DECUS
PROGRAM LIBRARY

DECUS NO.	8-466H
TITLE	POLY LISP P?S-08-1.1H
AUTHOR	Submitted by: Stanley Rabinowitz
COMPANY	Polytechnic Question Society Brooklyn, New York
DATE	March 1, 1971
SOURCE LANGUAGE	PAL III

ATTENTION

This is a USER program. Other than requiring that it conform to submittal and review standards, no quality control has been imposed upon this program by DECUS.

The DECUS Program Library is a clearing house only; it does not generate or test programs. No warranty, express or implied, is made by the contributor, Digital Equipment Computer Users Society or Digital Equipment Corporation as to the accuracy or functioning of the program or related material, and no responsibility is assumed by these parties in connection therewith.

POLY LIST
P?S-08-1.1H

DECUS Program Library Write-up DECUS NO. 8-466H

POLY LISP is a weak, but useful, LISP compiler. It can run stand-alone on a 4K PDP-8 without EAE, or it can be run off the RL Monitor system, with input files being passed to it from RL source files (when there is no more tape input data, POLY LISP will take the remainder of its input from the teletype).

The source for POLY LISP is in files called LISP1 through LISP9. When these files are listed using the octal radix (for the RL line numbers), the line numbers will correspond to the core locations that these lines will be loaded into during execution.

POLY LISP is a system and can be run from the RL Monitor by the command

$$\text{RUN LISP,file}_1\text{,file}_2\text{,...,file}_n$$

where file_1 through file_n ($n \leq 15$) are the RL source files (if any) which are to be passed to LISP as input. This method is recommended, for although you can type programs directly into POLY LIST, there is no provision to correct typing errors and there is no way to save your programs otherwise.

At the present time, there is no user's manual available for POLY LIST, or any other documentation for that manner. We hope you use and enjoy it, but the P?S and DECUS make no claims about its acceptibility.

ACKNOWLEDGEMENT

Parts of this manual were taken from the pamphlet, "SNOBOL, a String Manipulation Language" written by Griswold, Farber, and Polonsky, of Bell Telephone Laboratories.

POLY SNOBOL was originally written for the PDP-8 by Hank Maurer (a P?S and WCFMPG member), but because of subtle changes introduced by Mario DeNobili of the P?S, Mr. Maurer disclaims all responsibilities connected with the current operation of POLY SNOBOL.

Because the writing of POLY SNOBOL did not occur under the complete supervision of Mr. DeNobili, the P?S cannot guarantee support of this program. In fact, the P?S categorically denies that this program satisfies its high standards of excellency. However, by its very nature, the P?S does agree to accept any and all questions posed concerning this manual, POLY SNOBOL, or anything else. Address all queries to

POLY QUESTION SOCIETY
c/o Mario DeNobili
Box D
Polytechnic Institute of Brooklyn
333 Jay Street
Brooklyn, N.Y. 11201

You may also send for, at the above address, the two-volume treatise, "List of Apars for POLY SNOBOL" which will be available in about six years. Please include $199.98 to cover the cost of shipping.

TABLE OF CONTENTS.

Mrs. M. Hall
Hall
Peter

POLY SNOBOL

(Version 0.5)

January 1970

P?S-WCFMPG(HM)-SNOJOB-Ø1

POLY SNOBOL (Version 0.5)

1. Introduction

The ability to manipulate symbolic rather than numeric data is becoming increasingly important in programming. As symbolic manipulations become more complex, programming in machine-oriented languages becomes increasingly tedious and cumbersome. A number of programming languages have been developed to aid the programmer in such problems. As interest in language translation, program compilation and combinatorial problems has increased, many of these languages have been used for types of problems for which they were never intended. It is clear that more general symbol manipulation languages will materially expand the class of problems that can be programmed with reasonable time effort.

The string oriented symbolic language SNOBOL has been developed with these problems in mind. The choice of the string of symbols as the basic data structure in SNOBOL was made because most symbol manipulation problems of current interest may be naturally described in terms of string manipulations. Unfortunately, no standard notation or accepted system of operations exists for string manipulations. Three basic operations seem essential, however:

(i) creation of strings

(ii) examination of the contents of strings, and

(iii) alteration of strings depending on their contents.

A system for accomplishing these basic operations forms the nucleus for SNOBOL. In constructing the syntax and selecting the notation for SNOBOL, the potential programmer was given careful consideration. Emphasis has been placed on simplicity and intuitiveness while maintaining so far as possible the inherent power of a high-level programming language.

