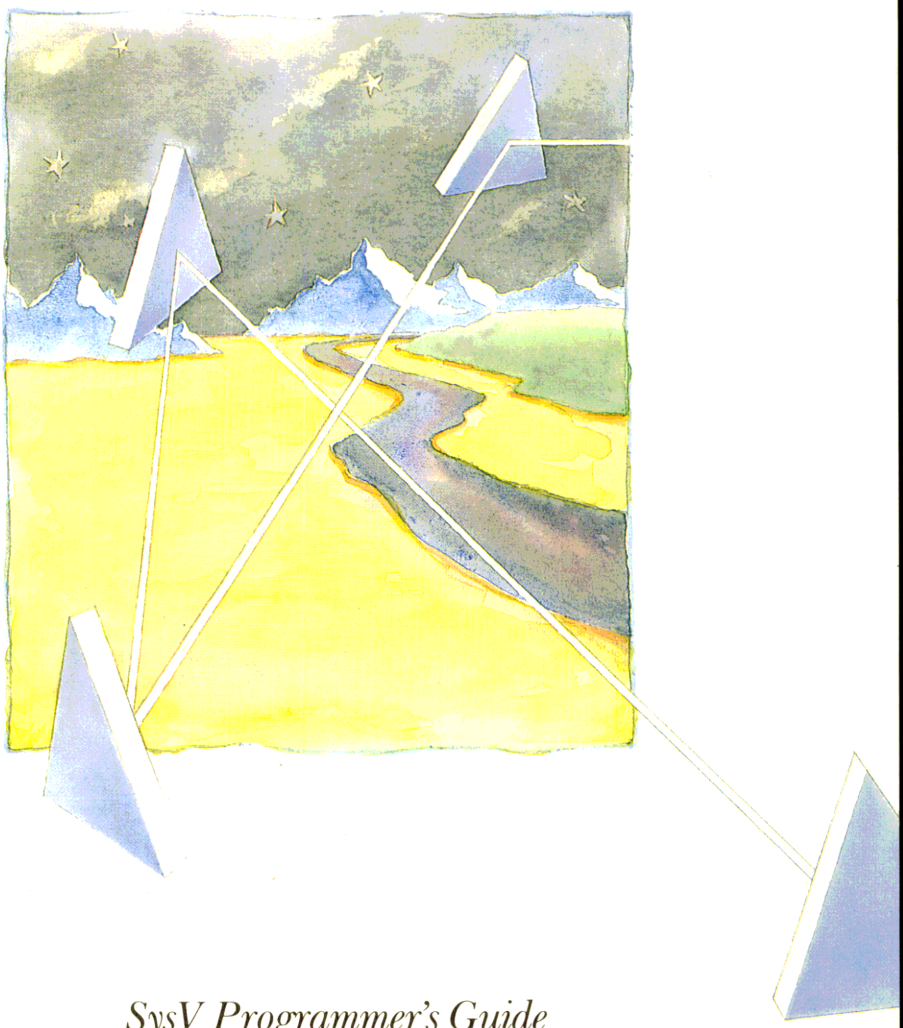

SysV Programmer's Guide
Volume I

apollo

SysV Programmer's Guide: Volume I

Order No. 017270-A00

First Printing: November 1989

10 9 8 7 6 5 4 3 2 1

Preface

The *SysV Programmer's Guide: Volume I* is the first book of a two-volume set that describes programming utilities available in Domain/OS System V. This manual is intended for programmers who are familiar with System V software and Domain/OS. It provides neither a general overview of Domain/OS SysV nor details of the implementation of the system. We assume that you are already familiar with the material in *Using Your SysV Environment.*

This volume is organized as follows:

Chapter 1	Describes the **awk** pattern scanning and processing language.
Chapter 2	Describes the **nawk** (**new awk**) pattern scanning and processing language.
Chapter 3	Describes the **lex** generator of lexical analysis programs.
Chapter 4	Describes the **yacc** compiler.
Chapter 5	Describes SysV file and record locking.
Chapter 6	Describes SysV inter-process communication.

The following chapters are available in the *SysV Programmer's Guide: Volume II:*

Chapter 7	Describes the **curses** and **terminfo** utilities.
Chapter 8	Describes the **make** utility.
Chapter 9	Describes **sccs** (source code control system).

Chapter 10	Describes the **mk** utility.
Appendix A	Describes the **dbx** utility.

Related Manuals

The file **/install/doc/apollo/os.v.***latest software release number***__manuals** lists current titles and revisions for all available manuals.

For example, at Software Release 10.2 (SR10.2) you may refer to the file **/install/doc/apollo/os.v.10.2__manuals** to check that you are using the correct version of manuals. You may also want to use this file to check that you have ordered all of the manuals that you need.

The same information is available online. In the UNIX environment, type **man manuals**. In the Aegis environment, **help manuals**.

Refer to the *Domain Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents.

For introductory information about the Domain/OS system and details about using the SysV environment, refer to the following documents:

- *Getting Started with Domain/OS* (2348)
- *Using Your SysV Environment* (11020)
- *SysV User's Guide*, Volumes I and II (017269 and 017624)
- *SysV Command Reference* (005798)
- *Domain Display Manager Command Reference* (11418)

For more information on programming in the Domain/OS SysV environment, refer to the following documents:

- *Domain/OS Call Reference*, Volumes 1 and 2 (7196 and 12888)
- *Domain/OS Programming Environment Reference* (11010)
- *Domain Binder and Librarian Reference* (4977)
- *Domain C Language Reference* (2093)
- *SysV Programmer's Guide*, Volumes I and II (017270 and 017625)
- *SysV Programmer's Reference* (005799)

- *Managing SysV System Software* (10853)

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** shell command description. You can view the same description online by typing:

$ man 1 mkapr (in the SysV environment)

% man 1 mkapr (in the BSD environment)

$ help mkapr (in the Aegis environment)

Alternatively, you may use the Reader's Response form at the back of this manual to submit comments about the manual.

Documentation Conventions

This manual uses the following symbolic conventions:

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Bold words in text indicate the first use of a new term. Filenames and pathnames are also in bold.
user-supplied values	Placeholders for symbols that you must supply are printed in italics. For example, the names chosen for call arguments appear in italics.
sample user input	In samples, information that the user enters appears in bold.
`examples`	Examples of program code and program output appear in `this typeface`.
[ ]	Square brackets enclose optional items in formats and command descriptions.

{ }	Braces enclose a list from which you must choose an item in formats and command descriptions.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/ ^	The notation CTRL/ or ^ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
...	Ellipses mean that the previous argument-prototype may be repeated.
⋮	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
–	An argument beginning with a dash ("–") usually means that it is an option-specifying argument used by the command itself, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with "–".

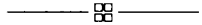

Table of Contents: Volume I

Chapter 1: **awk**

Chapter 2: **nawk**

Chapter 3: **lex**

Chapter 4: **yacc**

Chapter 5: File and Record Locking

Chapter 6: Inter-process Communication

Table of Contents: Volume II

Chapter 7: **curses/terminfo**

Chapter 8: **make**

Chapter 9: **mk**

Chapter 10: **sccs**

Appendix A: **dbx**

Chapter 1
awk

Chapter 1: awk

The **awk** Programming Language 1-1
- Program Structure 1-1
- Lexical Units 1-2
  - Numeric Constants 1-2
  - String Constants 1-3
  - Keywords 1-3
  - Identifiers 1-3
  - Operators 1-3
  - Record and Field Tokens 1-6
  - Comments 1-7
  - Tokens Used for Grouping 1-7
- Primary Expressions 1-8
  - Numeric Constants 1-8
  - String Constants 1-8
  - Variables 1-9
  - Functions 1-10
- Terms 1-12
  - Binary Terms 1-12
  - Unary Term 1-13
  - Incremented Vars 1-13
  - Parenthesized Terms 1-13
- Expressions 1-13
  - Concatenation of Terms 1-14
  - Assignment Expressions 1-14

Using **awk** 1-15

Input and Output 1-16
- Presenting Your Program for Processing 1-16
- Input: Records and Fields 1-17
- Sample Input File, **countries** 1-17
- Input: From the Command Line 1-19
- Output: Printing 1-20
- Output: to Different Files 1-25
- Output: to Pipes 1-25

Patterns 1-27
- **BEGIN** and **END** 1-27
- Relational Expressions 1-28
- Regular Expressions 1-29
- Combinations of Patterns 1-31
- Pattern Ranges 1-32

Actions 1-33
- Variables, Expressions, and Assignments 1-33
- Initialization of Variables 1-34
- Field Variables 1-35
- String Concatenation 1-36
- Special Variables 1-36
- Type 1-37
- Arrays 1-38

Special Features 1-40
- Built-in Functions 1-40
- Flow of Control 1-42
- Report Generation 1-45
- Cooperation with the Shell 1-47
- Multidimensional Arrays 1-48

Introduction

awk is a file-processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. **awk**:

- generates reports
- matches patterns
- validates data
- filters data for transmission

In the first part of this chapter, we give a general statement of the **awk** syntax. Then, under the heading "Using **awk**," we provide a number of examples that show the syntax rules in use.

Program Structure

An **awk** program is a sequence of statements of the form

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

awk runs on a set of input files. The basic operation of **awk** is to scan a set of 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

```
/x/  {print}
```

prints every input line that has an **x** in it.

An **awk** program has the following structure:

- a **BEGIN** section
- a **record** or main section
- an **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.

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 nonnull 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 nonnull sequence of digits as in **12e3**, **1.2e3**, **1.2e–3**, and **1.2E+3**. The maximum size and precision of a numeric constant 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 (almost) any length.

Keywords

Strings used as keywords are shown in Figure 1-1.

Keywords

BEGIN	**break**	**log**
END	**close**	**next**
FILENAME	**continue**	**number**
FS	**exit**	**print**
NF	**exp**	**printf**
NR	**for**	**split**
OFS	**getline**	**sprintf**
ORS	**if**	**sqrt**
OFMT	**in**	**string**
RS	**index**	**substr**
	int	**while**
	length	

Figure 1-1: **awk** Keywords

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 Figure 1-2.

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
++	prefix and postfix increments	++X and X++ are similar to X=X+1
—	prefix and postfix decrements	—X and X— are similar to X = X −1

Figure 1-2: **awk** Assignment Operators

Arithmetic operators are shown in Figure 1-3.

Symbol	Description
+	unary and binary plus
–	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

Figure 1-3: **awk** Arithmetic Operators

Relational operators are shown in Figure 1-4.

Symbol	Description
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Figure 1-4: **awk** Relational Operators

Logical operators are shown in Figure 1-5.

Symbol	Description
&&	and
||	or
!	not

Figure 1-5: **awk** Logical Operators

Regular expression matching operators are shown in the Figure 1-6.

Symbol	Description
~	matches
!~	does not match

Figure 1-6: Operators for Matching Regular Expressions in **awk**

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, as its value, 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, i.e., any nonnull 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 \t. The assignment statement **FS** =" " makes white space (a tab or blank) the field separator; and 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 nonnull 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 nonnull 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.

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 below.

Numeric Constants

The format of a numeric constant was defined previously in "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 in Figure 1-7.

Numeric Constant	Numeric Value	String Value
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

Figure 1-7: Numeric Values for String Constants

String Constants

The format of a string constant was defined previously 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 in Figure 1-8.

String Constant	Numeric Value	String Value
""	0	empty space
"a"	0	a
"XYZ"	0	XYZ
"0"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	50	.5e2

Figure 1-8: String Values for String Constants

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 ith field of the current input record. As with **$0**, if the ith field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the ith field is a number. **$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 arithmetic functions are in Figure 1-9.

Functions
exp (*expression*)
int (*expression*)
log (*expression*)
sqrt (*expression*)

Figure 1-9: Built-in Functions for Arithmetic and String Operations

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. String functions are shown in Figure 1-10.

String Functions

getline
index(*expression1*, *expression2*)
length(*expression*)
split(*expression*, *identifier*, *expression2*)
split(*expression*, *identifier*)
sprintf(*format*, *expression1*, *expression2...*)
substr(*expression1*, *expression2*)
substr(*expression1*, *expression2*, *expression3*)

Figure 1-10: **awk** String Functions

The function **getline** causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 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** 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 The function **split** uses the current value of **FS** to indicate the field separator. For example, after invoking

```
n = split ($0)
```

a[1], a[2], ..., a[*n*] is the same sequence of values as **$1**, **$2** ..., **$NF**.

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 programming language (except that the use of the asterisk, *, for field width or precision is not allowed).

The function **substr** returns the suffix of *string* starting at position The function **substr** 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** is equivalent to **substr** For example:

```
substr("abc", 2, 1) = "b"
substr("abc", 2, 2) = "bc"
substr("abc", 2, 3) = "bc"
```

Positions less than 1 are taken as 1. 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.

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
term binop term
unop term
incremented variable
(term)

Binary Terms

In a term of the form

term1 binop term2

binop can be one of the five binary arithmetic operators +, −, * (multiplication), /(division), % (modulus). The binary operator is applied to the numeric value of the operands *term1* and *term2*, and the result is the usual numeric value.

This numeric value is the preferred value, but it can be interpreted as a string value (see **Numeric Constants**). The operators *, /, and % have higher precedence than + and −. All operators are left associative.

Unary Term

In a term of the form

unop term

unop can be unary + or −. The unary operator is applied to the numeric value of *term*, and the result is the usual numeric value which is preferred. However, it can be interpreted as a string value. Unary + and − have higher precedence than *, /, and %.

Incremented Vars

An incremented variable has one of the forms

++ var
−− var
var ++
var −−

The *++ var* has the value *var* + 1 and has the effect of *var* = *var* + 1. Similarly, *−− var* has the value *var* − 1 and has the effect of *var* = *var* − 1. Therefore, *var* ++ has the same value as *var* and has the effect of *var* = *var* + 1. Similarly, *var* −− has the same value as *var* and has the effect of *var* = *var* − 1. The preferred value of an incremented variable is numeric.

Parenthesized Terms

Parentheses are used to group terms in the usual manner.

Expressions

An **awk** expression is one of the following:

term
term term ...
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 −. For example,

```
1+2 3+4
```

has the string (and numeric) value 37.

Assignment Expressions

An assignment expression is one of the forms

var asgnop expression

where *asgnop* is one of the six assignment operators:

=
+=
−=
*=
/=
%=

The preferred value of *var* is the same as that of *expression*.

In an expression of the form

var = expression

the numeric and string values of *var* become those of *expression*.

var op = expression

is equivalent to

var = var op expression

where *op* is one of: +, −, *, /, %. 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
```

Using awk

The remainder of this chapter undertakes to show the syntax rules of **awk** in action. The material is organized under the following topics:

- input and output
- patterns
- actions
- special features

Input and Output

Presenting Your Program for Processing

There are two ways to present your program of pattern/action statements to **awk** for processing:

1. If the program is short (a line or two), it is often easiest to make the program the first argument on the command line:

   awk ' *program* ' [*filename...*]

   where *program* is your **awk** program, and *filename...* is an optional input file(s). Note that there are single quotes around the program name in order for the shell to accept the entire string (program) as the first argument to **awk**. For example, write to the shell

   awk ' /x/ {print} ' file1

   to run the **awk** program /x/ {print} on the input file **file1**. If no input file is specified, **awk** expects input from the standard input, **stdin**. You can also specify that input comes from **stdin** by using the hyphen, –, as one of the files. The pattern-action statement

   awk ' *program* ' **file1** –

   looks for input from **file1** and from **stdin**. It processes first from **file1** and then from **stdin**.

2. Alternately, if your **awk** program is long or is one you want to preserve for re-use in the future, it is convenient to put the program in a separate file, **awk-prog**, for example, and tell **awk** to fetch it from there. This is done by using the **–f** option on the command line, as follows:

   awk –f awkprog *filename...* where *filename...* is an optional list of input

   files that may include **stdin** as is shown above.

These alternative ways of presenting your **awk** program for processing are illustrated by the following:

awk ' BEGIN {print "hello, world"; exit} '

prints

```
hello, world
```

on the standard output when given to the shell.

This **awk** program could be run by putting

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

in a file named **awkprog**, and then the command

awk –f awkprog

given to the shell would have the same effect as the first procedure.

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. The string of characters, thus read, is assigned to the variable **$0**.

Once **awk** has read in a record, it then views the record as being made up of fields. Unless changed by you, a field is a string of characters separated by blanks or tabs.

Sample Input File, countries

For use as an example, we have created the file, **countries**. **countries** 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 1-11: 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 these lines has either four or five fields if blanks and/or tabs separate the fields. This is what **awk** assumes unless told otherwise. In the above 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, it is done through the variable, **$0**. For example, the following action:

```
{print $0}
```

prints the entire record.

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 above example of the file **countries**, in the first record:

$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

... and so forth.

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 may have wanted because the field separator defaults to white space (tabs or blanks). **North America** and **South America** inconveniently contain a blank. Try it again with the following command line:

```
awk -F\t '{print $4, $1, $3}' countries
```

Input: From the Command Line

We have seen above, under "Presenting Your Program for Processing," that 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.

In **awk**, values may be assigned to variables from within an **awk** program. Because you do not declare types of variables, a variable is created simply by referring to it. An example of assigning a value to a variable is:

```
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:

```
awk ' {print x }' x=5 –
```

will print the value **5** on the standard output.

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, named **file1** in the example, the command is

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 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.

There is a separate option, **–F***x*, that is placed directly after the command **awk**. This changes the field separator from white space to the character *x*. For example:

awk –F: –f awkprog file1

changes the field separator, **FS**, to the 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.

