SEQUENT

DYNIX/ptx Programming Tools Guide

1003-48614-00

SEQUENT

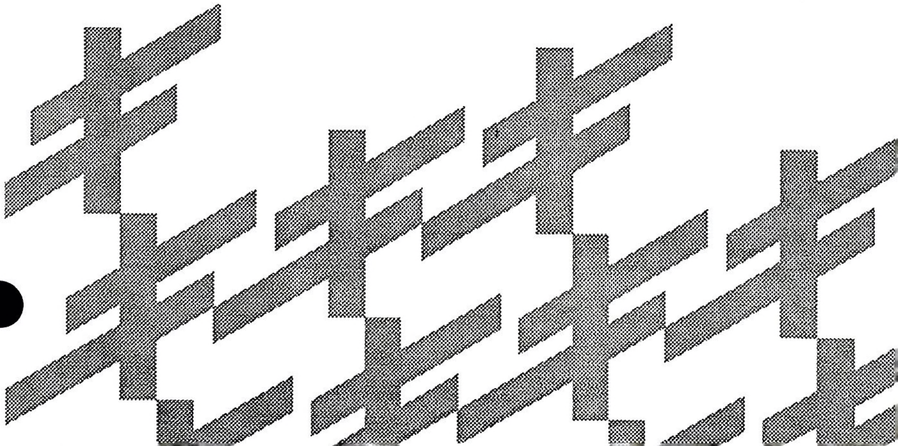
THIS SHEET LIFTER
MUST BE PLACED ON
TOP OF CONTENTS

GRAPHIC PRODUCTS

Chicago, Illinois 60609

DYNIX/ptx®
Programming Tools Guide

1003-48614-00
Sequent Computer Systems, Inc.



DYNIX and DYNIX/ptx are registered trademarks of Sequent Computer Systems, Inc.

Symmetry is a trademark of Sequent Computer Systems, Inc.

UNIX is a registered trademark of AT&T.

Copyright © 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America.

Contents

Figures

Tables

About This Guide

Overview

This document is written for programmers working in the DYNIX/ptx operating system environment. Its purpose is to familiarize programmers with a set of tools that make programming tasks easier. Each chapter has its own index. This document covers the following topics:

Shell Programming	Use of the shell to perform a series of tasks and commands.
nawk	A new version of a file-processing programming language.
awk	A file-processing programming language.
lex	A tool for implementing a lexical analyzer
yacc	A tool for imposing structure on the input to a program.
make	A tool for using a command file to perform a task.
SCCS	A tool, the Source Code Control System, to provide for file revision control.

Assumptions About the Reader

You should be familiar with the following topics:

- The DYNIX/ptx operating system
- The operating system shell
- Operating system documentation conventions

Notational Conventions

The following notational conventions are used in this manual:

- File names and user-defined parameters in text are shown in *italics*.
- System output and system calls and signals are shown in `even-width` font.
- Commands in text and examples of input to be entered literally are shown in **bold**.
- Environment variables are shown in **bold** upper or lowercase.

1. Shell Programming

1. Shell Programming

Chapter 1
Shell Programming

Tables

Chapter 1
Shell Programming

1.1 Introduction

You can use the shell to create programs for new commands. Such programs are also called shell scripts. Writing shell scripts is called *shell programming.* This chapter tells you how to create and execute shell programs using commands, variables, positional parameters, return codes, and basic programming control structures. Information on programming in the C shell and the Korn shell is found later in this chapter. Before you start shell programming, you should be familiar with the information contained in the following:

- "Using the Shell" and "Managing Processes" in the *User's Guide.*
- **sh**(1), **csh**(1), and **ksh**(1) in the *Reference Manual*

1.1.1 Creating a bin Directory for Executable Files

To make your shell programs accessible from all your directories, you can create a *bin* directory in your home directory. Create your shell programs in your *bin* directory, or move your completed programs to that directory. Be sure to set your **PATH** environment variable to include your *bin* directory. After you set the **PATH** environment variable, you can execute the programs in your *bin* directory from any directory within your home directory. For more information about the **PATH** environment variable, refer to "Using the Shell" in the *User's Guide.*

1.1.2 Naming Shell Programs

You can name your shell programs whatever you want, as long as the names meet the normal filename requirements. However, you should not give your programs the same name as system commands. If you do, the system may execute your command instead of the system command. If your **PATH** environment variable contains the *bin* directory ahead of the directory containing the system command, the shell finds your command and executes it. For example, if you name a program *mv* and then try to move a file, the operating system executes your program instead.

You can also have a problem if you name a shell program file something like **ls**. If your program contains an **ls** command, the shell reads it, but tries to execute your **ls** command, forming an infinite loop.

The operating system sets a limit on how many times an infinite loop can execute. After some time, the system displays the following error message:

```
Too many processes, cannot fork
```

One way to keep this from happening is to give the full pathname for the system's **ls** command, **/bin/ls**, when you write your own shell program.

The following **ls** shell program would work:

```
$ cat ls [Return]
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system **ls** command by using its full pathname, **/bin/ls**.

1.1.3 Creating a Shell Program

To create a shell program, follow these steps:

1. Use an editor to create a file containing the program.
2. Use the **sh** command to try out the program.
3. Use the **chmod** command to give execute permission to the file containing the program. If you want your group or all other users to be able to use the program, give them execute permission on the file.

NOTE

The operating system does not allow **setuid** *shell programs.*

4. If you didn't create the program in your *bin* directory, move the file containing the program to that directory.

For a basic example of shell programming, follow the steps in this section to create a simple shell program that performs the following tasks:

- Prints the current directory
- Lists the contents of that directory
- Displays this message on your terminal: `This is the end of the shell program.`

1. **Create a file containing the program.**

   Using an editor, create a file called *dl* and enter the following lines:

   ```
   pwd [Return]
   ls [Return]
   echo This is the end of the shell program. [Return]
   ```

   Now write and quit the file. You have just created a shell program.

2. **Test the program.**

   The **sh** command executes a shell program. It provides a good way to test your shell program. To execute the *dl* program, type this command:

   ```
   sh dl [Return]
   ```

   The **dl** command prints the pathname of the current directory, followed by a list of files in the current directory and the comment `This is the end of the shell program.`

3. **Use the chmod command to make the file executable.**

   ```
   $ chmod 700 dl [Return]
   ```

   If you want to verify the permissions on the *dl* file, use the **ls -l** command:

   ```
   $ ls -l dl [Return]
   -rwx------   1   annie   eng   48 Nov 15   10:50   dl
   ```

4. **Execute the dl command.**

   You can now execute the **dl** program by typing the following command line:

   ```
   $ dl [Return]
   ```

1.1.4 Running Programs

Bourne shell is by convention the shell interpreter. You can specify the shell to run your shell script by putting a line like the following is at the beginning of your shell script:

```
#!/bin/sh
```

Regardless of what shell the script was invoked from, or whether the **SHELL** environment variable is set, the shell script is run by that shell. If a shell is not specified at the beginning of a shell script, the following rules apply:

- If you are running Bourne shell, the shell script is run by the Bourne shell. (The **SHELL** environment variable is ignored by the Bourne shell.)
- If you are running in C shell and the first character in the file is a pound sign (#), the shell script is run by the C shell; otherwise; it it run by the Bourne shell.
- If you are running Korn shell, and the **SHELL** environment variable is set, the shell specified by **SHELL** is used, (The **SHELL** variable is automatically set to Korn shell if your default login is the Korn shell.)

1.2 Variables

Variables are the basic data objects, other than files, that shell programs manipulate. There are three main types of variables:

- Positional parameters
- Special parameters
- Named variables

1.2.1 Positional Parameters

A positional parameter is a variable within a shell program. It takes its value from an argument specified on the command line. Positional parameters are numbered and are referred to with a preceding dollar sign, **$**: **$1**, **$2**, **$3**, and so on.

A shell program can reference up to nine positional parameters. For example, the following command line invokes a shell program named **shell.prog**:

```
$ shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 [Return]
```

Positional parameter **$1** within the program is assigned the value **pp1**, positional parameter **$2** within the program is assigned the value **pp2**, and so on.

As another example, you could create a program called *pp* containing the **echo** commands shown in the following display:

```
$ cat pp [Return]
echo  The first positional parameter is: $1
echo  The second positional parameter is: $2
echo  The third positional parameter is: $3
echo  The fourth positional parameter is: $4
$
```

If you execute this shell program with the arguments **one**, **two**, **three**, and **four**, you obtain the following results:

```
$ pp one two three four [Return]
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

As another example, the **who** system command lists all users currently logged in on a system. You can create a simple shell program called **whoson** that will tell you if a particular user is currently working on the system.

To create this program, type the following command line in a file called *whoson*:

```
who | grep $1
```

The **who** command lists all current system users, and **grep** searches the output of the **who** command for a line containing the value in positional parameter **$1**.

After making the program executable, you can type a login name as an argument on the command line. For example, if you issue the **whoson** command with the argument *anne*, the shell substitutes *anne* for the parameter **$1** in the program. The shell then executes the program as if it were the following command:

```
who | grep anne [Return]
```

The output is as follows:

```
$ whoson anne [Return]
anne   tty26      Jan 24 13:35
$
```

If the owner of the specified login is not currently working on the system, **grep** fails and **whoson** prints no output.

The shell allows a command line to contain 128 arguments. However, a shell program can reference only nine positional parameters (**$1** through **$9**) at a given time. You can use the **shift** command to work around this restriction. The special parameter **$***, described in the next section, can also be used to access the values of all command line arguments.

1.2.2 Special Parameters

Shell programs can contain special parameters that return the number of arguments or actual parameters, or display return codes. These parameters are described in the following paragraphs.

Return Number of Arguments

$# This parameter, when referenced within a shell program, contains the number of arguments with which the shell program was invoked. Its value can be used anywhere within the shell program.

For example, the *get.num* shell program following includes this parameter.

```
$ cat get.num [Return]
echo The number of arguments is: $#
$
```

The program simply displays the number of arguments with which it is invoked. The following example shows that it was invoked with four arguments.

```
$ get.num test out this program [Return]
The number of arguments is: 4
$
```

Return All Parameters

The **$*** special parameter, when referenced within a shell program, contains all of the arguments with which the shell program was invoked, starting with the first argument. You are not restricted to nine parameters, as with the positional parameters **$1** through **$9**.

The following shell program, *show.param*, will **echo** all of the arguments you give to the command.

```
$ cat show.param [Return]
echo The parameters are: $*
$
```

In the following example, the command was given the arguments **Hello. It's a wonderful day.** The **show.param** program then echoes `Hello.  It's a wonderful day.`

```
$ show.param Hello. It's a wonderful day. [Return]
The parameters are: Hello.  It's a wonderful day.
$
```

The following example gives **show.param** more than nine arguments, which **show.param** echoes.

```
$ show.param 1 2 3 4 5 6 7 8 9 10 11 [Return]
The parameters are: 1 2 3 4 5 6 7 8 9 10 11
$
```

The **$*** parameter can be helpful if you use filename expansion to specify arguments to a shell command. For example, you might have several files in your directory named for chapters of a book: **chap1** through **chap7**. If you give **show.param** the argument **chap?**, it prints a list of all of those files.

```
$ show.param chap? [Return]
The parameters are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

Display Return Codes

Most shell commands issue return codes that indicate whether the command executed properly. A return code is also called an *exit status*. By convention, if a zero is returned, the command executed properly. Any other value indicates that the command did not execute properly. The return code is not printed automatically, but is available as the value of the shell special parameter **$?**.

After you execute a command, you can type the following line to display the return code.

```
echo $?
```

In the following example, the file *hi* exists in your directory and has read permission for you. The **cat** command behaves as expected and outputs the contents of the file. It exits with a return code of 0, which you can see using the parameter **$?**.

```
$ cat hi [Return]
This is file hi.
$ echo $? [Return]
```

In the next example, the *hello* file either does not exist or does not have read permission for you. The **cat** command prints a diagnostic message and exits with a return code of 2.

```
$ cat hello [Return]
cat: cannot open hello
$ echo $? [Return]
2
$
```

1.2.3 Named Variables

Named variables are variables to which you can assign values. You can create your own named variables. The first character of the name must be a letter or an underscore. The rest of the name can contain letters, underscores, and digits. Avoid using the name of a shell command as a variable name. You should also avoid using the names of environment variables.

When you use a named variable in a shell program, precede the name of the variable with a dollar sign ($), which refers to the value of the variable. When the shell executes the program, it substitutes the value of the variable for the variable name.

For example, the **name1** variable could have the value *myname*. If you include **$name1** in a program, the shell substitutes the value *myname* for each occurrence of the character string **$name1**.

There are several ways to assign a value to a named variable:

- Assign a value on the command line.
- Use the **read** command to assign input to the variable.
- Use command substitution to redirect the output from a command to a variable.
- Assign a positional parameter to the variable.

The following sections discuss each of these methods.

Assigning a Value on the Command Line

To assign a value on the command line, make an entry having this format:

```
named_variable=value
```

Type the name of the variable, followed by an equal sign (=) and the value. There are no spaces on either side of the equal sign (=).

In the following example, the value **myname** is assigned to the **name1** variable.

```
$ name1=myname (Return)
```

Using the read Command

When you include the **read** command in a shell program, you can prompt the user of the program to type the values of variables.

The **read** command has this format:

```
read variable
```

The value the **read** command assigns to *variable* is substituted for $*variable* wherever it is used in the program. If a program executes the **echo** command just before the **read** command, the program can display directions such as `Type yes or no`. The **read** command waits until you type a character string followed by Return, then makes that string the value of the variable.

The following example shows how to write a simple shell program called **num.please** to keep track of your telephone numbers. This program uses the following commands:

echo	Prompts you for a person's last name
read	Assigns the input value to the variable **name**
grep	Searches the *list* file for this variable

Your finished program should look like this:

```
$ cat num.please Return
echo Type the last name:
read name
grep $name list
$
```

The next example is a program called *mknum*, which creates a list. *mknum* includes the following commands:

echo	Prompts for a person's name
read	Assigns the person's name to the variable **name**
echo	Asks for the person's number
read	Assigns the telephone number to the variable **num**

echo Adds the values of the variables **name** and **num** to the *list* file

The double output redirection symbol (>>) in the last **echo** command tells the shell to append the output from the command to the end of the *list* file.

Following is the completed **mknum** program, followed by a **chmod** command to make the program executable.

```
$ cat mknum [Return]
echo Type name
read name
echo Type number
read num
echo $name $num >> list
$ chmod u+x mknum [Return]
$
```

In the following display, **mknum** is used to create a new listing for Mr. Niceguy. **num.please** is then executed to obtain Mr. Niceguy's phone number.

```
$ mknum [Return]
Type the name
Mr. Niceguy [Return]
Type the number
668-0007 [Return]
$ num.please [Return]
Type last name
Niceguy [Return]
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as its value.

Assigning Positional Parameters

The following format allows you to assign a positional parameter as the value of a named parameter:

var1=**$1**

For example, you can create a program called *simp.p* that assigns a positional parameter to a variable. The following display shows the commands in **simp.p**:

```
$ cat simp.p [Return]
var1=$1
echo $var1
$
```

When **simp.p** is invoked with the argument **skyscraper**, the program prints that value.

```
$ simp.p skyscraper [Return]
skyscraper
$
```

Using Command Output

You can use command substitution within a shell program to assign a value to a variable. The shell assigns the command output as the value of the variable. Command substitution has this format:

variable=`command`

On the following command line, the **date** command is piped into the **cut** command to display the current time:

```
$ date | cut -c12-19 [Return]
```

This command is included in a shell program called *t*.

```
$ cat t [Return]
time=`date | cut -c12-19`
echo The time is: $time
$
```

After making the *t* file executable, you can run the t program when you want to see the time.

```
$ chmod u+x t [Return]
$ t [Return]
The time is: 10:36
$
```

The command being substituted can also use use positional parameters, as shown below.

person=`who | grep $1`

In the following display, the program *log.time* keeps track of the *whoson* program results. The output of *whoson* is assigned to the variable **person** and added to the file *login.file* with the **echo** command. The last **echo** displays the value of **$person**, which is the same as the output from the **whoson** command.

```
$ cat log.time [Return]
person=`who | grep $1`
echo $person >> login.file
echo $person
$
```

The following display shows the system response to *log.time*.

```
$ log.time maryann [Return]
maryann      tty61          Apr 11 10:26
$
```

1.3 Discarding Output: /dev/null

The filesystem includes a file called */dev/null* where the shell discards unwanted output.

To try out this feature, type a command such as **who** that produces output, then redirect the output to */dev/null.*

```
$ who > /dev/null [Return]
$
```

Notice that the system responded with a shell prompt ($). The output from the **who** command was placed in */dev/null* and effectively discarded. You could use this method to test the exit status of a command when you do not want to display the output.

1.4 Shell Programming Statements

The shell programming language includes several statements that add flexibility to your programs:

- Comments let you document the functions of a program.
- The *here document* allows you to include lines that are redirected to be the input for a command in the program.
- The **exit** command allows you to use return codes and to terminate a program at a point other than the end of the program.
- The looping statements, **for**, **while**, and **until** allow a program to iterate through groups of commands in a loop.
- The conditional control commands, **if** and **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command allows a program to exit unconditionally from a loop.
- The **continue** command allows a program to skip to the next iteration of a loop without executing the remaining commands in the loop.

1.4.1 Comments

To make it easy for other users to understand your programs, you can include comments, or text, in each program you create. These comments should describe the actions the program performs. To include a comment, type a pound sign (#), then type the text. The pound sign tells the shell to ignore the remaining text on that line. You can type the pound sign at the beginning of a line, in which case the comment uses the entire line. You can also type a comment after a command. In this case, the shell executes the command and ignores the remainder of the line. A comment always ends at the end of the line.

For example, the shell ignores the following comment lines when it executes a program containing these lines:

```
# This program sends a generic birthday greeting.
# This program needs a login as
# the positional parameter.
```

Remember not to use a pound sign at the beginning of a program unless you want to force a specific shell to run the program. Refer to “Running Programs” earlier in this chapter for more information on the comment (#) character.

1.4.2 The here Document

A *here document* allows you to place into a shell program lines that are redirected to be the input of a command in that program. It is a way to provide input to a command in a shell program without using a separate file. The notation consists of the input redirection symbol (<<) and a delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters; an exclamation point (!) is often used.

The following diagram shows the general format for a here document:

```
command <<delimiter
. . .input lines. . .
delimiter
```

In the following example, the program *bday* uses a here document to send a birthday greeting by redirecting lines of input into the **mail** command. An exclamation mark is used as the delimiter.

```
$ cat bday [Return]
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

The here document contains this line:

```
Best wishes to you on your birthday
```

When you run the **bday** command, you must specify the recipient's login as the argument to the command. For example, to send this greeting to user *mary*, you would type this command:

```
$ bday mary [Return]
```

mary receives your greeting the next time she reads her mail messages.

```
$ mail [Return]
From mylogin Wed May 14 14:31 CDT 1986
Best wishes to you on your birthday
$
```

Using an Editor in a Shell Program

The **ed** line editor or the **vi** screen editor can be useful when creating a here document. For example, you could design a shell program that enters the **ed** editor, makes a global substitution to a file, writes the file, and then quits **ed**. The following display shows the contents of a program called **ch.text** that performs these tasks.

```
$ cat ch.text [Return]
echo Type the filename.
read file1
echo Type the text to change.
read old_text
echo Type the replacement text.
read new_text
ed - $file1 <<!
g/$old_text/s//$new_text/g
w
q
!
$
```

Notice that the editor is invoked as **ed –**. This option prevents the character count from being displayed on the screen.

The program uses three variables: *file1*, *old_text*, and *new_text*.

file	The name of the file to be edited
old_text	The text to change
new_text	The new text

When the program is run, it uses the **read** command to obtain the values of these variables. Once the variables are entered in the program, the here document redirects the global substitution, the **write** command, and the **quit** command into the **ed** command.

The following display shows sample responses to the program prompts:

```
$ ch.text [Return]
Type the filename.
memo [Return]
Type the text to change.
Dear John: [Return]
Type the replacement text.
To whom it may concern: [Return]
$
```

You can run the **cat** command on the changed file to examine the results of the global substitution.

```
$ cat memo [Return]
To whom it may concern:
$
```

The stream editor, **sed**(1), can also be used in shell programming.

1.4.3 Looping

In the previous examples in this chapter, the commands in shell programs have been executed in sequence. The **for**, **while**, and **until** looping statements allow a program to execute a command or sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands once for each member of a list. It has this format:

```
for variable
     in a_list_of_values
do
     command 1
     command 2
         .
         .
         .
     last command
done
```

For each iteration of the loop, the next member of the list is assigned to the variable given in the **for** clause. References to that variable may be made anywhere in the commands within the **do** clause.

It is easier to read a shell program if the looping statements are visually obvious. Since the shell ignores spaces at the beginning of lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can easily check to make sure each **do** has a corresponding **done** at the end of the loop.

The variable can be any name you choose. For example, if you call it **var**, then the values given in the list after the keyword **in** are assigned in turn to **var**; references to **$var** within the command list make the value available. If the **in** clause is omitted, the values for **var** are the complete set of arguments given to the command and available in the special parameter **$***. The command list between the keywords **do** and **done** are executed once for each value.

When the commands have been executed for the last value in the list, the program executes the next line below **done**. If there is no line, the program ends.

As an example, create a program to move files to another directory. It includes the following commands:

echo	Prompts the user for a pathname to the new directory.

read	Assigns the pathname to the variable **path.**
for *variable*	Calls the variable **file**; it can be referenced as **$file** in the command sequence.
in *list_of_values*	Supplies a list of values. If the **in** clause is omitted, the list of values is assumed to be **$*** (all the arguments entered on the command line).
do *command_sequence*	Provides a command sequence.

The statement for this program is as follows:

```
do
    mv $file $path/$file [Return]
done
```

The following example shows the text for the shell program **mv.file**:

```
$ cat mv.file [Return]
echo Please type the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done
$
```

In this program, the values for the variable *file* are already in the program. To change the files each time the program is invoked, assign the values using positional parameters or the **read** command. When positional parameters are used, the **in** keyword is not needed, as the next display shows:

```
$ cat mv.file [Return]
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done
$
```

You can move several files at once with this command by specifying a list of filenames as arguments to the command.

The while Loop

The **while** loop statement uses two groups of commands. It continues executing the sequence of commands in the second group, the **do...done** list, as long as the final command in the first group, the **while** list, returns an exit status of 0.

The **while** loop has this format:

```
while
    command1
    command2
      .
      .
    last command
do
    command1
    command2
      .
      .
    last command
done
```

For example, a program called **enter.name** uses a **while** loop to enter a list of names into a file. The program consists of the following command lines:

```
$ cat enter.name [Return]
while
   read x
do
   echo $x>>xfile
done
$
```

With some added refinements, the program looks like this:

```
$ cat enter.name [Return]
echo "Please type each person's name and press Return."
echo "Please end the list of names with Ctrl-D."
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

After the loop completes, the program executes the commands below **done**.

Because special characters are used in the **echo** command lines in the previous example, use double quotes to turn off the special meaning.

The next display shows the results of **enter.name**.

```
$ enter.name [Return]
Please type each person's name and press Return.
Please end the list of names with Ctrl-D.
Mary Lou [Return]
Janice [Return]
[Ctrl-D]
xfile contains the following names:
Mary Lou
Janice
$
```

After the loop completes, the program prints all of the names contained in **xfile**.

The until Loop

The **until** loop is similar to the **while** loop. A **while** loop continues execution as long as the last command in the first group of commands after the **while** returns a zero exit status. The **until** loop executes as long as the command after the **until** returns a nonzero exit status. As soon as a zero is returned, the loop terminates. Like the **while** loop, the commands between **do** and **done** may never execute. The format of the **until** loop is as follows:

```
until command
do
    command1
    command2
       .
       .
    last command
done
```

The **until** statement is useful for writing programs that wait for a particular event to occur.

1.4.4 Conditional Statements

Conditional statements allow shell programs to execute commands depending on whether or not certain conditions are met. The following paragraphs describe these conditional statements.

if...then

The **if** statement tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** statement ends with the keyword **fi**.

The **if** statement has this format:

```
if
      command1
      command2
         .
         .
      last command
  then
      command1
      command2
         .
         .
      last  command
fi
```

The following shell program called *search* demonstrates the use of the **if...then** statement. *search* uses the **grep** command to search for a word in a file. If **grep** is successful, an **echo** command prints a message indicating that the word is in the file.

```
$ cat search          [Return]
echo Type the word and the filename.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

The **read** command assigns values to two variables. The characters you type, up to the first space, are assigned to **word**. The rest of the characters, including embedded spaces, are assigned to *file*.

A problem with this program is the unwanted display of output from the **grep** command. If you want to dispose of the system response to the **grep** command in your program, use the file */dev/null*, changing the **if** command line to the following:

if grep $*word* $*file* > /dev/null

Now the **search** program responds only with the message specified after the **echo** command.

if...then...else

The **if...then** statement can issue an alternate set of commands with **else** when the **if** command sequence is false. **elif** is the equivalent of **else**.

It has this format:

```
if
     command1
     command2
          .
          .
     last command
  then
     command1
     command2
          .
          .
     last  command
  else (elif)
     command1
     command2
          .
          .
     last  command
fi
```

You could improve the **search** command so that it tells you when it cannot find a word, as well as when it can. The following display shows how the improved program looks:

```
$ cat search (Return)
echo Type the word and the filename.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

case...esac

The **case...esac** statement has a multiple choice format that allows you to choose one of several patterns and then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a closing parenthesis ( )) must be placed after the last character of each pattern. The command sequence for each pattern ends with two semicolons (;;). The **case** statement must end with **esac** (the letters of the word case reversed).

The **case** statement has the following format:

```
case word
in
   pattern1)
      command1
      command2
        .
        .
      last command
   ;;
   pattern2)
      command1
      command2
        .
        .
      last command
    ;;
    pattern3)
      command1
      command2
        .
        .
      last command
    ;;
    *)
      command 1
      command 2
        .
        .
     last command
   ;;"
esac
```

The **case** statement tries to match the *word* following **case** with the *pattern* in the first pattern section. If there is a match, the program executes the command lines after the first pattern and up to the corresponding **;;**.

If the *word* does not match the first pattern, the program proceeds to the second pattern. Once a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac.**

The asterisk (*) used as a pattern matches any *word,* allowing you to give a set of commands to be executed if no other pattern matches. To do this, it must be the last possible pattern in the **case** statement, so that the other patterns are checked first. This provides a useful way to detect erroneous or unexpected input.

A pattern can use the metacharacters *, ?, and [ ] for filename expansion.

The following *set.term* program contains an example of the **case...esac** statement. This program sets the **TERM** environment variable according to the type of terminal you are using.

In this example, the terminal is either a Teletype 4420, Teletype 5410, or Teletype 5420. *set.term* first checks to see whether the value of **term** is 4420. If it is, the program makes T4 the value of **TERM** and then goes to the command following **esac**. If the value of **term** is not 4420, the program checks for other values: 5410 and 5420. It executes the commands under the first pattern that it finds and then goes to the first command after the **esac** command.

The pattern *, meaning "everything else", is included at the end of the terminal patterns. It warns that you do not have a pattern for the terminal specified and allows you to exit the **case** statement. If you placed the * pattern first, the *set.term* program would never assign a value to **TERM**, as it would always match the first pattern *.

Following is the *set.term* program:

```
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
    in
        4420)
            TERM=T4
        ;;
        5410)
            TERM=T5
        ;;
        5420)
            TERM=T7
        ;;
        *)
        echo not a correct terminal type
        ;;
esac
export TERM
echo end of program
```

1.4.5 Testing Commands: test

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional statements. If the condition is true, the loop continues. If the condition is false, the loop ends and the next command is executed. Following are some of the options for the **test** command:

test –r *file*	True if the file exists and is readable
test –w *file*	True if the file exists and has write permission
test –x *file*	True if the file exists and is executable
test –s *file*	True if the file exists and has at least one character
test *var1* **–eq** *var2*	True if *var1* equals *var2*

test *var1* **–ne** *var2*   True if *var1* does not equal *var*

You might want to create a shell program to move your executable files in the current directory to your *bin* directory. You can use the **test –x** command to select the executable files. The following display shows the **for** statement that occurs in the *mv.file* program.

```
$ cat mv.file (Return)
echo type the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

The following program, called *mv.ex*, includes an **if test –x** statement in the **do...done** loop to move executable files only.

```
$ cat mv.ex (Return)
echo type the directory path
read path
for file
  do
    if test -x $file
       then
         mv $file $path/$file
    fi
  done
$
```

The directory path is the path from the current directory to the *bin* directory. However, if you use the value for the **HOME** environment variable, you will not need to type in the path each time. **$HOME** gives the path to the login directory. *$HOME/bin* gives the path to your *bin* directory.

In the following example, *mv.ex* does not prompt you to type in the directory name, and, therefore does not read the **PATH** variable:

```
$ cat mv.ex [Return]
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
$
```

Test the command, using all files in the current directory (indicated by the asterisk in the following example). The command lines shown in the following example execute the command from the current directory, then change to the *bin* directory and list the files in that directory. All executable files should be there.

```
$ mv.ex * [Return]
$ cd; cd bin; ls [Return]
list_of_executable_files
$
```

1.4.6 Terminating Programs: exit

A shell program normally terminates when the last command in the file is executed. However, you can use the **exit** command to terminate a program at another point. You can also use the **exit** command to issue return codes for a shell program.

The **exit** command has this format:

exit *n*

The **exit** command causes the shell program to terminate immediately. It is frequently used as part of an **if...then** loop. It tells the shell to exit the program if a certain condition is met (or not met).

The *n* represents the return code you want the shell to issue. If you don't specify a return code, the exit status is that of the last command executed before the **exit** command.

1.4.7 Unconditional Statements: break and continue

The **break** command unconditionally stops the execution of any loop in which it is encountered and goes to the next command after the **done**, **fi**, or **esac** statement. If there are no commands after that statement, the program ends.

In the *set.term* program, you could use the **break** command instead of **echo** to leave the program, as shown in the following example.

```
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
      in
         4420)
         TERM=T4
      ;;
      5410)
         TERM=T5
      ;;
      5420)
         TERM=T7
      ;;
      *)
         break
      ;;
esac
export TERM
echo end of program
```

The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

1.5 Debugging Programs

At times you might need to debug a program to find and correct errors. There are two options to the **sh** command that can help you debug a program:

sh –v *program*	Prints the shell input lines as they are read by the system
sh –x *program*	Prints commands and their arguments as they are executed

The following shell program, called **bug**, contains an error.

```
$ cat bug [Return]
today=`date`
echo enter person
read person
mail $1
$person
When you log off come into my office please.
$today.
MLH
$
```

The output from the **date** command is the value of the **today** variable.

The mail message sent to Tom ($1) at login **tommy** ($2) should read as follows:

```
$ mail [Return]
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1986
MLH
?
```

If you try to execute **bug**, you will have to press [Break] or [Delete] to end the program.

To debug this program, use **sh –v** to execute the program. This command prints the lines of the file as they are read by the system.

```
$ sh –v bug tom [Return]
today=`date`
echo enter person
enter person
read person
tom
mail $1
```

Notice that the output stops on the **mail** command, as there is a problem with **mail**. You must use the here document to redirect input into **mail**.

When you use **sh –x** to execute the program, the commands and their arguments print as they are read by the system.

```
$ sh –x bug tom [Return]
+date
today=Thu  Apr 10  11:07:23  CST  1986
+ echo enter person
enter person
+ read person
tom
+ mail tom
$
```

Once again, the program stops at the **mail** command. Notice that the shell has made substitutions for the variables.

The corrected **bug** program is as follows:

```
$ cat bug [Return]
today=`date`
echo enter person
read person
mail $1 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

1.6 Programming in the C Shell

Shell scripts run from the C shell are passed to the Bourne shell for execution unless explicitly told not to. When running the C shell, a pound sign (#) at the beginning of a shell script causes the script to be run by the C shell instead of the Bourne shell (unless the # is the start of a line specifying an alternative shell such as #!bin/sh). The following shell script invoked from the C shell would be run by the C-shell:

```
# a simple shell script
pwd
ls
echo This is a C-shell program.
```

The C-shell provides command-line substitution and programming statements like the Bourne shell. The following paragraphs describe some of the C-shell features that differ from the Bourne shell. For more information about the C shell, refer to "Using the Shell" in the *User's Guide* and to *csh*(1) in the *Reference Manual.* For information about job control in the C shell, refer to "Managing Processes" in the *User's Guide.*

1.6.1 Positional Parameters

Command-line arguments are assigned to the C-shell variable **argv**. Like the Bourne-shell positional parameters, brackets are used to indicate which argument is to be accessed. A range of arguments can be indicated with a hyphen. The following example illustrates the use of **argv**.

```
# command-line substitution
# accessing command-line arguments
echo There are $#argv arguments on the command line
echo The first argument is $argv[1]
echo The second through fifth arguments are $argv[2-5]
echo The last argument is $argv[$#argv]
```

The C-shell accepts the notation of the Bourne shell (**$1**, **$2**, and so forth), therefore, **$1** is the same as **$argv**[1] and **$*** is the same as **$argv**[*].