POLY SNOBOL is a subset of SNOBOL version 1, originally developed by Griswold, Farber, and Polonsky, of Bell Telephone Laboratories. It bears only faint resemblance to SNOBOL IV which is currently running on Poly's IBM 360.

2. Basic Concepts

2.1 Strings and String Names

The basic data structure in SNOBOL is a string of symbols. Names are assigned to strings to provide an easy way of referring to particular strings. The name of a string may be any string of numerals and/or letters. Thus the string with name

LINE1

may have the contents*

AROUND, AROUND THE SUN WE GO

The name of a string may be any length. All characters are significant.

2.2 String Formation

The most elementary type of string manipulation is the formation of strings. A string named LINE with the contents given above is formed by the following rule:

LINE = 'AROUND, AROUND THE SUN WE GO'

The pair of quotation marks specifies the literal contents of a string. Any symbols (except quotation marks) can be placed within the quotation marks. Strings can also be formed by concatenation. Thus the rule

LINE = 'AROUND, AROUND' 'THE SUN WE GO'

produces the same result as the preceding example.

Strings which have been named previously can be used to form new strings. For example, the rule

EXAMPLE = LINE

forms a string named EXAMPLE with the same contents as the string named LINE.

Both literals and named strings can be used in formation. The sequence of rules:

---

*This and the next few examples are taken from Archibald MacLeish, "Mother Goose's Garland," Collected Poems, 1917-1952, Houghton Mifflin Co., Boston, Massachusetts. Quoted by permission of the publishers.

```
LINE1 = 'AROUND, AROUND THE SUN WE GO'
LINE2 = 'THE MOON GOES ROUND THE EARTH.'
LINE3 = 'WE DO NOT DIE OF DEATH'
LINE4 = 'WE DIE OF VERTIGO.'
TEXT = LINE1 '/' LINE2 '/' LINE3 '/' LINE4
```

will form a composite string with slashes separating the lines in the conventional manner. Note that the spaces between string names and literals serve as break characters for distinguishing the elements to be concatenated. At least one space is required for separation, but more may be inserted.

In forming a string, the string itself may be used. Hence, after performing the two rules

```
NUMBER = '1'
NUMBER = NUMBER NUMBER '0'
```

the string NUMBER will contain the literals '110'.

The null string is a string of length zero. The statement

```
LINE =
```

sets LINE equal to the null string, i.e., clears the contents of LINE. Subsequently, the statement

```
LINE2 = 'ABC' LINE 'DEF'
```

give LINE2 the value 'ABCDEF'. The contents of each variables is initially the null string.

2.3 Pattern Matching

The process of examining the contents of a string for a given substring is called pattern matching. For example, in order to determine whether the string named LINE1 contains the literals 'ROUND', the following rule would suffice:

```
LINE1 'ROUND'
```

This rule is similar to a formation rule, but without the equal sign. The string LINE1 is scanned from the left for an occurance of the five literals 'ROUND' in succession. A pattern matching rule may succeed or fail. Section 3 describes how this success or failure may be recognized and used. If LINE1 is formed as above, the scan would be successful. The string being scanned is not altered in any way.

The pattern may be specified by concatenation of a number of literals and string names just as the contents of a string to be formed were specified. For example:

```
TEXT  LINE1  '/'  LINE2
```

specifies a scan of the string named TEXT for an occurance of the contents of the string LINE1 immediately followed by the literal '/' and in turn immediately followed by the contents of the string LINE2.

2.4 String Variables

The type of scanning described in the previous section is clearly limited. One might, for example, want to know whether a string contains one substring followed by another, but with the second substring not necessarily immediately after the first. A string variable is introduced to permit this kind of scanning. The rule

```
LINE  'AROUND'  *FILLER  'SUN'
```

is of this kind. Here we wish to know whether LINE contains 'AROUND' followed by 'SUN' with perhaps something between. The symbols *FILLER represent a string variable which takes care of this "something". If LINE is formed as in Section 2.2, this scan would be successful. A string variable may be any string name preceded by an asterisk.

A by-product of successfully matching a pattern containing a string variable is the formation of a new string which has the name given after the asterisk of the string variable. This newly formed string contains a copy of the substring of the scanned string where the string variable fitted, i.e., the "something" previously mentioned. In the example given, a string named FILLER would be formed with the literal contents ', AROUND THE'. This newly formed string is entirely independent of the scanned string.