Output: Printing

An action may have no pattern; in this case, the action is executed for all lines as in the 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,

awk '{ print $1, $3 }' countries

prints the name of the country and the population:

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

A semicolon at the end of statements is optional. **awk** accepts

```
{print $1}
```

and

```
{print $1;}
```

equally and takes them to mean the same thing. If you want to put two **awk** statements on the same line of an **awk** script, the semicolon is necessary, for example, if you want the number **5** printed:

```
{x=5; print x}
```

Parentheses are also optional with the print statement.

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

is the same as

```
{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 **OFS** is another special variable that can be changed by you. (These special variables are summarized below.) **print** also prints strings directly from your programs, as with the **awk** script

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

As we have already seen, **awk** makes available a number of special variables with useful values, for example, **FS** and **RS**. We introduce two other special variables 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. Thus,

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

prints each record number and the number of fields in each record followed by the record itself. Using this program on the file **countries** yields:

```
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
```

and the program

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

prints

```
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. Use

```
{print ""}
```

to print an empty line.

awk also provides the statement **printf** so that you can format output as desired. **print** uses the default format **%.6g** for each numeric variable printed.

printf "*format*", *expr*, *expr*, ...

formats the expressions in the list according to the specification in the string *format*, and prints them. The *format* statement is almost identical to that of **printf**(3S) in the C library. For example:

```
{ printf "%10s %6d %6d\n", $1, $2, $3 }
```

prints **$1** as a string of 10 characters (right justified). The second and third fields (6-digit numbers) make a neatly columned table.

```
    Russia   8650  262
    Canada   3852  24
     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**, **\t**, **\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 program

```
/x/
```

prints any record that contains the character **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

```
{ x="hello"; y="world"
print x,y
}
```

which prints

```
hello world
```

However, without the comma in the print statement as

```
{ x="hello"; y="world"
print x y
}
```

you get

```
helloworld
```

To get a comma on the output, you can either insert it in the print statement as in this case

```
{ x="hello"; y="world"
print x"," y
}
```

or you can change **OFS** in a **BEGIN** section as in

```
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.

Output: to Different Files

The UNIX 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 our input file **countries**, we 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"
}
```

Flow of control statements is discussed later.

In general, you may direct output into a file after a **print** or a **printf** statement by using a statement of the form

print > *"filename"*

where *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. Notice that there is an upper limit to the number of files that are written in this way. At present it is ten.

Output: to Pipes

It is also possible to direct printing into a pipe instead of a file. For example:

```
{
if ($2 == "XX") print | "mailx mary"
}
```

where **mary** is a person's login name. Any record with the second field equal to **XX** is sent to the user, **mary**, as mail. **awk** waits until the entire program is run before it executes the command that was piped to; in this case, the **mailx**(1) command. For example:

```
{
print $1 | "sort"
}
```

takes the first field of each input record, sorts these fields, and then prints them.

Another example of using a pipe for output is the following idiom, which guarantees that its output 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.

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

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

```
BEGIN {print "Country", "Area", "Population", "Continent"}
      {print}
```

which produces

```
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  18Africa
```

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** is the last if it is used.

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 tiny **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, type

```
$4 == "Asia" {print $1}
```

which produces

```
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. Thus,

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

selects lines that begin with **S**, **T**, **U**, and greater, which in this case are

```
USA      3615     219      North America
Sudan    968      19       Africa
```

In the absence of other information, fields are treated as strings, so the program

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

compares the first and fourth fields as strings of characters and prints the single line

```
Australia        2968       14 Australia
```

Regular Expressions

awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are 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, we could print all lines that begin with **A** with

```
/^A/
```

or all lines that begin with **A**, **B**, or **C** with

```
/^[ABC]/
```

or all lines that end with **ia** with

```
/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, + for one or more occurrences, and ? for zero or one occurrences. For example:

```
/x|y/  {print}
```

prints all records that contain either an **x** or a **y**.

```
/ax+b/        {print}
```

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}
```

prints all records that contain an **a** followed by zero or one **x** followed by a **b**. For example: ab, axb, yaxbPPP, CabD.

The two characters, . and *, have the same meaning as they have in **ed**(1) namely, . can stand for any character and * means zero or more occurrences of the character preceding it. For example:

/a.b/

matches any record that contains an **a** followed by any character followed by 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, axxb.

```
/ax*c/
```

matches a record that contains an **a** followed by zero or more **x**'s followed by a **c**. For example, it matches

```
ac
axc
pqraxxxxxxxxxc901
```

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 pattern

```
/\/*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, with the input file **countries** as before, the program

```
$1 ~ /ia$/        {print $1}
```

prints all countries whose name ends in **ia**:

```
Russia
Australia
India
Algeria
```

which is indeed different from lines that end in **ia**.

Combinations of Patterns

A pattern can be made up of similar patterns combined with the operators || (OR), && (AND), ! (NOT), and parentheses. For example:

```
$2 >= 3000 && $3 >= 100
```

selects lines where both area and population are large. For example:

```
Russia   8650  262  Asia
China    3692  866  Asia
USA      3615  219  North America
Brazil   3286  116  South America
```

while

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

selects lines with **Asia** or **Africa** as the fourth field. An alternate way to write this last expression is with a regular expression:

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

which says to select records where the 4th field matches **Africa** or begins with **Asia**.

&& and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma as in

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

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

prints all lines between the one containing **Canada** and the line containing **Brazil**. For example:

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

while

```
NR == 2, NR == 5 { ... }
```

does the action for lines 2 through 5 of the input. Different types of patterns may be mixed as in

```
/Canada/, $4 == "Africa"
```

which prints all lines from the first line containing **Canada** up to and including the next record whose fourth field is **Africa**.

The foregoing 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 the section "Flow of Control."

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.

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 program

```
{printf "%10s %6.1f\n", $1, (1000000 * $3) / ($2 * 1000)}
```

and the output

```
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, we could write

```
/Asia/   { pop += $3; ++n }
END      {print "total population of", n, "Asian countries is", pop }
```

which produces

```
total population of 3 Asian countries is 1765.
```

The operators, ++, —, −=, /=, * =, +=, and %= are available in **awk** as they are in C. The same is true of the ++ operator; it adds one to the value of a variable. The increment operators ++ and — (as in C) are used as prefix or as postfix operators. These operators are also used in expressions.

Initialization of Variables

In the previous example, we did not initialize **pop** nor **n**; yet everything worked properly. This is because (by default) variables are initialized to the 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.

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

which produces

```
China 866
```

Field Variables

Fields in **awk** share essentially all of the properties of variables. They are used in arithmetic and string operations, may be initialized to the null string, or have other values assigned to them. Thus, divide the second field by 1000 to convert the area to millions of square miles by

```
{ $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.

String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{ x = "hello"
  x = x ", world"
  print x
}
```

which prints the usual

```
hello, world
```

With input from the file **countries**, the following program:

```
/A/          { s = s " " $1 }
END          { print s }
```

prints

```
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

Special Variables

Some variables in **awk** have special meanings. These are detailed here and the complete list given.

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, by default is **%.6g**
FILENAME	The name of the input file currently being read. This is useful because **awk** commands are typically of the form

awk –f *program* **file1 file2 file3 ...**

Type

Variables (and fields) take on numeric or string values according to context. For example, in

```
pop += $3
```

pop is presumably a number, while in

```
country = $1
```

country is a string. In

```
maxpop < $3
```

the type of **maxpop** depends on the data found in **$3**. It is determined when the program is run.

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

```
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 by a subterfuge such as

```
expr + 0
```

and to string by

```
expr ""
```

This last expression is **string** concatenated with the null string.

Arrays

As well as ordinary variables, **awk** provides 1-dimensional arrays. Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value including non-numeric strings. As an example of a conventional numeric subscript, the statement

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

assigns the current input line to the **NR**th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following **awk** program:

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

The first line of this program records each input line into the array **x**. In particular, the following program

```
{ x[NR] = $1}
```

(when run on the file **countries**) produces an array of elements with

```
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 rather like the associative memory of Snobol tables. For example, we can write

```
/Asia/{pop["Asia"] += $3}
/Africa/{pop[Africa] += $3}
END     {print "Asia=" pop["Asia"], "Africa="pop["Africa"] }
```

which produces

```
Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus,

```
area[$1] = $2
```

uses the first field of a line (as a string) to index the array **area**.

Special Features

In this final section we describe the use of some special **awk** features.

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*) will return the length of *x* as a string.

With input from the file **countries**, the following **awk** program will print the longest country name:

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

The function **split**

```
split(s, array)
```

assigns the fields of the string **s** to successive elements of the array, **array**.

For example;

```
split("Now is the time", w)
```

assigns the value **Now** to **w**[1], **is** to **w**[2], **the** to **w**[3], and **time** to **w**[4]. All other elements of the array **w**[ ], if any, are set to the 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.

```
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. Also provided by **awk** are the math functions

sqrt
log
exp
int

They provide the square root function, the base **e** logarithm function, exponential and integral part functions. 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**. (See the *SysV Programmer's Reference*.)

The function **substr**

```
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, we could abbreviate the country names in the file **countries** by

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

which produces

```
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**

```
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**. For example:

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

sets **x** to the string produced by formatting the values of **$1** and **$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.

Flow of Control

awk provides the basic flow of control statements within actions

- **if-else**
- **while**
- **for**

with statement grouping as in C language.

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

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

There is also a **while** statement in **awk**.

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, which is:

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

So

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

is another **awk** program that prints all input fields, one per line.

There is an alternate form of the **for** statement that is useful for accessing the elements of an associative array in **awk**.

for (*i in array*) *statement*

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 will ensue 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 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

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

Note that the expression in the condition part of an **if, while**, or, **for** statement can include

- relational operators like <, <=, >, >=, ==, and !=
- regular expressions that are used with the matching operators ~ and !~
- the 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 is not executed (if there is one).

An **exit** that occurs in the main body of the **awk** program causes execution of the main body of the **awk** 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.

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 of this. 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, use the following program, called **list1**:

```
        {use[$1 "" $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
```

If you would like 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 command

awk –f list1 prog.usage | awk –f format1

gives the output

```
Brown:
        eqn     1
        spell   9
Jones:
        nroff   4
        spell   5
Smith:
        draw    9
        nroff   1
```

It is often useful to combine different **awk** scripts and other shell commands such as **sort**(1), as was done in the **list1** script.

Cooperation with the Shell

Normally, an **awk** program is either contained in a file or enclosed within single quotes as in

awk '{print $1}' ...

Since **awk** uses many of the same characters the shell does (such as **$** and the 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, we want a program called **field** such that

field n

runs the **awk** program

awk '{print $n}'

How does the value of *n* get into the **awk** program?

There are several ways to do this. 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 way to do this job 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 \; the **$1** is again replaced by the number when **field** is invoked.

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 2-dimensional array.

Chapter 2
nawk

Chapter 2: nawk

Introduction to **nawk** 2-1

Basic awk 2-1
Program Structure 2-2
Usage 2-3
Fields 2-3
Printing 2-4
Formatted Printing 2-5
Simple Patterns 2-6
Simple Actions 2-7
Built-In Variables 2-7
User-Defined Variables 2-8
Functions 2-8
A Handful of Useful One-Liners 2-8
Errors Messages 2-9

Patterns 2-11
BEGIN and END 2-11
Relational Expressions 2-12
Regular Expressions 2-13
Combinations of Patterns 2-16
Pattern Ranges 2-17

Actions 2-18
Built-In Variables 2-18
Arithmetic 2-19
Strings and String Functions 2-21
Number or String? 2-25
Arrays 2-29
User-Defined Functions 2-31

Some Lexical Conventions 2-32

Output 2-33
The print Statement 2-33
Output Separators 2-33
The printf Statement 2-34
Output into Files 2-35
Output into Pipes 2-36

Input 2-37
Files and Pipes 2-37
Input Separators 2-37
Multi-line Records 2-38
The getline Function 2-38
Command-Line Arguments 2-41

Using awk with Other Commands and the Shell 2-42
The system Function 2-42

Example Applications 2-44
Additional Examples 2-46
Word Frequencies 2-46
Accumulation 2-46
Random Choice 2-47
Shell Facility 2-47
Form-Letter Generation 2-48

awk Summary 2-49

Introduction to nawk

NOTE: This chapter describes the new version of **awk** released in UNIX System V Release 3.1 and described in **nawk**(1). An earlier version is described in **awk**(1). The new version will become the default in the next major UNIX system release. Until then, you should read **nawk** for **awk** in this chapter.

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. **awk** is a programming language that makes it easy to handle these and many other tasks of information retrieval and data processing. The name **awk** is an acronym constructed from the initials of its developers; it denotes the language and also the UNIX system command you use to run an **awk** program.

awk 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 **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping.

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

You should be familiar with the UNIX system and shell programming to use this chapter. Although you don't need other programming experience, some knowledge of the C programming language is beneficial, because many constructs found in **awk** are also found in C.

Basic awk

This section provides enough information for you to write and run some of your own programs.

Each topic presented is discussed in more detail in later sections.

Program Structure

The basic operation of **awk**(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 **awk** 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. awk Program Structure and Example

The example in the figure is a typical **awk** 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, **awk** 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. The 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.

Usage

There are two ways to run an **awk** program. First, you can type the command line

awk '*pattern-action statements*' optional list of input files

to execute the pattern-action statements on the set of named input files. For example, you could say

```
awk '{ print $1, $2 }' file1 file2
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk**(1) reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (–) as one of the input files. For example,

```
awk '{ print $3, $4 }' file1 -
```

says to read input first from file1 and then from the standard input.

The arrangement above is convenient when the **awk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the -f option to fetch it:

awk -f *program file* optional list of input files

For example, the following command line says to fetch and execute myprogram on input from the file file1:

```
awk -f myprogram file1
```

Fields

awk normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. **awk** then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the file countries, which contains information about the ten largest countries in the world. 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.

This file is typical of the kind of data **awk** 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. Fields are normally separated by sequences of blanks and/or tabs, so that the first record of countries would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default; fields separated by blanks and/or tabs. The first field within a file is called $1, the second $2, and so forth. The entire record is called $0.

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

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 **awk** program consisting of a single print statement

```
{ print }
```

so the command line

```
awk '{ print }' countries
```

prints each line of countries, copying the file to the standard output. The print statement can also be used to print parts of a record; for instance, the program

{ print $1, $3 }

prints the first and third fields of each record. Thus

awk '{ print $1, $3 }' countries

produces as output the sequence of lines:

```
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 which chapter, we show only **awk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the -f flag, as discussed in "**awk** Command Usage." In an example, if no input is mentioned, the input is assumed to be the file countries.

Formatted Printing

For more carefully formatted output, **awk** provides a C-like printf statement

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

which prints the expri's according to the specification in the string format. For example, the **awk** 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 an 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 the \n in the format specification. "The printf Statement" in this chapter contains a full description of printf.

Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. For example, the operator "==" tests for equality. To print the lines for which the fourth field equals the string "Asia", we can use the program consisting of the single pattern

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

With the file countries as input, this program yields

```
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 we want to print only countries with a population greater than 100 million. The program

```
$3 > 100
```

is all that is needed. (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.

Second, you could use patterns called regular expressions 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

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

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, BEGIN and END, that 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
```


Simple Actions

We have already seen the simplest action of an **awk** 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, **awk**(1) counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. So the program

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

prints the number of each line and how many fields it has, while

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

prints each record preceded by its record number.

User-Defined Variables

Besides providing built-in variables like NF and NR, **awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing 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 }
```

NOTE: **awk** 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

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

A Handful of Useful One-Liners

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you may find useful and instructive. They are not explained here, but any new constructs do appear 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:	`{ nc = nc + length($ 0) }`
	`END { print nc + NR }`
	(Adding NR includes in the total number of newlines.)

Print the total number of lines that contain the string Asia:	`/Asia/ { nlines++ }`
	`END { print nlines }`
	(The statement nlines++ has the same effect as nlines = nlines+1.)

Error Messages

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

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

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

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.

BEGIN and END

BEGIN and END are two special patterns that give you a way to control initialization and wrap-up in an **awk** 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 **awk** commands 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 **awk** 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, usually the only sensible 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

```
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
```

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. **awk** has six relational operators, and two regular expression matching operators, ~ (tilde), and !~, which are discussed in the next section, for making comparisons. Figure 2-3 shows these operators and their meanings.

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

Figure 2-3. awk Comparison Operators

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 **awk** can tell what is intended. The section "Number or String?" contains

more information about this.) 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, **awk** 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.

Regular Expressions

awk 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, like

```
/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 program

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

prints the first field of all lines in which the fourth field matches Asia, while the program

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

prints the first field of all lines in which the fourth field does not match Asia.

In regular expressions, the symbols

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

are the metacharacters with special meanings like the metacharacters in the UNIX shell. For example, the metacharacters ^ and $ match the beginning and end, respectively, of a string, and the metacharacter . ("dot") matches any single character. Thus,

```
/^.$/
```

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 [is a ^, this complements the class so it matches any character not in the set: /[^a-zA-Z]/ matches any non-letter. The program

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

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 of this nature are often used for data validation.

Parentheses () are used for grouping and the symbol | is used for alternatives. The program

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

matches lines containing any one of the four substrings apple pie, apple tart, cherry pie, or cherry tart.

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, the program

```
/b\$/
```

prints all lines containing b followed by a dollar sign.