Other parameters that may be used for command-line substitution are as follows:

$0	The name of the shell script
$$	The process ID of the parent shell
$?name	Equals 1 if the variable *name* is set; otherwise equals 0
$status	The status returned by the last command

1.6.2 Looping

The C-shell includes three loop structures; the **foreach** loop, the **while** loop, and the **repeat** loop.

The foreach Loop

The **foreach** loop structure is as follows:

```
foreach name (wordlist)
    command1
    command2
       .
       .
    last command
end
```

The variable *name* takes each value one at a time from *wordlist* and executes the commands until the end of the *wordlist* is reached.

In addition to providing an explicit *wordlist*, you can also provide values to *name* by doing any of the following:

1. Taking values from the output of a command by using the backquote command-substitution mechanism, for example:

   ```
   foreach color ('cat hues')
       echo I love a subtle shade of $color
   end
   ```

2. Taking values from the shell script command line: **foreach** *word* or **foreach** *word ($argv)* or **foreach** *word (argv[2-]).* These first two forms loop through the entire command-line argument list. The third form skips the first command line argument and loops through the argument list from the second through final arguments.

3. Taking filenames from a directory as values, for example: **foreach** *file (*)* or **foreach** *file (*.f)*. The first form processes all files in the current directory; the second processes all files ending in *.f* in the current directory.
4. Taking values from a shell variable; **foreach** *name* (**$PATH**). This form sets the *name* variable to each directory name in the **PATH** shell variable.

Output from the **foreach** loop cannot be redirected. (Output from a **for** loop in Bourne shell may be redirected.) However, you can redirect output within a **foreach** loop, as follows:

```
foreach color ('cat hues')
    echo I love a subtle shade of $color >> colorlikes
end
```

The while Loop

The C-shell **while** loop tests an expression at the beginning of each loop and continues executing the sequence of commands as long as the expression is true. The C-shell **while** loop has this form:

```
while expression
   command1
   command2
     .
     .
   last command
end
```

In the following example, looping continues until *color* is *green*. The **$<** notation indicates input coming from the command line.

```
# grass
# Usage: grass
echo What color is grass\?
set color=$<
```

```
while ( "$color" != "green" )
    echo Nope\! Guess again.
    set color=$<
end
```

The repeat Loop

In the **repeat** loop, *command* is executed *count* times. The *command* must be simple: it can have arguments and I/O redirection, but cannot contain pipes or lists of commands. The C-shell **repeat** loop has this form:

repeat *count command*

A simple example of a **repeat** loop follows:

```
% repeat 5 echo Happy Birthday
Happy Birthday
Happy Birthday
Happy Birthday
Happy Birthday
Happy Birthday
```

1.6.3 Conditional Statements

C-shell programming allows conditional statements from a simple **if** statement to the complex **if...then...else if...endif** statement. These statements test for conditions to be met, and then execute commands depending on whether conditions are met or not met.

if

A simple **if** statement has this format:

if (*expression*) *command*

The entire statement is one line; if *expression* is true, the command is executed; otherwise, it is not. Unlike the Bourne shell **if** statement, *expression* is enclosed in parentheses, and **then** is placed differently.

Redirection is allowed in *command,* but pipes and multiple commands are not allowed. The following is an example of a simple **if** command:

```
if ( $a + $b == $c) echo You are correct!
```

if...then...endif

This statement is used to execute multiple commands when a condition is met. The **if...then...endif** statement has this format:

> **if** (*expression*) **then**
>   *command1*
>   *command2*
>   .
>   .
>   *last command*
> **endif**

In the following example, when *time* is less than or equal to 10, two messages are displayed:

```
@ time = 5
echo $time minutes
if ($time <= 10 ) then
    echo The hamburger is done.
    echo Let\'s eat
k
endif
```

if...then...elseif...endif

This statement is used to test multiple conditions and then to execute multiple commands, depending on which condition is met. This is similar to the Bourne shell **case** statement.

The format is as follows:

```
if ( expression ) then
     command1
     command2
       .
       .
     last command
else if ( expression ) then
     command1
     command2
       .
       .
     last command
else
     command1
     command2
       .
       .
     last command
endif
```

The following example tests of doneness in hamburgers:

```
# hamburger
# Usage: hamburger number
echo How long has it cooked\?
@ time=$<
echo It´s cooked $time minutes.
if ( $time < 5 ) then
   echo This hamburger is rare.
        echo Let\'s eat
else if ( $time == 5 ) then
   echo This hamburger is medium.
        echo Let\'s eat
else if ( $time > 5 && $time < 10 ) then
   echo This hamburger is well done.
        echo Let\'s eat
```

```
else
    echo This hamburger is burnt.
    echo Let\'s go get pizza
endif
```

switch...breaksw...endsw

The **switch** structure is similar to the Bourne shell **case** structure and is similar in form to the C language **switch**. Values are compared with a list of choices (cases). When a match is found, all commands until the break switch (**breaksw**) keyword are executed. If no match is founds, the commands after the **default** label are executed.

The **switch** structure has the following form:

```
switch (string)
case string1:
      command1
      command2
        .
        .
      last command
      breaksw
case string2:
      command1
      command2
        .
        .
      last command
      breaksw
case string3:
      command1
      command2
        .
        .
      last command
      breaksw
default:
      command1
      command2
        .
        .
      last command
      breaksw
endsw
```

Compare the following example which is similar to the previous example:

```
# hamburger
# Usage: hamburger number
echo How long has it cooked\?
@ time=$<
echo It\'s cooked $time minutes.
switch ($time)
case 1:
    echo This hamburger is rare.
         echo Let\'s eat;
    breaksw
case 2:
    echo This hamburger is rare.
         echo Let\'s eat;
    breaksw
case 3:
    echo This hamburger is rare.
         echo Let\'s eat;
    breaksw
case 4:
    echo This hamburger is rare.
         echo Let\'s eat;
    breaksw
case 5:
    echo This hamburger is medium.
         echo Let\'s eat; breaksw
case 6:
    echo This hamburger is well done.
         echo Let\'s eat;
    breaksw
case 7:
    echo This hamburger is well done.
         echo Let\'s eat;
    breaksw
case 8:
    echo This hamburger is well done.
         echo Let\'s eat;
    breaksw
case 9:
    echo This hamburger is well done.
         echo Let\'s eat;
    breaksw
```

```
case 10:
    echo This hamburger is burnt.
    echo Let\'s go get pizza;
    breaksw
default
    breaksw
endsw
```

1.6.4 Unconditional Statements: break, breaksw, and continue

break and **continue** statements can be used in **foreach** and **while** loops. As in the Bourne shell, **break** exits a loop; **continue** skips to the next cycle of a loop. **breaksw** must be used with the **switch** statement.

1.6.5 Variable Modifiers

The C shell recognizes several suffixes that can be used to modify the interpretation of a regular shell variable. The following suffixes can be used for working with variables that have been set to filenames:

h	Remove a trailing pathname component, leaving the head
t	Remove all leading pathname components, leaving the tail
r	Remove a trailing *.xxx* extensions, leaving the root
e	Remove all but the trailing extension *.xxx*

These suffixes are used by appending a colon (:) and the suffix to a variable name, as shown in the following examples:

```
% set file = /usr/rainbow/colors/violet.c
% echo $file
/usr/rainbow/colors/violet.c
% echo $file:h
/usr/rainbow/colors/
% echo $file:t
violet.c
% echo $file:r
/usr/rainbow/colors/violet
% echo $file:e
c
```

1.6.6 Expressions

C-shell expressions use operands and operators. The operators are the same as in the C language with the addition of =~ (EQUALS) and !~ (DOESN'T EQUAL). The operators are listed in order of increasing precedence in Table 1-1.

Table 1-1
C-Shell Operators

|| &&	Logical OR and logical AND
| ^ &	Bitwise OR, EXCLUSIVE OR, and AND
== !=	EQUALS and DOESN'T EQUAL
=~ !~	EQUALS and DOESN'T EQUAL a string pattern
< <=	LESS THAN and LESS THAN OR EQUAL TO
> >=	GREATER THAN and GREATER THAN OR EQUAL TO
<< >>	Bitwise LEFT SHIFT and RIGHT SHIFT
+ -	PLUS and MINUS
* % /	MULTIPLY, MODULUS, and DIVIDE
!	Bitwise ONE'S COMPLEMENT
( )	Grouping

The operators are usually applied to character strings, to strings representing numbers, or to other expressions. The value of an expression can be assigned to a shell variable with the @ command. The @ command works much like the Bourne shell **set** command, except that it assigns the *value* of an expression, rather than a word, to a variable. In both cases, the variable is stored as a string; however, the @ command allows expressions with operators on the side of the assignment. In the following example, the **set** and @ commands are equivalent:

```
% set month = 11
% @ year = 1989
% echo $month $year
11 1989
```

In this example using the @ command, the value of an expression is assigned to the variable *total*:

```
% @ total = 1 + 2 + 3
% echo $total
6
```

You can use the C language increment and decrement operators with shell variables, and the @ command to increase or decrease the value of a variable by 1, for example:

```
% @ total = 100
% echo $total
100
% @ total++
% echo $total
101
% @ total--
% echo $total
100
```

Remember that numbers are stored as strings. The @ command instructs the shell to find the numerical value corresponding to the string, do the indicated operations, and convert the result back to a string variable.

Relational Expressions

Relational expressions have the value 1 if true and the value 0 if false, as in C. The *expression* part of **while** and **if** statements is often a relational expression. Most comparisons are numeric, however equality and inequality comparisons are string comparisons. Therefore, you can use **if** statements that start like these:

```
if ( $argv[1] == stop ) ...
if ( $argv[1] != sam ) ...
```

The following script uses the GREATER THAN operator along with numerical expressions:

```
# countdown
# Usage:  countdown number
#
@ start = $argv[1]     # number is the first argument
while ( $start > 0 )   # While start is greater than 0,
   echo $start\! ... # print start! ...
   @ start--           # Decrement start by 1
end                    # When start is 0,
echo Blast Off\!      # print Blast Off!
```

Strings can be numbers; however, note that strings with the same numeric value, for example 123 and 00123, are unequal because they stored as characters.

The C-shell EQUALS (=~) and DOESN'T EQUAL (!~) operators allow the right side of a relational expressions to be a pattern using the *, ?, and [] shell metacharacters. The following example checks for C programs in a the current directory, and copies them to the directory, *backup* adding the file extension *.back*. The *r* modifier is specified to remove the *.c* file extension.

```
# FORTRAN source file backup
foreach file (*)
     if ($file =~ *.f) cp $file backup/${file:r}.back
end
```

Testing Files

C shell allows you to inquire about files in C-shell scripts with a query of the form

–l name

–l is a letter from Table 1-X; *name* is the name of a file or directory. The following file attributes can be tested for in C-shell scripts:

Letter	Tests for
-r	Read access
-w	Write access
-x	Execute access
-e	Existence
-o	Ownership
-z	Zero size
-f	Plain file
-d	Directory

For example, the following shell script, *filecheck*, checks for owner read, write and execute permission on the file named on the command line. It also checks file ownership.

```
# check files for owner permissions and ownership
# Usage: filecheck filename
if (-r $argv[1] ) echo $argv[1] is readable.
if (-w $argv[1] ) echo $argv[1] is writable.
if (-x $argv[1] ) echo $argv[1] is executable.
if (-o $argv[1] ) then
       echo $argv[1] is owned by me
else
       echo $argv[1] is owned by someone else
endif
```

Testing Commands: test

The C-shell **for** and **while** statements test for truth or falsity of an expression. (Bourne shell tests for success or failure of a command with the **test** command.) You can test a C-shell command by enclosing it in braces ({}). You must put space between the braces and their contents. The following example checks to see if the **grep** command can find the string **abcdef** in any file whose name begins with alpha:

```
# test grep command
# Usage: commandtest
if ( { grep -s abcdef alpha* } ) then
    echo "Yep, I found abcdef"
else
    echo "Nope, can't find abcdef"
endif
```

1.7 Programming in the Korn Shell

The Korn shell supports all of the Bourne shell statements and provides some additional programming features. Like the C shell, the Korn shell has tilde substitution, built-in command testing and expressions, and command history. In addition, the Korn shell provides a mechanism for using functions. The following paragraphs describe some of the Korn shell features that differ from the Bourne and C shells. For more information about the Korn shell, refer to "Using the Shell" in the *User's Guide* and **ksh**(1) in the *Reference Manual*. For information about job control in the Korn shell, refer to "Managing Processes" in the *User's Guide*.

1.7.1 Looping

Besides the looping structures provided in the Bourne shell, the Korn shell includes the *select* loop.

The select Loop

The format for the **select** loop is as follows:

```
select variable in word1 word2 ...
do
    command1
    command2
      .
      .
    last command
done
```

select prints each word preceded by its relative number in the list (1 through *n*) to standard error. If **in** is omitted, the positional parameters are used as words. The Korn shell prompt (PS3) is printed and a line is read from standard input and assigned to the Korn-shell environment variable **REPLY**. If the line begins with one of the numbers 1 through *n*, *variable* is set to the corresponding *word* from the list. If the line begins with anything else, *variable* is set to null. In either case, the commands are executed and looping continues until an end-of-file is encountered, or a **break**, **exit**, or **return** is executed from inside the loop. The following example, containing a **case** statement, shows the use of **select** for a user menu:

```
PS3="Please select one of the above: "

select choice in Create Add Delete Read Write Exit
do
   case "$choice"
   in
   Create) ...;;
   Add   ) ...;;
   Delete) ...;;
   Read  ) ...;;
   Write ) ...;;
   Exit  ) exit 0;;
        *     ) echo "Bad Choice";;
```

```
    esac
  done
```

1.7.2 Functions

One of the most important additions to shell programming is the Korn shell's function capability. Functions are declared with the following format:

```
function name { list ;}
name () { list ;}
```

Functions can be called from a shell program, or from functions within a shell program—Korn shell functions can be recursive. Variables in functions may be global or local. A variable declared in the main shell script can be used and changed within a function. Conversely, variables within a function can be used by the main shell script, or enclosing function. Variables declared inside a function with **typeset** are not accessible outside the function, and are local or private to that function. If a global shell variable with the same name as a local function variable exists, the value of the shell variable is saved and restored when the function exits. Variables declared inside a function with **typeset -x** can be exported outside the function. Variables not declared with **typeset** are automatically exported.

References can be made inside a function to the arguments of the function call. The arguments are positional parameters and are referred to the same way as the main shell script parameters are referred to—**$1**, **$2**, and so forth. The positional parameters of a function are all local variables.

When a **return** is executed in a function, any exit trap set inside the function is executed before the return to the calling program. In the Bourne shell, an exit trap is not executed until the main program exits.

Once a function is defined, it can be used like any valid command and called in any Korn-shell script by including function declarations in the *.profile* or *.kshrc* file. The advantage of this mechanism over the C and Bourne shells is that the function runs in the same shell as the main shell script (or enclosing function), thus reducing the overhead of starting up another shell.

The following simple function prints the working directory:

```
function printdir {
echo $PWD
}
```

This shell script calls the **printdir** function:

```
cd $1
printdir
```

1.7.3 Built-in Integer Arithmetic

The Korn shell allows some data typing. The following paragraphs describe built-in commands for doing integer arithmetic.

The let command

The **let** command has the form:

let *expressions*

expressions are arithmetic expressions using shell variables that contain numeric values and the following operators listed in order of increasing precedence:

Table 1-2
Korn-Shell Operators

=	Assignment
== !=	EQUALS and DOESN'T EQUAL
> >=	GREATER THAN and GREATER THAN OR EQUAL TO
< <=	LESS THAN and LESS THAN OR EQUAL TO
+ -	PLUS and MINUS
* % /	MULTIPLY, MODULUS, and DIVIDE
!	Logical Negation
-	Unary MINUS

The **let** command works much like the Bourne shell **set** command, except that it assigns the *value* of an expression, rather than a word, to a variable. The **let** command, unlike the **set** command, allows expressions with operators on the right side of the assignment. Expressions containing spaces must be enclosed in quotes; white space separates expressions. The following example shows the **let** command followed by an expression:

```
read index
while let "index = index +1"
do
    echo "index : $index"
done
```

The test Command

The **test** command, which checks to see if certain conditions are true, recognizes any of the integer comparison operators in expressions:

Table 1-3
Integer Comparison Operators

–eq	equals
–ge	greater than or equal to
–gt	greater than
–le	less than or equal to
–lt	less than
–ne	not equal to

The **test** command also recognizes the following file operators:

Table 1-4
File Comparison Operators

–L *file*	True if *file* is a symbolic link
file1 **–nt** *file2*	True if *file1* is newer than *file2*
file1 **–ot** *file2*	True if *file1* is older than *file2*
file1 **–ef** *file2*	True if *file1* and *file2* are linked

File testing can also be done in Korn shell using double brackets as follows:

[[*expression* [*attribute_operator expression*]]]

The double brackets in bold indicate that the expression or expressions within are to be tested. The following file attributes can be tested for in Korn-shell scripts. A value of true is returned if the file exists and if the attribute is true:

Letter	Tests for
–r	Read access
–w	Write access
–x	Execute access
–f	Plain file
–d	Directory
–c	Character special file
–b	Block special file
–p	File is named pipe
–u	Set-user-id bit is set
–g	Set-group-ed bit is set
–k	Sticky bit is set
–s	Size is greater than zero
–L	Symbolic link
–O	Owner is the effective user id
–G	Group is the effective group id
–S	Special file of type socket

The following example compares the two strings to see if *dog* is greater than *cat*, and if the working directory has read and write permission. If the test is true, the value of the test expression is returned.

```
$ [[ dog > cat && ( –r $PWD && -w $PWD) ]] && print succeed
succeed
```

The typeset Command

The **typeset** command is used to change the attributes of shell variables. A complete list of the **typeset** command options are on the **ksh**(1) manual page.

In the Bourne shell and by default in the Korn shell, variables are strings. The **typeset** command can be used to declare shell variables to be integers, as follows:

typeset –i *variables*

If a variable is first declared to be an integer as in the following example, it can then be used without requiring the let command. Because Korn shell automatically aliases **integer** to **typeset –i**, the three examples following are equivalent:

```
typeset -i integer1
integer1="integer1 + 1"

let "integer1 = integer1 + 1"

integer integer1
integer1="integer1 + 1"
```

The **typeset** command options can be specified as separate arguments, such as **typeset –u –x**.

1.7.4 Manipulating Strings

In addition to the positional and special parameters used in the Bourne shell, the Korn shell provides the following parameter substitution operations:

${*parameter* } The value of *parameter* is substituted. Braces are required when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name; when a named parameter has a subscript; or for positional parameters. If *parameter* is an asterisk (*) or an at sign (@), all positional parameters, starting with $1 are substituted. If an array is used for *parameter* and the subscript * or @ is used, the value for each of the elements is substituted. The *pattern* specified may contain the special file substitution characters *, ?, and [...].

${ *#parameter* }
If *parameter* is an * or @, the number of positional parameters is substituted; otherwise, the length of *parameter* is substituted. This form is equivalent to $#.

${ *#array*[*] } The number of elements in *array* is substituted.

${ *parameter:-word*}
: If *parameter* is set and non-null, or user-specified as null, substitute the expanded value of *word*, otherwise expand to the value of *parameter*.

${ *parameter:=word*}
: If *parameter* is not set or is null, set *parameter* to *word*. Positional parameters cannot be assigned in this manner.

${ *parameter:?word*}
: If *parameter* is set and non-null, substitute its value, otherwise print *word* and exit from the shell. If *word* is omitted, a standard message prints.

${ *parameter:+word*}
: If *parameter* is set and non-null, substitute *word*, otherwise substitute nothing.

${*parameter#pattern* } or ${*parameter##pattern*}
: Substitute the value of *parameter* with **pattern** deleted from the left side. In the first form, the smallest part of the contents of *parameter* matching *pattern* is deleted; in the second form, the largest matching pattern is deleted.

${ *parameter%pattern* } or ${ *parameter%%pattern*
: }" Substitute the value of *parameter* with **pattern** deleted from the right side. In the first form, the smallest part of the contents of *parameter* matching *pattern* is deleted; in the second form, the largest matching pattern is deleted. The following example tests to see if object files (ending with *.o*) files exist for each C source file (ending with *.c*):

```
for file in *.c
do
    file=${file%.c}
    if [ ! -f "file.o" ]
    then
        echo "$file.o doesn't exist"
    fi
done
```

These are examples of parameter substitution using the forms above:

```
$ param=bigword
$ echo ${param%d}
bigwor
$ echo ${param%w*d}
big
$ echo ${param%%w*d}
big
$ echo ${param#?i}
gword
$ echo ${param##*o}
rd
$ echo ${param#bio}
bigword
```

The typeset Command

The **typeset** command can be used to create shell variables of fixed length with the following forms:

> **typeset –L***n variables*
> **typeset –R***n variables*

The **–L** and **–R** options define *variables* to be left- or right-justified, respectively, and *n* specifies the length in characters. If *n* is not specified, the length is taken from the length of the values assigned to the variables. In the following example, long lines are truncated to force columns to line up:

```
typeset -L25 first
IFS=' ^'
while read line
do
    set $line
    first=$1
eval echo "
done
```

If you enter "Bob", it is echoed back:

```
Bob
Bob
```

If you enter "What a nice day it is outside", the line is truncated at the 25th character and echoed back as follows:

```
What a nice day it is outside
What a nice day it is out
```

1.7.5 One-Dimensional Arrays

The Korn shell allows one-dimensional arrays up to 512 elements. Array indexing starts at zero. Arrays are not declared, and array elements are accessed by subscripts. Subscripts are enclosed in brackets. As asterisk enclosed in brackets refers to all elements of an array. The following example assigns values to the first to elements of array *animal*:

```
$ animal[0]='dog'  # first array element has a value of dog
$ animal[1]='cat'  # second array element has a value of cat
```

In the following example, the second element is echoed. When an array element is substituted, it must be enclosed in braces.

```
$ echo ${animal[1]}
cat
```

In this example, no subscript is specified, so the first element is assumed:

```
$ echo $animal
dog
```

You can print all the elements in an array with the following parameter substitution statement:

```
$ animal[0]='dog'
$ animal[1]='cat'
$ echo ${animal[*]}
dog cat
$
```

You can also print or substitute the number of elements in the array *animal* by using the parameter substitution statement **%{#animal[*]}**, as follows:

```
$ animal[0]='dog'
$ animal[1]='cat'
$ echo ${#animal[*]}
2
$
```

1.7.6 Miscellaneous Commands

The read Command

The **read** command is like the Bourne shell **read** command, except that some options have been added that provide for greater flexibility. The **read** command options are described on the **ksh**(1) manual page.

The print Command

The **print** command is similar to the Bourne shell **echo** command, with more options including what terminal mode to use and what file descriptor to write to. The **print** command options are described on the **ksh**(1) manual page.

The times Command

The **times** command prints the accumulated system times used by the shell and processes run from the shell. The Bourne shell prints only the system time used by the shell.

The trap Command

The Korn shell **trap** command allows the use of signal names as well as signal numbers. The format of the trap command is as follows:

```
trap [ arg ] [ signal ]
```

The **trap** command with no arguments prints a list of commands associated with each signal number.

Korn shell provides a *signal* called ERR that is used whenever a command has a nonzero exit status and, if *arg* is present, *arg* is executed. This trap is not inherited by functions.

The EXIT trap is local to functions; the specified commands are executed when the function exits, not when the shell program exits.

Index

2. nawk

2. nawk

Chapter 2
nawk

Figures

Tables

Chapter 2 ***nawk***

2.1 Introduction

NOTE

This chapter describes a new version of **awk** *(***nawk***)* The previous version of **awk** *is described in Chapter 3,* **awk**.

Suppose you want to tabulate some survey results stored in a file, print various reports summarizing these results, generate form letters, reformat a data file for one application package to use with another package, or count the occurrences of a string in a file. **nawk** is a programming language that makes it easy to handle these and many other tasks of information retrieval and data processing. The name **nawk** is an acronym constructed from the initials of its developers; it denotes the language and also the operating system command you use to run an **nawk** program.

nawk is an easy language to learn. It automatically does quite a few things that you have to program for yourself in other languages. As a result, many useful **nawk** programs are only one or two lines long. Because **nawk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **nawk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **nawk** and is intended to make it easy for you to start writing and running your own **nawk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **nawk** user, there's a summary of the language at the end of the chapter.

You should be familiar with the operating system and shell programming to use this chapter. Some knowledge of the C programming language is beneficial, because many constructs found in **nawk** are also found in C.

2.2 Basic nawk

This section provides enough information to allow you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

2.2.1 Program Structure

The basic operation of **nawk**(1) is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **nawk** program is a sequence of pattern-action statements, as Figure 2-1 shows.

Structure:

```
pattern    { action }
pattern    { action }
...
```

Example:

```
$1 == "address"  { print $2, $3 }
```

Figure 2-1. nawk Program Structure and Example..

The example in the figure is a typical **nawk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is *address*. In general, **nawk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in

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

the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

2.2.2 Running nawk

There are two ways to run an **nawk** program, shown in the following syntax block:

nawk -F*r* ['*program*' | **-f** *nawk-program-file*] [*file* ...] [-]

r	Any character or regular expression to be used as a field separator. Blanks and tabs are the default field separators.
program	The **nawk** program itself entered on the command line and enclosed in single quotes.
nawk-program-file	A file containing an **nawk** program.
file	An input file containing field-separated records for processing by **nawk**.
–	Standard input. If no input *file*s are specified on the command line, or if a hyphen is specified, **nawk** looks to *stdin* for input. If input *file*'s and a hyphen are specified on the command line, **nawk** uses both the files and *stdin* for input in the order they appear on the command line.

Running a Program from a Command Line

If the program is short, it is often easiest to make the program the first argument on the command line. The program must be enclosed in single quotes in order for the shell to accept the entire string as the first argument to **nawk**. In the following example, the **nawk** program (between the single quotes) matches the pattern *x* in *file1* and prints the lines that contain a match:

```
  nawk ' /x/ {print} ' file1
```

If no input *file* is specified, **nawk** expects input from standard input. You can also specify that input comes from *stdin* by using a hyphen (–) as one of the input files. In this example, the **nawk** program (between the single quotes) looks for input from *file1* and from *stdin*. Input from *file1* is processed first, and then input from *stdin*:

```
  nawk ' /x/ {print} ' file1 -
```

Running a Program in a File

If your **nawk** program is long or you want to save it for future use, you can store the program in a separate file (*nawkprog*, for example). Specifying the **–f** option on the command line tells **nawk** to fetch it, as follows:

```
  nawk -f nawkprog file1
```
where *file1* is the input file that may include *stdin* as is shown previously.

These alternative ways of presenting your **nawk** program for processing are illustrated by the following examples. The following **nawk** command example prints *hello, world* to standard output when given to the shell:

```
  nawk ' BEGIN {print "hello, world" exit} '
```

This **nawk** program could be put in a file name *nawkprog*:

```
  BEGIN {print "hello, world" exit}
```

The following command, given to the shell, would have the same effect as the previous procedure:

```
  nawk -f nawkprog
```

2.2.3 Fields

nawk normally reads its input one line, or record, at a time and splits each record into fields. By default, a record is a sequence of characters ending with a newline and a field is a string of nonblank, nontab characters.

The file *countries*, shown below, contains information about the ten largest countries in the world. This file is used as input for many of the **nawk** examples in this chapter. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is found. Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia. The white space between fields is a tab in the original input; a single blank separates *North* and *South* from *America*.

```
USSR        8650      262      Asia
Canada      3852       24      North America
China       3692      866      Asia
USA         3615      219      North America
Brazil      3286      116      South America
Australia   2968       14      Australia
India       1269      637      Asia
Argentina   1072       26      South America
Sudan        968       19      Africa
Algeria      920       18      Africa
```

Figure 2-2. The Sample Input File ***countries*****.**

This file is typical of the kind of data **nawk** is good at processing—a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. By default, fields are separated by one or more blanks or tabs; therefore, the first record of *countries* has four fields. You can set the field separator to the tab character only, so that each line has four fields, matching the meaning of the data. For the time being, use the default field separators: fields separated by one or more blanks or tabs. The first field within a line is **$1**, the second is **$2**, and so forth. The entire record is referred to as **$0**.

2.2.4 Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **nawk** program consisting of a single **print** statement:

```
{ print }
```

So the following command line prints each line of *countries*, copying the file to the standard output.

```
nawk '{ print }' countries
```

The **print** statement can also be used to print parts of a record; for example, the following program prints the first and third fields of each record:

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

Thus, the following command line produces as output a sequence of lines:

```
nawk '{ print $1, $3 }' countries
```

The output is as follows:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which by default is a single blank. Each line printed is terminated by the output record separator, which by default is a newline.

NOTE

In the remainder of this chapter, **nawk** *programs only are shown, not the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the* **nawk** *command, or by putting it in a file and invoking* **nawk** *with the* **–f** *flag, as discussed in "Running nawk" earlier in this chapter. In examples, if no input is mentioned, the input is assumed to be the file* countries.

2.2.5 Formatted Printing

For more carefully formatted output, **nawk** provides a C-like **printf** statement of the form:

printf *format, expr1, expr2, . . ., exprn*

which prints *expr1, expr2*, and so forth, according to the specification in the string *format*. For example, the **nawk** program

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field (**$1**) as a string of 10 characters (right justified), then a space, then the third field (**$3**) as a decimal number in a six-character field, then a newline (**\n**). With input from the file *countries*, this program prints the following aligned table:

```
      USSR    262
    Canada     24
     China    866
       USA    219
    Brazil    116
 Australia     14
     India    637
 Argentina     26
     Sudan     19
   Algeria     18
```

With **printf**, no output separators or newlines are produced automatically; you must create them yourself by using **\n** in the format specification. Refer to the "The **printf** Statement", later in this chapter, for a full description of **printf**.

2.2.6 Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **nawk** has three kinds of patterns:

- Relational expressions
- Regular expressions
- Special Patterns

Relational expressions are patterns that make comparisons. For example, the operator == tests for equality. To print the lines for which the fourth field equals the string *Asia*, you can use a program consisting of the following single pattern:

```
$4 == "Asia"
```

With the file *countries* as input, this program produces the following output:

```
USSR     8650     262      Asia
China    3692     866      Asia
India    1269     637      Asia
```

The complete set of comparisons is >, >=, <, <=, == (equal to), and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose you want to print only countries with a population greater than 100 million. The following program is all that is needed:

```
$3 > 100
```

Remember that the third field in the file *countries* is the population in millions. It prints all lines in which the third field exceeds 100.

Regular expressions are patterns that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters *US* anywhere; with the file *countries* as input, it prints this:

```
USSR     8650     262     Asia
USA      3615     219     North America
```

More information is found in "Regular Expressions" later in this chapter.

Two special patterns, **BEGIN** and **END**, match before the first record has been read and after the last record has been processed. This program uses **BEGIN** to print a title:

```
BEGIN    { print "Countries of Asia:" }
/Asia/   { print "      ", $1 }
```

The output is:

```
Countries of Asia:
      USSR
      China
      India
```

Refer to "Patterns," later in this chapter, for more information about **BEGIN** and **END**.

2.2.7 Simple Actions

We have already seen the simplest action of an **nawk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **nawk** counts the number of records read and the number of fields within the current record; you can use these counts in your **nawk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. So the following program prints the number of each line and how many fields it has:

```
{ print NR, NF }
```

This program, however, prints each record preceded by its record number.

```
{ print NR, $0 }
```

User-Defined Variables

Besides providing built-in variables like **NF** and **NR**, **nawk** lets you define your own variables, which you can use for storing data, doing arithmetic, and performing other tasks. The following example computes the total population and the average population represented by the data in the file *countries*:

```
    { sum = sum + $3 }
  END { print "Total population is", sum, "million"
    print "Average population of", NR, "countries is", sum/NR }
```

nawk initializes **sum** to zero before it is used. The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

nawk has built-in functions that handle common arithmetic and string operations. For example, there is a math function that computes square roots. There is also a string function that substitutes one string for another. **nawk** also lets you define your own functions. Functions are described in detail in "Actions", later in this chapter.