2.5 Replacement

One final rule permitting alteration of the contents of a string will complete the basic string manipulations. Suppose in the string LINE2 we wished to replace 'EARTH' by 'GLOBE'. The following rule will accomplish this

```
LINE2  'EARTH' = 'GLOBE'
```

This rule scans LINE2 for an occurence of 'EARTH'. If this scan is successful, 'EARTH' is then replaced by 'GLOBE'. Thus LINE2 would become 'THE MOON GOES ROUND THE GLOBE.'. If the scan fails, the string being scanned is not altered

As before, the pattern may be any combination of named strings, literals, and string variables. Only the substring matching the pattern is replaced. As a case of special interest, writing nothing to the right of the equal sign causes the substring found by the scan to be deleted. Thus

```
LINE2  'EARTH' =
```

would delete 'EARTH' from LINE2.

Any string formed as the result of a successful pattern match of a string variable on the left side of the equal sign can be used in the replacement on the right side. Thus

```
LINE1  'AROUND'  *FILLER  'SUN' = FILLER
```

would result in the deletion of 'AROUND' and 'SUN' from LINE1.

2.6 Back Referencing

In the example above the string formed as the result of a string variable in a successful pattern match was used for replacement in the same rule. It is even possible to use strings tentatively matched by string variables in the course of the scan. Thus a pattern may contain a string name which is the same as the name of a string variable used previously in the pattern. For example

```
*X  M  X
```

is a pattern containing such back referencing. Since the scan proceeds from left to right, an attempt to find an occurance of X will only be made after X is tentatively defined by *X. If

```
TEXT = '(C,D)(A,B)(D,C)(A,B)'
```

then the rule

```
TEXT  '(' *X ')' *Y '(' X ')'
```

would succeed, forming a string named X with the contents 'A,B'.

2.7 Other Types of String Variables

The string variable described in Section 2.4 was completely arbitrary in the sense that it could match any substring depending on the particular pattern and string being scanned. However, it is often desirable to restrict the types of substrings a string variable can match. For this purpose, there is another type of string variable.

2.7.1 Fixed-Length String Variables

A fixed-length string variable can only match a substring of specified length. A fixed-length string variable is indicated by appending to the string name a period and the length. The length may be expressed either by a literal integer or the name of a string containing an integer. Thus *PAD.'3' is a fixed-length string variable which can only match a substring of three characters. Similarly *MATCH.N where

N = '15'

can only match a substring of 15 characters.

3. Program Structure

In order to make use of the string manipulation facilities of SNOBOL, the rules are assembled into a program consisting of a number of statements which are executed in a prescribed order.

3.1 Statement Format

A statement in general consists of three parts, separated by blanks, in the following order.

(i) A label, naming the statement,

(ii) A rule, which may be one of the types described in Section 2, and

(iii) A go-to, which may conditionally specify which labeled statement is to be executed next.

3.1.1 Labels

A label may be any permissible string name, and must start at the beginning of the statement. The label on a statement is optional. If a statement has no label, it must begin with a blank. A line beginning with an asterisk is a comment and is not executed.

3.1.2 Rules

Various types of rules were described in Section 2. In all of these types, a rule may be considered to consist of four parts, separated by blanks, in the following order:

(i) A string to be manipulated, called the string reference,

(ii) A left side specifying a pattern,

(iii) An equal sign, and

(iv) A right side specifying a replacement.

The string reference is mandatory. Any of the rest of the rule parts may be absent, depending on the particular rule.

3.1.3 Go-to

The go-to consists of a slash followed by one or more of the following parts:

(i) An unconditional transfer, which has the form (BA), specifying that upon completion of the statement, the next statement to be executed is the statement with label BA.

(ii) A conditional transfer on failure, which has the form F(BB), specifying that if the statement fails, the statement with label BB is to be executed next.

(iii) A conditional transfer on success, which has the form S(BC), similar to failure transfer but with transfer to BC made on success.

Some examples of go-to's are:

```
/(MORGAN)
/F(TIME)
/S(ARBOR)F(RESET)
```

3.2 Program Format and Execution

A program consists of a sequence of statements followed by the statement:

```
END
```

which must start with a blank.

Statements are executed in succession unless a go-to specifies a transfer to some other statement in the program. In all situations where a go-to is not specified, control is transferred to the next statement in the program. The program execution terminates when a transfer to END is made.

As an example, consider the following simple program to remove all occurences of the letters A,E,I,O, and U from a string named TEXT (presumed to be already defined):

```
START   VOWEL = 'A,E,I,O,U,'
V1      VOWEL   *V ',' =                    /F(END)
V2      TEXT    V =                         /S(V2)F(V1)
        END
```

The program executing begins with the statement labeled START, consequently forming a string named VOWEL. The next statement executed is V1 which names the first vowel in VOWEL to be V, and deletes this vowel and the comma following it. This rule will not fall the first time it is executed, hence control is transferred to the subsequent rule V2.

V2 looks in TEXT for the vowel and if successful deletes it, transferring control to V2 once more. This loop continues until all occurences of the vowel have been removed. When V2 finally fails, control is transferred to V1 which selects another vowel from VOWEL and so on. When VOWEL is exhausted, the program is terminated by transferring to END.

4. Arithmetic

POLY SNOBOL allows no arithmetic or arithmetic operators. Numeric literals are not permitted except as the contents of strings. If the user really needs arithmetic in his program, he may simulate it using string manipulation of digits.

5. Indirectness

It is frequently convenient, and for many purposes necessary, to be able to introduce a level of indirectness. This is accomplished in SNOBOL by writing $ in front of the string name. Thus if the string FACTOR contains the literals 'TERM', writing $FACTOR is the same as writing TERM.

An example of the utility of such a feature is the ability of altering the effective go-to of a rule. Suppose I and J are strings containing numbers generated in the program. The rule

```
LABEL = 'B' I  J      /($LABEL)
```

first creates a string with literal contents depending on I and J. Suppose I is '3' and J is '2'. Then LABEL would be 'B32'. The go-to then transfers to the rule labeled B32. Thus indirectness here permits alteration of program flow depending on data (here I and J).

THE INDIRECTNESS OPERATOR MAY NOT BE NESTED.

The indirect feature is useful for specifying the return address of a subroutine. Suppose CAP is the label of the first rule of a subroutine and

```
        /($RET)
```

is the go-to of the last rule executed in CAP. A call to the subroutine which returns to the rule with label A5 is given by the following rule:

```
RET = 'A5'          /(CAP)
```

6. Input - Output

There are two special variables known as SYSPIT and SYSPOT (standing for system peripheral input tape and output tape respectively) which are used for all I/O operations.

All input data to SNOBOL immediately follows the END statement. Whenever the variable SYSPIT is referenced in a SNOBOL statement in a context where its value is needed, the next line of input data is read and used as the new value of SYSPIT. Input data is terminated by a carriage return which is not passed to SNOBOL.

Whenever the value of the variable SYSPOT is changed by a SNOBOL statement, the new value is printed out on the teletype, beginning on a new line. If its value is longer than a line long, SNOBOL will insert appropriate carriage return-line feed sequences.

In all other respects, SYSPIT and SYSPOT act as normal variables. Thus the statement

```
SYSPIT = SYSPOT LINE
```

does not cause either input or output to occur.

EXAMPLE: The following program segment reads words and prints the first five characters. If the input word has fewer than five characters, it prints an error message.

```
        SYSPOT = 'READY'
A       X = SYSPIT
        X = *WORD.'5'      /F(B)
        SYSPOT = WORD      /(A)
B       SYSPOT = 'WORD TOO SMALL'  /(A)
```

7. The Scanning Algorithm

In general, a pattern specified on the left side of a rule consists of a number of elements, i.e., named strings, literals or string variables.

Examples in the preceding sections have described the substrings which each type of elements can match. The way that a specified pattern matches a given string is usually clear. In cases where questions may arise, the following scanning algorithm, which describes the details of the pattern matching process, may be useful.

Rule 1: An attempt is made to match the first pattern element starting at the first symbol of the string. If this match cannot be made, the match is attempted starting at the next symbol of the string, and so on.

Rule 2: The matching process proceeds from left to right, successively matching pattern elements. Each pattern element matches the shortest possible substring.

Rule 3: If at some point an element cannot match a substring, an attempt is made to obtain a new match for the preceding pattern element. This new match is accomplished by extending the substring formerly matched to obtain the next shortest acceptable value. If this extension cannot be made, rule 3 is applied again. If there is no preceding element a newmatch is attempted according to rule 1.

Rule 4: If the last pattern element is an arbitrary string variable (i.e., not fixed-length); its matching substring is extended to the end of the string.

The pattern match succeeds when the last pattern element has been matched. The pattern match fails when the first element cannot be matched.

Note that an arbitrary string variable initially tries to match the null string, thus if A = 'XYZXYAZB' , the pattern match,

```
A  'Y'  *FILLER  'Z'
```

succeeds with *FILLER matching the null string.

8. Modes of Scanning

In addition to the scanning mode previously described (called UNANCHORED mode) there is another mode of scanning called ANCHORED mode. When in anchored mode, a pattern match is successful only if the matching substring begins at the leftmost character of the string being scanned.

SNOBOL initially starts in the unanchored mode.

The statements

ANCHOR

and

UNANCHOR

puts the system in anchored mode or unanchored mode respectively.

9. Data to SNOBOL Program

Data to a SNOBOL program must immediately follow the END statement. If there is no more data stored on tape files, SNOBOL continues to accept input data from the teletype.

10. ECHO Control

Source program and/or data entered through the teletype may be made to echo or not via the two commands

ECHO

and

UNECHO

respectively. The initial mode is determined by the parameter passed to SNOBOL at run time. The parameter 0 initially starts SNOBOL with echoing off. The parameter 1 initially starts SNOBOL with echoing on. No other parameters are significant to SNOBOL.

EXAMPLE:

RUN SNOBOL,FILE1

causes the source file, FILE1 to be passed to SNOBOL but not echoed.

11. Running POLY SNOBOL from the RL Monitor

To run POLY SNOBOL, source programs are normally written and then saved. They can be run by the command

RUN SNOBOL,$file_1$,$file_2$,...,$file_n$

where $file_1$, ... $file_n$ are the source files which will be strung together and passed to SNOBOL. A maximum of 15 files may be so passed (n = 15).

If more source or data is required, SNOBOL will take it from the teletype.

To request SNOBOL to list the source program as it processes it, the command

RUN SNOBOL=1,$file_1$,$file_2$,...,$file_n$

is used instead.

12. Error Messages

The error message,

EVIL

denotes the presence of a syntax error in the source program. If ECHO is on, this will be printed immediately after the faulty statement. Execution begins nevertheless. The following execution time errors may also occur:

(i)	BL	Bad Label.	A transfer was attempted to a string name which was not used as a label in the program.
(ii)	GC	Garbage Collect Error.	SNOBOL ran out of working space for program.
(iii)	ST	Symbol Table Full.	SNOBOL ran out of space in its symbol table. Too many string names were used.

In each case, control is returned to the monitor, if possible.

APPENDIX I

Summary of Syntax of POLY SNOBOL

A vertical bar or vertical stacking denotes alternatives.
Anything enclosed in square brackets, [ ], is optional.
An ellipsis ... denotes optional repetition of the immediately preceding syntactic unit one or more times.
Note that although a SNOBOL program may be syntactically correct, it may not run because of semantic errors.

DIGIT: Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

LETTER: 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

ALPHANUMERIC: LETTER | DIGIT

IDENTIFIER: LETTER [ALPHANUMERIC] ...

BLANKS: one or more spaces

TEXT: any sequence of ASCII characters

STRING-TEXT: any sequence of ASCII characters other than single quotes

VARIABLE: IDENTIFIER

STRING-LITERAL: ' STRING-TEXT '

ELEMENT: VARIABLE | STRING-LITERAL

TERM: ELEMENT | $ ELEMENT

EXPRESSION: TERM [BLANKS TERM] ...

LABEL: ALPHANUM [ALPHANUM] ...

SUBJECT: BLANKS TERM

BASIC-PATTERN: { EXPRESSION | * VARIABLE | * VARIABLE . ELEMENT }

PATTERN: BLANKS BASIC-PATTERN [BLANKS BASIC-PATTERN] ...

OBJECT: EXPRESSION

GOTO: { /(TERM) | /S(TERM) [F(TERM)] | /F(TERM) [S(TERM)] }

COMMENT: * TEXT

COMMAND: BLANKS [UN] { ECHO | ANCHOR }

APPENDIX I (continued)

ASSIGNMENT-STATEMENT: [LABEL] SUBJECT = [OBJECT] [GOTO]
PATTERN-MATCH: [LABEL] SUBJECT PATTERN [GOTO]
REPLACEMENT-STATEMENT: [LABEL] SUBJECT PATTERN = [OBJECT] [GOTO]
END-STATEMENT: bEND
CONTROL-STATEMENT: [LABEL] COMMAND [GOTO]

STATEMENT: { ASSIGNMENT-STATEMENT
PATTERN-MATCH
REPLACEMENT-STATEMENT
COMMENT
CONTROL-STATEMENT
END-STATEMENT }

APPENDIX II

Differences Between POLY SNOBOL and SNOBOL version 1

Differences:

The END statement starts in column 2 in POLY SNOBOL.

String variables also end with a * in SNOBOL version 1.

In SNOBOL version 1, fixed length string variables are specified by using a / instead of a .

Additional features of SNOBOL version 1 (from Bell Labs):

You can specify program starting label in END statement.

The character period may be used in an identifier.

Balanced string variables permitted.

Arithmetic operators allowed.

Additional features of POLY SNOBOL:

(UN)ANCHOR and (UN)ECHO

I/O is similar to the type used by SNOBOL3.