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

\b	backslash
\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 the program

```
/\t/
```

awk interprets any string or variable on the right side of a ~ or !~ as a regular expression. For example, we could have written the program

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

as

```
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 we want to match strings containing b followed by a dollar sign. The regular expression for this pattern is b\$. If we want to create a string to represent this regular expression, we 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 is given in Figure 2-4. The unary operators, *, +, and ? have the highest precedence, then concatenation, and then alternation |. All operators are left associative. r stands for any regular

expression.

Expression	Matches
c	any non-metacharacter c
\c	character c literally
^	beginning of string
$	end of string
[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
r1r2	r1 then r2 (concatenation)
r1|r2	r1 or r2 (alternation)

Figure 2-4. awk Regular Expressions

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (or), && (and), and ! (not). For example, suppose we 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 program

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

selects lines with Asia or Africa as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator |:

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

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

Pattern Ranges

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

pat1, pat2 {...}

In this case, the action is performed for each line between an occurrence of pat1, and the next occurrence of pat2 (inclusive). 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 program

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

prints the first five records of each input file with the name of the current input file prepended.

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.

Built-in Variables

Figure 2-5 lists the built-in variables that **awk** maintains. Some of these we have already met; others are used in this and later sections.

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	"\034"

Figure 2-5. awk Built-In Variables

Arithmetic

Actions can use conventional artithmetic expressions to compute numeric values. As a simple example, suppose we 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 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 **awk** recognizes and produces scientific (exponential) notation: 1e6, 1E6, 10e5, and 1000000 are numerically equal.

awk has assignment statements like those found in the C programming 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, we could write

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

Applied to countries, this program produces

```
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, yet everything works properly because **awk** 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 programming language:

```
pop += $3
```

has the same effect as

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

but 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 ^=. Their meanings are similar:

v "*operation*"= **e**

has the same effect as

v = **v** "*operation*" e

The increment operators are ++ and --. 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 --.

Assigment and increments and decrement operators may all be used in arithmetic expressions.

We use default initialization 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.

awk provides the built-in arithmetic functions shown in Figure 2-6.

Function	Value Returned
atan2(y,x)	arctangent of y/x in the range -pi to pi
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()

Figure 2-6. awk Built-In Arithmetic Functions

x and y are arbitrary expressions. The function rand() returns a pseudo-random 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.

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 programming language escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program

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

prints each record preceded by its record number and a colon, with no blanks. 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.

awk provides the built-in string functions shown in Figure 2-7. 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.

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*)	subsitute 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

Figure 2-7. awk Built-In String Functions

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 is made as long as possible.) It returns the number of substitutions made. The function gsub(r,s) is a synonym for gsub(r,s,$0). For example, the program

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

transcribes its input, replacing occurrences of USA by United States. 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. For example,

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

returns 2.

The length function returns the number of characters in its argument string; thus

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

prints each record, preceded by its length. ($0 does not include the input record separator.) The program

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

applied to the file countries prints the longest country name: Australia.

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 Sudan         968      19  Africa
6 2 Algeria        920      18  Africa
```

NOTE: match() matches the left-most longest matching string. For example, with the record

```
AsiaaaAsiaaaaan
```

as input, the program

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

matches the first string of a's and sets RSTART to 4 and RLENGTH to 3.

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. "The printf Statement" in this chapter contains a complete specification of

the format conventions. The statement

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

assigns to x the string produced by formatting the values of $1 and $2 as a 10-character string and a decimal number in a field of width at least six; x may be used in any subsequent computation.

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, we could abbreviate the country names in countries to their first three characters by invoking the program

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

on this file to produce

```
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 **awk** to recompute $0 and, therefore, the fields are separated by blanks (the default value of OFS), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file countries,

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

prints

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

by building s up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables $1, $2,...,$NF. Field variables share all the properties of other variables -- they may be used in arithmetic or string operations, and they may have values assigned to them. So, 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 }
```

or 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. Notice that 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: the value of $NF-1 is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, $(NF+1), has as its initial value the empty string. A new field can be created, however, 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, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

pop and $3 must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like

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

$1 and $2 must be strings to be concatenated, so they will be coerced if necessary.

In an assignment v = e or v op= e, the type of v becomes the type of e. In an ambiguous context like

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

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ 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" will succeed on any pair of the inputs

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

but fail on the inputs

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

There are two idioms 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, say

```
$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.

Control Flow Statements

awk provides if-else, while, do-while, and for statements, and statement grouping with braces, as in the C programming language.

The if statement syntax is

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, the expression is first evaluated. If it is non-zero and non-null, statement1 is executed; otherwise statement2 is executed. The else part 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 from "Arithmetic Functions" with an if statement results in

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

The while statement is exactly that of the C programming 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, to print all input fields one per line,

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

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

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

It has the same effect as

expression1
while (expression) {
  statement
  expression2
}

so

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

does the same job as the while example above. An alternate version of the for statement is described in the next section.

The do statement has the 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; 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 **awk** 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,

```
exit expr
```

causes the program to return the value of expr as its exit status. If there is no expr, the exit status is zero.

Arrays

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

As an example of a conventional numeric subscript, the statement

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

assigns the current input line to the NRth element of the array x. In fact, it is possible in principle (though perhaps slow) to read the entire input line into an array with the **awk** program

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

The first action merely records each input line in the 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"] }
```

On the file countries, this program generates

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

In this program if we 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 our task is to determine the total area in each continent of the file countries.

Any expression can be used as a subscript in an array reference. Thus

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

uses the string in the fourth field of the current input record to index the array area and in that entry accumulates the value of the second field:

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

Invoked on the file countries, this program produces

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

This program uses a form of the for statement that iterates over all defined subscripts of an array:

for (i in array) *statement*

executes statement with the variable i set in turn to each value of i for which array(i) has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when i or array is altered during the loop.

awk does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable SUBSEP). For example,

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

creates an array which behaves like a two-dimensional array; the subscript is the concatenation of i, SUBSEP, and j.

You can determine whether a particular subscript i occurs in an array arr by testing the condition i in arr, as in

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

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

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

Noth 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 fuction split. The function

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

splits the string s1:s2:s3 into three fields, using the separator :, and stores s1 in a[1], s2 in a[2], and s3 in a[3]. The number of fields found, here three, 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 the delete statement:

```
delete arrayname[subscript]
```


User-Defined Functions

awk provides user-defined functions. A function is defined as

```
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 left 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 (of course, using some input other than the file countries):

```
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.

Some Lexical Conventions

Comments may be placed in **awk** programs: they begin with the character # and end at the end of the line, as in

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

Statements in an **awk** 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.

Output

The print and printf statements are the two primary constructs that generate output. The print statement is used to generate simple output; printf is used for more carefully formatted output. Like the shell, **awk** 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.

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. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line use

```
print ""
```

Output Separators

The output field separator and record separator are held in the built-in variables OFS and ORS. Initially, 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.

The printf Statement

awk's printf statement is the same as that in C except that the * format specifier is not supported. The printf statement has the general form

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

where 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, as in Figure 2-8. Each specification begins with a %, ends with a letter that determines the conversion, and may include

-	left-justify expression in its field
width	pad field to this width as needed; fields that begin with a leading 0 are padded with zeros
.prec	maximum string width or digits to right of decimal point

Character	Prints Expression as
c	single character
d	decimal number
e	[-]d.dddddd E[+-]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

Figure 2-8. awk printf Conversion Characters

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

```
printf "%d", 99/2                    49
printf "%e", 99/2                    4.950000e+01
```

```
printf "%f", 99/2                 49.50000e
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.

Output into Files

It is possible to print output into files instead of to the 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 filenames have to be quoted; without quotes, bigpop and smallpop are merely uninitialized variables. If the output filenames 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 **awk** 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.

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 statement

print | "*command-line*"

causes the output of print to be piped into the command-line.

Although we have shown them here as literal strings enclosed in quotes, the command-line and file names can come from variables and the return values from functions, for instance.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The **awk** 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

```
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 (that is, 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.

Input

The most common way to give input to an **awk** program is to name on the command line the file(s) that contains the input. This is the method we've been using in this chapter. However, there are several other methods we could use, each of which this section describes.

Files and Pipes

You can provide input to an **awk** program by putting the input data into a file, say awkdata, and then executing

awk '*program*' awkdata

awk reads its standard input if no file names are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **awk**. For example, egrep(1) selects input lines containing a specified regular expression, but it can do so faster than **awk** since this is the only thing it does. We could, therefore, invoke the pipe

```
egrep 'Asia' countries | awk '...'
```

egrep quickly finds the lines containing Asia and passes them on to the **awk** program for subsequent processing.

Input Separators

With the default setting of the field separator FS, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1  field2
  field1
field1
```

When the field separator is a 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. For example,

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

sets it to an optional comma followed by any number of blanks and tabs. FS can also be set on the command line with the -F argument:

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

behaves the same as the previous example. Regular expressions used as field separators match the left-most longest occurrences (as in sub()), but do not match null strings.

Multi-line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed, though only in a limited way. If the built-in record separator variable RS is set to the empty string, as in

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

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multi-line records is to use

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

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. "The getline Function" and "Cooperation with the Shell" in this chapter show other examples of processing multi-line records.

The getline Function

awk's facility for automatically breaking its input into records that are more than one line long 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).

To illustrate, suppose we have input data consisting of multi-line records, each of which

begins with a line beginning with START and ends with a line beginning with STOP. The following **awk** program processes these multi-line 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 f array:

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

Notice that this code uses the fact that && evaluates its operands left to right and stops as soon as one is true.

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 statement

```
getline x
```

reads the next record into the variable x. No splitting is done; NF is not set. The statement

```
getline <"file"
```

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. The statement

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

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

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

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

This is because the < 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, the statement

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

executes who and pipes its output into getline. 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, the statement

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

pipes the output of date into the variable d, thus setting d to the current date. Figure 2-9 summarizes the 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

Figure 2-9. getline Function

Command-Line Arguments

The command-line arguments are available to an **awk** 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 **awk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments). The following command line contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
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, **awk** treats the next non-null element of ARGV (up to the current value of ARGC-1) as the name of the next input file.

There is one exception to the rule that an argument is a file name: if it is of the form

```
var=value
```

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

Using awk with Other Commands and the Shell

awk gains its greatest power when it is used in conjunction with other programs. Here we describe some of the ways in which **awk** programs cooperate with other commands.

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 program

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

calls the command cat 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.

Cooperation with the Shell

In all the examples thus far, the **awk** program was in a file and fetched from there using the -f flag, or it appeared on the command line enclosed in single quotes, as in

awk '{ print $1 }' ...

Since **awk** uses many of the same characters as the shell does, such as $ and ", surrounding the **awk** program with single quotes ensures that the shell will pass the entire program unchanged to the **awk** interpreter.

Now, consider writing a command addr that will search a file addresslist for name, address and telephone information. Suppose that addresslist contains names and addresses in which a typical entry is a multi-line record such as

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

Records are separated by a single blank line.

We want to search the address list by issuing commands like

```
addr Emlin
```

This is easily done by a program of the form

```
awk '
BEGIN    { RS = "" }
/Emlin/
```

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

```
awk '
BEGIN    { RS = "" }
/'$1'/
```

The quotes are critical here: the awk 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. On a UNIX system, addr can be made executable by changing its mode with the following command: chmod +x addr.

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

```
awk "
BEGIN    { RS =
/$1/
" addresslist
```

Here we must protect the quotes defining RS with backslashes, so that the shell passes them on to awk, 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 awk program that explicitly reads through the address list with getline:

```
awk '
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.

Example Applications

awk has been used in surprising ways. We have seen **awk** 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 **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. In this section, we will present a few more examples to illustrate some additional **awk** programs.

Generating Reports

awk is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file countries in which we 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, we 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 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 : as its field separator. The +0 -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 as 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
```

This output is in the right order but the wrong format. To transform the output to the desired form we run it through a second **awk** program format:

```
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

```
awk -f triples countries | awk -f format
```

gives us our desired report. As this example suggests, complex data transformation and formatting can often be reduced to a few simple awks and sorts.

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

Word Frequencies

Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab 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 we have two files, deposits and withdrawals, of records containing a name field and an amount field. For each name we 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:

```
awk '
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 will be 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 ith 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 **awk** program simulates (crudely) the history facility of the UNIX system shell. A line containing only = re-executes the last command executed. A line beginning with = cmd re-executes 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:

```
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.
```

and replacement text of this form:

```
field 1|field 2|field 3 one|two|three a|b|c
```

The BEGIN 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
        }
}
```

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.

awk Summary

Command Line

awk *program filenames*

awk –f *program-file filenames*

NOTE:

awk, can be used with the –F[s|t] option. –**Fs** sets field separator to string **s**; –**Ft** sets separator to tab

Patterns

BEGIN
END
/regular expression/
relational expression
pattern && pattern
pattern || pattern
(pattern)
!pattern
pattern, pattern

Control Flow Statements

if (expr) statement [else statement]
if (subscript in array) statement [else statement]
while (expr) statement
for (expr; expr; expr) statement
for (var in array) statement
do statement while (expr)
break
continue
next
exit [expr]
return [expr]

Input-Output

close(filename)	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.

Functions

func name(parameter list) { statement }
function name(parameter list) { statement }
function-name(expr, expr, ...)

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

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

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

Regular expressions (Increasing Precedence)

c	matches non-metacharacter c
\c	matches literal character c
^	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
(r)	grouping: matches r

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"

Limits

Any particular implementation of **awk** 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, e.g., 1e-38..1e+38

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 assignment

```
var = expr
```

its type is set to that of the expression. (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

```
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 string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(that is, concatenation with a null string).

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; for example

```
$1++
```

clearly implies that $1 is to be numeric, and

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

implies that $1 and $2 are both to be strings. Coercion is done as needed.

In the contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. 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. Non-existent fields (i.e., fields past NF) are treated this way, too.

As it is for fields, so it is for array elements created by split().

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus if arr[i] does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" so the if is satisfied. The special construction

```
if (i in arr) ...
```

determines if arr[i] exists without the side effect of creating it if it does not.

Chapter 3
lex

Chapter 3: lex

An Overview of **lex** Programming 3-1

Writing **lex** Programs 3-3
- The Fundamentals of **lex** Rules 3-3
  - Specifications 3-3
  - Actions 3-6
- Advanced **lex** Usage 3-7
  - Some Special Features 3-8
  - Definitions 3-12
  - Subroutines 3-14
- Using **lex** with **yacc** 3-15

Running **lex** under the UNIX System 3-18

An Overview of lex Programming

lex is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are 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 compiler writing case, 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 write programs in a standard language like C to handle them, too. In fact, what **lex** does is produce such C programs. (**lex** is therefore called a program generator.) What **lex** offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that 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, see the diagram in Figure 3-1. We begin with the **lex** source (often called the **lex** specification) that you, the programmer, write to solve the problem at hand. This **lex** source 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 that, in turn, must be compiled by a host language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic—user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

lex can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see

- how to write lex source to do some of these tasks
- how to translate lex source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that lex provides.

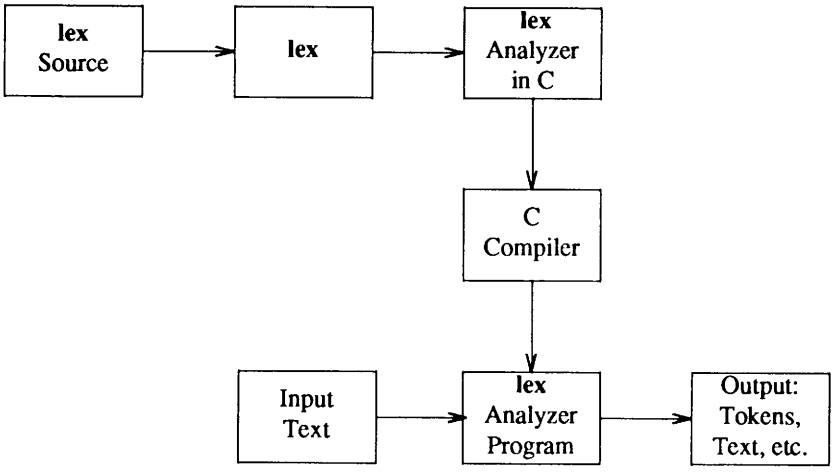

Figure 3-1: Creation and Use of a Lexical Analyzer with lex

Writing lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter **%%**. If a subroutines 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.

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 may 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 is just the beginning rules delimiter, **%%**. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in 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. For example,

```
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, you could specify the rule

```
orange;
```

Because you did not specify an action on the right (before the semi-colon), **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 **orange** above, most of the expressions that we want to search 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 may, in fact, be infinite. Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The + operator, for instance, means one or more occurrences of the preceding expression, the **?** means 0 or 1 occurrence(s) of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and * means 0 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. We will see an example in a moment.) So **m+** is a regular expression matching any string of **m**s such as each of the following:

```
mmm
m
mmmmm
mm
```

and **7*** is a regular expression matching any string of zero or more 7s:

```
77
77777

777
```

The string of blanks on the third line matches simply because it has no 7s 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. Somewhat more interestingly,

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether upper- or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[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 a *, 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
```

because **not_idenTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them as characters to look for in our search pattern. The last example, for instance, will not recognize text with an * in it. **lex** solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a \ is taken literally, that is, as part of the text to be searched for.

To use the backslash method to recognize, say, an * followed by any number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a \ itself, we need two backslashes: \\

Actions

Once **lex** recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. 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. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression Amelia Earhart and to note such recognition, the rule

```
"Amelia Earhart"    printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. **lex** uses the standard escape sequences from C like **\n** for end-of-line. To count lines we might have

```
\n    lineno++;
```

where **lineno**, like other C variables, is declared in the definitions section that we discuss later.

lex stores every character string that it recognizes in a character array called **yytext[]**. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you choose to) write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum, here) and print out each one as soon as it is found, your **lex** code might be

```
+?[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 **?** indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign, –, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that lets you process input text riddled with quite complicated patterns. These 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. Before considering these features, you may want to affirm your understanding thus far by examining an example drawing together several of the points already covered.

```
%%
-[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: ", 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 several 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 * 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 may use this variable to refer to any specific character just placed in the array **yytext**[]. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) in a just recognized integer, you might write

```
[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. *Prima facie*, any reserved word, for instance, could match two rules. 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.

lex follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above 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 it matches the longest character string possible and executes 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've in fact 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 statement

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index **k** until we read the first comma. Until then, we might have the assignment statement

```
DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the forward-looking slash, / (not 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

```
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 **$** 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]+$      ;
```

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, ^, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ]        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**(), **unput**(c), and **output**(c), respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input**(), thus:

```
\"          while (input() != '"');
```

Upon finding the first double quotation mark, the generated **a.out** will simply continue reading all subsequent characters so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you may 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 your subroutine section. Your new routines will then 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. These include **yymore**(), **yyless**(n), and **REJECT**. 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 interspersed with one 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**'s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is

```
B[^B]* { if (flag = 0)
                save = yyleng;
                flag = 1;
                yymore();
           else {
                importantno = save + yyleng;
                flag = 0; }
                }
```

where **flag**, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. 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 to the *n*th character in the original **yytext[]**. Suppose you are again in the code deciphering business and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say upper- or lowercase **Z**. The code you want might be

```
[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, the following will do:

```
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. Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be 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 will be 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. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file to be included in every file that needs them. One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#define**s for token names. Like the declarations, **#include** statements should come between %{ and %}, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the %} that ends your **#include**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within 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               [ ]
%%
-{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, newly added to the example and somewhat more complex than the others, is used in the WRITER'S WORKBENCH Software, an AT&T software product for promoting good writing. (See the *UNIX System WRITER'S WORKBENCH Software Release 3.0 User's Guide* for information on this product.) The rule ensures that a period always precedes a quotation mark at the end of a sentence. It would change example". to example."

Subroutines

You may 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 this 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 / if the comment ends unusually with a **/ . In this case, eventually having read an *, the analyzer finds that the next character is not the terminal / and must read some more. Second, the expression **yytext[yyleng–1]** picks out that last character read. 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, the **a.out** will read 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 may be exploited by many different programs would probably do best to 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.

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **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 this section because it covers 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 very name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want 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 perhaps slightly reminiscent of Ada:

```
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, the tokens returned, and the values assigned to **tokval**, are indeed integers. Good programming style dictates that we use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth 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. For example,

```
#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 statement

```
#include y.tab.h
```

into the definitions section of your **lex** source. 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).

Running lex under the UNIX System

As you review the following few steps, you might recall Figure 3-1 at the start of the chapter. To produce the lexical analyzer in C, run

lex lex.l

where **lex.l** is the file containing your **lex** specification. The name **lex.l** is conventionally the favorite, but you may use whatever name you want. The output file that **lex** produces is automatically called **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 spread across 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 will overwrite the previous one. Once you have all the generated **.c** files, you can compile all of them, of course, in one command line.

With the executable **a.out** produced, you are ready to analyze any desired input text. Suppose that the text is stored under the filename **textin** (this name is also arbitrary). The lexical analyzer **a.out** by default takes input from your terminal. To have it take the file **textin** as input, simply use redirection, thus:

a.out < textin

By default, output will appear on your terminal, but you can redirect this as well:

a.out < textin > textout

In running **lex** with **yacc**, either may be run first.

yacc –d grammar.y
lex lex.l

spawns 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.) To compile and link the output files produced, run

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 will call the **yacc** parser.

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument. If you care to see the C program, **lex.yy.c**, that **lex** generates on your terminal (the default output device), use the **-t** option.

```
lex -t lex.l
```

The **-v** option prints out for you a small set of statistics describing the so-called finite automata that **lex** produces with the C program **lex.yy.c**. (For a detailed account of finite automata and their importance for **lex**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 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 another entry in the definitions section of your **lex** source, as follows:

```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. (The **-v** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *SysV Programmer's Reference* page on **lex** for a list of all the options available with the **lex** command. In addition, review the paper by Lesk (the originator of **lex**) and Schmidt, "Lex—A Lexical Analyzer Generator," in volume 5 of the *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1986. It is somewhat dated, but offers several interesting examples.

This tutorial has introduced you to **lex** programming. As with any programming language, the way to master it is to write programs and then write some more.

Chapter 4
yacc

Chapter 4: yacc

Introduction 4-1

Basic Specifications 4-4
- Actions 4-6
- Lexical Analysis 4-9

Parser Operation 4-12

Ambiguity and Conflicts 4-17

Precedence 4-22

Error Handling 4-26

The **yacc** Environment 4-30

Hints for Preparing Specifications 4-32
- Input Style 4-32
- Left Recursion 4-32
- Lexical Tie-In 4-33
- Reserved Words 4-35

Advanced Topics 4-36
- Simulating **error** and **accept** in Actions 4-36
- Accessing Values in Enclosing Rules 4-36
- Support for Arbitrary Value Types 4-38
- **yacc** Input Syntax 4-40

Examples 4-43
1. A Simple Example 4-43
2. An Advanced Example 4-46

Introduction

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes:

- 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 code supplied for this rule, (an action) is invoked. Actions are fragments of C language 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. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

where **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 input

```
July  4, 1776
```

might be matched by the rule.

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 rules

```
month_name : 'J' 'a' 'n'  ;
month_name : 'F' 'e' 'b'  ;

           ...

month_name : 'D' 'e' 'c'  ;
```

might be used in the above example. 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 to the above example the rule

```
date  :  month '/' day '/' year  ;
```

allowing

```
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 human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The 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.

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).

A full specification file looks like:

declarations
%%
rules
%%
subroutines

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest legal **yacc** specification is

%%
rules

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where **A** represents a nonterminal symbol, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length and may be made up of letters, dots, underscores, and digits although a digit may not be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes, '. As in the C language, the backslash, \, is an escape character within literals, and all the C language escapes are recognized. Thus:

'\n'	newline
'\r'	return
'\''	single quote (')
'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	*xxx* in octal notation

are understood by **yacc**. For a number of technical reasons, 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, the vertical bar, |, can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus the grammar rules

```
A   :  B  C  D   ;
A   :  E  F   ;
A   :  G   ;
```

can be given to **yacc** as

```
A   :  B  C  D
    |  E  F
    |  G
    ;
```

by using the vertical bar. 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

```
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

```
%token   name1  name2 ...
```

in the declarations section. 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.

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 }. For example:

```
A   :  '('  B  ')'
    {
       hello( 1, "abc" );
    }
```

and

```
XXX   :  YYY  ZZZ
      {
         (void) printf("a message\n");
         flag = 25;
      }
```

are grammar rules with actions.

The dollar sign symbol, **$**, is used to facilitate communication between the actions and the parser, The pseudo-variable **$$** represents the value returned by the complete action. For example, the action

```
{  $$ = 1;  }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables **$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

```
A  :  B  C  D  ;
```

then **$2** has the value returned by **C**, and **$3** the value returned by **D**.

The rule

```
expr  :  '('  expr  ')'  ;
```

provides a common example. One would 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 by

```
expr  :    '('  expr  ')'
      {
          $$ = $2 ;
      }
```

By default, the value of a rule is the value of the first element in it (**$1**). Thus, grammar rules of the form

```
A  :  B  ;
```

frequently need not have an explicit action. In 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 rule below 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

```
$ACT    :   /* empty */
        {
            $$ = 1;
        }
        ;

A       :   B  $ACT  C
        {
            x = $2;
            y = $3;
        }
        ;
```

where **$ACT** is an empty action.

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

```
node( L, n1, n2 )
```

creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node. Then a parse tree can be built by supplying actions such as

```
expr   :   expr  '+'  expr
       {
           $$ = node( '+', $1, $3 );
       }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
%{   int variable = 0;   %}
```

could be placed in the declarations section making **variable** accessible to all of the actions. 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 the section "Advanced Topics."

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 language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
int yylex()
{
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch (c)
        {
           ...
           case '0':
           case '1':
           ...
           case '9':
           yylval = c - '0';
           return (DIGIT);
           ...
        }
        ...
}
```

to return the appropriate token.

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 to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

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 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 crafted by hand.

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 will not be discussed here. The parser itself, though, is relatively simple and understanding its usage will make 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**. A step of the parser 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 in 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. For example, in state 56 there may be an action

```
IF   shift 34
```

which says, 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.

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 action

```
.   reduce 18
```

refers to grammar rule 18, while the action

```
IF   shift 34
```

refers to state 34.

Suppose the rule

```
A   :   x  y  z   ;
```

is being reduced. The **reduce** action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer serve any useful purpose. After popping these states, a state is uncovered, which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of **A**. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a **goto** action. In particular, the 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

```
A   goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the **reduce** action turns back the clock in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the 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 pseudo-

variables **$1**, **$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the 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) will be discussed later.

Consider:

```
%token  DING  DONG  DELL
%%
rhyme   :   sound  place
        ;
sound   :   DING  DONG
        ;
place   :   DELL
        ;
```

as a **yacc** specification.

When **yacc** is invoked with the –v option, a file called **y.output** is produced with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) 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 _ character is used to indicate what has been seen and what is yet to come in each rule. The following input

```
DING  DONG  DELL
```

can be used to track the operations of the parser. 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

```
sound  :  DING  DONG
```

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**),

```
sound   goto 2
```

is obtained. 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, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr   :   expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called left association, the second right association.)

yacc detects such ambiguities when it is attempting to build the parser. Given the input

```
expr - expr - expr
```

consider the problem that confronts the parser. When the parser has read the second *expr*, the input seen

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees

```
expr - expr
```

it could defer the immediate application of the rule and continue reading the input until

```
expr - expr - expr
```

is seen. It could then apply the rule to the rightmost three symbols reducing them to

expr, which results in

```
expr - expr
```

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do one of two legal things, a shift or a reduction. 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. Note that there are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing 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** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat   :   IF  '('  cond  ')'  stat
       |   IF  '('  cond  ')'  stat  ELSE  stat
       ;
```

which is a fragment from a programming language involving an **if-then-else** statement. 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 form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
}
ELSE
      S2
```

or

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
      ELSE
            S2
}
```

where the second interpretation is the one given in most programming languages having this construct; each ELSE is associated with the last preceding un-ELSE'd IF. In this example, consider the situation where the parser has seen

```
IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the ELSE. It can immediately reduce by the simple **if** rule to get

```
IF  (  C1  )  stat
```

and then read the remaining input

```
ELSE  S2
```

and reduce

```
IF  (  C1  )  stat  ELSE  S2
```

by the **if-else** rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right-hand portion of

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
IF  (  C1  )  stat
```

which can be reduced by the simple **if** rule. This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things—there is a **shift-reduce** conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as

```
IF  (  C1  )  IF  (  C2  )  S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE

state 23

   stat  :  IF  (  cond  )  stat_          (18)
   stat  :  IF  (  cond  )  stat_ELSE  stat

   ELSE     shift 45
   .        reduce 18
```

where the first line describes the 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 the underline marks the portion of the grammar rules, which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot, .), is to be 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

```
stat  :  IF  '('  cond  ')'  stat
```

by grammar rule 18.

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.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description

```
%right  '='
%left  '+'  '-'
%left  '*'  '/'

%%

expr   :   expr  '='  expr
       |   expr  '+'  expr
       |   expr  '-'  expr
       |   expr  '*'  expr
       |   expr  '/'  expr
       |   NAME
       ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. 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 rules

```
%left  '+'  '-'
%left  '*'  '/'
%%
expr   :   expr  '+'  expr
       |   expr  '-'  expr
       |   expr  '*'  expr
       |   expr  '/'  expr
       |   '-'  expr      %prec  '*'
       |   NAME
       ;
```

might be used to give unary minus the same precedence as multiplication.

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.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and/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 form

```
stat   :   error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat   :   error  ';'
```

are somewhat easier. 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

```
input   :   error  '\n'
                {
                    (void) printf( "Reenter last line: " );
                }
                input
        {
           $$ = $4;
        }
        ;
```

is one way to do this. 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 statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input   :   error  '\n'
                {
                   yyerrok;
                   (void) printf( "Reenter last line: " );
                }
                input
        {
            $$ = $4;
        }
        ;
```

which is somewhat better.

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 statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine 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

```
stat    :   error
        {
            resynch();
            yyerrok  ;
            yyclearin;
        }
        ;
```

could perform this.

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.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language 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 (see "Lexical Analysis"), to obtain input tokens. 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 a **–ly** argument to the **cc**(1) command or to the loader. The source codes

```
main()
{
     return (yyparse());
}
```

and

```
# include <stdio.h>

yyerror(s)
        char *s;
{
        (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs. The argument to **yerror**() is a string containing an error message, usually the string **syntax error**. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the 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 (to read arguments, etc.), the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using **sdb**.

Hints for Preparing Specifications

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

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.

Example 1 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.

Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules. Rules of the form

```
name    :   name  rest_of_rule  ;
```

match this algorithm. These rules such as

```
list    :   item
        |   list  ','  item
        ;
```

and

```
seq     :   item
        |   seq  item
        ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :   item
        |   item  seq
        ;
```

the parser is a bit bigger; and the items are seen and reduced from right to left. 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 sequence specification as

```
seq     :   /* empty */
        |   seq  item
        ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
   int dflag;
%}
  ...  other declarations ...

%%

prog   :   decls  stats
       ;

decls  :   /* empty */
       {
             dflag = 1;
       }
       |   decls  declaration
       ;

stats  :   /* empty */
       {
             dflag = 0;
       }
       |   stats  statement
       ;

    ...  other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. 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 can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit 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 extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it this instance of **if** is a keyword and that instance is a variable. The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced Topics

This part discusses a number of advanced features of **yacc**.

Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of macros YYACCEPT and YYERROR. 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.

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. Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser 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** will automatically insert the appropriate union name so that no unwanted conversions take place. In addition, type checking commands such as **lint** are far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other subroutines, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes

```
%union
{
   body of union ...
}
```

in the declaration section. 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 construction

```
<name>
```

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  '+'  '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**.

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as **$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The example

```
rule   :   aaa
                {
                   $<intval>$ = 3;
                }
                bbb
       {
           fun( $<intval>2, $<other>0 );
       }
       ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of **$n** or **$$** to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold **int**s.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies, etc. are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have

an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs but never as part of C_IDENTIFIERs.

```
    /* grammar for the input to yacc */

    /* basic 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
        ;
```

continued

```
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
        ;
```

continued

```
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 ';'
        ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the operators

+, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignments.

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how 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 better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */
```

continued

```
%%          /* 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;
```

continued

```
        }
        |   expr '&' expr
        {
            $$ = $1 & $3;
        }
        |   expr '|' expr
        {
            $$ = $1 | $3;
        }
        | '-' expr  %prec UMINUS
        {
            $$ = -$2;
        }
        | LETTER
        {
            $$ = regs[$1];
        }
        | number
        ;

number  :   DIGIT
        {
            $$ = $1; base = ($1==0) ? 8 ; 10;
        }
        |   number DIGIT
        {
            $$ = base * $1 + $2;
        }
        ;

%%              /* 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 */
```

continued

```
        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);
}
```

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. Moreover, it also understands intervals written

(X, Y)

where X is less than or equal to Y. There are 26 interval valued variables A through Z that may also be used. The usage is similar to that in Example 1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using **typedef**. **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. 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** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines.

```
2.5 + (3.5 - 4.)
```

and

```
2.5 + (3.5, 4)
```

Notice that the 2.5 is to be used in an interval value expression in the second example, 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. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator–one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine **atof()** is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
     double lo, hi;
}  INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start lines

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG  /* indices into dreg, vreg arrays */


%token <dval> CONST       /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */
```

continued

```
  /* 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;
        }
```

continued

```
        |  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 )
```

continued

```
        {
                (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, $1.hi, $3 );
}
| dexp '*' vexp
{
        $$ = vmul( $1, $1, $3 );
}
| vexp '/' vexp
```

continued

```
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 );
        }
        | dexp '/' vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1, $1, $3 );
        }
        | '-' vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;
                $$.lo = -$2.hi;
        }
        | '(' vexp ')'
        }
                $$ = $2;
        }
        ;

%%                          /* beginning of subroutines section */

# define BSZ 50   /* buffer size for floating point number */

         /* lexical analysis */

int yylex( )
{
        register c;

                             /* skip over blanks */
        while ((c = getchar()) == ' ')
                ;
        if (isupper(c))
        {
            yylval.ival = c - 'A';
            return (VREG);
        }
        if (islower(c))
```

continued

```
{
    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 = ' ';
        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);
}
```

continued

```
        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);
}
```

continued

```
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));
}
```

Chapter 5
File and Record Locking

Chapter 5: File and Record Locking

Introduction 5-1

Terminology 5-2

File Protection 5-4
- Opening a File for Record Locking 5-4
- Setting a File Lock 5-6
- Setting and Removing Record Locks 5-9
- Getting Lock Information 5-13
- Deadlock Handling 5-16

Selecting Advisory or Mandatory Locking 5-17
- Caveat Emptor—Mandatory Locking 5-18
- Record Locking and Future Releases of the UNIX System 5-19

Introduction

Mandatory and advisory file and record locking both are available on current releases of the UNIX system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record

A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (i.e. **creat**(2), **open**(2), **read**(2), and **write**(2)). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat**(2), **open**(2), **read**(2), and **write**(2) system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the data base accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail**(1) command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. For our example we will open our file for both read and write access:

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


int fd;      /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
        extern void exit(), perror();

        /* get data base file name from command line and open the
         * file for read and write access.
         */
        if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
        }
        filename = argv[1];
        fd = open(filename, O_RDWR);
        if (fd < 0) {
        perror(filename);
        exit(2);
        }
        .
        .
        .
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl**(2) system call, the other using the */usr/group* standards compatible **lockf**(3) library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl**(2) system call is as follows:

```
#include <fcntl.h>
#define MAX_TRY10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK;/* setting a write lock */
lck.l_whence = 0;/* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L;/* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
if (errno == EAGAIN || errno == EACCES) {
/* there might be other errors cases in which
 * you might try again.
 */
if (++try < MAX_TRY) {
(void) sleep(2);
continue;
}
(void) fprintf(stderr,"File busy try again later!\n");
return;
}
perror("fcntl");
exit(2);
}
.
.
.
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because MAX_TRY has been exceeded

To perform the same task using the **lockf**(3) function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
if (errno == EAGAIN || errno == EACCES) {
/* there might be other errors cases in which
 * you might try again.
 */
if (++try < MAX_TRY) {
sleep(2);
continue;
}
(void) fprintf(stderr,"File busy try again later!\n");
return;
}
perror("lockf");
exit(2);
}
.
.
.
```

It should be noted that the **lockf**(3) example appears to be simpler, but the **fcntl**(2) example exhibits additional flexibility. Using the **fcntl**(2) method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf**(3) merely sets write (exclusive) locks; an additional system call (**lseek**(2)) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the inter-record pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```
struct record {
        .
        ./* data portion of record */
        .
        long prev;/* index to previous record in the list */
        long next;/* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are read
 * locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *    Set a write lock on "this".
 *    Return index to "this" record.
 * If any write lock is not obtained:
 *    Restore read locks on "here" and "next".
 *    Remove all other locks.
 *    Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
        struct flock lck;

        lck.l_type = F_WRLCK;/* setting a write lock */
        lck.l_whence = 0;/* offset l_start from beginning of file */
        lck.l_start = here;
        lck.l_len = sizeof(struct record);

        /* promote lock on "here" to write lock */
        if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
        }
        /* lock "this" with write lock */
```

continued

```
        lck.l_start = this;
        if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;
         * demote lock on "here" to read lock.
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
        }
        /* promote lock on "next" to write lock */
        lck.l_start = next;
        if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed;
         * demote lock on "here" to read lock,
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* and remove lock on "this".
         */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1);/* cannot set lock, try again or quit */
        }

        return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the **fcntl** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 *    Set a lock on "this".
 *    Return index to "this" record.
 * If any lock is not obtained:
 *    Remove all other locks.
 *    Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

        /* lock "here" */
        (void) lseek(fd, here, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
        }
        /* lock "this" */
        (void) lseek(fd, this, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);

        }

        /* lock "next" */
        (void) lseek(fd, next, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

        /* Lock on "next" failed.
```

continued

```
         * Clear lock on "here",
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));

        /* and remove lock on "this".
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);/* cannot set lock, try again or quit */

        }

        return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the F_GETLK command is used in the **fcntl** call. If the lock passed to **fcntl** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl**. That is, the lock data passed to **fcntl** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in **l_sysid** should be

ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process' lock, then the **l_type** field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of this file. */
        (void) printf("sysid   pid type    start   length\n");
        lck.l_whence = 0;
        lck.l_start = 0L;
        lck.l_len = 0L;
        do {
        lck.l_type = F_WRLCK;
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK) {
        (void) printf("%5d %5d   %c  %8d %8d\n",
        lck.l_sysid,
        lck.l_pid,
        (lck.l_type == F_WRLCK) ? 'W' : 'R',
        lck.l_start,
        lck.l_len);
        /* if this lock goes to the end of the address
         * space, no need to look further, so break out.
         */
        if (lck.l_len == 0)
        break;
        /* otherwise, look for new lock after the one
         * just found.
         */
        lck.l_start += lck.l_len;
        }
        } while (lck.l_type != F_UNLCK);
```

fcntl with the F_GETLK command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **lockf** to test for a lock on a file follows:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
switch (errno) {
case EACCES:
case EAGAIN:
(void) printf("file is locked by another process\n");
break;
case EBADF:
/* bad argument passed to lockf */
perror("lockf");
break;
default:
(void) printf("lockf: unknown error <%d>\n", errno);
break;
}
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking.

The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf**(3) function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod**(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
        .
        .
        .
        if (stat(filename, &buf) < 0) {
        perror("program");
        exit (2);
        }
        /* get currently set mode */
        mode = buf.st_mode;
        /* remove group execute permission from mode */
        mode &= ~(S_IEXEC>>3);
        /* set 'set group id bit' in mode */
        mode |= S_ISGID;
        if (chmod(filename, mode) < 0) {
        perror("program");
        exit(2);
        }
        .
        .
        .
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod**(1) command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *filename*

The **ls**(1) command was also changed to show this setting when you ask for the long listing format:

ls -l *filename*

causes the following to be printed:

```
-rw---l---   1 abc        other     1048576 Dec  3 11:44 filename
```

Caveat Emptor—Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the UNIX System

Provisions have been made for file and record locking in a UNIX system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the **sleep-when-blocked** feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Chapter 6
Inter-process Communication

Chapter 6: Inter-process Communication

Introduction 6-1

Messages 6-2
- Getting Message Queues 6-6
  - Using **msgget** 6-7
  - Example Program 6-12
- Controlling Message Queues 6-16
  - Using **msgctl** 6-16
  - Example Program 6-18
- Operations for Messages 6-24
  - Using **msgop** 6-24
  - Example Program 6-26

Semaphores 6-37
- Using Semaphores 6-39
- Getting Semaphores 6-42
  - Using **semget** 6-42
  - Example Program 6-46
- Controlling Semaphores 6-50
  - Using **semctl** 6-51
  - Example Program 6-53
- Operations on Semaphores 6-64
  - Using **semop** 6-64
  - Example Program 6-66

Shared Memory 6-72
- Using Shared Memory 6-73
- Getting Shared Memory Segments 6-76
  - Using **shmget** 6-76
  - Example Program 6-80

Controlling Shared Memory 6-84
Using **shmctl** 6-85
Example Program 6-86
Operations for Shared Memory 6-95
Using **shmop** 6-95
Example Program 6-96

Introduction

The UNIX system supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC.

Included in the chapter are several example programs that show the use of the IPC system calls. All of the example programs have been compiled and run on an AT&T 3B2 Computer.

Since there are many ways in the C Programming Language to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that the calls provide.

Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations:

- sending
- receiving

Before a message can be sent or received by a process, a process must have the UNIX operating system generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget**(2) system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl**(2) system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl**() to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. Senders block until there is room in the queue for their message. Receivers block until a message shows up in the queue. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (−1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store (header) information about each message that is being sent or received. Messages are queued until read. There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

All include files discussed in this chapter are located in the **/usr/include** or **/usr/include/sys** directories.

The structure definition for the associated data structure is as follows:

```
struct msqid_ds
{
    struct ipc_perm  msg_perm;   /* operation permission struct */
    struct msg       *msg_first; /* ptr to first message on q */
    struct msg       *msg_last;  /* ptr to last message on q */
    ushort           msg_cbytes; /* current # bytes on q */
    ushort           msg_qnum;   /* # of messages on q */
    ushort           msg_qbytes; /* max # of bytes on q */
    ushort           msg_lspid;  /* pid of last msgsnd */
    ushort           msg_lrpid;  /* pid of last msgrcv */
    time_t           msg_stime;  /* last msgsnd time */
    time_t           msg_rtime;  /* last msgrcv time */
    time_t           msg_ctime;  /* last change time */
};
```

It is located in the **#include <sys/msg.h>** header file. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 6-1.

The definition of the **ipc_perm** data structure is as follows:

```
struct ipc_perm
{
        ushort   uid;     /* owner's user id */
        ushort   gid;     /* owner's group id */
        ushort   cuid;    /* creator's user id */
        ushort   cgid;    /* creator's group id */
        ushort   mode;    /* access modes */
        ushort   seq;     /* slot usage sequence number */
        key_t    key;     /* key */
};
```

Figure 6-1: **ipc_perm** Data Structure

It is located in the **#include <sys/ipc.h>** header file; it is common for all IPC facilities.

The **msgget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **msgflg** argument that it receives:

- to get a new **msqid** and create an associated message queue and data structure for it
- to return an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget**() system call. For the first task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **msqid** is always returned with an associated message queue and data structure created for it. When the **ipcs** command is performed, for security reasons the KEY field for the **msqid** is all zeros.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not desire to have an existing **msqid** returned, a control command (IPC_EXCL) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in the "Using **msgget**" section of this chapter.

When performing the first task, the process which calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the "Controlling Message Queues" section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations [**msgop**()] and message control [**msgctl**()] can be used.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd**() and **msgrcv**(). Refer to the "Operations for Messages" section in this chapter for details of these system calls.

Message control is done by using the **msgctl**(2) system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (**msqid**)
- to change operation permissions for a message queue
- to change the **size** (**msg_qbytes**) of the message queue for a particular **msqid**
- to remove a particular **msqid** from the UNIX operating system along with its associated message queue and data structure

Refer to the "Controlling Message Queues" section in this chapter for details of the **msgctl**() system call.

Getting Message Queues

This section gives a detailed description of using the **msgget**(2) system call along with an example program illustrating its use.

Using msgget

The synopsis found in the **msgget**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int  msgget (key, msgflg)
key_t  key;
int msgflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX operating system.

The following line in the synopsis:

```
int msgget (key, msgflg)
```

informs you that **msgget**() is a function with two formal arguments that returns an integer type value, upon successful completion (**msqid**). The next two lines:

```
key_t  key;
int msgflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function upon successful completion is the message queue identifier (**msqid**) that was discussed earlier.

As declared, the process calling the **msgget**() system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **msgflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **msgflg** argument must be an integer type octal value and it will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **msgflg** argument. They are collectively referred to as "operation permissions." Figure 6-2 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 6-2: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **msg.h** header file which can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). Figure 6-3 contains the names of the constants which apply to the **msgget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 6-3: Control Commands (Flags)

The value for **msgflg** is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	01000	0 000 001 000 000 000
| ORed by User	=	00400	0 000 000 100 000 000
msgflg	=	01400	0 000 001 100 000 000

The **msgflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));

msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget**(2) page in the *SysV Programmer's Reference*, success or failure of this system call depends upon the argument values for **key** and **msgflg**. The system call will attempt to return a new **msqid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a **msqid** associated with it, and (**msgflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);

            or

   msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

The second condition is satisfied if the value for **key** is not already associated with a **msqid** and the bitwise ANDing of **msgflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**msgflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

msgflg	=	x 1 x x x	(x = immaterial)
& IPC_CREAT	=	0 1 0 0 0	
result	=	0 1 0 0 0	(not zero)

Since the result is not zero, the flag is set or "true."

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **msqid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msqid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **msqid** is returned if the system call is successful.

Refer to the **msgget**(2) page in the *SysV Programmer's Reference* for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 6-4) is a menu driven program which allows all possible combinations of using the **msgget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **msgget**(2) entry in the *SysV Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument
- **msqid**—used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

The system call is made next, and the result is stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 55). If **msqid** equals −1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the **msgget**(2) system call follows. It is suggested that the source program file be named **msgget.c** and that the executable file be named **msgget**. When compiling C programs that use floating point operations, the **−f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the message get, msgget(),
3    **system call capabilities.*/

4    #include    <stdio.h>
5    #include    <sys/types.h>
6    #include    <sys/ipc.h>
7    #include    <sys/msg.h>
8    #include    <errno.h>

9    /*Start of main C language program*/
10   main()
11   {
12       key_t key;              /*declare as long integer*/
13       int opperm, flags;
14       int msqid, opperm_flags;
15       /*Enter the desired key*/
16       printf("Enter the desired key in hex = ");
17       scanf("%x", &key);


18       /*Enter the desired octal operation
19         permissions.*/
20       printf("\nEnter the operation\n");
21       printf("permissions in octal = ");
22       scanf("%o", &opperm);
```

Figure 6-4: **msgget**() System Call Example (Sheet 1 of 3)

```
23        /*Set the desired flags.*/
24        printf("\nEnter corresponding number to\n");
25        printf("set the desired flags:\n");
26        printf("No flags                  = 0\n");
27        printf("IPC_CREAT                 = 1\n");
28        printf("IPC_EXCL                  = 2\n");
29        printf("IPC_CREAT and IPC_EXCL    = 3\n");
30        printf("           Flags          = ");

31        /*Get the flag(s) to be set.*/
32        scanf("%d", &flags);

33        /*Check the values.*/
34        printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35            key, opperm, flags);

36        /*Incorporate the control fields (flags) with
37          the operation permissions*/
38        switch (flags)
39        {
40        case 0:    /*No flags are to be set.*/
41            opperm_flags = (opperm | 0);
42            break;
43        case 1:    /*Set the IPC_CREAT flag.*/
44            opperm_flags = (opperm | IPC_CREAT);
45            break;
46        case 2:    /*Set the IPC_EXCL flag.*/
47            opperm_flags = (opperm | IPC_EXCL);
48            break;
49        case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
50            opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51        }
```

Figure 6-4: **msgget()** System Call Example (Sheet 2 of 3)

```
52        /*Call the msgget system call.*/
53        msqid = msgget (key, opperm_flags);

54        /*Perform the following if the call is unsuccessful.*/
55        if(msqid == -1)
56        {
57            printf ("\nThe msgget system call failed!\n");
58            printf ("The error number = %d\n", errno);
59        }

60        /*Return the msqid upon successful completion.*/
61        else
62            printf ("\nThe msqid = %d\n", msqid);
63        exit(0);
64    }
```

Figure 6-4: **msgget**() System Call Example (Sheet 3 of 3)

Controlling Message Queues

This section gives a detailed description of using the **msgctl** system call along with an example program which allows all of its capabilities to be exercised.

Using msgctl

The synopsis found in the **msgctl**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```


The **msgctl**() system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, it returns a −1.

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

IPC_STAT — return the status information contained in the associated data structure for the specified **msqid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area.

IPC_SET — for the specified **msqid**, set the effective user and group identification, operation permissions, and the number of bytes for the message queue.

IPC_RMID — remove the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 6-5) is a menu driven program which allows all possible combinations of using the **msgctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgctl**(2) entry in the *SysV Programmer's Reference*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid	used to store the IPC_SET value for the effective user identification
gid	used to store the IPC_SET value for the effective group identification
mode	used to store the IPC_SET value for the operation permissions
bytes	used to store the IPC_SET value for the number of bytes in the message queue (**msg_qbytes**)
rtrn	used to store the return integer value from the system call
msqid	used to store and pass the message queue identifier to the system call
command	used to store the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member is to be changed for the IPC_SET control command
msqid_ds	used to receive the specified message queue indentifier's data structure when an IPC_STAT control command is performed
***buf**	a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the **msg.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **msqid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier which is stored at the address of the **msqid** variable (lines 19, 20). This is required for every **msgctl** system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the UNIX operating system. Note that the ***buf** pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending upon the success or failure, the program returns the

same messages as for the other control commands.

The example program for the **msgctl()** system call follows. It is suggested that the source program file be named **msgctl.c** and that the executable file be named **msgctl**.

```
/*This is a program to illustrate
**the message control, msgctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    int uid, gid, mode, bytes;
    int rtrn, msqid, command, choice;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;

    /*Get the msqid, and command.*/
    printf("Enter the msqid = ");
    scanf("%d", &msqid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
    printf("IPC_STAT    =  1\n");
    printf("IPC_SET     =  2\n");
    printf("IPC_RMID    =  3\n");
    printf("Entry       =  ");
    scanf("%d", &command);
```

Figure 6-5: **msgctl()** System Call Example (Sheet 1 of 4)

```
28        /*Check the values.*/
29        printf ("\nmsqid =%d, command = %d\n",
30            msqid, command);

31        switch (command)
32        {
33        case 1:    /*Use msgctl() to duplicate
34              the data structure for
35                      msqid in the msqid_ds area pointed
36                      to by buf and then print it out.*/
37            rtrn = msgctl(msqid, IPC_STAT,
38                buf);
39            printf ("\nThe USER ID = %d\n",
40                buf->msg_perm.uid);
41            printf ("The GROUP ID = %d\n",
42                buf->msg_perm.gid);
43            printf ("The operation permissions = 0%o\n",
44                buf->msg_perm.mode);
45            printf ("The msg_qbytes = %d\n",
46                buf->msg_qbytes);
47            break;
48        case 2:    /*Select and change the desired
49                       member(s) of the data structure.*/
50            /*Get the original data for this msqid
51                  data structure first.*/
52            rtrn = msgctl(msqid, IPC_STAT, buf);
53            printf("\nEnter the number for the\n");
54            printf("member to be changed:\n");
55            printf("msg_perm.uid   = 1\n");
56            printf("msg_perm.gid   = 2\n");
57            printf("msg_perm.mode  = 3\n");
58            printf("msg_qbytes     = 4\n");
59            printf("Entry          = ");
```

Figure 6-5: **msgctl**() System Call Example (Sheet 2 of 4)

```
60          scanf("%d", &choice);
61          /*Only one choice is allowed per
62            pass as an illegal entry will
63                cause repetitive failures until
64            msqid_ds is updated with
65                IPC_STAT.*/

66          switch(choice){
67          case 1:
68              printf("\nEnter USER ID = ");
69              scanf ("%d", &uid);
70              buf->msg_perm.uid = uid;
71              printf("\nUSER ID = %d\n",
72                  buf->msg_perm.uid);
73              break;
74          case 2:
75              printf("\nEnter GROUP ID = ");
76              scanf("%d", &gid);
77              buf->msg_perm.gid = gid;
78              printf("\nGROUP ID = %d\n",
79                  buf->msg_perm.gid);
80              break;
81          case 3:
82              printf("\nEnter MODE = ");
83              scanf("%o", &mode);
84              buf->msg_perm.mode = mode;
85              printf("\nMODE = 0%o\n",
86                  buf->msg_perm.mode);
87              break;
```

Figure 6-5: **msgctl()** System Call Example (Sheet 3 of 4)

```
88              case 4:
89                  printf("\nEnter msq_bytes = ");
90                  scanf("%d", &bytes);
91                  buf->msg_qbytes = bytes;
92                  printf("\nmsg_qbytes = %d\n",
93                      buf->msg_qbytes);
94                  break;
95              }

96              /*Do the change.*/
97              rtrn = msgctl(msqid, IPC_SET,
98                  buf);
99              break;

100         case 3:    /*Remove the msqid along with its
101                        associated message queue
102                        and data structure.*/
103             rtrn = msgctl(msqid, IPC_RMID, NULL);
104         }
105         /*Perform the following if the call is unsuccessful.*/
106         if(rtrn == -1)
107         {
108             printf ("\nThe msgctl system call failed!\n");
109             printf ("The error number = %d\n", errno);
110         }
111         /*Return the msqid upon successful completion.*/
112         else
113             printf ("\nMsgctl was successful for msqid = %d\n",
114                 msqid);
115         exit (0);
116     }
```

Figure 6-5: **msgctl()** System Call Example (Sheet 4 of 4)

Operations for Messages

This section gives a detailed description of using the **msgsnd**(2) and **msgrcv**(2) system calls, along with an example program which allows all of their capabilities to be exercised.

Using msgop

The synopsis found in the **msgop**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it. It returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **msgsnd**() returns a −1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system parameter.

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl**() IPC_SET control command, but only the super-user can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB). If the IPC_NOWAIT flag is set, the system call will fail and return a −1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Receiving Messages

The **msgrcv**() system call requires five arguments to be passed to it, and it returns an integer value.

Upon successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a −1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The **msgsz** argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the **msgflg** argument.

The **msgtyp** argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if there is not a message on the message queue of the desired type (**msgtyp**) to be received. If the IPC_NOWAIT flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. Msgflg can also specify that the system call fail if the message is longer than the **size** to be received; this is done by not setting the MSG_NOERROR flag in the **msgflg** argument (**msgflg** & MSG_NOERROR = 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv**().

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 6-6) is a menu driven program which allows all possible combinations of using the **msgsnd**() and **msgrcv**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgop**(2) entry in the *Sysv Programmer's Reference*. Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

sndbuf used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13) The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message

	size (MSGMAX) for the 3B2 Computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.
rcvbuf	used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)
***msgp**	used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers
i	used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd**() system call; it is also used as a counter to output the received message for the **msgrcv**() system call
c	used to receive the input character from the **getchar**() function (line 50)
flag	used to store the code of IPC_NOWAIT for the **msgsnd**() system call (line 61)
flags	used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv**() system call (line 117)
choice	used to store the code for sending or receiving (line 30)
rtrn	used to store the return values from all system calls
msqid	used to store and pass the desired message queue identifier for both system calls
msgsz	used to store and pass the **size** of the message to be sent or received
msgflg	used to pass the value of flag for sending or the value of flags for receiving
msgtyp	used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl**() (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd

When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored at the address of the variable **msgtyp** (line 42), and then (line 43) it is put into the mtype member of the data structure pointed to by **msgp**.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the mtext array of the data structure (lines 48-51). This will continue until an end of file is recognized which for the **getchar**() function is a control-d (CTRL-D) immediately following a carriage return (<CR>). When this happens, the **size** of the message is determined by adding one to the **i** counter (lines 52, 53) as it stored the message beginning in the zero array element of mtext. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of **msgsz**.

The message is immediately echoed from the mtext array of the **sndbuf** data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, IPC_NOWAIT is logically ORed with **msgflg**; otherwise, **msgflg** is set to zero.

The **msgsnd**() system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed which should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure which are updated. They are described as follows:

msg_qnum represents the total number of messages on the message queue; it is incremented by one.

msg_lspid contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.

msg_stime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored at the address of **msqid** (lines 100-103).

The message type is requested, and it is stored at the address of **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending upon the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of **msgsz** (lines 134-137).

The **msgrcv**() system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure which are updated; they are described as follows:

msg_qnum contains the number of messages on the message queue; it is decremented by one.

msg_lrpid contains the process identification (PID) of the last process receiving a message; it is set accordingly.

msg_rtime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the **msgop**() system calls follows. It is suggested that the program be put into a source file called **msgop.c** and then into an executable file called **msgop**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1   /*This is a program to illustrate
2   **the message operations, msgop(),
3   **system call capabilities.
4   */

5   /*Include necessary header files.*/
6   #include    <stdio.h>
7   #include    <sys/types.h>
8   #include    <sys/ipc.h>
9   #include    <sys/msg.h>

10  struct msgbuf1 {
11      long    mtype;
12      char    mtext[8192];
13  } sndbuf, rcvbuf, *msgp;

14  /*Start of main C language program*/
15  main()
16  {
17      extern int errno;
18      int i, c, flag, flags, choice;
19      int rtrn, msqid, msgsz, msgflg;
20      long mtype, msgtyp;
21      struct msqid_ds msqid_ds, *buf;
22      buf = &msqid_ds;
```

Figure 6-6: **msgop()** System Call Example (Sheet 1 of 7)

```
23        /*Select the desired operation.*/
24        printf("Enter the corresponding\n");
25        printf("code to send or\n");
26        printf("receive a message:\n");
27        printf("Send            =  1\n");
28        printf("Receive         =  2\n");
29        printf("Entry           =  ");
30        scanf("%d", &choice);

31        if(choice == 1) /*Send a message.*/
32        {
33            msgp = &sndbuf; /*Point to user send structure.*/

34            printf("\nEnter the msqid of\n");
35            printf("the message queue to\n");
36            printf("handle the message = ");
37            scanf("%d", &msqid);

38            /*Set the message type.*/
39            printf("\nEnter a positive integer\n");
40            printf("message type (long) for the\n");
41            printf("message = ");
42            scanf("%d", &msgtyp);
43            msgp->mtype = msgtyp;

44            /*Enter the message to send.*/
45            printf("\nEnter a message: \n");

46            /*A control-d (^d) terminates as
47              EOF.*/
```

Figure 6-6: **msgop()** System Call Example (Sheet 2 of 7)

```
48          /*Get each character of the message
49            and put it in the mtext array.*/
50          for(i = 0; ((c = getchar()) != EOF); i++)
51              sndbuf.mtext[i] = c;

52          /*Determine the message size.*/
53          msgsz = i + 1;

54          /*Echo the message to send.*/
55          for(i = 0; i < msgsz; i++)
56              putchar(sndbuf.mtext[i]);

57          /*Set the IPC_NOWAIT flag if
58            desired.*/
59          printf("\nEnter a 1 if you want the\n");
60          printf("the IPC_NOWAIT flag set:  ");
61          scanf("%d", &flag);
62          if(flag == 1)
63              msgflg |= IPC_NOWAIT;
64          else
65              msgflg = 0;

66          /*Check the msgflg.*/
67          printf("\nmsgflg = 0%o\n", msgflg);

68          /*Send the message.*/
69          rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
70          if(rtrn == -1)
71          printf("\nMsgsnd failed.  Error = %d\n",
72                  errno);
73          else {
74              /*Print the value of test which
75                    should be zero for successful.*/
76              printf("\nValue returned = %d\n", rtrn);
```

Figure 6-6: **msgop**() System Call Example (Sheet 3 of 7)

```
77              /*Print the size of the message
78                sent.*/
79              printf("\nMsgsz = %d\n", msgsz);

80              /*Check the data structure update.*/
81              msgctl(msqid, IPC_STAT, buf);

82              /*Print out the affected members.*/

83              /*Print the incremented number of
84                messages on the queue.*/
85              printf("\nThe msg_qnum = %d\n",
86                  buf->msg_qnum);
87              /*Print the process id of the last sender.*/
88              printf("The msg_lspid = %d\n",
89                  buf->msg_lspid);
90              /*Print the last send time.*/
91              printf("The msg_stime = %d\n",
92                  buf->msg_stime);
93          }
94      }

95      if(choice == 2)  /*Receive a message.*/
96      {
97          /*Initialize the message pointer
98            to the receive buffer.*/
99          msgp = &rcvbuf;

100         /*Specify the message queue which contains
101               the desired message.*/
102         printf("\nEnter the msqid = ");
103         scanf("%d", &msqid);
```

Figure 6-6: **msgop**() System Call Example (Sheet 4 of 7)

```
104          /*Specify the specific message on the queue
105               by using its type.*/
106          printf("\nEnter the msgtyp = ");
107          scanf("%d", &msgtyp);

108          /*Configure the control flags for the
109               desired actions.*/
110          printf("\nEnter the corresponding code\n");
111          printf("to select the desired flags: \n");
112          printf("No flags                    =  0\n");
113          printf("MSG_NOERROR                 =  1\n");
114          printf("IPC_NOWAIT                  =  2\n");
115          printf("MSG_NOERROR and IPC_NOWAIT  =  3\n");
116          printf("               Flags        =  ");
117          scanf("%d", &flags);

118          switch(flags) {
119              /*Set msgflg by ORing it with the appropriate
120                        flags (constants).*/
121          case 0:
122              msgflg = 0;
123              break;
124          case 1:
125              msgflg |= MSG_NOERROR;
126              break;
127          case 2:
128              msgflg |= IPC_NOWAIT;
129              break;
130          case 3:
131              msgflg |= MSG_NOERROR | IPC_NOWAIT;
132              break;
133          }
```

Figure 6-6: **msgop()** System Call Example (Sheet 5 of 7)

```
134          /*Specify the number of bytes to receive.*/
135          printf("\nEnter the number of bytes\n");
136          printf("to receive (msgsz) = ");
137          scanf("%d", &msgsz);

138          /*Check the values for the arguments.*/
139          printf("\nmsqid =%d\n", msqid);
140          printf("\nmsgtyp = %d\n", msgtyp);
141          printf("\nmsgsz = %d\n", msgsz);
142          printf("\nmsgflg = 0%o\n", msgflg);

143          /*Call msgrcv to receive the message.*/
144          rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

145          if(rtrn == -1)  {
146              printf("\nMsgrcv failed.  ");
147              printf("Error = %d\n", errno);
148          }
149          else {
150              printf ("\nMsgctl was successful\n");
151              printf("for msqid = %d\n",
152                  msqid);

153              /*Print the number of bytes received,
154                it is equal to the return
155                value.*/
156              printf("Bytes received = %d\n", rtrn);
```

Figure 6-6: **msgop**() System Call Example (Sheet 6 of 7)

```
157                 /*Print the received message.*/
158                 for(i = 0; i<=rtrn; i++)
159                     putchar(rcvbuf.mtext[i]);
160             }
161             /*Check the associated data structure.*/
162             msgctl(msqid, IPC_STAT, buf);
163             /*Print the decremented number of messages.*/
164             printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165             /*Print the process id of the last receiver.*/
166             printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167             /*Print the last message receive time*/
168             printf("The msg_rtime = %d\n", buf->msg_rtime);
169         }
170     }
```

Figure 6-6: **msgop()** System Call Example (Sheet 7 of 7)

Semaphores

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. Semaphore sets are created by using the **semget**(2) system call.

The process performing the **semget**(2) system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl**(), semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl**() to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop**(2) system call (which is documented in the *SysV Programmer's Reference*):

- incremented
- decremented

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop**(2) system call. To decrement a semaphore, a minus (–) value of the desired magnitude is passed.

The UNIX operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the **semop**(2) system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, the process is returned a known error code (−1), and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop**(2), semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed or one "nonblocking operation" is unsuccessful and none are changed. All of this is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX operating system to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation which is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (−1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes.The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- process identification (PID) performing last operation
- number of processes awaiting the semaphore value to become greater than its current value
- number of processes awaiting the semaphore value to equal zero
- semaphore value

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C Programming Language data structure definition for the semaphore set (array member) is as follows:

```
struct sem
{
        ushort  semval;         /* value of semaphore */
        short   sempid;         /* pid of last operation */
        ushort  semncnt;        /* # awaiting semval > cval */
        ushort  semzcnt;        /* # awaiting semval = 0 */
};
```

It is located in the **#include <sys/sem.h>** header file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```
struct semid_ds
{
        struct ipc_perm sem_perm;   /* operation permission struct */
        struct sem      *sem_base;  /* ptr to first semaphore in set */
        ushort          sem_nsems;  /* # of semaphores in set */
        time_t          sem_otime;  /* last semop time */
        time_t          sem_ctime;  /* last change time */
};
```

It is also located in the **#include <sys/sem.h>** header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 6-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the "Messages" section.

The **semget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **semflg** argument that it receives:

- to get a new **semid** and create an associated data structure and semaphore set for it
- to return an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget**(2) system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it.

There is also a provision for specifying a **key** of value zero (0) which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **semid** is always returned with an associated data structure and semaphore set created for it. When the **ipcs** command is performed, the KEY field for the **semid** is all zeros.

When performing the first task, the process which calls **semget**() becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Semaphores" section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned, a control command (IPC_EXCL) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section of this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**(2)] and semaphore control [**semctl**()] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop**(). Refer to the "Operations on Semaphores" section in this chapter for details of this system call.

Semaphore control is done by using the **semctl**(2) system call. These control operations permit you to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values, status, of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular **semid** from the UNIX operating system along with its associated data structure and semaphore set

Refer to the "Controlling Semaphores" section in this chapter for details of the **semctl**(2) system call.

Getting Semaphores

This section contains a detailed description of using the **semget**(2) system call along with an example program illustrating its use.

Using semget

The synopsis found in the **semget**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

int  semget (key, nsems, semg)
key_t  key;
int nsems, semg;
```


The following line in the synopsis:

```
int semget (key, nsems, semflg)
```

informs you that **semget**() is a function with three formal arguments that returns an integer type value, upon successful completion (**semid**). The next two lines:

```
key_t  key;
int nsems, semflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier (**semid**) that was discussed above.

As declared, the process calling the **semget**() system call must supply three actual arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **semflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as "operation permissions." Figure 6-7 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

Figure 6-7: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define**'d in the **sem.h** header file which can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Figure 6-8 contains the names of the constants which apply to the **semget**(2) system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 6-8: Control Commands (Flags)

The value for **semflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
CW| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

The **semflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));

semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget**(2) entry in the *SysV Programmer's Reference*, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg**. The system call will attempt to return a new **semid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a **semid** associated with it, and (**semflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget ( 0, nsems, semflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

The second condition is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**semflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
   semflg = x 1 x x x    (x = immaterial)
& IPC_CREAT = 0 1 0 0 0

      result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or "true ."

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **semid** exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the key equals zero (IPC_PRIVATE).

Refer to the **semget**(2) manual page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (Figure 6-9) is a menu driven program which allows all possible combinations of using the **semget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **semget**(2) entry in the *SysV Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument
- **semid**—used for returning the semaphore set identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

The system call is made next, and the result is stored at the address of the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (−1), it is tested to see if an error occurred (line 63). If **semid** equals −1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget**(2) system call follows. It is suggested that the source program file be named **semget.c** and that the executable file be named **semget**.

```
1   /*This is a program to illustrate
2   **the semaphore get, semget(),
3   **system call capabilities.*/

4   #include    <stdio.h>
5   #include    <sys/types.h>
6   #include    <sys/ipc.h>
7   #include    <sys/sem.h>
8   #include    <errno.h>

9   /*Start of main C language program*/
10  main()
11  {
12      key_t key;      /*declare as long integer*/
13      int opperm, flags, nsems;
14      int semid, opperm_flags;

15      /*Enter the desired key*/
16      printf("\nEnter the desired key in hex = ");
17      scanf("%x", &key);

18      /*Enter the desired octal operation
19            permissions.*/
20      printf("\nEnter the operation\n");
21      printf("permissions in octal = ");
22      scanf("%o", &opperm);
```

Figure 6-9: **semget**() System Call Example (Sheet 1 of 3)

```
23        /*Set the desired flags.*/
24        printf("\nEnter corresponding number to\n");
25        printf("set the desired flags:\n");
26        printf("No flags                  = 0\n");
27        printf("IPC_CREAT                 = 1\n");
28        printf("IPC_EXCL                  = 2\n");
29        printf("IPC_CREAT and IPC_EXCL    = 3\n");
30        printf("           Flags          = ");
31        /*Get the flags to be set.*/
32        scanf("%d", &flags);

33        /*Error checking (debugging)*/
34        printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35            key, opperm, flags);
36        /*Incorporate the control fields (flags) with
37              the operation permissions.*/
38        switch (flags)
39        {
40        case 0:    /*No flags are to be set.*/
41            opperm_flags = (opperm | 0);
42            break;
43        case 1:    /*Set the IPC_CREAT flag.*/
44            opperm_flags = (opperm | IPC_CREAT);
45            break;
46        case 2:    /*Set the IPC_EXCL flag.*/
47            opperm_flags = (opperm | IPC_EXCL);
48            break;
49        case 3: /*Set the IPC_CREAT and IPC_EXCL
50                      flags.*/
51            opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52        }
```

Figure 6-9: **semget()** System Call Example (Sheet 2 of 3)

```
53        /*Get the number of semaphores for this set.*/
54        printf("\nEnter the number of\n");
55        printf("desired semaphores for\n");
56        printf("this set (25 max) = ");
57        scanf("%d", &nsems);

58        /*Check the entry.*/
59        printf("\nNsems = %d\n", nsems);

60        /*Call the semget system call.*/
61        semid = semget(key, nsems, opperm_flags);

62        /*Perform the following if the call is unsuccessful.*/
63        if(semid == -1)
64        {
65            printf("The semget system call failed!\n");
66            printf("The error number = %d\n", errno);
67        }
68        /*Return the semid upon successful completion.*/
69        else
70            printf("\nThe semid = %d\n", semid);
71        exit(0);
72    }
```

Figure 6-9: **semget**() System Call Example (Sheet 3 of 3)

Controlling Semaphores

This section contains a detailed description of using the **semctl**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using semctl

The synopsis found in the **semctl**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
        int val;
        struct semid_ds *bu;
        ushort array[];
} arg;
```

The **semctl**(2) system call requires four arguments to be passed to it, and it returns an integer value.

The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget**(2) system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following control commands (flags):

- GETVAL—return the value of a single semaphore within a semaphore set
- SETVAL—set the value of a single semaphore within a semaphore set

- GETPID—return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set
- GETNCNT—return the number of processes waiting for the value of a particular semaphore to become greater than its current value
- GETZCNT—return the number of processes waiting for the value of a particular semaphore to be equal to zero
- GETALL—return the values for all semaphores in a semaphore set
- SETALL—set all semaphore values in a semaphore set
- IPC_STAT—return the status information contained in the associated data structure for the specified **semid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**
- IPC_SET—for the specified semaphore set (**semid**), set the effective user/group identification and operation permissions
- IPC_RMID—remove the specified (**semid**) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

The **arg** argument is used to pass the system call the appropriate union member for the control command to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **semget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 6-10) is a menu driven program which allows all possible combinations of using the **semctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **semctl**(2) entry in the *SysV Programmer's Reference* Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- **semid_ds**—used to receive the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
- **c**—used to receive the input values from the **scanf**(3S) function, (line 117) when performing a SETALL control command
- **i**—used as a counter to increment through the union **arg.array** when displaying the semaphore values for a GETALL (lines 97-99) control command, and when initializing the **arg.array** when performing a SETALL (lines 115-119) control command
- **length**—used as a variable to test for the number of semaphores in a set against the **i** counter variable (lines 97, 115)
- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer from the system call which depends upon the control command or a −1 when unsuccessful
- **semid**—used to store and pass the semaphore set identifier to the system call
- **semnum**—used to store and pass the semaphore number to the system call

- **cmd**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member (**uid, gid, mode**) for the IPC_SET control command that is to be changed
- **arg.val**—used to pass the system call a value to set (SETVAL) or to store (GETVAL) a value returned from the system call for a single semaphore (union member)
- **arg.buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member)
- **arg.array**—used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the **sem.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The **arg** union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members.

The next important program aspect to observe is that although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl**(2) system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the **semnum** variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the **semnum** variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91).

Next, the system call is made and, upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl**(2) system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the UNIX operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl**(2) system call follows. It is suggested that the source program file be named **semctl.c** and that the executable file be named **semctl**.

```
1   /*This is a program to illustrate
2   **the semaphore control, semctl(),
3   **system call capabilities.
4   */

5   /*Include necessary header files.*/
6   #include    <stdio.h>
7   #include    <sys/types.h>
8   #include    <sys/ipc.h>
9   #include    <sys/sem.h>

10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      struct semid_ds semid_ds;
15      int c, i, length;
16      int uid, gid, mode;
17      int retrn, semid, semnum, cmd, choice;
18      union semun  {
19          int val;
20          struct semid_ds *buf;
21          ushort array[25];
22      } arg;

23      /*Initialize the data structure pointer.*/
24      arg.buf = &semid_ds;
```

Figure 6-10: **semctl**() System Call Example (Sheet 1 of 7)

```
25        /*Enter the semaphore ID.*/
26        printf("Enter the semid = ");
27        scanf("%d", &semid);

28        /*Choose the desired command.*/
29        printf("\nEnter the number for\n");
30        printf("the desired cmd:\n");
31        printf("GETVAL      =  1\n");
32        printf("SETVAL      =  2\n");
33        printf("GETPID      =  3\n");
34        printf("GETNCNT     =  4\n");
35        printf("GETZCNT     =  5\n");
36        printf("GETALL      =  6\n");
37        printf("SETALL      =  7\n");
38        printf("IPC_STAT    =  8\n");
39        printf("IPC_SET     =  9\n");
40        printf("IPC_RMID    =  10\n");
41        printf("Entry       =  ");
42        scanf("%d", &cmd);

43        /*Check entries.*/
44        printf ("\nsemid =%d, cmd = %d\n\n",
45            semid, cmd);

46        /*Set the command and do the call.*/
47        switch (cmd)
48        {
```

Figure 6-10: **semctl()** System Call Example (Sheet 2 of 7)

```
49        case 1: /*Get a specified value.*/
50            printf("\nEnter the semnum = ");
51            scanf("%d", &semnum);
52            /*Do the system call.*/
53            retrn = semctl(semid, semnum, GETVAL, 0);
54            printf("\nThe semval = %d\n", retrn);
55            break;
56        case 2: /*Set a specified value.*/
57            printf("\nEnter the semnum = ");
58            scanf("%d", &semnum);
59            printf("\nEnter the value = ");
60            scanf("%d", &arg.val);
61            /*Do the system call.*/
62            retrn = semctl(semid, semnum, SETVAL, arg.val);
63            break;
64        case 3: /*Get the process ID.*/
65            retrn = semctl(semid, 0, GETPID, 0);
66            printf("\nThe sempid = %d\n", retrn);
67            break;
68        case 4: /*Get the number of processes
69              waiting for the semaphore to
70              become greater than its current
71                    value.*/
72            printf("\nEnter the semnum = ");
73            scanf("%d", &semnum);
74            /*Do the system call.*/
75            retrn = semctl(semid, semnum, GETNCNT, 0);
76            printf("\nThe semncnt = %d", retrn);
77            break;
```

Figure 6-10: **semctl()** System Call Example (Sheet 3 of 7)

```
78          case 5: /*Get the number of processes
79                waiting for the semaphore
80                      value to become zero.*/
81              printf("\nEnter the semnum = ");
82              scanf("%d", &semnum);
83              /*Do the system call.*/
84              retrn = semctl(semid, semnum, GETZCNT, 0);
85              printf("\nThe semzcnt = %d", retrn);
86              break;

87          case 6: /*Get all of the semaphores.*/
88              /*Get the number of semaphores in
89                the semaphore set.*/
90              retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91              length = arg.buf->sem_nsems;
92              if(retrn == -1)
93                  goto ERROR;
94              /*Get and print all semaphores in the
95                specified set.*/
96              retrn = semctl(semid, 0, GETALL, arg.array);
97              for (i = 0; i < length; i++)
98              {
99                  printf("%d", arg.array[i]);
100                 /*Seperate each
101                   semaphore.*/
102                 printf("%c", ' ');
103             }
104             break;
```

Figure 6-10: **semctl**() System Call Example (Sheet 4 of 7)

```
105         case 7: /*Set all semaphores in the set.*/
106             /*Get the number of semaphores in
107               the set.*/
108             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109             length = arg.buf->sem_nsems;
110             printf("Length = %d\n", length);
111             if(retrn == -1)
112                 goto ERROR;
113             /*Set the semaphore set values.*/
114             printf("\nEnter each value:\n");
115             for(i = 0; i < length ; i++)
116             {
117                 scanf("%d", &c);
118                 arg.array[i] = c;
119             }
120             /*Do the system call.*/
121             retrn = semctl(semid, 0, SETALL, arg.array);
122             break;

123         case 8: /*Get the status for the semaphore set.*/
125             /*Get and print the current status values.*/
127             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
128             printf ("\nThe USER ID = %d\n",
129                 arg.buf->sem_perm.uid);
130             printf ("The GROUP ID = %d\n",
131                 arg.buf->sem_perm.gid);
132             printf ("The operation permissions = 0%o\n",
133                 arg.buf->sem_perm.mode);
134             printf ("The number of semaphores in set = %d\n",
135                 arg.buf->sem_nsems);
136             printf ("The last semop time = %d\n",
137                 arg.buf->sem_otime);
```

Figure 6-10: **semctl()** System Call Example (Sheet 5 of 7)

```
138             printf ("The last change time  = %d\n",
139                 arg.buf->sem_ctime);
140             break;

141         case 9:    /*Select and change the desired
142                        member of the data structure.*/
143             /*Get the current status values.*/
144             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145             if(retrn == -1)
146                 goto ERROR;
147             /*Select the member to change.*/
148             printf("\nEnter the number for the\n");
149             printf("member to be changed:\n");
150             printf("sem_perm.uid   = 1\n");
151             printf("sem_perm.gid   = 2\n");
152             printf("sem_perm.mode  = 3\n");
153             printf("Entry          = ");
154             scanf("%d", &choice);
155             switch(choice){

156             case 1: /*Change the user ID.*/
157                 printf("\nEnter USER ID = ");
158                 scanf ("%d", &uid);
159                 arg.buf->sem_perm.uid = uid;
160                 printf("\nUSER ID = %d\n",
161                     arg.buf->sem_perm.uid);
162                 break;

163             case 2: /*Change the group ID.*/
164                 printf("\nEnter GROUP ID = ");
165                 scanf("%d", &gid);
166                 arg.buf->sem_perm.gid = gid;
167                 printf("\nGROUP ID = %d\n",
168                     arg.buf->sem_perm.gid);
169                 break;
```

Figure 6-10: **semctl()** System Call Example (Sheet 6 of 7)

```
170             case 3: /*Change the mode portion of
171                   the operation
172                                 permissions.*/
173                 printf("\nEnter MODE = ");
174                 scanf("%o", &mode);
175                 arg.buf->sem_perm.mode = mode;
176                 printf("\nMODE = 0%o\n",
177                     arg.buf->sem_perm.mode);
178                 break;
179             }
180             /*Do the change.*/
181             retrn = semctl(semid, 0, IPC_SET, arg.buf);
182             break;
183         case 10:    /*Remove the semid along with its
184                       data structure.*/
185             retrn = semctl(semid, 0, IPC_RMID, 0);
186         }
187         /*Perform the following if the call is unsuccessful.*/
188         if(retrn == -1)
189         {
190     ERROR:
191             printf ("\n\nThe semctl system call failed!\n");
192             printf ("The error number = %d\n", errno);
193             exit(0);
194         }
195         printf ("\n\nThe semctl system call was successful\n");
196         printf ("for semid = %d\n", semid);
197         exit (0);
198     }
```

Figure 6-10: **semctl()** System Call Example (Sheet 7 of 7)

Operations on Semaphores

This section contains a detailed description of using the **semop**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using semop

The synopsis found in the **semop**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

The **semop**(2) system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned and when unsuccessful it returns a −1.

The **semid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget**(2) system call.

The **sops** argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed

- the control command (flags)

The ****sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. Sem**buf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file. The **nsops** argument specifies the length of the array (the number of structures in the array).

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- IPC_NOWAIT—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- SEM_UNDO—this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the IPC_NOWAIT flag set. That is, the blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations with the SEM_UNDO flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (Figure 6-11) is a menu driven program which allows all possible combinations of using the **semop**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *SysV Programmer's Reference* Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local to the program. The variables declared for this program and their purpose are as follows:

- **sembuf**[10]—used as an array buffer (line 14) to contain a maximum of ten **sembuf** type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop**(2) system call
- ***sops**—used as a pointer (line 14) to **sembuf**[10] for the system call and for accessing the structure members within the array
- **rtrn**—used to store the return values from the system call
- **flags**—used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop**(2) system call (line 60)
- **i**—used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- **nsops**—used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM
- **semid**—used to store the desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). Semid is stored at the address of the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). Sem**buf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl**() GETALL control command.

The example program for the **semop**(2) system call follows. It is suggested that the source program file be named **semop.c** and that the executable file be named **semop**.

```
1   /*This is a program to illustrate
2   **the semaphore operations, semop(),
3   **system call capabilities.
4   */

5   /*Include necessary header files.*/
6   #include    <stdio.h>
7   #include    <sys/types.h>
8   #include    <sys/ipc.h>
9   #include    <sys/sem.h>
10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      struct sembuf sembuf[10], *sops;
15      char string[];
16      int retrn, flags, sem_num, i, semid;
17      unsigned nsops;
18      sops = sembuf; /*Pointer to array sembuf.*/

19      /*Enter the semaphore ID.*/
20      printf("\nEnter the semid of\n");
21      printf("the semaphore set to\n");
22      printf("be operated on = ");
23      scanf("%d", &semid);
24      printf("\nsemid = %d", semid);
```

Figure 6-11: **semop**(2) System Call Example (Sheet 1 of 4)

```
25        /*Enter the number of operations.*/
26        printf("\nEnter the number of semaphore\n");
27        printf("operations for this set = ");
28        scanf("%d", &nsops);
29        printf("\nnosops = %d", nsops);

30        /*Initialize the array for the
31          number of operations to be performed.*/
32        for(i = 0; i < nsops; i++, sops++)
33        {

34            /*This determines the semaphore in
35              the semaphore set.*/
36            printf("\nEnter the semaphore\n");
37            printf("number (sem_num) = ");
38            scanf("%d", &sem_num);
39            sops->sem_num = sem_num;
40            printf("\nThe sem_num = %d", sops->sem_num);

41            /*Enter a (-)number to decrement,
42              an unsigned number (no +) to increment,
43              or zero to test for zero.  These values
44              are entered into a string and converted
45              to integer values.*/
46            printf("\nEnter the operation for\n");
47            printf("the semaphore (sem_op) = ");
48            scanf("%s", string);
49            sops->sem_op = atoi(string);
50            printf("\nsem_op = %d\n", sops->sem_op);
```

Figure 6-11: **semop**(2) System Call Example (Sheet 2 of 4)

```
51          /*Specify the desired flags.*/
52          printf("\nEnter the corresponding\n");
53          printf("number for the desired\n");
54          printf("flags:\n");
55          printf("No flags                    = 0\n");
56          printf("IPC_NOWAIT                  = 1\n");
57          printf("SEM_UNDO                    = 2\n");
58          printf("IPC_NOWAIT and SEM_UNDO     = 3\n");
59          printf("            Flags           = ");
60          scanf("%d", &flags);

61          switch(flags)
62          {
63          case 0:
64              sops->sem_flg = 0;
65              break;
66          case 1:
67              sops->sem_flg = IPC_NOWAIT;
68              break;
69          case 2:
70              sops->sem_flg = SEM_UNDO;
71              break;
72          case 3:
73              sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74              break;
75          }
76          printf("\nFlags = 0%o\n", sops->sem_flg);
77      }
```

Figure 6-11: **semop**(2) System Call Example (Sheet 3 of 4)

```
78        /*Print out each structure in the array.*/
79        for(i = 0; i < nsops; i++)
80        {
81            printf("\nsem_num = %d\n", sembuf[i].sem_num);
82            printf("sem_op = %d\n", sembuf[i].sem_op);
83            printf("sem_flg = %o\n", sembuf[i].sem_flg);
84            printf("%c", ' ');
85        }

86        sops = sembuf; /*Reset the pointer to
87                        sembuf[0].*/

88        /*Do the semop system call.*/
89        retrn = semop(semid, sops, nsops);
90        if(retrn == -1)  {
91            printf("\nSemop failed.  ");
92            printf("Error = %d\n", errno);
93        }
94        else {
95            printf ("\nSemop was successful\n");
96            printf("for semid = %d\n", semid);

97            printf("Value returned = %d\n", retrn);
98        }
99    }
```

Figure 6-11: **semop**(2) System Call Example (Sheet 4 of 4)

Shared Memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget**(2) system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat**(2) — shared memory attach
- **shmdt**(2) — shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl**(2) system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl**(2) system call.

System calls, which are documented in the *SysV Programmer's Reference*, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (−1) is returned to the process, and the external variable **errno** is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time
- last detach time
- last change time

The C Programming Language data structure definition for the shared memory segment data structure is located in the **/usr/include/sys/shm.h** header file. It is as follows:

```
/*
**    There is a shared mem id data structure for
**    each segment in the system.
*/

struct shmid_ds {
    struct ipc_perm    shm_perm;      /* operation permission struct */
    int                shm_segsz;     /* segment size */
    struct region      *shm_reg;      /* ptr to region structure */
    char               pad[4];        /* for swap compatibility */
    ushort             shm_lpid;      /* pid of last shmop */
    ushort             shm_cpid;      /* pid of creator */
    ushort             shm_nattch;    /* used only for shminfo */
    ushort             shm_cnattch;   /* used only for shminfo */
    time_t             shm_atime;     /* last shmat time */
    time_t             shm_dtime;     /* last shmdt time */
    time_t             shm_ctime;     /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 6-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the introduction section of "Messages."

The **shmget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **shmflg** argument that it receives:

- to get a new **shmid** and create an associated shared memory segment data structure for it
- to return an existing **shmid** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget**(2) system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it provided no system parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it. When the **ipcs** command is performed, the KEY field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (IPC_EXCL) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in the "Using **shmget**" section of this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Shared Memory" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop**()] and control [**shmctl**(2)] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat**(2) and **shmdt**(2). Refer to the "Operations for Shared Memory" section in this chapter for details of these system calls.

Shared memory segment control is done by using the **shmctl**(2) system call. It permits you to control the shared memory facility in the following ways:

- to determine the associated data structure status for a shared memory segment (**shmid**)
- to change operation permissions for a shared memory segment
- to remove a particular **shmid** from the UNIX operating system along with its associated shared memory segment data structure

Refer to the "Controlling Shared Memory" section in this chapter for details of the **shmctl**(2) system call.

Getting Shared Memory Segments

This section gives a detailed description of using the **shmget**(2) system call along with an example program illustrating its use.

Using shmget

The synopsis found in the **shmget**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int  shmget (key, size, shmflg)
key_t  key;
int size, shmflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX operating system. The following line in the synopsis:

```
int shmget (key, size, shmflg)
```

informs you that **shmget**(2) is a function with three formal arguments that returns an integer type value, upon successful completion (**shmid**). The next two lines:

```
key_t  key;
int size, shmflg;
```

declare the types of the formal arguments. The variable **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function upon successful completion is the shared memory identifier (**shmid**) that was discussed earlier.

As declared, the process calling the **shmget**(2) system call must supply three arguments to be passed to the formal **key**, **size**, and **shmflg** arguments.

A new **shmid** with an associated shared memory data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **shmflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **shmflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as "operation permissions." Figure 6-13 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 6-12: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **shm.h** header file which can be used for the user (OWNER). They are as follows:

SHM_R	0400
SHM_W	0200

Control commands are predefined constants (represented by all uppercase letters). Figure 6-14 contains the names of the constants that apply to the **shmget**() system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 6-13: Control Commands (Flags)

The value for **shmflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
shmflg	=	0 1 4 0 0	0 000 001 100 000 000

The **shmflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));

shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **shmget**(2) entry in the *SysV Programmer's ReferenceManual*, success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg**. The system call will attempt to return a new **shmid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0).
- Key does not already have a **shmid** associated with it, and (**shmflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmid = shmget ( 0 , size, shmflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**shmflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
    shmflg = x 1 x x x   (x = immaterial)
& IPC_CREAT = 0 1 0 0 0

    result = 0 1 0 0 0   (not zero)
```

Because the result is not zero, the flag is set or "true." SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** if the **key** equals zero (IPC_PRIVATE).

Refer to the **shmget**(2) manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 6-15) is a menu driven program which allows all possible combinations of using the **shmget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required header files as specified by the **shmget**(2) entry in the *SysV Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument
- **shmid**—used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one
- **size**—used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made next, and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 58). If **shmid** equals −1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget**(2) system call follows. It is suggested that the source program file be named **shmget.c** and that the executable file be named **shmget**.

When compiling C programs that use floating point operations, the **−f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1   /*This is a program to illustrate
2   **the shared memory get, shmget(),
3   **system call capabilities.*/

4   #include    <sys/types.h>
5   #include    <sys/ipc.h>
6   #include    <sys/shm.h>
7   #include    <errno.h>

8   /*Start of main C language program*/
9   main()
10  {
11      key_t key;              /*declare as long integer*/
12      int opperm, flags;
13      int shmid, size, opperm_flags;
14      /*Enter the desired key*/
15      printf("Enter the desired key in hex = ");
16      scanf("%x", &key);

17      /*Enter the desired octal operation
18        permissions.*/
19      printf("\nEnter the operation\n");
20      printf("permissions in octal = ");
21      scanf("%o", &opperm);
```

Figure 6-14: **shmget**(2) System Call Example (Sheet 1 of 3)

```
22        /*Set the desired flags.*/
23        printf("\nEnter corresponding number to\n");
24        printf("set the desired flags:\n");
25        printf("No flags                  = 0\n");
26        printf("IPC_CREAT                 = 1\n");
27        printf("IPC_EXCL                  = 2\n");
28        printf("IPC_CREAT and IPC_EXCL    = 3\n");
29        printf("           Flags          = ");
30        /*Get the flag(s) to be set.*/
31        scanf("%d", &flags);

32        /*Check the values.*/
33        printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34            key, opperm, flags);

35        /*Incorporate the control fields (flags) with
36          the operation permissions*/
37        switch (flags)
38        {
39        case 0:    /*No flags are to be set.*/
40            opperm_flags = (opperm | 0);
41            break;
42        case 1:    /*Set the IPC_CREAT flag.*/
43            opperm_flags = (opperm | IPC_CREAT);
44            break;
45        case 2:    /*Set the IPC_EXCL flag.*/
46            opperm_flags = (opperm | IPC_EXCL);
47            break;
48        case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
49            opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50        }
```

Figure 6-14: **shmget**(2) System Call Example (Sheet 2 of 3)

```
51        /*Get the size of the segment in bytes.*/
52        printf ("\nEnter the segment");
53        printf ("\nsize in bytes = ");
54        scanf ("%d", &size);

55        /*Call the shmget system call.*/
56        shmid = shmget (key, size, opperm_flags);

57        /*Perform the following if the call is unsuccessful.*/
58        if(shmid == -1)
59        {
60            printf ("\nThe shmget system call failed!\n");
61            printf ("The error number = %d\n", errno);
62        }
63        /*Return the shmid upon successful completion.*/
64        else
65            printf ("\nThe shmid = %d\n", shmid);
66        exit(0);
67    }
```

Figure 6-14: **shmget**(2) System Call Example (Sheet 3 of 3)

Controlling Shared Memory

This section gives a detailed description of using the **shmctl**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using shmctl

The synopsis found in the **shmctl**(2) entry in the *SysV Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The **shmctl**(2) system call requires three arguments to be passed to it, and **shmctl**(2) returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **shmctl**() returns a −1.

The **shmid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The **cmd** argument can be replaced by one of following control commands (flags):

- IPC_STAT—return the status information contained in the associated data structure for the specified **shmid** and place it in the data structure pointed to by the ***buf** pointer in the user memory area
- IPC_SET—for the specified **shmid**, set the effective user and group identification, and operation permissions
- IPC_RMID—remove the specified **shmid** along with its associated shared memory segment data structure

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Only the super-user can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 6-16) is a menu driven program which allows all possible combinations of using the **shmctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmctl**(2) entry in the *SysV Programmer's Reference*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer value from the system call
- **shmid**—used to store and pass the shared memory segment identifier to the system call
- **command**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the **shmid_ds** data structure in this program (line 16) uses the data structure located in the **shm.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier which is stored at the address of the **shmid** variable (lines 18-20). This is required for every **shmctl**(2) system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127).

The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132-135), and the **shmid** along with its associated message queue and data structure are removed from the UNIX operating system. Note that the ***buf** pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **shmctl**(2) system call follows. It is suggested that the source program file be named **shmctl.c** and that the executable file be named **shmctl**.

```
1   /*This is a program to illustrate
2   **the shared memory control, shmctl(),
3   **system call capabilities.
4   */

5   /*Include necessary header files.*/
6   #include    <stdio.h>
7   #include    <sys/types.h>
8   #include    <sys/ipc.h>
9   #include    <sys/shm.h>

10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      int uid, gid, mode;
15      int rtrn, shmid, command, choice;
16      struct shmid_ds shmid_ds, *buf;
17      buf = &shmid_ds;

18      /*Get the shmid, and command.*/
19      printf("Enter the shmid = ");
20      scanf("%d", &shmid);
21      printf("\nEnter the number for\n");
22      printf("the desired command:\n");
```

Figure 6-15: **shmctl**(2) System Call Example (Sheet 1 of 6)

```
23        printf("IPC_STAT    =  1\n");
24        printf("IPC_SET     =  2\n");
25        printf("IPC_RMID    =  3\n");
26        printf("SHM_LOCK    =  4\n");
27        printf("SHM_UNLOCK  =  5\n");
28        printf("Entry       =  ");
29        scanf("%d", &command);

30        /*Check the values.*/
31        printf ("\nshmid =%d, command = %d\n",
32            shmid, command);

33        switch (command)
34        {
35        case 1:    /*Use shmctl() to duplicate
36              the data structure for
37                      shmid in the shmid_ds area pointed
38                      to by buf and then print it out.*/
39            rtrn = shmctl(shmid, IPC_STAT,
40                buf);
41            printf ("\nThe USER ID = %d\n",
42                buf->shm_perm.uid);
43            printf ("The GROUP ID = %d\n",
44                buf->shm_perm.gid);
45            printf ("The creator's ID = %d\n",
46                buf->shm_perm.cuid);
47            printf ("The creator's group ID = %d\n",
48                buf->shm_perm.cgid);
49            printf ("The operation permissions = 0%o\n",
50                buf->shm_perm.mode);
51            printf ("The slot usage sequence\n");
```

Figure 6-15: **shmctl**(2) System Call Example (Sheet 2 of 6)

```
52              printf ("number = 0%x\n",
53                  buf->shm_perm.seq);
54              printf ("The key= 0%x\n",
55                  buf->shm_perm.key);
56              printf ("The segment size = %d\n",
57                  buf->shm_segsz);
58              printf ("The pid of last shmop = %d\n",
59                  buf->shm_lpid);
60              printf ("The pid of creator = %d\n",
61                  buf->shm_cpid);
62              printf ("The current # attached = %d\n",
63                  buf->shm_nattch);
64              printf("The in memory # attached = %d\n",
65                  buf->shm_cnattach);
66              printf("The last shmat time = %d\n",
67                  buf->shm_atime);
68              printf("The last shmdt time = %d\n",
69                  buf->shm_dtime);
70              printf("The last change time = %d\n",
71                  buf->shm_ctime);
72              break;

                /* Lines 73 - 87 deleted */
```

Figure 6-15: **shmctl**(2) System Call Example (Sheet 3 of 6)

```
88        case 2:    /*Select and change the desired
89                      member(s) of the data structure.*/

90            /*Get the original data for this shmid
91                 data structure first.*/
92            rtrn = shmctl(shmid, IPC_STAT, buf);

93            printf("\nEnter the number for the\n");
94            printf("member to be changed:\n");
95            printf("shm_perm.uid   = 1\n");
96            printf("shm_perm.gid   = 2\n");
97            printf("shm_perm.mode  = 3\n");
98            printf("Entry          = ");
99            scanf("%d", &choice);
100           /*Only one choice is allowed per
101             pass as an illegal entry will
102                 cause repetitive failures until
103             shmid_ds is updated with
104                 IPC_STAT.*/
```

Figure 6-15: **shmctl**(2) System Call Example (Sheet 4 of 6)

```
105         switch(choice){
106         case 1:
107             printf("\nEnter USER ID = ");
108             scanf ("%d", &uid);
109             buf->shm_perm.uid = uid;
110             printf("\nUSER ID = %d\n",
111                 buf->shm_perm.uid);
112             break;

113         case 2:
114             printf("\nEnter GROUP ID = ");
115             scanf("%d", &gid);
116             buf->shm_perm.gid = gid;
117             printf("\nGROUP ID = %d\n",
118                 buf->shm_perm.gid);
119             break;

120         case 3:
121             printf("\nEnter MODE = ");
122             scanf("%o", &mode);
123             buf->shm_perm.mode = mode;
124             printf("\nMODE = 0%o\n",
125                 buf->shm_perm.mode);
126             break;
127         }
128         /*Do the change.*/
129         rtrn = shmctl(shmid, IPC_SET,
130             buf);
131         break;
```

Figure 6-15: **shmctl**() System Call Example (Sheet 5 of 6)

```
132         case 3:    /*Remove the shmid along with its
133                        associated
134                        data structure.*/
135             rtrn = shmctl(shmid, IPC_RMID, NULL);
136             break;

137         case 4: /*Lock the shared memory segment*/
138             rtrn = shmctl(shmid, SHM_LOCK, NULL);
139             break;
140         case 5: /*Unlock the shared memory
141                        segment.*/
142             rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
143             break;
144         }
145         /*Perform the following if the call is unsuccessful.*/
146         if(rtrn == -1)
147         {
148             printf ("\nThe shmctl system call failed!\n");
149             printf ("The error number = %d\n", errno);
150         }
151         /*Return the shmid upon successful completion.*/
152         else
153             printf ("\nShmctl was successful for shmid = %d\n",
154                 shmid);
155         exit (0);
156     }
```

Figure 6-15: **shmctl**(2) System Call Example (Sheet 6 of 6)

Operations for Shared Memory

This section gives a detailed description of using the **shmat**(2) and **shmdt**(2) system calls, along with an example program which allows all of their capabilities to be exercised.

Using shmop

The synopsis found in the **shmop**(2) entry in the *SysV Programmer's Reference* is as follows:


```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat**(2) system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. Upon successful completion, this value will be the address in core memory where the process is attached to the shared memory segment and when unsuccessful it will be a –1.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The shm**addr** argument can be zero or user supplied when passed to the **shmat**(2) system call. If it is zero, the UNIX operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address. It would be wise to let the operating system pick addresses so as to improve portability.

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat**() system call.

Further details are discussed in the example program for **shmop**(). If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt**(2) system call requires one argument to be passed to it, and **shmdt**(2) returns an integer value.

Upon successful completion, zero is returned; and when unsuccessful, **shmdt**(2) returns a −1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 6-17) is a menu driven program which allows all possible combinations of using the **shmat**(2) and **shmdt**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *SysV Programmer's Reference*. Note that in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **flags**—used to store the codes of SHM_RND or SHM_RDONLY for the **shmat**(2) system call
- **addr**—used to store the address of the shared memory segment for the **shmat**(2) and **shmdt**(2) system calls
- **i**—used as a loop counter for attaching and detaching
- **attach**—used to store the desired number of attach operations
- **shmid**—used to store and pass the desired shared memory segment identifier
- **shmflg**—used to pass the value of flags to the **shmat**(2) system call
- **retrn**—used to store the return values from both system calls
- **detach**—used to store the desired number of detach operations

This example program combines both the **shmat**(2) and **shmdt**(2) system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 17-21).

A loop is entered using the attach variable and the i counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat**(2) system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66-68).

If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the i counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the **addr** variable (line 84). Then, the **shmdt**(2) system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the **shmop**(2) system calls follows. It is suggested that the program be put into a source file called **shmop.c** and then into an executable file called **shmop**.

```
1   /*This is a program to illustrate
2   **the shared memory operations, shmop(),
3   **system call capabilities.
4   */

5   /*Include necessary header files.*/
6   #include    <stdio.h>
7   #include    <sys/types.h>
8   #include    <sys/ipc.h>
9   #include    <sys/shm.h>
10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      int flags, addr, i, attach;
15      int shmid, shmflg, retrn, detach;



16      /*Loop for attachments by this process.*/
17      printf("Enter the number of\n");
18      printf("attachments for this\n");
19      printf("process (1-4).\n");
20      printf("       Attachments = ");

21      scanf("%d", &attach);
22      printf("Number of attaches = %d\n", attach);
```

Figure 6-16: **shmop()** System Call Example (Sheet 1 of 4)

```
23        for(i = 1; i <= attach; i++) {
24            /*Enter the shared memory ID.*/
25            printf("\nEnter the shmid of\n");
26            printf("the shared memory segment to\n");
27            printf("be operated on = ");
28            scanf("%d", &shmid);
29            printf("\nshmid = %d\n", shmid);


30            /*Enter the value for shmaddr.*/
31            printf("\nEnter the value for\n");
32            printf("the shared memory address\n");
33            printf("in hexadecimal:\n");
34            printf("            Shmaddr = ");
35            scanf("%x", &addr);
36            printf("The desired address = 0x%x\n", addr);

37            /*Specify the desired flags.*/
38            printf("\nEnter the corresponding\n");
39            printf("number for the desired\n");
40            printf("flags:\n");
41            printf("SHM_RND                 = 1\n");
42            printf("SHM_RDONLY              = 2\n");
43            printf("SHM_RND and SHM_RDONLY = 3\n");
44            printf("            Flags       = ");
45            scanf("%d", &flags);
```

Figure 6-16: **shmop**() System Call Example (Sheet 2 of 4)

```
46          switch(flags)
47          {
48          case 1:
49              shmflg = SHM_RND;
50              break;
51          case 2:
52              shmflg = SHM_RDONLY;
53              break;
54          case 3:
55              shmflg = SHM_RND | SHM_RDONLY;
56              break;
57          }
58          printf("\nFlags = 0%o\n", shmflg);

59          /*Do the shmat system call.*/
60          retrn = (int)shmat(shmid, addr, shmflg);
61          if(retrn == -1)  {
62              printf("\nShmat failed.  ");
63              printf("Error = %d\n", errno);
64          }
65          else {
66              printf ("\nShmat was successful\n");
67              printf("for shmid = %d\n", shmid);
68              printf("The address = 0x%x\n", retrn);
69          }
70      }

71      /*Loop for detachments by this process.*/
72      printf("Enter the number of\n");
73      printf("detachments for this\n");
74      printf("process (1-4).\n");
75      printf("       Detachments = ");
```

Figure 6-16: **shmop()** System Call Example (Sheet 3 of 4)

```
76          scanf("%d", &detach);
77          printf("Number of attaches = %d\n", detach);
78          for(i = 1; i <= detach; i++) {

79              /*Enter the value for shmaddr.*/
80              printf("\nEnter the value for\n");
81              printf("the shared memory address\n");
82              printf("in hexadecimal:\n");
83              printf("            Shmaddr = ");
84              scanf("%x", &addr);
85              printf("The desired address = 0x%x\n", addr);

86              /*Do the shmdt system call.*/
87              retrn = (int)shmdt(addr);
88              if(retrn == -1)  {
89                  printf("Error = %d\n", errno);
90              }
91              else {
92                  printf ("\nShmdt was successful\n");
93                  printf("for address  = 0%x\n", addr);

94              }
95          }
96      }
```

Figure 6-16: **shmop**() System Call Example (Sheet 4 of 4)

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *SysV Programmer's Guide: Volume I*
Order No.017270-A00

Your Name _____ Date _____

Organization _____

Street Address _____

City _____ State _____ Zip _____

Telephone number (____) _____

When you use the Apollo system, what job(s) do you perform?

- ☐ Programming
- ☐ Hardware Engineering
- ☐ Other (describe)_____
- ☐ Application End User
- ☐ System Administration

How many years of **experience** do you have in using the Apollo system:

_____

What programming **languages** do you use with the Apollo system?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold