ULTRIX

Guide to the nawk Utility

Order Number: AA-PBKPA-TE
June 1990

Product Version:	nawk Version 1.0
Operating System and Version:	ULTRIX Version 4.0 or higher

This manual is a tutorial description of the `nawk` text-processing utility and programming language.

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 Basic Concepts

2 Simple Arithmetic

3 Patterns and Regular Expressions

4 Actions and Control Structures

5 String Manipulation

6 Arrays

7 User-Defined Functions

8 Enhancing Your nawk Programs

A Order of Operations

B Example Files

Examples

Tables

About This Manual

The *Guide to the nawk Utility* introduces the important principles and concepts of the `nawk` programming language and utility, and shows how they can be used for productive programming. This manual is a tutorial that teaches you how to use `nawk`; it is also a reference manual that you can use later.

Audience

This manual is a guide for intermediate users of the ULTRIX system. If you are a novice user, you might want to read the chapter on regular expressions in *The Big Gray Book: The Next Step with ULTRIX* before using this manual.

Organization

This book contains eight chapters and two appendixes. The following list gives a brief description of the book's contents:

Chapter 1	Introduces `nawk` and describes the basic concepts of the language.
Chapter 2	Describes how to use `nawk` to perform mathematical calculations.
Chapter 3	Describes how to use pattern matching and regular expressions in `nawk` programs.
Chapter 4	Describes the actions you can make `nawk` perform, and discusses how to use control structures to create more powerful `nawk` programs.
Chapter 5	Describes how to manipulate strings with `nawk`.
Chapter 6	Describes how to use arrays of information with `nawk`.
Chapter 7	Describes how to create your own custom functions for `nawk` programs.
Chapter 8	Describes how to tailor your `nawk` programs.
Appendix A	Describes the order in which `nawk` performs operations when executing a program.
Appendix B	Contains copies of the example files used in this manual.

Related Documents

The Little Gray Book: An ULTRIX Primer introduces the ULTRIX operating system and some of the tools and utilities discussed here, and is a handy reference as you read this book.

The Big Gray Book: The Next Step with ULTRIX provides more information on ULTRIX utilities. The *Guide to the nawk Utility* is a thorough tutorial description of an enhanced version of the `awk` utility discussed in *The Big Gray Book.*

Another excellent reference for `nawk` is *The AWK Programming Language*, by Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan (Addison-Wesley, 1988). Aho, Weinberger, and Kernighan created `awk`, of which `nawk` is an enhanced version, at AT&T Laboratories.

The ULTRIX Reference Pages provide details of the commands and utilities described in this book. Experienced programmers may prefer to turn directly to `nawk`(1) in the Reference Pages.

Conventions

The following typeface conventions are used in this manual:

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	In syntax descriptions and function definitions, this typeface is used to indicate terms that you must type exactly as shown.
filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
macro	In text, bold type is used to introduce new terms.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
CTRL/*x*	This symbol is used in examples to indicate that you must hold down the CTRL key while pressing the key *x* that follows the slash. When you use this key combination, the system sometimes echoes the resulting character, using a circumflex (^) to represent the CTRL key (for example, ^C for CTRL/C). Sometimes the sequence is not echoed.

Basic Concepts 1

The nawk language is an easy-to-use programming language that lets you work with information that is stored in files. With nawk programs, you can do these things:

- Display all of the information in a file, or selected pieces of information
- Perform calculations with numeric information from a file
- Prepare reports based on information from a file
- Analyze text for spelling and frequency of words and letters

At first glance, these operations seem elementary. However, later chapters show how they can be combined to perform complicated tasks.

You will find that nawk is a good first programming language. It allows most of the logical constructs of modern computing languages: `if-else` statements, `while` and `for` loops, function calls, and so on. It is easy to learn, and allows beginners to get results with little effort. At the same time, it introduces all the important concepts of programming and prepares users for more complicated languages.

Every programming language has its own way of looking at the world. To write programs in the language, you must learn to see things from the language's point of view.

This chapter examines the fundamentals of nawk:

- The kind of information it works with
- The "shape" of a nawk program
- How to run nawk programs

1.1 Data Files

Almost all nawk programs work with **data**. Programs can obtain data typed in from the terminal or from the output of other commands (through **pipes**); but usually data is obtained from **data files**.

Data files for nawk are always `text` files. This means that the files contain readable text, made up of letters, digits, punctuation characters, and so on. For example, you could create a data file containing information about the hobbies of a group of people. Each line in this file would give a person's name, one of that person's hobbies, how many hours a week the person spends on the hobby, and how much money the hobby costs per year. Using a separate line for each of a person's hobbies, the file might look like this:

```
Jim     reading          15     100.00
Jim     bridge            4      10.00
Jim     role-playing      5      70.00
Linda   bridge           12      30.00
```

```
Linda    cartooning          5      75.00
Katie    jogging            14     120.00
Katie    reading            10      60.00
John     role-playing        8     100.00
John     jogging             8      30.00
Andrew   wind-surfing       20    1000.00
Lori     jogging             5      30.00
Lori     weight-lifting     12     200.00
Lori     bridge              2       0.00
```

If you want to follow the examples using this file, create a copy of the file and name it `hobbies`. There are other example files used in this manual; you might want to create copies of them as well. Appendix B contains copies of all the example files.

1.1.1 Records

A `nawk` data file is a collection of **records**. A record contains a number of pieces of information about a single item; these pieces are called **fields**. In the hobbies file, each line is a separate record, giving a complete set of information about one person's hobby.

Records are separated by a **record separator character**, which is usually the `new-line` character. A new-line character shows where one line of text ends and another begins; in a file using new-line as a record separator, each line of the file is a separate record. All the examples in this manual use the new-line character as a record separator.

1.1.2 Fields

A record consists of a number of **fields**. A field is a single piece of information. For example, the following record from the `hobbies` file contains four fields:

```
Jim      reading          15      100.00
```

The information in the first field is `Jim`, the second is `reading`, and so on.

Specify fields in the same order in each record; that way `nawk` and other tools can easily access a particular piece of information in any record.

The fields of a record are separated by one or more **field separator characters**. In the hobbies file, strings of blank characters (spaces) separate the fields.

By default, `nawk` uses white space (any number of blanks or tab characters) to separate fields. You can change this default, as you will see in Section 1.3.1.

1.2 The Shape of a Program

A `nawk` program looks like this:

```
pattern { actions }
pattern { actions }
pattern { actions }
   .
   .
   .
```

Each line is a separate instruction or `rule`. The `nawk` utility looks through the data files record by record and executes the rules, in the given order, on each record.

1.2.1 Simple Patterns

A rule has this form:

[*pattern*] [{*actions*}]

The form of a rule is called its **syntax**. This syntax indicates that the given set of actions is to be performed on every record that meets a certain set of conditions. The conditions are given by the *pattern* part of the rule. The brackets indicate that both the *pattern* part and the *actions* part are optional.

The *pattern* of a rule often looks for records that have a particular value in some field. The notation `$1` stands for the first field of a record, `$2` stands for the second field, and so on. The special notation `$0` represents the entire record. A pair of equal signs (==) stands for "is equal to." For example:

```
$2 == "jogging" { print }
```

This rule tells `nawk` to print any record whose second field is `jogging`.

This rule is a complete `nawk` program. If you ran this program on the `hobbies` file, `nawk` would look through the file record by record (line by line). Whenever a line had `jogging` as its second field, `nawk` would print the complete record. The output from the program would therefore be as follows:

```
Katie   jogging          14     120.00
John    jogging           8      30.00
Lori    jogging           5      30.00
```

Here is another example; ask yourself what the following `nawk` program does:

```
$1 == "John" { print }
```

As you probably guessed, this program prints every record that has `John` as its first field. The output would be as follows:

```
John    role-playing      8     100.00
John    jogging           8      30.00
```

The same sort of search can be performed on any text database. The only difference is that databases tend to contain a great deal more data than the example contains.

The previous examples both used the `print` action. In fact, this action does not have to be written explicitly; if a `nawk` rule does not contain an action, `print` is assumed. The two example programs you've seen could have been written as follows, with the same effect:

```
$2 == "jogging"
```

and

```
$1 == "John"
```

The use of the two equal signs (==) is an example of a **comparison** operation. The `nawk` language recognizes several other types of comparison:

!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

For example, consider each of the following rules as complete programs, and decide what the programs do with the `hobbies` file:

```
(a)   $1 != "Linda" { print }
(b)   $3 > 10
(c)   $4 < 100.00
(d)   $4 <= 100.00
```

These rules have the following effects:

(a) Prints all records whose first field is not `Linda`.

(b) Prints all records whose third field is greater than 10. Remember that when there is no explicit action, `print` is assumed.

(c) Prints all records whose fourth field is less than 100.00.

(d) Prints all records whose fourth field is less than or equal to 100.00.

1.2.2 Numbers and Strings

In the previous examples, there are quotation marks (") around `Linda` in (a), but none in any of the other rules. The `nawk` language distinguishes between **string values**, which are enclosed in quotation marks, and **numeric values**, which are not.

A string value is a sequence of characters like `="abc"`. Any characters are allowed, even digits, as in `"abc123"`. Strings can contain any number of characters. A string with zero characters is called the **null string** and is written `""`.

A numeric value is mostly made up of digits, but it can also have a sign and a decimal point. The following are all valid numerical values in `nawk`:

```
10    0.34    -78    +2.56    -.92
```

The `nawk` language does not let you put commas inside numbers. For example, you must write 1000 instead of 1,000.

Note

The `nawk` utility lets you use exponential or scientific notation. Exponents are given as `e` or `E` followed by an optionally signed exponent. Thus, the following values are all equivalent:

```
1E3    1.0e3    10E2    1000
```

When numbers are compared (with operators like > and <), comparisons are made in accordance with the usual rules of arithmetic. When strings are compared, comparisons are made in accordance with the ASCII[1] collating order. This is a little like alphabetical order; for example:

```
$1 >= "Katie"
```

This program will print out the `Katie`, `Linda`, and `Lori` lines, as you would expect from alphabetical order. However, ASCII collating order differs from alphabetical order in a number of respects; for example, lowercase letters are greater than uppercase ones, so that `a` is greater than `Z`.

The complete ASCII collating order is given in the `ascii`(7) Reference Page.

[1] ASCII is an abbreviation for American Standard Code for Information Interchange; most computer systems use the ASCII code to represent characters.

1.2.3 The Print Action

So far, the only action you have learned is `print`. As you have seen, `print` can display an entire record. It can also display selected fields of the record, as in the following example:

```
$2 == "bridge" { print $1 }
```

This rule displays the first field of every record whose second field is `bridge`. The output is as follows:

```
Jim
Linda
Lori
```

The `print` command can display more than one field. If you give `print` a list of fields separated by commas, `print` displays the given fields separated by single blanks. For example:

```
$1 == "Jim" { print $2,$3,$4 }
```

This program produces the following output:

```
reading 15 100.00
bridge 4 10.00
role-playing 5 70.00
```

The `print` action can display strings and numbers along with fields. For example:

```
$1 == "John" { print "$",$4 }
```

This program's output looks like this:

```
$ 100.00
$ 30.00
```

In this example, the `print` action prints out a string containing a dollar sign (`$`) followed by a blank, followed by the value of the fourth field in each selected record.

As an exercise, predict the output of the following programs:

```
(a)  $1 == "Lori" { print $1,"spends $", $4,"on",$2 }
(b)  $2 == "jogging" { print $1,"jogs",$3,"hours a week" }
(c)  $4 > 100.00 { print $1, "has an expensive hobby" }
```

1.2.4 Additional Points About Rules

You can put any number of extra blanks and tabs into `nawk` patterns and actions. For example:

```
{ print $1 , $2 , $3 }
```

You can leave out the pattern part of a rule. In this case, the action part is applied to every record in the file. The following example is a complete `nawk` program that displays every record in the data file.

```
{ print }
```

You can leave out the action part of a rule. In this case, the default action is `print`. The following example is a complete `nawk` program that displays every record whose first field is `Andrew`:

```
$1 == "Andrew"
```

This is equivalent to the following:

```
$1=="Andrew" { print }
```

When a nawk program contains several rules, nawk applies every appropriate rule to the first record, then every appropriate rule to the second record, and so on. Rules are applied in order. For example:

```
$1 == "Linda"
$2 == "bridge" { print $1 }
```

This program produces the following output:

```
Jim
Linda    bridge           12      30.00
Linda
Linda    cartooning        5      75.00
Lori
```

The nawk program looks through the file record by record. The following record is the first to satisfy one of the patterns:

```
Jim      bridge            4      10.00
```

As a result, nawk prints out the first field of the record (as dictated by the second rule). The next record of interest is

```
Linda    bridge           12      30.00
```

This record satisfies the pattern of the first rule, so the whole record is printed. It also satisfies the pattern of the second rule, so the first field is printed again. The nawk program continues through the file, record by record, executing the appropriate actions when the pattern is satisfied.

1.3 Running nawk Programs

You can run nawk programs in two ways:

- From a command line
- From a program file

The following sections describe these two methods.

1.3.1 The nawk Command Line

The simplest nawk command line has the following form:

nawk *'program' datafile*

The nawk program is enclosed in apostrophes, or single quotation marks ('). The *datafile* argument gives the name of the data file. For example, the following command executes the program $1 == "Linda" on the hobbies file:

```
% nawk '$1 == "Linda"' hobbies
```

You can also type in a multiline program within apostrophes, provided that the shell you are using allows this construction. For example:

```
nawk '
   $1 == "Linda"
   $2 == "bridge" { print $1 }
   ' hobbies
```

As mentioned in a previous section, the default is for nawk to assume that record fields are separated by space and tab characters. If the data file uses different field

separator characters, you must indicate this on the command line. You do this with an option of the following form:

–F*string*

The *string* lists the characters used to separate fields. For example:

```
nawk -F":" '{ print $3 }'' file.dat
```

This rule indicates that the given data file uses colons (:) to separate fields in its records. The `-F` option must come before the quoted program rules.

1.3.2 Program Files

Short programs like the ones discussed in this chapter can be entered on a single command line. Later chapters discuss longer programs, which cannot be typed on a single line. Such programs are most easily executed from a **program file**.

A program file is a text file that contains a `nawk` program. You can create program files with any text editor. For example, you might create a program file named `lbprog.nawk` that contains the following lines:

```
$1 == "Linda"
$2 == "bridge" { print $1 }
```

To execute a program on a particular data file, use the following command:

`nawk -f` *`progfile datafile`*

The name *progfile* is the name of the file that contains the `nawk` program, and *datafile* is the name of the data file. The following example runs the program in `lbprog.nawk` on the data in `hobbies`:

```
nawk -f lbprog.nawk hobbies
```

If the data file does not use the default separator characters, you must specify a `-F` option after the *progfile* name. For example:

```
nawk -f prog.nawk -F":" file.dat
```

As an exercise, execute the examples in this chapter on the `hobbies` file. Run some from the command line and some from program files.

1.3.3 Sources of Data

If you do not specify a data file on the command line, `nawk` reads data from the terminal. If you issue a command as in the following example, `nawk` prints the first word of every line you type in:

```
nawk '{ print $2 }'
```

When you are entering data from the terminal, mark the end of the data by typing CTRL/D. For example:

```
% nawk '{ print $1 }'
Jim     reading          15     100.00
reading
Jim     bridge            4      10.00
bridge
Jim     role-playing      5      70.00
role-playing
Linda   bridge           12      30.00
bridge
```

```
Linda    cartooning          5       75.00
cartooning
CTRL/D
%
```

You can specify several data files on the `nawk` command line. For example:

```
nawk -f progfile data1 data2 data3 ...
```

When `nawk` finishes reading the first data file, `data1`, it moves to `data2`, and so on.

1.3.4 Saving nawk Output

You can save a `nawk` program's output in a file by using **output redirection**. To do this, specify a right angle bracket (>) and a file name at the end of any `nawk` command line. For example:

```
nawk -f progfile datafile >outfile
```

This command line writes the output from the `nawk` program to a file named `outfile`. In this case, the output is not displayed on the terminal screen. For more information about redirection, see the chapter on the shell in *The Little Gray Book: An ULTRIX Primer*.

Simple Arithmetic 2

The nawk language makes it easy for you to perform calculations with numbers contained in data files. This chapter discusses how nawk does arithmetic and shows examples of programs using these features.

Note that nawk performs arithmetic operations in exactly the same way as the C programming language. Therefore, knowledge of nawk is good preparation for learning C.

2.1 Arithmetic Operations

Here is an example of a nawk program that uses simple arithmetic:

```
$3 > 10 { print $1, $2, $3-10 }
```

In the print statement, $3-10 subtracts 10 from the value of the third field in the record. The print statement prints this result. If you apply this program to the hobbies file shown in the previous chapter, the output will be as follows:

```
Jim reading 5
Linda bridge 2
Katie jogging 4
Andrew wind-surfing 10
Lori weight-lifting 2
```

The program works like this: if someone spends more than 10 hours on a hobby, the program prints the person's name, the name of the hobby, and the number of extra hours the person spends on the hobby (the number of hours more than 10).

The notation $3-10 is called an arithmetic **expression**. It performs an arithmetic operation and comes up with a result; the result of the arithmetic is called the **value** of the expression.

The nawk language recognizes the arithmetic operations shown in Table 2-1.

Table 2-1: Arithmetic Operations

Operation	Operator	Example
Addition	A + B	2+3 is 5
Subtraction	A - B	7–3 is 4
Multiplication	A * B	2*4 is 8
Division	A / B	6/3 is 2
Negation	- A	– 9 is –9

Table 2-1: (continued)

Operation	Operator	Example
Remainder	`A % B`	7%3 is 1
Exponentiation	`A ^ B`	3^2 is 9

The remainder operation is also known as the **modulus** or **integer remainder** operation. The value of a modulus operation is the integer remainder you get when you divide A by B. For example:

```
7 % 3
```

This expression has a value of 1, because when you divide 7 by 3, you get a quotient of 2 and a remainder of 1.

The value for the **exponentiation** operation `A ^ B` is the value of A raised to the exponent B. For example:

```
3 ^ 2
```

This expression has the value 9 (that is, 3×3).

Here are some programs that perform simple arithmetic with the `hobbies` file. Try to figure out what they do and what they will print out.

```
(a)  $1 == "Katie" { print $2, $3/7 }
(b)  { print $1, $2, $3/7 }
(c)  $1 == "Jim" { print $1, $2, "$", $4/52 }
(d)  { print $1, "$", $4*1.05 }
```

After you have thought about the programs, run them to see if they produce the output you have predicted. An explanation of each program follows:

(a) Because field 3 gives the average number of hours per week that a person spends on a hobby, `$3/7` shows the average number of hours per day. Program (a) therefore prints out the number of hours per day Katie spends on each of her hobbies.

(b) This is a variation on program (a). It prints out the number of hours per day each person spends on each hobby.

(c) Field 4 gives the amount of money a person spent this year on a particular hobby. Dividing this by 52 gives the average amount of money spent per week.

(d) If the current inflation rate is 5 percent, multiplying this year's expenses by 1.05 will give the amount of money the same person might expect to spend next year. This is the information that program (d) prints out.

2.1.1 Operation Ordering

Expressions can contain several operations. For example:

```
A+B*C
```

As is customary in mathematics, all multiplications and divisions (and remainder operations) are performed before additions and subtractions. When handling the expression `A+B*C`, `nawk` performs `B*C` first and then adds `A`. The value of `2+3*4` is therefore 14 (3×4 first, then add 2). If you want a particular operation done first, enclose it in parentheses. For example:

```
(A+B)*C
```

When evaluating this expression, nawk performs the addition before the multiplication. Therefore, (2+3)*4 is 20. (Add 2 and 3 first, then multiply by 4.) For example, consider the following program:

```
{ print $4/($3*52) }
```

Field 4 is the amount of money a person spent on a hobby in the last year. Field 3 is the average number of hours a week the person spent on that hobby, so $3*52 is the number of hours in 52 weeks (one year). The value $4/($3*52) is therefore the amount of money that the person spent on the hobby per hour.

Appendix A shows the order of evaluation for nawk expressions.

2.2 Formatted Output

With nawk, you can specify the format you want your output to take. For example:

```
$1 == "Jim" { print "$", $4/52 }
```

This program produces the following output:

```
$ 1.923077
$ .192308
$ 1.346154
```

This output shows the amount of money per week that Jim spent on his hobbies. However, it is customary to write money amounts with only two digits after the decimal point. How can you change the program to make the money amounts look more normal? The answer is to use the printf action instead of print. The printf statement lets you specify the **format** in which output should be printed.

A printf action has the following form:

{ printf *format-string*, *value*, *value*, ... }

The *format-string* indicates the format in which output should be printed. The *values* give the data to be printed.

A format string contains two kinds of items:

- **Normal characters**, which are just printed out as is
- **Placeholders**, which are replaced with values given later in the printf action

As an example, try running the following program on the hobbies file:

```
$2 == "bridge" { printf "%5s plays bridge\n", $1 }
```

This nawk program will produce the following output:

```
  Jim plays bridge
Linda plays bridge
 Lori plays bridge
```

The following format string has one placeholder, %5s:

```
"%5s plays bridge\n"
```

The first (and only) value printed by this program is $1; when the printf statement prints its output, the placeholder is replaced by the value of field 1. The rest of the format string is printed as is. (Note that the format string ends in \n; this symbol is explained in Section 2.2.2.

2.2.1 Placeholders

The form of a placeholder tells nawk how to print out the associated value. All placeholders begin with a percent sign (%) and end in a letter. Table 2-2 shows the most common letters used in placeholders.

Table 2-2: Format String Placeholders

Placeholder	Description
d	An integer in decimal form (base 10)
e	A floating point number in scientific notation, as in -d.dddddd E+dd
f	A floating point number in conventional form, as in -ddd.dddddd
g	A floating point number in either e or f form, whichever is shorter; also, non-significant zeroes are not printed
o	An unsigned integer in octal form (base 8)
s	A string
x	An unsigned integer in hexadecimal form (base 16)

For example, the following format string contains two placeholders:

```
"%s %d\n"
```

The notation %s represents a string and %d represents a decimal integer.

You can put additional information between the percent sign and the letter at the end of the placeholder. If you put an integer there, as in %5s, the number is used as a **width**. The corresponding value is printed using (at least) the given number of characters. For example:

```
$2 == "bridge" { printf "%5s plays bridge\n", $1 }
```

Here, the value of the string $1 replaces the placeholder %5s and is always printed using at least five characters. The output, therefore, is as follows:

```
  Jim plays bridge
Linda plays bridge
 Lori plays bridge
```

If you did not specify the 5 in the placeholder, the output would be different. For example:

```
$2 == "bridge" { printf "%s plays bridge\n", $1 }
```

This program produces the following output:

```
Jim plays bridge
Linda plays bridge
Lori plays bridge
```

If no width is given, nawk prints values using the smallest number of characters possible.

The nawk language also lets you put a minus sign (-) in front of the number in the width position. The amount of output space will be the same, but the information will be left-justified. For example:

```
$2 == "bridge" { printf "%-5s plays bridge\n", $1 }
```

This program's output looks like this:

```
Jim   plays bridge
Linda plays bridge
Lori  plays bridge
```

A placeholder for a floating point number may also contain a **precision**. This is written as a decimal point followed by an integer. A precision determines the number of digits to be printed after the decimal point in a floating point number. For example:

```
$1 == "John" { printf "$%.2f\n", $4/52 }
```

Here, the placeholder `%.2f` indicates that all floating point numbers are to be printed with two digits after the decimal point. This program produces the following output:

```
$1.92 on role-playing
$.58 on jogging
```

Using both a width and a precision can improve the appearance of your program's output. For example:

```
$1 == "John" { printf "$%4.2f on %s\n", $4/52, $2 }
```

This program's output looks like this:

```
$1.92 on role-playing
$0.58 on jogging
```

The `%4.2f` indicates that the corresponding floating point value are to be printed with a width of four characters, with two characters after the decimal point. Note that the decimal point itself is counted in the width.

Here are a few more `nawk` programs that work on the `hobbies` file. Predict what each will print out, and run them to see if your prediction is right.

```
(a)   { printf "%6s %s\n", $1, $2 }
(b)   { printf "%20s: %2d hours/week\n", $2, $3 }
(c)   $1=="Katie" { printf "%20s: $%6.2f\n",$2,$4 }
```

2.2.2 Escape Sequences

All of the format strings shown so far have ended in `\n`. This kind of construct is called an **escape sequence**. All escape sequences are made from a backslash character (\ ) followed by one, two, or three other characters.

You use escape sequences inside strings to represent special characters. In particular, the `\n` escape sequence represents the new-line character. A `\n` in a `printf` format string tells `nawk` to start printing output at the beginning of a new line. For example:

```
$1 == "Lori" { printf " %s", $2 }
```

This program produces the following output:

```
jogging weight-lifting bridge
```

The output is all on one line; without the `\n` escape sequence, `printf` does not start new lines. This action is different from that of `print`, which begins a new line each time it executes.

You can use the `\n` escape sequence in the middle of a format string. For example:

```
$1 == "John" { printf "%s:\n   %d\n",$2,$3 }
```

This program's output looks like this:

```
role-playing:
   8
jogging:
   8
```

The first new-line escape sequence starts a new line after the colon; the second starts a new line after the value of `$3`.

Table 2-3 shows the valid `nawk` escape sequences.

Table 2-3: Escape Sequences for nawk

Escape	Interpretation	Escape	Interpretation
`\"`	Quotation mark	`\n`	New-line
`\a`	Audible bell	`\r`	Carriage return
`\b`	Backspace	`\t`	Horizontal tab
`\f`	Formfeed	`\v`	Vertical tab
`\`*ooo*	ASCII character, octal *ooo*		

Use the escape sequence `\"` (a backslash followed by a quotation mark) when you want a string to contain an actual quotation mark. For example:

```
"He said, \"Hello\"."
```

By entering this escape sequence, you indicate that the quotation mark character is inside the string; if you left out the backslash, `nawk` would think that the quotation mark before `Hello` was marking the end of the string.

Because a backslash followed by another character looks like an escape sequence, you must type two backslashes (`\\`) if you want to put a single backslash character in a string. For example:

```
{ print "The backslash (\\) character" }
```

The output from this program is as follows:

```
The backslash (\) character
```

2.3 Variables

Suppose you want to find out how many people have jogging as a hobby. To do this, you have to look through the `hobbies` file, record by record, and keep a count of the number of records that have `jogging` in their second field. This means you must remember the count from one record to the next.

A `nawk` program remembers information by using **variables**. A variable is a storage place for information. Every variable has a name and a value. A variable is given a value with an action of the following form:

name = *value*

The `nawk` utility **assigns** the specified value to the variable that has the given name. The following example assigns the value 0 (zero) to the variable `count`:

```
count = 0
```

Do not confuse the assignment operator (=) with the equality test operator (==). A

single equal sign (=) stores a value in a variable. A pair of equal signs (==) tests to see if two values are equal.

You can use variables in expressions. For example:

```
count + 1
```

The value of this expression is the current value of `count` plus 1.

Now consider the action in the following example:

```
count = count + 1
```

Your `nawk` program first finds the value of `count + 1` and then assigns this value to `count`. This action increases the value of `count` by 1. You can use this kind of action in a program to count how many people have jogging as a hobby.

```
BEGIN { count = 0 }                                    1
$2 == "jogging" { count = count + 1 }                  2
END { printf "%d people like jogging.\n", count }      3
```

A line by line review of this program follows:

1 When a rule has `BEGIN` as its pattern, the associated action is performed before `nawk` has looked at any of the records in the data file. Therefore, `nawk` begins by assigning the value 0 to `count`.

2 This line adds one to `count` every time `nawk` finds a record with `jogging` in the second field.

3 When a rule has `END` as its pattern, the associated action is performed after `nawk` has looked at all records in the data files specified on the command line. Thus, after `nawk` has looked at all the records, the `printf` action prints out the count of people who jog. The output from the program will be as follows:

```
3 people like jogging.
```

Notice how the value of `count` is printed out in place of the `%d` placeholder.

Here are a few more programs that use variables. Examine the programs and try to figure out what they are doing.

```
(a)  BEGIN  { count = 0 }
     $1 == "John" { count = count + 1 }
     END  { printf "John has %d hobbies.\n", count }

(b)  BEGIN  { sum = 0 }
     $1 == "Linda" { sum = sum + $4 }
     END  { printf "Linda spends $%6.2f a year\n",sum }

(c)  BEGIN  { hours = 0 }
     $1 == "Lori" { hours = hours + $3 }
     END  { printf "Lori passes %d hours/week\n",hours }
```

Here is what each of these programs does:

(a) This program counts the number of hobbies that John has.

(b) This program adds up the amount of money that Linda spent on hobbies in the past year.

(c) This program calculates the number of hours a week that Lori spends on her hobbies.

Using variables, you can write even more complex programs. For example, consider the following:

```
BEGIN  { sum = 0; count = 0 }
```

```
$2 == "role-playing" {
      count = count + 1
      sum = sum + $4
}
END   { printf "Average per person: $%6.2f\n",sum/count }
```

This program has two variables. The `count` variable keeps track of the number of people with role-playing as a hobby, and `sum` keeps track of the amount of money spent on role-playing. When `sum` is divided by `count`, the result is the average amount spent on role-playing.

Notice that the action part of the `BEGIN` rule contains two assignment instructions. A semicolon is used to separate the two instructions. The second rule in the program also has two assignments:

```
count = count + 1
sum = sum + $4
```

These two instructions are on separate lines. When an action contains more than one instruction, you can separate the instructions with semicolons or put them on separate lines.

Variables can be used in the pattern part of a rule. For example:

```
BEGIN   { max = 0 }
$3 > max { max = $3 }
END   { printf "The maximum time is %d hours.\n", max }
```

This program finds the maximum value of field 3 in the `hobbies` file. The maximum is set to 0 to start. Then, if a record has a value in field 3 that is greater than the current value of `max`, `max` is set to this new value. At the end of the data file, `max` will hold the largest value found.

As an exercise, try to write a `nawk` program that examines the `hobbies` file and calculates the average number of hours per week that someone spends on any one hobby. Then write a program that calculates the average number of hours per year that a person spends on any one hobby.

2.3.1 The Increment and Decrement Operators

You know how to advance the value held in a variable with an addition operation:

```
count = count + 1
```

This is such a common operation that `nawk` has a special operator for **incrementing** variables by 1:

```
count++
```

A pair of minus signs (`--`) is the counterpart of `++`. This operator **decrements** (subtracts 1 from) the current value of a variable. For example, to subtract 1 from `count`, you could use either of these two forms:

```
count = count -1
count--
```

2.3.2 Initial Values

If you use any variable in an arithmetic expression before you assign the variable a value, the variable is automatically given the value 0. This means that the `BEGIN` rule in the following program could be left out:

```
BEGIN  { count = 0 }
$2 == "jogging" { count = count + 1 }
END  { printf "%d people jog\n", count }
```

2.3.3 Built-In Record-Oriented Variables

The nawk language has several **built-in** variables that you can use in your programs. You do not have to assign values to these variables; nawk automatically assigns the values for you. Table 2-4 describes some of the important numeric built-in variables. These variables have to do with information about records.

Table 2-4: Built-In Record-Oriented Variables

Variable	Description
NR	Contains the number of records that have been read so far. When nawk is looking at the first record, NR has the value 1; when nawk is looking at the second record, NR has the value 2; and so on. In a BEGIN rule, NR has the value 0. In an END rule, NR contains the total number of records that were read. The following rule prints the total number of data records read by the nawk program: `END  { print NR }`
FNR	Like NR, but counts the number of records that have been read so far from the current file. When several data files are given on the nawk command line, FNR is set back to 1 when nawk begins reading each new file. Thus, the following rule will print the line number in the current file, followed by a colon, followed by the contents of the current line: `{ printf "%d:%s\n",FNR,$0 }`

NF	Gives the number of fields in the current record. For the hobbies file, NF is 4 for each line because there are four fields in each record. In an arbitrary text file, NF gives the number of words on the current line in the file; by default, the fields of a file are assumed to be separated by blanks, so each word on a line is considered to be a separate field. The following program therefore prints out the total number of words in the file:
	`{ count = count + NF }`
	`END  { print count }`

You can use built-in variables in place of any other variable or value. For example, they can appear in the pattern part of a rule. For example:

```
NF > 10 { print }
```

This rule prints out any record that has more than ten fields. Here is another example:

```
NR == 5 { print }
```

This rule prints out record 5 in a file; the pattern selection criterion is true only when NR is 5.

Try to predict what the following example will do:

```
{ print $NF }
```

Because NF is the number of fields in the current record, it is also the number of the last field in the record. Therefore, $NF refers to the *contents* of the last field in a record, and the command in the previous example prints the last field in every record in the data file.

To test your understanding of almost everything discussed in this chapter, try to predict what the following rule will print:

```
(NR % 5) == 0
```

The expression NR% calculates the remainder of NR divided by 5. The rule prints out a record whenever this remainder is equal to 0. Therefore, the rule prints out every fifth record from the data file.

As an exercise, write nawk programs to do the following:

(a) Print every record that does not have exactly three fields.

(b) Print the total number of words and total number of lines in a text file. (This is two thirds of what the wc(1) command does.)

(c) Print the total number of records that have either four fields or five fields.

(d) Print the average number of words per line in a text file.

Write these programs and test them by running them on arbitrary text files. Once you have solutions that work, compare them against the following answers:

```
(a)     NF != 3

(b)         { words = words + NF }
        END { printf "Words = %d, Lines = %d\n",
                  words, NR }

(c)     NF == 4 { count = count + 1 }
        NF == 5 { count = count + 1 }
        END     { print count }

(d)         { words = words + NF }
        END { print "Average = %d\n", words/NR }
```

There are often several ways to write a given program; your solutions may differ from the ones presented here.

2.4 Arithmetic Functions

In nawk, a **function** can be compared to a car assembly line: you feed in various parts and raw materials at one end, and you get out a complete product at the other end. In nawk, a function is fed data values (called the **arguments** of the function) and the final product is also a data value (called the **result** of the function).

You may already be familiar with this kind of function in mathematics. For example, mathematics uses **sin** to stand for a function that calculates the trigonometric sine of an angle. If you "feed" an angle into the **sin** function, the number returned is the trigonometric sine of the given angle. The angle is the argument of the function, and the sine is the result.

In nawk, you use functions inside expressions. For example:

```
y = sin(x)
```

The right hand side of the assignment is a **function call**. The name of the function is sin; this name is immediately followed by the function's arguments, which are

enclosed in parentheses. When a nawk program contains a function call, nawk calculates the result of the function and uses that result in the expression that contains the function call. In the statement y=sin(x), nawk calculates the number that is the sine of the given angle and then assigns that number to the variable y.

Another nawk function is sqrt, whose result is the square root of its argument. The following statement assigns the value 4 to x:

```
x = sqrt(16)
```

To show how you can use these functions, suppose you have a set of data that contains one number per line. Here is a program that reads these numbers and prints out the square root of each:

```
{ printf "Number: %f, Root: %f\n", $1, sqrt($1) }
```

You can run this program with the following command line, and then type in numbers from the terminal:

```
% awk '{ printf "Number: %f, Root: %f\n", $1, sqrt($1) }'
```

Each time you press the RETURN key at the end of the line, nawk prints out the square root of the number.

Any argument of a function can be an expression instead of a single value. For example:

```
y = sin(2*x)
```

Your nawk program will calculate the value of the expression and then use the resulting value as the argument of the function.

The nawk language recognizes the most common mathematical functions, as shown in Table 2-5.

