adm/lpd-errs:
```

The **lp** entry specifies the file name to open for output. In this case it could be left out since "/dev/lp" is the default. The **br** entry sets the baud rate for the tty line and the **fs** entry sets CRMOD, no parity, and XTABS (see *tty*(4)). The **tr** entry indicates a form-feed should be printed when the queue empties so the paper can be torn off without turning the printer off-line and pressing form feed. The **of** entry specifies the filter program *lpf* should be used for printing the files; more will be said about filters later. The last entry causes errors to be written to the file "/usr/adm/lpd-errs" instead of the console.

4.1.2. Remote printers

Printers which reside on remote hosts should have an empty **lp** entry. For example, the following printcap entry would send output to the printer named "lp" on the machine "ucbvax".

```
lp|default line printer:\
        :lp=:rm=ucbvax:rp=lp:sd=/usr/spool/vaxlpd:
```

The **rm** entry is the name of the remote machine to connect to; this name must appear in the /etc/hosts database, see *hosts*(5). The **rp** capability indicates the name of the printer on the

remote machine is "lp"; in this case it could be left out since this is the default value. The **sd** entry specifies "/usr/spool/vaxlpd" as the spooling directory instead of the default value of "/usr/spool/lpd".

4.2. Output filters

Filters are used to handle device dependencies and to perform accounting functions. The output filter **of** is used to filter text data to the printer device when accounting is not used or when all text data must be passed through a filter. It is not intended to perform accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owners' login name, identifying the begining and ending of jobs, etc. The other filters (if specified) are started for each file printed and perform accounting if there is an **af** entry. If entries for both **of** and one of the other filters are specified, the output filter is used only to print the banner page; it is then stopped to allow other filters access to the printer. An example of a printer which requires output filters is the Benson-Varian.

```
va|varian|Benson-Varian: \
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf: \
        :tf=/usr/lib/rvcat:mx#2000:pl#58:tr=\f:
```

The **tf** entry specifies "/usr/lib/rvcat" as the filter to be used in printing *troff*(1) output. This filter is needed to set the device into print mode for text, and plot mode for printing *troff* files and raster images (see *va*(4V)). Note that the page length is set to 58 lines by the **pl** entry for 8.5" by 11" fan-fold paper. To enable accounting, the varian entry would be augmented with an **af** filter as shown below.

```
va|varian|Benson-Varian: \
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf: \
        :if=/usr/lib/vpf:tf=/usr/lib/rvcat:af=/usr/adm/vaacct: \
        :mx#2000:pl#58:tr=\f:
```

5. Output filter specifications

The filters supplied with 4.2BSD handle printing and accounting for most common line printers, the Benson-Varian, the wide (36") and narrow (11") Versatec printer/plotters. For other devices or accounting methods, it may be necessary to create a new filter.

Filters are spawned by *lpd* with their standard input the data to be printed, and standard output the printer. The standard error is attached to the **lf** file for logging errors. A filter must return a 0 exit code if there were no errors, 1 if the job should be reprinted, and 2 if the job should be thrown away. When *lprm* sends a kill signal to the *lpd* process controlling printing, it sends a SIGINT signal to all filters and descendents of filters. This signal can be trapped by filters which need to perform cleanup operations such as deleting temporary files.

Arguments passed to a filter depend on its type. The **of** filter is called with the following arguments.

ofiler **−w**width **−l**length

The *width* and *length* values come from the **pw** and **pl** entries in the printcap database. The **if** filter is passed the following parameters.

filter [**−c**] **−w**width **−l**length **−i**indent **−n** login **−h** host accounting file

The **−c** flag is optional, and only supplied when control characters are to be passed uninterpreted to the printer (when the **−l** option of *lpr* is used to print the file). The **−w** and **−l** parameters are the same as for the **of** filter. The **−n** and **−h** parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from *printcap*.

All other filters are called with the following arguments:

filter **−x**width **−y**length **−n** login **−h** host accounting file

The **−x** and **−y** options specify the horizontal and vertical page size in pixels (from the **px** and **py** entries in the printcap file). The rest of the arguments are the same as for the **if** filter.

6. Line printer Administration

The *lpc* program provides local control over line printer activity. The major commands and their intended use will be described. The command format and remaining commands are described in *lpc*(8).

abort and **start**

Abort terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by *lpr*). This is normally used to forciblly restart a hung line printer daemon (i.e., *lpq* reports that there is a daemon present but nothing is happening). It does not remove any jobs from the queue (use the *lprm* command instead). *Start* enables printing and requests *lpd* to start printing jobs.

enable and **disable**

Enable and *disable* allow spooling in the local queue to be turned on/off. This will allow/prevent *lpr* from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the *root* user can still use *lpr* to put jobs in the queue but no one else can. The other main use is to prevent users from putting jobs in the queue when the printer is expected to be unavailable for a long time.

restart

Restart allows ordinary users to restart printer daemons when *lpq* reports that there is no daemon present.

stop

Stop is used to halt a spooling daemon after the current job completes; this also disables printing. This is a clean way to shutdown a printer in order to perform maintenence, etc. Note that users can still enter jobs in a spool queue while a printer is *stopped.*

topq

Topq places jobs at the top of a printer queue. This can be used to reorder high priority jobs since *lpr* only only provides first-come-first-serve ordering of jobs.

7. Troubleshooting

There are a number of messages which may be generated by the the line printer system. This section categorizes the most common and explains the cause for their generation. Where the message indicates a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer. This would be one of the names from the *printcap* database.

7.1. LPR

lpr: *printer*: **unknown printer**

The *printer* was not found in the *printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the /etc/printcap file.

lpr: *printer*: **jobs queued, but cannot start daemon.**

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket /dev/printer to be sure it still exists (if it does not exist, there is no *lpd* process running). Use

```
% ps ax | fgrep lpd
```

to get a list of process identifiers of running lpd's. The *lpd* to kill is the one which is not listed in any of the "lock" files (the lock file is contained in the spool directory of each printer). Kill the master daemon using the following command.

```
% kill pid
```

Then remove /dev/printer and restart the daemon (and printer) with the following commands.

```
% rm /dev/printer
% /usr/lib/lpd
```

Another possibility is that the *lpr* program is not setuid *root*, setgid *spooling*. This can be checked with

```
% ls -lg /usr/ucb/lpr
```

lpr: *printer*: **printer queue is disabled**

This means the queue was turned off with

```
% lpc disable printer
```

to prevent *lpr* from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with *lpc*.

7.2. LPQ

waiting for *printer* **to become ready (offline ?)**

The printer device could not be opened by the daemon. This can happen for a number of reasons, the most common being that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, etc. The actual reason is dependent on the meaning of error codes returned by system device driver. Not all printers supply sufficient information to distinguish when a printer is off-line or having trouble (e.g. a printer connected through a serial line). Another possible cause of this message is some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with *lpc*.

printer **is ready and printing**

The *lpq* program checks to see if a daemon process exists for *printer* and prints the file *status*. If the daemon is hung, a super user can use *lpc* to abort the current daemon and start a new one.

waiting for *host* **to come up**

This indicates there is a daemon trying to connect to the remote machine named *host* in order to send the files in the local queue. If the remote machine is up, *lpd* on the remote machine is probably dead or hung and should be restarted as mentioned for *lpr*.

sending to *host*

The files should be in the process of being transferred to the remote *host*. If not, the local daemon should be aborted and started with *lpc*.

Warning: *printer* **is down**

The printer has been marked as being unavailable with *lpc*.

Warning: no daemon present

The *lpd* process overseeing the spooling queue, as indicated in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an *lpd*, use

```
% lpc restart printer
```

7.3. LPRM

lprm: *printer*: **cannot restart printer daemon**

This case is the same as when *lpr* prints that the daemon cannot be started.

7.4. LPD

The *lpd* program can write many different messages to the error log file (the file specified in the **lf** entry in *printcap*). Most of these messages are about files which can not be opened and usually indicate the *printcap* file or the protection modes of the files are not correct. Files may also be inaccessible if people manually manipulate the line printer system (i.e. they bypass the *lpr* program).

In addition to messages generated by *lpd*, any of the filters that *lpd* spawns may also log messages to this file.

7.5. LPC

could't start printer

This case is the same as when *lpr* reports that the daemon cannot be started.

cannot examine spool directory

Error messages beginning with "cannot ..." are usually due to incorrect ownership and/or protection mode of the lock file, spooling directory or the *lpc* program.

UNIX MASTER INDEX

The **UNIX Master Index** is a cumulative index; it brings together the indexes of all the UNIX volumes. The Master Index appears at the end of each volume.

Each entry is followed by one or more shortened volume titles, indicating the volumes in which the topic is discussed and the pages containing the information. The volumes and their shortened titles are shown in the following table:

Shortened	**Volume Title**
General use	*GEN*
Programming	*PGM*
System manager	*SYS*

If a topic is discussed in two or more volumes, the shortened volume names are presented in alphabetical order. For example, an entry in the Master Index might appear in the following way:

ed line editor

description, *GEN* 4–8 to 4–9, *SYS* 4–6

ed__.hup file

saving text, *GEN* 2–6

This entry indicates that a description of the ed line editor can be found on pages 4–8 through 4–9 of the *GEN* volume and page 4–6 of the *SYS* volume. The ed__.hup file is discussed on page 3–43 of the *GEN* volume.

ACRONYMS AND MNEMONICS

The acronym (or mnemonic) is the preferred entry. The acronym is cross-referred from the complete form.

DEFINITIONS

Defined terms and glossary terms are indexed.

HOMONYMS

Things of the same name but different meaning are followed by a descriptive word or by an abbreviation in parentheses.

KEYS FOR EXAMPLES, FIGURES, TABLES, AND FOOTNOTES

Page references for example, figure, and table index entries are keyed. Example:

Example	4-13E
Figure	4-13F
Table	4-13T
Footnote	4-13n

NONALPHABETIC CHARACTERS

Entries containing leading nonalphabetic characters (symbols, numbers, and punctuation) are placed at the beginning of the index. Nonalphabetic characters within index entries are sorted before alphabetic characters.

Nonalphabetic characters that serve as terms are indexed in a spelled-out form whenever possible.

INDEX

! command (DC)
descripton, *GEN* 2-58
! command (ed)
escaping to use UNIX command, *GEN* 3-51E
! command (ex)
description, *GEN* 3-95
! command (Mail)
marking commands for the shell, *GEN* 2-28
! escape (Mail)
description, *GEN* 2-25
$ character (ed)
printing last line, *GEN* 3-28
% command (DC)
descripton, *GEN* 2-57
% prompt
defined, *GEN* 3-5
& command (ex)
description, *GEN* 3-96
+ command (DC)
descripton, *GEN* 2-57
- command (DC)
descripton, *GEN* 2-57
- command (Mail)
printing previous message, *GEN* 2-28
.. file
defined, *GEN* 4-63
/etc/passwd file
defined, *GEN* 4-66
/etc/rc command file
starting network servers, *SYS* 5-49
/sys directory
contents, *SYS* 5-36T
/sys/sys directory
file prefixes, *SYS* 5-36T
/usr/spool/mail directory
system mailbox and, *GEN* 2-17
0 command
defined, *GEN* 5-88
0 command (troff)
right-justifying digits, *GEN* 5-87
0 macro (me)
specifying section titles for contents, *GEN* 5-41
1822 interface
See imp network interface driver
1c command (me)
defined, *GEN* 5-43
returning one-column format, *GEN* 5-35
1C command (ms)
returning one-column format, *GEN* 5-6
2c command (me)
defined, *GEN* 5-43
specifying two-column format, *GEN* 5-35
2C command (ms)
specifying two-column format, *GEN* 5-6

3Com Ethernet controller
See ec network interface driver
4.2BSD file system
file set, *SYS* 5-32T
4.2BSD Interprocess Communication Primer
See also Interprocess communication
4.2BSD Interprocess Communication Primer, *SYS* 3-5 to 3-28
4.2BSD Line Printer Spooler Manual, *PGM* 4-99 to 4-105
See also Line printer spooling system (4.2BSD)
4.2BSD system
4.1BSD files and, *SYS* 5-32 to 5-34
4.1BSD language processors and, *SYS* 5-34
adding device drivers, *SYS* 5-88
adding users, *SYS* 5-43
bug fixes and changes, *SYS* 1-3 to 1-21
changes to the kernel, *SYS* 5-3 to 5-15
configuring for networking support, *SYS* 5-47 to 5-51
configuring multiple networks, *SYS* 5-48
creating boot floppy, *SYS* 5-35
disk space and, *SYS* 5-18
distribution format, *SYS* 5-18
hardware supported, *SYS* 5-17
installing on VAX/VMS, *SYS* 5-17 to 5-71
making boot cassette, *SYS* 5-35
setting up, *SYS* 5-35 to 5-46
source directory organization, *SYS* 5-89T
system manual, *PGM* 4-15 to 4-52
tailoring to your site, *SYS* 5-43
upgrading, *SYS* 5-32 to 5-34
4.2BSD System Manual, *PGM* 4-15 to 4-52
: command (DC)
description, *GEN* 2-63
: escape (Mail)
description, *GEN* 2-25
; command (DC)
description, *GEN* 2-63
< symbol
meaning, *GEN* 2-10
= command (sed)
defined, *GEN* 3-114
> symbol
meaning, *GEN* 2-10
? escape (Mail)
description, *GEN* 2-26
[...]
pattern-matching and, *GEN* 2-8
*** command (troff)**
entering comments in macros, *GEN* 5-89
_exit function
description, *PGM* 1-8

A

a command (ed)
defined, *GEN* 3-34
using, *GEN* 3-25 to 3-26
a command (edit)
entering, *GEN* 3-6E
a command (ex)
description, *GEN* 3-88
A command (me)
defined, *GEN* 5-46
a command (sed)
See also i command (sed)
defined, *GEN* 3-108
A command (vi)
defined, *GEN* 3-78
a command (vi)
defined, *GEN* 3-80
a option (hunt)
defined, *GEN* 5-148
a option (inv)
defined, *GEN* 5-147
a option (troff)
defined, *GEN* 5-50
a.out file
as assembler and, *GEN* 6-53
defined, *GEN* 4-63
aardvark game
4.2BSD and, *SYS* 1-17
ab command (ex)
See also una command (ex)
description, *GEN* 3-87
AB command (me)
defined, *GEN* 5-46
AB command (ms)
entering abstract in text, *GEN* 5-5
ab command (nroff/troff)
message output, *GEN* 5-81
abbreviate command (ex)
See ab command (ex)

abort command (lpc)
description, *PGM* 4–103
Absolute pathname
See also Relative pathname
defined, *GEN* 4–63
description, *GEN* 4–33
Abstract
entering with –ms, *GEN* 5–5
ac command (me)
defined, *GEN* 5–46
ACC LH/DH IMP interface
See acc network driver
acc network driver
4.2BSD improvement, *SYS* 1–15
Accent
creating with troff, *GEN* 5–88E
entering with –ms, *GEN* 5–9
new in –ms, *GEN* 5–19
access system call
4.2BSD improvement, *SYS* 1–10
ACM (Association for Computing Machinery)
formatting papers for, *GEN* 5–46
acommute routine
operators and, *PGM* 2–67 to 2–68
Action statement (awk)
description, *PGM* 3–7 to 3–9
Active system
defined, *SYS* 5–123
Acute accent
See Metacharacters
ad command (nroff/troff)
defined, *GEN* 5–61
j register and, *GEN* 5–81
ad driver
4.2BSD improvement, *SYS* 1–15
ad.c device driver
4.2BSD improvement, *SYS* 5–12
ADB debugging program
4.2BSD improvement, *SYS* 1–5
C and, *GEN* 2–15
description, *PGM* 3–51 to 3–77
addbib utility
See also refer
description, *SYS* 1–5
addch routine
defined, *PGM* 4–80
Addition
DC and, *GEN* 2–60
Additive operator
description, *GEN* 2–53
Address (edit)
defined for buffer line, *GEN* 3–18
Address (sed)
description, *GEN* 3–107 to 3–108
Address Resolution Protocol
See arp driver
addstr routine
defined, *PGM* 4–81
Advisory lock
compared to hard lock, *SYS* 1–33
AE command (ms)
TL command and, *GEN* 5–6
af command (nroff/troff)
defined, *GEN* 5–66
Aho, A.V., & others
awk programming language, *PGM* 3–5 to 3–12
AI command (ms)
formatting author's institution name, *GEN* 5–5
Alias
defined, *GEN* 2–21, 2–38, 4–63
removing from shell, *GEN* 4–52
specifying, *GEN* 2–21
alias command (C shell)
See also unalias command (C shell)
displaying aliases, *GEN* 4–50E
alias command (Mail)
See also alternates command (Mail)
See also metoo option
defining an alias, *GEN* 2–21
description, *GEN* 2–29
restriction, *GEN* 2–21
alias facility
shell command files and, *GEN* 4–43
startup and, *GEN* 4–44
uses for, *GEN* 4–43 to 4–44
aliens game
distribution and, *SYS* 1–17
Allman, E.
–Me Reference Manual, *GEN* 5–39 to 5–48
introduction to SCCS, *PGM* 3–23 to 3–37
sendmail, *SYS* 3–59 to 3–71
Sendmail Installation and Operation Guide, *SYS* 2–27 to 2–60
writing papers with nroff using –me, *GEN* 5–21 to 5–38
Allocator
description, *GEN* 2–59 to 2–60
design rationale, *GEN* 2–63

ALT key
See ESCAPE key
alternates command (Mail)
description, *GEN* 2-29
am command (nroff/troff)
defined, *GEN* 5-64
AM macro
diacritical marks and, *GEN* 5-19
Ampersand character (C shell)
background jobs and, *GEN* 4-45
routing output, *GEN* 4-44
Ampersand character (ed)
meaning, *GEN* 3-42
printing, *GEN* 3-42
s command and, *GEN* 3-33 to 3-34
turning off, *GEN* 3-34
uses, *GEN* 3-42
Ampersand character (edit)
repeating s command, *GEN* 3-20
Ampersand character (shell)
multitasking and, *GEN* 1-29
ANAME operator (C compiler)
defined, *PGM* 2-65
ANSI Standard X3.9 1978
exceptions to, *PGM* 2-88
extensions, *PGM* 2-82 to 2-83
append command (ed)
See a command (ed)
append command (edit)
See a command (edit)
append command (ex)
See a command (ex)
Append mode
See Input mode
append option (Mail)
defined, *GEN* 2-34
Appendix
specifying page numbers, *GEN* 5-46
apply program
description, *SYS* 1-5
ar
4.2BSD improvement, *SYS* 1-5
ar command (me)
defined, *GEN* 5-44
Arabic number
setting page number, *GEN* 5-44
arff program
4.2BSD improvement, *SYS* 1-18
args command (ex)
description, *GEN* 3-88
Argument (C shell)
defined, *GEN* 4-63
Argument (C shell) (Cont.)
expanding, *GEN* 4-60 to 4-61
Argument (nroff)
defined, *GEN* 5-21
argv variable (C shell)
defined, *GEN* 4-63
script files and, *GEN* 4-53
Arithmetic expression (troff)
entering, *GEN* 5-92
Arithmetic language
See BC language
Arnold, K.C.R.C.
Screen package, *PGM* 4-75 to 4-98
Arnold, K.C.R.C., & Toy, M.C.
guide to the dungeons of doom, *GEN* 6-17 to 6-25
arp driver
4.2BSD improvement, *SYS* 1-15
ARPA File Transfer Protocol
ftp program and, *SYS* 1-6
ARPA Telnet protocol
See telnet program
ARPANET
sending mail to, *GEN* 2-26
UUCP network and, *GEN* 2-26
Array (awk)
description, *PGM* 3-9
Array element
defined, *GEN* 2-51
Array identifier
description, *GEN* 2-50
as assembler
command line format, *GEN* 6-53E
defined, *GEN* 6-53
errors, *GEN* 6-64
reference manual, *GEN* 6-53 to 6-64, *PGM* 4-53 to 4-65
segment types, *GEN* 6-54
as command (nroff/troff)
defined, *GEN* 5-64
ask option (Mail)
defined, *GEN* 2-34
prompting for subject header, *GEN* 2-20
setting, *GEN* 2-20
askcc option (Mail)
defined, *GEN* 2-34
asm.sed file
4.2BSD improvement, *SYS* 5-13
Assembler
replacing, *SYS* 5-118
Assignment operator
description, *GEN* 2-53

Assignment statement (as)
defined, *GEN* 6-56
Assignment statement (BC)
value and, *GEN* 2-48
Association for Computing Machinery
See ACM
Asterisk character
dot character and, *GEN* 3-40
ed and, *GEN* 3-33
printing multiple files, *GEN* 2-8
shell and, *GEN* 4-33
turning off, *GEN* 2-8
uses, *GEN* 3-40 to 3-41
zero and, *GEN* 3-41
Asymmetric protocol
defined, *SYS* 3-17
At sign
See also CTRL-H
See also u command (edit)
deleting a line, *GEN* 3-8E
entering in text, *GEN* 2-4
erasing characters on input line, *GEN* 2-4
printing, *GEN* 3-39
AU command (ms)
formatting author's name in text, *GEN* 5-5
Author institution
formatting in text, *GEN* 5-5
Author name
formatting in text, *GEN* 5-5
Auto array
specifying, *GEN* 2-54
auto statement (BC)
forming, *GEN* 2-55
autoconf.c file
4.2BSD improvement, *SYS* 5-13
Autoconfiguration
building systems with config, *SYS* 5-73 to 5-105
hardware devices and, *SYS* 5-75
requirements for VAX/VMS, *SYS* 5-95
autoindent option (ex)
description, *GEN* 3-97
autoindent option (vi)
enabling, *GEN* 3-67
lisp and, *GEN* 3-68
using, *GEN* 3-73
autoprint option (ex)
description, *GEN* 3-98
autoprint option (Mail)
defined, *GEN* 2-34
autowrite option (ex)
description, *GEN* 3-98
awk programming language
command line format, *PGM* 3-5
compared with grep, *PGM* 3-5
defined, *GEN* 2-13, *PGM* 3-5
description, *PGM* 3-5 to 3-12
design, *PGM* 3-9 to 3-10
execution time compared, *PGM* 3-12T
fields, *PGM* 3-5
implementation, *PGM* 3-10
printing output, *PGM* 3-6
program structure, *PGM* 3-5
records, *PGM* 3-5
uses, *PGM* 3-10
variables, *PGM* 3-8

B

B command (me)
defined, *GEN* 5-46
specifying bibliographic section, *GEN* 5-33
b command (me)
See also rh command (me)
defined, *GEN* 5-42, 5-44
entering, *GEN* 5-26
specifying bold font, *GEN* 5-36
specifying fill mode, *GEN* 5-26
B command (ms)
specifying boldface, *GEN* 5-8
b command (sed)
defined, *GEN* 3-114
b command (troff)
creating large brackets, *GEN* 5-88E
B command (vi)
defined, *GEN* 3-78
b command (vi)
defined, *GEN* 3-80
B flag (tar)
reading block records, *SYS* 1-9
writing block records, *SYS* 1-9
b option (troff)
defined, *GEN* 5-50
B_CALL flag
4.2BSD improvement, *SYS* 5-6
ba command (me)
defined, *GEN* 5-45
backgammon game
See also teachgammon program
4.2BSD improvement, *SYS* 1-17

Background command (C shell)
defined, *GEN* 4-63
Background job
description, *GEN* 4-45 to 4-48
reading input from terminal, *GEN* 4-47E
suspending, *GEN* 4-46
Backslash character
erasing, *GEN* 2-4
Backslash character (ed)
context search and, *GEN* 3-43
restriction, *GEN* 3-33
searching for, *GEN* 3-39E
special characters and, *GEN* 3-39
Backslash character (troff)
translating for typesetter, *GEN* 5-86
Backus Functional Programming Language
See FP programming language
Bad block forwarding
support, *SYS* 1-18
bad144 program
4.2BSD improvement, *SYS* 1-18
Baden, S.
Berkeley FP User Manual, PGM 2-359 to 2-391
badsect program
See also fsck program
4.2BSD improvement, *SYS* 1-18
Base (BC)
See also ibase; obase
description, *GEN* 2-44 to 2-45
bc command (me)
defined, *GEN* 5-43
starting a column, *GEN* 5-35
BC language
C language and, *GEN* 2-43
defined, *GEN* 2-43
description, *GEN* 2-43 to 2-55
displaying library of math functions, *GEN* 2-49
output bases and, *GEN* 2-45
restriction, *GEN* 2-43
simple computations and, *GEN* 2-43 to 2-44
subscript restriction, *GEN* 2-46
BC program
exiting, *GEN* 2-49
bcmp library routine
4.2BSD improvement, *SYS* 1-14
bcopy library routine
4.2BSD improvement, *SYS* 1-14
bd command (troff)
defined, *GEN* 5-59
BDATA operator (C compiler)
defined, *PGM* 2-64
beautify option (ex)
description, *GEN* 3-98
BEGIN/END pattern
description, *PGM* 3-6
Bell character
printing, *GEN* 3-37
Benson-Varian printer
output filters and, *PGM* 4-102
Berkeley font catalogue, *GEN* 6-27 to 6-51
Berkeley FP User's Manual, PGM 2-359 to 2-391
See also FP programming language
Berkeley network
See Berknet
Berkeley Pascal programming language
user's manual, *PGM* 2-159 to 2-209
Berkeley Pascal User Manual
***See also* Pascal programming language**
Berkeley Pascal User Manual, PGM 2-159 to 2-209
Berkeley system
See UNIX Operating System
Berkeley VAX/UNIX Assembler Reference Manual, PGM 4-53 to 4-65
See also as assembler
Berknet
sending mail to, *GEN* 2-27
bg command (C shell)
continuing background jobs, *GEN* 4-46E
defined, *GEN* 4-64
running suspended job in background, *GEN* 4-47
bi command (me)
defined, *GEN* 5-44
Bibliographic citations
formatting, *GEN* 2-13, 5-18, 5-33
specifying, *GEN* 5-34F
Bibliographic databases
See roffbib program, *SYS* 1-8
Bibliography
See Bibliographic citations
bin directory
defined, *GEN* 4-64

Binary date
Mail program and, *GEN* 2-37
Binary operator (C compiler)
description, *PGM* 2-66
Binary option (Mail)
See Option (Mail)
bind system call
assigning socket name, *SYS* 3-7E
binding names to sockets, *SYS* 1-10
specifying association, *SYS* 3-25
Bit mask
creating, *SYS* 3-11
bl command (me)
defined, *GEN* 5-44
Blau, R., & Joyce, J.
Edit tutorial, *GEN* 3-3 to 3-23
Block device
description, *SYS* 5-20
Block map
layout of blocks and fragments, *SYS* 1-27F
Block of text
footnotes and, *GEN* 5-36
indenting from left and right, *GEN* 5-86E
index entries and, *GEN* 5-36
keeping together in text, *GEN* 5-26
Block size
selecting, *SYS* 5-41
Boldface
entering, *GEN* 5-8
Bootstrap monitor
loading, *SYS* 5-65 to 5-68
Bootstrap procedure
booting from tape, *SYS* 5-22
description, *SYS* 5-22 to 5-31
details, *SYS* 5-59 to 5-64
messages about console bootstrap cassette, *SYS* 5-71
messages about the distributed console media, *SYS* 5-69
messages about the distributed system, *SYS* 5-70
Bootstrap program
4.2BSD improvement, *SYS* 5-15
loading, *SYS* 5-25
Bourne shell
background command, *GEN* 4-3E
changing prompt, *GEN* 4-6
command execution, *GEN* 4-23 to 4-24
command grammar, *GEN* 4-26
Bourne shell (Cont.)
command substitution and, *GEN* 4-18 to 4-20
command syntax, *GEN* 4-3
defined, *GEN* 4-3
description, *GEN* 4-3 to 4-27
error handling, *GEN* 4-21
error signals, *GEN* 4-21F
fault handling, *GEN* 4-21
group set and, *SYS* 1-8
invoking, *GEN* 4-24
prompt, *GEN* 4-6
redirecting input, *GEN* 4-4
redirecting output, *GEN* 4-3
Bourne, S.R.
introducing the UNIX shell, *GEN* 4-3 to 4-27
Bourne, S.R., & Maranzano, J.F.