2.2.8 Some Useful One-Liners

Although **nawk** can be used to write large programs of some complexity, many programs are not much more complicated than what you've seen so far. Here is a collection of other short programs that you may find useful and instructive. Any new constructs are explained later in this chapter.

Print last field of each input line:
```
{ print $NF }
```

Print 10th input line:
```
NR == 10
```

Print last input line:
```
     { line = $0}
END { print line }
```

Print input lines that don't have four fields:
```
NF != 4     { print $0, "does not have 4 fields" }
```

Print input lines with more than four fields:
```
NF > 4
```

Print input lines with last field more than 4:
```
$NF > 4
```

Print total number of input lines:
```
END { print NR }
```

Print total number of fields:

```
          { nf = nf + NF }
     END { print nf }
```

Print total number of input characters (adding **NR** includes the number of newlines in the total):

```
          { nc = nc + length($0) }
     END { print nc + NR }
```

Print the total number of lines that contain the string *Asia*.
The statement **nlines++** has the same effect as **nlines = nlines + 1**:

```
     /Asia/ { nlines++ }
     END { print nlines }
```

2.2.9 Error Messages

If you make an error in an **nawk** program, you generally get an error message. For example, trying to run the program

```
  $3 < 200 { print ( $1 }
```

generates the following error messages:

```
  nawk: syntax error at source line 1
   context is
          $3 < 200 { print ( >>> $1 } <<<
  nawk: illegal statement at source line 1
           1 extra (
```

Some errors are detected while your program is running. For example, if you try to divide a number by zero, **nawk** stops processing and reports the input record number (**NR**) and the line number in the program.

2.3 Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

2.3.1 BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **nawk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once, before the **nawk** command starts to read its first input record. The pattern **END** matches the end of the input, after the last record has been processed.

The following **nawk** program uses **BEGIN** to set the field separator to tab (\t) and to put column headings on the output. The field separator is stored in a built-in variable called **FS** . Although **FS** can be reset at any time, the logical place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. Notice that a long line can be continued after a comma:

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
               "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
        area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file *countries* as input, this program produces the following:

```
   COUNTRY   AREA   POP   CONTINENT
      USSR   8650   262   Asia
    Canada   3852    24   North America
     China   3692   866   Asia
       USA   3615   219   North America
    Brazil   3286   116   South America
 Australia   2968    14   Australia
     India   1269   637   Asia
 Argentina   1072    26   South America
     Sudan    968    19   Africa
   Algeria    920    18   Africa

     TOTAL  30292  2201
```

2.3.2 Relational Expressions

An **nawk** pattern can be any expression involving comparisons between strings of characters or numbers. **nawk** has six relational operators and two regular expression matching operators, ~ (tilde) and !~, for making comparisons. Table 2-1 shows these operators and their meanings.

Table 2-1
nawk Relational Operators

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **nawk** can tell what is intended. Refer to "Number or String" later in this chapter.) Thus, the pattern *$3>100* selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters *S* through *Z*, namely:

```
USSR    8650    262 Asia
USA 3615    219 North America
Sudan   968 19  Africa
```

In the absence of any other information, **nawk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file *countries* as input, prints the single line for which this test succeeds:

```
Australia   2968    14    Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

2.3.3 Regular Expressions

nawk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions and are like those in **egrep**(1) and **lex**(1). The simplest regular expression is a string of characters enclosed in slashes, for example:

```
/Asia/
```

This program prints all input records that contain the substring *Asia*; if a record contains *Asia* as part of a larger string like *Asian* or *Pan-Asiatic*, it is also printed. In general, if *re* is a regular expression, then the pattern */re/* matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators (matches) and !~ (does not match). The following program prints the first field of all lines in which the fourth field matches *Asia*:

```
$4 ~ /Asia/ { print $1 }
```

The following program prints the first field of all lines in which the fourth field does not match *Asia*:

```
$4 !~ /Asia/ { print $1 }
```

In regular expressions, the following symbols are metacharacters with special meanings like the metacharacters in the operating system shell:

```
\ ^ $ . [] * + ? () |
```

For example, the metacharacters ^ and **$** match the beginning and end, respectively, of a string, and the metacharacter . (dot) matches any single character. Thus, the following string matches all records that contain exactly one character:

```
/^.$/
```

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, */[ABC]/* matches records containing any one of *A*, *B*, or *C* anywhere. Ranges of letters or digits can be abbreviated within brackets: */[a-zA-Z]/* matches any single letter.

If the first character after the bracket ([) is a circumflex (^), the class is complemented so that it matches any character not in the set: */[^a-zA-Z]/* matches any nonletter. The following program prints all records in which the second field is not a string of one or more digits (^ for beginning of string, *[0-9]+* for one or more digits, and **$** for end of string). Programs like this are often used for data validation:

```
$2 !~ /^[0-9]+$/
```

Parentheses ( ) are used for grouping and the pipe symbol (|) is used for alternatives. The following program matches lines containing any one of the four substrings *apple pie*, *apple tart*, *cherry pie*, or *cherry tart*:

```
/(apple|cherry) (pie|tart)/
```

To turn off the special meaning of a metacharacter, precede it with a \ (backslash). Thus, the following program prints all lines containing *b* followed by a dollar sign:

```
/b\$/
```

In addition to recognizing metacharacters, the **nawk** command recognizes the following C language escape sequences within regular expressions and strings:

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\*ddd*	octal value *ddd*
\\"	quotation mark
\*c*	any other character *c* literally

For example, to print all lines containing a tab, use this program:

```
/\t/
```

nawk interprets any string or variable on the right side of a ~ (tilde) or !~ as a regular expression. For example, consider this program:

```
$2 !~ /^[0-9]+$/
```

You could have written it like this:

```
BEGIN     { digits = "^[0-9]+$" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters like ^[*0-9*]+$. When a literal quoted string like "^[*0-9*]+$" is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose you want to match strings containing *b* followed by a dollar sign. The regular expression for this pattern is *b*\**$**. If you want to create a string to represent this regular expression, you must add one more backslash: "b\\\\**$**". The two regular expressions on each of the following lines are equivalent:

```
x ~ "b\\$"x ~ /b\$/
x ~ "b\$" x ~ /b$/
x ~ "b$"  x ~ /b$/
x ~ "\\t" x ~ /\t/
```

The precise form of regular expressions and the substrings they match are given in Table 2-2. The unary operators *, +, and ? have the highest precedence, followed by concatenation, and then alternative |. All operators are left associative. *r* stands for any regular expression.

Table 2-2
nawk Regular Expressions

Expression	Matches
c	any nonmetacharacter *c*
\c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
r1 r2	*r1* then *r2* (concatenation)
r1|*r2*	*r1* or *r2* (alternative)

2.3.4 Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (or), && (and), and ! (not). For example, suppose you want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is *Asia* and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The following program selects lines with *Asia* or *Africa* as the fourth field.

```
$4 == "Asia" || $4 == "Africa"
```

Another way to write the previous program is to use a regular expression with the alternative operator |:

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator ! has the highest precedence, followed by &&, and finally ||. The operators && and || evaluate their operands from left to right; evaluation stops as soon as truth or falsity is determined.

2.3.5 Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in the following:

pat1, pat2 { ... }

In this case, the action is performed for each line between an occurrence of *pat1* and the next occurrence of *pat2*, inclusive. As an example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string *Canada* up through the next occurrence of the string *Brazil*:

```
Canada   3852    24   North America
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

Similarly, since **FNR** is the number of the current record in the current input file (and *FILENAME* is the name of the current input file), the following program prints the first five records of each input file with the name of the current input file prepended.

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

2.4 Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

2.4.1 Built-in Variables

Table 2-3 lists the built-in variables that **nawk** maintains. Some of these have already been described; others are used in this and later sections.

Table 2-3
nawk Built-in Variables

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by **match**()	-
RLENGTH	length of string matched by **match**()	-
SUBSEP	subscript separator	"e034"

2.4.2 Arithmetic Expressions

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose you want to print the population density for each country in the file *countries*. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression "**1000 * $3 / $2** gives the population density in people per square mile. The following program

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

applied to the file *countries* prints the name of each country and its population density:

```
      USSR   30.3
    Canada    6.2
     China  234.6
       USA   60.6
    Brazil   35.3
 Australia    4.7
     India  502.0
 Argentina   24.3
     Sudan   19.6
   Algeria   19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, % (remainder), and ^ (exponentiation; ** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **nawk** recognizes and produces scientific (exponential) notation: `1e6`, `1E6`, `10e5`, and `1000000` are numerically equal.

nawk has assignment statements like those found in the C language. The simplest form is the assignment statement *v* = *e* where *v* is a variable or field name, and *e* is an expression. For example, to compute the number of Asian countries and their total population, you could write this:

```
$4 == "Asia"  { pop = pop + $3; n = n + 1 }
END           { print "population of", n,
                      "Asian countries in millions is", pop }
```

Applied to *countries*, the program produces this:

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, because **nawk** initializes each variable with the string value "" and the numeric value 0.

The assignments in the previous program can be written more concisely using the operators **+=** and **++** :

```
$4 == "Asia"  { pop += $3; ++n }
```

The operator **+=** is borrowed from the C language:

```
pop += $3
```

This has the same effect as the following:

```
pop = pop + $3
```

Note, however, that the **+=** operator is shorter and runs faster. The same is true of the **++** operator, which adds one to a variable.

The abbreviated assignment operators are **+=**, **–=**, ***=**, **/=**, **%=**, and **^=**. The following assignment statements are equivalent:

```
v op = e
v = v op e
```

The increment and decrement operators are **++** and **–**, respectively. As in C, they may be used as prefix (**++x**) or postfix (**x++**) operators. If **x** is 1, then **i=++x** increments **x**, then sets **i** to 2, while **i=x++** sets i to 1, then increments **x** . An analogous interpretation applies to prefix and postfix –.

Assignment, increment, and decrement operators may all be used in arithmetic expressions.

Default initialization is used to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3   { maxpop = $3; country = $1 }
END           { print country, maxpop }
```

Note, however, that this program would not be correct if all values of **$3** were negative.

nawk provides the built-in arithmetic functions shown in Table 2-4.

Table 2-4
nawk Built-in Arithmetic Functions

Function	Value Returned
atan2(*y*,*x*)	arctangent of y/x in the range $-\pi$ to π
cos(*x*)	cosine of *x*, with *x* in radians
exp(*x*)	exponential function of *x*
int(*x*)	integer part of *x* truncated towards 0
log(*x*)	natural logarithm of *x*
rand()	random number between 0 and 1
sin(*x*)	sine of *x*, with *x* in radians
sqrt(*x*)	square root of *x*
srand(*x*)	*x* is new seed for **rand**()

The expressions *x* and *y* are arbitrary expressions. The function **rand**() returns a pseudorandom floating-point number in the range (0,1), and **srand**(*x*) can be used to set the seed of the generator. If **srand**() has no argument, the seed is derived from the time of day.

2.4.3 Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in *"abc"* or *"hello, everyone"*. String constants may contain the C escape sequences for special characters listed in "Regular Expressions" earlier in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The following program prints each record preceded by its record number and a colon, with no blanks:

```
  { print NR ":" $0 }
```

The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

nawk provides the built-in string functions shown in Table 2-5. In this table, *r* represents a regular expression (either as a string or as */r/*), *s* and *t* string expressions, and *n* and *p* integers.

Table 2-5
nawk Built-in String Functions

Function	Description
gsub(*r,s***)**	substitute *s* for *r* globally in current record, return number of substitutions
gsub(*r,s,t***)**	substitute *s* for *r* globally in string *t*, return number of substitutions
index(*s,t***)**	return position of string *t* in *s*, 0 if not present
length(*s***)**	return length of *s*
match(*s,r***)**	return the position in *s* where *r* occurs, 0 if not present
split(*s,a***)**	split *s* into array *a* on FS, return number of fields
split(*s,a,r***)**	split *s* into array *a* on *r*, return number of fields
sprintf(*fmt,expr-list***)**	return *expr-list* formatted according to format string *fmt*
sub(*r,s***)**	substitute *s* for first *r* in current record, return number of substitutions
sub(*r,s,t***)**	substitute *s* for first *r* in *t*, return number of substitutions
substr(*s,p***)**	return suffix of *s* starting at position *p*
substr(*s,p,n***)**	return substring of *s* of length *n* starting at position *p*

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(1). The function **gsub(***r,s,t***)** replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. As in **ed**, the leftmost match is used and made as long as possible, and the number of substitutions made is returned. The function **gsub(***r,s***)** is a synonym for **gsub(***r,s,***$0)**. For example, the following program transcribes its input, replacing occurrences of *USA* with *United States*:

```
  { gsub(/USA/, "United States"); print }
```

The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index**(*s,t*) returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s*. The first character in a string is at position 1. The following returns 2:

```
  index("banana", "an")
```

The **length** function returns the number of characters in its argument string. This program prints each record, preceded by its length. Note that **$0** does not include the input record separator:

```
  { print length($0), $0 }
```

The following program applied applied to the file *countries* prints the longest country name, *Australia*:

```
  length($1) > max    { max = length($1); name = $1 }
  END                 { print name }
```

The **match**(*s,r*) function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function also sets two built-in variables **RSTART** and **RLENGTH** . **RSTART** is set to the starting position of the match in the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. If a match does not occur, **RSTART** is 0 and **RLENGTH** is –1. For example, the following program finds the first occurrence of the letter *i* followed by, at most, one character followed by the letter *a* in a record:

```
  { if (match($0, /i.?a/))
          print RSTART, RLENGTH, $0 }
```

It produces the following output on the file *countries*:

```
  17 2 USSR 8650    262     Asia
  26 3 Canada   3852    24      North America
  3 3 China 3692    866     Asia
  24 3 USA  3615    219     North America
  27 3 Brazil   3286    116     South America
  8 2 Australia 2968    14      Australia
  4 2 India 1269    637     Asia
  7 3 Argentina 1072    26      South America
  17 3 Sudan968     19      Africa
  6 2 Algeria   920     18      Africa
```

match() matches the leftmost longest matching string. For example, with the record *AsiaaaAsiaaaaan* as input, the following program matches the first string of *a*'s and sets **RSTART** to 4 and **RLENGTH** to 3.

```
{ if (match($0, /a+/))  print RSTART, RLENGTH, $0 }
```

The function **sprintf**(*format, expr1, expr2, . . ., exprn*) returns (without printing) a string containing *expr1*, expr2, . . ., exprn formatted according to the **printf** specifications in the string *format*. Refer to "The **printf** Statement", later in this chapter, for a complete specification of the format conventions. The following statement assigns to *x* the string produced by formatting the values of **$1** and **$2** as a ten-character string and a decimal number in a field of width at least six; **x** may be used in any subsequent computation.

```
x = sprintf("%10s %6d", $1, $2)
```

The function **substr**(*s,p,n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s,p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, you could abbreviate the country names in *countries* to their first three characters by invoking the following program on this file:

```
{ $1 = substr($1, 1, 3); print }
```

This produces the following results:

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting **$1** in the program forces **nawk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are concatenated by writing them one after another in an expression. For example, when invoked on file *countries*, the following program builds *s* up a piece at a time from an initially empty string:

```
{ s = s substr($1, 1, 3) " " }
```

```
END   { print s }
```

This is the result:

```
USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

2.4.4 Field Variables

The fields of the current record can be referred to by the field variables **$1**, **$2**, ..., **$NF**. Field variables share all of the properties of other variables—they may be used in arithmetic or string operations, and they may have values assigned to them. For example, you can divide the second field of the file *countries* by 1000 to convert the area from thousands to millions of square miles:

```
{ $2 /= 1000; print }
```

You can also assign a new string to a field:

```
BEGIN                          { FS = OFS = "\t" }
$4 == "North America"          { $4 = "NA" }
$4 == "South America"          { $4 = "SA" }
                               { print }
```

The **BEGIN** action in this program resets the input field separator **FS** and the output field separator **OFS** to a tab. The **print** in the fourth line of the program prints the value of **$0** after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, **$(NF–1)** is the second to last field of the current record. The parentheses are needed, since the value of **$NF–1** is 1 less than the value in the last field.

The initial value of a nonexistent field variable, such as **$(NF+1)**, is the empty string. However, a new field can be created by assigning a value to it. For example, the following program invoked on the file *countries* creates a fifth field giving the population density:

```
BEGIN   { FS = OFS = "\t" }
        { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record. The maximum is 100 fields per record.

2.4.5 Number or String

Variables, fields, and expressions can have both a numeric value and a string value, depending on the context. For example, in the context of an arithmetic expression such as the one that follows, *pop* and **$3** must be treated numerically since **+=** is a numeric operator; they are coerced to numeric values if necessary;

```
pop += $3
```

In a string context such as the following, **$1** and **$2** must be treated as strings since **:** is a string operator; they are coerced to string values if necessary:

```
print $1 ":" $2
```

In an assignment such as *v* = *e* or *v op=e*, the type of *v* becomes the type of *e*. In an ambiguous context such as `$1 == $2`, the type of the comparison depends on whether the fields are numeric or string. This can only be determined when the program runs and differs from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison `$1 == $2` succeeds on any pair of the following inputs:

```
1     1.0     +1     0.1e+1     10E-1    001
```

However, it fails on these inputs:

```
(null) 0
(null) 0.0
0a  0
1e50   1.0e50
```

There are two methods for coercing an expression of one type to the other:

number ""	concatenate a null string to a *number* to coerce it to type *string*
string + 0	add zero to a *string* to coerce it to type *numeric*

Thus, to force a string comparison between two fields, enter this:

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

2.4.6 Control Flow Statements

nawk provides **if–else**, **while**, **do–while**, and **for** statements, and statement grouping with braces, as in the C language.

The **if** statement syntax is as follows:

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

The *expression* acting as the conditional has no restrictions; it can include the relational operators **<**, **<=**, **>**, **>=**, **==**, and **!=**; the regular expression matching operators **~** and **!~**; the logical operators **||**, **&&**, and **!**; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, *expression* is evaluated first. If it is non-zero and non-null, *statement1* is executed; otherwise *statement2* is executed. The **else** is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program found in "Arithmetic Functions", earlier in this chapter, with an **if** statement results in the following program:

```
{     if (maxpop < $3) {
              maxpop = $3
              country = $1
      }
}
END   { print country, maxpop }
```

The **while** statement is like that of the C language:

while (*expression*) *statement*

The *expression* is evaluated; if it is non-zero and non-null, the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, this program prints all input fields one per line:

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

The **for** statement is like that of the C language:

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

It has the same effect as the following:

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

The following example does the same job as the **while** example above:

```
{ for (i = 1; i <= NF; i++)  print $i }
```

An alternative version of the **for** statement is described in the next section.

The **do** statement has the following form:

do *statement* **while** (*expression*)

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for** statements, which test for completion at the top of the loop.

The following example of a **do** statement prints all lines except those between *start* and *stop*:

```
/start/ {
            do {
    getline x
            } while (x !~ /stop/)
        }
        { print }
```

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **nawk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action, if any, is executed. Within the **END** action, the following statement causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero:

exit *expr*

2.4.7 Arrays

nawk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they are declared by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the following statement assigns the current input line to the **NR**th element of the array **x**:

```
x[NR] = $0
```

It is possible in principle (though slow) to read the entire input into an array with the following **nawk** program:

```
      { x[NR] = $0 }
END   { ... processing ... }
```

The first action merely records each input line in array **x** indexed by line number; processing is done in the **END** statement.

Array elements may also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array *pop*. The **END** action prints the total population of these two continents:

```
/Asia/ { pop["Asia"] += $3 }
/Africa/  { pop["Africa"] += $3 }
END { print "Asian population in millions is", pop["Asia"]
      print "African population in millions is",
                  pop["Africa"] }
```

Using the input file *countries*, the program above generates this output:

```
Asian population in millions is 1765
African population in millions is 37
```

If you had used *pop[Asia]* instead of *pop["Asia"]*, the expression would have used the value of the variable *Asia* as the subscript, and since the variable is uninitialized, the values would have been accumulated in *pop*[""].

Suppose you want to determine the total area in each continent of the file *countries*. Any expression can be used as a subscript in an array reference. The following expression uses the string in the fourth field of the current input record to index the array *area*:

```
area[$4] += $2
```

The value of the second field is accumulated in that entry:

```
BEGIN  { FS = "\t" }
    { area[$4] += $2 }
END { for (name in area)
              print name, area[name] }
```

Using the input file *countries*, this program produces the following output:

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

The following program uses a form of the **for** statement that iterates all defined subscripts of an array: it executes *statement* with the variable **i** set in turn to each value of **i** for which *array*[**i**] has been defined.

for (*i* **in** *array*) *statement*

The loop is executed once for each defined subscript. Defined subscripts are chosen in random order. Results are unpredictable when **i** or *array* is altered during the loop.

nawk does not provide multidimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by a string (stored in the variable **SUBSEP**). For example, the following statements create an array which behaves like a two-dimensional array; the subscript is the concatenation of **i**, **SUBSEP**, and **j**:

```
for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
             arr[i,j] = ...
```

You can determine whether a particular subscript **i** occurs in an array *area* by testing the condition **i** in *area*, as in the following:

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating *area["Africa"]*, which would happen if you used this:

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array *area* contains an element with value *Africa*.

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The following function splits the string *s1:s2:s3* into three fields, using a colon as the separator, and stores *s1* in *a[1]*, *s2* in *a[2]*, and *s3* in *a[3]*:

```
split("s1:s2:s3", a, ":")
```

The number of fields found here (3 in this case) is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element may be deleted with a **delete** statement of the form:

delete *arrayname* [*subscript*]

2.4.8 User-Defined Functions

nawk provides user-defined functions. A function is defined as follows:

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

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function, these variables refer to the actual parameters when the function is called. There must be no space between the function name and the open parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function:

```
function fact(n) {
     if (n <= 1)
          return 1
     else
          return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables.) The **return** statement is optional, but the returned value is undefined if it is not included.

2.4.9 Some Lexical Conventions

Comments can be placed in **nawk** programs; they begin with the character **#** and end at the end of the line like this:

```
print x, y     # this is a comment
```

Statements in an **nawk** program normally occupy a single line. Several statements may occur on a single line if they are separated by semicolons. A long statement may be continued over several lines by terminating each continued line by a backslash. (It is not possible to continue a "..." string.) This explicit continuation is rarely necessary, however, since statements continue automatically if the line ends with a comma (for example, as might occur in a **print** or **printf** statement) or after the operators **&&** and **||**.

Several pattern-action statements may appear on a single line if separated by semicolons.

2.5 Output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used for simple output; **printf** is used for more carefully formatted output. Like the shell, **nawk** lets you redirect output, so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

2.5.1 The print Statement

The statement

print *expr1*, *expr2*, . . ., *exprn*

prints the string value of each expression separated by the output field separator followed by the output record separator. A **print** statement with no arguments is an abbreviation for the following:

```
print $0
```

To print an empty line use this:

```
print ""
```

2.5.2 Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. By default, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN   { OFS = ":"; ORS = "\n\n" }
        { print $1, $2 }
```

Notice that `{ print $1 $2 }` prints the first and second fields with no intervening output field separator, because **$1 $2** is a string consisting of the concatenation of the first two fields.

2.5.3 The printf Statement

nawk's printf statement is the same as that in C except that the asterisk (*) format specifier is not supported. The **printf** statement has the following general form:

printf *format*, *expr1*, *expr2*, . . ., *exprn*

format is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list. Table 2-6 lists each specification. Specifications begin with a percent sign (%) and end with a letter that determines the conversion.

Table 2-6
nawk printf Conversion Characters

Letter	Prints Expression as
c	single character
d	decimal number
e	[–]**d.ddddddE**[+–]**dd**
f	[–]**ddd.dddddd**
g	**e** or **f** conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

The following additional modifiers for formatting **printf** output can be placed between the % and the specification letter.

–	Left-justifies the expression in its field.
width	Pads the output field to this width as needed; fields that begin with a leading **0** are padded with zeros.
.precision	Specifies maximum string width or digits to the right of decimal point.

Here are some examples of **printf** statements along with the corresponding output:

```
             Statement                          Output

printf "%d", 99/2                   49
printf "%e", 99/2                   4.950000e+01
printf "%f", 99/2                   49.500000
printf "%6.2f", 99/2                49.50
printf "%g", 99/2                   49.5
printf "%o", 99                     143
printf "%06o", 99                   000143
printf "%x", 99                     63
printf "|%s|", "January"            |January|
printf "|%10s|", "January"          |   January|
printf "|%-10s|", "January"         |January   |
printf "|%.3s|", "January"          |Jan|
printf "|%10.3s|", "January"        |       Jan|
printf "|%-10.3s|", "January"       |Jan       |
printf "%%"                         %
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

2.5.4 Output into Files

It is possible to print output into files instead of to standard output by using the **>** and **>>** redirection operators. For example, the following program invoked on the file *countries* prints all lines where the population (third field) is bigger than 100 into a file called *bigpop*, and all other lines into *smallpop*:

```
$3 > 100   { print $1, $3 >"bigpop" }
$3 <= 100  { print $1, $3 >"smallpop" }
```

Notice that the file names have to be quoted; without quotes, *bigpop* and *smallpop* are merely uninitialized variables. If the output file names were created by an expression, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the **>** operator has higher precedence than concatenation; without parentheses, the concatenation of *tmp* and *FILENAME* would not work.

NOTE

Files are opened once in an **nawk** *program. If* **>** *is used to open a file, its original contents are overwritten. But if* **>>** *is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.*

2.5.5 Output into Pipes

It is also possible to direct printing into a pipe, with a command on the other end, instead of into a file. The following statement causes the output of **print** to be piped into the *command-line*:

```
print | "command-line"
```

Although they are shown here as literal strings enclosed in quotes, the command-line and filenames can come from variables and the return values from functions.

Suppose you want to create a list of continent-population pairs, sorted alphabetically by continent. The **nawk** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called *pop*. Then it prints each continent and its population, and pipes this output into the **sort** command.

```
BEGIN    { FS = "\t" }
         { pop[$4] += $3 }
END      { for (c in pop)
               print c ":" pop[c] | "sort" }
```

Invoked on the file *countries*, this program yields the following:

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (for example, the pipe above is literally named **sort**), but they are created and opened only once in the entire run. So, in the last example, for all **c** in *pop*, only one sort pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement **close**(*file*) closes a file or pipe; *file* is the string used to create it in the first place, as in

```
  close("sort")
```

When opening or closing a file, different strings are different commands.

2.6 Input

The most common way to give input to an **nawk** program is to name the file(s) containing the input on the command line. There are several other methods of giving input to **nawk**, each of which are described in the following section.

2.6.1 Files and Pipes

Input can be given to **nawk** by putting the input data into a file, say *nawkdata*, and then executing the following:

```
  nawk 'program' nawkdata
```

nawk reads its standard input if no file names are given (refer to "Running **nawk**" earlier in this chapter); thus, a second common arrangement is to have another program pipe its output to **nawk**. For example, **egrep**(1) selects input lines containing a specified regular expression, but it can do so faster than **nawk** since this is the only thing it does. In the following example, **egrep** quickly finds the lines containing *Asia* and passes them on to **nawk** for processing:

```
  egrep 'Asia' countries | nawk '...'
```

2.6.2 Input Separators

Since the default setting of the field separator **FS** is blanks or tabs with leading blanks discarded, each of these lines has the same first field:

```
      field1   field2
    field1
  field1
```

When the field separator is set to tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. The following example sets the field separator to an optional comma followed by any number of blanks and tabs:

```
BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" }
```

FS can also be set on the command line by specifying the **–F** option. This example behaves the same as the previous example:

```
nawk -F'(,[ \t]*)|([ \t]+)' '...'
```

Regular expressions used as field separators match the leftmost longest occurrences (as in **sub()**), but do not match null strings.

2.6.3 Multiline Records

Records are normally separated by newlines, so that each line is a record, but this can be changed in a limited way. If the built-in record separator variable **RS** is set to the empty string, as in the following example, input records can be several lines long; a sequence of empty lines separates records:

```
BEGIN   { RS = "" }
```

A common way to process multiline records is to use the following statement to set the record separator to an empty line and the field separator to a newline:

```
BEGIN   { RS = ""; FS = "\n" }
```

The limit on the length of a record is 2500 characters. For more information about processing multiline records, refer to "The getline Function" and "Cooperation with the Shell". later in this chapter.

2.6.4 The getline Function

nawk's facility for automatically breaking its input into records that are more than one line long later in this chapter. is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting **RS** to null doesn't work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function **getline** can be used to read input either from the current input or from a file or pipe, by redirection analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF**, **NR**, and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and –1 if some error occurred (such as failure to open a file).

For example, suppose you have input data consisting of multiline records, each of which begins with a line beginning with **START** and ends with a line beginning with **STOP**. The following **nawk** program processes these multiline records, a line at a time, putting the lines of the record into consecutive entries of an array:

```
f[1] f[2] ... f[nf]
```

Once the line containing **STOP** is encountered, the record can be processed from the data in the array **f**. Notice that this code relies on the fact that **&&** evaluates its operands from left to right and stops as soon as one is true:

```
/^START/ {
          f[nf=1] = $0
          while (getline && $0 !~ /^STOP/)
               f[++nf] = $0
          # now process the data in f[1]...f[nf]
          ...
}
```

The same job can also be done by the following program:

```
/^START/ && nf==0   { f[nf=1] = $0 }
nf > 1 { f[++nf] = $0 }
/^STOP/   { # now process the data in f[1]...f[nf]
     ...
     nf = 0
}
```

The following statement reads the next record into the variable **x**. No splitting is done and **NF** is not set:

```
getline x
```

This statement reads from *file* instead of the current input. It has no effect on **NR** or **FNR**, but field splitting is performed and **NF** is set:

```
getline <"file"
```

This statement gets the next record from *file* into *x*; no splitting is done, and **NF**, **NR**, and **FNR** are untouched:

```
getline x <"file"
```

If a filename is an expression, it must be placed in parentheses for correct evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) {  ... }
```

This is because **<** has precedence over concatenation. Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets **x** to read the file tmp and not *tmp <value of FILENAME>*.

Also, if you use this **getline** statement form, a statement like

```
while ( getline x < file ) { ... }
```

loops forever if the file cannot be read, because **getline** returns –1, not zero, if an error occurs. A better way to write this test is:

```
while ( getline x < file > 0) { ... }
```

It is also possible to pipe the output of another command directly into **getline**. For example, this statement executes **who** and pipes its output into **getline**:

```
while ("who" | getline)
     n++
```

Each iteration of the **while** loop reads one more line and increments the variable **n**, so after the **while** loop terminates, **n** contains a count of the number of users.

Similarly, this statement pipes the output of **date** into the variable **d**, thus setting **d** to the current date:

```
"date" | getline d
```

Table 2-7 summarizes the **getline** function.

Table 2-7
getline Function

Form	Sets
getline	**$0, NF, NR, FNR**
getline *var*	*var*, **NR, FNR**
getline <*file*	**$0, NF**
getline *var* <*file*	*var*
cmd | **getline**	**$0, NF**
cmd | **getline** *var*	*var*

2.6.5 Command-Line Arguments

The command-line arguments are available to an **nawk** program: the array **ARGV** contains the elements **ARGV[0]**, ..., **ARGV[ARGC–1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **nawk**); the remaining arguments are others given on the command line (excluding the program and any optional arguments). The following command line contains an **nawk** program that echoes the arguments that appear after the program name:

```
nawk '
BEGIN {
    for (i = 1; i < ARGC; i++)
        printf "%s ", ARGV[i]
    printf "\n"
}' $*
```

The arguments may be modified or added to; **ARGC** may be altered. As each input file ends, **nawk** treats the next non-null element of **ARGV** (up to the current value of **ARGC–1**) as the name of the next input file.

The following form is an exception to the rule that an argument is a file name:

var=value

If this form is used, the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a file name. If *value* is a string, no quotes are needed.

2.7 Using nawk with Other Commands and the Shell

nawk is most powerful when it is used in conjunction with other programs. The following sections describe some of the ways in which **nawk** programs cooperate with other commands.

2.7.1 The system Function

The built-in function **system**(*command-line*) executes the command *command-line*, which may well be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed.

For example, the following program calls the **cat** command to print the file named in the second field of every input record whose first field is **#include** after stripping any <, >, or " that might be present:

```
$1 == "#include"  { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

2.7.2 Cooperation with the Shell

In all the examples given in this chapter so far, the **nawk** program was in a file and specified with the **-f** option, or it appeared on the command line enclosed in single quotes. Since **nawk** uses many of the same characters as the shell does (such as **$** and "), the single quotes around an **nawk** program ensure that the shell passes the entire program unchanged to **nawk**.

Now, consider writing a command **addr** that searches a file *addresslist* for name, address, and telephone information. Suppose that *addresslist* contains names and addresses in which a typical entry is a multiline record such as the following:

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

You want to search the address list by issuing commands like this:

```
addr Emlin
```

That is easily done by a program of the following form:

```
nawk '
BEGIN  { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called *addr* that contains the following:

```
nawk '
BEGIN  { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **nawk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The **$1** is outside the quotes, visible to the shell, which therefore replaces it by the pattern **Emlin** when the command **addr Emlin** is invoked. The file **addr** can be made executable by changing its mode with the following operating system command: **chmod +x addr**.

A second way to implement **addr** relies on the fact that the shell substitutes for **$** parameters within double quotes:

```
nawk "
BEGIN  { RS = \"\" }
/$1/
" addresslist
```

Here you must protect the quotes defining **RS** with backslashes so that the shell passes them on to **nawk**, uninterpreted by the shell. **$1** is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr** *pattern* is invoked.

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **nawk** program that explicitly reads through the address list with **getline**:

```
nawk '
BEGIN    { RS = ""
           while (getline < "addresslist")
              if ($0 ~ ARGV[1])
                 print $0
} ' $*
```

All processing is done in the **BEGIN** action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

2.8 Sample Applications

nawk has been used in surprising ways. We have seen **nawk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **nawk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. This section presents a few more examples to illustrate some additional **nawk** programs.

2.8.1 Generating Reports

nawk is especially useful for producing reports that summarize and format information. Suppose you wish to produce a report from the file *countries* in which you list the continents alphabetically and, after each continent, its countries in decreasing order of population:

```
Africa:
    Sudan           19
    Algeria         18

Asia:
    China          866
    India          637
    USSR           262

Australia:
    Australia       14

North America:
    USA            219
    Canada          24

South America:
    Brazil         116
    Argentina       26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, you create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program *triples* which uses an array *pop* indexed by subscripts of the form *continent:country* to store the population of a given country. The **print** statement in the **END** section of the program creates the list of continent-country-population triples that are piped to the **sort** routine.

```
BEGIN   { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END     { for (cc in pop)
              print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for **sort** deserve special mention. The **–t:** argument tells **sort** to use a colon as its field separator. The **+0 and –1** arguments make the first field the primary sort key. In general, *+i –j* makes fields *i+1*, *i+2*, ..., *j* the sort key. If *–j* is omitted, the fields from *i+1* to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (**n** is for numeric, **r** for reverse order). Invoked on the file *countries*, this program produces the following output:

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

The output is in the right order but the wrong format. To transform the output into the desired form, run it through a second **nawk** program *format* shown here:

```
BEGIN   { FS = ":" }
{       if ($1 != prev) {
```

```
                print "\n" $1 ":"
                prev = $1
            }
            printf "\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

```
nawk -f triples countries | nawk -f format
```

gives us our desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **nawk**s and **sort**s.

2.8.2 Additional Examples

The following sections describe some other uses for **nawk**. In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

Word Frequencies

The first example illustrates associative arrays for counting. Suppose you want to count the number of times each word appears in the input, where a word is any contiguous sequence of nonblank, nontab characters. The following program prints the word frequencies, sorted in decreasing order:

```
      { for (w = 1; w <= NF; w++) count[$w]++ }
END   { for (w in count) print count[w], w | "sort -nr" }
```

The first statement uses the array *count* to accumulate the number of times each word is used. Once the input has been read, the second **for** loop pipes the final count along with each word into the **sort** command.

Accumulation

Suppose you have two files, *deposits* and *withdrawals*, of records containing a name field and an amount field. For each name you want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
nawk '
FILENAME == "deposits"     { balance[$1] += $2 }
FILENAME == "withdrawals"   { balance[$1] -= $2 }
END                         { for (name in balance)
                                    print name, balance[name]
} ' deposits withdrawals
```

The first statement uses the array *balance* to accumulate the total amount for each name in the file *deposits*. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name is created by the second statement. The **END** action prints each name with its net balance.

Random Choice

The following function prints (in order) **k** random elements from the first **n** elements of the array **A**. In the program, **k** is the number of entries that still need to be printed, and **n** is the number of elements yet to be examined. The decision of whether to print the *i*th element is determined by the test **rand() < k/n**.

```
function choose(A, k, n) {
        for (i = 1; n > 0; i++)
              if (rand() < k/n--) {
                      print A[i]
                      k--
              }
        }
}
```

Shell Facility

The following **nawk** program simulates (crudely) the history facility of the operating system shell. A line containing only **=f** reexecutes the last command executed. A line beginning with **=f** *cmd* reexecutes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
                system(x[NR] = x[NR-1])
            else
                for (i = NR-1; i > 0; i--)
                      if (x[i] ~ $2) {
                              system(x[NR] = x[i])
                              break
                      }
                next }

/./             { system(x[NR] = $0) }
```

Form-Letter Generation

The following program generates form letters, using a template stored in a file called *form.letter* as follows:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

Replacement text of this form is generated:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The **BEGINf** action stores the template in the array *template*; the remaining action cycles through the input data, using **gsub** to replace template fields of the form $*n* with the corresponding data fields.

```
BEGIN {    FS = "|"
                while (getline <"form.letter")
                    line[++n] = $0
}
{       for (i = 1; i <= n; i++) {
                    s = line[i]
                    for (j = 1; j <= NF; j++)
                                gsub("\\$"j, $j, s)
                    print s
        }
}
```

2.9 nawk Summary

2.9.1 Command Line

nawk *'program'*
nawk *'program'* –
nawk *'program' files*
nawk *'program' files* –
nawk –f *nawk-program-file files*
nawk –f *nawk-program-file files* –
nawk –F*r* sets field separator to a character or string;
–Ft sets separator to tab

A hyphen (–), with or without *files* specified, indicates input coming from *stdin*.

2.9.2 Patterns

BEGIN
END
/regular expression/
relational expression
pattern **&&** *pattern*
pattern **| |** *pattern*
(*pattern*)
!*pattern*
pattern, *pattern*

2.9.3 Control Flow Statements

if (*expr*) *statement* [**else** *statement*]
if (*subscript* **in** *array*) *statement* [**else** *statement*]
while (*expr*) *statement*
for (*expr*; *expr*; *expr*) *statement*
for (*subscript* **in** *array*) *statement*
or (*var* **in** *array*) *statement*
do *statement* **while** (*expr*)
break
continue
next
exit [*expr*]
return [*expr*]

2.9.4 Input-Output

close(*file*)	close *file*
getline	set **$0** from next input record; set **NF, NR, FNR**
getline <*file*	set **$0** from next record of *file*; set **NF**
getline *var*	set *var* from next input record; set **NR, FNR**
getline *var* <*file*	set *var* from next record of *file*
print	print current record
print *expr-list*	print expressions
print *expr-list* >*file*	print expressions on *file*
printf *fmt*, *expr-list*	format and print
printf *fmt*, *expr-list* >*file*	format and print on *file*
system(*cmd-line*)	execute command *cmd-line*, return status

In **print** and **printf** above, >>*file appends to the file*, and | *command* writes on a pipe. Similarly, *command* | **getline** pipes into **getline**. **getline** returns 0 on end of file, and –1 on error.

2.9.5 Functions

func *name*(*parameter list*) { *statement* }
function *name*(*parameter list*) { *statement* }
function-name(*expr*, *expr*, ...)

2.9.6 String Functions

gsub(*r*,*s*,*t*)	Substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use **$0**
index(*s*,*t*)	Return index of string *t* in string *s*, or 0 if not present
length(*s*)	Return length of string *s*
match(*s*, *r*)	Return position in *s* where regular expression *r* occurs, or 0 if *r* is not present

split(*s,a,r***)**	Split string *s* into array *a* on regular expression *r*, return number of fields; if *r* omitted, **FS** is used in its place
sprintf(*fmt, expr-list***)**	Print *expr-list* according to *fmt*, return resulting string
sub(*r,s,t***)**	Like gsub except only the first matching substring is replaced
substr(*s,i,n***)**	Return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

2.9.7 Arithmetic Functions

atan2(*y,x***)**	Arctangent of y/x in radians
cos(*expr***)**	Cosine (angle in radians)
exp(*expr***)**	Exponential
int(*expr***)**	Truncate to integer
log(*expr***)**	Natural logarithm
rand()	Random number between 0 and 1
sin(*expr***)**	Sine (angle in radians)
sqrt(*expr***)**	Square root
srand(*expr***)**	New seed for random number generator; use time of day if no *expr*

2.9.8 Operators (Increasing Precedence)

= += -= *= /= %= ^=	assignment
?:	conditional expression
||	logical OR

&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
blank	string concatenation
+ –	add, subtract
* / %	multiply, divide, mod
+ – !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
$	field

2.9.9 Regular Expressions (Increasing Precedence)

c	matches nonmetacharacter *c*
\c	matches literal character *c*
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[*abc*...]	character class matches any of *abc*...
[^*abc*...]	negated class matches any but *abc*... and newline
r1 | *r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r+	matches one or more *r*'s
*r**	matches zero or more *r*'s
r?	matches zero or one *r*'s
(*r*)	grouping: matches *r*

2.9.10 Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0..**ARGC-1**)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)

NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default **%.6g**)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	index of first character matched by **match**(); 0 if no match
RLENGTH	length of string matched by **match**(); –1 if no match
SUBSEP	separates multiple subscripts in array elements; default "\034"

2.9.11 Limits

Any particular implementation of **nawk** enforces some limits. Here are typical values:

100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per **printf** string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the local
    machine, (1e–38..1e+38)

2.9.12 Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the following assignment, its type is set to that of the expression:

var = *expr*

(Assignment includes +=, –=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in the following example, the type of *v1* becomes that of *v2*:

v1 = *v2*

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by the following subterfuge:

```
expr + 0
```

The type of any expression can be coerced to string by this subterfuge (that is, concatenation with a null string):

```
expr ""
```

Uninitialized variables have the numeric value 0 and the string value "". Accordingly, if *x* is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But the following is false:

```
if (x == "0") ...
```

The type of a field is determined by context when possible. The following example clearly implies that **$1** is to be numeric:

```
$1++
```

This example implies that **$1** and **$2** are both to be strings:

```
$1 = $1 "," $2
```

Coercion is done as needed.

In contexts where types cannot be reliably determined, the type of each field is determined on input, for example:

```
if ($1 == $2) ...
```

All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value ""; they are not numeric. Nonexistent fields (i.e., fields past **NF**), and array elements created by **split()**. are treated this way, too.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if *arr*[*i*] does not currently exist, the following example causes it to exist with the value "" so the **if** is satisfied:

```
if (arr[i] == "") ...
```

The following special construction determines if *arr*[*i*] exists without the side effect of creating it if it does not:

```
if (i in arr) ...
```

Index

3. awk

3. awk

Chapter 3
awk

Figures

Tables

Chapter 3
awk

3.1 Introduction

NOTE

This chapter describes an earlier version of **awk**. *If you want to use this version of* **awk**, *specify* **awk** *on the invocation line. If you want to use the newer version of* **awk**, *refer to Chapter 2.*

awk is a file-processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. **awk** performs the following tasks:

- Generates reports
- Matches patterns
- Validates data
- Filters data for transmission

The first part of this chapter shows a general statement of the **awk** syntax. Examples show the syntax rules in use in "Using **awk**", later in this chapter.

3.1.1 Program Structure

An **awk** program is a sequence of statements of the following form:

pattern {action}
pattern {action}
...

awk operates on a set of input files, scanning the input lines, in order, one at a time. In each line, **awk** searches for the pattern described in the **awk** program. If that pattern is found in the input line, a corresponding action is performed. In this way, each statement of the **awk** program is executed for a given input line. When all the patterns are tested, the next input line is fetched and the **awk** program is once again executed from the beginning.

In the **awk** command, either the pattern or the action may be omitted, but not both. If there is no action for a pattern, the matching line is simply printed. If there is no pattern for an action, then the action is performed for every input line. The null **awk** program does nothing. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

For example, this **awk** program prints every input line that has an *x* in it:

```
/x/ {print}
```

An **awk** program has the following structure:

```
BEGIN section
record or main section
END section
```

The **BEGIN** section is run before any input lines are read, and the **END** section is run after all the data files are processed. The record section is run over and over for each separate line of input. The words **BEGIN** and **END** are actually special patterns recognized by **awk**.

Values are assigned to variables from the **awk** command line. The **BEGIN** section is run before these assignments are made.

3.1.2 Running awk

There are two ways to run an **awk** program, shown in the following syntax block:

```
awk -Fc [ 'program'              ] [file ...] [-]
        [ -f awk-program-file    ]
```

c	Any character to be used as a field separator. Blanks and tabs are the default field separators.
program	The **awk** program itself entered on the command line and enclosed in single quotes.
awk-program-file	A file containing an **awk** program.
file	An input file containing field-separated records for processing by **awk**.
–	Standard input. If no input *files* are specified on the command line, or if a hyphen is specified, **awk** looks to *stdin* for input. If input *files* and a hyphen are specified on the command line, **awk** uses both the files and *stdin* for input in the order they appear on the command line.

Running a Program from a Command Line

If the program is short, it is often easiest to make the program the first argument on the command line. The program must be enclosed in single quotes in order for the shell to accept the entire string as the first argument to **awk**. In the following example, the **awk** program (between the single quotes) matches the pattern *x* in *file1* and prints the lines that contain a match:

```
awk ' /x/ {print} ' file1
```

If no input *file* is specified, **awk** expects input from standard input. You can also specify that input comes from *stdin* by using a hyphen (–) as one of the input files. In this example, the **awk** program (between the single quotes) looks for input from *file1* and from *stdin*. Input from *file1* is processed first, followed by input from *stdin*:

```
awk ' /x/ {print} ' file1 -
```

Running a Program in a File

If your **awk** program is long or you want to save it for future use, you can store the program in a separate file, for example *awkprog*. Specifying the **-f** option on the command line tells **awk** to fetch it, as follows:

```
  awk -f awkprog file1
```

The input file *file1* may include *stdin* as is shown previously.

These alternative ways of presenting your **awk** program for processing are illustrated by the following examples. Consider the following command:

```
  awk ' BEGIN {print "hello, world" exit} '
```

When this command is given to the shell, the following string is printed to standard output:

```
  hello, world
```

This **awk** program could be run by putting the following line in a file named *awkprog*:

```
  BEGIN {print "hello, world" exit}
```

The following command given to the shell would have the same effect as the previous procedure:

```
  awk -f awkprog
```

3.1.3 Lexical Units

All **awk** programs are made up of lexical units called tokens. In **awk** there are eight token types:

1. numeric constants
2. string constants
3. keywords
4. identifiers
5. operators
6. record and field tokens
7. comments
8. tokens used for grouping

Numeric Constants

A numeric constant is either a decimal constant or a floating constant. A *decimal constant* is a non-null sequence of digits containing at most one decimal point as in 12, 12., 1.2, and .12. A *floating constant* is a decimal constant followed by *e* or *E*, followed by an optional + or – sign, followed by a non-null sequence of digits as in 12e3, 1.2e3, 1.2e–3, and 1.2E+3. The maximum size and precision of numeric constants are machine dependent.

String Constants

A string constant is a sequence of zero or more characters surrounded by double quotes as in "", "a", "ab", and "12". A double quote is put in a string by preceding it with a backslash (\) as in "He said, \" Sit! \"". A newline is put in a string by using **\n** in its place. No other characters need to be escaped. Strings can be any length.

Keywords

The following strings are used as keywords:

BEGIN	**OFMT**	**getline**	**print**
END	**RS**	**if**	**printf**
FILENAME	**break**	**in**	**split**
FS	**close**	**index**	**sprintf**
NF	**continue**	**int**	**sqrt**
NR	**exit**	**log**	**string**
OFS	**exp**	**next**	**substr**
ORS	**for**	**number**	**while**
			length

Identifiers

Identifiers in **awk** serve to denote variables and arrays. An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore. Uppercase and lowercase letters are different.

Operators

awk has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in **egrep**(1) and **lex**(1).

Assignment operators are shown in Table 3-1.

Table 3-1
awk Assignment Operators

Symbol	Usage	Description
=	assignment	
+=	plus-equals	X += Y is similar to X = X+Y
–=	minus-equals	X–=Y is similar to X = X–Y
*=	times-equals	X *= Y is similar to X = X*Y
=	divide-equals	X /= Y is similar to X = X/Y
%=	mod-equals	X %= Y is similar to X = X%Y
++	increments	++X and X++ are similar to X=X+1
– –	decrements	– – X and X – – are similar to X = X –1

Arithmetic operators are shown in Table 3-2.

Table 3-2
awk Arithmetic Operators

Symbol	Description
+	unary and binary plus
–	unary and binary minus
*	multiplication
	division
%	modulus
(...)	grouping

Relational operators are shown in Table 3-3.

Table 3-3
awk Relational Operators

Symbol	Description
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Logical operators are shown in Table 3-4.

Table 3-4
awk Logical Operators

Symbol	Description
&&	and
||	or
!	not

Regular expression matching operators in **awk** are shown in the Table 3-5.

Table 3-5
Operators for Matching Regular Expressions

Symbol	Description
~	matches
!~	does not match

Record and Field Tokens

$0 is a special variable whose value is that of the current input record. **$1**, **$2**, and so forth, are special variables whose values are those of the first field, the second field, and so forth, of the current input record. The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input record. Thus **$NF** has the value of the last field of the current input record. Notice that the first field of each record is numbered 1 and that the number of fields can vary from record to record. None of these variables is defined in the action associated with a **BEGIN** or **END** pattern, where there is no current input record.

The keyword **NR** (Number of Records) is a variable whose value is the number of input records read so far. The first input record read is 1.

Record Separators. The keyword **RS** (Record Separator) is a variable whose value is the current record separator. The value of **RS** is initially set to newline, indicating that adjacent input records are separated by a newline. Keyword **RS** may be changed to any character, *c*, by executing the assignment statement **RS** = "*c*" in an action.

Field Separator. The keyword **FS** (Field Separator) is a variable indicating the current field separator. Initially, the value of **FS** is a blank, indicating that fields are separated by white space (any non-null sequence of blanks and tabs). Keyword **FS** is changed to any single character, *c*, by executing the assignment statement **F** = "*c*" in an action or by using the optional command line argument **–F***c*. Two values of *c* have special meaning: space and tab (\t). The assignment statement **FS** =" " makes white space (a tab or blank) the field separator; on the command line, **–F\t** makes a tab the field separator.

If the field separator is not a blank, then there is a field in the record on each side of the separator. For instance, if the field separator is 1, the record *1XXX1* has three fields. The first and last are null. If the field separator is blank, then fields are separated by white space and none of the **NF** fields are null.

Multiline Records. The assignment **RS** =" " makes an empty line the record separator and makes a non-null sequence (consisting of blanks, tabs, and possibly a newline) the field separator. With this setting, none of the first **NF** fields of any record are null.

Output Record and Field Separators. The value of **OFS** (Output Field Separator) is the output field separator. It is put between fields by **print**. The value of **ORS** (Output Record Separators) is put after each record by **print**. Initially, **ORS** is set to a newline and **OFS** to a space. These values may change to any string by assignments such as **ORS** = "*abc*" and **OFS** = "*xyz*".

Comments

A comment is introduced by a # and terminated by a newline. For example:

```
  #    this line is a comment
```

A comment can be appended to the end of any line of an **awk** program.

Tokens Used for Grouping

Tokens in **awk** are usually separated by non-null sequences of blanks, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces {...} surround actions, slashes /../ surround regular expression patterns, and double quotes "..." surround string constants.

3.1.4 Primary Expressions

In **awk**, patterns and actions are made up of expressions. The basic building blocks of expressions are the primary expressions:

numeric constants
string constants
variables
functions

Each expression has both a numeric and a string value, one of which is usually preferred. The rules for determining the preferred value of an expression are explained in "Expressions" later in this chapter.

Numeric Constants

The format of a numeric constant was defined earlier in this chapter (refer to "Lexical Units"). Numeric values are stored as floating-point numbers. The string value of a numeric constant is computed from the numeric value. The preferred value is the numeric value. Numeric values for string constants are shown in Table 3-6.

Table 3-6
Numeric Values for String Constants

Numeric Constant	Numeric Value	String Value
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

String Constants

The format of a string constant was defined earlier in "Lexical Units." The numeric value of a string constant is 0 unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself. String values for string constants are shown in Table 3-7.

Table 3-7
String Values for String Constants

String Constant	Numeric Value	String Value
""	0	empty space
"a"	0	a
"XYZ"	0	XYZ
"o"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	0.5	.5e2

Variables

A variable is one of the following:

identifier
identifier [*expression*]
$term

The numeric value of any uninitialized variable is 0, and the string value is the empty string.

An *identifier* by itself is a simple variable. A variable of the form *identifier* [*expression*] represents an element of an associative array named by *identifier*. The string value of *expression* is used as the index into the array. The preferred value of *identifier* or *identifier* [*expression*] is determined by context.

The variable **$0** refers to the current input record. Its string and numeric values are those of the current input record. If the current input record represents a number, then the numeric value of **$0** is the number and the string value is the literal string. The preferred value of **$0** is string unless the current input record is a number. **$0** cannot be changed by assignment.

The variables **$1**, **$2**, **...** refer to fields 1, 2, and so forth, of the current input record. The string and numeric value of **$i** for **1<=i<=NF** are those of the **i**th field of the current input record. As with **$0**, if the **i**th field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the **i**th field is a number. **$i** may be changed by assignment; the value of **$0** is changed accordingly.

In general, **$term** refers to the input record if **term** has the numeric value 0 and to field **i** if the greatest integer in the numeric value of **term** is **i**. If **i**<0 or if **i**>=100, then accessing **$i** causes **awk** to produce an error diagnostic. If **NF**<**i**<=100, then **$i** behaves like an uninitialized variable. Accessing **$i** for **i** > **NF** does not change the value of **NF**.

Functions

awk has a number of built-in functions that perform common arithmetic and string operations. The built-in arithmetic functions are as follows:

exp (*expression*)
int (*expression*)
log (*expression*)
sqrt (*expression*)

These functions (**exp, int, log,** and **sqrt**) compute the exponential, integer part, natural logarithm, and square root, respectively, of the numeric value of *expression*. The (*expression*) may be omitted; then the function is applied to **$0**. The preferred value of an arithmetic function is numeric. The built-in string functions are as follows:

getline
index(*expression1*, *expression2*)
length(*expression*)
split(*expression*, *identifier*, *expression2*)
split(*expression*, *identifier*)
sprintf(*format*, *expression1*, *expression2...*)
substr(*expression1*, *expression2*)
substr(*expression1*, *expression2*, *expression3*)

The function **getline** causes the next input record to replace the current record. One is returned if there is a next input record; zero is returned if there is no next input record. The value of **NR** is updated.

The function **index**(*e1*,*e2*) takes the string value of expressions *e1* and *e2* and returns the first position of where *e2* occurs as a substring in *e1*. If *e2* does not occur in *e1*, index returns 0. For example:

```
index ("abc", "bc")=2
index ("abc", "ac")=0.
```

The function **length** without an argument returns the number of characters in the current input record. With an expression argument, **length**(*e*) returns the number of characters in the string value of *e*. For example:

```
length ("abc")=3
length (17)=2.
```

The function **split**(*e, array, sep*) splits the string value of expression *e* into fields that are then stored in *array*[1], *array*[2], ..., *array*[*n*] using the string value of *sep* as the field separator. **split** returns the number of fields found in *e*. The function **split**(*e, array*) uses the current value of **FS** to indicate the field separator. For example, after invoking the following function, **a**[**2**], ..., **a**[*n*] is the same sequence of values as **$1, $2 ..., $NF**:

```
n = split ($0, a), a[1],
```

The function **sprintf**(*f, e1, e2, ...*) produces the value of expressions *e1, e2, ...* in the format specified by the string value of the expression *f*. The format control conventions are those of the **printf**(3S) statement in the C language, except that the use of the asterisk, *, for field width or precision is not allowed.

The function **substr**(*string, pos*) returns the suffix of *string* starting at position *pos*. The function **substr**(*string, pos, length*) returns the substring of *string* that begins at position *pos* and is *length* characters long. If *pos* + *length* is greater than the length of *string*, then **substr**(*string, pos, length*) is equivalent to **substr**(*string, pos*). For example:

```
substr("abc", 2, 1) = "b"
substr("abc", 2, 2) = "bc"
substr("abc", 2, 3) = "bc"
```

If *position* is less than 1, 1 is used. A negative or zero *length* produces a null result. The preferred value of **sprintf** and **substr** is string. The preferred value of the remaining string functions is numeric.

3.1.5 Terms

Various arithmetic operators are applied to primary expressions to produce larger syntactic units called terms. All arithmetic is done in floating-point. A term has one of the following forms:

primary expression
term1 binop term2
unop term
incremented variable
(term)

Binary Terms

Binary terms are of the following form:

term1 binop term2

binop can be one of the five binary arithmetic operators: + (addition), – (subtraction), * (multiplication), / (division), and % (modulus). The binary operator is applied to the numeric value of the operands *term1* and *term2*, and the result is the numeric value. This numeric value is the preferred value, but it can be interpreted as a string value (see "Numeric Constants" earlier in this chapter). The operators *, /, and % have higher precedence than + and –. All operators are left-associative.

Unary Term

Unary terms are of the following form:

unop term

unop can be unary + or –. The unary operator is applied to the numeric value of *term*, and the result is the numeric value which is preferred. However, it can be interpreted as a string value. Unary + and – have higher precedence than *, /, and %.

Incremented Variables

An incremented variable has one of the following forms:

++ *var*
– – *var*
var ++
var – –

Table 3-8 shows the incremented variables, their value, and the effect of the incremented variable. The preferred value of an incremented variable is numeric.

Table 3-8
Incremented Variables

Form	Value	Effect
++ *var*	*var + 1*	*var = var + 1*
– – *var*	*var – 1*	*var = var – 1*
var ++	*var*	*var = var + 1*
var – –	*var = var – 1*	*var = var – 1*

Terms Enclosed in Parentheses

Parentheses are used to group terms in the usual manner.

3.1.6 Expressions

An **awk** expression is one of the following forms:

term
term1 term2 ...
var asgnop expression

Concatenation of Terms

In an expression of the form *term1 term2 ...*, the string value of the terms are concatenated. The preferred value of the resulting expression is a string value. Concatenation of terms has lower precedence than binary **+** and **–**. This example has the string (and numeric) value of 37:

```
1+2 3+4
```

Assignment Statements

Assignment statements have the form:

var asgnop expression
asgnop is one of the following assignment operators:

```
=
+=
−=
*=
=
%=
```

The preferred value of *var* is the same as that of *expression*.

In a statement of the form *var* = *expression*, the numeric and string values of *var* become those of *expression*. The form *var op* = *expression* is equivalent to *var* = *var op expression*, where *op* is one of the following operators:

+ (addition),
– (subtraction),
* (multiplication),
/ (division), and
% (modulus).

The *asgnops* are right-associative and have the lowest precedence of any operator. Thus, a += b *= c–2 is equivalent to the sequence of assignments

```
b = b * (c-2)
a = a + b
```

3.2 Input and Output

Input to **awk** can come from one or more files, or from standard input. Output can go to standard output, to a printer, to one or more files, or be piped to another program. For more information on handing input to **awk**, refer to "Running **awk**" earlier in this chapter.

3.2.1 Input Records and Fields

awk reads its input one record at a time. Unless changed by you, a record is a sequence of characters from the input ending with a newline character or with an end-of-file. **awk** reads in characters until it encounters a newline or end-of-file and assigns the string of characters to the variable **$0**.

Once **awk** has read in a record, it then views the record as being made up of fields. Unless you change it, a field is a string of characters separated by blanks or tabs.

3.2.2 Sample Input File, countries

For use as an example, the file *countries*, shown in Figure 3-1, has been created. The file contains the area in thousands of square miles, the population in millions, and the continent for the ten largest countries in the world. (Figures are from 1978; Russia is placed in Asia.)

```
Russia      8650      262      Asia
Canada      3852       24      North America
China       3692      866      Asia
USA         3615      219      North America
Brazil      3286      116      South America
Australia   2968       14      Australia
India       1269      637      Asia
Argentina   1072       26      South America
Sudan        968       19      Africa
Algeria      920       18      Africa
```

Figure 3-9. Sample Input File, *countries*.

The wide spaces are tabs in the original input and a single blank separates North and South from America. We use this data as the input for many of the **awk** programs in this chapter since it is typical of the type of material that **awk** is best at processing (a mixture of words and numbers arranged in fields or columns separated by blanks and tabs).

Each of the lines in the file has either four or five fields, depending on whether blanks, or blanks and tabs, are defined as field separators. For example, if blanks only are used as field separators, the lines containing North America and South America have five fields. If tabs only are defined as field separators, North America and South America are each considered to be one field. In the example, the first record is

```
Russia  8650    262     Asia
```

When this record is read by **awk**, it is assigned to the variable **$0**. If you want to refer to this entire record, do it through the variable, **$0**. For example, the following action prints the entire record:

```
{print $0}
```

Fields within a record are assigned to the variables **$1**, **$2**, **$3**, and so forth; that is, the first field of the present record is referred to as **$1** by the **awk** program. The second field of the present record is referred to as **$2** by the **awk** program. The ith field of the present record is referred to as **$i** by the **awk** program. Thus, in the file *countries*, the field values of the first record are as follows:

$1 is equal to the string "Russia"
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string "Asia"
$5 is equal to the null string

To print the continent, followed by the name of the country, followed by its population, use the following command:

```
awk '{print $4, $1, $3}' countries
```

You'll notice that this does not produce exactly the output you wanted because the field separator defaults to white space (tabs or blanks). *North America* and *South America* inconveniently contain a blank. The **–F** option allows you to designate only tabs as field separators so that the blanks in *North America* and *South America* are not used as field separators. This command line prints the desired results:

```
awk -F\t '{print $4, $1, $3}' countries
```

3.2.3 Input From the Command Line

As shown above, under "Presenting Your Program for Processing," you can give your program to **awk** for processing by either including it on the command line enclosed by single quotes, or by putting it in a file and naming the file on the command line (preceded by the **–f** flag). It is also possible to set variables from the command line.

Values may be assigned to variables from within an **awk** program. Variable types are not declared: a variable is created simply by referring to it. The following is an example of assigning a value to a variable:

```
x=5
```

This statement in an **awk** program assigns the value 5 to the variable *x*. This type of assignment can be done from the command line. This provides another way to supply input values to **awk** programs. For example, the following command prints the value **5** to *stdout*:

```
awk ' {print x }' x=5 -
```

The minus sign at the end of this command is necessary to indicate that input is coming from *stdin* instead of a file called **x=5**. After entering the command, the user must proceed to enter input. The input is terminated with a Ctrl-D.

If the input comes from a file (*file1* in the example), the command is this:

```
awk '{print x}' file1
```

It is not possible to assign values to variables used in the **BEGIN** section in this way.

If it is necessary to change the record separator and the field separator, it is useful to do so from the command line as in the following example:

```
awk -f awkprog RS=":" file1
```

Here, the record separator is changed to the colon (:) character. This causes your program in the file *awkprog* to run with records separated by the colon instead of the newline character and with input coming from *file1*. It is similarly useful to change the field separator from the command line.

The separate option, **–F***x*, that is placed directly after the command **awk** can also change the field separator from white space to the character *x*. Consider this example:

```
awk -F:  -f awkprog file1
```

This command changes the field separator, **FS**, to the colon (:) character.

Note that if the field separator is specifically set to a tab (that is, with the **-F** option or by making a direct assignment to **FS**), then blanks are not recognized by **awk** as separating fields. However, the reverse is not true. Even if the field separator is specifically set to a blank, tabs are still recognized by **awk** as separating fields.

3.2.4 Output Printing

An action may have no pattern; in this case, the action is executed for all lines as in this simple printing program:

```
{print}
```

This is one of the simplest actions performed by **awk**. It prints each line of the input to the output. More useful is to print one or more fields from each line. For instance, using the file *countries* that was used earlier, the following program prints the name of the country and the population:

```
awk '{  print $1, $3 }' countries
```

The output produced is this:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 14
Sudan 19
Algeria 18
```

A semicolon at the end of statements is optional. **awk** accepts the following statements and considers them to be equivalent:

```
{print $1}
{print $1;}
```

If you want to put two **awk** statements on the same line of an **awk** script, the semicolon is necessary. For example, this program prints the number 5:

```
{x=5; print x}
```

Parentheses are optional with the print statement. The following two statements are equivalent:

```
{print $3, $2}
{print ($3, $2)}
```

Items separated by a comma in a **print** statement are separated by the current output field separator (normally spaces, even though the input is separated by tabs) when printed. The output field separator (**OFS**) is another special variable that you can change. (Refer to "Special Variables" later in this chapter.) **print** also prints strings directly from your programs, as with the following **awk** script:

```
{print "hello, world"}
```

As you have already seen, **awk** includes a number of special variables with useful values, for example **FS** and **RS**. Two other special variables are shown in the next example.

NR and **NF** are both integers that contain the number of the present record and the number of fields in the present record, respectively. This statement prints each record number and the number of fields in each record followed by the record itself.

```
{print NR, NF, $0}
```

Using this program on the file *countries* yields the following:

```
1 4 Russia      8650    262     Asia
2 5 Canada      3852    24      North America
3 4 China       3692    866     Asia
4 5 USA         3615    219     North America
5 5 Brazil      3286    116     South America
6 4 Australia   2968    14      Australia
7 4 India       1269    637     Asia
8 5 Argentina   1072    26      South America
9 4 Sudan       968     19      Africa
10 4 Algeria    920     18      Africa
```

The following program prints the record number and the first field of each record:

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

Using this program on the file *countries* yields the following:

```
1 Russia
2 Canada
3 China
4 USA
5 Brazil
6 Australia
7 India
8 Argentina
9 Sudan
10 Algeria
```

This is an easy way to supply sequence numbers to a list. **print**, by itself, prints the input record, using the default format **%.6g** for each numeric variable printed. To print an empty line, type the following:

```
{print ""}
```

awk also provides the **printf** statement so that you can format output as desired. **printf** formats the expressions in the list according to the specification in the string *format*, and prints them.

printf "*format*", *expr*, *expr*, ...

The *format* statement is almost identical to that of **printf**(3S) in the C library. The following example prints **$1** as a string of 10 characters (right justified):

```
{ printf "%10s %6d %6d\n", $1, $2, $3 }
```

The second and third fields (6-digit numbers) make a neatly columned table:

```
   Russia   8650  262
   Canada   3852  244
    China   3692  866
      USA   3615  219
   Brazil   3286  116
Australia   2968  14
    India   1269  637
Argentina   1072  26
    Sudan   968   19
  Algeria   920   18
```

With **printf**, no output separators or newlines are produced automatically—you must add them as in this example. The escape characters **\n** (newline), **\t** (tab), **\b** (backspace), and **\r** (carriage return) may be specified.

There is a third way that printing can occur on standard output when a pattern without an action is specified. In this case, the entire record, **$0**, is printed. For example, the following program prints any record that contains the character **x**.

```
/x/
```

There are two special variables that go with printing, **OFS** and **ORS**. By default, these are set to blank and the newline character, respectively. The variable **OFS** is printed on the standard output when a comma occurs in a **print** statement such as the following:

```
{ x="hello"; y="world"
print x,y
}
```

The statement above prints the following:

```
hello world
```

However, without the comma in the print statement the spacing is incorrect:

```
{ x="hello"; y="world"
print x y
}
```

The statement above prints the following:

```
helloworld
```

To get a comma in the output, you can either insert it in the print statement or you can change **OFS** in a **BEGIN** section. The following statements produce a comma in the output:

```
{ x="hello"; y="world"
print x"," y
}
```

or

```
BEGIN {OFS=", "}
{ x="hello"; y="world"
print x, y
}
```

Both of these last two scripts yield `hello, world`. Note that the output field separator is not used when **$0** is printed.

3.2.5 Output to Files

The operating system shell allows you to redirect standard output to a file. **awk** also lets you direct output to many different files from within your **awk** program. For example, with your input file *countries*, you want to print all the data from countries of Asia in a file called *ASIA*, all the data from countries in Africa in a file called *AFRICA*, and so forth. This is done with the following **awk** program:

```
{ if ($4 == "Asia") print > "ASIA"
  if ($4 == "Europe") print > "EUROPE"
  if ($4 == "North") print > "NORTH_AMERICA"
  if ($4 == "South") print > "SOUTH_AMERICA"
  if ($4 == "Australia") print > "AUSTRALIA"
  if ($4 == "Africa") print > "AFRICA"
}
```

Control flow statements are discussed in “Flow of Control” later in this chapter.

In general, you may direct output into a file after a **print** or a **printf** statement by using a statement of the following form:

`print > "`*filename*`"`

filename is the name of the file receiving the data. The **print** statement may have any legal arguments to it.

Notice that the filename is quoted. Without quotes, filenames are treated as uninitialized variables and all output then goes to *stdout*, unless redirected on the command line.

If **>** is replaced by **>>**, output is appended to the file rather than overwriting it. A maximum of ten files can be written in this way.

3.2.6 Output to Pipes

It is also possible to direct printing into a pipe instead of a file. In the following example, *mary* is a person's login name. Any record with the second field equal to *XX* is sent to the user, *mary*, as mail;

```
{
if ($2 == "XX") print | "mailx mary"
}
```

awk executes the entire program before it executes the command after a pipe; in the case above, the **mailx**(1) command. The following example takes the first field of each input record, sorts these fields, and then prints them:

```
{
print $1 | "sort"
}
```

The following example guarantees that the output from **print** always goes to your terminal:

```
{
print ... | "cat -v > /dev/tty"
}
```

Only one output statement to a pipe is permitted in an **awk** program. In all output statements involving redirection of output, the files or pipes are identified by their names, but they are created and opened only once in the entire run.

3.3 Patterns

A pattern in front of an action acts as a selector that determines if the action is to be executed. A variety of expressions are used as patterns:

- Certain keywords
- Arithmetic relational expressions
- Regular expressions
- Combinations of these

3.3.1 BEGIN and END

The keyword, **BEGIN**, is a special pattern that matches the beginning of the input before the first record is read. The keyword, **END**, is a special pattern that matches the end of the input after the last line is processed. **BEGIN** and **END** thus provide a way to gain control before and after processing for initialization and wrapping up.

As you have seen, you can use **BEGIN** to put column headings on the output. The following line puts a heading on the population table:

```
BEGIN {print "Country", "Area", "Population", "Continent"}
        {print}
```

The output looks like this:

```
Country Area Population Continent

Russia 8650    262 Asia
Canada 3852    24  North America
China  3692    866 Asia
USA 3615    219 North America
Brazil 3286    116 South America
Australia 2968    14  Australia
India  1269    637 Asia
Argentina 1072    26  South America
Sudan  968 19  Africa
Algeria    920 18  Africa
```

Formatting is not very good here; **printf** would do a better job and is generally used when appearance is important.

Recall also, that the **BEGIN** section is a good place to change special variables such as **FS** or **RS**. For example:

```
BEGIN { FS= "\t"
    printf "Country\t\t  Area\tPopulation\tContinent\n\n"}
    {printf "%-10s\t%6d\t%6d\t\t% -14s\n", $1, $2, $3, $4}
END {print "The number of records is", NR}
```

In this program, **FS** is set to a tab in the **BEGIN** section and as a result all records in the file *countries* have exactly four fields. Note that if **BEGIN** is present, it is the first pattern; **END** must be last if it is used.

3.3.2 Relational Expressions

An **awk** pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population, use

```
$3 > 100
```

This **awk** program is a pattern without an action, so it prints each line whose third field is greater than 100 as follows:

```
Russia   8650   262   Asia
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
India    1269   637   Asia
```

To print the names of the countries that are in Asia, use the following program:

```
$4 == "Asia" {print $1}
```

The output produced is this:

```
Russia
China
India
```

The conditions tested are **<** , **<=** , **==**, **!=**, **>=**, and **>**. In such relational tests if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings. The following statement selects lines that begin with *S*, *T*, *U*, and greater:

```
$1 >= "S"
```

In the file *countries*, these are as follows;

```
USA      3615     219      North America
Sudan    968      19       Africa
```

In the absence of other information, fields are treated as strings, so the following program compares the first and fourth fields as strings of characters:

```
$1 == $4
```

The following single line is printed:

```
Australia        2968       14 Australia
```

3.3.3 Regular Expressions

awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are called *regular expressions*. The simplest regular expression is a literal string of characters enclosed in slashes:

```
/Asia/
```

This is a complete **awk** program that prints all lines that contain any occurrence of the name *Asia*. If a line contains *Asia* as part of a larger word like *Asiatic*, it is also printed (but there are no such words in the *countries* file.)

awk regular expressions include regular expression forms found in the text editor, **ed**(1), and the pattern finder, **grep**(1), in which certain characters have special meanings.

For example, you could print all lines that begin with *A*:

```
/^A/
```

You could print all lines that begin with *A*, *B*, or *C*:

```
/^[ABC]/
```

Or, you could print all lines that end with *ia*:

```
/ia$/
```

In general, the circumflex (^) indicates the beginning of a line. The dollar sign ($) indicates the end of the line, and characters enclosed in brackets ([ ]) match any one of the characters enclosed. In addition, **awk** allows parentheses for grouping, the pipe (|) for alternatives, a plus sign (+) for one or more occurrences, and a question mark (?) for zero or one occurrences. This example prints all records that contain either an *x* or a *y*:

```
/x|y/  {print}
```

This example prints all records that contain an *a* followed by one or more *x*'s followed by a *b*, for example *axb, Paxxxxxxxb, QaxxbR*:

```
/ax+b/ {print}
```

The following example prints all records that contain an *a* followed by zero or one *x* followed by a *b*, for example *ab, axb, yaxbPPP, or CabD*:

```
/ax?b/ {print}
```

The two characters, period (.), and asterisk (*) have the same meaning as they have in **ed**(1), namely, a period can stand for any character and an asterisk means zero or more occurrences of the character preceding it. The following example matches any record that contains an *a* followed by any character followed by a *b*.

```
/a.b/
```

That is, the record must contain an *a* and a *b* separated by exactly one character. For example, */a.b/* matches *axb*, *aPb* and *xxxxaXbxx*, but not *ab* or *axxb*.

The following example matches a record that contains an *a* followed by zero or more *x*'s followed by a *c*. For example, it matches *ac, axc*, or *pqraxxxxxxxxxxc901*:

```
/ax*c/
```

Just as in **ed**(1), it is possible to turn off the special meaning of metacharacters such as ^ and * by preceding these characters with a backslash. An example of this is the following pattern which matches any string of characters enclosed in slashes:

```
/\/*\//
```

You can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, using the input file *countries* as before, the following program prints all countries whose name ends in *ia*, which is different from lines that end in *ia*:

```
$1 ~ /ia$/        {print $1}

   Russia
   Australia
   India
   Algeria
```

3.3.4 Combinations of Patterns

A pattern can be made up of similar patterns combined with the operators **||** (OR), **&&** (AND), **!** (NOT), and parentheses. This example selects lines where the area (**$2**) is equal to or greater than 3000 square miles and the population (**$3**) is equal to or greater than 100 million:

```
$2 >= 3000 && $3 >= 100
```

The following output is produced:

```
Russia   8650   262   Asia
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

This example selects lines with *Asia* or *Africa* as the fourth field:

```
$4 == "Asia" || $4 == "Africa"
```

An alternative way to write this last expression is with a regular expression which selects records where the fourth field matches *Africa* or begins with *Asia*:

```
$4 ~ /^Asia|Africa)$/
```

&& and **||** guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

3.3.5 Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma:

pattern1, *pattern2*   { *action* }

In this case, the *action* is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* inclusive. As an example with no action, the following pattern prints all lines between the one containing *Canada* and the line containing *Brazil*:

```
/Canada/,/Brazil/
```

The following output is produced;

```
Canada   3852   24    North America
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

This examples does the action for lines 2 through 5 of the input:

```
NR == 2, NR == 5 { ... }
```

Different types of patterns may be mixed, such as in the following example which prints all lines from the first line containing *Canada* up to and including the next record whose fourth field is *Africa*.

```
/Canada/, $4 == "Africa"
```

NOTE

This discussion of pattern matching pertains to the pattern portion of the pattern/action **awk** *statement. Pattern matching can also take place inside an* **if** *or* **while** *statement in the action portion. See "Flow of Control", later in this chapter.*

3.4 Actions

An **awk** action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

3.4.1 Variables, Expressions, and Assignments

awk provides the ability to do arithmetic and to store the results in variables for later use in the program. As an example, consider printing the population density for each country in the file *countries*.

```
{print $1, (1000000 * $3) / ($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile.

```
Russia 30.289
Canada 6.23053
China 234.561
```

```
USA 60.5809
Brazil 35.3013
Australia 4.71698
India 501.97
Argentina 24.2537
Sudan 19.6281
Algeria 19.5652
```

The formatting is not good; using **printf** instead gives the output a better look. The following statement produces better output:

```
{printf "%10s %6.1f\n", $1, (1000000 * $3) / ($2 * 1000)}
```

The output produced is this:

```
    Russia   30.3
    Canada    6.2
     China  234.6
       USA   60.6
    Brazil   35.3
 Australia    4.7
     India  502.0
 Argentina   24.3
     Sudan   19.6
   Algeria   19.6
```

Arithmetic is done internally in floating-point. The arithmetic operators are **+**, **–**, *****, **/**, and **%** (modulus).

To compute the total population and number of countries from Asia, you could write the following statement:

```
/Asia/  { pop += $3; ++n }
END {print "total population of", n, "Asian countries is", pop }
```

The output produced is:

```
total population of 3 Asian countries is 1765.
```

The operators **++**, **– –**, **–=**, **/=**, *** =**, **+=**, and **%=** are available in **awk** as they are in C. The **++** operator, for example, adds one to the value of a variable. The increment and decrement operators, **++** and **– –**, are used as in the C language. These operators are also used in expressions.

3.4.2 Initialization of Variables

In the previous example, you did not initialize *pop* or *n*; yet everything worked properly. This is because (by default) variables are initialized to a null string, which has a numerical value of 0. This eliminates the need for most initialization of variables in **BEGIN** sections. We can use default initialization to advantage in this program, which finds the country with the largest population. This program produces the output `China 866`:

```
maxpop < $3 {
        maxpop = $3
        country = $1
        }
END     {print country, maxpop}
```

3.4.3 Field Variables

Fields in **awk** share the properties of variables. They are used in arithmetic and string operations, may be initialized to a null string, or have other values assigned to them. For example, divide the second field by 1000 to convert the area to millions of square miles with

```
{ $2 /= 1000; print }
```

or process two fields into a third with

```
BEGIN    { FS = "\t" }
         { $4 = 1000 * $3 / $2; print }
```

or assign strings to a field as in

```
/USA/  { $1 = "United States" ; print }
```

which replaces **USA** by **United States** and prints the affected line:

```
United States 3615 219 North America
```

Fields are accessed by expressions, thus **$NF** is the last field and **$(NF – 1)** is the second to the last. Note that the parentheses are needed since **$NF – 1** is 1 less than the value in the last field.

3.4.4 String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{ x = "hello"
  x = x ", world"
  print x
}
```

This results in the usual output:

```
hello, world
```

With input from the file *countries*, the following program prints countries beginning with *A*.

```
/A/         { s = s " " $1 }
END         { print s }
```

```
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

3.4.5 Special Variables

Some variables in **awk** have special meanings. These are detailed here.

NR	Number of the current record.
NF	Number of fields in the current record.
FS	Input field separator (by default it is set to a blank or tab).
RS	Input record separator (by default it is set to the newline character).
$i	The ith input field of the current record.
$0	The entire current input record.
OFS	Output field separator (by default it is set to a blank).

ORS	Output record separator (by default it is set to the newline character).
OFMT	The format for printing numbers, with the print statement, is **%.6g** by default.
FILENAME	The name of the input file currently being read. This is useful because **awk** commands are typically of this form: **awk -f** *program file1 file2 file3 ...*

3.4.6 Type

Variables (and fields) take on numeric or string values according to context. In this example, *pop* is presumably a number:

```
pop += $3
```

In this example *country* is a string.

```
country = $1
```

In the following example, the type of *maxpop* depends on the data found in **$3**. It is determined when the program is run:

```
maxpop < $3
```

In general, each variable and field is potentially a string or a number, or both at any time. When a variable is set by the assignment such as `v = expr`, its type is set to that of *expr*. (Assignment also includes **+=**, **++**, **-=**, and so forth.) An arithmetic expression is of the type **number**; a concatenation of strings is of type **string**. If the assignment is a simple copy as in `v1 = v2` then the type of *v1* becomes that of *v2*.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression may be coerced to numeric with `expr + 0` and to string with `expr ""`. This last expression is **string** concatenated with a null string.

3.4.7 Arrays

As well as ordinary variables, **awk** provides one-dimensional arrays. Array elements are not declared; they come into existence by being mentioned. Subscripts may have any non-null value including non-numeric strings. As an example of a conventional numeric subscript, the following statement assigns the current input line to the **NR**th element of the array *x*:

```
x[NR] = $0
```

In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following **awk** program:

```
         { x[NR] = $0 }
END      { ... program ... }
```

The first line of this program records each input line into the array *x*. When run on the file *countries*, the following program produces an array of elements:

```
{ x[NR] = $1}
```

The output looks like this:

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
         ...  and so forth.
```

Arrays are also indexed by non-numeric values that give **awk** a capability that resembles the associative memory of Snobol tables. For example, the following program prints the specified country and its population:

```
/Asia/{pop["Asia"] += $3}
/Africa/{pop[Africa] += $3}
END      {print "Asia=" pop["Asia"], "Africa="pop["Africa"] }
```

The output looks like this:

```
Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus, the following statement uses the first field of a line (as a string) to index the array *area*:

```
area[$1] = $2
```

3.5 Special Features

The final section describes the use of some special **awk** features.

3.5.1 Built-In Functions

The function **length** is provided by **awk** to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0 }
```

In this case the variable **length** means **length($0)**, the length of the present record. In general, **length**(x) returns the length of x as a string.

With input from the file *countries*, the following **awk** program prints the longest country name:

```
length($1) > max  {max = length($1); name = $1 }
END               {print name}
```

The function **split**(*s, array*) assigns the fields of the string *s* to successive elements of the array, *array*.

The following example assigns the value *Now* to w[1], *is* to w[2], *the* to w[3], and *time* to w[4].

```
split("Now is the time", w)
```

All other elements of the array w[ ], if any, are set to a null string.

It is possible to have a character other than a blank as the separator for the elements of *w*. For this, use **split** with three elements as follows:

```
n = split(s, array, sep)
```

This splits the string *s* into *array[1], ..., array[n]*. The number of elements found is returned as the value of **split**. If the *sep* argument is present, its first character is used as the field separator; otherwise, **FS** is used. This is useful if, in the middle of an **awk** script, it is necessary to change the record separator for one record. **awk** also provides the following math functions:

sqrt
log
exp
int

They provide the square root function, the base **e** logarithm function, the exponential function, and integral part function. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C math library (**int** corresponds to the **libm floor** function) and so they have the same return on error as those in **libm**. (Refer to the *Programmer's Reference Manual.*)

The function **substr**(*s,m,n*) produces the substring of *s* that begins at position *m* and is at most *n* characters long. If the third argument (*n* in this case) is omitted, the substring goes to the end of *s*. For example, you could abbreviate the country names in the file *countries* with the following statement:

```
{ $1 = substr($1, 1, 3); print }
```

The output looks like this:

```
Rus   8650   262   Asia
Can   3852    24   North America
Chi   3692   866   Asia
USA   3615   219   North America
Bra   3286   116   South America
Aus   2968    14   Australia
Ind   1269   637   Asia
Arg   1072    26   South America
Sud    968    19   Africa
Alg    920    18   Africa
```

If *s* is a number, **substr** uses its printed image:

```
substr(123456789,3,4)=3456.
```

The function **index**(*s1,s2*) returns the leftmost position where the string *s2* occurs in *s1* or zero if *s2* does not occur in *s1*.

The function **sprintf** formats expressions as the **printf** statement does but assigns the resulting expression to a variable instead of sending the results to *stdout*. This example sets *x* to the string produced by formatting the values of **$1** and **$2**:

```
x = sprintf("%10s %6d", $1, $2)
```

The *x* may then be used in subsequent computations.

The function **getline** immediately reads the next input record. Fields **NR** and **$0** are set but control is left at exactly the same spot in the **awk** program. **getline** returns 0 for the end of file and a 1 for a normal record.

3.5.2 Flow of Control

awk provides the following basic control statements within actions with statement grouping as in the C language:

if-else
while
for

The **if** statement is used as follows:

if (*condition*) *statement1* **else** *statement2*

The *condition* is evaluated and, if it is true, *statement1* is executed; otherwise, *statement2* is executed. The **else** part is optional. Several statements enclosed in braces, { }, are treated as a single statement. Rewriting the maximum population computation from the pattern section with an **if** statement results in the following:

```
{       if (maxpop < $3) {
                maxpop = $3
                country = $1
        }
}
END     { print country, maxpop }
```

The **while** statement is used as follows:

while (*condition*) *statement*

The *condition* is evaluated; if it is true, the *statement* is executed. The *condition* is evaluated again and, if true, the *statement* is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields, one per line:

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

Another example is the Euclidean algorithm for finding the greatest common divisor of **$1** and **$2**:

```
{printf "the greatest common divisor of " $1 "and ", $2, "is"
while ($1 != $2) {
   if ($1 > $2) $1 -= $2
   else      $2 -= $1
}
printf $1 "\n"
}
```

The **for** statement is like that of C, as follows:

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

This is another **awk** program that prints all input fields, one per line:

```
{        for (i = 1 ; i <= NF; i++)
                 print $i
}
```

There is an alternative form of the **for** statement that is useful for accessing the elements of an associative array in **awk**, as follows:

for (*i in array*) *statement*

This form executes *statement* with the variable *i* set in turn to each subscript of *array*. The subscripts are each accessed once but in undefined order. Chaos ensues if the variable *i* is altered or if any new elements are created within the loop. For example, you could use the **for** statement to print the record number followed by the record of all input records after the main program is executed.

```
        { x[NR] = $0 }
END     { for(i in x) print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
BEGIN   {FS="\t"}
        {population[$4] += $3}
END     {for(i in population)
                print i, population[i]
   }
```

In this program, the body of the *for* loop is executed for *i* equal to the string *Asia*, then for *i* equal to the string *North America*, and so forth until all the possible values of *i* are exhausted (that is, until all the strings of names of countries are used). Note, however, the order in which the loops are executed is not specified. If the loop associated with *Canada* is executed before the loop associated with the string *Russia*, such a program produces this:

```
South America 26
Africa 16
Asia 637
Australia 14
North America 219
```

Note that the expression in the condition part of an **if, while**, or, **for** statement can include the following:

- Relational operators like **<**, **<=**, **>**, **>=**, **==**, and **!=**
- Regular expressions that are used with the matching operators   and **!**
- Logical operators **||**, **&&**, and **!**
- Parentheses for grouping

The **break** statement (when it occurs within a **while** or **for** loop) causes an immediate exit from the **while** or **for** loop.

The **continue** statement, when it occurs within a **while** or **for** loop, causes the next iteration of the loop to begin.

The **next** statement in an **awk** program causes **awk** to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between **getline** and **next**: **getline** does not skip to the top of the **awk** program.)

If an **exit** statement occurs in the **BEGIN** section of an **awk** program, the program stops executing and the **END** section, if there is one, is not executed.

An **exit** that occurs in the main body of an **awk** program causes execution of the main body of that program to stop. No more records are read and the **END** section is executed.

An **exit** in the **END** section causes execution to terminate at that point.

3.5.3 Report Generation

The flow of control statements in the last section are especially useful when **awk** is used as a report generator. **awk** is useful for tabulating, summarizing, and formatting information. We have seen an example of **awk** tabulating populations in the last section. Here is another example. Suppose you have a file *prog.usage* that contains lines of three fields: name, program, and usage:

```
Smith   draw   3
Brown   eqn    1
Jones   nroff  4
Smith   nroff  1
Jones   spell  5
Brown   spell  9
Smith   draw   6
```

The first line indicates that Smith used the **draw** program three times. If you want to create a program that has the total usage of each program along with the names in alphabetical order and the total usage, you could use the following program, called *list1*:

```
        {use[$1 "\t" $2] += $3}
  END   {for (np in use)
              print np "        " use[np]  | "sort +0 +2nr"
     }
```

This program produces the following output when used on the input file, *prog.usage*:

```
Brown   eqn    1
Brown   spell  9
Jones   nroff  4
Jones   spell  5
Smith   draw   9
Smith   nroff  1
```

To format the previous output so that each name is printed only once, pipe the output of the previous **awk** program into the following program, called *format1*:

```
{         if ($1 != prev) {
                  print $1 ":"
                  prev = $1
          }
          print " " $2 " " $3
}
```

The variable **prev** is used to ensure each unique value of **$1** prints only once. The following command combines the program usage for each person:

```
awk -f list1 prog.usage | awk -f format1
```

The output is this:

```
Brown:
          eqn      1
          spell    9
Jones:
          nroff    4
          spell    5
Smith:
          draw     9
          nroff    1
```

It is often useful to combine different **awk** scripts with other shell commands such as **sort**(1), as was done in the *list1* program above.

3.5.4 Cooperation with the Shell

Normally, an **awk** program is either contained in a file or enclosed within single quotes:

```
awk '{print $1}' ...
```

Since **awk** uses many of the same characters the shell does (such as the dollar sign and double quote), surrounding the program by single quotes ensures that the shell passes the program to **awk** intact.

Consider writing an **awk** program to print the *n*th field, where *n* is a parameter determined when the program is run. That is, you want a program called *field* such that

```
field n
```

runs the **awk** program

```
awk '{print $n}'
```

There are several ways to get the value of *n* into the **awk** program. One is to define *field* as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written there is only one argument, even though there are two sets of quotes. The **$1** is outside the quotes, visible to the shell, and therefore substituted properly when *field* is invoked.

Another method relies on the fact that the shell substitutes for **$** parameters within double quotes.

```
awk "{print \$ $1}"
```

Here the trick is to protect the first **$** with a backslash; the **$1** is again replaced by the number when *field* is invoked.

3.5.5 Multidimensional Arrays

You can simulate the effect of multidimensional arrays by creating your own subscripts. For example:

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
                mult[i "," j] = . . .
```

creates an array whose subscripts have the form *i,j*; that is, 1,1; 1,2; and so forth; and thus simulate a two-dimensional array.

Index

4. lex

4. lex

Chapter 4
lex

Chapter 4 *lex*

4.1 Introduction

lex is a software tool that helps you solve problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you can check the spelling of words for errors; in code enciphering, you can translate certain patterns of characters into others; and in compiler writing, you can identify the tokens (smallest meaningful sequences of characters) in the program to be compiled. The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. In the case of compiler writing, creating the ability to solve the problem requires implementing the compiler's lexical analyzer. Hence the name **lex**.

It is not essential to use **lex** to handle problems of this kind. You could also write programs in a standard language such as C to handle them. **lex** produces such C programs and is therefore called a *program generator*. **lex** offers a faster, easier way to create programs that perform these tasks. Its weakness is that the C programs it creates that are often longer than necessary for the task at hand and execute more slowly than they otherwise might. In many applications, this is a minor consideration, and the advantages of using **lex** considerably outweigh it.

To understand what **lex** does, refer to the diagram in Figure 4-1. The **lex** source, often called the **lex** specification, consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the **lex** program generator. The output of the program generator is a C program (**lex** analyzer in C) that is compiled by a C compiler to generate the executable object program (**lex** analyzer program **a.out**) that performs the lexical analysis.

Finally, **a.out** takes as input any source file and produces the desired output, such as altered text or a list of tokens.

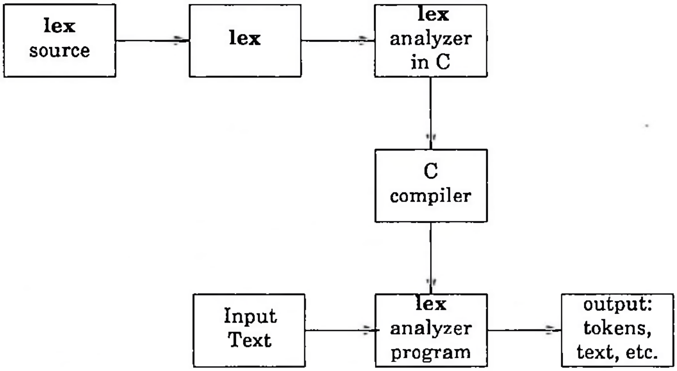


Figure 4-1. Creation and use of a lexical analyzer with lex.

lex can also be used to collect statistical data from the input, such as character count, word length, and number of occurrences of a word.

This chapter describes how to perform the following tasks:

- Write **lex** source.
- Translate **lex** source.
- Compile, link, and execute the lexical analyzer in C.
- Run the lexical analyzer program.

4.2 Writing lex Programs

A **lex** specification can consist of three sections: definitions, rules, and user subroutines. The rules section is mandatory. Definitions and user subroutines are optional, but must appear in the indicated order if they are present. When all sections are present, a full specification file looks like this:

```
definitions
%%
rules
%%
subroutines
```

4.2.1 The Fundamentals of lex Rules

The rules section opens with the delimiter *%%*. If a subroutine section follows, another *%%* delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program. Because the definition and subroutine sections are optional, the smallest **lex** program specification is as follows:

```
%%
```

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term *specification*. It can mean either the entire **lex** source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, **lex** writes out the input exactly as it finds it. So, the simplest **lex** program consists only of the beginning rules delimiter: *%%*. It writes out the entire input with no changes at all. Typically, the rules are more elaborate than that.

Specifications

The patterns that you are interested in are specified with a notation called *regular expressions*. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all.

```
apple
orange
pluto
```

These three regular expressions match any occurrences of those character strings in an input text. If you want to have your lexical analyzer, **a.out**, remove every occurrence of *orange* from the input text, specify the following rule. The semicolon indicates the end of a line.

orange;

Because you did not specify an action before the semicolon, **lex** does nothing but print out the original input text with every occurrence of this regular expression removed (that is, without any occurrence of the string *orange* at all).

Unlike the *orange* example, most expressions to be searched for cannot be specified so easily. The expression itself might simply be too long.

More commonly, the class of desired expressions is too large; it can, in fact, be infinite. Using operators, regular expressions can be formed signifying any expression of a certain class. The operators are as follows:

+	One or more occurrences of the preceding expression.
?	Zero on one occurrence(s) of the preceding expression (this is equivalent to saying that the preceding expression is optional).
(*)	Zero or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. An example is shown later.)

For example, **m+** is a regular expression matching any string of *m*'s:

```
mmm
m
mmmmm
mm
```

And **7*** is a regular expression matching any string of zero or more 7's:

```
77
77777

777
```

The string of blanks on the third line matches simply because it has no 7's in it at all.

Brackets ([ ]) indicate any one character from the string of characters specified between the brackets. Thus, **[dgka]** matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma here would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen (–). The sequence **[a–z]**, for instance, indicates any lowercase letter.

The following regular expression matches any letter (whether uppercase or lowercase), any digit, an asterisk, an ampersand, or a pound sign.

```
[A-Za-z0-9*&#]
```

Consider the following input text. A lexical analyzer with the previous specification in one of its rules recognizes the *, &, r, and # characters. On recognition of these characters, the lex analyzer performs whatever action the rule specifies (no action is specified here); and prints the rest of the text as it stands.

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

The operators are powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is as follows:

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by an asterisk, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the asterisk, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

```
e
pay
distance
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers:

```
not_idenTIFER
5times
$hello
```

This occurs because `not_idenTIFER` has an embedded underscore, `5times` starts with a digit rather than a letter, and `$hello` starts with a special character.

Special characters can be searched for in two ways. The character can be enclosed in single quotes ('), or, as in the C language, a backslash (\) can precede the special character. All C language escapes are recognized. The following are understood by **lex**:

'\n'	newline
'\r'	return
'\''	single quote (´)

'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'\*xxx*'	*xxx* in octal notation

For example, to use the backslash method to recognize an asterisk followed by any number of digits, you can use this pattern:

```
\*[1-9]*
```

Actions

Actions are program fragments in the C language that specify the action to be taken once **lex** recognizes a string matching the regular expression. Actions can do input and output, call subroutines, and alter arrays and variables. When **lex** recognizes a matching string at the start of a rule, it looks to the right of the rule for the action to be performed. An action can consist of as many statements as are needed for the job at hand. If the action is more than one line, enclose it in braces to inform **lex** that the action is for one rule only. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. You might want to print out a message noting that the text has been found or a message transforming the text in some way. For example, the following rule recognizes the expression `Amelia Earhart` and notes this recognition:

```
"Amelia Earhart"   printf("found Amelia");
```

The next example replaces a lengthy medical term with its acronym:

```
Electroencephalogram    printf("EEG");
```

To count the lines in a text, **lex** must recognize end-of-lines and increment a line counter. **lex** uses the standard escape sequences from C such as **\n** for end-of-line. The following example shows a line-counter where **lineno**, like other C variables, is declared in the definitions section of the **lex** source:

```
\n   lineno++;
```

lex stores every character string that it recognizes in a character array called **yytext[]**. The contents of the array can be printed or manipulated with actions. The following example counts the total number of all digit strings in an input text, prints the running total of the number of digit strings (not their sum) and prints out each one as soon as it is found:

```
+?[1-9]+            { digstrngcount++;
                      printf("%d",digstrngcount);
                      printf("%s", yytext);    }
```

This specification matches digit strings whether they are preceded by a plus sign (+) or not, because the question mark (?) indicates that the preceding plus sign is optional. In addition, it catches negative digit strings because that portion following the minus sign (–) matches the specification. The next section explains how to distinguish negative from positive integers.

4.2.2 Advanced lex Usage

lex provides features that let you process input text containing complicated patterns. The features include rules that decide what specification is relevant when more than one seems so at first, functions that transform one matching pattern into another, and the use of definitions and subroutines. Consider the following example:

```
%%
-[0-9]+          printf("negative integer");
+?[0-9]+         printf("positive integer");
-0.[0-9]+        printf("negative fraction, no whole number part");
rail[ ]+road     printf("railroad is one word");
crook            printf("Here's a crook");
function         subprogcount++;
G[a-zA-Z]*      { printf("may have a G word here: %s", yytext);
                   Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and –1. The use of the terminating **+** in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for *railroad* matches cases where one or more blanks intervene between the two syllables of the word. In the cases of *railroad* and *crook*, you may have simply printed a synonym rather than the messages stated. The rule recognizing a *function* simply increments a counter. The last rule illustrates three points:

- The braces specify an action sequence extending over several lines.
- Its action uses the **lex** array **yytext**[ ], which stores the recognized character string.

- Its specification uses the asterisk (*) to indicate that zero or more letters may follow the **G**.

Some Special Features

Besides storing the recognized character string in **yytext**[], **lex** automatically counts the number of characters in a match and stores it in the variable **yyleng**. You can use this variable to refer to any specific character just placed in the array **yytext**[]. Because C numbers locations in an array starting with 0, you could write the following lines to print out the third digit (if there is one) in a just recognized integer.

```
[1-9]+          {if  (yyleng > 2)
                 printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. For example, in the lexical analyzer example developed later in the section on **lex** and **yacc**, the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

NOTE

lex *follows the rule that says: "Where there is a match with two or more rules in a specification, the first rule is the one whose action is executed."*

By placing the rule for **end** and the other reserved words before the rule for identifiers, you ensure that your reserved words are recognized.

Another potential problem arises from cases where a pattern you are searching for is the prefix of another. For instance, the last two rules in the previous lexical analyzer example are designed to recognize > and >= . If the text has the string >= at one point, you might worry that the lexical analyzer would stop as soon as it recognized the > character to execute the rule for >, rather than read the next character and execute the rule for >=.

NOTE

lex *follows the rule that says: "Match the longest character string possible and execute the rule for that."*

Here it would recognize the >= and act accordingly. As a further example, the rule would enable you to distinguish + from ++ in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you have found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the DO statement in FORTRAN. In the following statement, you cannot be sure that the first 1 is the initial value of the index **k** until you read the first comma.

```
DO 50 k = 1 , 20, 1
```

Until then, you might have this assignment statement:

```
DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the slash (/) instead of the backslash (\), which signifies that what follows is trailing context, something not to be stored in **yytext**[], because it is not part of the token itself. So the rule to recognize the FORTRAN DO statement could be as follows:

```
30/[ ]*[0-9][ ]*[a-z A-Z0-9]+=[a-z A-Z0-9]+,    printf("found DO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

lex uses the dollar sign ($) as an operator to mark a special trailing context—the end of line. (It is therefore equivalent to **\n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

```
[  \t]+$        ;
```

If you want to match a pattern only when it starts a line, use the circumflex (^) as the operator. For example, the **nroff** formatter demands that you never start a line with a blank, so you might want to create a rule to check input to **nroff**:

```
^[ ]        printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks:

input()	Read another character
unput(**c**)	Put a character back
output(**c**)	Write a character to output

One way to ignore all characters between two special characters, such as between a pair of double quotation marks, would be to use **input**() as follows:

```
  \"          while (input() != '"');
```

Upon finding the first double quotation mark, **a.out** continues reading all subsequent characters so long as none is a quotation mark, and does not look for a match again until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you can use standard I/O routines in C to rewrite the functions **input**(), **unput**(**c**), and **output**. These and other programmer-defined functions should be placed in the subroutine section of the **lex** source. The new routines replace the standard ones. The standard **input**(), in fact, is equivalent to **getchar**(), and the standard **output**(**c**) is equivalent to **putchar**(**c**).

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. They are described in detail in the following paragraphs and are as follows:

yymore()	Recognizes succeeding characters and appends them to those already in **yytext**.
yyless(*n*)	Resets the end point of a string to be considered the *n*th character in the original **yytext**.
REJECT	Jumps to the next rule without changing the contents of **yytext**.

Recall that the text matching a given specification is stored in the array **yytext**[]. In general, once the action is performed for the specification, the characters in **yytext**[] are overwritten with succeeding characters in the input stream to form the next match. The function **yymore**(), by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext**[]. This lets you do one thing and then another, when one string of characters is significant and a longer one including the first is significant as well. Consider a character string bound by **B**s and containing a **B** at an arbitrary location.

```
B...B...B
```

In a simple code-deciphering situation, you may want to count the number of characters between the first and second **B** and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.)

```
     B[^B]*          { if (flag = 0)
                           save = yyleng;
                           flag = 1;
                           yymore();
                       else      {
                           importantno = save + yyleng;
                           flag = 0; }
                         }
```

flag, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section of the **lex** source. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyless**(*n*) lets you reset the end point of the string to be considered as the *n*th character in the original **yytext**[]. Suppose you are again in deciphering code and want to work with only half the characters in a sequence ending with a certain character, say upper or lowercase **Z**:

```
[a-yA-Y]+[Zz]      {  yyless(yyleng/2);
                   ... process first half of string... }
```

Finally, the function **REJECT** lets you more easily process strings of characters even when they overlap or contain one another as parts. **REJECT** does this by immediately jumping to the next rule and its specification without changing the contents of **yytext**[]. If you want to count the number of occurrences both of the regular expression `snapdragon` and of its subexpression `dragon` in an input text, you could use the following example:

```
     snapdragon       {countflowers++; REJECT;}
     dragon           countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions `comedian` and `diana`, even where the input text has sequences such as `comediana..`:

```
comedian        {comiccount++; REJECT;}
diana           princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, the counters and other necessary variables are declared in the definitions section commencing the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. For **lex** source this section is optional, but in most cases it is necessary. External definitions have the same form and function that they have in C. They declare that variables globally defined elsewhere (perhaps in another source file) are accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later:

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it is accessible in the routine, say a parser, that calls it. If, on the other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself right after the left brace ({).

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Since some variable declarations and **lex** definitions may be needed by more than one **lex** source file, they can be placed in a separate file to be included in every **lex** file that needs them. For example, the file *y.tab.h* should be included when using **lex** with **yacc**. **yacc** generates parsers that call a lexical analyzer that can contain **#defines** for token names. Like the declarations, **#include** statements should come between %{ and }%.

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, place your abbreviations for regular expressions to be used in the rules section that ends your **#include**'s and declarations after the %}. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. Later, when you use abbreviations in your rules, be sure to enclose them in braces.

NOTE

The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times.

```
D             [0-9]
L             [a-zA-Z]
B             [ \t]
%%
-{D}+          printf("negative integer");
+?{D}+         printf("positive integer");
-0.{D}+        printf("negative fraction");
G{L}*          printf("may have a G word here");
rail{B}+road    printf("railroad is one word");
crook          printf("criminal");
 \"\./{B}+     printf(".\"");
   .              .
   .              .
```

The last rule ensures that a period always precedes a quotation mark at the end of a sentence. It changes `example".` to `example."`

Subroutines

You might want to use subroutines in **lex** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function **put_in_tabl**(), to be discussed in the next section on **lex** and **yacc**, is a good candidate for a subroutine.

Another reason to place a routine in the subroutine section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between /* and */.

```
"/*"                    skipcmnts();
.
.                 /* rest of rules */
%%
skipcmnts()
{
        for(;;)
        {
           while (input() != '*');
           if (input() != '/') {
                 unput(yytext[yyleng-1]);
           else return;
        }
 }
```

There are three points of interest in this example. First, the **unput(c)** function (putting back the last character read) is necessary to avoid missing the final slash (/) if the comment ends unusually with a **/ . In this case, eventually having read an asterisk (*), the analyzer finds that the next character is not the terminal slash (/) and must read some more. Second, the expression **yytext[yyleng–1]** picks out that last character read. And third, this routine assumes that the comments are not nested. (This is indeed the case with the C language.) If, unlike C, they are nested in the source text, after **input**()ing the first */ ending the inner group of comments, **a.out** reads the rest of the comments as if they were part of the input to be searched for patterns.

Other examples of subroutines would be programmer-defined versions of the I/O routines **input**(), **unput(c)**, and **output**(), discussed above. Subroutines such as these that can be exploited by many different programs would probably best be stored in their own individual file or library to be called as needed. The appropriate **#include** statements would then be necessary in the definitions section.

4.2.3 Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you might want to use the **yacc** program tool. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in Chapter 5.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read Chapter 5 because it contains information of interest to all **lex** programmers.

The lexical analyzer that **lex** generates (not the file that stores it) takes the name **yylex**(). This name is convenient because **yacc** calls its lexical analyzer by this name. To use **lex** to create the lexical analyzer for the parser of a compiler, you need to end each **lex** action with the statement **return** *token*, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called *y.tab.c* by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >. Consider the following portion of **lex** source for a lexical analyzer for some programming language:

```
begin                          return(BEGIN);
end                            return(END);
while                          return(WHILE);
if                             return(IF);
package                        return(PACKAGE);
reverse                        return(REVERSE);
loop                           return(LOOP);
[a-zA-Z][a-zA-Z0-9]*         { tokval = put_in_tabl();
                                 return(IDENTIFIER); }
[0-9]+                       { tokval = put_in_tabl();
                                 return(INTEGER); }
\+                          { tokval = PLUS;
                                 return(ARITHOP); }
\-                         { tokval = MINUS;
                                 return(ARITHOP); }
>                            { tokval = GREATER;
                                  return(RELOP); }
>=                           { tokval = GREATEREQL;
                                 return(RELOP); }
```

Despite appearances, tokens returned and values assigned to **tokval** are indeed integers. Good programming style dictates the use of informative terms such as **BEGIN**, **END**, and **WHILE**, to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C.

```
#define BEGIN  1
#define END  2
    .
#define PLUS 7
    .
```

If the need arises to change the integer for some token type, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to insert the following statement into the definitions section of your **lex** source.

```
        #include y.tab.h
```

The file *y.tab.h* provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on, with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, a function **put_in_tabl**() places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl**() assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl**() would be a routine that the compiler writer might place in the subroutines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable PLUS, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a **+** sign is recognized, the action assigns to **tokval** the value 7, which indicates the **+**. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by ARITHOP or RELOP).

4.3 Running lex

As you review the following steps, recall Figure 4-1 at the start of this chapter.

To produce the lexical analyzer in C, use the following command syntax:

```
lex [-ctvnc] [file] ...
```

file A file containing **lex** source.

The command line options are as follows:

–c Indicates that the actions specified in *file* are in C. This is the default.

–t Causes the output file produced by **lex**, *lex.yy.c*, to be written to standard out.

–v Provides a one-line summary of statistics. **lex** uses a table (a two-dimensional array in C) to keep track of the states used by the **lex** rules. When **–v** is used, the number of states used is indicated by the value preceding (%n). By default, the maximum number of states is 500. If your **lex** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing an entry like this in the definitions section of your **lex** source:

```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. If you need to increase the maximum number of state transitions beyond 2000, the designated parameter is %a:

```
%a 2800
```

where %a is the number of packed transitions used.

–n Suppresses the statistics summary. This is the default.

In the following command example, *lex.l* is the file containing the **lex** specification:

```
$ lex lex.l
```

The name *lex.l* is the conventional favorite, but you can use whatever name you want. **lex** automatically produces an output file named *lex.yy.c*; this is the lexical analyzer program that you created with **lex**. You then compile and link this as you would any C program, making sure that you invoke the **lex** library with the **–ll** option:

```
$ cc lex.yy.c –ll
```

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you need not supply your own **main()**.

If you have the **lex** specification in several files, you can run **lex** with each of them individually, but be sure to rename or move each *lex.yy.c* file (with **mv**) before you run **lex** on the next one. Otherwise, each overwrites the previous one. Once you have generated the *.c* files, you can compile all of them in one command line.

With the executable **a.out** produced, you are ready to analyze any desired input text. The lexical analyzer, **a.out**, by default takes input from your terminal. If text is stored in a file, use redirection to use the file as input. For example, if the text is stored under the filename *textin*, the following command uses *textin* as input:

```
$ a.out < textin
```

By default, output appears on your terminal, but you can redirect output as well:

```
$ a.out < textin > textout
```

4.3.1 Running lex with yacc

In running **lex** with **yacc**, either may be run first. The following commands spawn a parser in the file *y.tab.c*. (The **–d** option creates the file *y.tab.h*, which contains the **#define** statements that associate the **yacc** assigned integer token values with the user-defined token names.)

```
$ yacc –d grammar.y
$ lex lex.l
```

To compile and link the output files produced, run this command:

```
$ cc lex.yy.c y.tab.c –ly –ll
```

Note that the **yacc** library is loaded (with the **–ly** option) before the **lex** library (with the **–ll** option) to ensure that the **main()** program supplied calls the **yacc** parser.

Index

5. yacc

5. yacc

Chapter 5
yacc

Chapter 5
yacc

5.1 Introduction

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes these items:

- A set of rules to describe the elements of the input
- Code to be invoked when a rule is recognized
- Either a definition or declaration of a low-level routine to examine the input

yacc then turns the specification into a C language function that examines the input stream. This function, called a *parser*, works by calling the low-level input scanner. The low-level input scanner, called a *lexical analyzer*, picks up items from the input stream. The selected items are known as *tokens*. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user-supplied code for that rule, an action, is invoked. Actions are fragments of C code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. Following is an example of a grammar rule:

```
date  :  month_name  day  ','  year  ;
```

`date`, `month_name`, `day`, and `year` represent constructs of interest; presumably, `month_name`, `day`, and `year` are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in evaluating the input. With proper definitions, the following input might be matched by the rule:

```
July 4, 1776
```

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the following rules might be used in the previous example:

```
month_name : 'J' 'a' 'n'   ;
month_name : 'F' 'e' 'b'   ;

                 ...

month_name : 'D' 'e' 'c'   ;
```

While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add the following rule to the above example:

```
date   :  month '/' day '/' year    ;
```

It allows `7/4/1776` as a synonym for `July 4, 1776` on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan, input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases often can be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructs that are difficult for **yacc** to handle are also frequently difficult for users to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The remainder of this chapter describes the following subjects:

- Basic process of preparing a **yacc** specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions to improve the style and efficiency of the specifications
- Advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

5.2 Basic Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, grammar rules, and subroutines. The sections are separated by double percent signs, *%%* (the percent sign is generally used in **yacc** specifications as an escape character).

When all sections are used, a full specification file looks like this:

declarations
%%
rules
%%
subroutines

The declarations and subroutines sections are optional. The smallest legal **yacc** specification is as follows:

```
%%
rules
```

Blanks, tabs, and newlines are ignored, but they cannot appear in names or multicharacter reserved symbols. Comments can appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

5.2.1 Rules

The rules section is made up of one or more grammar rules. A grammar rule has this form:

```
A   :   BODY   ;
```

A represents a nonterminal symbol; BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names can be of any length and can be made up of letters, dots, underscores, and digits. A digit cannot be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule can represent tokens or nonterminal symbols.

A *literal* consists of a character enclosed in single quotes ('). As in the C language, the backslash (\) is an escape character within literals, and all the C escapes are recognized. **yacc** understands the following literals:

'\n'	newline
'\r'	return
'\''	single quote (')
'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'\*xxx*'	*xxx* in octal notation

The NULL character (\0 or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, a vertical bar (|) can be used to avoid rewriting the left-hand side. The semicolon at the end of a rule is dropped before a vertical bar. Thus the following sets of grammar rules are equivalent:

```
A    :   B   C   D    ;
A    :   E   F    ;
A    :   G    ;

A    :   B   C   D
     |   E   F
     |   G
     ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by the following:

```
epsilon :   ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon**.

Names representing tokens must be declared. This is most simply done by writing the following in the declarations section:

```
%token   name1  name2 ...
```

Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword.

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually, the end-marker represents some reasonably obvious I/O status, such as end-of-file or end-of-record.

5.2.2 Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. Actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in curly braces ({ and }). The following examples show grammar rules with actions:

```
A    :  '('  B  ')'
     {
        hello( 1, "abc" );
     }

XXX    :  YYY  ZZZ
       {
          (void) printf("a message\n");
          flag = 25;
       }
```

The dollar sign symbol ($) is used to facilitate communication between the actions and the parser. The pseudovariable **$$** represents the value returned by the complete action. For example, the following action returns the value of one; in fact, that's all it does:

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudovariables **$1, $2, ... $***n*. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is as follows, then **$2** has the value returned by **C**, and **$3** the value returned by **D**:

```
A   :  B  C  D   ;
```

The following rule provides a common example:

```
expr    :    '('  expr  ')'    ;
```

You might expect the value returned by this rule to be the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated as follows:

```
expr    :       '('   expr   ')'
        {
                $$ = $2 ;
        }
```

By default, the value of a rule is the value of the first element in it (**$1**). Thus, grammar rules of the following form frequently need not have an explicit action:

```
A   :   B     ;
```

In previous examples, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This action is assumed to return a value accessible through the usual **$** mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the following rule, the effect is to set **x** to 1 and **y** to the value returned by **C**.

```
A   :   B
            {
                $$ = 1;
            }
            C
    {
            x = $2;
            y = $3;
    }
    ;
```

Actions that do not terminate a rule are handled by **yacc** by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string.

The interior action is the action triggered by recognizing this added rule. **yacc** treats the above example as if it had been written as follows, where **$ACT** is an empty action:

```
$ACT    :       /* empty */
        {
                $$ = 1;
        }
        ;
```

```
A       :   B  $ACT  C
        {
            x = $2;
            y = $3;
        }
        ;
```

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function node written so that the call creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node:

```
node( L, n1, n2 )
```

Then a parse tree can be built by supplying actions such as the following in the specification:

```
expr    :   expr  '+'  expr
        {
            $$ = node( '+', $1, $3 );
        }
```

The user can define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example, the following could be placed in the declarations section, making **variable** accessible to all of the actions:

```
%{   int variable = 0;   %}
```

Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far, all the values are integers. A discussion of values of other types is found in "Advanced Topics" later in this chapter.

5.2.3 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. To return the relevant token, a portion of the lexical analyzer might look like this:

```
int yylex()
{
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch (c)
    {
        ...
        case '0':
        case '1':
        ...
        case '9':
        yylval = c - '0';
        return (DIGIT);
        ...
    }
    ...
}
```

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the subroutines section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily-modified lexical analyzers. The only pitfall involves using any token names in the grammar that are reserved or significant in C or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used carelessly.

In the default situation, token numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. If the **yacc** command is invoked with the **–d** option, a file called *y.tab.h* is generated. *y.tab.h* contains **#define** statements for the tokens.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker must have token number 0 or be negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN), which do not fit any theoretical framework and whose lexical analyzers must be written by hand.

5.3 Parser Operation

yacc turns the specification file into a C language procedure, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and is not discussed here. The parser itself, though, is relatively simple and understanding its usage makes treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token called the *look-ahead token*. The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—**shift**, **reduce**, **accept**, and **error**. Parsing is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead token, if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and the look-ahead token being processed or left alone.

The **shift** action is the most common action the parser takes. Whenever a **shift** action is taken, there is always a look-ahead token.

The following example says that in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

```
IF   shift 34
```

The **reduce** action keeps the stack from growing without bounds. **reduce** actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether or not to **reduce** (usually it is not necessary). In fact, the default action, represented by a dot, is often a **reduce** action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The following action refers to grammar rule 18:

```
.   reduce 18
```

The next action refers to state 34:

```
IF   shift 34
```

Suppose the following rule is being reduced:

```
   A   :   x  y  z     ;
```

The **reduce** action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing **x**, **y**, and **z** and no longer serve any useful purpose. After popping these states, a state is uncovered, which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of **A**. A new state is obtained, pushed onto the stack, and parsing continues.

However, there are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. In any case, the uncovered state contains an entry such as the following causing state 20 to be pushed onto the stack and become the current state:

```
    A    goto 20
```

In effect, the **reduce** action turns back the clock in the parse, popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a **shift** takes place, the external variable **yylval** is copied onto the value stack. After the return from the user code, the reduction is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudovariables **$1**, **$2**, and so on, refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job. The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) is discussed later in this chapter.

Consider the following entry as a **yacc** specification:

```
%token  DING  DONG  DELL
%%
rhyme      :    sound  place
           ;
sound      :    DING  DONG
           ;
place      :    DELL
           ;
```

When **yacc** is invoked with the **–v** option, a file called *y.output* is produced with a readable description of the parser. (Refer to "Running **yacc**" later in this chapter.) The *y.output* file corresponding to the previous **yacc** specification (with some statistics stripped off the end) follows:

```
state 0
        $accept   :   _rhyme   $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept   :    rhyme_$end

        $end  accept
        .  error

state 2
        rhyme   :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound    :   DING_DONG

        DONG  shift 6
        .  error
```

```
state 4
        rhyme  :  sound  place_    (1)

            .   reduce  1

state 5
        place  :  DELL_    (3)

            .   reduce  3

state 6
        sound   :  DING  DONG_    (2)

            .   reduce  2
```

The actions for each state are specified and there is a description of the parsing rules being processed in each state. The underscore (_) character is used to indicate what has been seen and what is yet to come in each rule. The following input can be used to track the operations of the parser:

```
DING  DONG  DELL
```

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by the following, which is rule 2:

```
sound  :   DING  DONG
```

Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**), the following is obtained:

```
sound    goto 2
```

State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, DELL, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The **goto** in state 2 on **place** (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a **goto** on **rhyme** causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by **$end** in the *y.output* file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, and so on. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

5.4 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the following grammar rule shows that one way of forming an arithmetic expression is to put two expressions together with a minus sign between them:

```
expr    :    expr  '-'  expr
```

Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. Consider the following input:

```
expr  -  expr  -  expr
```

The rule allows this input to be structured in either of the following ways:

```
(  expr  -  expr  )  -  expr

expr  -  (  expr  -  expr  )
```

The first is called left association, the second, right association.

yacc detects such ambiguities when it is attempting to build the parser. Given the following input, consider the problem that confronts the parser:

```
expr  -  expr  -  expr
```

When the parser has read the second *expr*, the following input seen matches the right side of the grammar rule above:

```
expr  -  expr
```

The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to the following (the left side of the rule):

```
expr
```

The parser then reads the final part of the input:

```
-  expr
```

The parser reduces this final part. The effect of this is to take the left-associative interpretation.

Alternatively, if the parser sees

```
  expr  -  expr
```

it could defer the immediate application of the rule and continue reading the input until it sees

```
  expr  -  expr  -  expr
```

It could then apply the rule to the rightmost three symbols, reducing them to `expr`. This results in the following being left:

```
  expr  -  expr
```

Now the rule can be reduced once more. The effect is to take the right-associative interpretation.

You can see that given the grammar `expr - expr`, the parser can do one of two legal things: **shift** or **reduce**. It has no way of deciding between them. This is called a **shift-reduce** conflict. It may also happen that the parser has a choice of two legal reductions. This is called a **reduce-reduce** conflict. There are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a *disambiguating rule*.

yacc invokes two default disambiguating rules:

1. In a **shift-reduce** conflict, the default is to do the shift.
2. In a **reduce-reduce** conflict, the default is to reduce by the earlier grammar rule (in the **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** produces parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider the following fragment from a programming language involving an **if-then-else** statement:

```
stat    :    IF  '('  cond  ')'  stat
        |    IF  '('  cond  ')'  stat  ELSE  stat
        ;
```

In these rules, IF and ELSE are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple **if** rule and the second the **if-else** rule.

These two rules form an ambiguous construction because input of the following form can be structured according to these rules in two ways:

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

The two constructions are as follows:

```
IF   ( C1 )
{
        IF   ( C2 )
                S1
}
ELSE
        S2
```

or

```
IF   ( C1 )
{
        IF   ( C2 )
                S1
        ELSE
                S2
}
```

The second is the one given in most programming languages having this construct; each ELSE is associated with the last preceding un-ELSE'd IF. In this example, consider the situation where the parser has seen the following entry and is looking at the ELSE:

```
IF  (  C1  )  IF  (  C2  )  S1
```

It can immediately reduce by the simple **if** rule to get the following:

```
IF  (  C1  )  stat
```

It then reads the remaining input:

```
ELSE  S2
```

and reduces by the **if-else** rule as follows:

```
IF  (  C1  )  stat  ELSE  S2
```

This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right-hand portion of

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
IF  (  C1  )  stat
```

which can be reduced by the simple **if** rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again, the parser can do two valid things—there is a **shift-reduce** conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as the following have already been seen:

```
IF  (  C1  )  IF  (  C2  )  S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**–v**) option output file. For example, the output corresponding to the above conflict state might be as follows:

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE
```

```
state 23

    stat  :  IF  (  cond  )  stat_         (18)
    stat  :  IF  (  cond  )  stat_ELSE  stat

    ELSE     shift 45
    .        reduce 18
```

The first line describes the **shift-reduce** conflict, giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that grammar rules which have been seen are marked by an underscore. Thus, in the example, in state 23 the parser has seen input corresponding to the following statement. The two grammar rules shown are active at this time.

```
IF  (  cond  )  stat
```

The parser can do one of two things. If the input symbol is ELSE, it could shift into state 45. State 45 has, as part of its description, the following line because the ELSE has been shifted in this state:

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

In state 23, the alternative action (describing a dot, .), is done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not ELSE, the parser reduces to the following by grammar rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed in parentheses after those rules, which can be reduced. In most states, there is a reduce action possible in the state and this is the default command. The user who encounters unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

5.5 Precedence

One common situation in which the disambiguating rules are insufficient is the parsing of arithmetic expressions. Most commonly used constructions for arithmetic expressions can be described by the notion of precedence levels for operators, together with information about left or right associativity. Parsers with ambiguous grammars that have appropriate disambiguating rules are faster and easier to write than parsers constructed from unambiguous grammars. The idea is to write grammar rules for all binary and unary operators desired of the following form:

expr : expr OP expr
expr : UNARY expr

This creates an ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and to construct a parser with the correct precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus, the following entry describes the precedence and associativity of the four arithmetic operators:

```
%left   '+'   '-'
%left   '*'   '/'
```

Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right-associative operators, and the keyword **%nonassoc** is used to describe operators, like **.LT.** in FORTRAN, that may not associate with themselves. Thus, the following entry is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**:

```
A  .LT.  B  .LT.  C
```

As an example of the behavior of these declarations, the description

```
%right  '='
%left   '+'   '-'
%left   '*'   '/'
```

```
%%

expr    :    expr  '='  expr
        |    expr  '+'  expr
        |    expr  '-'  expr
        |    expr  '*'  expr
        |    expr  '/'  expr
        |    NAME
        ;
```

might be used to structure the input

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows in order to perform the correct precedence of operators.

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus, –.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the following rules might be used to give unary minus the same precedence as multiplication:

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :    expr  '+'  expr
        |    expr  '-'  expr
        |    expr  '*'  expr
        |    expr  '/'  expr
        |    '-'  expr        %prec  '*'
        |    NAME
        ;
```

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a **reduce-reduce** conflict or there is a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a **shift-reduce** conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action—**shift** or **reduce**—associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; nonassociating implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in a cookbook fashion until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

5.6 Error Handling

Error handling is difficult; many of the problems are semantic ones. When an is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, or, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides the token name **error**. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the following form means that on a syntax error the parser attempts to skip over the statement in which the error is seen:

```
  stat    :   error
```

More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and starts processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, and so on.

The error rules described above are general but difficult to control. Rules such as the following are somewhat easier:

```
  stat    :   error  ';'
```

Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any cleanup action associated with it performed.

Another form of **error** rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example shows one way to do this:

```
input   :   error  '\n'
                 {
                     (void) printf( "Reenter last line: " );
                 }
                 input
        {
            $$ = $4;
        }
        ;
```

There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The following statement in an action resets the parser to its normal mode.

```
  yyerrok ;
```

The last example can be rewritten as follows, which is somewhat better:

```
input   :   error  '\n'
                 {
                    yyerrok;
                    (void) printf( "Reenter last line: " );
                 }
                 input
        {
            $$ = $4;
        }
        ;
```

As previously mentioned, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The following statement in an action will have this effect:

```
yyclearin ;
```

For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset. A rule similar to the following could perform this.

```
stat    :    error
        {
           resynch();
           yyerrok   ;
           yyclearin;
        }
        ;
```

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

5.7 Running yacc

Use the following command syntax to run **yacc**:

```
yacc [-vdlt] grammar
```

The replaceable parameter is as follows:

grammar	**yacc** *specifications.*

The command line options are as follows:

–v	Produce the file *y.output*, which contains a description of the parsing tables and reports on conflicts and ambiguities in the grammar.
–d	Produces *y.tab.h* with *#define* statements that associate **yacc** token codes with the user-declared token names so that source files other than *y.tab.c* can access the token codes.
–l	Produces *y.tab.c* so that it contains no *#line* constructs.
–t	Produces debugging code in *y.tab.c*.

5.8 The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C subroutines called *y.tab.c*. The function produced by **yacc** is called **yyparse**(); it is an integer-valued function. When it is called, it in turn repeatedly calls **yylex**(), the lexical analyzer supplied by the user to obtain input tokens (refer to "Lexical Analysis" earlier in this chapter). Eventually, an error is detected, **yyparse**() returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse**() returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a routine called **main**() must be defined that eventually calls **yparse**(). In addition, a routine called **yyerror**() is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of **main**() and **yerror**(). The library is accessed by specifying the **–ly** option to the **cc**(1) or **ld**(1) command. The following source codes show the triviality of these default programs:

```
main()
{
        return (yyparse());
}
```

and

```
# include <stdio.h>

yyerror(s)
        char *s;
{
        (void) fprintf(stderr, "%s\n", s);
}
```

The argument to **yerror**() is a string containing an error message, usually the string **syntax error**. The average application should do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main**() routine is probably supplied by the user, the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are.

5.9 Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy-to-change, and clear specifications. The individual subsections are independent.

5.9.1 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. A few style hints follow:

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

The first simple example in "Examples", later in this chapter, is written following this style, as are the examples in this section (where space permits). The user must decide about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

5.9.2 Left Recursion

The algorithm used by the **yacc** parser encourages so called left-recursive grammar rules. Rules of the following form match this algorithm:

```
name    :    name  rest_of_rule  ;
```

These rules, such as the following, frequently arise when writing specifications of sequences and lists:

```
list    :    item
        |    list  ','  item
        ;
```

and

```
seq     :    item
        |    seq  item
        ;
```

In each case, the first rule is reduced for the first item only; the second rule is reduced for the second and all succeeding items.

With right-recursive rules, the parser is a bit bigger and the items are seen and reduced from right to left:

```
seq     :    item
        |    item  seq
        ;
```

More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the following sequence specification as using an empty rule:

```
seq     :    /* empty */
        |    seq  item
        ;
```

Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts may arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know.

5.9.3 Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example, the following specifies a program that consists of zero or more declarations followed by zero or more statements.

```
%{
   int dflag;
%}
  ...  other declarations ...

%%

prog     :    decls  stats
         ;
```

```
decls   :   /* empty */
        {
                dflag = 1;
        }
        |   decls   declaration
        ;

stats   :   /* empty */
        {
                dflag = 0;
        }
        |   stats   statement
        ;

     ...  other rules ...
```

The flag **dflag** is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back-door approach may not be desirable, but it represents a way of doing some things that are difficult to do otherwise.

5.9.4 Reserved Words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer and to tell it that this instance of **if** is a keyword while that instance is a variable. Therefore, it is better that the keywords be reserved, that is, forbidden for use as variable names.

5.10 Advanced Topics

This section discusses a number of advanced features of **yacc**.

5.10.1 Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of the macros YYACCEPT and YYERROR. The YYACCEPT macro causes **yyparse**() to return the value 0, YYERROR causes the parser to behave as if the current input symbol had been a syntax error, **yyerror**() is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context-sensitive syntax checking.

5.10.2 Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

```
sent    :   adj  noun  verb  adj  noun
        {
            look at the sentence ...
        }
        ;
adj     :   THE
        {
                $$ = THE;
        }
        |   YOUNG
        {
                $$ = YOUNG;
        }
        ...
        ;
noun    :   DOG
        {
            $$ = DOG;
        }
        |   CRONE
        {
            if( $0 == YOUNG )
            {
                (void) printf( "what?\n" );
            }
            $$ = CRONE;
        }
        ;
        ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. This is possible only when a great deal is known about what might precede the symbol **noun** in the input. There is a distinctly unstructured flavor to this. Nevertheless, at times this mechanism prevents trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

5.10.3 Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with each token and nonterminal symbol having a value. When the value is referenced through a **$$** or **$n** construction, **yacc** automaticallys insert the appropriate union name so that no unwanted conversions take place. In addition, type-checking commands such as **lint**(1) are silent.

There are three mechanisms used to provide for this typing. First, you can define the union. This must be done by the user since other subroutines, notably the lexical analyzer, must know about the union member names. Second, you can associate a union member name with tokens and nonterminals. Finally, you can define the type of those values where **yacc** cannot easily determine the type.

To declare the union, the user includes the following in the declaration section:

```
%union
{
   body of union ...
}
```

This declares the **yacc** value stack and the external variables **yylval** and **yyval** to have type equal to this union. If **yacc** was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file as YYSTYPE.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The following construction is used to indicate a union member name.

```
<name>
```

If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Specifying the following causes any reference to values returned by these two tokens to be tagged with the union member name **optype**:

```
%left  <optype>  '+'  '-'
```

Another keyword, **%type**, is used to associate union member names with nonterminals. To associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**, you could say the following:

```
%type  <nodetype>  expr  stat
```

There are a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no predetermined type. Similarly, reference to left-context values (such as **$0**) leaves **yacc** with no easy way of determining the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The following example shows this usage:

```
rule    :    aaa
                   {
                       $<intval>$ = 3;
                   }
                   bbb
        {
             fun( $<intval>2, $<other>0 );
        }
        ;
```

This syntax is not great, but the situation arises rarely.

A sample **yacc** specification is given in the advanced example in "Examples" at the end of this chapter. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of **$n** or **$$** to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold **ints**.

5.10.4 yacc Input Syntax

This section describes the **yacc** input syntax by showing a **yacc** specification. Context dependencies are not considered. Although **yacc** accepts an LALR(1) grammar, the **yacc** input specification language is most naturally specified as an LR(2) grammar. Problems arise when an identifier occurs in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs, but never as part of C_IDENTIFIERs.

The following displays grammar for the input to **yacc**.

```
    /* basic entries */
%token      IDENTIFIER    /* includes identifiers and literals */
%token      C_IDENTIFIER  /* identifier (but not literal) followed by a : */
%token      NUMBER        /* [0-9]+ */

     /*     reserved words: %type=>TYPE %left=>LEFT,etc. */

%token      LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token      MARK    /* the %% mark */
%token      LCURL   /* the %{ mark */
%token      RCURL   /* the %} mark */

    /*  ASCII character literals stand for themselves */

%token     spec

%%

spec    :     defs MARK rules tail
        ;

tail    :     MARK
        {
                 In this action, eat up the rest of the file
        }
        |     /* empty: the second MARK is optional */
        ;

defs    :     /* empty */
        |     defs def
        ;
def     :     START IDENTIFIER
        |     UNION
        {
                 Copy union definition to output
        }
        |    LCURL
        {
                     Copy C code to output file
        }
             RCURL
        |    rword tag nlist
        ;

rword   :    TOKEN
        |    LEFT
        |    RIGHT
        |    NONASSOC
        |    TYPE
        ;

tag     :    /* empty: union tag is optional */
        |    '<' IDENTIFIER '>'
        ;

nlist   :   nmno
        |   nlist nmno
        |   nlist ',' nmno
        ;

nmno    :   IDENTIFIER          /* Note: literal illegal with % type */
        |   IDENTIFIER NUMBER   /* Note: illegal with % type */
        ;

   /* rule section */

rules   :   C_IDENTIFIER rbody prec
```

```
        |  rules rule
        ;
rule    :  C_IDENTIFIER rbody prec
        |  '|' rbody prec
        ;


rbody   :  /* empty */
        |  rbody IDENTIFIER
        |  rbody act
        ;

act     :  '{'
           {
                 Copy action translate $$ etc.
           }
           '}'
        ;

prec    :  /* empty */
        |  PREC IDENTIFIER
        |  PREC IDENTIFIER act
        |  prec ';'
        ;
```

5.11 Examples

The first example gives a complete **yacc** application for a desk calculator. The second example modifies the desk calculator application to add floating-point arithmetic.

5.11.1 A Simple Example

The following example is a **yacc** application for a small desk calculator with 26 registers labeled **a** through **z**. The calculator accepts assignments and arithmetic expressions made up of the following operators:

+	addition
–	subtraction
*	multiplication
/	division
%	mod operator
&	bitwise and
|	bitwise or

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in C , an integer that begins with zero is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator shows how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably done better by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */

%%          /* beginning of rules section */

list        :  /* empty */
            |  list stat '\n'
            |  list error '\n'
            {
               yyerrok;
            }
            ;
```

```
stat        :  expr
            {
               (void) printf( "%d\n", $1 );
            }
            |  LETTER '=' expr
            {
               regs[$1] = $3;
            }
            ;

expr        :  '(' expr ')'
            {
                  $$ = $2;
            }
            |  expr '+' expr
            {
                  $$ = $1 + $3;
            }
            |  expr '-' expr
            {
                  $$ = $1 - $3;
            {
            |  expr '*' expr
            {
                  $$ = $1 * $3;
            }
            |  expr '/' expr
            {
                  $$ = $1 / $3;
            }
            |   exp '%' expr
            {
                   $$ = $1 % $3;
            }
            |   expr '&' expr
            {
                  $$ = $1 & $3;
            }
            |   expr '|' expr
            {
                  $$ = $1 | $3;
            }
            |  '-' expr  %prec UMINUS
            {
                  $$ = -$2;
            }
            |  LETTER
            {
                  $$ = reg[$1];
            }
```

```
          |  number
          ;

number    :   DIGIT
          {
                $$ = $1; base = ($1==0) ? 8 ; 10;
          }
          |   number DIGIT
          {
                $$ = base * $1 + $2;
          }
          ;

%%      /* beginning of subroutines section */

int yylex( )    /* lexical analysis routine */
{               /* return LETTER for lowercase letter, */
                /* yylval = 0 through 25 */
                /* returns DIGIT for digit, yylval = 0 through 9 */
                /* all other characters are returned immediately */

          int c;
                      /*skip blanks*/
          while ((c = getchar()) == ' ')
                ;

                      /* c is now nonblank */

          if (islower(c))
          {
                     yylval = c - 'a';
                     return (LETTER);
          }
          if (isdigit(c))
          }
                     yylval = c - '0';
                     return (DIGIT);

          }
          return (c);
}
```

5.11.2 An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example in Example 1 is modified to provide a desk calculator that does floating-point interval arithmetic. The calculator understands floating-point constants; the arithmetic operations +, – *, /, unary – **a** through **z**. It also understands intervals written `(X,Y)`, where **X** is less than or equal to **Y**. There are 26 interval valued variables **A** through **Z** that may also be used. The usage is similar to that in the previous example; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example shows some interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using **typedef**. The **yacc** value stack can also contain floating-point scalars and integers (used to index into the arrays holding the variable values). The entire strategy depends on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** throws away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates the use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. The scalar can be automatically promoted to an interval if the context demands an interval value. This causes *shift-reduce* and *reduce-reduce* conflicts when the grammar is run through **yacc**. The problem can be seen by looking at the following input lines:

```
2.5 + (3.5 - 4.)
2.5 + (3.5, 4)
```

In the second line, the 2.5 is meant to be an interval value expression, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. It may be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is avoided by having two rules for each binary interval valued operator—one when the left operand is a scalar and one when the left operand is an interval. In the second line, the right operand must be an interval, so the conversion is applied automatically. There are still cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict is resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This method of handling multiple types is useful. If there are more than two expression types, the number of rules needed would increase dramatically and the conflicts would increase even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating-point constants. The C library routine **atof()** is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar causing the syntax error, and causing error recovery in the parser.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
     double lo, hi;
}  INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG  /* indices into dreg, vreg arrays */
```

```
%token <dval> CONST       /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */

  /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS   /* precedence for unary minus */

%%      /* beginning of rules section */

lines   :  /* empty */
        |  lines line
        ;
line    :  dexp '\n'
        {
                   (void) printf("%15.8f\n",$1);
        }
        |  vexp '\n'
        {
                   (void) printf("(%15.8f, %15.8f)\n", $1.lo, $1.hi);

        }
        |  DREG '=' dexp '\n'
        {
                 dreg[$1] = $3;
        }
        |  VREG '=' vexp '\n'
        {
                 vreg[$1] = $3;
        }
        |  error '\n'
        {
                   yyerrok;
        }
        ;

dexp    :  CONST
        |  DREG
        {
                  $$ = dreg[$1];
        }
        |  dexp '+' dexp
        {
                    $$ = $1 + $3;
        }
        |  dexp '-' dexp
```

```
          {
                    $$ = $1 - $3;
          }
          |  dexp '*' dexp
          {
                    $$ = $1 * $3;
          }
          |  dexp '/' dexp
          {
                    $$ = $1 / $3;
          }
          |  '-' dexp    %prec UMINUS
          {
                   $$ = -$2;
          }
          |  '(' dexp')'
          {
                    $$ = $2;
          }
          ;

vexp      :  dexp
          {
                  $$.hi = $$.lo = $1;
          }
          |  '(' dexp ',' dexp ')'
          {
                  $$.lo = $2;
                  $$.hi = $4;
                  if( $$.lo > $$.hi )

                  {
                         (void) printf("interval out of order \n");
                         YYERROR;
                  }
          }
          |  VREG
          {
                    $$ = vreg[$1];
          }
          |  vexp '+' vexp
          {
                 $$.hi = $1.hi + $3.hi;
                 $$.lo = $1.lo + $3.lo;
          }
          |  dexp '+' vexp
          {
                 $$.hi = $1 + $3.hi;
                 $$.lo = $1 + $3.lo;
          }
```

```
        |   vexp '-' vexp
        {
                $$.hi = $1.hi - $3.lo;
                $$.lo = $1.lo - $3.hi;
        }
        |   dvep '-' vdep
        {
                $$.hi = $1 - $3.lo;
                $$.lo = $1 - $3.hi
        }
        |   vexp '*' vexp
        {
                $$ = vmul( $1.lo,$.hi,$3 )
        }
        |   dexp '*' vexp
        {
                $$ = vmul( $1, $1, $3 )
        }
        |   vexp '/' vexp

        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |   dexp '/' vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |   '-' vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;$$.lo = -$2.hi
        }
        |   '(' vexp ')'
        }
                $$ = $2
        }
        ;

%%          /* beginning of subroutines section */

# define BSZ 50   /* buffer size for floating point number */

        /* lexical analysis */

int yylex( )
{
        register int c;

                                /* skip over blanks */
```

```
while ((c = getchar()) == ' ')
;
if (isupper(c))
{
    yylval.ival = c - 'A'
    return (VREG);
}
if (islower(c))

{
    yylval.ival = c - 'a',
    return( DREG );
}

    /* gobble up digits. points, exponents */

if (isdigit(c) || c == '.')
{
    char buf[BSZ+1], *cp = buf;
    int dot = 0, exp = 0;

    for(; (cp - buf) < BSZ ; ++cp, c = getchar())
    {
        *cp = c;
         if (isdigit(c))
           continue;
         if (c == '.')
        {
         if (dot++ || exp)
           return ('.');   /* will cause syntax error */
           continue;
        }
        if( c == 'e')
        {
         if (exp++)
           return ('e');   /* will cause syntax error */
         continue;
        }
              /* end of number */
        break;
    }

    *cp = '\0';
    if (cp - buf >= BSZ)
       (void) printf("constant too long - truncated\n");
    else
       ungetc(c, stdin);   /* push back last char read */
    yylval.dval = atof(buf);
    return (CONST);
}
```

```
        return (c);
}
INTERVAL
hilo(a, b, c, d)
        double a, b, c, d;
{
          /* returns the smallest interval containing a, b, c, and d */

          /* used by *,/ routine */
        INTERVAL v;

        if (a > b)
        {
              v.hi = a;
              v.lo = b;
        }
        else
        {
              v.hi = b;
              v.lo = a;
        }
        if (c > d)
        {
              if (c > v.hi)
                  v.hi = c;
              if (d < v.lo)
                  v.lo = d;
        }
        else
        }
              if (d > v.hi)
                  v.hi = d;
              if (c < v.lo)
                  v.lo = c;
        }
        return (v);
}

INTERVAL
vmul(a, b, v)
        double a, b;
        INTERVAL v;
{
        return (hilo(a * v.hi, a * v,lo, b * v.hi, b * v.lo));
}
dcheck(v)
        INTERVAL v;
{
        if (v.hi >= 0. && v.lo <= 0.)
        {
                (void) printf("divisor interval contains 0.\n");
                return (1);
        }
        return (0);
{

INTERVAL
vdiv(a, b, v)
        double a, b;
        INTERVAL v;
{
   return (hilo(a / v.hi, a / v,lo, b / v.hi, b / v.lo));
}
```

Index

6. make

6. make

Chapter 6
make

Figures

Chapter 6
make

6.1 Introduction

The **make**(1) command provides a method for maintaining, updating, and regenerating up-to-date versions of programs that consist of a number of files.

make keeps track of the following items:

- File-to-file dependencies
- Files that were modified and the impact those modifications have on other files
- The exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

make performs these tasks:

- Finds the target in the description file.
- Ensures that all the files on which the target depends (the files needed to generate the target) exist and are up-to-date.
- Creates the target file if any of the generators have been modified more recently than the target.

The description file that holds the information on interfile dependencies and command sequences is, by convention, called *makefile*, *Makefile*, or *s.[mM]akefile*. The **make** command looks for these default description filenames in the order given and regenerates the target regardless of the number of files edited since the last **make**. In most cases, the *makefile* (description file) is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures that the regeneration is done in the prescribed way. The description file, no matter what its name, is conventionally referred to as "the makefile." In this chapter, it is referred to as the *makefile* or the description file. Refer to **make**(1) for information on running **make** in parallel.

6.2 Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up-to-date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies; **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- A user-supplied description file (*makefile*, *Makefile*, or *s.[mM]akefile*)
- Filenames and last-modified times from the filesystem
- Built-in rules to bridge some of the gaps

Consider a simple example in which a program named **prog** is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **math** library. By convention, the output of the C language compilations is found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs.h*, but that *z.c* does not. That is, *x.c* and *y.c* have this line:

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lm  -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named *makefile*, the **make** command would use the *makefile* to perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs.h*. In the example above, the first line states that **prog** depends on three *.o* files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that *x.o* and *y.o* depend on the file *defs.h*. From the filesystem, **make** discovers that there are three *.c* files corresponding to the needed *.o* files and uses built-in rules on how to generate an object from a C source file (that is, issue a **cc –c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, **make** announces this fact and stops. If, however, *defs.h* has been edited, *x.c* and *y.c* (but not *z.c*) are recompiled; **prog** is then created from the new *x.o* and *y.o* files and the existing *z.o* file. If only the file *y.c* has changed, only it is recompiled. It is still necessary, however, to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The following command would regenerate *x.o* if *x.c* or *defs.h* had changed.

```
$ make x.o
```

A useful programming method is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of the ability of **make** to generate files and substitute macros (For information about macros, refer to "Description Files and Substitutions" later in this chapter.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by the macro definition. A macro is invoked by preceding the name with a dollar sign.

Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$***, **$@, **$?**, and **$<** are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following description file fragment shows the assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
   . . .
```

The following command loads the three objects with both the **lex** (–ll) and the **math** (–lm) libraries, because macro definitions on the command line override definitions in the description file.

```
$ make LIBES="-ll -lm"
```

(In operating system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, the following is a description file that might be used to maintain the **make** command itself. (The code for **make** is spread over a number of C language source files and has **yacc** grammar.) The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:  $(OBJECTS)
```

```
            $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
            @size make

$(OBJECTS):   defs.h

cleanup:
            -rm *.o gram.c
            -du

install:
            @size make /usr/bin/make
            cp make /usr/bin/make && rm make

lint :      dosys.c doname.c files.c main.c misc.c gram.c
            $(LINT) dosys.c doname.c files.c main.c misc.c \
            gram.c

          # print files that are out-of-date
          # with respect to "print" file.

print: $(FILES)
       pr $? | $(LP)
       touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing **make** in a directory containing only the source and description files:

```
cc   -O -c main.c
cc   -O -c doname.c
cc   -O -c misc.c
cc   -O -c files.c
cc   -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc   -O -c gram.c
cc   main.o doname.o misc.o files.o dosys.o
      gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself is suppressed by the at sign (@) in the *makefile*.

6.3 Description Files and Substitutions

The following section explains the elements of a description file.

6.3.1 Comments

The comment convention is that a pound sign (#) and all subsequent characters on the same line are ignored. Blank lines and lines beginning with a pound sign are totally ignored. For example, the fourth, fifth, and sixth lines in the following description file fragment would be ignored:

```
lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

       # print files that are out-of-date
       # with respect to "print" file.
```

6.3.2 Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash (\). If the last character of a line is a backslash, the backslash, the new line, and all following blanks and tabs are replaced by a single blank. For example, the second line in this description file fragment is continued to the third line:

```
lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

       # print files that are out-of-date
       # with respect to "print" file.
```

6.3.3 Macro Definitions

A macro definition is an identifier (name) followed by an equals sign (=). The identifier must not be preceded by a colon or a tab. The identifier (a string of letters and digits) is the macro name to the left of the equals sign (trailing blanks and tabs are stripped). The identifier is assigned the string of characters following the equals sign (leading blanks and tabs are stripped).

The following lines are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns **LIBES** the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (Refer to Figure 6-2 at the end of this chapter.)

6.3.4 General Form

The general form of an entry in a description file is as follows:

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
 . . .
```

Items inside brackets may be omitted, and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters, such as * and ?, are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line, or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a pound sign (#), except when the pound sign is in quotes.

6.3.5 Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the more common single-colon case, a command sequence may be associated with, at most, one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked.

In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double-colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in "Archive Libraries" later in this chapter.)

6.3.6 Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (–s option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the –i option has been specified on the **make** command line, if the fake target name **.IGNORE** appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (for example, **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten by the shell before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; the **$*** macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **DEFAULT** are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**.

6.3.7 Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list have been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single-character macro. The **F** refers to the filename part of the single-character macro. These additions are useful when building hierarchical *makefiles* because they allow access to directory names for purposes of using the **cd** command of the shell.

Consider the following example:

```
$ cd $(<D); $(MAKE) $(<F)
```

If the use of **$<** would produce the pathname string "dir/big/see", the command above, when passed to the shell, would be as follows:

```
$ cd dir/big: make see
```

6.3.8 Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation becomes apparent when you are maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (that is, those files ending in *.c*). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the information that **make** generates.

6.4 Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(**MAKE**) appears anywhere in a shell command line, the line is executed even if the **–n** option is set. Since **–n** is exported across invocations of **make** (through the **MAKEFLAGS** variable), only the **make** command itself is executed. This feature is useful when a hierarchy of **makefile**s describe a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done is printed, including output from lower-level invocations of **make**.

6.4.1 Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **–r** option is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name **.SUFFIXES**. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform an *.r* file to an *.o* file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for **.SUFFIXES** in the description file. The dependents are added to the usual list. A **.SUFFIXES** line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

6.4.2 Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file or SCCS C source file
.f	FORTRAN source file or SCCS FORTRAN source file
.s	Assembler source file or SCCS Assembler source file
.y	**yacc** source grammar or SCCS **yacc** source grammar
.l	**lex** source grammar or SCCS **ex** source grammar
.h	Header file or SCCS header file
.sh	Shell file or SCCS shell file

Figure 6-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

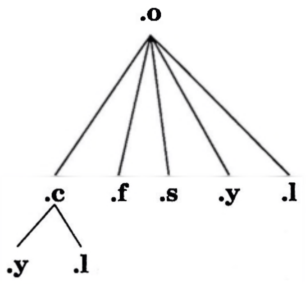


Figure 6-1. Summary of default transformation path.

If the file *x.o* is needed and an *x.c* is found in the description or directory, the *x.o* file would be compiled. If there is also an *x.l*, that source file would be run through **lex** before compiling the result. However, if there is no *x.c* but there is a *2.l*, **make** would discard the intermediate C language file and use the direct link as shown in Figure 6-1.

By knowing the macro names used, it is possible to change the names of some of the compilers used in the default or the options with which they are invoked. The compiler names are the macros **as**, **cc**, **fortran**, **yacc**, and **lex**. The following command causes the **newcc** command to be used instead of the usual C language compiler.

```
$ make CC=newcc
```

The macros **ASFLAGS**, **CFLAGS**, **F77FLAGS**, **YFLAGS**, and **LFLAGS** may be set to cause these commands to be issued with optional options. Thus, the following command causes the **cc** command to include debugging information.

```
$ make "CFLAGS=-g"
```

6.4.3 Archive Libraries

The **make** program has an interface to archive libraries. A user can name a member of a library in the following manner:

```
projlib(object.o)
     or
projlib((encrypt))
```

The second method actually refers to an entry point of an object file within the library. **make** looks through the library, locates the entry point, and translates it to the correct object filename.

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
projlib::    projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::    projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

. . . and so on for each object . . .

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename, in most cases, being the only difference.

The **make** command also gives the user access to a rule for building libraries. The rule is indicated by the *.a* suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the *.o* file. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde syntax is described later in this chapter.) The user may define other needed rules in the description file.

The two-member library in the previous *makefile* example is maintained with the following shorter *makefile*:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
            @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file and to the filename minus the suffix (*pfile1.c* and *pfile1* respectively). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

The following paragraphs explain in detail what **make** does when it sees this construction:

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume that the object in the library is out of date with respect to *pfile1.c*. Also, there is no *pfile1.o* file.

1. **make projlib.**
2. Before **make**ing **projlib**, check each dependent of **projlib**.
3. Generate **projlib**(*pfile1.o*). It is a dependent of **projlib**.
4. Before generating **projlib**(*pfile1.o*), check each dependent of **projlib**(*pfile1.o*). (There are none.)
5. Use internal rules to try to create **projlib**(*pfile1.o*). (There is no explicit rule.) Note that **projlib**(*pfile1.o*) has parentheses in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parentheses imply the **.a**

suffix. In this sense, the **.a** is hard-wired into **make**.

6. Break up the name **projlib**(*pfile1.o*) into **projlib** and *pfile1.o*. Define two macros, **$@** (=**projlib**) and **$*** (=*pfile1*).
7. Look for a rule *.X***.a** and a file *$*.X*. The first *.X* (in the .SUFFIXES list) that fulfills these conditions is *.c*, so the rule is **.c.a**, and the file is *pfile1.c*. Set **$<** to be *pfile1.c* and execute the rule. In fact, **make** must then compile *pfile1.c*.
8. The library has been updated. Execute the command associated with the **projlib:** dependency:

   ```
   @echo projlib up-to-date
   ```

It should be noted that to let *pfile1.o* have dependencies, the following syntax is required:

```
projlib(pfile1.o):        $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parentheses.

6.5 Source Code Control System Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, *s.* precedes the filename part of the complete pathname.

To allow **make** easy access to the prefix *s.*, the tilde (~) is used as an identifier of SCCS files. Hence, *.c~.o* refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is as follows:

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

.c~
.f~
.y~
.l~
.s~
.sh~
.h~

The following rules involving SCCS transformations are internally defined:

.c~
.f~
.sh~
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:

The user can define other rules and suffixes. The tilde provides a handle on the SCCS filename format so that this is possible.

6.5.1 The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the operating system program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
        $(CC) $(CFLAGS) $< -o $@
```

In fact, this **.c:** rule is internally defined so that no *makefile* is necessary at all. The user only needs to type the following:

```
$ make cat dd echo date
```

(**cat**, **dd**, **echo**, and **date** are all operating system single-file programs.) All four C language source files are passed through the preceding shell command line associated with the **.c:** rule. The internally defined single suffix rules are as follows:

.c:
.c~:
.f:
.f~:
.sh:
.sh~:

Others can be added in the *makefile* by the user.

6.5.2 include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** reads. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **include**s are supported.

6.5.3 SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named *s.makefile* or *s.Makefile* exists, **make** does a **get** on the file, then reads and removes the file.

6.5.4 Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current item to the left of the colon (which is **$@**). The form **$$(@F)** also exists, allowing access to the file part of **$@**. Thus, in the following example, the dependency is translated at execution time to the string **cat.c**.

```
cat:    $$@.c
```

This is useful for building a large number of executable files, each of which has only one source file. For instance, the operating system command directory could have a *makefile* like this:

```
CMDS = cat dd echo date cmp comm chown
```

```
$(CMDS):          $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single-file programs. For multiple-file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of *$$@*. Again, it is evaluated at execution time. Its usefulness becomes evident when you try to maintain the **/usr/include** directory from a makefile in the */usr/src/head* directory. Thus, the */usr/src/head/makefile* would look like this:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This *makefile* would completely maintain the */usr/include* directory whenever one of the above files in */usr/src/head* was updated.

6.6 Running make

To run **make**, use the following command syntax:

```
make [-f makefile] [-p] [-i] [-k] [-s] [-r] [-n] [-e] [-t] [-q] [names]
```

Refer to **make**(1) in the *Reference Manual* for more information on running **make** and for running **make** in parallel. The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

```
$ make [ options ] [ macro definitions ] [ targets ]
```

All macro definition arguments (arguments with embedded equals signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The options are as follows:

–f Description filename. The next argument is assumed to be the name of a description file. A filename of – denotes the standard input. If there are no **–f** arguments, the file named *makefile* or *Makefile* or *s.[mM]akefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

–p Print out the complete set of macro definitions and target descriptions.

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

–k Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

–r	Do not use the built-in rules.
–n	No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.
–e	Environmental variables override assignments within *makefile*s.
–t	Touch the target files (causing them to be up-to-date) rather than issue the usual commands.
–q	Question. The **make** command returns a zero or nonzero status code, depending on whether the target file is or is not up-to-date.

The following arguments are evaluated in the same manner as options:

.DEFAULT	If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.
.PRECIOUS	Dependents on this target are not removed when a quit or interrupt key is pressed.
.SILENT	Same effect as the **–s** option.
.IGNORE	Same effect as the **–i** option.

Finally, the remaining arguments are assumed to be the names of targets to be made and are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

6.6.1 Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence must be considered for this to work properly. The following paragraphs describe **make**'s interaction with the environment. A macro (environment variable), **MAKEFLAGS**, is maintained by **make**. The macro is defined as the collection of all input option arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line options and assignments in the *makefile* update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, the **MAKEFLAGS** macro must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input option argument and is processed as such. (The only exceptions are the **–f**, **–p**, and **–r** options.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).
4. Read the *makefile*(s). The assignments in the *makefile*(s) override the environment. This order is chosen so that when a *makefile* is read and executed, you know what to expect. That is, you get what is seen unless the **–e** option is used. The command line option **–e** tells **make** to have the environment override the *makefile* assignments. Thus, if **make –e ...** is typed, the variables in the environment override the definitions in the *makefile*. Also, **MAKEFLAGS** overrides the environment if assigned. This is useful for further invocations of **make** from the current *makefile*.

In summary, the precedence of assignments, in order from least binding to most binding, is as follows:

1. Internal definitions
2. Environment
3. *makefile*(s)
4. Command line

The **–e** option has the effect of rearranging the order:

1. Internal definitions
2. *makefile*(s)
3. Environment
4. Command line

This order is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

6.7 Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has a line like this:

```
#include "defs.h"
```

then the object file *x.o* depends on *defs.h*; the source file *x.c* does not. If *defs.h* is changed, nothing is done to the file *x.c* while file *x.o* must be recreated.

The **–n** option is useful in discovering what **make** would do. The following command orders **make** to print out the commands that **make** would issue without actually taking the time to execute them.

```
$ make -n
```

If a change to a file is certain to be small in character (for example, adding a comment to an *include* file), the **–t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the following command (touch silently) causes the relevant files to appear up-to-date.

```
$ make -ts
```

NOTE

Care is necessary because this mode of operation subverts the intention of **make** *and destroys all memory of the previous relationships.*

6.8 Internal Rules

The standard set of internal rules used by **make** are reproduced below:

```
#
#   SUFFIXES RECOGNIZED BY MAKE
#

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~

#
#   PREDEFINED MACROS
#
```

```
MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
#
#   SINGLE SUFFIX RULES
#

.c:
    $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@

.c~:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
    -rm -f $*.c

.f:
    $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@

.f~:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
    -rm -f $*.f

.sh:
    cp $< $@; chmod 0777 $@

.sh~:
    $(GET) $(GFLAGS) $<
    cp $*.sh $*; chmod 0777 $@
    -rm -f $*.sh
```

```
#
#   DOUBLE SUFFIX RULES
#

.c~.c  .f.f  .s.s  .sh.sh  .y.y  .l.l  .h.h:
	$(GET) $(GFLAGS) $<

.c.a:
	$(CC) -c $(CFLAGS) $<
	$(AR) $(ARFLAGS) $@ $*.o
	rm -f $*.o

.c~.a:
	$(GET) $(GFLAGS) $<
	$(CC) -c $(CFLAGS) $*.c
	$(AR) $(ARFLAGS) $@ $*.o
	rm -f $*.[co]

.c.o:
	$(CC) $(CFLAGS) -c $<

.c~.o:
	$(GET) $(GFLAGS) $<
	$(CC) $(CFLAGS) -c $*.c
	-rm -f $*.c

.f.a:
	$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
	$(AR) $(ARFLAGS) $@ $*.o
	-rm -f $*.o

.f~.a:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
	$(AR) $(ARFLAGS) $@ $*.o
	-rm -f $*.[fo]

.f.o:
	$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f

.f~.o:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
	-rm -f $*.f
```

```
.s~.a:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[so]

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    -rm -f $*.s

.l.c :
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@

.l~.c:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    mv lex.yy.c $@

.l.o:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    rm lex.yy.c
    mv lex.yy.o $@
    -rm -f $*.l

.l~.o:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    $(CC) $(CFLAGS) -c lex.yy.c
    rm -f lex.yy.c $*.l
    mv lex.yy.o $*.o

.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@

.y~.c :
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.y
    mv y.tab.c $*.c
    -rm -f $*.y
```

```
.y.o:
   $(YACC) $(YFLAGS) $<
   $(CC) $(CFLAGS) -c y.tab.c
   rm y.tab.c
   mv y.tab.o $@

.y~.o:
   $(GET) $(GFLAGS) $<
   $(YACC) $(YFLAGS) $*.y
   $(CC) $(CFLAGS) -c y.tab.c
   rm -f y.tab.c $*.y
   mv y.tab.o $*.o
```

Index

7. SCCS

7. SCCS

Chapter 7
SCCS

Figures

Tables

Chapter 7
SCCS

7.1 Introduction

The Source Code Control System (SCCS) is a maintenance and revision tracking tool that runs under the operating system. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and documentation.

The original file or any set of changes can be retrieved. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version: why the changes were made, who made them, when they were made.

This chapter covers the following topics:

- SCCS for Beginners: how to make, retrieve, and update an SCCS file
- Delta Numbering: how versions of an SCCS file are named
- SCCS Command Conventions: what rules apply to SCCS commands
- SCCS Commands: the SCCS commands and their more useful arguments
- SCCS Files: how to protect, format, and audit SCCS files

7.2 SCCS For Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

7.2.1 Terminology

A *delta* is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is automatically assigned an SID (SCCS IDentification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file (that is, its first delta) is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on.

A *tree* represents the entire SCCS structure for a file, including the trunk and all branches.

A *node* is any location on a tree where a delta or branching occurs.

A *trunk* of an SCCS tree is the main stem of an SCCS tree not including any branches that is, release numbers are in a straight progression with each delta dependent on the preceding deltas.

A *trunk delta* is a change dependent on all previous deltas. Trunk delta names contain a release number and a level number.

A *branch delta* is a change to a delta that is not dependent on all previous deltas. Branch delta names have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number.

A *leaf delta* is the last delta on a branch, that is, it has the highest sequence number.

7.2.2 Using admin to Create an SCCS File

Suppose, for example, you have a file called *lang* that is simply a list of five programming language names. Use a text editor to create file *lang* containing the following list:

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your *lang* file can be given to SCCS using the **admin** (*administer SCCS file*) command. The following command creates an SCCS file from the *lang* file:

```
$ admin -ilang s.lang
```

All SCCS files must have names that begin with *s.*, hence *s.lang*. The **–i** option, together with its value *lang*, tells **admin** to create an SCCS file and initialize it with the contents of the file *lang*.

The **admin** command replies:

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Its significance is described with the **get** command under “SCCS Commands” later in this chapter.

Remove the *lang* file. It is no longer needed because it exists now under SCCS as *s.lang*.

```
$ rm lang
```

7.2.3 Using get to Retrieve a File

Enter the **get** command as follows:

```
$ get s.lang
```

This retrieves *s.lang* and prints this message:

```
1.1
5 lines
```

The message tells you that **get** retrieved version 1.1 of the file, which contains five lines of text.

The retrieved text is placed in a new file known as a *g.file*. SCCS forms the *g.file* name by deleting the prefix *s.* from the name of the SCCS file and recreating the original file *lang*.

If you list the contents of your directory, you see both *lang* and *s.lang*. SCCS retains *s.lang* for use by other users. The **get s.lang** command creates *lang* as a read-only file and keeps no information regarding its creation.

The following **get** command informs SCCS that you are going to make changes to the file:

```
$ get –e s.lang
```

get –e causes SCCS to create *lang* for both reading and writing (editing). It also places certain information about *lang* in another new file, called the *p.file* (*p.lang* in this case), which is needed later by the **delta** command.

get –e prints the same messages as **get**, except that now the SID for the first delta you create is issued:

```
1.1
new delta  1.2
5 lines
```

Change *lang* by adding two more programming languages:

```
SNOBOL
ADA
```

7.2.4 Using delta to Record Changes

Now use the **delta** command as follows:

```
$ delta s.lang
```

delta prompts with this message:

```
comments?
```

Your response should be an explanation of why the changes were made. For example, you could type this line:

added more languages

delta now reads the *p.file*, *p.lang*, and determines what changes you made to *lang*. It does this by doing its own **get** to retrieve the original version and applying the **diff**(1) command to the original version and the edited version. Next, **delta** stores the changes in *s.lang* and destroys the no longer needed *p.lang* and *lang* files.

When this process is complete, **delta** outputs these messages:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to *s.lang*.

7.2.5 Additional Information about get

The following command retrieves the latest version of the file *s.lang*, now 1.2:

```
$ get  s.lang
```

SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, any of the following commands retrieve version 1.2:

```
$ get s.lang
$ get –r1 s.lang
$ get –r1.2 s.lang
```

The numbers following **–r** are SIDs. When you omit the level number of the SID (as in **get –r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command:

```
$ get  –e  –r2  s.lang
```

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output means that version 1.2 has been retrieved, and that 2.1 is the version **delta** creates:

```
1.2
new delta 2.1
7 lines
```

If the file is now edited, for example, by deleting COBOL from the list of languages, and **delta** is executed:

```
delta  s.lang
comments? deleted COBOL from list of languages
```

You will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, and so on), or another new release can be created in a similar manner.

7.2.6 The help Command

If the following command is executed, a message in output:

```
$ get lang
```

The message is as follows:

```
ERROR [lang]: not an SCCS file (co1)
```

The error message code, **co1**, can be used with **help** to get an explanation of the message:

```
help co1
```

This gives the following explanation of why **get lang** produced an error message:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s."
```

help is useful whenever there is doubt about the meaning of almost any SCCS message.

7.3 Delta Numbering

Deltas are like the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, and so on. The first number (1 in this case) is called the release number. The numbers after the decimal point (1, 2, and 3 in this case) are called the level numbers. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because you can change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 7-1.

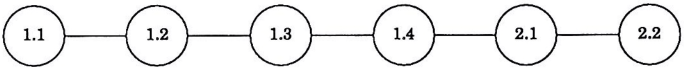


Figure 7-1. Evolution of an SCCS file. Note that a major change is indicated by a new release number (2) and that all subsequent deltas apply to that new number"

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the *trunk* of an SCCS tree.

Some situations require branching an SCCS tree. That is, changes are planned to a given delta that are not dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 7-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble are applied as a delta to version 1.3 (the version in production use). This creates a new version that is then released to the user but does not affect the changes being applied for release 2 (deltas 1.4, 2.1, 2.2, and so on). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 is 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, and so on. This is shown in Figure 7-2.

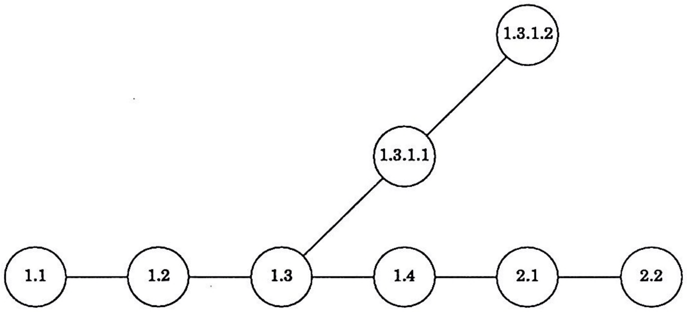


Figure 7-2. Tree structure with branch deltas.

The branch number is incremented only when a delta is created that starts a new branch off of an existing branch, as shown in Figure 7-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, and so on), but the secondary branch number remains the same.

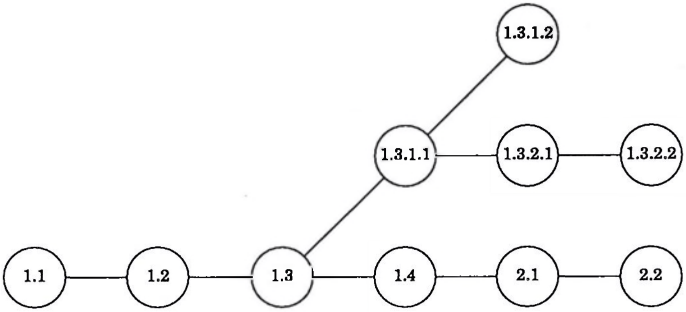


Figure 7-3. Extended branching concept.

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

7.4 SCCS Command Conventions

SCCS commands accept two types of arguments: command options and filenames. Command options begin with a minus sign (–) followed by a lowercase letter and, in some cases, a value.

Filenames or directory names specify the files that the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files in the named directories are silently ignored (because of permission modes using **chmod**(1)).

In general, filename arguments may not begin with a minus sign. If a filename of – (a minus sign) is specified, the command reads the standard input for lines and takes each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with the commands **find**(1) or **ls**(1), for example.

Command options are processed before filenames; therefore, options may be interspersed with filenames. Filenames are processed left to right. Somewhat different conventions apply to **help**(1), **what**(1), **sccsdiff**(1), and **val**(1), detailed later in this chapter under "SCCS Commands."

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of the flags are discussed in this chapter; for a complete description, refer to **admin**(1) in the *Reference Manual.*

The distinction between real user (**passwd**(1)) and effective user is of concern in discussing various actions of SCCS commands. For this discussion, assume that the real user and the effective users are the same—the person logged into the operating system (refer to "Protection" later in this chapter).

7.4.1 x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the *x.file*. This ensures that the SCCS file is not damaged if processing terminates abnormally. SCCS names the *x.file* by replacing the *s.* of the SCCS filename with a *x.* prefix. The *x.file* is created in the same directory as the SCCS file, given the same mode (see **chmod**(1)), and is owned by the effective user. When processing is complete, the old SCCS file is destroyed and the modified *x.file* is renamed (*x.* is relaced by *s.*); it becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the *z.file*. SCCS forms its name by replacing the **s.** of the SCCS filename with a *z.* prefix. The *z.file* contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. The *z.file* is created with access permission mode 444 (read only) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, you can ignore *x.files* and *z.files*. They are useful only in the event of system crashes or similar situations.

7.4.2 Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

7.5 SCCS Commands

This section describes the major features of the SCCS commands and their most common arguments. Full descriptions of each command are in the *Reference Manual.*

Here is a quick-reference overview of the commands:

get	Retrieves versions of SCCS files
unget	Undoes the effect of a **get –e** prior to the file being **delta**ed
delta	Applies deltas (changes) to SCCS files and creates new versions
admin	Initializes SCCS files, manipulates their descriptive text, and controls delta creation rights
prs	Prints portions of an SCCS file in user-specified format
sact	Prints information about files that are currently out for edit
help	Gives explanations of error messages
rmdel	Removes a delta from an SCCS file, allowing removal of deltas created by mistake
cdc	Changes the commentary associated with a delta
what	Searches any operating system file(s) for all occurrences of a special pattern and prints out what follows it (useful for identifying information inserted by the **get** command)
sccsdiff	Shows differences between any two versions of an SCCS file

comb	Combines consecutive deltas into one to reduce the size of an SCCS file
val	Validates an SCCS file
vc	Specifies a filter for version control

7.5.1 The get Command

The **get**(1) command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the *g.file*. It is created in the current directory and is owned by the real user. The mode assigned to the *g.file* depends on how the **get** command is used.

The most common use of **get** is as follows:

```
$ get s.abc
```

This command retrieves the latest version of file *abc* from the SCCS file tree trunk and produces output like the following on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

The output shows that version 1.3 of file *s.abc* was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The generated *g.file* (file *abc*) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce *g.file*s only for inspection, compilation, and so on. It is not intended for editing.

When several files are specified, the same information is output for each one. The command

```
$ get s.abc s.xyz
```

produces the following output:

```
s.abc:
1.3
67 lines
No id keywords (cm7)
```

```
s.xyz:
1.7
85 lines
No id keywords (cm7)
```

Identification Keywords

In generating a *g.file* for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, and so forth, within the *g.file*. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example, **%I%** is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, **%H%** and **%M%** are the current date and name of the *g.file*. Consider the following PL/I language declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

Executing **get** on an SCCS file containing this declaration gives (for example) the following output:

```
DCL  ID  CHAR(100)  VAR  INIT('MODNAME  2.3  07/18/85');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the ID keywords, refer to **get**(1) in the *Reference Manual.*

Retrieval of Different Versions

By default, **get** retrieves the most recently created delta of the highest-numbered trunk release of an SCCS file. Any other version can be retrieved with **get –r** by specifying the version's SID. The following command retrieves version 1.3 of file *s.abc*:

```
$ get -r1.3 s.abc
```

The following message is displayed on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly, as follows:

```
$ get -r1.5.2.3 s.abc
```

It produces a message like this on the standard output:

```
1.5.2.3
234 lines
```

When an SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in the following command, causes retrieval of the trunk delta with the highest-level number within the given release.

```
$ get -r3 s.abc
```

Thus, the above command might produce this output:

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume that release 9 does not exist in file *s.abc* and release 7 is the highest-numbered release below 9. Executing the following command produces output indicating that trunk delta 7.6 is the latest version of file *s.abc* below release 9, as follows:

```
$ get -r9 s.abc
7.6
420 lines
```

Similarly, omitting the sequence number, as in the following command, results in the retrieval of the branch delta with the highest sequence number on the given branch. Consider this command:

```
$ get -r4.3.2 s.abc
```

This might result in the following output:

```
4.3.2.8
89 lines
```

If the branch does not exist, an error message results.

get –t retrieves the latest (top) version of a particular release when no **–r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5, the following command might produce this output:

```
$ get -r3 -t s.abc
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce this output:

```
3.2.1.5
46 lines
```

Retrieval with Intent to Make a Delta

Invoking **get** with the **-e** option indicates an intent to make a delta. First, **get** checks the user list to determine if the login name or group ID of the person executing **get** is present. The login name or group ID must be present for the user to be allowed to make deltas. (Refer to "The **admin** Command", later in this chapter, for a discussion of making user lists.) Second, **get** checks to make sure that the release number (R) of the version being retrieved satisfies the following relation:

```
floor is less than or equal to R, which is
less than or equal to ceiling
```

By checking this relation, **get** determines if the release being accessed is a protected release. The floor and ceiling are flags in the SCCS file representing start and end of range. Third, **get** checks to see that the R is not locked against editing. The lock is a flag in the SCCS file. Last, **get** checks to see whether multiple concurrent edits are allowed for the SCCS file by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the checks succeed, **get -e** causes the creation of a *g.file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable *g.file* already exists, **get** terminates with an error. This prevents inadvertent destruction of a *g.file* being edited for the purpose of making a delta.

Any ID keywords appearing in the *g.file* are not substituted by **get -e** because the generated *g.file* is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the *g.file*. Thus, the following message is never output when **get -e** is used.

```
No id keywords (cm7)
```

In addition, **get –e** causes the creation (or updating) of a *p.file* that is used to pass information to the **delta** command.

The following command produces a message like this on the standard output:

```
$ get –e s.abc
1.3
new delta 1.4
67 lines
```

Undoing a get –e

There may be times when a file is retrieved for editing in error; there is really no editing that needs to be done at this time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **get –r** or **get** –t are used with **–e**, the version retrieved for editing is the one specified by **–r** or –t.

get –i and **–x** are used to specify a list of deltas to be included and excluded, respectively. Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. For example, two deltas can interfere when each one changes the same line of the retrieved *g.file*. A warning shows the range of lines within the retrieved *g.file* where the problem may exist. You should examine the *g.file* to determine what the problem is and what action to take, for example, editing the file.

NOTE

Use **get –i** *and* **get –x** *with extreme care.*

get –k is used either to regenerate a *g.file* that may have been accidentally removed or ruined after **get –e**, or simply to generate a *g.file* in which the replacement of ID keywords has been suppressed. A *g.file* generated by **get –k** is identical to one produced by **get –e**, but no processing related to the *p.file* takes place.

Concurrent Edits of Different Deltas

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get –e** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p.file* created by **get –e** is named by automatic replacement of the SCCS filename's prefix *s.* with a *p.* prefix. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The *p.file* contains the following information for each delta that is still in progress:

- The SID of the retrieved version
- The SID given to the new delta when it is created
- The login name of the real user executing **get**

The first execution of **get –e** causes the creation of a *p.file* for the corresponding SCCS file. Subsequent executions only update the *p.file* with a line containing the above information. Before updating, however, **get** checks to assure that no entry already in the *p.file* specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, you are informed that other deltas are in progress and processing continues. If the check fails, an error message results.

Note that concurrent executions of **get** must be done from different directories. Subsequent executions from the same directory attempt to overwrite the *g.file*, which is an SCCS error condition. In practice, this problem does not arise since each user normally has a different working directory. Refer to “Protection”, later in this chapter, for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 7-1 shows the possible SID components a user can specify with **get**, the version that is then retrieved by **get**, and the resulting SID for the delta that **delta** creates.

Table 7-1
Determination of New SID

SID Specified in get*	–b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R< mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1

* R, L, B, and S mean release, level, branch, and sequence numbers in the SID; m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

† The –**b** option is effective only if the **b** flag (see **admin(1)**) is present in the file. An entry of – means irrelevant.

‡ This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this table applies.

§ This is used to force the creation of the first delta in a new release.

** hR is the highest existing release that is lower than the specified, nonexistent release R.

Table 7-1
Determination of New SID (cont.)

SID Specified in get*	–b Key-Letter Used†	Other Condition	SID Retrieved by get	SID of Delta to be Created by delta
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

* R, L, B, and S mean release, level, branch, and sequence numbers in the SID; m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

† The –b option is effective only if the **b** flag (see **admin(1)**) is present in the file. An entry of – means irrelevant.

Concurrent Edits of Same SID

Under normal conditions, more than one **get –e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get –e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus, the second **get** command below can immediately follow the first without an intervening **delta** command:

```
$ get -e s.abc
1.1
new delta 1.2
5 lines

$ get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

In this case, a **delta** command after the first **get** produces delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **delta** after the second **get** produces delta 1.1.1.1.

Command Options that Affect Output

get –p causes the retrieved text to be written to the standard output rather than to a *g.file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. For example, **get –p** is used to create a *g.file* with an arbitrary name:

```
$ get -p s.abc > arbitrary-file-name
```

get –s suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. **get –s** is used to prevent nondiagnostic messages from appearing on the your terminal and is often used with **–p** to pipe the output, as follows:

```
$ get -p -s s.abc | pg
```

get –g suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file, the following command outputs the SID 4.3, if it exists, in the SCCS file *s.abc* or an error message if it does not:

```
$ get -g -r4.3 s.abc
```

Another use of **get –g** is in regenerating a *p.file* that may have been accidentally destroyed, as follows:

```
$ get -e -g s.abc
```

get –l causes SCCS to create an *l.file*. It is named by replacing the *s.* of the SCCS filename with *l.*, created in the current directory with mode 444 (read only) and owned by the real user. The *l.file* contains a table (whose format is described under **get**(1). showing the deltas used in constructing a particular version of the SCCS file. For example, the following command generates an *l.file* showing the deltas applied to retrieve version 2.3 of file *s.abc*:

```
$ get -r2.3 -l s.abc
```

Specifying **p** with **–l** causes the output to be written to the standard output rather than to the *l.file*, as follows:

```
$ get -lp -r2.3 s.abc
```

get –g can be used with **–l** to suppress the retrieval of the text.

get –m identifies the changes applied to an SCCS file. Each line of the *g.file* is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

get –n causes each line of a *g.file* to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep**(1). For example, the following commands find all lines that match a given pattern in the latest version of each SCCS file in a directory:

```
$ get -p -n -s directory | grep pattern
```

If both **–m** and **–n** are specified, each line of the generated *g.file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of **–n**) and is followed by the line in the format produced by **–m**. Because use of **–m** and/or **–n** causes the contents of the *g.file* to be modified, such a *g.file* must not be used for creating a delta. Therefore, neither **–m** nor **–n** may be specified together with **get –e**.

Refer to **get**(1) for a full description of additional command options.

7.5.2 The delta Command

The **delta**(1) command is used to incorporate changes made to a *g.file* into the corresponding SCCS file—that is, to create a delta and, therefore, a new version of the file.

The **delta** command requires the existence of a p.file (created via **get –e**). It examines the *p.file* to verify the presence of an entry containing the user's login name. If none is found, an error message results.

The **delta** command performs the same permission checks that **get –e** performs. If all checks are successful, **delta** determines what has been changed in the *g.file* by using **diff**(1) to compare it with its own temporary copy of the *g.file* as it was before editing. This temporary copy of the *g.file* is called the *d.file* and is obtained by performing an internal **get** on the SID specified in the *p.file* entry.

The required *p.file* entry is the one containing the login name of the user executing **delta**, because the user who retrieved the *g.file* must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get –e** more than once on the same SCCS file. Then, **delta –r** must be used to specify the SID that uniquely identifies the *p.file* entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is as follows:

```
$ delta s.abc
```

The **delta** command prompts for comments:

```
comments?
```

You reply with a description of why the delta is being made, ending the reply with a newline character. Your response may be up to 512 characters long with newlines, not intended to terminate the response, escaped by backslashes (\).

If the SCCS file has a **v** flag, **delta** first prompts with the Modification Requests (MRs) prompt on the standard output:

```
MRs?
```

The standard input is read for MR numbers, separated by blanks or tabs, and ended with a newline character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

delta –y or **delta –m** can be used to enter comments and MR numbers on the command line rather than through the standard input, as follows:

$ **delta** –y"*descriptive comment*" **–m**"*mrnum1 mrnum2*" **s.abc**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two options are useful when **delta** is executed from within a shell procedure.

NOTE

delta –m *is allowed only if the SCCS file has a* **v** *flag.*

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the *p.file*) and the number of lines inserted, deleted, and left unchanged, for example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the your perception of the changes made to a *3g.file*, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the *g.file*. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited *g.file*.

If you are in the process of making a delta and the **delta** command finds no ID keywords in the edited *g.file*, the following message is issued after the prompts for commentary but before any other output:

```
No id keywords (cm7)
```

This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a *g.file* that was created by a **get** without **-e** (ID keywords are replaced by **get** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the *g.file*. Or, it is possible that the file had no ID keywords. In any case, the delta is created unless there is an **i** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta is not created.

After the processing of an SCCS file is complete, the corresponding *p.file* entry is removed from the *p.file*. All updates to the *p.file* are made to a temporary copy, the *q.file*, whose use is similar to the use of the *x.file* described earlier under "SCCS Command Conventions." If there is only one entry in the *p.file*, then the *p.file* itself is removed.

In addition, **delta** removes the edited *g.file* unless **-n** is specified. For example, the following command keeps the *g.file* after processing:

```
$ delta -n s.abc
```

delta -s suppresses all output normally directed to the standard output, other than `comments?` and `MRs?`. Thus, use of **-s** with **-y** (and **-m**) keeps **delta** from reading standard input or writing standard output.

The differences between the *g.file* and the *d.file* constitute the delta and may be printed on the standard output by using **delta -p**. The format of this output is similar to that produced by **diff**(1).

7.5.3 The admin Command

The **admin**(1) command is used to administer SCCS files—that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of options with **admin** or are assigned default values if no options are supplied. The same options are used to change the parameters of existing SCCS files.

Two command options are used in detecting and correcting corrupted SCCS files. (Refer to "Auditing" later in this chapter.)

Newly created SCCS files are given access permission mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

7.5.4 Creation of SCCS files

An SCCS file can be created by executing a command in this form:

```
$ admin -ifirst s.abc
```

The value **first** with **–i** is the name of a file from which the text of the initial delta of the SCCS file *s.abc* is to be taken. Omission of a value with **–i** means **admin** is to read the standard input for the text of the initial delta.

The following command is equivalent to the previous example:

```
$ admin -i s.abc < first
```

If the text of the initial delta does not contain ID keywords, the following message is issued by **admin** as a warning:

```
No id keywords (cm7)
```

However, if the command also sets the **i** flag (not to be confused with the **–i** option), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin –i**.

admin –r is used to specify a release number for the first delta. Thus, the following command means that the first delta should be named 3.1 rather than the normal 1.1.

```
$ admin -ifirst -r3 s.abc
```

Because **–r** has meaning only when creating the first delta, its use is permitted only with **–i**.

Inserting Commentary for the Initial Delta

When you create an SCCS file, you may want to record why you did it. Comments (**admin –y**) and MR numbers (**–m**) can be entered in exactly the same way as a **delta**.

If **–y** is omitted, a comment line in the following form is automatically generated:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

If you want to supply MR numbers (**admin –m**), the **v** flag must be set via **–f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary in the SCCS file (refer to **sccsfile**(4) in the *Reference Manual*):

```
$ admin -ifirst -mmrnum1 -fv s.abc
```

Note that **–y** and **–m** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin –t**.

When an SCCS file is first being created and **–t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the following command specifies that the descriptive text is to be taken from file *desc*:

```
$ admin -ifirst -tdesc s.abc
```

When processing an existing SCCS file, **–t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus, the following command specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*:

```
$ admin -tdesc s.abc
```

Omission of the filename after the **–t** option, as in the following command, causes the removal of the descriptive text from the SCCS file:

```
$ admin -t s.abc
```

The flags of an SCCS file may be initialized or changed by **admin –f**, or deleted with **–d**.

SCCS file flags are used to direct certain actions of the various commands. Refer to **admin**(1) for a description of all the flags. For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

admin –f is used to set flags and, if desired, their values. For example, the following command sets the **i** and **m** (module name) flags:

```
$ admin -ifirst -fi -fmmodname s.abc
```

The value *modname* specified for the **m** flag is the value that the **get** command uses to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the **%M%** ID keyword.) Several **–f** options may be supplied on a single **admin** command line, and they may be used whether the command is creating a new SCCS file or processing an existing one.

admin –d is used to delete a flag from an existing SCCS file. As an example, the following command removes the **m** flag from the SCCS file:

```
$ admin -dm s.abc
```

Several **–d** options may be used on an **admin** command line and may be intermixed with **–f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **admin –a** is used. For example, the following command adds the login names **xyz** and **wql** and the group ID **1234** to the list:

```
$ admin -axyz -awql -a1234 s.abc
```

admin –a (add) may be used whether creating a new SCCS file or processing an existing one.

admin –e (erase) is used to remove login names or group IDs from the list.

7.5.5 The prs Command

The **prs**(1) command prints all or part of an SCCS file on the standard output. If **prs –d** is used, the output is in a format called *data specification*. Data specification is a string of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. The following example is defined as the data keyword replaced by the SID of a specified delta:

```
:I:
```

Similarly, **:F:** is the data keyword for the SCCS filename currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of keywords, refer to **prs**(1) in the *Reference Manual.*

There is no limit to the number of times a data keyword may appear in a data specification. For example, this **prs** command might produce the following output:

```
$ prs -d":I: this is the top delta for :F: :I:" s.abc
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **prs –r**. For example, this **prs** command might produce the following output:

```
$ prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **–r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **prs –l** or **prs –e**. The use of **prs –e** substitutes data keywords for the SID designated with **–r** and all deltas created earlier, while **prs –l** substitutes data keywords for the SID designated with **–r** and all deltas created later. Thus, the following command might produce the following output:

```
$ prs -d:I: -r1.4 -e s.abc
1.4
1.3
1.2.1.1
1.2
1.1
```

And this command might produce this output:

```
$ prs -d:I: -r1.4 -l s.abc
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **–e** and **–l**.

7.5.6 The sact Command

sact(1) is a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get –e** command, and the date and time the **get –e** was executed. It is a useful command for an administrator.

7.5.7 The help Command

The **help**(1) command prints the syntax of SCCS commands and of messages that appear on your terminal. Arguments to **help** are either SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. Explanatory information is printed on the standard output. If no information is found, an error message is printed. When more than one argument is used, each is processed independently, and an error resulting from one does not stop the processing of the others.

NOTE

There is no conflict between the **help** *command of SCCS and the operating system* **help***(1) utilities. The installation procedure for each package checks for the prior existence of the other.*

Explanatory information related to a command is a synopsis of the command. For example, consider the following command and its output:

```
$ help ge5 rmdel
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
  rmdel –rSID  name ...
```

7.5.8 The rmdel Command

The **rmdel**(1) command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a *leaf delta*. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 7-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **–r** option is mandatory with **rmdel**. It is used to specify the complete SID of the delta to be removed. Thus, the following command specifies the removal of trunk delta 2.3:

```
$ rmdel –r2.3 s.abc
```

Before removing the delta, **rmdel** checks that the release number (R) of the given SID satisfies this relation:

```
floor less than or equal to R less than or equal to ceiling
```

The **rmdel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list, or the user list must be empty. Also, the release specified cannot be locked against editing; if the **l** flag is set, the release must not be contained in the list (refer to **admin**(1)). If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

7.5.9 The cdc Command

The **cdc**(1) command is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (that is, **–r** and full SID are necessary), although the delta need not be a leaf delta. For example, the following command specifies that the commentary of delta 3.4 is to be changed:

```
$ cdc –r3.4 s.abc
```

New commentary is then prompted for as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus, the following command inserts **mrnum3** and deletes **mrnum1** for delta 1.4:

```
$ cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number, inserted correct MR number
```

The `MRs?` prompt appears only if the **v** flag has been set. Modification Requests are described earlier in this chapter.

7.5.10 The what Command

The **what**(1) command is used to find identifying information within any file whose name is given as an argument. No command options are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the *%Z%* ID keyword (refer to **get**(1)). **what** prints, on standard output, whatever follows the string until the first double quote ("), greater-than sign (>), backslash (\), newline, or nonprinting NULL character.

For example, consider an SCCS file called *s.prog.c* containing the following line:

```
char  id[]= "%W%";
```

The following command results in a *g.file* that is is used to produce *prog.o* and *a.out*:

```
$ get -r3.4 s.prog.c
```

Then, this **what** command produces the following output:

```
$ what prog.c prog.o a.out
prog.c:
  prog.c: 3.4
prog.o:
  prog.c: 3.4
a.out:
  prog.c: 3.4
```

The string searched for by **what** need not be inserted with an ID keyword of **get**; it can be inserted in any convenient manner.

7.5.11 The sccsdiff Command

The **sccsdiff** command prints, on standard output, differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff –r** in the same way as with **get –r**. SID numbers must be specified as the first two arguments. Any following options are interpreted as arguments to the **pr**(1) command (which prints the differences) and must appear before any filenames. The SCCS files to be processed are named last. Directory names and a name of – (a minus sign) are not acceptable to **sccsdiff**.

The following is an example of the the **sccsdiff** command:

```
$ sccsdiff -r3.4 -r5.6 s.abc
```

The differences are printed in the same format as **diff**(1).

7.5.12 The comb Command

The **comb**(1) command lets you try to reduce the size of an SCCS file. It generates a shell procedure on standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any command options, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 7-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

The following are some of the options you can use with this command:

comb –s	Generates a shell procedure that produces a report of the percentage of space (if any) that would be saved. This is often useful as an advance step.
comb –p	Allows you to specify the oldest delta you want preserved.
comb –c	Allows you to specify a list of deltas you want preserved. All other deltas are discarded. (Refer to **get**(1) for the delta list syntax.)

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

7.5.13 The val Command

The **val**(1) command is used to determine whether a file is an SCCS file meeting the characteristics specified by certain command options. It checks for the existence of a particular delta when the SID for that delta is specified with **–r**.

The string following **–y** or **–m** is used to check the value set by the **t** or **m** flag, respectively. Refer to **admin**(1) for a description of these flags.

The **val** command treats the special argument – (minus sign) differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until a Ctrl-D (end-of-file) is entered. This permits one **val** command with different values for command options and file arguments. For example, the following command line first checks if file *s.abc* has a value **c** for its type flag and value **abc** for the module name flag:

```
$ val -
-yc -mabc s.abc
-mxyz -ypl1 s.xyz
Ctrl-D
```

Once this is done, **val** processes the remaining file, in this case *s.xyz*.

The **val** command returns an 8-bit code. Each bit set shows a specific error. In addition, an appropriate diagnostic is printed unless suppressed by **–s**. (Refer to **val**(1) for a description of errors and codes.) A return code of 0 means all files met the characteristics specified.

7.5.14 The vc Command

The **vc**(1) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *Reference Manual.*

7.6 SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

7.6.1 Protection

SCCS relies on the capabilities of the operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files—that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the *x.file*; refer to "SCCS Command Conventions"). When processing is done, the old file is automatically removed and the *x.file* renamed (s. prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (for example, on large development projects), one user—that is, one user ID—must be chosen as the owner of the SCCS files. This person administers the files (that is, uses the **admin** command) and is SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and, if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the "set user ID on execution" bit on (refer to **chmod**(1) in the *Reference Manual*). This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**. A project-dependent interface program could be custom-built for each project.

7.6.2 Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the logical sum of all the characters of the file (not including the checksum itself)
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas
Flags	Indicators that control certain actions of SCCS commands
Descriptive Text	Usually a summary of the contents and purpose of the file
Body	The text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **sccsfile**(4). The checksum is discussed under "Auditing," following.

Since SCCS files are ASCII files, they can be processed by non-SCCS commands like **ed**(1), **grep**(1), and **cat**(1). This is convenient when an SCCS file must be modified manually (for example, a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when you simply want to look at the file.

CAUTION

Use extreme care when modifying SCCS files with non-SCCS commands. Using non-SCCS commands may damage the SCCS tree structure or SCCS files.

7.6.3 Auditing

When a system or hardware malfunction destroys an SCCS file, any command issues an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with **ed**(1)). No SCCS command processes a corrupted SCCS file except the **admin** command with **–h** or **–z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin –h** and specify all SCCS files:

```
$ admin -h s.file1 s.file2 ...
    or
$ admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the following message is produced for that file:

```
corrupted file (co6)
```

The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process does not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(1) command periodically and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

```
$ admin -z s.file
```

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file is no longer detectable.

Index