Table 2-5: Common Mathematical Functions

Function	Result	Function	Result
sin(*x*)	Sine of *x*, where *x* is in radians	sqrt(*x*)	Square root of *x*
cos(*x*)	Cosine of *x*, where *x* is in radians	int(*x*)	Integer part of *x*
atan2(*y*,*x*)	Arctangent of *y*/*x* in range $-\pi$ to π radians	rand()	Random number *n*, $0 \le n < 1$
log(*x*)	Natural logarithm (base *e*)	srand(*x*)	Sets *x* as seed for rand()
exp(*x*)	Exponential (e^x)		

Several of these functions need a little more explanation.

The int function takes a floating point number as an argument and returns an integer. The integer is the floating point number without its fractional part. For example:

```
int(6.3)
```

This expression has the value 6. The following expression has the value –7. Note

that the fractional part is removed (truncated), not rounded.

```
int(-7.4)
```

The next expression has the value 8:

```
int(8.99999)
```

A call to `rand` returns a random number greater than or equal to 0 and less than 1. In this way, you can get a sequence of random numbers. You can use `srand` to set the starting point **(seed)** for a random number sequence. If you set the seed to a particular value, you will always get the same sequence of numbers from `rand`. This is useful if you want a program to use `rand` but obtain uniform results every time the program runs.

As an example of how you can use `rand`, here is a sequence of instructions that could be used in a `nawk` program to simulate a roll of two six-sided dice.

```
die1 = int(6 * rand() + 1)
die2 = int(6 * rand() + 1 )
```

The function call `rand()` obtains a random floating point number from 0 to 1 (not including 1). Note that the function call needs the parentheses, even though `rand` requires no argument values. Multiplying the random number by 6 gives a floating point value from 0 to 6 (not including 6). Adding 1 gives a floating point value from 1 to 7 (not including 7). Applying the `int` function to this floating point value drops the fraction part, giving an integer from 1 to 6.

Patterns and Regular Expressions 3

So far, this manual has discussed three kinds of patterns: comparisons, and the special patterns BEGIN and END. This chapter discusses a fourth kind: **regular expressions**.

A regular expression is a way of telling nawk to select records that contain certain strings of characters. For example, the following rule tells nawk to print all records that contain the string ri:

```
/ri/ { print }
```

Applying this rule to the hobbies file produces this output:

```
Jim     bridge           4      10.00
Linda   bridge          12      30.00
Lori    jogging          5      30.00
Lori    weight-lifting  12     200.00
Lori    bridge           2       0.00
```

All these records contain ri, either in Lori or bridge.

Regular expressions are always enclosed in slashes. For example:

```
/ing/
```

This expression finds all the records that contain ing.

The nawk language pays attention to the case of letters in regular expressions. For example,

```
/li/
```

will print the record that contains weight-lifting; however, the /li/ does not match the Linda records because the L in Linda is uppercase.

It is important to recognize the difference between two rules like the following:

```
$1 == "Lori"
/Lori/
```

To satisfy the first of these patterns, a record must have its first field exactly equal to the string Lori. If the first field is Lorie, for example, the comparison will not be true and the pattern will not be satisfied. With the regular expression /Lori/ the string Lori can appear anywhere in the record, and can be all or part of a field. This regular expression would match a string like Lorie.

3.1 Using Matching Expressions

If the pattern in a rule is a regular expression, nawk looks for a matching string anywhere in a record. Sometimes, however, you only want to look for a matching string in a particular field of a record. In this case, you can use a **matching expression**.

Two types of expressions check for matches:

- The following expression is true if the string matches the given regular expression:

  string ~ /regular-expression/

- The following expression is true if the string does not match the given regular expression:

  string !~ /regular-expression/

The statement in the following program looks for matching strings; applied to the `hobbies` file, it will print all records that have `ri` contained somewhere in the second field:

```
$2 ~ /ri/
```

This example produces the following output:

```
Jim     bridge          4       10.00
Linda   bridge         12       30.00
Lori    bridge          2        0.00
```

The following rule looks for nonmatching strings; it will print all records that do not have the letter `J` somewhere in the first field:

```
$1 !~ /J/
```

Note that the following two patterns are equivalent because `$0` represents the whole record:

```
/Lori/
$0 ~ /Lori/
```

3.2 Metacharacters

Several characters have special meanings when they are used in regular expressions. These special characters, known as **metacharacters**, are described in Table 3-1.

Table 3-1: Metacharacters Recognized by nawk

Character	Description
^	Stands for the beginning of a field. For example: `$2 ~ /^b/ { print }` This rule prints any record whose second field begins with `b`.
$	Stands for the end of a field. For example: `$2 ~ /g$/ { print }` This rule prints any record whose second field ends with `g`.
.	Matches any single character (except the new-line). For example: `$2 ~ /i.g/ { print }` This rule selects the records with fields containing `ing`, and also selects the records containing `bridge` (idg).

Table 3-1: (continued)

Character	Description
|	Means "or." For example: `/Linda|Lori/` This regular expression matches either of the strings `Linda` or `Lori`.
*	Indicates zero or more repetitions of a character. For example, `/ab*c/` matches `abc`, `abbc`, `abbbc`, and so on. It also matches `ac` (zero repetitions of `b`). The asterisk is most frequently used in conjunction with the period (`.*`). Because the period matches any character except the new-line, the period/asterisk combination matches an arbitrary string of zero or more characters. For example: `$2 ~ /^r.*g$/ { print }` This rule prints any record whose second field begins with `r`, ends in `g`, and has any set of characters between (for example, `reading` and `role-playing`).
+	Similar to the asterisk, but stands for one or more repetitions of a string. For example, `/ab+c/` matches `abc`, `abbc`, and so on; but it does not match `ac`.
?	Similar to the asterisk, but stands for zero or one repetitions of a string. For example. `/ab?c/` matches `ac` and `abc`, but not `abbc`, and so on.
{*m*,*n*}	Indicates *m* to *n* repetitions of a character (where *m* and *n* are both integers). For example, `/ab{2,4}c/` matches `abbc`, `abbbc`, and `abbbbc`, but nothing else.

[*X*]	Matches any one of the set of characters *X* given inside the brackets. For example:
	`$1 ~ /^[LJ]/ { print }`
	This rule prints any record whose first field begins with either `L` or `J`. As a special case, `[:lower:]` inside brackets stands for any lowercase letter, `[:upper:]` inside brackets stands for any uppercase letter, `[:alpha:]` inside brackets stands for any letter, and `[:digit:]` inside brackets stands for any digit. For example:
	`/[[:digit:][:alpha:]]/`
	This expression matches a digit or letter.

[^*X*]	Matches any one character that is not in the set *X* that follows the circumflex (`^`). For example:
	`$1 ~ /^[^LJ]/ { print }`
	This rule prints any record whose first field does not begin with `L` or `J`.
	`$1 ~ /^[^[:digit:]]/ { print }`
	This rule prints any record whose first field does not begin with a digit.

(*X*)	Matches anything that the regular expression *X* does. Parentheses are used to control the way in which other special characters behave. For example, the asterisk (`*`) normally applies to the single character that immediately precedes it. For example, `/abc*d/` matches `abd`, `abcd`, `abccd`, and so on. However, `/a(bc)*d/` matches `ad`, `abcd`, `abcbcd`, and so on.

When a metacharacter appears in a regular expression, it usually has its special meaning. If you want to use one of these characters literally (without its special meaning), put a backslash in front of the character. For example, the following statement prints all records that contain a dollar sign ($) followed by a 1:

```
/\$1/ { print }
```

If you wrote the expression without the backslash, nawk would search for records in which the end of the record is followed by a 1, which is impossible.

Because the backslash has this special meaning, it too is considered a metacharacter. If you want to create a regular expression that matches a backslash, you must therefore use two backslashes (\\ ).

3.3 Using Matching Expressions with Strings

Until now, you have seen matching operations that contain regular expressions inside slash (/) characters. Matching operations can also refer to normal strings; for example:

```
$1 ~ "xyz"
```

This has the same effect as the following statement:

```
$1 ~ /xyz/
```

Regular expressions are compiled when the program is read. To use a string as a regular expression, nawk constructs a **dynamic regular expression** out of the string. Dynamic regular expressions take more time to compile than regular expressions, but they are more powerful.

When a matching operation uses a string instead of a regular expression, and the string contains one or more metacharacters, the situation is a little bit tricky. If you want to **escape** a metacharacter (have it taken literally), you must use two backslashes instead of one. For example, suppose you want to look for strings of the form "$1.00" in field 4 of a record. Using regular expressions, you would write the statement as follows to show that both the dollar sign ($) and the period (.) should be taken literally:

```
$4 ~ /\$1\.00/
```

With strings, you would have to write the statement like this:

```
$4 ~ "\\$1\\.00"
```

Two backslashes are needed instead of one. The reason is simple: as discussed in Chapter 2, you need to type two backslashes inside a quoted string to get the effect of one. For example:

```
{ print "The backslash character: \\" }
```

This program prints the following:

```
The backslash character: \
```

To match an actual backslash with a dynamic regular expression, you must use four, as in:

```
$1 ~ "\\\\"
```

The literal string "\\\\" is read by nawk and turned into a string consisting of "\\". When used as a dynamic regular expression, this will match one backslash.

3.4 Applying Actions to a Group of Lines

Pattern ranges let you apply an action to a group of lines. A rule that applies to a pattern range has the following form:

pattern1 , *pattern2* { *action* }

This rule performs the given *action* on every line, starting at an occurrence of *pattern1* and ending at the next occurrence of *pattern2* (inclusive). For example:

```
NR == 1, NR == 10 { print $1 }
```

This rule prints the first field of each of the first 10 input lines. It starts when NR is 1 and ends when NR is 10. Here is another example, using the `hobbies` file as its data file:

```
/Jim/, /Linda/ { print $2 }
```

This example produces the following output:

```
reading
bridge
role-playing
bridge
```

As you can see, this program prints the second field of all lines between an occurrence of `Jim` and an occurrence of `Linda`.

After `nawk` has found a record matching *pattern2*, it begins to look for a line matching *pattern1* again. In the following example, `nawk` prints the first range of records from `reading` to `role`, then starts looking for `reading` again.

```
/reading/, /role/
```

The output from this program looks like this:

```
Jim     reading          15     100.00
Jim     bridge            4      10.00
Jim     role-playing      5      70.00
Katie   reading          10      60.00
John    role-playing      8     100.00
```

It is important to remember that `nawk` starts performing the rule's action as soon as there is a record that matches *pattern1*. A `nawk` program does not check to make sure that there is a line matching *pattern2* in the rest of the file. For example:

```
/Lori/, /Jim/ { print $2 }
```

In this case, `nawk` begins printing at the first record that contains `Lori`, and continues until it reaches the end of the file, finding no record that matches the second pattern, `Jim`.

3.5 Combining Conditions in Patterns

A double ampersand (`&&`) operator means **AND**. It is used to combine conditions in patterns. For example:

```
$3 > 10 && $4 > 100.00 { print $1, $2 }
```

In this case, `nawk` prints the first and second fields of any record where `$3` is greater than 10 and `$4` is greater than 100.00. Here is another example:

```
$1 ~ /J/ && $4 < 50.00
```

This rule prints all records in which the first field `$1` contains a `J` and the fourth field `$4` is less than 50.00.

The double vertical bar (`||`) operator means **OR**. It is also used to combine conditions in patterns. For example:

```
$1 == "Linda" || $1 == "Lori"
```

This rule prints any record whose first field is either `Linda` or `Lori`. Here is another example:

```
/jogging/ || /reading/ { sum = sum + $4 }
END  { print sum }
```

This program calculates the total money spent by hobbyists on both jogging and reading (because `sum` is increased if the hobby is either `jogging` or `reading`). This program is equivalent to the following program:

```
/jogging|reading/ { sum = sum + $4 }
END  { print sum }
```

These last two examples demonstrate that there are often several ways of writing the same program.

The double ampersand and double vertical bar operators can only be used to combine complete pattern expressions. For example, you cannot write a pattern like this:

```
$1 == "Linda" || "Lori"
```

You must write this kind of pattern this way:

```
$1 == "Linda" || $1 == "Lori"
```

For practice with the concepts discussed in this chapter, write programs that do the following:

(a) Print every record that begins with `A` and contains more than four fields.

(b) Print the number of records that contain a dollar sign (`$`).

(c) Print records 10 through 20 of every data file.

(d) Print every tenth record of a file, plus the record that immediately follows the tenth record (records 10 and 11, records 20 and 21, and so on).

When you have written your programs, compare them against the solutions that follow. Remember that there may be several ways to write the same program.

(a)
```
/^A/ && NF > 4
```

(b)
```
/\$/ { count = count + 1 }
END   { print count }
```

(c)
```
FNR == 10, FNR == 20
```

(d)
```
(NR % 10) == 0, (NR % 10) == 1
```
or
```
((NR % 10) == 0) || ((NR % 10) == 1)
```

Actions and Control Structures 4

So far, you have learned three actions: `print`, `printf`, and assignments. In this chapter, you will examine a wide variety of constructs that may appear in the action part of a `nawk` rule. Note that most of these are virtually identical to constructs in the C programming language.

4.1 Adding Comments

A comment is a note inside your program, explaining what the program is doing. Your `nawk` program ignores comments, so they do not affect how your program behaves, but they do help explain what is going on.

A comment begins with a number sign (#). When `nawk` sees the number sign in a program (outside of a quoted string or regular expression), it ignores the rest of the line. For example:

```
# This program adds up the hours John spends on hobbies
/John/ { sum = sum + $3 }     # field 3 is hours
END    { print sum }
```

The first line of this program explains what the program is doing. This is useful when you have a number of `nawk` programs stored in different files and you cannot remember which program is which. A comment at the beginning of the program lets you identify the program without having to read through the code and figure out what is going on.

The following example shows another way in which you can use comments:

```
/John/ { sum = sum + $3 }     # field 3 is hours
```

A comment on the end of a line can give further information about what that line is doing. In this case, it explains the meaning of the number in field 3 of the record.

It is a good practice to use comments in your programs. Without meaningful comments, you may find it difficult to understand a program if you look at it several months after you wrote it. Comments also make it easier for others to understand the programs you write.

4.2 The if Statement

An `if` statement lets you perform an action if a specified condition is true. The statement has the following form:

if (*expression*) *statement1* else *statement2*

Typically, the *expression* in an `if` statement has a true/false value. If the value is true, *statement1* is performed; otherwise, *statement2* is performed. The `else` *statement2* part is optional.

To see how `if` statements are used, consider the following programs, which examine a file of baseball scores. This file is named `baseball`, and it looks like this:

```
Brewers         5       Tigers          9
Brewers         2       Blue Jays       6
Blue Jays       8       Red Sox         7
  .
  .
  .
```

Each line gives the home team first and the visitors second. Fields in each record are separated by tab characters (shown here as wide spaces) instead of single blanks, because some team names contain blanks. This means that you must use the following option when you run command-line `nawk` programs on the baseball file:

```
-F"\t"
```

This option is equivalent to having the following line in a `nawk` program file:

```
BEGIN   { FS = "\t" }
```

(The built-in `FS` variable is explained in Chapter 5.)

Consider the following program:

```
{ if ($2 > $4) print "Home"
  else         print "Visitor" }
```

This program prints `Home` when the home team's score (`$2`) is greater than the visiting team's, and prints `Visitor` otherwise.

The `else` part of an `if` statement can be omitted. In this case, `nawk` does nothing if the *expression* of the `if` statement is not true. For example:

```
$1 ~ /Tigers/  { if ($2 > $4) win++ }
END  { print win }
```

This is a simple program that looks at all the Tigers' home games and prints out the number of times the Tigers won. On records where `$2` is not greater than `$4`, `nawk` takes no action.

As a more complicated example, consider this program:

```
$1 ~ /Yankees/ { if ($2 > $4) print "Home Win"
                   else         print "Home Loss" }
$3 ~ /Yankees/ { if ($4 > $2) print "Away Win"
                   else         print "Away Loss" }
```

This program runs through the baseball scores looking for games involving the Yankees. Appropriate messages are written for each possible outcome.

This next program is similar to the previous program. However, this program keeps track of the number of wins and losses, at home and away, then prints these values at the end:

```
$1 ~ /Yankees/ {
                    if ($2 > $4) hw++
                    else         hl++
                }
$3 ~ /Yankees/ {
                    if ($4 > $2) aw++
                    else         al++
                }
END  {
         printf "Home Wins: %d\n", hw
         printf "Home Losses: %d\n", hl
         printf "Away Wins: %d\n", aw
         printf "Away Losses: %d\n", al
     }
```

4.2.1 A Word on Style

Note the way in which indentation is used in the preceding program:

- Except in trivial cases, the program begins a new line after after every opening brace ({).
- Every `else` is lined up under the corresponding `if`.
- Parallel statements, like the sequence of `printf` instructions, are lined up underneath each other.

It is not necessary to write `nawk` programs in this way, but appropriate indentation and spacing make programs easier to read and understand. Your style for writing programs can also help you spot errors as you type in your program. For example, if you always try to make opening and closing braces line up, it is easy to notice if you leave out a brace.

The indentation format used in the rest of this guide demonstrates a clean readable programming style. All programmers develop personal preferences as they become familiar with a language, and you may decide to deviate from this guide's style in some respects. The important thing is to have a style and to follow it consistently in all your programs. It may not make much difference now, when your programs are relatively simple; but as your programs become more complex, you will find that style will be an important aid to writing programs that work correctly.

4.3 Using Compound Statements

In an `if` statement, you might sometimes want to perform several instructions. You can do this by enclosing the instructions in braces. Such a construct is called a **compound statement**.

For example, consider the following program:

```
{
        if ($2 > $4) {
                homewin++
                printf "The %s defeated the %s.\n", $1, $3
        } else {
                homeloss++
                printf "The %s defeated the %s.\n", $3, $1
        }
}
END  {
        printf "The home team won %d times.\n", homewin
        printf "The home team lost %d times.\n", homeloss
}
```

The first action is applied to every record in the file. It keeps a count of how many times the home team wins and how many times the home team loses. It also prints out a line telling who defeated whom. The `END` action summarizes the results after they have been calculated.

As another example, the following program examines the games involving the Orioles:

```
$1 ~ /Orioles/ {
        if ($2 > $4) {
                win++         # Home win
                printf "%s: %d, %s: %d\n",$1,$2,$3,$4
        } else {
                loss++        # Home loss
```

```
                printf "%s: %d, %s: %d\n",$3,$4,$1,$2
        }
}
$3 ~ /Orioles/ {
        if ($4 > $2) {
                win++           # Away win
                printf "%s: %d, %s: %d\n",$3,$4,$1,$2
        } else {
                loss++      # Away loss
                printf "%s: %d, %s: %d\n",$1,$2,$3,$4
        }
}
END  {
        printf "Wins: %d, Losses: %d\n", win, loss
}
```

Each line of output from the first two actions will have the following form:

`Winning team:` *score*`, Losing team:` *score*

The final line of output (from the `END` rule) summarizes the Orioles' wins and losses.

Examine this program closely to see how it works. The program is straightforward, but you should make sure you understand how it covers all the possible cases.

One `if` statement can contain another. For example, the previous program could have been written as follows:

```
/Orioles/ {
        if ($2 > $4) {          # Home team wins
                printf "%s: %d, %s: %d\n",$1,$2,$3,$4
                if ($1 ~ /Orioles/)
                        win++
                else
                        loss++
        } else {                # Home team loses
                printf "%s: %d, %s: %d\n",$3,$4,$1,$2
                if ($3 ~ /Orioles/)
                        win++
                else
                        loss++
        }
}
END  {
        printf "Wins: %d, Losses: %d\n", win, loss
}
```

This version of the program determines whether the game was won by the home team, prints out the scores with the winner first, and then checks to see if the Orioles were the home team or the visitors. The previous version of the program split the problem into two parts: one action performed when the Orioles were the home team and one when they were not.

4.4 The while Loop

A `while` loop repeats one or more other instructions as long as a given condition holds true. A `while` loop has the following format:

while (*expression*) *statement*

The *statement* can be a single statement or a compound statement. For example, the file `numbers` contains a set of one to ten random numbers on each line. The following program adds up the numbers on each line and prints the line's total:

```
{
        sum = 0
        i = 1
        while (i <= NF) {
                sum = sum + $i
                i = i + 1
        }
        print sum
}
```

The variable `i` counts fields in the record. While `i` is less than or equal to the total number of fields in the record, the `while` loop adds the value of the *i*th field to `sum` and then adds 1 to `i`. The loop then starts again; if the new value of `i` is still less than or equal to the total number of fields, the loop adds the value of the next field. The loop stops when `i` is greater than `NF`.

As another example, here is a program that uses the same data file and prints out the maximum value on each line:

```
{
        max = $1                # starting max is field 1
        i = 2
        while (i <= NF) {
                if ($i > max) max = $i
                i = i + 1
        }
        print max
}
```

On each line, the variable `max` starts out with the value of the first field (the first number). The `while` loop then moves across the record number by number, using an `if` statement to test whether a field is greater than the current value of `max`. If a greater value is found, `max` is assigned the new maximum value. After the loop, the maximum value is printed.

What does this program do if there is only one number on a particular line? In that case, `NF` would be 1. The `nawk` program would execute the following statements and find that `i` was already greater than `NF`:

```
max = $1
i = 2
while (i <= NF) ...
```

Therefore, `nawk` would not execute any of the instructions in the `while` loop at all. If the condition part of a `while` loop is false when the loop is first encountered, the statements in the loop are not executed.

As an exercise, try to write a program that reads a normal text file and writes out the text, one word per line.

4.5 The for Loop

A `for` loop is another way to repeat instructions as long as a given condition holds true. A `for` loop has the following format:

for (*expression1*;*expression2*;*expression3*) *statement*

This loop is equivalent to the following instruction sequence:

```
expression1
while (expression2) {
        statement
        expression3
}
```

For example, you could write the exercise given at the end of Section 4.4 as follows:

```
{
        for (i = NF; i > 0; i--)
                printf "%s ", $i
        printf "\n"
}
```

The program that prints the maximum value in an input line could be written as follows:

```
{
        max = $1
        for (i = 2; i <= NF; i++)
                if ($i > max) max = $i
        print max
}
```

As you can see, the `for` loop is just a short-hand way of writing a certain kind of `while` loop. Another form of the `for` loop is described in Chapter 6.

4.6 The next Statement

The `next` statement tells `nawk` to skip immediately to the next record in the data file. In the following example, a `next` statement is added to the baseball score program from Section 4.2.

```
{
        if (NF < 4) {
                printf "Not enough fields: %s\n", $0
                next
        }
        if ($2 > $4) print "Home Win"
        else print "Home loss"
}
```

If a particular record has less than four fields, this program will print a warning message and skip to processing the next record. This bypasses the rest of the instructions in the rule. It also bypasses any other rules that might normally be applied to this record. As this example shows, `next` is often used when a program finds a record that does not have the format you expect.

You can also use `next` to skip to the next record if you do not want the record processed by any of the remaining rules. For example:

```
$1 ~ /Orioles/  {count++; next}
$3 ~ /Orioles/  {count++}
```

This program prevents the record from being counted twice if it happens to have `Orioles` in both the first and third fields. You could also write this program as follows:

```
($1 ~ /Orioles/) || ($3 ~ /Orioles/) { count++ }
```

Using the `next` instruction inside a `BEGIN` rule tells `nawk` to start normal processing (by reading the first record of the first file). In other words, the `next` instruction indicates that you have finished the action associated with the `BEGIN` pattern.

4.7 The exit Statement

The `exit` statement makes a `nawk` program behave as if it has just reached the end of data input. No further input is read. If there is an `END` action, it is executed before the program terminates. As with `next`, `exit` is often used when input data is found to be in error.

If `exit` appears inside the `END` action, it terminates the program immediately.

String Manipulation 5

The preceding chapters have used quoted strings extensively. This chapter discusses strings in more detail and shows the various operations that manipulate strings.

5.1 String Variables

In Chapter 2, you learned how to use numeric variables: variables that contained numbers. Variables can also contain strings. For example:

```
a = "string"
```

This statement assigns a string to a variable `a`. As an example of how this can be used, here is a simple program that checks a text file for duplicate lines (places where two adjacent lines are identical):

```
{
     if ($0 == lastline) printf "%d: %s\n", FNR, $0
     lastline = $0
}
```

The variable `lastline` represents the contents of the previous line in the file. In the action of the program, the current record `$0` is compared to the previous record (stored in `lastline`). If the two are equal, the `printf` action prints the line number `FNR` and the contents of the line. At the end of the action, `lastline` is assigned the contents of the current line (so that it can be compared to the next line).

You might wonder what `lastline` contains when the program first begins. After all, nothing is assigned to `lastline` until the first line has been read. All string variables begin with a `null string` value. A null string is a string, but it contains no characters. It is written `""`. When used in an arithmetic expression, a null string has the value 0.

As another example of a program that uses string variables, here is a program that writes out the last line of a file:

```
    { line = $0 }
END { print line }
```

The value of each input line is assigned to the variable `line`. At the end of the file, `line` contains the contents of the last line in the file. Therefore, the `END` action prints out the contents of that line.

5.1.1 Built-In String Variables

In Chapter 3, you learned about the built-in numeric variables `NF`, `NR`, and `FNR`. The `nawk` language also provides the built-in string variables shown in Table 5-1.

Table 5-1: Built-In String Variables

Variable	Description
FILENAME	Contains the name of the current input file. For example, when you apply programs to the hobbies file, the value of FILENAME is hobbies (if that is the file you are using). If the input is coming from the nawk standard input, the value of FILENAME is the string "-".
FS	The **field separator** string. Specifies the character that is used to separate fields in the current file. The default value for FS is " " (a single blank), which as a special case matches both blank and tab. However, if the command line contains a -F option specifying a different field separator, FS is a string containing the given separator character. A program can also assign values to FS to indicate new field separator characters. For example, you could create a data file whose first line gives the character that is to be used to separate fields in the records in the rest of the file. A nawk program could then contain the following rule: `FNR == 1 { FS = $0 }` This says that the field separator string FS is to be assigned the contents of the first record in the current data file. The character in this line will then be used as the field separator for the rest of the file (unless the program changes the value of FS again). Any FS value of more than one character is used as a regular expression. See the INPUT section of the nawk(1) reference page for details.

RS	The **input record separator** string. Just as FS specifies the string that is used to separate fields within records, RS specifies the string that is used to separate one record from another. By default, RS contains a new-line character, which means that input records are separated by new-line characters. However, a different character may be assigned to RS. For example, the following statement says that input records are separated by semicolons (;):
	`RS = ";"`
	This would let you have several records on one line, or a single record that extends over several lines.
	To separate records by empty lines, specify the following:
	`RS = ""`

OFS	The **output field separator** string. When the print action is used to print several values, as in { print A, B, C }, the output field separator string is printed between each two of the values. By default, OFS contains a single blank character. However, if you make the assignment OFS = " : ", the output values will be separated by space-colon-space.
ORS	The **output record separator** string. When the print action is used, the output record separator is printed at the end of each record. By default, ORS is the new-line character.
OFMT	The **default output format** for numbers when they are printed by print. This is a format string like the one used by printf. By default, it is %.6g, indicating that numbers are to be printed with a maximum of six digits after the decimal point. By changing OFMT, you can display more or less precision.

5.1.2 String vs. Numeric Variables

Because string variables start out with the null string value while numeric variables start out as 0, the question arises: how can nawk differentiate between string and numeric variables, especially when execution is starting and a variable has not been used yet? The answer is that a variable is assumed to contain a string unless you use it as a number. For example, if you have a program that consists of

```
{ print X }
```

with no value assigned to X, the variable is assumed to be a string. Thus, the output will be a blank line for each line of input; if X had been taken as a number, the output would be zero for each line of input.

In an action like X = $1, the variable X will be taken as a number if the form of $1 looks like a number; otherwise, it will be taken as a string. Consider the record in the following example:

```
3 ...
```

Here, the first field looks like a number, so X will normally be taken to be a numeric variable. On the other hand, consider this example:

```
7ABC ...
```

The first field cannot be a number (even though it starts with a digit), so X will be taken to be a string variable.

There are times when you want a value to be treated as a string, even though it looks like a number. For example, suppose a file contains the string 1e1. In some contexts, this could be a number (with an exponential part); in other contexts, you might want to interpret this as a string. To make sure that a value is taken as a string, even when it might look numeric, concatenate it with an empty string, by placing a pair of quotation marks (" ") after it. For example:

```
X = $2 ""
```

This makes sure that the value in $2 is interpreted as a string, even if it looks like a number. Therefore, X will be a string variable.

Similarly, if you want to make sure that a value is taken to be a number, just add zero to it. For example:

```
X = $3 + 0
```

In this case, $3 will be taken to be a number because it is involved in an arithmetic operation. What happens if $3 is not a valid number? If $3 starts with something that looks like a number, as in 7ABC, the numeric value of the string is the number. Thus, the numeric value of 7ABC is 7. If the field does not start with anything that looks like a number, the numeric value of the string is zero. Thus the numeric value of ABC is 0.

5.2 String Concatenation

When a line in a program contains two or more strings that are separated only by blank characters, the strings are concatenated (joined) into one long string. The following expression is an example of string concatenation:

```
$2 ""
```

The following action prints the contents of the first three fields, joined together into one string:

```
{ print $1 $2 $3 }
```

Suppose your input line is:

```
A B C
```

Then the output will be as follows:

```
ABC
```

Consider the following example as applied to the `hobbies` file:

```
$1 ~ /John/ { print "$" $4 }
```

This example's output looks like this:

```
$100.00
$30.00
```

The dollar sign (`$`) is concatenated with the contents of the fourth field in all the appropriate records.

5.3 String Manipulation Functions

Chapter 3 introduced numeric functions like `sin` and `sqrt`. The `nawk` language also provides the following functions that perform string operations:

`length`

Returns an integer that is the length of the current record (the number of characters in the record, without the new-line on the end). For example, the following program calculates the total number of characters in a file (except for new-line characters):

```
    { sum = sum + length }
END { print sum }
```

`length(`*s*`)`

Returns an integer that is the length of the string *s*. For example, the following program prints out the length of the first field in each record of the data file:

```
{ print length($1) }
```

The function call `length($0)` is equivalent to `length`.

`gsub(`*regexp*`,`*replacement*`)`

Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the current record. For example:

```
{
    gsub(/John/,"Jonathan")
    print
}
```

This program checks every record in the data file for the regular expression `John`. Every matching string is replaced with `Jonathan` and printed out. As a result, the output of the program is exactly like the input except that every occurrence of `John` has been changed to `Jonathan`. This form of the `gsub` function returns an integer that tells how many substitutions were made in the current record. This result will be zero if the record has no strings that match *regexp*.

`sub(`*regexp,replacement*`)`

Works like `gsub`, except that it only replaces the first occurrence of a string matching *regexp* in the current record.

`gsub(`*regexp,replacement,string_var*`)`

Puts the replacement string *replacement* in place of every string matching the regular expression *regexp* in the string *string_var*. For example:

```
{
    gsub(/John/,"Jonathan",$1)
    print
}
```

This program is similar to the previous program, but the replacement is only made in the first field of each record. This form of the `gsub` function returns an integer that tells how many substitutions were made in *string_var*.

`sub(`*regexp,replacement,string_var*`)`

Works like `gsub`, except that it replaces only the first occurrence of a string matching *regexp* in the string *string_var*.

`index(`*string,substring*`)`

Searches the given *string* for the appearance of the given *substring*. If the *substring* cannot be found, `index` returns zero; otherwise, it returns the number (origin 1) of the character in *string* where *substring* begins. For example:

```
index("abcd","cd")
```

This program returns the integer 3 because `cd` is found beginning at the third character of `abcd`.

`match(`*string,regexp*`)`

Determines if *string* contains a substring that matches the regular expression (pattern) *regexp*. If so, `match` returns an index giving the position of the matching substring within *string*; if not, it returns zero. This function also sets a variable named `RSTART` to the index where the matching string starts, and sets a variable named `RLENGTH` to the length of the matching string.

`substr(`*string,pos*`)`

Returns the last part of *string*, beginning at a particular character position. The argument *pos* is an integer, giving the number of a character. Numbering begins at 1. For example:

```
substr("abcd",3)
```

The value of this expression is the string `cd`.

`substr(`*string,pos,length*`)`

Returns the part of *string* that begins at the character position given by *pos* and has the length given by *length*. For example:

```
substr("abcdefg",3,2)
```

The value of this expression is `cd` (a string of length 2 beginning at position 3).

`sprintf(`*format,value1,value2,...*`)`

Returns the string value that would be printed by the following `printf` action:

```
printf(format,value1,value2,...)
```

For example,

```
str = sprintf("%d %d!!!\n",2,3)
```

assigns the string

```
"2 3!!!\n"
```

to the string variable `str`.

`tolower(`*string*`)`

Returns the value of *string*, but with all the letters in lowercase. (This function is not found in all versions of `awk`.)

`toupper(`*string*`)`

Returns the value of *string*, but with all the letters in uppercase. (This function is not found in all versions of `awk`.)

`ord(`*string*`)`

Converts the first character of *string* into a number. This number gives the decimal value of the character in the ASCII character set. (This function is not found in all versions of `awk`.)

Arrays 6

In most programming languages, an array is an ordered list of values, similar to a table of information. Arrays in nawk are more flexible than arrays in most other languages, but it is helpful to begin by discussing the traditional concept of an array.

6.1 Arrays with Integer Subscripts

The simplest sort of array is a list of values (either numbers or strings). The values in the list are called the **elements** of the array.

Elements in an array are most commonly referred to by number. For example, the first element in the array could be number 1, the second could be number 2, and so on. These numbers are called **subscripts** of the array elements.

A nawk array has a name, similar to a variable name. To refer to an element of an array, you give the name of the array followed by brackets containing the element's subscript. For example:

```
arr[3]
```

This statement refers to element 3 in an array named arr.

A statement like the following creates an array named arr whose elements are all the fields of the current record:

```
for (i=1; i<=NF; i++)
    arr[i] = $i
```

The following program stores the entire contents of the input file in an array called lines:

```
{ lines[NR] = $0 }
```

Remember that the variable NR is incremented by 1 for each line that is read in, so the elements in the lines array will be the lines of the input file, in order.

The following program reads the contents of a data file and stores the input in lines:

```
     { lines[NR] = $0 }
END  { for (i=NR; i>0; i--) print lines[i] }
```

When all the lines have been read in, the END action prints out the lines in reverse order. The program therefore reads lines of text and then prints them in reverse order.

As another example of the simple use of arrays, suppose you have a file that contains 12 columns of numbers and you want to add up the numbers in each column. You could do this with the following program:

```
     { for (i=1; i<=12; i++) sum[i] = sum[i] + $i }
END  { for (i=1; i<=12; i++) print sum[i] }
```

Each element in the array called `sum` holds a running total of the sum of numbers in the corresponding column.

Notice that the previous examples make extensive use of the `for` statement. This is true of many programs that use arrays.

Also notice that you do not need a special statement to create (declare) an array. If a statement in a program contains a name followed by a value in brackets, the name is assumed to refer to an array, and the array is created automatically. A name must not be used as both a variable and an array in the same `nawk` program.

6.2 Generalized Arrays

Most programming languages let you create arrays that use numbers as subscripts; `nawk` also lets you create arrays that have string values as subscripts. For example, here is a program that calculates how much each person spends on all his or her hobbies.

```
{ money[$1] += $4 }
```

The array in this program is named `money`; the subscripts are the names of the people in the `hobbies` file. The elements of the array are therefore as follows:

```
money["Jim"]
money["Linda"]
money["John"]
   .
   .
   .
```

(Note that the following statements are equivalent:

```
money[$1] += $4
money[$1] = money[$1] + $4
```

This notation is explained in Section 8.3.)

Apply this program to the following input record:

```
Jim     reading          15     100.00
```

The action becomes

```
money["Jim"] += 100.00
```

As with all numeric variables, `money["Jim"]` starts out with a value of zero. At the end of the program, the array element will contain the amount of money that Jim spends on all his hobbies.

To print the contents of the `money` array, you can use a new form of the `for` statement:

```
for (s in money) print s, money[s]
```

This form of the `for` statement executes the `print` action once for every value that is used as a subscript for the `money` array. In each loop, the variable `s` has one of the subscript values. Therefore, the first time through the loop, `s` might have the value `Jim`, the next time `Linda`, and so on. The order is undefined. Therefore, the complete program prints out the amount that each person spends on his or her hobbies:

```
     { money[$1] += $4 }
END  { for (s in money) print s, money[s] }
```

Run this program to see how it works. After you have done so, replace the `print`

action with `printf` to produce more understandable output.

Generalized arrays have a wide variety of applications. For example, the following program produces a list of all the words used in an input text file:

```
     { for (i=1; i<=NF; i++)
          wordlist[$i] = 1 }
END  { for (x in wordlist)
          print x }
```

Assigning 1 to each element of `wordlist` is just a dummy action; the important thing is that the program creates an element of `wordlist` whose subscript value is one of the words in the input text file. The `for` loop in the `END` action then prints out all the words that were used as subscript values; this list is the set of all words used in the file.

As an exercise, modify the preceding program so that it keeps a count of how often each word is used in the input file. At the end, the program should print out each word that appears in the file and how often the word was used.

6.2.1 String Subscripts vs. Numeric Subscripts

This chapter began by showing arrays with numeric subscripts because those types of arrays are most familiar to programmers. However, all `nawk` array subscripts are converted to strings. For example, the subscript in `a[1]` is converted to a string, giving `a["1"]`. In `a[01]`, the numeric subscript is first converted to its simplest form, `a[1]`, which is then converted to the string `a["1"]` as before.

Floating point subscripts are converted to the simplest equivalent integer, then converted to the corresponding string. Thus `a[1.0]` is converted to `a[1]` and then converted to `a["1"]`. Therefore, the following forms are all equivalent:

```
a[1]   a[1.0]   a["1"]
```

Note that the array element `a["01"]` is not equivalent to the ones in the preceding examples because `"1"` is not the same string as `"01"`.

6.3 Deleting Array Elements

Because array elements are stored in the computer's memory, you can decrease memory requirements by deleting elements when you are finished using them. To do this, use the following statement:

`delete` *arrayname*`[`*subscript*`]`

For example:

```
delete money["Jim"]
```

As an extension of standard `awk`, the following statement deletes the entire array:

```
delete money
```

This statement is equivalent to the following:

```
for (ind in money)
    delete money[ind]
```

6.4 Multidimensional Arrays

The nawk language lets you define arrays with more than one subscript. Subscripts are separated by commas and enclosed in brackets, as in the following example:

```
a[1,2] = 3
b["cat", "dog", "bird"] = "horse"
```

The following example creates a multidimensional array that records different animal names:

```
name["chicken", "female"] = "hen"
name["chicken", "male"] = "rooster"
name["chicken", "young"] = "chick"
name["cattle", "female"] = "cow"
name["cattle", "male"] = "bull"
name["cattle", "young"] = "calf"
```

As you can see, it is simple to create and manipulate a database that is just a multidimensional nawk array.

User-Defined Functions 7

Previous chapters discuss numeric functions like `sin` and `sqrt`, and string functions like `gsub` and `length`. This chapter shows how `nawk` lets you create your own functions to perform similar kinds of operations.

7.1 Defining Functions

In a `nawk` program, a function definition looks like this:

function *name*(*argument-list*) {
    statements
}

The *argument-list* is a list of one or more names, separated by commas, that represent argument values passed to the function. When an argument name is used in the *statements* of a function, it is replaced by a copy of the corresponding argument value.

For example, here is a simple function that takes a single numeric argument `N` and returns a random integer between 1 and `N` (inclusive):

```
function random(N) {
        return (int(N * rand() + 1))
}
```

This function uses two built-in functions discussed in Chapter 3: `rand` (which returns a random floating point number between 0 and 1) and `int` (which returns the integer part of a floating point number). The expression `N * rand() + 1` yields a random floating point number between 1 and `N+1` (not including `N+1` itself). Applying the `int` function to this floating point number obtains an integer between 1 and `N`. The `return` statement returns this value as the result of the function `random`.

Once you define the `random` function, you can use it anywhere in your program that you would use other functions.

For example, if you have a file that contains people's names in its first field, and each of these people is going to roll two six-sided dice, you could simulate this situation with the following program:

```
function random(N) {
        return (int(N * rand() + 1))
}
{
        score = random(6) + random(6)
        printf "%s rolls %d\n", $1, score
}
```

This program consists of a definition for the `random` function and a rule to be applied to every record in the file. The `score` variable contains the sum of two simulated six-sided die rolls. This value is printed, along with the name of the person who rolled the dice.

You can test this program on the `hobbies` file. Remember, however, that the file contains several lines for most people, so the output will show more than one roll per person.

As another example of the `random` function, here is the program used to generate the random baseball scores in the `baseball` file. The input data file contains a single line giving the names of baseball teams (separated by tabs).

```
BEGIN  { FS = "\t" }  # Tab is field separator
function random(N) {
        # Produce random number between 1 and N
        return ( int(N * rand() + 1) )
}
{
        # Read in names of baseball teams
        for (i = 1; i <= NF; i++)
                team[i] = $i

        # Generate 100 random scores
        for (i = 1; i <= 100; i++) {

                # Choose teams
                hometeam = team[random(NF)]
                visteam = team[random(NF)]

                # Make sure teams are different
                while (hometeam == visteam)
                        visteam = team[random(NF)]

                # Generate scores
                homescore = random(13)
                visscore = random(13)

                # Make sure scores are different
                while (homescore == visscore)
                        visscore = random(13)

                # Print out score
                printf "%s\t%d\t",hometeam,homescore
                printf "%s\t%d\n",visteam,visscore
        }
}
```

The comments in the program should make it easy to understand what is happening in each section. The program chooses two different teams at random from the list in an input file. It then assigns each team a random score from 1 to 13 (a range typical of baseball scores) and prints the results with two `printf` statements. (We could also have used a single `printf` statement.)

As another example of the `random` function, here is the program used to generate the random lists of numbers in the `numbers` file:

```
function random(N) {
        # Produce random integer between 1 and N
        return ( int(N * rand() + 1) )
}
BEGIN {
        for (i = 1; i <= 30; i++) {
                for (j = random(10); j > 0; j--)
                        printf "%d ",random(100)
                printf "\n"
        }
        exit
}
```

This program has only a BEGIN rule. This rule prints out 30 lines, each of which contains a random number of integers in the range 1 to 100. Note that random is used both to choose the integers and to decide how many of these integers will appear on each line.

7.2 Recursion

A function can call itself; this process is called **recursion**. One example of a recursive function is the factorial function, which is called with the following form:

```
factorial(N)
```

This factorial function produces the number that is the product of all positive integers less than or equal to N. For example:

```
factorial(4)
```

The result of this expression is 4×3×2×1, or 24. The factorial of any N less than 1 is defined as 1.

The following function definition defines the factorial function recursively:

```
function factorial(N) {
        if (N <= 1)
                return 1
        else
                return N * factorial(N-1)
}
```

If N is less than or equal to 1, the factorial is 1. Otherwise, the factorial of N is N times the factorial of N-1. Thus the factorial of 4 (4×3×2×1) is 4 times the factorial of 3 (3×2×1). The factorial function calls itself recursively to figure out the appropriate result.

By the way, the factorial function demonstrates that a function can have more than one return statement. When a return statement is executed, the function immediately stops executing and returns the given value as the function result.

7.3 Call By Value

When a program calls a user-defined function, nawk makes copies of the argument values passed to the function and the function does all its work using those copies. For example, suppose a program is using a variable named X and calls a user-defined function F:

```
F(X)
```

The function F is given a copy of the current value of X. Because F only has a copy, the function cannot affect the current value of X: For example, consider this program:

```
function exchange(A,B) {
        temp = A
        A = B
        B = temp
}
{
        exchange($1,$2)
        print $0
}
```

In this program, it appears that the `exchange` function swaps the values of arguments `A` and `B`. The value of `A` is temporarily stored in `temp`; the value of `B` is assigned to `A` and the saved value of `A` is assigned to `B`. Now, when the main rule of the program issues the function call `exchange($1,$2)` does `nawk` swap the values of the first two fields of the current record? No, the function is only working with copies of the two fields; the function does not change the fields themselves.

Note that the definition of `exchange` does not have a `return` statement. It is not necessary for functions to return values. If a function does not have a `return` statement, the function ends when the last statement is executed.

If a function does not use `return` to return a result, do not use that function as if it did return a result. A function with no `return` statement yields a meaningless (undefined) result value.

7.4 Passing Arrays to Functions

When an array is passed as an argument to a function, it is passed **by reference**. This means that the function works with the actual array, not with a copy. Anything that the function does to the array has an effect on the original array.

For example, the `split` function is a built-in function that takes an array as an argument. It has the following form:

split(*string,array*)

The `split` function breaks up the *string* into fields, and assigns each of the fields to an element of the *array*. The first field is assigned to *array* `[1]`, the next to *array* `[2]`, and so on. Fields are assumed to be separated with the field separator string `FS`. If you want to use a different field separator string, you can use the following format:

split(*string,array,fsstring*)

The value of *fsstring* is the field separator string you want to use instead of `FS`. The result of `split` is the number of fields that *string* contained.

Note that `split` actually changes the elements of `array`. When an array is passed to a function, the function may change the array elements.

Enhancing Your nawk Programs 8

This chapter discusses additional ways you can tailor your nawk programs to serve your needs.

8.1 The getline Function

The getline function reads input from the current data file or from a different file. The function has several different forms, discussed in the sections that follow.

8.1.1 Reading from the Current Input

In its simplest form, getline is called as follows:

```
getline
```

This reads a new record from the current data file. The function automatically changes the value of $0 and all the other field values. It also changes variables like NF, NR, and FNR. In other words, using getline in this way is exactly like what happens when nawk reads in a new record in the normal way. For example:

```
/XYZ/ { print ; getline ; print }
```

First, this rule prints any record that contains the string XYZ. Next, the getline function reads the next record, and the final print prints that new record. Therefore, the rule prints every record that contains XYZ and also the record that follows (regardless of what the next record contains).

When getline reads a new record, the previous record is discarded; subsequent rules are applied to the new record, if appropriate. For example:

```
/XYZ/ { print ; getline ; print }
/ABC/ { ... some action ... }
```

The ABC rule in this program will be applied to the new record (if appropriate); it will not be applied to the XYZ record because that record is discarded when the new record is read.

If a call to getline appears in the BEGIN action, nawk immediately starts reading the first data file specified on the command line.

8.1.2 Reading a Line into a String Variable

The getline function can also be called in the following form:

getline *variable*

This form reads a new line from the current data file but assigns the contents of the line to the named string variable. The variables NR and FNR are changed to reflect that another record has been read from the input data file; however, the contents of $0 and NF are unchanged. Therefore, the following example reads a line into the variable X and compares this new line to the old line that is still stored in $0:

```
{
     getline X
     if (X == $0)
            print "Duplicate line"
}
```

8.1.3 Reading from a New File

Another form of `getline` reads a line from a different file instead of the current data file:

getline *var* <"*filename*"

This form of the function reads a line from the given file and stores the contents of the line in the string variable *var*. For example, here is a simple program that compares the current data file to another file named `testfile` and prints out a message if the two are not identical:

```
{
     getline X <"testfile"
     if ($0 != X)
            print "Not identical!"
}
```

This rule is executed for every line in the data file. Every time the action is executed, the `getline` function reads a new line from `testfile` and compares it with the current line from the data file. For every line read from the current data file, another line is read from `testfile` and the two lines are compared. If the two files differ at any point, the message "Not identical!" is printed.

A program may also call `getline` with the form

getline <"*filename*"

In this case, a line is read from the given file and assigned to `$0`. The value of `NF` is changed to reflect the new record in `$0`, but the variables `NR` and `FNR` are not changed because the record was not read from the current data file.

8.1.4 Reading from Other Commands

The `getline` function can also be used to read data produced by another command or program:

"*command*" | getline *var*

This form of the function executes the given *command* and gathers the command's output. The first line of output is piped into (assigned to) the string variable *var*. For example, the following program executes the `date` command and assigns the output of the command to the string variable `now`:

```
"date" | getline now
```

The following statements read the current date into the variable `now` and check to see if the date string contains `Apr`:

```
"date" | getline now
if (now ~ /.*Apr.*/)
    print "April Shower Time!"
```

You can also pipe command output into `$0`. This is done with a statement of the following form:

"*command*" | getline

This form of `getline` changes the value of `$0` and `NF` but does not change `NR` or `FNR`.

8.1.5 Redirecting Output to Files and Pipes

You can redirect the output of `print` and `printf` to a file or a pipe. Details are given in the Output section of the `nawk`(1) reference page.

Only a limited number of files and pipes can be opened at one time. You can use the `close` function to close files during execution. In this way, any number of files and pipes can be used during the execution of a `nawk` program. You can `close` both input files (used by `getline`) and output files (used by `print` and `printf`).

8.2 The system Function

The previous section showed how you can execute programs and system commands from `nawk` programs using the `getline` function. You can also execute commands with the `system` function. This function has the following form:

system("*command line*")

The following statement executes a `cd` command to change the current directory to directory `XYZ`:

```
system("cd XYZ")
```

8.3 Compound Assignments

The `nawk` language lets you use a shorthand notation for some common assignment operations. For example, the following statements are equivalent:

```
sum = sum + value
sum += value
```

Note, however, that the second form is simpler to write.

The += operation is an example of a **compound assignment**. Table 8-1 shows all the compound assignment operations of `nawk` and their equivalents:

Table 8-1: Compound Assignments

Compound Operation	Equivalent	Compound Operation	Equivalent
`A += B`	`A = A + B`	`A /= B`	`A = A / B`
`A -= B`	`A = A - B`	`A %= B`	`A = A % B`
`A *= B`	`A = A * B`	`A ^= B`	`A = A ^ B`

For example, you could use the following program on the `hobbies` file to calculate how many hours a week John spends on his hobbies:

```
/John/ { sum += $3 }
```

8.4 The sortgen Program

It can be difficult to remember all of the options to the `sort` command. As an example of the power of `nawk`, this section presents a `nawk` program, named `sortgen`, that generates the correct options for a specification.

The `sortgen` program is described in detail in *The AWK Programming Language*. Briefly, `sortgen` takes a description of the layout of the fields in a record and emits a command line for `sort` that will carry out the desired `sort`.

Note that `sortgen` uses 1-origin (the first field to be sorted on is field 1), and writes the `sort` command line to use `sort`'s 0-origin field labeling. Example 8-1 shows the definition of `sortgen`:

Example 8-1: sortgen Program for nawk

```
# sortgen - generate sort command
#   input: sequence of lines describing sort options
#   output: command line for sort

BEGIN { key = 0 }

/no |not |n't / { print "error: cannot do negatives:", $0; ok = 1 }

# rules for global variables
        { ok = 0 }
/uniq|discard.*(iden|dupl)/     { uniq = " -u"; ok = 1 }
/separ.*tab|tab.*separ/         { sep = "t'\t'"; ok = 1 }
/separ/                 { for (i = 1; i <= NF; i++)
                                if (length($i) == 1)
                                        sep = "t'" $i "'"
                          ok = 1
                        }
/key/   { key++; dokey(); ok = 1 }      # new key; must come in order

# rules for each key

/dict/                          { dict[key] = "d"; ok = 1 }
/ignore.*(space|blank)/         { blank[key] = "b"; ok = 1 }
/fold|case/                     { fold[key] = "f"; ok = 1 }
/num/                           { num[key] = "n"; ok = 1 }
/rev|descend|decreas|down|oppos/ { rev[key] = "r"; ok = 1 }
/month/                         { month[key] = "M"; ok = 1 }
/forward|ascend|increas|up|alpha/ { next }      # this is default
!ok             { print "error: cannot understand:", $0 }

END     {               # print flags for each key
        cmd = "sort" uniq
        flag = dict[0] blank[0] fold[0] rev[0] num[0] month[0] sep
        if (flag) cmd = cmd " -" flag
        for (i = 1; i <= key; i++)
                if (pos[i] != "") {
                        flag = pos[i] dict[i] blank[i] fold[i]
                        flag = flag rev[i] num[i] month[i]
                        if (flag) cmd = cmd " +" flag
                        if (pos2[i]) cmd = cmd " -" pos2[i]
                }
        print cmd
}
function dokey(   i) {          # determine position of key
        for (i = 1; i <= NF; i++)
                if ($i ~ /^[0-9]+$/) {
                        pos[key] = $i - 1       # sort uses 0-origin
```

Example 8-1: (continued)

```
                        break
                }
        for (i++; i <= NF; i++)
                if ($i ~ /^[0-9]+$/) {
                        pos2[key] = $i
                        break
                }
        if (pos[key] == "")
                printf("error: invalid key specification: %s\n", $0)
        if (pos2[key] == "")
                pos2[key] = pos[key] + 1
}
```

Order of Operations A

This appendix lists the order of operations for `nawk`, from highest precedence (operations done first) to lowest (operations done last). You can use parentheses `()` to change this ordering.

Operators	Description
`$`*i* `V[a]`	field, array element
`V++ V-- ++V --V`	increment, decrement
`A^B`	exponentiation
`+A -A !A`	unary plus, unary minus, logical NOT
`A*B A/B A%B`	multiplication, division, remainder
`A+B A-B`	addition, subtraction
`A B`	string concatenation
`A<B A>B A<=B A>=B A!=B A==B`	comparison
`A~B A!~B`	regular expression matching
`A in V`	array membership
`A && B`	logical AND
`A || B`	logical OR
`A ? B : C`	conditional expression
`V=B V+=B V-=B V*=B V/=B V%=B V^=B`	assignment

In this table, `A`, `B`, and `C` can be any expression; *i* is any expression yielding an integer; and `V` is any variable.

Example Files B

This appendix contains copies of all the example files used in this manual.

The hobbies File

Fields in this file are separated by **spaces**. When creating files that will use `nawk`'s default value for `FS`, you can enter a single space or as many spaces as needed to make the fields align neatly.

```
Jim      reading          15     100.00
Jim      bridge            4      10.00
Jim      role-playing      5      70.00
Linda    bridge           12      30.00
Linda    cartooning        5      75.00
Katie    jogging          14     120.00
Katie    reading          10      60.00
John     role-playing      8     100.00
John     jogging           8      30.00
Andrew   wind-surfing     20    1000.00
Lori     jogging           5      30.00
Lori     weight-lifting   12     200.00
Lori     bridge            2       0.00
```

The baseball File

Fields in this file are separated by **tabs**. Note that the fields do not line up uniformly when you look at the file on your terminal. This irregularity occurs because exactly *one* tab is used between fields; using multiple tabs to make the fields line up in neat columns would result in `nawk`'s seeing two adjacent tabs as the field separators before and after an empty field. When creating the `baseball` file, key in the information as in this example:

```
% cat > baseball
Brewers[TAB]5[TAB]Tigers[TAB]9
  .
  .
  .
[CTRL/D]
```

Here is the file:

```
Brewers 5       Tigers  9
Brewers 2       Blue Jays       6
Blue Jays       8       Red Sox 7
Indians 6       Blue Jays       7
Yankees 7       Brewers 2
Orioles 10      Indians 1
Brewers 6       Yankees 3
Red Sox 3       Indians 12
Red Sox 6       Yankees 2
Blue Jays       8       Brewers 2
```

```
Orioles 2       Blue Jays       7
Indians 6       Blue Jays       9
Orioles 6       Blue Jays       12
Red Sox 7       Blue Jays       11
Yankees 9       Indians 10
Brewers 4       Blue Jays       5
Tigers  9       Blue Jays       10
Tigers  10      Red Sox 9
Brewers 10      Red Sox 9
Indians 4       Tigers  12
Blue Jays       8       Brewers 5
Yankees 11      Tigers  2
Orioles 5       Red Sox 6
Yankees 12      Blue Jays       13
Orioles 1       Red Sox 8
Yankees 5       Brewers 4
Orioles 6       Indians 13
Indians 12      Tigers  9
Red Sox 3       Blue Jays       12
Blue Jays       9       Orioles 8
Yankees 9       Orioles 6
Orioles 10      Indians 7
Red Sox 5       Orioles 2
Yankees 13      Brewers 6
Orioles 4       Brewers 6
Yankees 11      Indians 9
Tigers  4       Indians 13
Red Sox 3       Brewers 10
Yankees 1       Indians 8
Yankees 8       Tigers  10
Orioles 1       Blue Jays       12
Blue Jays       9       Indians 8
Indians 8       Blue Jays       9
Brewers 2       Orioles 5
Brewers 2       Indians 7
Orioles 7       Indians 2
Yankees 4       Orioles 6
Red Sox 11      Orioles 12
Tigers  6       Brewers 13
Indians 11      Yankees 12
Orioles 8       Red Sox 7
Yankees 9       Brewers 13
Tigers  8       Indians 7
Indians 1       Blue Jays       8
Blue Jays       8       Red Sox 5
Indians 12      Tigers  9
Yankees 8       Indians 5
Indians 2       Orioles 12
Brewers 6       Red Sox 2
Brewers 13      Indians 9
Blue Jays       9       Tigers  7
Orioles 2       Yankees 11
Orioles 1       Blue Jays       9
Red Sox 5       Yankees 9
Brewers 3       Tigers  13
Blue Jays       8       Red Sox 6
Blue Jays       11      Brewers 5
Tigers  7       Brewers 3
Brewers 2       Tigers  5
Blue Jays       9       Red Sox 1
Red Sox 4       Indians 5
Yankees 12      Orioles 5
Brewers 4       Blue Jays       8
Tigers  2       Blue Jays       8
Orioles 4       Blue Jays       6
```

```
Orioles 10      Brewers 3
Tigers  5       Red Sox 2
Brewers 9       Tigers  12
Blue Jays       11      Tigers  1
Yankees 2       Blue Jays       13
Brewers 12      Orioles 6
Indians 4       Tigers  8
Red Sox 2       Tigers  7
Yankees 6       Brewers 11
Indians 8       Brewers 11
Yankees 8       Red Sox 11
Orioles 4       Yankees 5
Red Sox 9       Yankees 10
Yankees 8       Tigers  13
Indians 3       Brewers 8
Indians 1       Blue Jays       12
Red Sox 8       Brewers 13
Brewers 7       Orioles 6
Indians 11      Yankees 4
Yankees 3       Red Sox 11
Orioles 9       Indians 6
Indians 12      Red Sox 11
Tigers  11      Orioles 12
Brewers 7       Indians 9
Red Sox 13      Brewers 8
```

The numbers File

Fields in this file are separated by **spaces**.

```
74 33 66
8 87 40
68 46
53 40 5 45 50
19 54 12 55 35 70 77 5 22 100
44 21 66 43 20
58 98 44 12 2 20 12 60 55 12
2 43
10 46 1 57
46
58 7 52 83 90 43 63 69 64
17 2 46 42 14 84 7 65
83 63 73 63 15 59 71 63
35 82 24
14 23 60 35 94 95 82 82 10
48 59 33 39 99
90 88
51 50 58 1 56 86 94 19 31 26
50 36 42 41 95
40 76 88 68
7 94 5 5 49 68 56
44 69 41 45 33 72 47 60 49 35
96 21
46 52 47 26 26 45 89 34 79 65
36 28 93 63 20 17 73 96
5 56 88 79 60
55 1 1 91 12 36 67 58
42 12 57 63
55 13 35
33 11 47
```

Index

Special Characters

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to the nawk Utility
AA-PBKPA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line

Reader's Comments

ULTRIX
Guide to the nawk Utility
AA-PBKPA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

What do you like best about this manual? _____

What do you like least about this manual? _____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line