ADB debugging program, *PGM* 3-51 to 3-77
Box (nroff/troff)
creating smallest, *GEN* 5-68
box routine
defined, *PGM* 4-81
Boxing
description, *GEN* 5-69
entering, *GEN* 5-8 to 5-9
bp command (me)
See also pa command (me)
specifying blank column, *GEN* 5-35
specifying page break, *GEN* 5-23
bp command (nroff/troff)
See also ns command (nroff/troff)
defined, *GEN* 5-59
br command (me)
starting a line, *GEN* 5-24
br command (nroff/troff)
defined, *GEN* 5-60
Braces
argument expansion and, *GEN* 4-60E
Braces (EQN)
typesetting in proper size, *GEN* 5-100E
Brackets (Bourne shell)
matching any single character, *GEN* 4-34
Brackets (DC)
placing character string on stack, *GEN* 2-58
Brackets (ed)
appearing in character class, *GEN* 3-41

Brackets (ed) (Cont.)
deleting line numbers, *GEN* 3-41, 3-41E
Brackets (EQN)
typesetting in proper size, *GEN* 5-100E
Brackets (Mail)
beginning a line with, *GEN* 2-26
Brackets (nroff/troff)
creating, *GEN* 5-88E
creating large, *GEN* 5-68
BRANCH operator (C compiler)
defined, *PGM* 2-65
Break
defined, *GEN* 5-22
space and, *GEN* 5-23
specifying, *GEN* 5-24
break command (C shell)
See also breaksw command (C shell)
csh script and, *GEN* 4-58
defined, *GEN* 4-64
break statement (awk)
defined, *PGM* 3-9
break statement (BC)
forming, *GEN* 2-54
breaksw command (C shell)
defined, *GEN* 4-64
exiting from switch statement, *GEN* 4-58
Broadcast message
sending, *SYS* 3-27E
Broadcast packet
See also Broadcast message
datagram sockets and, *SYS* 3-27
Broken bar
shell and, *GEN* 2-27
BSS operator (C compiler)
defined, *PGM* 2-64
bss segment (as)
See also Assignment statement (as)
See also Location counter (as)
description, *GEN* 6-54
bss statement
defined, *GEN* 6-59
bstring library
4.2BSD improvement, *SYS* 1-14
btlgammon game
See backgammon game
buf.h file
4.2BSD improvement, *SYS* 5-6
Buffer
defined, *GEN* 3-4
Buffer (Cont.)
ed and, *GEN* 3-25
writing part of, *GEN* 3-22
Buffer (nroff/troff)
flushing output buffer, *GEN* 5-73
Buffer (vi)
description, *GEN* 3-54
system commands and, *GEN* 3-68
types of, *GEN* 3-62
BUFSIZ
defined, *PGM* 1-21
bugfiler program
4.2BSD improvement, *SYS* 1-19
Built-in (M4)
See Command (M4)
built-in command (C shell)
defined, *GEN* 4-64
bx command (me)
boxing words, *GEN* 5-37
defined, *GEN* 5-44
byte statement (as)
defined, *GEN* 6-59
bzero library routine
4.2BSD improvement, *SYS* 1-14

C

C argument (nroff)
specifying, *GEN* 5-27
c command (DC)
descripton, *GEN* 2-58
c command (ed)
defined, *GEN* 3-34
using, *GEN* 3-31 to 3-32
c command (edit)
description, *GEN* 3-18
c command (ex)
description, *GEN* 3-88
C command (me)
defined, *GEN* 5-46
c command (me)
centering blocks of text, *GEN* 5-27
defined, *GEN* 5-43, 5-46
specifying a chapter without number, *GEN* 5-33
specifying chapters, *GEN* 5-33
c command (sed)
defined, *GEN* 3-109
C command (vi)
defined, *GEN* 3-78
C compiler
description, *PGM* 2-63 to 2-77
as programming tool, *GEN* 2-15

C compiler (Cont.)
replacing, *SYS* 5–118
c escape (Mail)
description, *GEN* 2–25
C flag (lint)
creating libraries from C source code, *SYS* 1–7
c flag (mkey)
specifying file of common words, *GEN* 5–147
C library
reinstalling, *SYS* 5–56E
c macro (me)
defined, *GEN* 5–46
c number register (nroff/troff)
defined, *GEN* 5–81
c operator (vi)
defined, *GEN* 3–80
C option (hunt)
defined, *GEN* 5–148
C option (tar)
forcing chdir operations in an operation, *SYS* 1–9
c option (uucp)
defined, *SYS* 5–132
C preprocessor
if statements and, *SYS* 1–5
line numbers and, *SYS* 1–5
C program
debugging, *PGM* 3–53 to 3–58
C programming language
See also M4 macro processor
CAI script for, *GEN* 6–7
command line format, *PGM* 1–3
computers supporting, *GEN* 2–15
programming in, *GEN* 2–14 to 2–15
reference manual, *PGM* 2–5 to 2–35
supporting programs, *GEN* 2–15
C Programming Language Reference Manual, The, *PGM* 2–5 to 2–35
See also C programming language
C shell
4.2BSD improvement, *SYS* 1–5
built-in commands, *GEN* 4–50 to 4–52
compared to other command interpreters, *GEN* 4–30
defined, *GEN* 4–29
details for terminal users, *GEN* 4–39 to 4–52
history list and, *GEN* 4–41
interrupts and, *GEN* 4–36
C shell (Cont.)
introduction, *GEN* 4–29 to 4–74
logging in, *GEN* 4–39
metacharacters and, *GEN* 4–32
overwriting files and, *GEN* 4–41
purpose of, *GEN* 4–29
using from the terminal, *GEN* 4–30 to 4–38
C shell variables
description, *GEN* 4–40 to 4–41
set command and, *GEN* 4–40E
c2 command (nroff/troff)
defined, *GEN* 5–67
CAI script, *GEN* 6–9E to 6–11E
description, *GEN* 6–6 to 6–7
prerequisites, *GEN* 6–6
prerequisites for the writer, *GEN* 6–8
types of, *GEN* 6–7
Campbell, R.
line printer spooling system (4.2BSD), *PGM* 4–99 to 4–105
CANBSIZ parameter
description, *SYS* 5–121
canfield game
See also cfscores program
4.2BSD improvement, *SYS* 1–17
Carbon copy
See CC: list
Caret
See Circumflex character (ed)
case branch
description, *GEN* 4–8 to 4–9
form of, *GEN* 4–8E
case command (C shell)
defined, *GEN* 4–64
cat command (C shell)
collecting files, *PGM* 1–5E
combining files, *GEN* 3–48, 3–48E
defined, *GEN* 4–64
listing system users, *GEN* 4–35E
printing files, *GEN* 2–7
printing merged files, *GEN* 2–11
printing pipeline information, *GEN* 2–11
terminating, *GEN* 4–36
cat program
See cat command (C shell)
CBRANCH operator (C compiler)
defined, *PGM* 2–66
cc
dbx and, *SYS* 1–5
cc command (nroff/troff)
defined, *GEN* 5–67

CC: list
See also askcc option
adding people to, *GEN* 2-25
cctab table
defined, *PGM* 2-68
cd command (C shell)
See also pushd command (C shell)
changing working directory, *GEN* 2-10
defined, *GEN* 4-64
description, *GEN* 2-29
working directory and, *GEN* 4-48
ce command (me)
entering, *GEN* 5-24
ce command (nroff/troff)
defined, *GEN* 5-61
Cedilla
See Metacharacters
Centering
blocks of text, *GEN* 5-27, 5-61
specifying, *GEN* 5-24
ch command (nroff/troff)
defined, *GEN* 5-65
Change bars (nroff/troff)
specifying, *GEN* 5-72
change command (ed)
See c command (ed)
change command (edit)
See c command (edit)
change command (ex)
See c command (ex)
change directory command
See cd command (C shell)
Changequote command (M4)
description, *PGM* 2-395E
Chapter
formatting, *GEN* 5-33
inserting in table of contents automatically, *GEN* 5-46
specifying page numbers, *GEN* 5-46
specifying without number, *GEN* 5-33
Chapter-oriented document
formatting, *GEN* 5-34F
Character class
circumflex within, *GEN* 3-42
defined, *GEN* 3-41
forming, *GEN* 3-33E
lowercase letters and, *GEN* 3-41
number ranges and, *GEN* 3-41
special characters and, *GEN* 3-41
specifying exceptions, *GEN* 3-42
uppercase letters and, *GEN* 3-41
chase game
obsolete, *SYS* 1-17
chdir command (C shell)
See cd command (C shell)
Cherry, L., & Morris, R.
BC and, *GEN* 2-43 to 2-55
DC and, *GEN* 2-57 to 2-64
Cherry, L.L., & Kernighan, B.W.
typesetting mathematics, *GEN* 5-97 to 5-104
Typesetting Mathematics - User's Guide, *GEN* 5-105 to 5-114
Cherry, L.L., & Vesterman, W.
style and diction programs, *GEN* 5-163 to 5-177
chfn
4.2BSD improvement, *SYS* 1-5
chgrp
4.2BSD improvement, *SYS* 1-5
ching game
4.2BSD improvement, *SYS* 1-17
chmod command (Bourne shell)
making a file executable, *GEN* 4-7E
marking executable files, *GEN* 2-12
chsh command (C shell)
defined, *GEN* 4-64
CHSHR file
incoming mail and, *GEN* 2-17
chshrc file
putting into effect before next login, *GEN* 4-51
Circle
See Metacharacters
Circumflex (edit)
searching and, *GEN* 3-20
Circumflex character (ed)
at beginning of line and, *GEN* 3-40
meaning, *GEN* 3-33
uses, *GEN* 3-40
Circumflex character (me)
See Metacharacters
clear routine
defined, *PGM* 4-81
clearok routine
defined, *PGM* 4-81
Client process
See also Server process
description, *SYS* 3-19
Clist segment
setting number, *SYS* 5-122

close function
description, *PGM* 1-11
clrtoeol routine
defined, *PGM* 4-81
cmp program
defined, *GEN* 4-64
co command (edit)
description, *GEN* 3-15
co command (ex)
description, *GEN* 3-88
Code generation (C compiler)
description, *PGM* 2-68 to 2-76
matching table entries against trees, *PGM* 2-69
Column
specifying, *GEN* 5-43
specifying headers for continuing pages, *GEN* 5-42
specifying headers for continuing pages with a macro, *GEN* 5-75E
specifying in text file, *GEN* 5-6
starting, *GEN* 5-35
text formatting commands for double columns, *GEN* 5-15E, 5-35
Comma character (ed)
compared with semicolon, *GEN* 3-45
COMMA operator (C compiler)
defined, *PGM* 2-66
Command (Bourne shell)
See also specific commands
grammar, *GEN* 4-26
grouping, *GEN* 4-14
Command (C shell)
See also Program
See also specific commands
defined, *GEN* 4-64
reference list, *GEN* 4-63 to 4-74
regenerating, *SYS* 5-118
repeating, *GEN* 4-41 to 4-43, 4-51E
substituting output for, *GEN* 4-61E
suspending temporarily, *GEN* 4-36
terminating, *GEN* 4-35 to 4-38
typing, *GEN* 2-4
within quotation marks, *GEN* 4-60
Command (DC)
See also specific commands
for human use
Command (DC)
for human use (Cont.)
reference list, *GEN* 2-57 to 2-59
how they work, *GEN* 2-57
Command (ed)
See also specific commands
description, *GEN* 3-25
reference list, *GEN* 3-34
Command (ex)
See also specific commands
addressing primitives, *GEN* 3-87
combining addressing primitives, *GEN* 3-87
exceeding thresholds, *GEN* 3-86
reference list, *GEN* 3-87 to 3-96
structure of, *GEN* 3-86
syntax, *GEN* 3-87E
Command (M4)
See also specific commands
reference list, *PGM* 2-398
Command (Mail)
See also specific commands
reference list, *GEN* 2-28 to 2-33, 2-39T
Command (make)
defined, *PGM* 3-16
Command (nroff)
description, *GEN* 5-22 to 5-25
Command (nroff/troff)
See also specific commands
reference list, *GEN* 5-51
Command (vi)
See also specific commands
case and, *GEN* 3-59
ex 3.5 changes and, *GEN* 3-103
for file manipulation, *GEN* 3-71T
preceding counts and, *GEN* 3-70
Command file
description, *GEN* 1-29
Command line
running two programs with one, *GEN* 2-11
Command line flag (Mail)
See Flag (Mail)
Command mode (ex)
defined, *GEN* 3-85
Command name
defined, *GEN* 4-64
Command procedure
See Shell procedure
Command substitution
See also Modifier (C shell)
defined, *GEN* 4-65

Command-list
defined, *GEN* 4-8
grouping commands, *GEN* 4-14
Comment (awk)
defined, *PGM* 3-9
Comment (BC)
convention, *GEN* 2-49, 2-50
Comment (ex)
description, *GEN* 3-86
Comment (nroff/troff)
specifying, *GEN* 5-67
Communication domain
defined, *SYS* 3-6
Component
defined, *GEN* 4-65
Compound statement (BC)
forming, *GEN* 2-54
Computer-aided instruction
See CAI scripts
comsat program
4.2BSD improvement, *SYS* 1-19
CON operator (C compiler)
defined, *PGM* 2-66
Conditional
See if/endif commands
conf.c file
4.2BSD improvement, *SYS* 5-14
installing device driver and, *SYS* 5-119
conf.h file
4.2BSD improvement, *SYS* 5-6
config program
4.2BSD improvement, *SYS* 1-19
adding nonstandard system facilities, *SYS* 5-96
defined, *SYS* 5-73
description, *SYS* 5-73 to 5-105
device defaults, *SYS* 5-99 to 5-100
files generated by, *SYS* 5-76
modifying system code, *SYS* 5-88
modifying system configuration, *SYS* 5-76
prerequisite information, *SYS* 5-74
profiled systems and, *SYS* 5-78
specifying options items, *SYS* 5-75
Configuration clause
description, *SYS* 5-80
Configuration file
contents, *SYS* 5-76
creating, *SYS* 5-76
grammar, *SYS* 5-97 to 5-98
specifying devices, *SYS* 5-81
Configuration file (Cont.)
specifying multiple bootable images, *SYS* 5-80
syntax, *SYS* 5-79 to 5-83
VAX-11/780 sample, *SYS* 5-84 to 5-87
connect system call
datagram sockets and, *SYS* 3-10
errors, *SYS* 3-8
establishing connection between sockets, *SYS* 1-10
initiating connection, *SYS* 3-8E
Connect time accounting
summarizing, *SYS* 5-56
Connection
accepting, *SYS* 3-9E
receiving, *SYS* 3-8 to 3-9
Constant (BC)
defined, *GEN* 2-50
Context search (ed)
backslash character and, *GEN* 3-43
defined, *GEN* 3-35
methods, *GEN* 3-30 to 3-31
question mark character and, *GEN* 3-43
repeating a search, *GEN* 3-31
reverse order and, *GEN* 3-31
slashes and, *GEN* 3-39
Context search (edit)
d command and, *GEN* 3-16
delete command and, *GEN* 3-16C
move command and, *GEN* 3-15
repeating, *GEN* 3-20E
reversing, *GEN* 3-20
s command and, *GEN* 3-20
continue command (C shell)
defined, *GEN* 4-65
continue statement (awk)
defined, *PGM* 3-9
Control character (C shell)
defined, *GEN* 4-65
Control character (nroff/troff)
changing, *GEN* 5-67
commands and, *GEN* 5-56
Control character (vi)
in text file, *GEN* 3-61
Control statement (BC), *GEN* 2-47E
description, *GEN* 2-47 to 2-48
Cooper, E., & others
4.2BSD System Manual, *PGM* 4-15 to 4-52

copy command (C shell)
See cp command (C shell)
copy command (edit)
See co command (edit)
copy command (ex)
See co command (ex)
copy command (Mail)
See also save command (Mail)
description, *GEN* 2–29
using, *GEN* 2–23E
copy program
loading, *SYS* 5–24E
mini-root file system and, *SYS* 5–24
Core dump file
defined, *GEN* 4–65
program faults and, *GEN* 1–31
terminating a program and, *GEN* 4–37
Cover sheet
entering in text file, *GEN* 5–5
formatting commands, *GEN* 5–5E
cp command (C shell)
4.2BSD improvement, *SYS* 1–5
copying a file, *GEN* 2–7E, 3–47
defined, *GEN* 4–65
saving a file, *GEN* 3–47E
cpu type parameter (config)
defined, *SYS* 5–79
CR key
See RETURN key
Crash
recovering files after, *GEN* 3–22
creat function
description, *PGM* 1–10
creat system call
obsolete in 4.2BSD, *SYS* 1–10
cref program
defined, *GEN* 2–13
crmode routine
defined, *PGM* 4–84
crt option (Mail)
paging mail, *GEN* 2–20
type command and, *GEN* 2–32
crt0.ex file
4.2BSD improvement, *SYS* 5–13
cs command (troff)
defined, *GEN* 5–58
csh program
See C shell
cshrc file
defined, *GEN* 4–65
logging in and, *GEN* 4–39
CSPACE operator (C compiler)
defined, *PGM* 2–64
css network driver
4.2BSD improvement, *SYS* 1–15
ctags
4.2BSD improvement, *SYS* 1–5
ctime library
4.2BSD improvement, *SYS* 1–14
CTRL-B
defined, *GEN* 3–75
description, *GEN* 3–56
CTRL-C
ULTRIX–32 and, *GEN* 2–1
CTRL-D
See also CTRL-U
defined, *GEN* 3–75
description, *GEN* 3–56
CTRL-E
defined, *GEN* 3–75
description, *GEN* 3–56
CTRL-F
defined, *GEN* 3–75
description, *GEN* 3–56
CTRL-G
defined, *GEN* 3–75
vi and, *GEN* 3–57
CTRL-H
See also At sign
See also u command (edit)
defined, *GEN* 3–75
deleting characters, *GEN* 3–7
CTRL-J
defined, *GEN* 3–75
CTRL-L
defined, *GEN* 3–75
CTRL-M
defined, *GEN* 3–75
CTRL-N
defined, *GEN* 3–75
CTRL-P
defined, *GEN* 3–76
CTRL-R
defined, *GEN* 3–76
CTRL-U
See also CTRL-D
defined, *GEN* 3–76
description, *GEN* 3–56
ULTRIX–32 and, *GEN* 2–1
CTRL-Y
defined, *GEN* 3–76
description, *GEN* 3–56
CTRL-Z
defined, *GEN* 3–76

cu command (nroff)
defined, *GEN* 5–67
cu program
See tip program
Current line
printing, *GEN* 3–11E
curses library
4.2BSD improvement, *SYS* 1–14
Cursor motion optimization
stand alone, *PGM* 4–78 to 4–80
Cursor positioning key
terminals and, *GEN* 3–55
Cut mark
specifying for troff, *GEN* 5–74E
Cutting and pasting
See cp command (ed)
See m command (ed)
See mv program (ed)
with ed, *GEN* 3–49 to 3–51
with UNIX commands, *GEN* 3–47 to 3–49
cwd variable (C shell)
defined, *GEN* 4–65
working directory and, *GEN* 4–41
Cylinder group
description, *SYS* 1–26, 2–8
Czech
See Metacharacters

D

d command (DC)
descripton, *GEN* 2–58
d command (ed)
defined, *GEN* 3–34
using, *GEN* 3–29
d command (edit)
context search and, *GEN* 3–16
description, *GEN* 3–15
d command (ex)
description, *GEN* 3–88
d command (me)
defined, *GEN* 5–43
d command (sed)
defined, *GEN* 3–108
D command (vi)
defined, *GEN* 3–78
d escape (Mail)
description, *GEN* 2–24
d flag (Mail)
See also debug option
debugging information and, *GEN* 2–36
d flag (make)
defined, *PGM* 3–17
d operator (vi)
defined, *GEN* 3–80
d option (inv)
defined, *GEN* 5–147
d option (uucico)
defined, *SYS* 5–135
d option (uuclean)
defined, *SYS* 5–137
d option (uucp)
defined, *SYS* 5–131
DA command (ms)
specifying date on text page, *GEN* 5–9
da command (nroff/troff)
defined, *GEN* 5–65
Daisy wheel printer
setting for 12-pitch, *GEN* 5–39
DARPA File Transfer Protocol server program
See ftpd program
DARPA Internet
network architecture support, *SYS* 1–15
DARPA Internet protocol
support, *SYS* 5–47
DARPA Request For Comments #833
sendmail and, *SYS* 1–4
DARPA Simple Mail Transfer Protocol
sendmail and, *SYS* 1–4
DARPA TELNET protocol
See telnetd server program
DARPA Trivial File Transfer Protocol
See tftpd server program
Dash
specifying em dash, *GEN* 5–47
Data block
kinds of, *SYS* 2–12
Data file
defined, *SYS* 5–131
DATA operator (C compiler)
defined, *PGM* 2–64
Data segment (as)
description, *GEN* 6–54
data statement
defined, *GEN* 6–59
Data Translation A/D converter
See ad driver
Datagram socket
See also Raw socket

Datagram socket (Cont.)
creating for on-machine use, *SYS* 3–7E
defined, *SYS* 3–6
description, *SYS* 3–10
sending broadcast packets on networks, *SYS* 3–27
Date
specifying with -me, *GEN* 5–47
specifying with -ms, *GEN* 5–9
date command (C shell)
defined, *GEN* 4–65
using, *GEN* 2–4
dbx symbolic debugger
description, *SYS* 1–4
Pascal compiler pc and, *SYS* 1–8
DC program
See also BC language
defined, *GEN* 2–57
description, *GEN* 2–57 to 2–64
internal arithmetic and, *GEN* 2–60
programming, *GEN* 2–62
de command (nroff/troff)
See also ig command (nroff/troff)
defined, *GEN* 5–64
defining macros, *GEN* 5–89E
Dead.letter file, *GEN* 2–24
canceling mail and, *GEN* 2–18
debug option (Mail)
See also -d flag
defined, *GEN* 2–34
Debugging
defined, *GEN* 4–65
DecWriter III printer
setting for serial lines, *PGM* 4–101E
Default
defined, *GEN* 4–65
define command (M4)
description, *PGM* 2–393 to 2–395
define keyword (BC), *GEN* 2–46E
define program (EQN)
description, *GEN* 5–100
define statement (BC)
forming, *GEN* 2–55
delay routine
description, *PGM* 2–76
Delayed text
defined, *GEN* 5–28
delch routine
defined, *PGM* 4–82
delete command (ed)
See d command (ed)
delete command (edit)
See d command (edit)
delete command (ex)
See d command (ex)
delete command (Mail)
See also autoprint option (Mail)
See also dt command (Mail)
See also undelete command (Mail)
abbreviating, *GEN* 2–20
description, *GEN* 2–29
keeping message from mbox, *GEN* 2–20E
DELETE key
defined, *GEN* 4–65
description, *GEN* 3–55
ULTRIX–32 and, *GEN* 2–1
deleteln routine
defined, *PGM* 4–82
delivermail program
See sendmail program
delwin routine
defined, *PGM* 4–85
DES encryption algorithm
chips and, *SYS* 4–11
Description file (make), *PGM* 3–14E
See also -f flag (make)
description, *PGM* 3–15 to 3–16
Detached command
defined, *GEN* 4–65
Device driver
converting local to 4.2BSD, *SYS* 5–4
CSR value list, *SYS* 5–61
I/O system and, *PGM* 4–67 to 4–73
installing new, *SYS* 5–119
prerquisites, *SYS* 5–89
Device name
convention, *SYS* 5–19
devices.vax file
4.2BSD improvement, *SYS* 5–11
df
reporting disk space in kilobytes, *SYS* 1–5
dh.c device driver
4.2BSD improvement, *SYS* 5–12
di command (nroff/troff)
defined, *GEN* 5–64
diverting output to a macro, *GEN* 5–94
Diacritical marks
available
reference list, *GEN* 5–19

Diacritical marks (Cont.)
entering with EQN, *GEN* 5-100
Diagnostic
defined, *GEN* 4-65
Diagnostic output
redirecting, *GEN* 4-44E
Dial-up network
description, *SYS* 5-123 to 5-129
operation, *SYS* 5-124
processing, *SYS* 5-125 to 5-126
protocol and, *SYS* 5-124, 5-126
security, *SYS* 5-125
starting your network, *SYS* 5-128
transmission speed, *SYS* 5-127
uses, *SYS* 5-126
Diction program
See also Style program
description, *GEN* 5-163 to 5-177
diff utility
comparing files, *GEN* 2-13
dir
4.2BSD improvement, *SYS* 1-16
dir.h file
4.2BSD improvement, *SYS* 5-6
directories command
See dirs command (C shell)
Directory
See also Home directory
See also Root directory
See also Working directory
allocating, *SYS* 1-33
alternate name for, *GEN* 2-10
changing, *GEN* 2-10
changing working directory, *GEN* 2-10
creating, *GEN* 2-10
defined, *GEN* 4-66, *PGM* 4-10
description, *GEN* 1-21, 2-9
determining, *GEN* 2-10
listing basic, *GEN* 2-9
moving up one level, *GEN* 2-10E
organization changes for 4.2BSD, *SYS* 5-4
project-related, *GEN* 4-48
removing, *GEN* 2-10E
security of, *SYS* 4-4
Directory data block
defined, *SYS* 2-12
directory library
4.2BSD improvement, *SYS* 1-14
directory option (ex)
description, *GEN* 3-98
Directory stack
defined, *GEN* 4-66
dirs command (C shell)
See also pwd command (C shell)
compared with pwd, *GEN* 4-49
defined, *GEN* 4-66
saving name of previous directory, *GEN* 4-49
Disk
balancing load, *SYS* 5-39
configuring load, *SYS* 5-37 to 5-43
defined, *GEN* 3-4
dividing into partitions, *SYS* 5-38
formatting, *SYS* 5-22 to 5-24
reporting space in kilobytes, *SYS* 1-5
reporting usage in kilobytes, *SYS* 1-5
space limits, *SYS* 4-3
space per device, *SYS* 5-38, 5-39T
Disk bandwith
4.2BSD improvement, *SYS* 1-3
Disk driver
UNIX implementation and, *PGM* 4-9
Disk partition
description, *SYS* 5-19
sizes, *SYS* 5-38
Disk quota
4.2BSD improvement, *SYS* 1-18
disabling, *SYS* 2-4
enabling, *SYS* 2-4
enforcing, *SYS* 5-57
per filesystem, *SYS* 1-4
per user, *SYS* 1-4
recovering from over quota condition, *SYS* 2-3
restricting, *SYS* 1-35
setting, *SYS* 2-4
types of, *SYS* 2-3
Disk quota system
configuration requirement, *SYS* 5-57
description, *SYS* 2-3 to 2-5
establishing, *SYS* 2-4
history, *SYS* 2-5
including, *SYS* 2-4E
programs, *SYS* 5-57
diskpart program
4.2BSD improvement, *SYS* 1-19
disktab file
4.2BSD improvement, *SYS* 1-16
Display (nroff)
defined, *GEN* 5-25, 5-42
description, *GEN* 5-25 to 5-27
specifying in fill mode, *GEN* 5-26

Display (nroff) (Cont.)
text formatting commands for, *GEN* 5-15E
distrib routine
description, *PGM* 2-68
Distribution tape
constructing, *SYS* 5-59 to 5-61
contents, *SYS* 5-59T
Diversion (troff)
description, *GEN* 5-94
divert command (M4)
description, *PGM* 2-396
Division
DC and, *GEN* 2-61
divnum command (M4)
description, *PGM* 2-396
DL-11W
See kg driver
dmc network interface driver
4.2BSD improvement, *SYS* 1-15
DMC-11/DMR-11 point-to-point communications device
See dmc network interface driver
dmf.c device driver
4.2BSD improvement, *SYS* 5-12
dnl command (M4)
description, *PGM* 2-397
Document preparation
description, *GEN* 2-12 to 2-14
hints, *GEN* 2-13 to 2-14
reading list, *GEN* 2-16
DOD Standard TCP/IP network communication protocols
support for, *SYS* 1-3
Dollar sign character (ed)
end of line and, *GEN* 3-39
meaning, *GEN* 3-33, 3-40
p command and, *GEN* 3-28
printing value, *GEN* 3-35
Dollar sign character (edit)
equal sign and, *GEN* 3-17
printing last buffer line, *GEN* 3-17
searching and, *GEN* 3-20
domain.h file
4.2BSD improvement, *SYS* 5-5
don't command (sed)
defined, *GEN* 3-113
Dot character (C shell)
at beginning of file, *GEN* 4-34
defined, *GEN* 4-63
separating filename components, *GEN* 4-33
Dot character (ed)
determining value, *GEN* 3-29E
equal sign and, *GEN* 3-35
line number defaults and, *GEN* 3-44 to 3-45
meaning, *GEN* 3-38, 3-39
meaning for context searching, *GEN* 3-33
p command and, *GEN* 3-28
printing, *GEN* 3-39
s command and, *GEN* 3-29
setting with semicolon, *GEN* 3-45 to 3-46
using, *GEN* 3-28, 3-33
Dot character (edit)
equal sign and, *GEN* 3-17
uses, *GEN* 3-17
Dot character (nroff/troff)
See Control character (nroff/troff)
specifying lines of, *GEN* 5-88
dot option (Mail)
See also ignoreof option
defined, *GEN* 2-34
Doublespacing
specifying, *GEN* 5-23
drtest program
4.2BSD improvement, *SYS* 1-19
DS command (ms)
specifying line breaks, *GEN* 5-8
ds command (nroff/troff)
defined, *GEN* 5-64
defining strings, *GEN* 5-89
DSTFLAG parameter
description, *SYS* 5-122
dt command (Mail)
description, *GEN* 2-29
dt command (nroff/troff)
defined, *GEN* 5-65
du command (C shell)
defined, *GEN* 4-66
reporting disk usage in kilobytes, *SYS* 1-5
du program
See du command (C shell)
dump program
See also rdump program
4.2BSD improvement, *SYS* 1-16, 1-19
using, *SYS* 5-53
dumpdef command (M4)
description, *PGM* 2-397
dumpfs program
4.2BSD improvement, *SYS* 1-19

Dungeons of doom
See Rogue game
Dynamic string storage allocator
See Allocator

E

e command (ed)
defined, *GEN* 3-34
using, *GEN* 3-27, 3-49E
e command (edit)
copying a file, *GEN* 3-14
r option and, *GEN* 3-23
u command and, *GEN* 3-16
e command (ex)
description, *GEN* 3-88
E command (vi)
defined, *GEN* 3-79
e command (vi)
defined, *GEN* 3-80
e escape (Mail)
description, *GEN* 2-24
e flag (sed)
defined, *GEN* 3-106
e modifier (C shell)
extracting filename extension, *GEN* 4-57E
e option (nroff)
defined, *GEN* 5-50
ec command (nroff/troff)
defined, *GEN* 5-66
ec network interface driver
4.2BSD improvement, *SYS* 1-15
echo command (C shell)
defined, *GEN* 4-66
echo routine
defined, *PGM* 4-84
ed line editor
See also edit line editor
See also ex line editor
accessing, *GEN* 3-25
adding text, *GEN* 3-25
addressing lines, *GEN* 3-43 to 3-46
advanced editing, *GEN* 3-37 to 3-52
backslash character and, *GEN* 3-33
breaking lines, *GEN* 3-42
CAI script for, *GEN* 6-7
changing text, *GEN* 3-31 to 3-32
command summary, *GEN* 3-34
context searching, *GEN* 3-30 to 3-31
ed line editor (Cont.)
copying lines, *GEN* 3-51
creating text, *GEN* 3-25
deleting text, *GEN* 3-29
description, *GEN* 2-6
escaping to use UNIX command, *GEN* 3-51
global commands, *GEN* 3-32
inserting text, *GEN* 3-31 to 3-32
interrupting, *GEN* 3-46
introduction, *GEN* 3-25 to 3-35
joining lines, *GEN* 3-42
line number defaults, *GEN* 3-44 to 3-45
marking a line, *GEN* 3-50
moving text, *GEN* 3-32, 3-50
printing a file, *GEN* 2-7
printing lines, *GEN* 3-27
reading a file, *GEN* 3-27
rearranging a line, *GEN* 3-43
repeating searches, *GEN* 3-44
searching for first occurrence of text string, *GEN* 3-46
sed and, *GEN* 3-105
setting dot, *GEN* 3-45 to 3-46
specifying lines with text patterns, *GEN* 3-46 to 3-47
specifying the second occurrence of text string, *GEN* 3-46
substituting text, *GEN* 3-29
supporting tools, *GEN* 3-51 to 3-52
using special characters, *GEN* 3-33
writing a file, *GEN* 3-26
ed.hup file
saving text, *GEN* 2-6
edcompatible option (ex)
description, *GEN* 3-98
edit command (ed)
See e command (ed)
edit command (edit)
See e command
edit command (ex)
See e command (ex)
edit command (Mail)
See also visual command (Mail)
description, *GEN* 2-29
edit line editor
See also ed line editor
See also ex line editor
accessing, *GEN* 3-5 to 3-6
adding text, *GEN* 3-9
correcting text, *GEN* 3-9

edit line editor (Cont.)
current line and, *GEN* 3–11
defined, *GEN* 3–3
entering text, *GEN* 3–6
ex editor and, *GEN* 3–23
finding a line, *GEN* 3–11E
issuing UNIX command from, *GEN* 3–21
messages, *GEN* 3–6
moving around in the buffer, *GEN* 3–17
opening a file, *GEN* 3–9E, 3–14E
prerequisites, *GEN* 3–3
printing current line number, *GEN* 3–11
printing nonprinting characters, *GEN* 3–10
quitting, *GEN* 3–8
reversing last command, *GEN* 3–16
saving modified text, *GEN* 3–13
searching for characters, *GEN* 3–10, 3–10E
tutorial, *GEN* 3–3 to 3–23
Editing
hints for, *GEN* 2–13
Editor
See ed editor
See edit editor
See ex editor
See Screen editor
See sed stream editor
See vi screen editor
EDITOR option (Mail)
defined, *GEN* 2–33
setting, *GEN* 2–33
specifying an editor, *GEN* 2–24
edquota program
4.2BSD improvement, *SYS* 1–19
ef command (me)
defined, *GEN* 5–41
efftab table
defined, *PGM* 2–68
EFL programming language
description, *PGM* 2–123 to 2–157
eh command (me)
defined, *GEN* 5–41
el command (nroff/troff)
defined, *GEN* 5–71
else command (C shell)
See also if/endif commands (C shell)
See also then command (C shell)
defined, *GEN* 4–66
else command (Mail)
See also if/endif commands (Mail)
description, *GEN* 2–30
else statement (awk)
defined, *PGM* 3–9
Elz, R.
disk quota system, *SYS* 2–3 to 2–5
em
defined, *GEN* 5–86
em command (nroff/troff)
defined, *GEN* 5–65
Em dash
in nroff/troff output, *GEN* 5–19
Emphasis
See Boldface
See Italic
See Overstriking
See Underlining
en network interface driver
4.2BSD improvement, *SYS* 1–16
enable/disable command (lpc)
description, *PGM* 4–103
endif command (C shell)
See if/endif commands (C shell)
endif command (Mail)
See if/endif commands (Mail)
endif statement (as)
See if/endif statement (as)
endwin routine
defined, *PGM* 4–85
Entry file
defined, *GEN* 5–145
Environment (C shell)
displaying, *GEN* 4–51E
Environment (nroff/troff)
description, *GEN* 5–71, 5–94
eo command (nroff/troff)
defined, *GEN* 5–66
EOF (End of File)
defined, *GEN* 2–5, 4–66
EOF operator (C compiler)
defined, *PGM* 2–64
EOF value
defined, *PGM* 1–21
description, *PGM* 1–4
ep command (me)
defined, *GEN* 5–42
EQ command (EQN)
specifying continuation, *GEN* 5–35
specifying equations, *GEN* 5–34
supplementing with troff commands, *GEN* 5–101
EQ command (me)
defined, *GEN* 5–45

EQ command (ms)
specifying equations, *GEN* 5-10
EQN program
See also NEQN program
CAI script for, *GEN* 6-7
connecting output to troff, *GEN* 5-101
deficiencies, *GEN* 5-102
defined, *GEN* 5-105
description, *GEN* 5-33, 5-97 to 5-104
forcing extra white space, *GEN* 5-99
formatting mathematics, *GEN* 2-13
grammar, *GEN* 5-101
language design, *GEN* 5-98
language theory, *GEN* 5-101
quoting an input string, *GEN* 5-100
Equal sign (ed)
dot character and, *GEN* 3-35
Equation
continuing, *GEN* 5-35E
formatting, *GEN* 5-33
numbering, *GEN* 5-34
setting with -ms, *GEN* 5-10
text formatting commands for, *GEN* 5-16E
Erase character
See also Backspace character
default, *GEN* 4-30
erase routine
defined, *PGM* 4-82
errno cell
description, *PGM* 1-12
errno.h file
4.2BSD improvement, *SYS* 5-5
error
troff messages and, *SYS* 1-5
error bells option (ex)
description, *GEN* 3-98
Error condition (fsck)
conventions, *SYS* 2-14
Error log file
examining, *SYS* 5-53
Error message (ed)
description, *GEN* 3-26
errprint command (M4)
description, *PGM* 2-397
Escape character (Mail)
changing, *GEN* 2-26
Escape character (nroff/troff)
description, *GEN* 5-66
Escape character(C shell)
defined, *GEN* 4-66
escape command
See ! command (ed)
ESCAPE key
description, *GEN* 3-55
escape option (Mail)
changing escape character, *GEN* 2-26
defined, *GEN* 2-34
Escape sequence (nroff/troff)
reference list, *GEN* 5-54
ev command (nroff/troff)
changing environment, *GEN* 5-94
description, *GEN* 5-72
eval command (M4)
description, *PGM* 2-396
Evans and Sutherland Picture System 2
See ps.c device driver
EVEN operator (C compiler)
defined, *PGM* 2-64
even statement (as)
defined, *GEN* 6-59
ex command (ex)
See e command (ex)
ex command (nroff/troff)
defined, *GEN* 5-72
ex line editor
See also ed line editor
See also edit line editor
See also sed stream editor
See also vi screen editor
3.5 changes, *GEN* 3-102
command line format, *GEN* 3-83
editing modes, *GEN* 3-85
encryption code and, *GEN* 3-102
entering multiple commands on a line, *GEN* 3-86
errors and, *GEN* 3-85
file manipulation, *GEN* 3-84 to 3-85
limitations, *GEN* 3-101
printing current line number, *GEN* 3-95
printing version number, *GEN* 3-94
recovering from crash, *GEN* 3-85
recovering work, *GEN* 3-85E
reference manual, *GEN* 3-83 to 3-104
starting, *GEN* 3-83
vi and, *GEN* 3-73

Ex Reference Manual, *GEN* 3-83 to 3-104
See also ex line editor
Examples
entering with troff, *GEN* 5-89
Exception word list (nroff/troff)
specifying, *GEN* 5-69
Exclamation mark (C shell)
using in command arguments, *GEN* 4-35
Exclamation mark character (ed)
shell command and, *GEN* 3-35
Exclamation mark character (edit)
shell command and, *GEN* 3-21
Exclusive lock
process and, *SYS* 1-3
execl function
See also execv
See also fork function
description, *PGM* 1-13
Execute file
defined, *SYS* 5-133 to 5-134
execv routin
description, *PGM* 1-13
exit command (C shell)
defined, *GEN* 4-66
exit command (Mail)
description, *GEN* 2-30
exit function
error handling and, *PGM* 1-8
exit statement (awk)
defined, *PGM* 3-9
exit status
defined, *GEN* 4-66
exp function (awk)
defined, *PGM* 3-8
Expansion
defined, *GEN* 4-67
Exponentiation
DC and, *GEN* 2-61
Exponentiation operator
description, *GEN* 2-52
EXPR operator (C compiler)
defined, *PGM* 2-65
Expression
defined, *GEN* 4-67
Expression (as)
defined, *GEN* 6-56
types of
reference list, *GEN* 6-57
Expression (BC)
See also Primitive expression
defined, *GEN* 2-50 to 2-53
length, *GEN* 2-51
Expression (C shell)
evaluating, *GEN* 4-55
Expression operator (as)
reference list, *GEN* 6-57
Expression statement (as)
defined, *GEN* 6-55
Expression statement (BC)
description, *GEN* 2-54
Extended Fortran Language
See EFL programming language
Extension
defined, *GEN* 4-67
External security code
password security and, *SYS* 4-12
eyacc
4.2BSD improvement, *SYS* 1-5

F

F argument (nroff)
specifying fill mode, *GEN* 5-26
f command (ed)
defined, *GEN* 3-34
determining the filename, *GEN* 3-49
renaming a file, *GEN* 3-49E
f command (edit)
description, *GEN* 3-21
f command (ex)
description, *GEN* 3-89
f command (me)
defined, *GEN* 5-43
entering, *GEN* 5-28
f command (troff)
mixing fonts within a line, *GEN* 5-86
mixing fonts within a word, *GEN* 5-86
F command (vi)
defined, *GEN* 3-79
using, *GEN* 3-61
f command (vi)
defined, *GEN* 3-80
using, *GEN* 3-61
f flag (Mail)
defined, *GEN* 2-36
reading mail from specified file, *GEN* 2-21
f flag (make)
defined, *PGM* 3-17
f flag (mkey)
reading file list, *GEN* 5-147
f flag (sed)
defined, *GEN* 3-106

f flag (su)
fast su and, *SYS* 1-9
f macro (me)
defined, *GEN* 5-42
F option (hunt)
defined, *GEN* 5-148
f option (troff)
defined, *GEN* 5-50
f77 I/O library
4.2BSD improvement, *SYS* 1-6
description, *PGM* 2-79 to 2-88
error messages, *PGM* 2-85 to 2-87
exceptions to ANSI standard, *PGM* 2-88
Fabry, R., & others
4.2BSD System Manual, *PGM* 4-15 to 4-52
Fabry, R.S., & others
4.2BSD Interprocess Communication Primer, *SYS* 3-5 to 3-28
fast file system, *SYS* 1-23 to 1-38
networking implementation notes, *SYS* 3-29 to 3-57
factor program
4.2BSD improvement, *SYS* 1-17
fastboot script
See also fasthalt script
4.2BSD improvement, *SYS* 1-19C
fasthalt script
See also fastboot script
4.2BSD improvement, *SYS* 1-19
fc command (nroff/troff)
defined, *GEN* 5-66
fchmod system call
4.2BSD improvement fchmod, *SYS* 1-10
fchown system call
4.2BSD improvement, *SYS* 1-10
fclose function
description, *PGM* 1-7
fcntl system call
4.2BSD improvement, *SYS* 1-10
FCON operator (C compiler)
defined, *PGM* 2-66
fed font editor
value of, *SYS* 1-6
Feldman, S.I.
EFL programming language, *PGM* 2-123 to 2-157
Make program, *PGM* 3-13 to 3-21
Feldman, S.I., & Weinberger, P.J.
Fortran 77 compiler, *PGM* 2-89 to 2-109
feof macro
breakpoints and, *PGM* 1-21
ferror macro
breakpoints and, *PGM* 1-21
fflush function
description, *PGM* 1-8
fg command (C shell)
defined, *GEN* 4-67
running background job in foreground, *GEN* 4-47E
running suspended job in foreground, *GEN* 4-47
fgets function
description, *PGM* 1-8
fgrep
hunt program and, *GEN* 5-148
fi command (nroff/troff)
defined, *GEN* 5-61
Field (awk)
description, *PGM* 3-8
Field (nroff/troff)
defined, *GEN* 5-66
Figure
specifying blank page for, *GEN* 5-44
specifying ruling for, *GEN* 5-45
specifying space for, *GEN* 5-44
FILE
defined, *PGM* 1-21
File
See also File system
See also specific files
advisory locking and, *SYS* 1-3
appending, *GEN* 3-48
appending contents to mail, *GEN* 2-24
arranging, *GEN* 2-10
CAI script for, *GEN* 6-7
combining, *GEN* 2-10, 3-48, 3-49
comparing, *GEN* 2-13
copying, *GEN* 2-7E, 3-47
copying from other directories, *GEN* 2-9
creating, *GEN* 2-6
defined, *GEN* 2-6, 3-3, *PGM* 4-10
description, *GEN* 1-20
displaying, *GEN* 2-10
handling multiple, *GEN* 2-8
I/O device and, *GEN* 1-21
marking executable, *GEN* 2-12
merging multiple, *GEN* 2-14E
open limit, *PGM* 1-11
opening with edit, *GEN* 3-14
optimal size, *SYS* 1-28

File (Cont.)
paging, *GEN* 2-7
printing, *GEN* 2-7
printing from other directories, *GEN* 2-9
printing merged, *GEN* 2-11
printing multiple, *GEN* 2-7, 2-8, 2-11
printing on high-speed printer, *GEN* 2-7
programs executed by the shell and, *GEN* 1-27
protection information, *SYS* 4-3
recovering with edit, *GEN* 3-22
removing, *GEN* 3-48
removing multiple from directory, *GEN* 2-10E
renaming, *GEN* 2-7
replacing the terminal, *GEN* 2-10
sending to several people, *GEN* 2-11
size of, *GEN* 1-23, 2-13
splitting, *GEN* 2-13
truncating to specific length, *SYS* 1-4
viewing in other directories, *GEN* 2-9
writing part of, *GEN* 3-49
writing to disk, *GEN* 3-8
File (C shell)
See also specific files
accessing from other directories, *GEN* 4-34
directing input from, *GEN* 4-32E to 4-33E
inputting to, *GEN* 4-31
maintaining related, *GEN* 4-53
outputting from, *GEN* 4-31
redirecting terminal output to, *GEN* 4-31E
terminating a command, *GEN* 4-36E
File (line printer system)
reference list, *PGM* 4-99
File (M4)
manipulating, *PGM* 2-396
File (vi)
quitting, *GEN* 3-63
recovering, *GEN* 3-66
writing, *GEN* 3-63
file command
symbolic links and, *SYS* 1-6
file command (edit)
See f command (edit)
file command (ex)
See f command (ex)
file command (Mail)
See folder command (Mail)
File descriptor
changing assignments, *GEN* 1-28
description, *PGM* 1-8
File locking
description, *SYS* 1-33
File pointer
defined, *PGM* 1-5
File system
accessing directories on old and new systems, *SYS* 1-33
block size, *SYS* 2-8
checking structural integrity, *SYS* 2-10
data structure, *PGM* 4-12F
defined, *PGM* 4-10 to 4-13
description, *GEN* 1-20 to 1-24
fixing corrupted, *SYS* 2-10 to 2-13
fragmentation of, *SYS* 2-9
implementation, *PGM* 4-11
implementing, *GEN* 1-24 to 1-26
overview, *SYS* 2-8 to 2-9
protecting, *GEN* 1-22
removable volume and, *GEN* 1-22
updating, *SYS* 2-9
File system (4.2BSD)
See also File system (Bell)
allocating data blocks, *SYS* 1-30
allocating directories, *SYS* 1-30
allocating new blocks, *SYS* 1-29
allocation strategy, *SYS* 1-30
block size, *SYS* 1-26
block size and wasted space, *SYS* 1-27T
compared to previous file system, *SYS* 1-23 to 1-38
creating file versions, *SYS* 1-35
fragments and, *SYS* 1-27
free blocks and, *SYS* 1-28
hardware parameters and, *SYS* 1-28 to 1-29
implementing layout, *SYS* 5-42
layout policies, *SYS* 1-29 to 1-30
locking files, *SYS* 1-33
moving, *SYS* 5-54
optimizing storage, *SYS* 1-26
organization, *SYS* 1-26 to 1-30
performance, *SYS* 1-31 to 1-32
quotas and, *SYS* 2-4
reading rates, *SYS* 1-31T
restricting quota, *SYS* 1-35

File system (4.2BSD) (Cont.)
selecting parameters, *SYS* 5-40 to 5-41
software engineering, *SYS* 1-36
space overhead, *SYS* 1-28
writing rates, *SYS* 1-31T
File system (Bell)
description, *SYS* 1-25
File System Check Program
See fsck program
file.h file
4.2BSD improvement, *SYS* 5-6
Filelist file
creating, *GEN* 2-10
Filename
4.2BSD changes, *SYS* 5-4
arbitrary length and, *SYS* 1-3
changing, *GEN* 3-47, 3-47W
restriction, *GEN* 3-47
conventions for, *GEN* 2-8
description, *GEN* 1-21
edit editor and, *GEN* 3-21
folder name and, *GEN* 2-23
maximum length, *SYS* 1-33
renaming in same file system, *SYS* 1-4
specifying, *GEN* 3-8
suggestions, *GEN* 2-7
Filename (C shell)
base part and, *GEN* 4-63
characters in, *GEN* 4-33
defined, *GEN* 4-67
Filename expansion
defined, *GEN* 4-67
FILENAME variable (awk)
determining current input file, *PGM* 3-6
files file
4.2BSD improvement, *SYS* 5-11
adding device driver and, *SYS* 5-89
files.vax file
4.2BSD improvement, *SYS* 5-11
Fill mode
specifying, *GEN* 5-26
Filling (nroff/troff)
description, *GEN* 5-60 to 5-61
filsys.h file
See fs.h file
Filter
calling, *PGM* 4-103E
creating for printers, *PGM* 4-102
defined, *GEN* 4-4
description, *GEN* 1-28
find
finding symbolic links, *SYS* 1-6
Find key
defined, *GEN* 5-144
First page
entering in text file, *GEN* 5-5
fl command (nroff/troff)
defined, *GEN* 5-73
Flag (C shell)
purpose of, *GEN* 4-31
Flag (ex)
description, *GEN* 3-86
Flag (Mail)
reference list, *GEN* 2-41T
Flag option (C shell)
defined, *GEN* 4-67
Flag option (Mail)
defined, *GEN* 2-38
flags field (config
description, *SYS* 5-82
Floating keep, *GEN* 5-26F
defined, *GEN* 5-26
flock system call
4.2BSD improvement, *SYS* 1-10
fmt command
formatting outgoing mail, *GEN* 2-26
fo command (me)
defined, *GEN* 5-41
entering, *GEN* 5-23
Foderaro, J.K., & others
Franz Lisp Manual, The, *PGM* 2-211 to 2-358
Folder
specifying for file, *GEN* 2-23
folder command (Mail)
See also folders command (Mail)
description, *GEN* 2-30
directing Mail to a folder, *GEN* 2-23
Folder directory
specifying, *GEN* 2-23
Folder facility
description, *GEN* 2-23
folder option (Mail)
defined, *GEN* 2-34
Folders
maintaining, *GEN* 2-23
folders command (Mail)
See also folder command (Mail)
description, *GEN* 2-30
listing folder set, *GEN* 2-23
Font
changing, *GEN* 5-58, 5-86

Font (Cont.)
command list, *GEN* 5-51
default, *GEN* 5-58
defined, *GEN* 5-36
description, *GEN* 5-36 to 5-37
mixing within a line, *GEN* 5-86
mixing within a word, *GEN* 5-37, 5-86
setting, *GEN* 5-39
specifying, *GEN* 5-44, 5-85
specifying for a word, *GEN* 5-36E
specifying for more than one word, *GEN* 5-36
style examples, *GEN* 5-78T
switching, *GEN* 5-36
Font library
installing, *SYS* 5-31
Footer
See also Header
formatting, *GEN* 5-41 to 5-42
specifying, *GEN* 5-23
Footnote
See also Delayed text
entering, *GEN* 5-8, 5-28, 5-43
entering with a macro, *GEN* 5-76E
numbered automatically, *GEN* 5-17
resetting the numbering, *GEN* 5-46
separating footnotes, *GEN* 5-43
specifying point size, *GEN* 5-8
text formatting commands for, *GEN* 5-15E
fopen function
See also fclose function
See also open function
calling, *PGM* 1-5E
description, *PGM* 1-5
for loop
description, *GEN* 4-7
form, *GEN* 4-8E
for statement (awk)
defined, *PGM* 3-9
for statement (BC)
forming, *GEN* 2-54
process, *GEN* 2-47
writing, *GEN* 2-47
For system call
description, *GEN* 1-26
foreach command (C shell), *GEN* 4-56E
defined, *GEN* 4-67
exiting loop, *GEN* 4-58
foreach command (C shell) (Cont.)
performing similar commands, *GEN* 4-60E
Foreground
defined, *GEN* 4-67
Foreground job
continuing, *GEN* 4-46
description, *GEN* 4-45 to 4-48
suspending, *GEN* 4-46
fork function
description, *PGM* 1-14
Form feed character
printing, *GEN* 3-37
Form letter
using with nroff/troff, *GEN* 5-72
format program
4.2BSD improvement, *SYS* 1-18, 1-19, 5-15
formatting disks, *SYS* 5-22 to 5-24
loading, *SYS* 5-23
Fortran
See f77 I/O library
See Fortran 77
See Ratfor language
Fortran 77
C and, *GEN* 2-15
running old programs, *PGM* 2-83
Fortran 77 compiler
4.2BSD improvement, *SYS* 1-4
description, *PGM* 2-89 to 2-109
Fortran I/O
See also f77 I/O library
constraints, *PGM* 2-80 to 2-82
execution, *PGM* 2-80
forms of, *PGM* 2-79 to 2-80
general concepts, *PGM* 2-79 to 2-80
logical units and, *PGM* 2-80
unit numbers and, *PGM* 2-80
fortune game
4.2BSD improvement, *SYS* 1-17
Forward slash
searching for, *GEN* 3-39
fp command
specifying fonts on the typesetter, *GEN* 5-86
fp compiler/interpreter
Functional Programming language and, *SYS* 1-6
FP programming language
description, *PGM* 2-359 to 2-391
fpr program
printing Fortran files, *SYS* 1-6

fprintf function
description, *PGM* 1-7
Fraction
setting with troff, *GEN* 5-86E
specifying with EQN, *GEN* 5-99
Fragment size
selecting, *SYS* 5-41
frame.h file
4.2BSD improvement, *SYS* 5-13
Franz Lisp Manual, The, *PGM* 2-211 to 2-358
See also Franz Lisp system
Franz Lisp system
user manual, *PGM* 2-211 to 2-358
from command (Mail)
description, *GEN* 2-30
message lists and, *GEN* 2-28
from keyword (EQN), *GEN* 5-100E
Front matter
specifying, *GEN* 5-33
fs
4.2BSD improvement, *SYS* 1-16
FS command (ms)
specifying footnotes, *GEN* 5-8
FS variable (awk)
defined, *PGM* 3-6
fs.h file
4.2BSD improvement, *SYS* 5-5
fscanf function
See also sscanf function
description, *PGM* 1-8
fsck program
See also badsect program
4.2BSD improvement, *SYS* 1-19
checking connectivity, *SYS* 2-12
checking directory data blocks, *SYS* 2-12
checking free blocks, *SYS* 2-10
checking inode block count, *SYS* 2-12
checking inode links, *SYS* 2-11
checking inode state, *SYS* 2-11
checking super-block, *SYS* 2-10
description, *SYS* 2-7 to 2-25
error conditions, *SYS* 2-14 to 2-25
rebuilding block allocation maps, *SYS* 2-11
fsplit program
splitting multi-function Fortran files, *SYS* 1-6
fstab library
4.2BSD improvement, *SYS* 1-15
fstat system call
4.2BSD improvement, *SYS* 1-11
fsync system call
4.2BSD improvement, *SYS* 1-11
ft command (troff)
defined, *GEN* 5-59
specifying fonts, *GEN* 5-86
FTP server
description, *SYS* 5-50
ftp server program
ARPA file transfer protocol and, *SYS* 1-6
ftpd server program
4.2BSD improvement, *SYS* 1-19
ftpusers file
description, *SYS* 5-50
ftruncate system call
4.2BSD improvement, *SYS* 1-11
Function (BC)
description, *GEN* 2-45 to 2-46
number permitted, *GEN* 2-45
Function call
defined, *GEN* 2-51
Function identifier
description, *GEN* 2-50
fz command (nroff/troff)
specifying font size, *GEN* 5-81

G

g command (ed)
defined, *GEN* 3-34
process, *GEN* 3-46
s command and, *GEN* 3-46E
s command restriction and, *GEN* 3-47
specifying line numbers, *GEN* 3-47
specifying lines with text patterns, *GEN* 3-46 to 3-47
specifying more than one command, *GEN* 3-47
using, *GEN* 3-32
g command (edit)
description, *GEN* 3-19
p command and, *GEN* 3-19
substitute command and, *GEN* 3-19
uppercase letters and, *GEN* 3-19
using, *GEN* 3-19E
g command (ex)
description, *GEN* 3-89
G command (sed)
defined, *GEN* 3-113
g command (sed)
defined, *GEN* 3-113

G command (vi)
defined, *GEN* 3-79
finding text lines, *GEN* 3-57
g flag (sed)
defined, *GEN* 3-110
g option (hunt)
defined, *GEN* 5-148
g option (troff)
defined, *GEN* 5-50
g option (uucp)
defined, *SYS* 5-132
gcore program
creating a core dump of running process, *SYS* 1-6
genassym.c file
4.2BSD improvement, *SYS* 5-14
getc macro
defined, *PGM* 1-6
getch routine
defined, *PGM* 4-84
getchar macro
input and, *PGM* 1-4
getdtablesize system call
4.2BSD improvement, *SYS* 1-11
getgroups system call
4.2BSD improvement, *SYS* 1-11
gethostbynameandnet routine, *SYS* 3-13E
gethostid system call
4.2BSD improvement, *SYS* 1-11
gethostname system call
4.2BSD improvement, *SYS* 1-11
getitimer system call
4.2BSD improvement, *SYS* 1-11
getpagesize system call
4.2BSD improvement, *SYS* 1-11
getpass library
4.2BSD improvement, *SYS* 1-14
getpriority system call
4.2BSD improvement, *SYS* 1-11
getrlimit system call
4.2BSD improvement, *SYS* 1-11
getservbyname routine
specifying a protocl, *SYS* 3-14
getsockopt system call
4.2BSD improvement, *SYS* 1-11
getstr routine
defined, *PGM* 4-84
gettable program
4.2BSD improvement, *SYS* 1-19
retrieving NIC host data base, *SYS* 5-48
gettimeofday system call
4.2BSD improvement, *SYS* 1-11
gettimeofday system call (Cont.)
specifying value, *SYS* 5-74
gettmode routine
defined, *PGM* 4-88
variables set by, *PGM* 4-90T
getty program
See also gettytab file
4.2BSD improvement, *SYS* 1-18, 1-19
gettytab file
4.2BSD improvement, *SYS* 1-16
getwd library
4.2BSD improvement, *SYS* 1-15
getyx routine
defined, *PGM* 4-85
GID
description, *SYS* 4-4
global command (ed)
See g command (ed)
See v command (ed)
global command (edit)
See g command (edit)
global command (ex)
See g command (ex)
globl statement (as)
defined
go flag
accessing sdb symbol information, *SYS* 1-5
goto command (C shell)
defined, *GEN* 4-67
form of, *GEN* 4-58E
gprof command
profiled systems and, *SYS* 5-78
gprof program
See also gprof.h file
displaying execution time, *SYS* 1-6
gprof.h file
4.2BSD improvement, *SYS* 5-5
Graham, S.L., & others
Berkeley Pascal User Manual, *PGM* 2-159 to 2-209
Grave accent
See Metacharacters
Greek letters
setting with -ms, *GEN* 5-10
setting with troff, *GEN* 5-86E
troff command list, *GEN* 5-96
grep command (C shell)
defined, *GEN* 4-67
grep program
finding lines with combinations of text patterns, *GEN* 3-51

grep program (Cont.)
finding lines without specified text, *GEN* 3-51E
finding specified text in a set of files, *GEN* 3-51, 3-51E
nonalphabetic characters and, *GEN* 3-51
spell and, *GEN* 2-13
using, *GEN* 2-13E
Grep program
searching for text patterns, *GEN* 2-13
Group Identification Number
See GID
Group set
description, *SYS* 1-3
grouping command (sed)
defined, *GEN* 3-113
groups program
display access list for user's group, *SYS* 1-6

H

H command (sed)
defined, *GEN* 3-113
h command (sed)
defined, *GEN* 3-113
h command (troff)
moving text backwards on a line, *GEN* 5-87
specifying horizontal motion, *GEN* 5-68
H command (vi)
defined, *GEN* 3-79
h escape (Mail)
description, *GEN* 2-25
h flag (Mail)
defined, *GEN* 2-36
H macro (me)
specifying column heads on continuing pages, *GEN* 5-42
h macro (me)
defined, *GEN* 5-42
h option (inv)
defined, *GEN* 5-147
h option (nroff)
defined, *GEN* 5-81
Haley, C.B., & others
Berkeley Pascal User Manual, PGM 2-159 to 2-209
hangman game
4.2BSD improvement, *SYS* 1-17
Hard limit
defined, *SYS* 2-3
Hard lock
compared to advisory lock, *SYS* 1-33
Hardcopy terminal
vi and, *GEN* 3-73
hardtabs option (ex)
description, *GEN* 3-98
Hash character
See Sharp character
Hat
See Circumflex character (ed)
hc command (nroff/troff)
defined, *GEN* 5-69
he command (me)
defined, *GEN* 5-41
entering, *GEN* 5-23
head command (C shell)
defined, *GEN* 4-68
Header
See also Footer
formatting, *GEN* 5-41 to 5-42
specifying, *GEN* 5-23
suppressing, *GEN* 2-36
Header field
defined, *GEN* 2-38
headers command (Mail)
See also ignore command (Mail)
abbreviating, *GEN* 2-30
description, *GEN* 2-30
help command (Mail)
description, *GEN* 2-30
restriction, *GEN* 2-30
using, *GEN* 2-22
Henry, R.R., & Reiser, J.F.
Berkeley VAX/UNIX Assembler Reference Manual, PGM 4-53 to 4-65
Here document
description, *GEN* 4-9 to 4-10
Hexadecimal notation
BC language and, *GEN* 2-44
hier
4.2BSD improvement, *SYS* 1-17
history command (C shell)
defined, *GEN* 4-68
repeating previous commands, *GEN* 4-43
History list
description, *GEN* 4-41 to 4-43
using, *GEN* 4-42E
hl command (me)
defined, *GEN* 5-45

hl command (me) (Cont.)
figures and, *GEN* 5-26
hold command (Mail)
See also preserve command (Mail)
description, *GEN* 2-31
hold option (Mail)
defined, *GEN* 2-34
storing mail, *GEN* 2-20
Home directory
defined, *GEN* 4-68
returning to, *GEN* 4-49
HOME variable (Bourne shell)
description, *GEN* 4-11
home variable (C shell)
displaying your home directory, *GEN* 4-41
Horizonal line
See Ruling
Horton, M., & Joy, W.
editing with vi, *GEN* 3-53 to 3-82
Ex Reference Manual, *GEN* 3-83 to 3-104
Host name
represented by hostent structure, *SYS* 3-12E
Hostent structure
getting for host, *SYS* 3-13E
hostid program
displaying system unique identifier, *SYS* 1-6
hostname program
setting host name, *SYS* 1-6
hosts database
4.2BSD improvement, *SYS* 1-16
hosts.equiv file
description, *SYS* 5-49
hp.c device driver
4.2BSD improvement, *SYS* 5-14
htable program
converting NIC host data base, *SYS* 5-48
hunt program
defined, *GEN* 5-146
description, *GEN* 5-148
fgrep and, *GEN* 5-148
options list, *GEN* 5-148
timing, *GEN* 5-149
hw command (nroff/troff)
defined, *GEN* 5-69
hx command (me)
defined, *GEN* 5-41
hy command (nroff/troff)
defined, *GEN* 5-69
hy network interface driver
4.2BSD improvement, *SYS* 1-16
Hyphen
entering with text, *GEN* 5-22
Hyphenation (nroff/troff)
automatic, *GEN* 5-69
command list, *GEN* 5-52
Hyphenation indicator character
specifying, *GEN* 5-69
HZ parameter
description, *SYS* 5-122

I

i command (DC)
changing the base of input numbers, *GEN* 2-62
description, *GEN* 2-59
i command (ed)
defined, *GEN* 3-34
using, *GEN* 3-31 to 3-32
i command (ex)
description, *GEN* 3-89
i command (me)
defined, *GEN* 5-44
specifying italic font, *GEN* 5-36
I command (ms)
specifying italic, *GEN* 5-8
i command (sed)
See also a command (sed)
defined, *GEN* 3-109
I command (vi)
defined, *GEN* 3-79
i command (vi)
defined, *GEN* 3-81
description, *GEN* 3-58
i flag (Mail)
See also ignore option
defined, *GEN* 2-36
i flag (make)
defined, *PGM* 3-17
i flag (mkey)
ignoring lines, *GEN* 5-147
I option
changed to -i, *SYS* 1-6
i option
specifying directory search paths, *SYS* 1-6
i option (hunt)
defined, *GEN* 5-148
i option (inv)
defined, *GEN* 5-148
i option (nroff/troff)
defined, *GEN* 5-49

i-list
description, *GEN* 1-24
i-node
defined, *PGM* 4-10
file description and, *GEN* 1-24
i-number
defined, *GEN* 1-24
I/O
essentials of, *GEN* 1-23 to 1-24
I/O request
multiplexing among sockets and files, *SYS* 3-11
I/O system
description, *PGM* 4-8 to 4-10
overview, *PGM* 4-67 to 4-73
ibase
defined, *GEN* 2-44, 2-51
icheck program
4.2BSD improvement, *SYS* 1-19
ident parameter (config)
defined, *SYS* 5-79
Identifier
defined, *GEN* 2-51
kinds of, *GEN* 2-50
Identifier (as)
defined, *GEN* 6-53
ie command (nroff/troff)
defined, *GEN* 5-71
if command (Bourne shell)
description, *GEN* 4-13 to 4-14
if command (C shell)
See if/endif commands (C shell)
if command (Mail)
See if/endif commands (Mail)
if command (nroff/troff)
defined, *GEN* 5-71
if/endif commands (C shell)
See also else command (C shell)
See also then command (C shell)
defined, *GEN* 4-66, 4-68
forms of, *GEN* 4-56 to 4-57
if/endif commands (Mail)
description, *GEN* 2-31
restriction, *GEN* 2-31
if/endif commands (nroff/troff)
description, *GEN* 5-93 to 5-94
reference list, *GEN* 5-52
if/endif statement (as)
defined, *GEN* 6-59
if statement (as)
See if/endif statement (as)
if statement (awk)
defined, *PGM* 3-9
if statement (BC)
forming, *GEN* 2-54
restriction, *GEN* 2-47
writing, *GEN* 2-47
ifdef command (M4)
description, *PGM* 2-395
ifelse command (M4)
description, *PGM* 2-397
IFS variable
defined, *GEN* 4-12
ig command (nroff/troff)
defined, *GEN* 5-73
ignore command (Mail)
description, *GEN* 2-31
ignore option (Mail)
See also i flag (Mail)
defined, *GEN* 2-34
ignorecase option (ex)
description, *GEN* 3-98
ignoreeof variable (C shell)
defined, *GEN* 4-68
setting, *GEN* 4-41E
ignoreof option (Mail)
See also dot option
defined, *GEN* 2-34
ik driver
4.2BSD improvement, *SYS* 1-16
ik.c device driver
4.2BSD improvement, *SYS* 5-12
Ikonas frame buffer graphics device interface
See ik driver
Ikonas frame buffer graphics interface
See ik.c device driver
il network interface driver
4.2BSD improvement, *SYS* 1-16
Image
defined, *GEN* 1-26
imp network interface driver
4.2BSD improvement, *SYS* 1-16
IMP-11A LH/DH IMP interface
See css network driver
in command (me)
See also ix command (me)
entering, *GEN* 5-24
in command (nroff/troff)
defined, *GEN* 5-62
in_cksum.c file
4.2BSD improvement, *SYS* 5-13
include command (M4)
description, *PGM* 2-396
incr command (M4)
description, *PGM* 2-395

indent program
formatting C program source, *SYS* 1-6
Indention
command list, *GEN* 5-51
resetting base, *GEN* 5-45
specifying, *GEN* 5-24
specifyng with nroff/troff, *GEN* 5-62
Index
See Table of contents
index command (M4)
description, *PGM* 2-397
Index entry
specifying, *GEN* 5-43
Indexing
description, *GEN* 5-143 to 5-155
Indirect block
inode and, *SYS* 2-8
init program
4.2BSD improvement, *SYS* 1-19
description, *GEN* 1-30
init_main.c file
contents, *SYS* 5-8
init_sysent.c file
contents, *SYS* 5-8
initscr routine
defined, *PGM* 4-86
inode
allocations states, *SYS* 2-11
defined, *SYS* 2-8
disk space and, *SYS* 2-8
types of, *SYS* 2-11
Inode table
setting size, *SYS* 5-121
inode.h file
4.2BSD improvement, *SYS* 5-6
input
defined, *GEN* 4-68
Input base
DC and, *GEN* 2-62
Input mode
description, *GEN* 3-7
Input/output
See I/O
insch routine
defined, *PGM* 4-82
Insert command (ed)
See i command (ed)
insert command (ex)
See i command (ex)
insert command (vi)
See i command (vi)
insertln routine
defined, *PGM* 4-82
install command, *SYS* 5-55E
install script
installing software, *SYS* 1-6
int function (awk)
defined, *PGM* 3-8
Interlan Ethernet interface
See il network interface driver
Intermediate language (C compiler)
description, *PGM* 2-63 to 2-66
Internet address
binding, *SYS* 3-24 to 3-26
binding in Internet domain, *SYS* 3-8E
binding with wildcard address, *SYS* 3-25E
Internet port
printing, *SYS* 3-16E
Interprocess communication
description, *SYS* 3-5 to 3-28
transferring data, *SYS* 3-9E
Interprocess communication facilities
4.2BSD improvement, *SYS* 1-3
Interrupt message
description, *GEN* 3-9
Interrupt signal
See also oninvr command (C shell)
See also stty command (C shell)
creating, *GEN* 1-31
defined, *GEN* 4-68
ignoring, *GEN* 2-36
scripts and, *GEN* 4-59
intro system call
4.2BSD improvement, *SYS* 1-10
inv program
defined, *GEN* 5-146
description, *GEN* 5-147
options list, *GEN* 5-147
Inverted indexes
See Indexing
I/O library
restriction, *GEN* 2-15
ioctl system call
4.2BSD improvement, *SYS* 1-11
ioctl.h file
4.2BSD improvement, *SYS* 5-6
iostat
reporting kilobytes per second transferred for each disk, *SYS* 1-6

ip command (me)
See also np command
defined, *GEN* 5-40
specifying with label, *GEN* 5-30
IP command (ms)
indenting paragraphs, *GEN* 5-7
references and, *GEN* 5-7E
isprint library
4.2BSD improvement, *SYS* 1-14
it command (nroff/troff)
defined, *GEN* 5-65
Italic
See also Underlining
bolding, *GEN* 5-44
specifying, *GEN* 5-8
troff and, *GEN* 5-66
ix command (me)
defined, *GEN* 5-44

J

j command (ed)
joining lines, *GEN* 3-42, 3-43E
j command (ex)
description, *GEN* 3-90
J command (vi)
defined, *GEN* 3-79
j number register (nroff/troff)
defined, *GEN* 5-81
Job
defined, *GEN* 4-45, 4-69
determining current job, *GEN* 4-46
suspending, *GEN* 4-46
Job control command
See also bg command (C shell)
See also fg command (C shell)
See also kill command (C shell)
See also stop command (C shell)
defined, *GEN* 4-69
Job name
beginning character, *GEN* 4-46
Job number
defined, *GEN* 4-69
description, *GEN* 4-45
jobs command (C shell)
defined, *GEN* 4-69
displaying jobs, *GEN* 4-47E
Johnson, S.C.
Lint command, *PGM* 3-39 to 3-50
tour through portable C compiler, *PGM* 2-37 to 2-61
Yacc, *PGM* 3-79 to 3-111
join command (ex)
See j command (ex)
Joy, W.
C shell introduction, *GEN* 4-29 to 4-74
Joy, W., & Horton, M.
editing with vi, *GEN* 3-53 to 3-82
Ex Reference Manual, *GEN* 3-83 to 3-104
Joy, W., & Leffler, S.J.
4.2BSD on VAX/VMS, *SYS* 5-17 to 5-71
Joy, W., & others
4.2BSD Interprocess Communication Primer, *SYS* 3-5 to 3-28
4.2BSD System Manual, *PGM* 4-15 to 4-52
Berkeley Pascal User Manual, *PGM* 2-159 to 2-209
fast file system, *SYS* 1-23 to 1-38
networking implementation notes, *SYS* 3-29 to 3-57
Joyce, J., & Blau, R.
Edit tutorial, *GEN* 3-3 to 3-23
Justifying (nroff/troff)
command list, *GEN* 5-51
description, *GEN* 5-60 to 5-61

K

k command (DC)
description, *GEN* 2-59
scale value and, *GEN* 2-60
k command (ed)
marking a line, *GEN* 3-50E
k command (ex)
See also mark command (ex)
description, *GEN* 3-90
k escape sequence (nroff/troff)
description, *GEN* 5-68
k flag (mkey)
specifying number of keys, *GEN* 5-147
k number register (nroff/troff)
defined, *GEN* 5-81
Keep
See also Floating keep
defined, *GEN* 5-26
footnotes and, *GEN* 5-35 to 5-36
index entries and, *GEN* 5-35 to 5-36
text formatting commands for, *GEN* 5-15E

keep option (Mail)
defined, *GEN* 2-34
keepsave option (Mail)
See also nosave option
defined, *GEN* 2-35
kern_acct.c file
contents, *SYS* 5-8
kern_clock.c file
4.2BSD improvement. *SYS* 5-8
kern_descrip.c file
contents, *SYS* 5-8
kern_exec.c file
contents, *SYS* 5-8
kern_exit.c file
contents, *SYS* 5-8
kern_fork.c file
contents, *SYS* 5-8
kern_mman.c file
contents, *SYS* 5-8
kern_proc.c file
contents, *SYS* 5-8
kern_prot.c file
contents, *SYS* 5-8
kern_resource.c file
contents, *SYS* 5-8
kern_sign.c file
contents, *SYS* 5-8
kern_subr.c file
contents, *SYS* 5-8
kern_synch.c file
contents, *SYS* 5-8
kern_time.c file
contents, *SYS* 5-8
kern_xxx.c file
contents, *SYS* 5-8
Kernel
4.2BSD improvement, *SYS* 5-3 to 5-15
configuration, *SYS* 5-36 to 5-37
implementation, *PGM* 4-5 to 4-8
implementing devices, *SYS* 5-37
kernel.h file
4.2BSD improvement, *SYS* 5-5
Kernighan, B.W.
advanced editing with ed, *GEN* 3-37 to 3-52
introduction to ed, *GEN* 3-25 to 3-35
Ratfor language, *PGM* 2-111 to 2-122
troff tutorial, *GEN* 5-83 to 5-96
UNIX for beginners, *GEN* 2-3 to 2-16
Kernighan, B.W., & Cherry, L.L.
typesetting mathematics, *GEN* 5-97 to 5-104
Typesetting Mathematics - User's Guide, *GEN* 5-105 to 5-114
Kernighan, B.W., & Lesk, M.E.
computer-naided instruction for UNIX, *GEN* 6-3 to 6-16
Kernighan, B.W., & others
awk programming language, *PGM* 3-5 to 3-12
Kernighan, B.W., & Ritchie, D.M.
M4 macro processor, *PGM* 2-393 to 2-398
programming UNIX, *PGM* 1-3 to 1-24
Kessler, P.B., & others
Berkeley Pascal User Manual, *PGM* 2-159 to 2-209
Key
defined, *GEN* 5-147
selected by program, *GEN* 5-145
Key file
defined, *GEN* 5-145
Key letters
reference list, *GEN* 5-152
Key-making program
format used, *GEN* 5-145
Keyword
supplementing, *GEN* 5-150
Keyword (BC)
reserved
reference list, *GEN* 2-50
Keyword parameter
description, *GEN* 4-17 to 4-25
Keyword statement (as)
defined, *GEN* 6-56
reference list, *GEN* 6-59 to 6-60
KF command (ms)
moving blocks of text, *GEN* 5-9
kg driver
4.2BSD improvement, *SYS* 1-16
kgclock.c device driver
4.2BSD improvement, *SYS* 5-12
kgmon program
See also gmon.out file
4.2BSD improvement, *SYS* 1-19
Kill character
default, *GEN* 4-30
kill command (C shell)
background commands and, *GEN* 4-37
background jobs and, *GEN* 4-47E
defined, *GEN* 4-69

kill command (C shell) (Cont.)
killing processes, *GEN* 2-11
suspended jobs and, *GEN* 4-47
killpg library routine
See killpg system call
killpg system call
4.2BSD improvement, *SYS* 1-11
KL-11
See kg driver
Kowalski, T.J., & McKusick, M.K.
fsck, *SYS* 2-7 to 2-25
KS command (ms)
keeping text blocks together, *GEN* 5-9, 5-94E

L

L argument (nroff)
centering and, *GEN* 5-27
specifying, *GEN* 5-27
l command (DC)
programming DC, *GEN* 2-62
l command (ed)
backspaces and, *GEN* 3-37
description, *GEN* 3-37
long lines and, *GEN* 3-37
p command and, *GEN* 3-37
tabs and, *GEN* 3-37
l command (me)
centering list elements, *GEN* 5-27
defined, *GEN* 5-42
entering, *GEN* 5-25
specifying fill mode, *GEN* 5-26
specifying left justification, *GEN* 5-27
L command (vi)
defined, *GEN* 3-79
l flag (mkey)
specifying items to be ignored, *GEN* 5-147
L number register (nroff/troff)
defined, *GEN* 5-81
l option (C shell)
description, *GEN* 2-6
l option (hunt)
defined, *GEN* 5-148
L-devices file
defined, *SYS* 5-139
L-dialcodes file
defined, *SYS* 5-139
L.sys file
contents, *SYS* 5-135
defined, *SYS* 5-141
ownership of, *SYS* 5-138
Label (as)
See Name label; Numeric label
label command (sed)
defined, *GEN* 3-114
LABEL operator (C compiler)
defined, *PGM* 2-65
last
displaying remote host, *SYS* 1-6
lastcomm
indicating program activity, *SYS* 1-7
Layer, K., & others
Franz Lisp Manual, The, *PGM* 2-211 to 2-358
lc command (nroff/troff)
defined, *GEN* 5-66
LCK file
description, *SYS* 5-143
Leader character (nroff/troff)
setting, *GEN* 5-66
uninterpreted, *GEN* 5-66
Leadering
specifying with troff, *GEN* 5-88
Leading
See Vertical spacing
LEARN driver program
defined, *GEN* 6-3
description, *GEN* 2-6
directory structure, *GEN* 6-8
experience with students, *GEN* 6-8
introduction to UNIX, *GEN* 6-3 to 6-16
sequence of events, *GEN* 6-9
vi and, *SYS* 1-7
leaveok routine
defined, *PGM* 4-86
Leffler, S.J.
building 4.2BSD systems with config, *SYS* 5-73 to 5-105
improvements in 4.2BSD, *SYS* 1-3 to 1-21
kernel and 4.2BSD, *SYS* 5-3 to 5-15
Leffler, S.J., & Joy, W.N.
4.2BSD on VAX/VMS, *SYS* 5-17 to 5-71
Leffler, S.J., & others
4.2BSD Interprocess Communication Primer, *SYS* 3-5 to 3-28
4.2BSD System Manual, *PGM* 4-15 to 4-52
fast file system, *SYS* 1-23 to 1-38

Leffler, S.J., & others (Cont.)
networking implementation notes, *SYS* 3-29 to 3-57
left keyword (EQN), *GEN* 5-100E
len command (M4)
description, *PGM* 2-397
length function (awk)
defined, *PGM* 3-8
Leres, C., & Shoens, K.
Mail Reference Manual, *GEN* 2-17 to 2-41
Lesk, M.E.
formatting tables, *GEN* 5-115 to 5-131
inverted indexes, *GEN* 5-143 to 5-155
preparing documents with -ms, *GEN* 5-13 to 5-16
updating publication lists, *GEN* 5-155 to 5-162
using -ms macros with troff and nroff, *GEN* 5-5 to 5-12
Lesk, M.E., & Kernighan, B.W.
computer-aided instruction for UNIX, *GEN* 6-3 to 6-16
Lesk, M.E., & Nowitz, D.A.
a dial-up network of UNIX systems, *SYS* 5-123 to 5-129
Lesk, M.E., & Schmidt, E.
Lex program generator, *PGM* 3-113 to 3-125
Lex program generator
description, *PGM* 3-113 to 3-125
LG command (ms)
increasing type size, *GEN* 5-8
lg command (troff)
defined, *GEN* 5-66
libc.a library
remaking, *SYS* 5-120
libI77.a library
See f77 I/O library
Life game
program for, *PGM* 4-94E
Ligature (troff)
types available, *GEN* 5-66
limit command (C shell)
displaying current limitations, *GEN* 4-51E
setting limits, *GEN* 4-51E
Line
See Line drawing (nroff/troff)
Line dot
See Dot character (ed)
Line drawing (nroff/troff)
description, *GEN* 5-68
Line length (nroff/troff)
specifying, *GEN* 5-62, 5-86
Line printer
setting for serial lines, *PGM* 4-101
setting remote, *PGM* 4-101
Line printer control program
See lpc program
Line Printer Dameon
See lpd program
Line Printer Queue program
See lpq program
Line printer spooling system
devices supported, *PGM* 4-99, *SYS* 5-44
file list, *SYS* 5-44
setting up, *SYS* 5-44
Line printer spooling system (4.2BSD)
See also lpc program; pac program
4.2BSD improvement, *SYS* 1-4, 1-7, 1-18
controlling access, *PGM* 4-100 to 4-101
error messages, *PGM* 4-103 to 4-105
filters and, *PGM* 4-102
setting up, *PGM* 4-101 to 4-102
user manual, *PGM* 4-99 to 4-105
Line spacing
See Vertical spacing
Linking
description, *GEN* 1-21
Lint command
checking C programs, *PGM* 3-39 to 3-50
lint command
C and, *GEN* 2-15
creating libraries from C source code, *SYS* 1-7
LINT configuration file
using, *SYS* 5-88E
LINT file
4.2BSD improvement, *SYS* 5-11
LINTRUP request
See fcntl system call
lisp option (ex)
description, *GEN* 3-99
lisp option (vi)
setting, *GEN* 3-68
Lisp program
See also vlp program
4.2BSD improvement, *SYS* 1-7

Lisp program (Cont.)
editing with vi, *GEN* 3-68
List
defined, *GEN* 5-25
specifying in text, *GEN* 5-25
text formatting commands for, *GEN* 5-15E
text formatting commands for nested, *GEN* 5-15E
list command
See ls command (C shell)
List command (ed)
See l command (ed)
list command (ex)
description, *GEN* 3-90
list command (Mail)
description, *GEN* 2-31
list files command
See ls command (C shell)
list option (ex)
description, *GEN* 3-99
listen system call
4.2BSD improvement, *SYS* 1-11
incoming requests and, *SYS* 3-9E
ll command (me)
See also xl command (me)
defined, *GEN* 5-45
ll command (nroff/troff)
defined, *GEN* 5-62
resetting line length, *GEN* 5-86E
ln
creating symbolic links, *SYS* 1-7
lo command (me)
defined, *GEN* 5-45
lo network interface
4.2BSD improvement, *SYS* 1-16
load command (DC)
See l command (DC)
local command (Mail)
description, *GEN* 2-31
Local motion
defined, *GEN* 5-67
Location counter (as)
See also bss segment
defined, *GEN* 6-55
Locore.c file
4.2BSD improvement, *SYS* 5-13
locore.s file
4.2BSD improvement, *SYS* 5-14
installing device drive and, *SYS* 5-119
LOG file
description, *SYS* 5-142
log function (awk)
defined, *PGM* 3-8
Logging in
description, *GEN* 2-3 to 2-4
prerequisites, *GEN* 2-3
procedure, *GEN* 3-5
recording attempts, *SYS* 4-12
Logging out, *GEN* 3-8E
description, *GEN* 2-5
Login directory
startup file and, *GEN* 2-12
login file
See also logout file
background jobs and, *GEN* 4-48E
defined, *GEN* 4-69
logging in and, *GEN* 4-39, 4-39E
rlogin server and, *SYS* 1-7
telnetd server program and, *SYS* 1-7
Login shell
See also Script file
defined, *GEN* 4-69
logging in and, *GEN* 4-39
logout command
exiting from UNIX, *GEN* 3-8
logout command (C shell)
defined, *GEN* 4-69
logout file
See also login file
C shell and, *GEN* 4-39
defined, *GEN* 4-69
London, T.B., & Reiser, J.F.
regenerating system software, *SYS* 5-117 to 5-122
setting up UNIX/32V V1.0, *SYS* 5-107 to 5-115
longjmp library
old semantics and, *SYS* 1-15
longjump library
4.2BSD improvement, *SYS* 1-15
longname routine
defined, *PGM* 4-86
lookbib command
checking the data base, *GEN* 5-150
Loop
variables and, *GEN* 4-60
Low-level I/O
description, *PGM* 1-8 to 1-12
lp command (me)
defined, *GEN* 5-40
entering, *GEN* 5-29

LP command (ms)
specifying block paragraphs, *GEN* 5-5
lp.c device driver
4.2BSD improvement, *SYS* 5-12
lpc program
4.2BSD improvement, *SYS* 1-4, 1-18, 1-19
description, *PGM* 4-100
lpd program
description, *PGM* 4-99
requests understood
reference list, *PGM* 4-100
lpd server program
4.2BSD improvement, *SYS* 1-20
lpq program
4.2BSD improvement, *SYS* 1-7
description, *PGM* 4-100
lpr command (C shell)
defined, *GEN* 4-69
lpr program
lpd and, *PGM* 4-100
lprm program
4.2BSD improvement
description, *PGM* 4-100
lq command (me)
specifying quotation marks, *GEN* 5-38
ls command (C shell)
4.2 BSD improvement, *SYS* 1-7
defined, *GEN* 4-69
description, *GEN* 2-6
listing files in three columns, *GEN* 2-11
specifying numeric sort, *GEN* 4-32E
ls command (Mail)
displaying files on your terminal, *GEN* 2-10
ls command (me)
entering, *GEN* 5-23
ls command (nroff/troff)
defined, *GEN* 5-61
lseek system call
4.2BSD improvement, *SYS* 1-11
description, *PGM* 1-11
lt command (nroff/troff)
defined, *GEN* 5-70

M

m command (e)
reversing two adjacent lines, *GEN* 3-50E
m command (ed)
caution, *GEN* 3-50
defined, *GEN* 3-34
moving text, *GEN* 3-50E
using, *GEN* 3-32
m command (edit)
context search and, *GEN* 3-15
moving text, *GEN* 3-14
m command (ex)
description, *GEN* 3-90
M command (vi)
defined, *GEN* 3-79
m command (vi)
defined, *GEN* 3-81
m escape (Mail)
description, *GEN* 2-25
m option (nroff/troff)
defined, *GEN* 5-49
m option (uuclean)
defined, *SYS* 5-137
m option (uucp)
defined, *SYS* 5-132
m1 command (me)
defined, *GEN* 5-41
m2 command (me)
defined, *GEN* 5-41
m3 command (me)
defined, *GEN* 5-42
m4 command (me)
defined, *GEN* 5-42
M4 macro processor
arguments, *PGM* 2-395
arithmetic built-ins, *PGM* 2-395
command line format, *PGM* 2-393
conditionals, *PGM* 2-397
defining macros, *PGM* 2-393 to 2-395
description, *PGM* 2-393 to 2-398
manipulating files, *PGM* 2-396
manipulating strings, *PGM* 2-397
operation, *PGM* 2-393
printing, *PGM* 2-397
m4 macro processor
4.2BSD improvement, *SYS* 1-7
machdep.c file
4.2BSD improvement, *SYS* 5-14
machine file
4.2BSD improvement, *SYS* 5-4
Machine instruction statement (as)
syntax, *GEN* 6-60 to 6-63
machine type parameter (config)
defined, *SYS* 5-79
Macro (M4)
defining, *PGM* 2-393 to 2-395

Macro (nroff)
defined, *GEN* 5-35
defining, *GEN* 5-35E
naming, *GEN* 5-35
using, *GEN* 5-35E
Macro (nroff/troff)
arguments, *GEN* 5-63
defined, *GEN* 5-62
description, *GEN* 5-62 to 5-65
diversions, *GEN* 5-63
printing, *GEN* 5-73
traps, *GEN* 5-64
Macro (troff)
arguments and, *GEN* 5-92 to 5-93
arguments and blanks, *GEN* 5-93
arguments and trailing punctuation, *GEN* 5-92
Macro (vi)
See also Word abbreviation
types of, *GEN* 3-68
Macro definition (make), *PGM* 3-15E
defined, *PGM* 3-15
Macro-invocation trap (nroff/troff)
description, *GEN* 5-64
magic option (ex)
description, *GEN* 3-96
magic option (ex)
description, *GEN* 3-99
Magnetic tape
FORTRAN-77 and, *PGM* 2-84
Mail
adding to mail list, *GEN* 2-25
answering, *GEN* 2-19 to 2-20
C shell watching for, *GEN* 4-39E
canceling, *GEN* 2-18
changing the subject line, *GEN* 2-25
commands to be executed by the shell, *GEN* 2-28
defined, *GEN* 2-38
deleting, *GEN* 2-20
description, *GEN* 2-5
filing, *GEN* 2-24
format, *GEN* 2-37
forwarding, *GEN* 2-25
holding in system mailbox, *GEN* 2-31
including in other mail, *GEN* 2-25
indicating indirect recipients, *GEN* 2-25
keeping, *GEN* 2-35
keeping outgoing, *GEN* 2-35
length restricted, *GEN* 2-37
Mail (Cont.)
line width, *GEN* 2-37
maintaining groups of mail, *GEN* 2-23
message lists and user names, *GEN* 2-28
notification of, *GEN* 2-17
paging, *GEN* 2-20
process, *GEN* 2-17
protecting, *GEN* 2-34E
reading, *GEN* 2-18 to 2-19
reading in home directory, *GEN* 2-21
reading next, *GEN* 2-19
reading other people's, *GEN* 2-36
recovering deleted, *GEN* 2-30
saving related in a file, *GEN* 2-32
searching for subjects, *GEN* 2-28
sending, *GEN* 2-18
sending multiple messages, *GEN* 2-28
sending remote, *SYS* 5-126
sending source program text, *GEN* 2-33
sending to file, *GEN* 2-27
sending to folder, *GEN* 2-27
sending to list, *GEN* 2-21
sending to multiple users, *GEN* 2-18
sending to other machines, *GEN* 2-26 to 2-27
sending to programs, *GEN* 2-27
sending to user name, *GEN* 2-27
specifying mailbox, *GEN* 2-36
terms defined, *GEN* 2-38
writing to others online, *GEN* 2-5
mail command
abbreviating, *GEN* 2-20
description, *GEN* 2-31
uses of, *GEN* 2-18
Mail list
editing, *GEN* 2-25
Mail program
setting up, *SYS* 5-44
mail program
4.2BSD improvement, *SYS* 1-7
defined, *GEN* 4-69
escaping temporarily to command mode, *GEN* 2-26
escaping temporarily to shell, *GEN* 2-25
reading folders, *GEN* 2-23
reference manual, *GEN* 2-17 to 2-41

mail program (Cont.)
senting source program text, *GEN* 2-33
shell and, *GEN* 2-32
suspending, *GEN* 4-37E
using, *GEN* 2-17 to 2-41
Mail Reference Manual
See also Mail program
Mail routing facility
See sendmail
mail system
See also sendmail
MAIL variable
description, *GEN* 4-11
mailaddr
4.2BSD improvement, *SYS* 1-17
Mailbox
defined, *GEN* 2-38
mailrc file, *GEN* 2-21E
defined, *GEN* 2-21
specifying folder directory, *GEN* 2-23
make command
command line format, *PGM* 3-16
operation, *PGM* 3-16 to 3-17
make depend command
system source code and, *SYS* 5-77
make directory command
See mkdir command (C shell)
make program
See also makefile
4.2BSD improvement, *SYS* 1-7
C and, *GEN* 2-15
defined, *GEN* 4-69
description, *PGM* 3-13 to 3-21
description file for, *PGM* 3-18 to 3-20
maintaining related files, *GEN* 4-53
operation, *PGM* 3-13 to 3-15
suffix list, *PGM* 3-17
transformation paths
summary, *PGM* 3-17
warnings, *PGM* 3-20
MAKEDEV script
See also MAKEDEV.local file
4.2BSD improvement, *SYS* 1-20
makefile
See also make program
defined, *GEN* 4-69
description, *GEN* 4-53
modifying for uucp, *SYS* 5-139
makefile.vax file
contents, *SYS* 5-11
makelinks command
source modules and, *SYS* 5-78
maketemp command (M4)
description, *PGM* 2-396
man command (Bourne shell)
printing the UNIX manual, *GEN* 4-15
printing UNIX manual, *GEN* 4-16F
man command (C shell)
accessing online programmer's manual, *GEN* 4-63E, 4-69E
using, *GEN* 2-6
Manual
defined, *GEN* 4-69
map command (ex)
See also unmap command (ex)
description, *GEN* 3-90
Maranzano, J.F., & Bourne, S.R.
ADB debugging program, *PGM* 3-51 to 3-77
Margin number
setting, *GEN* 5-44
mark command (ex)
See also k command (ex)
description, *GEN* 3-90
Mass storage
UNIX interfaces, *SYS* 1-36
MASSBUS
description, *SYS* 5-18
specifying, *SYS* 5-19
MASTER mode
description, *SYS* 5-135
Mathematics
text formatting commands for, *GEN* 5-14E
typesetting, *GEN* 5-97 to 5-104, 5-105 to 5-114
MAXMEM parameter
description, *SYS* 5-121
MAXUMEM parameter
See also MAXMEM parameter
description, *SYS* 5-121
MAXUPRC parameter
description, *SYS* 5-121
maxusers parameter (config)
defined, *SYS* 5-79
mba.c device driver
4.2BSD improvement, *SYS* 5-14
mbox command (Mail)
abbreviating, *GEN* 2-22
description, *GEN* 2-31
saving unread mail, *GEN* 2-22

mbox file
mail and, *GEN* 2-20
system mailbox and, *GEN* 2-20
mbuf.h file
4.2BSD improvement, *SYS* 5-5
mc command (nroff/troff)
defined, *GEN* 5-72
McKusick, M.K., & Kowalski, T.J.
fsck, *SYS* 2-7 to 2-25
McKusick, M.K., & others
4.2BSD System Manual, PGM 4-15 to 4-52
Berkeley Pascal User Manual, PGM 2-159 to 2-209
fast file system, *SYS* 1-23 to 1-38
McMahon, L.E.
sed stream editor and, *GEN* 3-105 to 3-114
me macro package
initializing, *GEN* 5-40
naming convention, *GEN* 5-39
predefined strings, *GEN* 5-47
reference manual, *GEN* 5-39 to 5-48
Me Reference Manual, GEN 5-39
See also me macro package
mem.c file
4.2BSD improvement, *SYS* 5-14
Memorandum
text formatting commands for, *GEN* 5-14E
mesg option (ex)
description, *GEN* 3-99
Message
See also Mail
defined, *GEN* 2-38
Message list
defined, *GEN* 2-28, 2-38
Metacharacters (Bourne shell)
defined, *GEN* 4-5
quoting, *GEN* 4-5
quoting a string, *GEN* 4-5E
quoting mechanisms, *GEN* 4-20F
reference list, *GEN* 4-27
Metacharacters (C shell)
defined, *GEN* 4-69
description, *GEN* 4-32
reference list, *GEN* 4-62
using with command arguments, *GEN* 4-35
Metacharacters (ed)
character classes and, *GEN* 3-41
deleting, *GEN* 3-38
Metacharacters (ed) (Cont.)
delimiting text for s command, *GEN* 3-39
editing with, *GEN* 3-37 to 3-43
entering, *GEN* 3-33
reference list, *GEN* 3-33
searching for, *GEN* 3-39, 3-41
Metacharacters (ed) (ed)
combining, *GEN* 3-40
description, *GEN* 3-38 to 3-42
Metacharacters (ex)
X and, *GEN* 3-96
Metacharacters (me)
reference list, *GEN* 5-47
Metacharacters (nroff/troff)
specifying, *GEN* 5-79
Metacharacters (troff)
automatically translated, *GEN* 5-86
command list, *GEN* 5-96
entering, *GEN* 5-86
metoo option (Mail)
defined, *GEN* 2-35
MFLAGS macro
supplying flags to make, *SYS* 1-7
mille game
4.2BSD improvement, *SYS* 1-17
Mini-root file system
booting from, *SYS* 5-25
copying, *SYS* 5-24
Minus sign
translating for troff, *GEN* 5-86
mk command (nroff/troff)
See also rt command (nroff/troff); sp command (nroff/troff)
defined, *GEN* 5-60
mkdir command
4.2BSD improvement, *SYS* 1-7
creating directories, *GEN* 2-10
mkdir command (C shell)
creating a directory, *GEN* 4-48
defined, *GEN* 4-70
mkdir system call
4.2BSD improvement, *SYS* 1-11
mkey program
defined, *GEN* 5-146
description, *GEN* 5-147
mkfs program
See newfs program
4.2BSD improvement, *SYS* 1-20
mman.h file
future plans and, *SYS* 5-5
Modifier (C shell)
See also Command substitution

Modifier (C shell) (Cont.)
defined, *GEN* 4-70
description, *GEN* 4-57
restriction, *GEN* 4-57n
more program
defined, *GEN* 4-70
paging mail, *GEN* 2-20
terminal screen and, *GEN* 4-37
Morris, R., & Cherry, L.
BC and, *GEN* 2-43 to 2-55
DC and, *GEN* 2-57 to 2-64
Morris, R., & Thompson, K.
password system, *SYS* 4-7 to 4-12
mos
old version of -ms, *GEN* 5-17
Mosher, D., & others
4.2BSD System Manual, *PGM* 4-15 to 4-52
mount command
unprivileged users and, *SYS* 4-5
mount program
4.2BSD improvement, *SYS* 1-20
mount.h file
4.2BSD improvement, *SYS* 5-6
Move command (ed)
See m command (ed)
move command (edit)
See m command
move command (ex)
See m command (ex)
move routine
defined, *PGM* 4-83
mpx system call
See socket system call and related system calls
ms macro package
See also -mos
4.2BSD improvement, *SYS* 1-18
CAI script for, *GEN* 6-7
command reference list, *GEN* 5-11
default settings, *GEN* 5-9
entering cover sheet, *GEN* 5-5
entering first page, *GEN* 5-5
entering page footer, *GEN* 5-6
entering page heading, *GEN* 5-6
entering paragraphs, *GEN* 5-5
entering section heads, *GEN* 5-6
keeping text blocks together, *GEN* 5-9
order for input commands, *GEN* 5-12F
preparing documents, *GEN* 5-13 to 5-16
ms macro package (Cont.)
printing files on the terminal, *GEN* 5-9E
register name reference list, *GEN* 5-11
revised version, *GEN* 5-17 to 5-19
specifying column format, *GEN* 5-6
using with troff and nroff, *GEN* 5-5 to 5-12
ms package
description, *GEN* 2-12
formatting a document with nroff, *GEN* 2-13
formatting a document with troff, *GEN* 2-12
MSGBUFS parameter
description, *SYS* 5-122
mt
showing state of tape drive, *SYS* 1-7
mtab
4.2BSD improvement, *SYS* 1-16
Multiplication
DC and, *GEN* 2-61
Multiplicative operator
description, *GEN* 2-52
Multitasking
description, *GEN* 1-29
MV command
renaming a file, *GEN* 2-7
mv program
4.2BSD improvement, *SYS* 1-7
mv program (ed)
renaming a file, *GEN* 3-47
mvcur routine
defined, *PGM* 4-88
mvwin routine
defined, *PGM* 4-86

N

n command (ex)
description, *GEN* 3-90
n command (sed)
defined, *GEN* 3-108
N command (vi)
See also n command (vi)
defined, *GEN* 3-79
n command (vi)
See also N command (vi)
defined, *GEN* 3-81
N flag (Mail)
See also noheader option

N flag (Mail) (Cont.)
defined, *GEN* 2–36
n flag (Mail)
defined, *GEN* 2–36
n flag (make)
defined, *PGM* 3–17
n flag (mkey)
ignoring words, *GEN* 5–147
n flag (sed)
defined, *GEN* 3–106
n option
specifying numeric sort, *GEN* 4–32
n option (inv)
defined, *GEN* 5–148
n option (nroff/troff)
defined, *GEN* 5–49
n option (uuclean)
defined, *SYS* 5–137
n1 command (me)
defined, *GEN* 5–44
n2 command (me)
defined, *GEN* 5–44
Name label (as)
defined, *GEN* 6–55
NAME operator (C compiler)
defined, *PGM* 2–66
Named expression
defined, *GEN* 2–51
nami routine
See also nami.h file
nami.h file
4.2BSD improvement, *SYS* 5–5
NBUF parameter
description, *SYS* 5–121
NCALL parameter
description, *SYS* 5–122
NCARGS parameter
description, *SYS* 5–122
NCLIST parameter
description, *SYS* 5–122
ND command (ms)
cover sheet and, *GEN* 5–9
ne command (nroff/troff)
defined, *GEN* 5–59
NEQN program
See also EQN program
description, *GEN* 5–33
formatting mathematics, *GEN* 2–13
net library
4.2BSD improvement, *SYS* 1–15
net program
UNIX distribution and, *SYS* 1–7
netstat program
displaying network statistics, *SYS* 1–7, 5–51E
displaying routing table contents, *SYS* 5–51E
Network
See Dial-up network
See uucp system
troubleshooting, *SYS* 5–57
Network data base
files list, *SYS* 5–48
Network library routines
description, *SYS* 3–12 to 3–16
Network name
represented by netent structure, *SYS* 3–13E
Network server program
included with system, *SYS* 5–50T
started up automatically at boot time, *SYS* 5–49T
network server program
reference list, *SYS* 5–49
Network Systems Hyperchannel Adapter
See hy network interface driver
Networking
implementation, *SYS* 3–29 to 3–57
networks database
4.2BSD improvement, *SYS* 1–16
newfs program
See also mkfs program
4.2BSD improvement, *SYS* 1–18, 1–20
newgrp command
See Group set
newwin routine
defined, *PGM* 4–86
next command (ex)
See n command (ex)
next command (Mail)
abbreviating, *GEN* 2–31
description, *GEN* 2–31
next statement (awk)
defined, *PGM* 3–9
NF variable (awk)
determining number of fields, *PGM* 3–6
NFILE parameter
description, *SYS* 5–121
NH command (ms)
entering section heads, *GEN* 5–6E
specifying numbered section heads, *GEN* 5–6

nh command (nroff/troff)
defined, *GEN* 5-69
NIC host data base
retrieving, *SYS* 5-48E
NINODE parameter
description, *SYS* 5-121
nl routine
defined, *PGM* 4-87
NLABEL operator (C compiler)
defined, *PGM* 2-64
nm command (nroff/troff)
defined, *GEN* 5-70
NMOUNT parameter
description, *SYS* 5-121
nn command (nroff/troff)
defined, *GEN* 5-70
Nobreak control character
changing, *GEN* 5-67
noclobber variable (C shell)
defined, *GEN* 4-70
protecting files and, *GEN* 4-41
NOFILE parameter
description, *SYS* 5-121
noglob variable (C shell), *GEN* 4-56E
defined, *GEN* 4-70
noheader option (Mail)
See also -N flag
See also quiet option
defined, *GEN* 2-35
nosave option (Mail)
See also keepsave option
defined, *GEN* 2-35
notify command (C shell)
See also notify variable
defined, *GEN* 4-70
reporting job complete, *GEN* 4-47
notify variable (C shell)
See also notify command (C shell)
background jobs and, *GEN* 4-45
Nowitz, D.A.
implementing uucp, *SYS* 5-131 to 5-144
Nowitz, D.A., & Lesk, M.E.
a dial-up network of UNIX systems, *SYS* 5-123 to 5-129
np command (me)
defined, *GEN* 5-40
numbering paragraphs automatically, *GEN* 5-31E
NPROC parameter
description, *SYS* 5-121
nr command (me)
indenting sections, *GEN* 5-32E
nr command (me) (Cont.)
specifying with li, *GEN* 5-30
nr command (nroff/troff)
defined, *GEN* 5-65
NR variable (awk)
determining current record number, *PGM* 3-5
nroff text processor
See also nroff/troff text processor
See also troff text processor
calling, *GEN* 5-21E
defined, *GEN* 2-12
device resolution and, *GEN* 5-56
entering text, *GEN* 5-22
formatting a document with -ms, *GEN* 2-13
function, *GEN* 5-22
invoking, *GEN* 5-49
stopping printer to change paper, *GEN* 5-49
writing papers using -me, *GEN* 5-21 to 5-38
nroff/troff text processor
See also -ms macros
See also nroff text processor
See also troff text processor
-ms macros and, *GEN* 5-5 to 5-12
boxing words, *GEN* 5-69
breaking a line, *GEN* 5-60
character set, *GEN* 5-57
character translation, *GEN* 5-66
concealed newlines and, *GEN* 5-67
contol characters beginning lines, *GEN* 5-60
defined, *GEN* 5-49
description, *GEN* 2-12
error messages, *GEN* 5-73
input, *GEN* 5-56
justifying text, *GEN* 5-61
marking horizontal space, *GEN* 5-68
numbering output lines, *GEN* 5-70
numerical expressions, *GEN* 5-57
numerical parameters, *GEN* 5-56
post processors and, *GEN* 5-50
preprocessors and, *GEN* 5-50
specifying conditional input, *GEN* 5-71
specifying indention, *GEN* 5-62
specifying line length, *GEN* 5-62
specifying page margins, *GEN* 5-74E

nroff/troff text processor (Cont.)
specifying vertical spacing, *GEN* 5-61
switching environment, *GEN* 5-71
transparent throughput, *GEN* 5-67
transposing characters, *GEN* 5-67
underlining words, *GEN* 5-69
user's manual, *GEN* 5-49 to 5-81
writing paragraph macros, *GEN* 5-75E
Nroff/Troff User's Manual
update, *GEN* 5-81
Nroff/Troff User's Manual, *GEN* 5-49 to 5-81
See also nroff/troff text processor
ns command (nroff/troff)
defined, *GEN* 5-62
NTEXT parameter
description, *SYS* 5-122
nu command (edit)
printing text with line numbers, *GEN* 3-11
nu command (ex)
description, *GEN* 3-91
NULL
defined, *PGM* 1-21
NULL operator (C compiler)
defined, *PGM* 2-66
Null statement (as)
defined, *GEN* 6-55
Number
internal representation in DC, *GEN* 2-59
right justifying with troff, *GEN* 5-87
number command (DC)
descripton, *GEN* 2-57
number command (edit)
See nu command (edit)
number command (ex)
See nu command (ex)
number option (ex)
description, *GEN* 3-99
Number register (nroff/troff)
See also nr command (nroff/troff)
See also specific registers
command list, *GEN* 5-52, 5-55
description, *GEN* 5-65 to 5-66
Number register (troff)
description, *GEN* 5-91 to 5-92
predefined, *GEN* 5-91
Numeric label (as)
defined, *GEN* 6-55
nx command (nroff/troff)
defined, *GEN* 5-72

O

o command (DC)
changing the output base, *GEN* 2-62
description, *GEN* 2-59
o command (ex)
See also open option
description, *GEN* 3-91
line editing and, *GEN* 3-85
o command (nroff/troff)
description, *GEN* 5-68
O command (Rogue)
using, *GEN* 6-23
O command (vi)
See also o command (vi)
See also slowopen option
defined, *GEN* 3-79
o command (vi)
See also O command (vi)
defined, *GEN* 3-81
o option (hunt)
defined, *GEN* 5-148
o option (nroff/troff)
defined, *GEN* 5-49
obase
defined, *GEN* 2-44, 2-51
Octal
converting to decimal, *GEN* 2-44
od
4.2BSD improvement, *SYS* 1-7
of command (me)
defined, *GEN* 5-41
of filter
calling, *PGM* 4-102E
printers and, *PGM* 4-102
OF macro
specifying page footers, *GEN* 5-19
OFS variable
defined, *PGM* 3-6
oh command (me)
defined, *GEN* 5-41
OH macro
specifying page headings, *GEN* 5-19
oldcsh
4.2BSD and, *SYS* 1-7
onintr command (C shell)
See also Interrupt signal
defined, *GEN* 4-70

open command (ex)
See o command ex)
open function
See also open function
description, *PGM* 1-10
open option (ex)
description, *GEN* 3-99
open system call
4.2BSD improvement, *SYS* 1-11
Operators
available, *GEN* 2-43
optim routine (C compiler)
description, *PGM* 2-66 to 2-67
optim routine (C shell)
See also unoptim routine (C shell)
optimize option (ex)
description, *GEN* 3-99
Option (C shell)
combining, *GEN* 2-6
Option (ex)
See also specific options
reference list, *GEN* 3-97 to 3-101
Option (Mail)
See also specific options
defined, *GEN* 2-38
reference list, *GEN* 2-33 to 2-36, 2-40T
setting, *GEN* 2-32, 2-32E
Option (nroff/troff)
invoking, *GEN* 5-50
reference list, *GEN* 5-49 to 5-50
Option (vi)
See also specific options
listing values, *GEN* 3-65
reference list, *GEN* 3-65
setting, *GEN* 3-65
setting automatically, *GEN* 3-65
options parameter (config)
defined, *SYS* 5-79
ORS variable
defined, *PGM* 3-6
os command (nroff/troff)
defined, *GEN* 5-62
Ossanna, J.F.
Nroff/Troff User's Manual, GEN 5-49 to 5-81
Out of band data
description, *SYS* 3-23
flushing I/O on receipt, *SYS* 3-23F
Output
defined, *GEN* 4-70
Output base
DC and, *GEN* 2-62
over keyword (EQN)
specifying fractions, *GEN* 5-99E
overlay routine
defined, *PGM* 4-83
Overstrike command (nroff/troff)
See o command (nroff/troff)
Overstriking
creating with troff, *GEN* 5-88
overwrite routine
defined, *PGM* 4-83

P

p command (DC)
descripton, *GEN* 2-58
p command (ed)
defined, *GEN* 3-34
printing a line, *GEN* 3-28
printing all lines, *GEN* 3-28
printing last line, *GEN* 3-28
printing lines, *GEN* 3-27
stopping, *GEN* 3-28
using, *GEN* 3-27 to 3-28
p command (edit)
printing buffer contents, *GEN* 3-10
u command and, *GEN* 3-16
p command (ex)
description, *GEN* 3-91
P command (me)
defined, *GEN* 5-46
specifying front matter, *GEN* 5-33
p command (sed)
defined, *GEN* 3-111
P command (vi)
See also p command (vi)
defined, *GEN* 3-79
p command (vi)
See also P command (vi)
defined, *GEN* 3-81
p escape (Mail)
description, *GEN* 2-24
p flag (make)
defined, *PGM* 3-17
p flag (sed)
defined, *GEN* 3-110
p macro (me)
defined, *GEN* 5-41
P number register (nroff/troff)
defined, *GEN* 5-81
p option (hunt)
defined, *GEN* 5-149
p option (inv)
defined, *GEN* 5-148

p option (troff)
defined, *GEN* 5–50
p option (uuclean)
defined, *SYS* 5–137
pa command (me)
defined, *GEN* 5–44
pac program
4.2BSD improvement, *SYS* 1–18, 1–20
Page
command list, *GEN* 5–51
formatting the last page with a macro, *GEN* 5–77E
printing specific, *GEN* 5–49
setting margins with nroff/troff, *GEN* 5–74E
specifying blank, *GEN* 5–44
specifying new, *GEN* 5–23
Page commands
description, *GEN* 5–59
Page footer
entering in text file, *GEN* 5–6
specifying, *GEN* 5–70
specifying for multiple columns with a macro, *GEN* 5–75E
specifying with troff, *GEN* 5–91
varying on alternate pages, *GEN* 5–19
Page header
entering in text file, *GEN* 5–6
specifying for multiple columns with a macro, *GEN* 5–75E
specifying formats for alternating, *GEN* 5–71
specifying with troff, *GEN* 5–90
Page heading
specifying, *GEN* 5–70
varying on alternate pages, *GEN* 5–19
Page layout
specifying, *GEN* 5–23
Page number
setting arabic, *GEN* 5–44
setting roman, *GEN* 5–44
specifying, *GEN* 5–59, 5–91
specifying for appendix, *GEN* 5–46
specifying for chapter, *GEN* 5–46
Page offset (nroff/troff)
specifying, *GEN* 5–59
Page trap (nroff/troff)
description, *GEN* 5–64
pagesize program
printing system page size, *SYS* 1–7
Paging
defined, *GEN* 3–13
versus scrolling, *GEN* 3–56
Paper
formatting, *GEN* 5–34F
Paragraph, *GEN* 5–40
-me restrictions, *GEN* 5–40
creating decorative initial capital with troff, *GEN* 5–86
editing with vi, *GEN* 3–61
entering in text file, *GEN* 5–5
indenting, *GEN* 5–7 to 5–8
numbering automatically, *GEN* 5–31
specifying, *GEN* 5–22
specifying block format, *GEN* 5–29
specifying hanging indent format, *GEN* 5–29
specifying hanging indent format with a macro, *GEN* 5–75E
specifying indention, *GEN* 5–30
specifying indention amount, *GEN* 5–39E
vi definition, *GEN* 3–61
writing a macro for, *GEN* 5–75E
paragraph option (ex)
description, *GEN* 3–99
param.c file
contents, *SYS* 5–11, 5–103
param.h file
See also kernel.h file
4.2BSD improvement, *SYS* 5–6, 5–13
Parentheses (BC)
primitive expression and, *GEN* 2–51
Parentheses (EQN)
typesetting in proper size, *GEN* 5–100E
Pascal programming language
See Berkeley Pascal programming language
Passive system
defined, *SYS* 5–123
passwd
concurrent updates to password file and, *SYS* 1–8
Password
entering, *GEN* 3–5
Password entry program
predictable passwords and, *SYS* 4–10
random numbers and, *SYS* 4–11

Password file
restricting users, *GEN* 1–31
security and, *SYS* 4–8
Password system
history, *SYS* 4–7 to 4–12
Pasting and cutting
See m command (ed)
PATH variable (Bourne shell)
description, *GEN* 4–11 to 4–12
path variable (C shell)
See also rehash command (C shell)
default value, *GEN* 4–40
defined, *GEN* 4–40, 4–70
Pathname
See also Absolute pathname
defined, *GEN* 2–9, 4–71
description, *GEN* 4–33
Pattern (awk)
description, *PGM* 3–6 to 3–7
Pattern space
defined, *GEN* 3–106
pc
4.2BSD improvement, *SYS* 1–8
pc command (nroff/troff)
defined, *GEN* 5–70
pc/pi
4.2BSD improvement, *SYS* 1–8
pcb.h file
4.2BSD improvement, *SYS* 5–14
pcl network interface driver
4.2BSD improvement, *SYS* 1–16
pd command (me)
defined, *GEN* 5–43
pdx debugger
pi and, *SYS* 1–8
Period
See Dot character (ed)
perror function
description, *PGM* 1–12
perror library
4.2BSD improvement, *SYS* 1–15
pg flag
collecting information for gprof, *SYS* 1–5
pg option
creating images for gprof, *SYS* 1–6
phones database
See also tip program
4.2BSD improvement, *SYS* 1–17
Phototypesetter
defined, *GEN* 5–98
stopping automatically to reload, *GEN* 5–49
Phototypesetting
See nroff/troff text processor
PHYSPAGES parameter
description, *SYS* 5–121
pi command (nroff)
defined, *GEN* 5–72
Picture System 2 graphics device
See ps driver
piles program (EQN)
description, *GEN* 5–100
Pipe
defined, *GEN* 1–26, 2–11, *PGM* 1–14
description, *GEN* 2–11, *PGM* 1–14 to 1–17
optimal size, *SYS* 1–28
programs and, *GEN* 2–11
pipe system call
description, *PGM* 1–15 to 1–17
Pipeline, *GEN* 4–4E
combining command input/output, *GEN* 4–32
defined, *GEN* 2–11, 4–4, 4–71
description, *GEN* 4–32 to 4–33
elements in, *GEN* 2–11
files read from terminal and, *GEN* 2–11
pl command (nroff/troff)
defined, *GEN* 5–59
Plain data block
defined, *SYS* 2–12
pm command (nroff/troff)
defined, *GEN* 5–73
pn command (nroff/troff)
defined, *GEN* 5–59
po command (nroff/troff)
defined, *GEN* 5–59
setting left margin, *GEN* 5–86E
Point size
changing, *GEN* 5–38, 5–58
defaults, *GEN* 5–38
setting, *GEN* 5–84
pop directory command
See popd command (C shell)
popd command (C shell)
See also pushd command (C shell)
defined, *GEN* 4–71
without argument, *GEN* 4–49
Port
defined, *GEN* 4–71
Port number
algorithm for selecting, *SYS* 3–26
overriding selection algorithm, *SYS* 3–26E

Portable C Compiler
description, *PGM* 2–37 to 2–61
Posting file
defined, *GEN* 5–145
Pound sign
See Sharp character
pp command (me)
See also ip command (me)
See also lp command (me)
defined, *GEN* 5–40
description, *GEN* 5–22
meaning of, *GEN* 2–12
pr command (C shell)
defined, *GEN* 4–71
printing files, *GEN* 2–7
printing files in three columns, *GEN* 2–11
pre command (edit)
recovering files, *GEN* 3–22
Preface
formatting, *GEN* 5–34F
Preliminary text
See Front matter
preserve command (edit)
See pre command (edit)
preserve command (ex)
description, *GEN* 3–91
preserve command (Mail)
See also hold command (Mail)
abbreviating, *GEN* 2–22
description, *GEN* 2–31
keeping mail in your system mailbox, *GEN* 2–21
primes program
4.2BSD improvement, *SYS* 1–17
Primitive expression
description, *GEN* 2–51
Print command
See p command
print command (awk)
description, *PGM* 3–6
print command (edit)
See p command (edit)
print command (ex)
See p command (ex)
print command (Mail)
See also ignore command (Mail)
description, *GEN* 2–29
ignored fields and, *GEN* 2–31
Print file
UNIX and, *PGM* 2–83
print working directory command
See pwd command (C shell)
printcap file
4.2BSD improvement, *SYS* 1–17
creating, *PGM* 4–101
printenv command (C shell)
See also setenv command (C shell)
defined, *GEN* 4–71
printf function
See also fprintf function
output and, *PGM* 1–4
printf statement (awk)
formatting output, *PGM* 3–6
printw routine
defined, *PGM* 4–83
proc.h file
4.2BSD improvement, *SYS* 5–7
Process
See also ps command (C shell)
See also System process
See also User process
defined, *GEN* 1–26, 4–71
maximum active, *SYS* 5–121
maximum per user, *SYS* 5–121
setting maximum files for, *SYS* 5–121
space for, *SYS* 5–121
stopping, *GEN* 2–11
syncronizing, *GEN* 1–27
terminating, *GEN* 1–27
Process control
data structure, *PGM* 4–6F
description, *PGM* 4–5 to 4–6
Process number
defined, *GEN* 2–11
determining, *GEN* 2–11
Process stack
setting growth increment, *SYS* 5–121
setting initial size, *SYS* 5–121
Process time accounting
summarizing, *SYS* 5–56
PROFIL operator (C compiler)
defined, *PGM* 2–65
profil system call
4.2BSD improvement, *SYS* 1–12
profile file
login and, *GEN* 4–6
shell and, *GEN* 2–12
Profiled system
description, *SYS* 5–78
PROG operator (C compiler)
defined, *PGM* 2–64
Program
See also Command (C shell)

Program (Cont.)
defined, *GEN* 3-3, 4-71
editing with vi, *GEN* 3-67
executing, *GEN* 1-26
executing from another, *PGM* 1-12
maintaining with make, *PGM* 3-13 to 3-21
running simultaneously, *GEN* 2-11
running two with one command line, *GEN* 2-11
saving output, *GEN* 2-11
setting maximum executing, *SYS* 5-122
stopping, *GEN* 2-4, 2-11
Programmer's manual
See Manual
Programming
reading list, *GEN* 2-16
tools for, *GEN* 2-14 to 2-15
translating a language, *GEN* 2-15
Prompt
defined, *GEN* 4-71
Prompt character
defined, *GEN* 2-4
prompt option (ex)
description, *GEN* 3-99
Protection mode
description, *PGM* 1-10
Proteon proNET ring network controller
See vv network interface driver
Protocol name
represented by protoent structure, *SYS* 3-13, 3-14E
protocol switch table
See also protosw.h file
protocols database
4.2BSD improvement, *SYS* 1-17
protosw.h file
4.2BSD improvement, *SYS* 5-5
ps command (C shell)
See also Process
4.2BSD improvement, *SYS* 1-8
defined, *GEN* 4-72
determining the process number, *GEN* 2-11
displaying all programs running, *GEN* 2-11
displaying unstarted background jobs, *GEN* 4-48
ps command (troff)
defined, *GEN* 5-58
ps command (troff) (Cont.)
setting point size, *GEN* 5-84
ps driver
4.2BSD improvement, *SYS* 1-16
ps.c device driver
4.2BSD improvement, *SYS* 5-12
PS1 variable
defined, *GEN* 4-12
PS2 variable
defined, *GEN* 4-12
Pseudo device
specifying, *SYS* 5-82
Pseudo terminal
creating, *SYS* 5-48E
description, *SYS* 3-24
remote login sessions and, *SYS* 3-24
Pseudo-font
description, *GEN* 5-37
restriction, *GEN* 5-37
psignal library
4.2BSD improvement, *SYS* 1-15
pstat program
4.2BSD improvement, *SYS* 1-20
ptx program
defined, *GEN* 2-13
pty driver
4.2BSD improvement, *SYS* 1-16
pu command (ex)
description, *GEN* 3-91
Publication list
indexing, *GEN* 5-143 to 5-155
updating, *GEN* 5-155 to 5-162
pup_cksum.c file
4.2BSD improvement, *SYS* 5-13
purchar function
output and, *PGM* 1-4
push directory command
See pushd command (C shell)
push directory command (C shell)
See pushd command
pushd command (C shell)
See also cd command (C shell)
See also popd command (C shell)
defined, *GEN* 4-70
saving name of previous directory, *GEN* 4-49
without argument, *GEN* 4-49
put command (ex)
See pu command (ex)
putc macro
See also fflush function
defined, *PGM* 1-6

pwd command (C shell)
See also dirs command (C shell)
4.2BSD improvement, *SYS* 1–8
defined, *GEN* 4–72
print your directory name, *GEN* 2–9
working directory pathname and, *GEN* 4–48E
PX macro
description, *GEN* 5–18

Q

Q command
quitting ed, *GEN* 2–6
q command (DC)
descripton, *GEN* 2–58
q command (ed)
defined, *GEN* 3–34
using, *GEN* 3–26
q command (edit)
exiting without saving edits, *GEN* 3–13
using, *GEN* 3–8
q command (ex)
See also wq command (ex)
description, *GEN* 3–91
q command (me)
defined, *GEN* 5–42, 5–44
entering, *GEN* 5–25
specifying quoted text, *GEN* 5–38
q command (sed)
defined, *GEN* 3–114
Q command (vi)
defined, *GEN* 3–79
q flag (make)
defined, *PGM* 3–17
q option (nroff/troff)
defined, *GEN* 5–49
qsort library
4.2BSD improvement, *SYS* 1–15
Question mark character (C shell)
description, *GEN* 4–34
Question mark character (DC)
description, *GEN* 2–59
pattern matching and, *GEN* 2–8
Question mark character (ed)
context search and, *GEN* 3–43
quiet option (Mail)
See also noheader option
defined, *GEN* 2–35
Quit command (ed)
See q command (ed)
quit command (edit)
See q command (edit)
quit command (ex)
See q command (ex)
quit command (Mail)
abbreviating, *GEN* 2–22
description, *GEN* 2–31
saving typed mail, *GEN* 2–22
Quit signal
defined, *GEN* 4–72
terminating a program, *GEN* 4–37
quit statement (BC)
description, *GEN* 2–55
quot program
4.2BSD improvement, *SYS* 1–20
Quota
exceeding, *GEN* 3–22
Quota file
comparing with allocated disk space, *SYS* 2–4
description, *SYS* 2–5
Quota system
See Disk quota system
quota system call
4.2BSD improvement, *SYS* 1–12
quota.h file
4.2BSD improvement, *SYS* 5–5
quota_kern.c file
contents, *SYS* 5–9
quota_subr.c file
contents, *SYS* 5–9
quota_sys.c file
contents, *SYS* 5–9
quota_ufs.c file
contents, *SYS* 5–9
quotacheck program
4.2BSD improvement, *SYS* 1–20
quotaon program
See also quotaoff
4.2BSD improvement, *SYS* 1–20
Quotation
defined, *GEN* 4–72
setting apart, *GEN* 5–25
Quotation marks (C shell)
using metacharacters in command arguments, *GEN* 4–35
Quotation marks (me)
making compatible for printers and typesetters, *GEN* 5–38
translating for typesetter, *GEN* 5–38
Quotation marks (ms)
translating for typesetter, *GEN* 5–19

Quotation marks (nroff)
specifying font, *GEN* 5-36
Quotation marks (troff)
translating, *GEN* 5-86
Quoted string statement (BC)
forming, *GEN* 2-54

R

r command (ed)
defined, *GEN* 3-34
using, *GEN* 3-27
without line address, *GEN* 3-49
r command (edit)
description, *GEN* 3-22
r command (ex)
description, *GEN* 3-91
r command (me)
defined, *GEN* 5-44
specifying roman font, *GEN* 5-36
R command (ms)
restoring regular font, *GEN* 5-8
r command (sed), *GEN* 3-112E
defined, *GEN* 3-112
R command (vi)
See also r command (vi)
defined, *GEN* 3-79
r command (vi)
See also R command (vi)
defind, *GEN* 3-81
r escape (Mail)
description, *GEN* 2-24
r flag (cp)
file system tree and, *SYS* 1-5
r flag (Mail)
defined, *GEN* 2-36
r flag (make)
defined, *PGM* 3-17
r modifier (C shell)
extracting filename root, *GEN* 4-57E
r option (edit)
recovering files, *GEN* 3-23
r option (nroff/troff)
defined, *GEN* 5-49
r option (uucp)
defined, *SYS* 5-132
r option (uux)
description, *SYS* 5-133
RA60 disk drive
See uda driver
RA80 disk drive
See uda driver
RA81 disk drive
See uda driver
Rand MH system
mail program and, *SYS* 1-7
random library
4.2BSD improvement, *SYS* 1-15
Ratfor language
See also EFL programming language
See also M4 macro processor
C and, *GEN* 2-15
description, *PGM* 2-111 to 2-122
Raw device
description, *SYS* 5-20
raw routine
defined, *PGM* 4-85
Raw socket
See also Datagram socket
defined, *SYS* 3-6
rb command (me)
defined, *GEN* 5-44
RC command (me)
defined, *GEN* 5-46
rc program
4.2BSD improvement, *SYS* 1-20
rcexpr routine
arguments, *PGM* 2-68
rcp program
cp support and, *SYS* 1-8
rd command (nroff/troff)
defined, *GEN* 5-72
rdump program
See also rmt program
4.2BSD improvement, *SYS* 1-18, 1-20
re command (me)
defined, *GEN* 5-45
Read command (ed)
See r command (ed)
read command (edit)
See r command (edit)
read command (ex)
See r command (ex)
read function
description, *PGM* 1-9
Read only mode (ex)
description, *GEN* 3-85
read system call
4.2BSD improvement, *SYS* 1-12
Read-ahead
description, *GEN* 2-4
readlink system call
4.2BSD improvement, *SYS* 1-12

readv system call
4.2BSD improvement, *SYS* 1-12
record option (Mail)
defined, *GEN* 2-35
recover command (edit)
description, *GEN* 3-22
recover command (ex)
description, *GEN* 3-92
recv system call
4.2BSD improvement, *SYS* 1-12
previewing data, *SYS* 3-10
transferring data, *SYS* 3-9E
recvfrom system call
4.2BSD improvement, *SYS* 1-12
receiving data, *SYS* 3-10E
recvmsg system call
See also sendmsg system call
4.2BSD improvement, *SYS* 1-12
Redirection
defined, *GEN* 4-72
redraw option (ex)
description, *GEN* 3-99
refer program
See also Refer system.if ref
output, *GEN* 5-152E
placing a reference in a paper, *GEN* 5-150
Refer system
See also addbib utility
See also Indexing
4.2BSD improvement, *SYS* 1-8
description, *GEN* 5-133 to 5-142
formatting bibliographic citations, *GEN* 2-13
Reference
formatting, *GEN* 5-151
overriding numbering, *GEN* 5-155
private file of, *GEN* 5-155
Reference file
defined, *GEN* 5-151
refresh routine
defined, *PGM* 4-83
Register
changing for text formatting, *GEN* 5-16
used by -ms
reference list, *GEN* 5-11
regtab table
defined, *PGM* 2-68
Regular expression (ex)
defined, *GEN* 3-96
description, *GEN* 3-96 to 3-97
reference list, *GEN* 3-96
rehash command (C shell)
See also path variable
adding commands to directory and, *GEN* 4-40
defined, *GEN* 4-72
required for current path, *GEN* 4-51
Reiser, J.F., & Henry. R.R.
Berkeley VAX/UNIX Assembler Reference Manual, PGM 4-53 to 4-65
Reiser, J.F., & London, T.B.
regenerating system software, *SYS* 5-117 to 5-122
setting up UNIX/32V V1.0, *SYS* 5-107 to 5-115
Relational operator
description, *GEN* 2-53
form, *GEN* 2-47
Relative pathname
See also Absolute pathname
defined, *GEN* 4-72
Reliably delivered message socket (unsupported)
defined, *SYS* 3-6
Remainder
DC and, *GEN* 2-61
remap option (ex)
description, *GEN* 3-99
remote database
See also tip program
4.2BSD improvement, *SYS* 1-17
Remote login program, *SYS* 3-15F
Remote login server program
main loop, *SYS* 3-18F
pseudo terminals and, *SYS* 3-24
Remote system
calling, *SYS* 5-125
rename system call
4.2BSD improvement, *SYS* 1-12
description, *SYS* 1-35
renice program
4.2BSD improvement, *SYS* 1-20
reorder routine
description, *PGM* 2-76 to 2-77
repeat command (C shell)
defined, *GEN* 4-72
repeating a command, *GEN* 4-51
Reply command (Mail)
See also reply command (Mail)
abbreviating, *GEN* 2-20
answering mail, *GEN* 2-19
answering the sender only, *GEN* 2-20

Reply command (Mail) (Cont.)
definition, *GEN* 2-29
reply command (Mail)
See also Reply command (Mail)
description, *GEN* 2-32
report option (ex)
description, *GEN* 3-100
repquota program
4.2BSD improvement, *SYS* 1-20
Request (nroff)
See Command (nroff)
Reserved word
reference list, *GEN* 4-27
reset command
include file and, *SYS* 1-8
resource.h file
4.2BSD improvement, *SYS* 5-5
restart command (lpc)
description, *PGM* 4-103
restor program
See restore program
restore program
See also rrestore
4.2BSD improvement, *SYS* 1-18
restore server program
See also tar program
RETRN operator (C compiler)
defined, *PGM* 2-65
RETURN key
commands and, *GEN* 2-4
description, *GEN* 3-55
moving the cursor in vi, *GEN* 3-57
return statement (BC)
form of, *GEN* 2-46
forming, *GEN* 2-55
rew command (ex)
description, *GEN* 3-92
rewind command (ex)
See rew command (ex)
rexecd server program
4.2BSD improvement, *SYS* 1-20
rhosts file
description, *SYS* 5-49
Ritchie, D.M.
C Programming Language Reference Manual, The, *PGM* 2-5 to 2-35
I/O system, *PGM* 4-67 to 4-73
standard I/O library, *PGM* 1-21 to 1-24
system security, *SYS* 4-3 to 4-5
tour through C compiler, *PGM* 2-63 to 2-77
Ritchie, D.M. (Cont.)
UNIX Assembler Reference Manual, *GEN* 6-53 to 6-64
Ritchie, D.M., & Kernighan, B.W.
M4 macro processor, *PGM* 2-393 to 2-398
programming UNIX, *PGM* 1-3 to 1-24
Ritchie, D.M., & Thompson, K.
implementation of file system and user command interface, *GEN* 1-19 to 1-34
rk.c device driver
4.2BSD improvement, *SYS* 5-12
RKO7 disk
See va driver
rl option (uucico)
defined, *SYS* 5-135
rl.c device driver
4.2BSD improvement, *SYS* 5-12
RL11 controller
See rl.c device driver
RLABEL operator (C compiler)
defined, *PGM* 2-65
rlogin server program
.login file and, *SYS* 1-7
cu program and, *SYS* 1-8
description, *SYS* 1-8
rlogind server program
4.2BSD improvement, *SYS* 1-20
rm command (nroff/troff)
defined, *GEN* 5-64
rm command (shell)
deleting files, *GEN* 2-7
recover command (edit) and, *GEN* 3-22
removing a file, *GEN* 3-48E
rmdir command
4.2BSD improvement, *SYS* 1-8
rmdir system call
4.2BSD improvement, *SYS* 1-12
rmt program
4.2BSD improvement, *SYS* 1-20
rn command (nroff/troff)
defined, *GEN* 5-64
RNAME operator (C compiler)
defined, *PGM* 2-65
ro command (me)
defined, *GEN* 5-44
roffbib program
bibliographic databases and, *SYS* 1-8
rogue game
4.2BSD improvement, *SYS* 1-17

rogue game (Cont.)
command reference list, *GEN* 6-19 to 6-21
displaying top players, *GEN* 6-25
fighting, *GEN* 6-21
objects you can find, *GEN* 6-21
option reference list, *GEN* 6-24
playing, *GEN* 6-17 to 6-25
rooms, *GEN* 6-21
sample screen, *GEN* 6-18F
scoring, *GEN* 6-24
screen layout, *GEN* 6-18 to 6-19
screen symbol reference list, *GEN* 6-19
setting options, *GEN* 6-23
ROGUEOPTS variable
using, *GEN* 6-23
Roman number
setting page number, *GEN* 5-44
specifying for front matter, *GEN* 5-33
Root directory
defined
description, *GEN* 1-21
Root file system
block size, *SYS* 5-40
dump and, *SYS* 5-54
rebuilding, *SYS* 5-32
restoring, *SYS* 5-26
route program
4.2BSD improvement, *SYS* 1-20
description, *SYS* 5-51
routed server program
4.2BSD improvement, *SYS* 1-20
description, *SYS* 5-51
RP command (ms)
specifying cover sheet, *GEN* 5-5
RP06 disk
bad block forwarding support, *SYS* 1-18
rr command (nroff/troff)
defined, *GEN* 5-66
rrestore program
See also rmt program
4.2BSD improvement, *SYS* 1-20
RS command (ms)
specifying indention level, *GEN* 5-7
rs command (nroff/troff)
defined, *GEN* 5-62
RS variable (awk)
defined, *PGM* 3-6
rsh command
See also rshd server program
rsh server program
executing remote commands, *SYS* 1-8
rshd server program
4.2BSD improvement, *SYS* 1-20
rsp.h file
4.2BSD improvement, *SYS* 5-13
rt command (nroff/troff)
See also mk command (nroff/troff); sp command (nroff/troff)
defined, *GEN* 5-60
RUBOUT character
ignoring while sending mail, *GEN* 2-34
RUBOUT key
See DELETE key
Ruling
specifying, *GEN* 5-88
specifying for figure, *GEN* 5-45
specifying in text, *GEN* 5-26
with tab character, *GEN* 5-87E
Ruling (nroff/troff)
outside text margin, *GEN* 5-72
Running foot
See Page footer
Running head
See Page header
Runtime routine (C)
handling network addresses and values, *SYS* 3-15T
ruptime program
See also rwhod server program
displaying status for cluster, *SYS* 1-8
output, *SYS* 3-20E
rwho program
See also rwhod server program
displaying users on clusters, *SYS* 1-8
rwho server program
description, *SYS* 3-20 to 3-22
simplified form, *SYS* 3-21F
rwhod server program
4.2BSD improvement, *SYS* 1-21
rx driver
4.2BSD improvement, *SYS* 1-16
rx.c device driver
4.2BSD improvement, *SYS* 5-12
RX02 floppy disk unit
See rx driver
rx1 flag (me)
setting 12 pitch, *GEN* 5-39

RX211 floppy disk controller
See rx.c device driver
rxformat program
4.2BSD improvement, *SYS* 1-21

S

s command (DC)
affecting register content, *GEN* 2-62
descripton, *GEN* 2-58
destructive, *GEN* 2-63
programming DC, *GEN* 2-62
s command (ed)
ampersand character and, *GEN* 3-34
breaking lines, *GEN* 3-42
changing all occurrences, *GEN* 3-30
changing every occurrence, *GEN* 3-38E
defined, *GEN* 3-34
deleting text, *GEN* 3-30
delimiters, *GEN* 3-30
description, *GEN* 3-37 to 3-38
g command and, *GEN* 3-46E
g command restriction and, *GEN* 3-47
rearranging a line, *GEN* 3-43
undoing the last substitution, *GEN* 3-38
using, *GEN* 3-29
s command (edit)
replacing text, *GEN* 3-11
uppercase letters and, *GEN* 3-19
s command (ex)
See also & command (ex)
description, *GEN* 3-92
S command (vi)
defined, *GEN* 3-79
s command (vi)
defined, *GEN* 3-81
s escape (Mail)
description, *GEN* 2-25
s flag (ln)
creating symbolic links, *SYS* 1-7
s flag (Mail)
defined, *GEN* 2-36
s flag (make)
defined, *PGM* 3-17
s flag (mkey)
ignoring labels, *GEN* 5-147
s macro (me)
defined, *GEN* 5-43
s option (nroff/troff)
defined, *GEN* 5-49
s option (uucico)
defined, *SYS* 5-135
s option (uucp)
defined, *SYS* 5-132
s option (uulog)
defined, *SYS* 5-137
sail game
4.2BSD improvement, *SYS* 1-17
save command (Mail)
See also write command (Mail)
abbreviating, *GEN* 2-32
system mailbox and, *GEN* 2-23
SAVE operator (C compiler)
defined, *PGM* 2-65
savehist variable
saving history across terminal sessions, *SYS* 1-5
savetty routine
defined, *PGM* 4-88
sc command (me)
defined, *GEN* 5-47
Scale
defined, *GEN* 2-45, 2-51
increasing value, *GEN* 2-45E
limits, *GEN* 2-45
printing current value, *GEN* 2-45E
rules for, *GEN* 2-45
Scale factor
defined, *GEN* 2-59
Scale indicator
attaching to numbers for troff, *GEN* 5-92
Scale register
description, *GEN* 2-60
Scaling
BC language and, *GEN* 2-45
scanf function
See also fscanf function
input and, *PGM* 1-4
scanw routine
defined, *PGM* 4-85
SCCS
introduction, *PGM* 3-23 to 3-37
Schmidt, E., & Lesk, M.E.
Lex program generator, *PGM* 3-113 to 3-125
Scratch character
creating a scratch file, *GEN* 4-31
Scratch file
creating, *GEN* 4-31
defined, *GEN* 4-72

Scratch file (Cont.)
Fortran and, *PGM* 2-83
Screen (Screen package)
defined, *PGM* 4-75
updating, *PGM* 4-92E
updating, *PGM* 4-76 to 4-77
Screen (vi)
breaking lines at right margin, *GEN* 3-67
controlling window size, *GEN* 3-65
refreshing, *GEN* 3-64
Screen editor
invoking from Mail, *GEN* 2-24
screen option (Mail)
defined, *GEN* 2-35
Screen package
description, *PGM* 4-75 to 4-98
input functions, *PGM* 4-78
reference list, *PGM* 4-84 to 4-85
miscellaneous functions
reference list, *PGM* 4-85 to 4-88
output functions, *PGM* 4-78
reference list, *PGM* 4-80 to 4-84
prerequisites, *PGM* 4-75
starting, *PGM* 4-77
terminal information and, *PGM* 4-79
Script
See also Script file
script
4.2BSD improvement, *SYS* 1-8
Script file, *GEN* 4-55E
See also Login shell
See also make command (C shell)
break statement and, *GEN* 4-58
commands useful to writers of, *GEN* 4-53
comments in, *GEN* 4-59
creating, *GEN* 2-10, 3-52E
defined, *GEN* 3-51, 4-53, 4-72
interrupts and, *GEN* 4-59
invoking, *GEN* 4-53
making executable, *GEN* 4-53
preventing variable substitution by the shell, *GEN* 4-59
shell input and, *GEN* 4-58
Script.out file
creating, *GEN* 2-11
scroll routine
defined, *PGM* 4-88
Scrolling
versus paging, *GEN* 3-56
scrollok routine
defined, *PGM* 4-87
sdb symbolic debugger
See also dbx symbolic debugger
accessing symbol information, *SYS* 1-5
locating, *SYS* 1-8
support, *SYS* 1-6
search command (edit)
See Context search (edit)
Search path
See PATH variable
Section
editing with vi, *GEN* 3-61
indenting, *GEN* 5-32E
vi definition, *GEN* 3-62
Section head
coordinating numbers with chapter numbers, *GEN* 5-41
entering in text file, *GEN* 5-6
indenting, *GEN* 5-7E
numbering automatically, *GEN* 5-31 to 5-32, 5-40 to 5-41
numbering automatically with a macro, *GEN* 5-75E
specifying beginning number, *GEN* 5-32E
specifying unnumbered, *GEN* 5-32E
text formatting commands for, *GEN* 5-14E
sections option (ex)
description, *GEN* 3-100
Security
dial-up network and, *SYS* 5-125
UNIX and, *SYS* 4-3 to 4-5
uucp system and, *SYS* 5-138
sed stream editor
address types, *GEN* 3-107 to 3-108
command line format, *GEN* 3-105E
defined, *GEN* 2-13, 3-52
description, *GEN* 3-105 to 3-114
ed and, *GEN* 3-105
functions, *GEN* 3-108 to 3-114
operation, *GEN* 3-105 to 3-106
taking commands from a file, *GEN* 3-52E
uses, *GEN* 3-105
seek function
See also lseek
description, *PGM* 1-12

select system call
4.2BSD improvement, *SYS* 1-12
multiplexing I/O requests, *SYS* 3-11E
Semicolon character (ed)
compared with comma, *GEN* 3-45
setting dot, *GEN* 3-45 to 3-46
send system call
4.2BSD improvement, *SYS* 1-12
transferring data, *SYS* 3-9E
sendbug program
See also bugfiler program
submitting 4.2BSD bug reports, *SYS* 1-8
sendmail
installation and operation guide, *SYS* 2-27 to 2-60
Sendmail Installation and Operation Guide, *SYS* 2-27 to 2-60
See also sendmail
sendmail option (Mail)
defined, *GEN* 2-35
sendmail program
See also mailaddr
See also sendmail option
See also syslog server program
4.2BSD improvement, *SYS* 1-4, 1-21
implementing aliases, *GEN* 2-21
sendmsg system call
See also recvmsg system call
4.2BSD improvement, *SYS* 1-12
sendto primitive
sending data, *SYS* 3-10E
sendto system call
4.2BSD improvement, *SYS* 1-12
Sentence
editing with vi, *GEN* 3-61
vi definition, *GEN* 3-61
Sequenced packet socket (unsupported)
defined, *SYS* 3-6
Server process
See also Client process
description, *SYS* 3-17
Service name
represented by the servent structure, *SYS* 3-14
Service process
See also Service server
Service server
See also Xerox Courier protocol
description, *SYS* 3-17
services database
4.2BSD improvement, *SYS* 1-17
set command (C shell)
C shell variables and, *GEN* 4-40E
defined, *GEN* 4-72
set command (ex)
description, *GEN* 3-92
set command (Mail)
See also unset command (Mail)
forms of, *GEN* 2-20
options and, *GEN* 2-32
restriction, *GEN* 2-21
Set terminal options command
See stty command (C shell)
Set-GID bit
description, *SYS* 4-4
security and, *SYS* 4-5
Set-UID bit
description, *SYS* 4-4
security and, *SYS* 4-5
setbuf library routine
See also setbuffer library routine
setbuffer library routine
See also setbuf library routine
4.2BSD improvement, *SYS* 1-14
setenv command (C shell)
See also printenv command (C shell)
defined, *GEN* 4-73
setting variables in environment, *GEN* 4-51E
setgid system call
See setregid system call
Sethi-Ullman algorithm
C compiler and, *PGM* 2-69 to 2-70
setifaddr program
4.2BSD improvement, *SYS* 1-21
setlinebuf library routine
4.2BSD improvement, *SYS* 1-14
setquota system call
4.2BSD improvement, *SYS* 1-12
SETREG operator (C compiler)
defined, *PGM* 2-65
setregid system call
4.2BSD improvement, *SYS* 1-12
setreuid system call
4.2BSD improvement, *SYS* 1-12
setterm routine
defined, *PGM* 4-88
setuid system call
See setreuid system call
SFCON operator (C compiler)
defined, *PGM* 2-66

SG command (ms)
specifying signature line, *GEN* 5-9
sh command (ex)
description, *GEN* 3-92
sh command (me)
See also uh command (me)
defined, *GEN* 5-40
numbering section heads, *GEN* 5-31 to 5-32
SH command (ms)
specifying unnumbered section head, *GEN* 5-6
sh program
See Bourne shell
Shared lock
multiple processes and, *SYS* 1-3
Sharp character
printing, *GEN* 3-39
Sharp character (#)
entering in text, *GEN* 2-4
erasing last character typed, *GEN* 2-4
shell comments and, *GEN* 4-57
Shell
See also C shell
See Bourne shell
defined, *GEN* 4-73
description, *GEN* 1-27 to 1-31
implementing, *GEN* 1-29
shell command (ex)
See sh command (ex)
shell command (Mail)
See also SHELL option
description, *GEN* 2-32
executing Shell command from Mail, *GEN* 2-22
shell option (ex)
description, *GEN* 3-100
SHELL option (Mail)
defined, *GEN* 2-33
setting, *GEN* 2-32
specifying, *GEN* 2-20
Shell procedure
debugging, *GEN* 4-15
defined, *GEN* 4-7
description, *GEN* 4-7 to 4-16
Shell program
definition, *GEN* 2-11
description, *GEN* 2-11 to 2-12
escaping to from Mail, *GEN* 2-25
profile file and, *GEN* 2-12
programming aids, *GEN* 2-14
as programming language, *GEN* 2-14
Shell program (Cont.)
reading a file for commands, *GEN* 2-12
specifying for Mail, *GEN* 2-20
Shell script
See Script file
shiftwidth option (ex)
description, *GEN* 3-100
Shoens, K., & Leres, C.
Mail Reference Manual, *GEN* 2-17 to 2-41
showmatch option (ex)
description, *GEN* 3-100
showmatch option (vi)
lisp and, *GEN* 3-68
shutdown system call
4.2BSD improvement, *SYS* 1-12
data pending and, *SYS* 3-10E
sigblock system call
4.2BSD improvement, *SYS* 1-12
SIGCHLD signal
constructing server processes, *SYS* 3-27
reaping child processes, *SYS* 3-28E
SIGIO signal
4.2BSD improvement, *SYS* 1-13, 5-7
interrupt-drive I/O and, *SYS* 3-27
Signal
defined, *GEN* 4-73
description, *PGM* 1-17 to 1-20
handling methods, *GEN* 4-22
Signal facilities
4.2BSD improvement, *SYS* 1-3
signal function
descripton, *PGM* 1-17 to 1-20
signal.h file
4.2BSD improvement, *SYS* 5-7
signals and, *PGM* 1-17
Signataure line
specifying, *GEN* 5-9
sigpause system call
4.2BSD improvement, *SYS* 1-12
SIGPROF signal
4.2BSD improvement, *SYS* 1-13, 5-7
sigsetmask system call
4.2BSD improvement, *SYS* 1-12
sigstack system call
4.2BSD improvement, *SYS* 1-12
sigsys system call
See signal facilities

SIGTINT signal
See SIGIO signal
SIGURG signal
4.2BSD improvement, *SYS* 1-13, 5-7
out of band data and, *SYS* 3-27
sigvec system call
4.2BSD improvement, *SYS* 1-13
SIGVTALRM signal
4.2BSD improvement, *SYS* 1-13, 5-7
sinclude command (M4)
description, *PGM* 2-396
SINCR parameter
description, *SYS* 5-121
Singlespacing
specifying, *GEN* 5-23
size keyword (EQN)
changing point size, *GEN* 5-100
sk command (me)
defined, *GEN* 5-44
Sklower, K.L., & others
Franz Lisp Manual, The, *PGM* 2-211 to 2-358
Slash
See Backslash
Slow terminal
editing on, *GEN* 3-64
vi and, *GEN* 3-74
slowopen option (ex)
description, *GEN* 3-100
SM command (ms)
decreasing type size, *GEN* 5-8
SMAPSIZ parameter
description, *SYS* 5-122
SMTP
See DARPA Simple Mail Transfer Protocol
SNAME operator (C compiler)
defined, *PGM* 2-65
so command (ex)
See so command (ex)
description, *GEN* 3-92
so command (nroff/troff)
defined, *GEN* 5-72
interpolating file name, *GEN* 5-81
SO_DEBUG option
network and, *SYS* 5-57
Socket
binding, *SYS* 3-7
creating, *SYS* 3-7
description, *SYS* 3-6 to 3-11
discarding, *SYS* 3-10, 3-10E
naming, *SYS* 3-6
Socket (Cont.)
optimal size, *SYS* 1-28
process group and, *SYS* 3-23
types of, *SYS* 3-6
Socket name
binding to UNIX domain socket, *SYS* 3-8E
description, *SYS* 3-7
Socket system call
creating a socket, *SYS* 3-7E
socket system call
4.2BSD improvement, *SYS* 1-13
failure, *SYS* 3-7
socket.h file
4.2BSD improvement, *SYS* 5-5
socketpair system call
4.2BSD improvement, *SYS* 1-13
socketvar.h file
4.2BSD improvement, *SYS* 5-5
Soft limit
defined, *SYS* 2-3
Software maintenance
using network for, *SYS* 5-127
SOH
See Leader character (nroff/troff)
sort program
defined, *GEN* 2-13, 4-73
specifying numeric sort, *GEN* 4-32E
sortbib command
sorting bibliographic databases and, *SYS* 1-9
Source Code Control System
See SCCS
source command
description, *GEN* 2-32
source command (C shell)
defined, *GEN* 4-73
effecting changes to .chshrc immediately, *GEN* 4-51
Source file
locating
reference list, *SYS* 5-117
Source management system
defined, *PGM* 3-23
sp command (me)
See also bl command (me)
entering, *GEN* 5-23
sp command (nroff/troff)
defined, *GEN* 5-62
setting, *GEN* 5-84
Space character
edit and, *GEN* 3-7

Special character
See Metacharacters
searching, *GEN* 3-21
Spell
defined, *GEN* 2-13
detecting spelling errors, *GEN* 2-13
sprintf function
See also fprintf function
description, *PGM* 1-8
sprintf function (awk)
defined, *PGM* 3-8
sptab table
defined, *PGM* 2-68
SQFILE
description, *SYS* 5-142
sqrt function (awk)
defined, *PGM* 3-8
sqrt keyword, *GEN* 2-44E
defined, *GEN* 2-51
sqrt operator (EQN)
creating square roots, *GEN* 5-100
Square root
creating with EQN, *GEN* 5-100
DC and, *GEN* 2-61
Square root (BC), *GEN* 2-44
ss command (troff)
defined, *GEN* 5-58
sscanf function
description, *PGM* 1-8
SSIZE parameter
description, *SYS* 5-121
SSPACE operator (C compiler)
defined, *PGM* 2-64
Stack command (DC)
description, *GEN* 2-62
Standalone I/O library
4.2BSD improvement, *SYS* 5-15
Standard error output file
description, *PGM* 1-6
Standard I/O library
call formats, *PGM* 1-21 to 1-24
defined, *PGM* 1-5
description, *PGM* 1-5 to 1-8, 1-21 to 1-24
Standard input
See Input
typing form letters or text with nroff/troff, *GEN* 5-72
Standard input file
description, *PGM* 1-6
Standard output
See Output
Standard output file
description, *PGM* 1-6
standout routine
defined, *PGM* 4-84
Star
See Asterisk character
start command (lpc)
description, *PGM* 4-103
Startup file
running, *GEN* 2-12
stat system call
4.2BSD improvement, *SYS* 1-13
stat.h file
4.2BSD improvement, *SYS* 5-7
Statement (as)
description, *GEN* 6-55 to 6-56
Statement (BC)
See also specific statements
description, *GEN* 2-54 to 2-55
typing several on one line, *GEN* 2-48
Status
defined, *GEN* 4-73
status command (mt)
showing state of tape drive, *SYS* 1-7
stderr file pointer
description, *PGM* 1-6
error handling and, *PGM* 1-7
stdin file pointer
description, *PGM* 1-6
stdio library
4.2BSD improvement, *SYS* 1-14
stdout file pointer
description, *PGM* 1-6
stop command (C shell)
background jobs and, *GEN* 4-46E
defined, *GEN* 4-73
stop command (ex)
Berkeley TTY driver and, *GEN* 3-102
description, *GEN* 3-93
stop command (lpc)
description, *PGM* 4-103
Stopped message
suspending jobs and, *GEN* 4-46
Storage class
description, *GEN* 2-53
store command (DC)
See s command (DC)
Stream socket
See also Datagram socket
creating in Internet domain, *SYS* 3-7E

Stream socket (Cont.)
defined, *SYS* 3-6
String (C shell)
defined, *GEN* 4-73
String (nroff/troff)
defined, *GEN* 5-62
description, *GEN* 5-62 to 5-65
String statement (as)
defined, *GEN* 6-56
strip
4.2BSD improvement, *SYS* 1-9
STST file
description, *SYS* 5-143
stterm routine
variables set by, *PGM* 4-89T to 4-90T
stty command
DEC standard values and, *SYS* 1-9
stty command (C shell)
background jobs and, *GEN* 4-48
defined, *GEN* 4-73
Style program
See also Diction program
description, *GEN* 5-163 to 5-177
su
4.2BSD improvement and, *SYS* 1-9
sub keyword (EQN)
specifying subscripts, *GEN* 5-99
subr_mcount.c file
contents, *SYS* 5-9
subr_prf.c file
contents, *SYS* 5-9
subr_rmap.c file
contents, *SYS* 5-9
subr_xxx.c file
contents, *SYS* 5-9
Subscript
specifying, *GEN* 5-47
Subscript (EQN)
specifying, *GEN* 5-99
Subscript (nroff/troff)
specifying, *GEN* 5-68
Subscript (troff)
specifying, *GEN* 5-87E
Subscripted variable
defined, *GEN* 2-46 to 2-47
Substitute command
See s command
substitute command (edit)
See s command (edit)
substitute command (ex)
See s command (ex)
substitute command (sed), *GEN* 3-111E
description, *GEN* 3-110 to 3-111
special characters and, *GEN* 3-110
Substitution
See also Expansion
defined, *GEN* 4-73
substr command (M4)
description, *PGM* 2-397
substr function (awk)
defined, *PGM* 3-8
Subtraction
DC and, *GEN* 2-60
subwin routine
defined, *PGM* 4-87
Suffix list (make), *PGM* 3-17
description, *PGM* 3-21
Summary information
contents, *SYS* 2-8
sup keyword (EQN)
specifying superscripts, *GEN* 5-99
Super user
security and, *SYS* 4-4
Super-block
description, *SYS* 2-8
Superscript
specifying, *GEN* 5-47
Superscript (EQN)
specifying, *GEN* 5-99
Superscript (nroff/troff)
specifying, *GEN* 5-68
Superscript (troff)
specifying, *GEN* 5-87E
Suspended job
defined, *GEN* 4-73
description, *GEN* 4-36
sv command (me)
specifying blank lines, *GEN* 5-44
sv command (nroff/troff)
defined, *GEN* 5-62
Swap space configuration
4.2BSD improvement, *SYS* 1-4
swapgeneric.c file
4.2BSD improvement, *SYS* 5-14
swapon system call
4.2BSD improvement, *SYS* 1-13
SWIT operator (C compiler)
defined, *PGM* 2-65
switch command (C shell)
defined, *GEN* 4-73
exiting from, *GEN* 4-58
forms of, *GEN* 4-58

sx command (me)
defined, *GEN* 5-41
Symbolic link
description, *SYS* 1-3, 1-34
Symbolic link data block
defined, *SYS* 2-12
SYMDEF operator (C compiler)
defined, *PGM* 2-64
symlink system call
4.2BSD improvement, *SYS* 1-13
Symmetric protocol
defined, *SYS* 3-17
sys directory
file prefixes, *SYS* 5-8T
sys_errno
printing, *PGM* 1-12
sys_generic.c file
contents, *SYS* 5-9
sys_inode.c file
contents, *SYS* 5-9
sys_machdep.c file
4.2BSD improvement, *SYS* 5-13
sys_process.c file
contents, *SYS* 5-9
sys_socket.c file
contents, *SYS* 5-9
syscmd command (M4)
description, *PGM* 2-396
sysline program
maintaining terminal status, *SYS* 1-9
syslog server program
4.2BSD improvement, *SYS* 1-21
System function
description, *PGM* 1-12
System identifier
defined, *SYS* 5-74
System mailbox file
commands for folders and, *GEN* 2-23
hold option and, *GEN* 2-32
incoming mail and, *GEN* 2-17
mbox and, *GEN* 2-20
storing mail, *GEN* 2-20, 2-21
System management
best reference, SYS
System process
defined, *PGM* 4-5
System time
4.2BSD improvement, *SYS* 1-4
System-wide file
defined, *GEN* 2-21
Systems Industries 9700 tape drive
See ut.c device driver
systm.h file
See also kernel.h file
4.2BSD improvement, *SYS* 5-7
sz command (me)
changing point size, *GEN* 5-38W
defined, *GEN* 5-44

T

t command (ed)
compared with m command, *GEN* 3-51
creating a series of variable lines, *GEN* 3-51
t command (ex)
See copy command (ex)
t command (sed)
defined, *GEN* 3-114
T command (vi)
defined, *GEN* 3-79
t command (vi)
defined, *GEN* 3-81
t escape (Mail)
description, *GEN* 2-25
T flag (Mail)
defined, *GEN* 2-36
t flag (make)
defined, *PGM* 3-17
T option (hunt)
defined, *GEN* 5-149
t option (hunt)
defined, *GEN* 5-149
T option (nroff)
defined, *GEN* 5-50
t option (troff)
defined, *GEN* 5-50
ta command (nroff/troff)
defined, *GEN* 5-66
Tab
resetting, *GEN* 5-45
setting multiple, *GEN* 5-87
Tab character
printing, *GEN* 3-37
terminals without, *GEN* 2-4
Tab character (nroff/troff)
setting, *GEN* 5-66
uninterpreted, *GEN* 5-66
Tab replacement character
See tc command (troff), *GEN* 5-87
Tab stop
setting, *GEN* 3-61n
vi and, *GEN* 3-61

Table
breaking across pages, *GEN* 5-10
continuing, *GEN* 5-35
entering with -ms, *GEN* 5-8
floating, *GEN* 5-45
formatting, *GEN* 2-13, 5-33
keeping on one page, *GEN* 5-42
text formatting commands for, *GEN* 5-16E
Table of contents
entering, *GEN* 5-28
formatting, *GEN* 5-34F
producing, *GEN* 5-18, 5-18E
specifying multiple, *GEN* 5-29
specifying section titles for, *GEN* 5-41
specifying without leadering, *GEN* 5-29
Tables
formatting, *GEN* 5-115 to 5-131
tabstop option (ex)
description, *GEN* 3-100
Tag
defined, *GEN* 5-145
tag command (ex)
description, *GEN* 3-93
Tag file
defined, *GEN* 5-145
taglength option (ex)
description, *GEN* 3-100
tags option (ex)
3.5 changes, *GEN* 3-103
description, *GEN* 3-100
tail
4.2BSD improvement, *SYS* 1-9
talk program
description, *SYS* 1-9
tar program
4.2BSD improvement, *SYS* 1-9, 1-17
tbl program
description, *GEN* 5-33, 5-115 to 5-131
formatting tables, *GEN* 2-13
tc command (nroff/troff)
defined, *GEN* 5-66
tc command (troff)
replacing tab character, *GEN* 5-87
TCP program
See trpt program
teachgammon program
4.2BSD improvement, *SYS* 1-17
Technical memorandum
text formatting commands for, *GEN* 5-13E
Tektronix 4025 terminal
command character for, *GEN* 3-76
Tektronix 4027 terminal
command character for, *GEN* 3-76
telnet program
ARPA Telnet protocol and, *SYS* 1-9
telnetd server program
.login file and, *SYS* 1-7
4.2BSD improvement, *SYS* 1-21
term option (ex)
description, *GEN* 3-101
Terminal
See also Hardcopy terminal
See also Pseudo terminal
See also Screen (Screen package)
See also Screen package
See also Slow terminal
See also Uppercase terminal
configuring, *SYS* 5-42
programs changing mode of, *GEN* 4-48
replacing with a file, *GEN* 2-10
specifying output type with nroff, *GEN* 5-50
specifying standard output with troff, *GEN* 5-50
specifying type, *GEN* 3-54E
strange behavior, *GEN* 2-4
supported
reference list, *GEN* 2-3
switch settings, *GEN* 2-3
type codes, *GEN* 3-53T
without tabs, *GEN* 2-4
Terminal screen
defined, *PGM* 4-75
Termination
defined, *GEN* 4-73
terse option (ex)
description, *GEN* 3-101
test command
Bourne shell and, *GEN* 4-12
Text editor
See ed editor
defined, *GEN* 3-3, 3-25
See also Edit editor, *GEN* 3-3
Text Formatting
See also nroff/troff text processor
Text input mode (ex)
defined, *GEN* 3-85

Text segment (as)
description, *GEN* 6-54
text statement
defined, *GEN* 6-59
tftpd server program
4.2BSD improvement, *SYS* 1-21
TH command (me)
continuing a table, *GEN* 5-35E
th command (me)
defined, *GEN* 5-45
formatting a thesis, *GEN* 5-33
then command (C shell)
See also else command (C shell)
See also if/endif commands (C shell)
defined, *GEN* 4-73
Thesis
formatting, *GEN* 5-18, 5-33, 5-45
text formatting commands for, *GEN* 5-13E
Thompson, K.
UNIX implementation, *PGM* 4-5 to 4-14
Thompson, K., & Morris, R.
password system, *SYS* 4-7 to 4-12
Thompson, K., & Ritchie, D.M.
implementation of file system and user command interface, *GEN* 1-19 to 1-34
ti command (me)
entering, *GEN* 5-24
ti command (nroff/troff)
defined, *GEN* 5-62
ems and, *GEN* 5-86
Tilde character (C shell)
accessing files from other directories, *GEN* 4-34
Tilde character (me)
See Metacharacters
Tilde escape (Mail)
defined, *GEN* 2-24
description, *GEN* 2-24 to 2-26
lines beginning with, *GEN* 2-26
printing summary of, *GEN* 2-26
reference list, *GEN* 2-40T
time command (C shell)
defined, *GEN* 4-74
timing a command, *GEN* 4-52E
time.h file
4.2BSD improvement, *SYS* 5-7
timeout option (ex)
description, *GEN* 3-102
TIMEZONE parameter
description, *SYS* 5-122
timezone parameter (config)
defined, *SYS* 5-79
tip program
cu program as front end, *SYS* 1-5
description, *SYS* 1-4, 1-9
Title page
formatting informal, *GEN* 5-46
specifying, *GEN* 5-32, 5-45
TL command (ms)
AE command and, *GEN* 5-6
tl command (nroff/troff)
defined, *GEN* 5-70
tl command (troff)
printing page numbers, *GEN* 5-91E
tm command (nroff/troff)
defined, *GEN* 5-73
TM file
description, *SYS* 5-142
TM macro
description, *GEN* 5-18
tm.c device driver
4.2BSD improvement, *SYS* 5-12
to keyword (EQN), *GEN* 5-100E
Token
defined, *GEN* 2-50
top command (Mail)
See also toplines option
abbreviating, *GEN* 2-32
description, *GEN* 2-32
toplines option (Mail)
defined, *GEN* 2-35
setting, *GEN* 2-32E
topq command (lpc)
description, *PGM* 4-103
touchwin routine
defined, *PGM* 4-87
Toy, M.C., & Arnold, K.C.R.C.
guide to the dungeons of doom, *GEN* 6-17 to 6-25
tp command (me)
defined, *GEN* 5-45
specifying a title page, *GEN* 5-32
specifying title page, *GEN* 5-33E
tr command (nroff/troff)
defined, *GEN* 2-13, 5-67
using, *GEN* 2-13E
transfer command
See t command (ed)
translit command (M4)
description, *PGM* 2-397
Transparent throughput (nroff/troff)
specifying, *GEN* 5-67

Trap
description, *GEN* 1-31
trap command (Bourne shell)
fault handling, *GEN* 4-21 to 4-23
trap.c file
4.2BSD improvement, *SYS* 5-14
trek game
4.2BSD improvement, *SYS* 1-17
troff text processor
See also EQN program
See also ms macro package
See also nroff text processor
See also nroff/troff text processor
See also tbl program
defined, *GEN* 2-12, 5-83
defining macros, *GEN* 5-89 to 5-90
defining strings, *GEN* 5-88, 5-89
device resolution and, *GEN* 5-56
drawing horizontal and vertical lines of characters, *GEN* 5-88
entering arithmetic expressions, *GEN* 5-92
entering commands, *GEN* 5-83
environments, *GEN* 5-94
formatting a document with -ms, *GEN* 2-12
indenting lines, *GEN* 5-86
invoking, *GEN* 5-49
moving characters up and down, *GEN* 5-87
moving text backwards on a line, *GEN* 5-87
setting point sizes, *GEN* 5-84
setting tabs, *GEN* 5-86
setting vertical spacing, *GEN* 5-84
specifying cut mark, *GEN* 5-74E
specifying fonts, *GEN* 5-85
specifying fonts on the typesetter, *GEN* 5-86
specifying metacharacters, *GEN* 5-86
specifying page heading, *GEN* 5-90
specifying unpaddable characters, *GEN* 5-88
stopping phototypesetter to reload, *GEN* 5-49
tutorial, *GEN* 5-83 to 5-96
trpt program
4.2BSD improvement, *SYS* 1-21
truncate system call
4.2BSD improvement, *SYS* 1-13
TS command (me)
continuing tables, *GEN* 5-35
defined, *GEN* 5-45
formatting tables, *GEN* 5-35
ts driver
4.2BSD improvement, *SYS* 1-16
ts.c device driver
4.2BSD improvement, *SYS* 5-13
tset command (C shell)
defined, *GEN* 4-74
using, *GEN* 4-30E
tstp routine
defined, *PGM* 4-88
tty
See also ttydev.h file
handling, *SYS* 5-6
tty character
See also ttychars.h file
handling, *SYS* 5-5
tty command (C shell)
defined, *GEN* 4-74
tty.c file
4.2BSD improvement, *SYS* 5-9
tty.h file
4.2BSD improvement, *SYS* 5-7
tty_bk.c file
obsolete, *SYS* 5-9
tty_conf.c file
contents, *SYS* 5-9
tty_pty.c file
4.2BSD improvement, *SYS* 5-9
tty_subr.c file
contents, *SYS* 5-9
tty_tb.c file
contents, *SYS* 5-9
tty_tty.c file
contents, *SYS* 5-9
ttychars.h file
4.2BSD improvement, *SYS* 5-5
ttydev.h file
4.2BSD improvement, *SYS* 5-6
tu driver
4.2BSD improvement, *SYS* 1-16
tu.c file
4.2BSD improvement, *SYS* 5-14
TU58 cartridge tape cassette
See uu driver
See uu.c device driver
TU80 tape drive
See ts driver
tunefs program
4.2BSD improvement, *SYS* 1-21

Tuthill, B.
-ms revised version, *GEN* 5-17 to 5-19
using refer, *GEN* 5-133 to 5-142
Twinkle program
description, *PGM* 4-92E
motion optimization and, *PGM* 4-97E
Two-column output
See Column
type command (Mail)
See print command (Mail)
abbreviating, *GEN* 2-18
description, *GEN* 2-32
reading mail and, *GEN* 2-18 to 2-19
Type-number (refer)
reference list, *GEN* 5-152
Typesetting Mathematics - User's Guide, *GEN* 5-105 to 5-114
Typing
correcting mistakes, *GEN* 2-4
Typo
defined, *GEN* 2-13
detecting spelling errors, *GEN* 2-13

U

u command (ed)
using, *GEN* 3-38
u command (edit)
See also At sign
See also CTRL-H
description, *GEN* 3-16
recovering files, *GEN* 3-23
u command (ex)
description, *GEN* 3-93
u command (me)
defined, *GEN* 5-44
u command (troff)
specifying superscripts and subscripts, *GEN* 5-87
U command (vi)
defined, *GEN* 3-79
u command (vi)
defined, *GEN* 3-81
u flag (Mail)
defined, *GEN* 2-36
u option (uulog)
defined, *SYS* 5-137
uba.c device driver
4.2BSD improvement, *SYS* 5-13
uba_ctrl structure
description, *SYS* 5-93
uba_device structure
description, *SYS* 5-94
uba_driver structure
description, *SYS* 5-90
ud_addr routine
description, *SYS* 5-93
ud_attach routine
description, *SYS* 5-92
ud_dgo routine
description, *SYS* 5-93
ud_dinfo routine
description, *SYS* 5-93
ud_dname routine
description, *SYS* 5-93
ud_minfo routine
description, *SYS* 5-93
ud_mname routine
description, *SYS* 5-93
ud_probe routine
description, *SYS* 5-91
ud_slave routine
description, *SYS* 5-91
ud_xclu routine
description, *SYS* 5-93
uda driver
4.2BSD improvement, *SYS* 1-16
uda.c device driver
4.2BSD improvement, *SYS* 5-13
uf command (nroff/troff)
defined, *GEN* 5-67
ufs_alloc.c file
contents, *SYS* 5-9
ufs_bio.c file
contents, *SYS* 5-10
ufs_bmap.c file
contents, *SYS* 5-10
ufs_dsort.c file
contents, *SYS* 5-10
ufs_fio.c file
contents, *SYS* 5-10
ufs_inode.c file
contents, *SYS* 5-10
ufs_machdep.c file
4.2BSD improvement, *SYS* 5-13
ufs_mount.c file
contents, *SYS* 5-10
ufs_nami.c file
contents, *SYS* 5-10
ufs_subr.c file
contents, *SYS* 5-10
ufs_syscalls.c file
contents, *SYS* 5-10

ufs_tables.c file
contents, *SYS* 5-10
ufs_xxx.c file
contents, *SYS* 5-10
uh command (me)
defined, *GEN* 5-41
specifying unnumbered section heads, *GEN* 5-32E
ui_addr routine
description, *SYS* 5-95
ui_alive routine
description, *SYS* 5-95
ui_ctlr routine
description, *SYS* 5-94
ui_dk routine
description, *SYS* 5-95
ui_driver routine
description, *SYS* 5-94
ui_flags routine
description, *SYS* 5-95
ui_hd routine
description, *SYS* 5-95
ui_intr routine
description, *SYS* 5-95
ui_mi routine
description, *SYS* 5-95
ui_physaddr routine
description, *SYS* 5-95
ui_slave routine
description, *SYS* 5-94
ui_type routine
description, *SYS* 5-95
ui_ubanum routine
description, *SYS* 5-94
ui_unit routine
description, *SYS* 5-94
UID
description, *GEN* 1-22, *SYS* 4-4
uio.h file
4.2BSD improvement, *SYS* 5-6
uipc_domain.c file
contents, *SYS* 5-10
uipc_mbuf.c file
contents, *SYS* 5-10
uipc_pipe.c file
contents, *SYS* 5-10
uipc_proto.c file
contents, *SYS* 5-10
uipc_socket.c file
contents, *SYS* 5-10
uipc_socket2.c file
contents, *SYS* 5-10
uipc_syscalls.c file
contents, *SYS* 5-10
uipc_usrreq.c file
contents, *SYS* 5-10
ul command
4.2BSD improvement, *SYS* 1-9
ul command (me)
See also u command (me)
entering, *GEN* 5-25
troff and, *GEN* 5-36
UL command (ms)
underlining a word, *GEN* 5-8
ul command (nroff/troff)
defined, *GEN* 5-67
ul command (troff)
specifying italic lines, *GEN* 5-86
ULTRIX-32
See also UNIX
ULTRIX-32 Operating System
getting started, *GEN* 2-1 to 2-64
um_cmd routine
description, *SYS* 5-94
um_ctrl routine
description, *SYS* 5-94
um_driver routine
description, *SYS* 5-94
um_hd routine
description, *SYS* 5-94
um_intr routine
description, *SYS* 5-94
um_tab routine
description, *SYS* 5-94
um_ubinfo routine
description, *SYS* 5-94
Umlat
See Metacharacters
un network interface driver
4.2BSD improvement, *SYS* 1-16
un.h file
4.2BSD improvement, *SYS* 5-6
una command (ex)
See also abcommand (ex)
description, *GEN* 3-93
unabbreviate command (ex)
See una command (ex)
unalias command (C shell)
See also alias command (C shell)
defined, *GEN* 4-74
Unary operator
defined, *GEN* 2-52
Unary operator (C compiler)
description, *PGM* 2-66
unctrl routine
defined, *PGM* 4-87
undelete command (Mail)
See also delete command (Mail)

undelete command (Mail) (Cont.)
abbreviating, *GEN* 2–33
description, *GEN* 2–33
Underlining
See also Italic
nroff and, *GEN* 5–66
on the typesetter, *GEN* 5–8
specifying, *GEN* 5–8, 5–25
technique for, *GEN* 3–42
Undo command
See u command
undo command (edit)
See u command (edit)
undo command (ex)
See u command (ex)
Ungermann-Bass network interface unit
See un network interface driver
ungetc function
description, *PGM* 1–8
UNIBUS
device naming, SYS 5–20
UNIBUS device driver
support routines, *SYS* 5–95
univec.c file
installing device driver and, *SYS* 5–119
UNIX Assembler Reference Manual, GEN 6–53 to 6–64
See also as assembler
UNIX Operating System
See also 4.2BSD
See also ULTRIX-32
See also VAX UNIX system
bootstrapping and 4.2BSD, *SYS* 5–15
building process, *SYS* 5–76 to 5–78
building with config, *SYS* 5–73 to 5–105
changes in 4.2BSD, *SYS* 1–3 to 1–21
computer-aided instruction for, *GEN* 6–3 to 6–16
crashing, *SYS* 4–3
defined, *GEN* 3–3
design considerations, *GEN* 1–31
device naming, *SYS* 5–19
distinguishing block and raw devices, *SYS* 5–20
for beginners, *GEN* 2–3 to 2–16
getting started, *GEN* 6–15 to 6–16
hardware environment, *GEN* 1–20
implementation, *PGM* 4–5 to 4–14
UNIX Operating System (Cont.)
introduction, *GEN* 1–19 to 1–20
managing
See SYS
other operating systems and, *PGM* 4–13
programming, *PGM* 1–3 to 1–24
reading list, *GEN* 2–15
software environment, *GEN* 1–20
UNIX Programmer's Manual
accessing on line, *GEN* 2–5
UNIX/32V Operating System
hardware requirements, *GEN* 1–4
highlights, *GEN* 1–3 to 1–18
recreating, *SYS* 5–119
regenerating system software, *SYS* 5–117 to 5–122
setting up V1.0, *SYS* 5–107 to 5–115
tuning, *SYS* 5–121 to 5–122
UNIX/32V Programmer's Manual
online, *GEN* 1–11
unlink function
description, *PGM* 1–11
unlink system call
See mkdir command
unmap command (ex)
See also map command (ex)
description, *GEN* 3–93
unoptim routine (C shell)
See also optim routine (C shell)
description, *PGM* 2–67 to 2–68
Unpaddable space character (nroff/troff)
defined, *GEN* 5–60, 5–88
specifying for digits, *GEN* 5–88
specifying for spaces, *GEN* 5–88
unpcb.h file
4.2BSD improvement, *SYS* 5–6
unset command (C shell)
defined, *GEN* 4–74
unset command (Mail)
See also set command (Mail)
description, *GEN* 2–33
until statement (C shell)
See also while statement (C shell)
description, *GEN* 4–13
up driver
4.2BSD improvement, *SYS* 1–16
up.c device driver
4.2BSD improvement, *SYS* 5–13
Uppercase terminal
vi and

User ID
See UID
User Identification Number
See UID
User identification number
See UID
User process
defined, *PGM* 4–5
user.h file
4.2BSD improvement, *SYS* 5–7
USERFILE
defined, *SYS* 5–140
USR directory
block size, *SYS* 5–40
description, *GEN* 2–9
rebuilding, *SYS* 5–32
setting up, *SYS* 5–28
ut.c device driver
4.2BSD improvement, *SYS* 5–12
utime system call
See utimes system call
utimes system call
4.2BSD improvement, *SYS* 1–13
utmp file
See also wtmp file
4.2BSD improvement, *SYS* 1–17
uu driver
4.2BSD improvement, *SYS* 1–16
uu.c device driver
4.2BSD improvement, *SYS* 5–12
uucico program
defined, *SYS* 5–131
description, *SYS* 5–124, 5–134 to 5–137
functions, *SYS* 5–125
starting, *SYS* 5–125, 5–134
starting with shell file, *SYS* 5–143
uuclean program
defined, *SYS* 5–131
description, *SYS* 5–137
uucp command
command line format, *SYS* 5–131
defined, *SYS* 5–125
description, *SYS* 5–131 to 5–133
transferring files between machines, *SYS* 5–132E
UUCP network
ARPANET and, *GEN* 2–26
uucp program
defined, *SYS* 5–131
uucp system
4.2BSD improvement, *SYS* 1–4, 1–9, 5–45
uucp system (Cont.)
administration, *SYS* 5–142 to 5–144
defined, *SYS* 5–131
directory list, *SYS* 5–45
file list, *SYS* 5–45 to 5–46
implementing, *SYS* 5–131 to 5–144
installing, *SYS* 5–138 to 5–142
login entry and, *SYS* 5–144
security and, *SYS* 5–138
setting up, *SYS* 5–45 to 5–46
uucp.h file
modifying for uucp, *SYS* 5–138
uulog program
defined, *SYS* 5–131
description, *SYS* 5–137
uusnap program
description, *SYS* 1–9
uux command
command line format, *SYS* 5–133
defined, *SYS* 5–125
description, *SYS* 5–133 to 5–134
providing remote output, *SYS* 5–127
uux program
defined, *SYS* 5–131
uuxqt program
defined, *SYS* 5–131
description, *SYS* 5–137

V

v command (DC)
descripton, *GEN* 2–58
v command (ed)
defined, *GEN* 3–34
specifying line numbers, *GEN* 3–47
specifying lines without text patterns, *GEN* 3–46 to 3–47
using, *GEN* 3–33
v command (troff)
creating decorative initial capital, *GEN* 5–87E
moving characters up and down, *GEN* 5–87
specifying vertical motion, *GEN* 5–68
v escape (Mail)
description, *GEN* 2–24
v flag (Mail)
See also verbose option
defined, *GEN* 2–36

v option (inv)
defined, *GEN* 5-148
va driver
4.2BSD improvement, *SYS* 1-16
va.c file
4.2BSD improvement, *SYS* 5-13
Valued option (Mail)
See also Option (Mail)
defined, *GEN* 2-20
Variable (BC)
declaring automatic, *GEN* 2-46
number permitted, *GEN* 2-45
Variable (Bourne shell)
description, *GEN* 4-10 to 4-12
reference list, *GEN* 4-11
Variable (C shell)
accessing components, *GEN* 4-54
checking for assigned value, *GEN* 4-53
defined, *GEN* 4-74
removing definition from shell, *GEN* 4-52
removing from environment, *GEN* 4-52
Variable (Screen package)
reference list, *PGM* 4-77
Variable expansion
See Expansion
See Variable
Variable substitution
description, *GEN* 4-53
VAX UNIX system
accounting, *SYS* 5-56
booting, *SYS* 5-52
booting for single user, *SYS* 5-52
changing from single user to multiuser status, *SYS* 5-52
changing to multiuser from single user status, *SYS* 5-52
checking file system, *SYS* 5-53
file maintenance list, *SYS* 5-57
monitoring system performance, *SYS* 5-54
operating procedures, *SYS* 5-52
regenerating, *SYS* 5-55
resource control, *SYS* 5-56
tracking changes, *SYS* 5-56
VAX-11/750
configuration file, *SYS* 5-85
VAX-11/750 console cassette interface
See tu driver
VAX-11/780
configuration file, *SYS* 5-84
VAX/VMS Operating System
autoconfiguration, *SYS* 5-89 to 5-95
data structure sizing rules, *SYS* 5-103 to 5-105
VAX/VMS system sources
directory list, *SYS* 5-4
ve command (ex)
description, *GEN* 3-94
verbose option (Mail)
See also -v flag
defined, *GEN* 2-35
verbose variable (C shell)
defined, *GEN* 4-74
Version
suppressing for Mail, *GEN* 2-35
version command (ex)
See ve command ex)
Vertical bar (EQN)
typesetting in proper size, *GEN* 5-100E
Vertical spacing
setting with troff, *GEN* 5-84
Vesterman, W., & Cherry, L.L.
style and diction programs, *GEN* 5-163 to 5-177
vfontinfo program
font information and, *SYS* 1-9
vfork system call
future plans, *SYS* 1-13
vgrind
4.2BSD improvement, *SYS* 1-9
vgrindefs file
4.2BSD improvement, *SYS* 1-17
vi command (ex)
See also open option
3.5 changes, *GEN* 3-102
description, *GEN* 3-94
screen editing and, *GEN* 3-85
vi screen editor
4.2BSD improvement, *SYS* 1-9
changing words, *GEN* 3-60
character editing, *GEN* 3-59
character editing, low level, *GEN* 3-61
character functions, *GEN* 3-75T
characters for making corrections in input mode, *GEN* 3-72T
commands for file manipulation, *GEN* 3-71T
deleting lines, *GEN* 3-60
deleting words, *GEN* 3-59
description, *GEN* 3-53 to 3-82

vi screen editor (Cont.)
determining state of file, *GEN* 3–57
editing programs, *GEN* 3–67
ending a session, *GEN* 3–55
ex 3.5 changes and, *GEN* 3–103 to 3–104
ex and, *GEN* 3–73
executing shell command from, *GEN* 3–63
ignoring case, *GEN* 3–72
inserting text, *GEN* 3–58
invoking, *GEN* 3–54E
line editing, *GEN* 3–60
manipulating files, *GEN* 3–70
marking return points, *GEN* 3–64
moving blocks of text, *GEN* 3–62
moving in the file, *GEN* 3–56 to 3–58
moving on the screen, *GEN* 3–57
moving to previous position, *GEN* 3–57
moving within a line, *GEN* 3–57
option list, *GEN* 3–65
presenting lines, *GEN* 3–69
recovering lost files, *GEN* 3–66
recovering lost lines, *GEN* 3–66
reversing your changes, *GEN* 3–60
saving changes automatically, *GEN* 3–63
searching for strings in text, *GEN* 3–56, 3–71
sentences and, *GEN* 3–61
view command (ex)
description, *GEN* 3–102
view command (vi)
reading a file, *GEN* 3–58
vipw program
4.2BSD improvement, *SYS* 1–21
vipw script
See vipw program
visual command (ex)
See vi command (ex)
visual command (Mail)
See also edit command (Mail)
description, *GEN* 2–33
VISUAL option (Mail)
defined, *GEN* 2–33
setting, *GEN* 2–33
specifying an editor, *GEN* 2–24
vlimit system call
See getrlimit system call
vlp program
printing lisp programs, *SYS* 1–9
vm_machdep.c file
4.2BSD improvement, *SYS* 5–13
vm_mem.c file
contents, *SYS* 5–11
vm_mon.c file
contents, *SYS* 5–11
vm_page.c file
4.2BSD improvement, *SYS* 5–11
vm_proc.c file
contents, *SYS* 5–11
vm_pt.c file
contents, *SYS* 5–11
vm_sched.c file
contents, *SYS* 5–11
vm_subr.c file
contents, *SYS* 5–11
vm_sw.c file
contents, *SYS* 5–11
vm_swap.c file
contents, *SYS* 5–11
vm_swp.c file
contents, *SYS* 5–11
vm_text.c file
contents, *SYS* 5–11
vmmac.h file
4.2BSD improvement, *SYS* 5–7
vmparam.h file
4.2BSD improvement, *SYS* 5–7, 5–13
vmstat program
4.2BSD improvement, *SYS* 1–9
monitoring system activity, *SYS* 5–54
vmsystm.h file
4.2BSD improvement, *SYS* 5–7
vpr program
shell scripts and, *SYS* 1–10
vread system call
obsolete, *SYS* 1–13
vs command (nroff/troff)
defined, *GEN* 5–61
setting, *GEN* 5–84
vswapon system call
See swapon system call
vtimes system call
See getrusage system call
vv network interface driver
4.2BSD improvement, *SYS* 1–16
vwidth program
troff width tables and, *SYS* 1–10
vwrite system call
obsolete, *SYS* 1–13

W

w command (ed)
defined, *GEN* 3-34
e command and, *GEN* 3-27
entering text into a file, *GEN* 2-6
saving lines for input, *GEN* 3-50
using, *GEN* 3-26
w command (edit)
description, *GEN* 3-22
u command and, *GEN* 3-16
using, *GEN* 3-8
w command (ex)
See also wq command (ex)
description, *GEN* 3-94
w command (nroff/troff)
description, *GEN* 5-68
w command (sed)
defined, *GEN* 3-111
W command (vi)
defined, *GEN* 3-80
w command (vi)
defined, *GEN* 3-81
w escape (Mail)
description, *GEN* 2-24
w flag (mkey)
specifying a file, *GEN* 5-147
w flag (sed)
defined, *GEN* 3-110
w option (troff)
defined, *GEN* 5-50
wait function
description, *PGM* 1-14
wait system call
See also wait.h file
4.2BSD improvement, *SYS* 1-14
wait.h file
4.2BSD improvement, *SYS* 5-6
wait3 system call
See also wait.h file
4.2BSD improvement, *SYS* 1-14
warn option (ex)
description, *GEN* 3-101
Wasley, D.L.
introduction to f77 I/O library, *PGM* 2-79 to 2-88
wc command (C shell)
4.2 BSD improvements, *SYS* 1-10
defined, *GEN* 2-13, 4-74
printing a list of files and, *GEN* 2-11
WDATA operator (C compiler)
defined, *PGM* 2-64
Weinberger, P.J., & Feldman, S.I.
Fortran 77 compiler, *PGM* 2-89 to 2-109
Weinberger, P.J., & others
awk programming language, *PGM* 3-5 to 3-12
wh command (nroff/troff)
defined, *GEN* 5-65
whereis
4.2BSD improvement, *SYS* 1-10
which
4.2BSD improvement, *SYS* 1-10
while statement (awk)
defined, *PGM* 3-9
while statement (BC), *GEN* 2-47
forming, *GEN* 2-54
writing, *GEN* 2-47
while statement (C shell)
See also until statement (C shell)
defined, *GEN* 4-74
description, *GEN* 4-12 to 4-13
exiting, *GEN* 4-58
form of, *GEN* 4-12E
forms of, *GEN* 4-58
who command
4.2BSD improvement, *SYS* 1-10
printing list of people logged on, *GEN* 2-11E
using, *GEN* 2-4
Width command (nroff/troff)
See w command (nroff/troff)
winch routine
defined, *PGM* 4-86
Window
defined, *PGM* 4-75
description, *PGM* 4-76
moving, *GEN* 2-33
window option (ex)
description, *GEN* 3-101
window option (Mail)
headers command and, *GEN* 2-30
WINDOW structure
defined, *PGM* 4-91E
description, *PGM* 4-76
Word (C shell)
defined, *GEN* 4-74
Word (nroff/troff)
defined, *GEN* 5-60
Word abbreviation
See also Macro (vi)
description, *GEN* 3-69
Word list
specifying for hyphenation, *GEN* 5-69

Work file
defined, *SYS* 5–132
Working directory
changing, *GEN* 4–48
changing background job to foreground job and, *GEN* 4–50
changing with programs, *GEN* 4–50
defined, *GEN* 4–74
description, *GEN* 4–48 to 4–50
wq command (ex)
See also xit command (ex)
description, *GEN* 3–94
wrapmargin option (ex)
3.5 changes, *GEN* 3–102
description, *GEN* 3–101
wrapscan option (ex)
description, *GEN* 3–101
write command (C shell)
defined, *GEN* 4–74
write command (ed)
See w command (ed)
write command (edit)
See w command (edit)
write command (ex)
See w command (ex)
write command (Mail)
See also save command (Mail)
description, *GEN* 2–33
write function
description, *PGM* 1–9
write system call
4.2BSD improvement, *SYS* 1–14
writeany option (ex)
description, *GEN* 3–101
writev system call
4.2BSD improvement, *SYS* 1–14
wtmp file
See also utmp file
4.2BSD improvement, *SYS* 1–17

X

x command (Mail)
exiting Mail, *GEN* 2–22
x command (me)
defined, *GEN* 5–43
entering, *GEN* 5–29
X command (sed)
defined, *GEN* 3–113
X command (vi)
defined, *GEN* 3–80
x command (vi)
defined, *GEN* 3–81
x option (uucico)
defined, *SYS* 5–135
x option (uuclean)
defined, *SYS* 5–138
x option (uucp)
defined, *SYS* 5–132
x option (uux)
description, *SYS* 5–133
Xerox Courier protocol
description, *SYS* 3–17
Xerox experimental Ethernet controller
See en network interface driver
Xerox NS Sequenced Packet protocol
sequenced packet socket and, *SYS* 3–6
Xerox Routing Information Protocol
See routed program
xit command (ex)
See also wq command (ex)
description, *GEN* 3–94
xl command (me)
defined, *GEN* 5–45
xp command (me)
defined, *GEN* 5–43
XP macro
description, *GEN* 5–18
XS macro
description, *GEN* 5–18
xtr script file
running, *SYS* 5–26E

Y

Y command (vi)
defined, *GEN* 3–80
using, *GEN* 3–62
y operator
See also Y command (vi)
moving blocks of text, *GEN* 3–62
ya command (ex)
description, *GEN* 3–95
Yacc
See also Lex program generator
description, *PGM* 3–79 to 3–111
yank command (ex)
See ya command (ex)

Z

z command (DC)
description, *GEN* 2–59

z command (edit)
printing a screen of text, *GEN* 3–12, 3–13E
z command (ex)
description, *GEN* 3–95
z command (Mail)
description, *GEN* 2–33
z command (me)
defined, *GEN* 5–42
entering, *GEN* 5–26
specifying fill mode, *GEN* 5–26
z command (nroff/troff)
creating overstruck characters, *GEN* 5–88
z command (nroff/troff) (Cont.)
description, *GEN* 5–68
z command (vi)
defined, *GEN* 3–81
positioning screen text, *GEN* 3–64
z option (nroff/troff)
defined, *GEN* 5–81
Zero
as legal line number, *GEN* 3–46
ZZ command (vi)
defined, *GEN* 3–80
description, *GEN* 3–55

Notes:

Notes:

Notes:

Notes: