CHAPTER 1
THE AWK PROGRAMMING LANGUAGE

A program is a solution to a problem, formulated in the syntax of a particular language. It is a small step from writing complex editing scripts with `sed` to writing programs with `awk`, but it is a step that many writers may fear to take. "Script" is less loaded a term than "program" for many people, but an editing script is still a program.

Each programming language has its own "style" that lends itself to performing certain tasks better than other languages. Anyone can scan a reference page and quickly learn a language's syntax, but a close examination of programs written in that language is usually required before you understand how to apply this knowledge. In this sense, a programming language is simply another tool; you need to know not only how to use it but also when and why it is used.

We recommend that you learn more than one programming language. We have already looked at a number of different programs or scripts written for and executed by the shell, `ex`, and `sed`. As you learn the `awk` programming language, you will notice similarities and differences. Not insignificantly, an `awk` script looks different from a shell script. The `awk` language shares many of the same basic constructs as the shell's programming language, yet `awk` requires a slightly different syntax. The `awk` program's basic operations are not much different from `sed`'s. reading standard input one line at a time, executing instructions that consist of two parts, *pattern* and *procedure*, and writing to standard output.

More importantly, `awk` has capabilities that make it the tool of choice for certain tasks. A programming language is itself a program that was written to solve certain kinds of problems for which adequate tools did not exist. The `awk` program was designed for text-processing applications, particularly those in which information is structured in records and fields. The major capabilities of `awk` that we will demonstrate in upcoming pages are as follows:

- definable record and field structure
- conditional and looping constructs
- assignment, arithmetic, relational, and logical operators
- numeric and associative arrays
- formatted print statements
- built-in functions

A quick comparison of a single feature will show you how one programming language can differ from another. You will find it much easier to perform arithmetic operations in `awk` than in the shell. To increment the value of `x` by 1 using the shell, you'd use the following line:

```
x=`expr $x + 1`
```

The `expr` command is a UNIX program that is executed as a separate process returning the value of its arguments. In `awk`, you only have to write:

```
++x
```

This is the same as `x = x + 1`. (This form could also be used in `awk`.)

□ Invoking awk □

The awk program itself is a program that, like sed, runs a specified program on lines of input. You can enter awk from the command line, or from inside a shell script.

```
$ awk 'program' files
```

Input is read a line at a time from one or more *files*. The *program*, enclosed in single quotation marks to protect it from the shell, consists of *pattern* and *procedure* sections. If the pattern is missing, the procedure is performed on all input lines:

```
$ awk '{print}' sample                    Prints all lines in sample file
```

The procedure is placed within braces. If the procedure is missing, lines matching the pattern are printed:

```
$ awk "/programmer's guide/" sample       Prints lines matching pattern
                                          in sample file
```

The awk program allows you to specify zero, one, or two pattern addresses, just like sed. Regular expressions are placed inside a pair of slashes (/). In awk, patterns can also be made up of expressions. An expression (or a primary expression so as not to confuse it with a regular expression) can be a string or numeric constant (for example, *red* or *1*), a variable (whose value is a string or numeric), or a function (which we'll look at later).

You can associate a pattern with a specific procedure as follows:

```
/pattern1/ { procedure1 }

/pattern2/ { procedure2 }

{ procedure 3 }
```

Like sed, only the lines matching the particular pattern are the object of a procedure, and a line can match more than one pattern. In this example, the third procedure is performed on all input lines. Usually, multi-line awk scripts are placed in a separate file and invoked using the -f option:

```
$ awk -f awkscript sample
```

□ Records and Fields □

Perhaps the most important feature of awk is that it divides each line of input into fields. In the simplest case, each field contains a single word, delimited by a blank space. The awk program allows you to reference these fields by their position in the input line, either in patterns or procedures. The symbol $0 represents the entire input line. $1, $2, ... refer, by their position in the input line, to individual fields.

We'll demonstrate some of these capabilities by building an awk program to search through a list of acronyms in a file. Each acronym is listed along with its meaning. If we print the first field of each line, we'll get the name of the acronym:

```
$ awk '{print $1}' sample
BASIC
CICS
COBOL
DBMS
GIGO
GIRL
```

We can construct a useful program that would allow you to specify an acronym and get its description. We could use awk just like grep:

```
$ awk '/BASIC/' sample
BASIC Beginner's All-Purpose Symbolic Instruction Code
```

However, there are three things we'd like to do to improve this program and make better use of awk's capabilities:

1. Limit the pattern-matching search.
2. Make the program more general and not dependent on the particular acronym that is the subject of the search.
3. Print only the description.

▫ Testing Fields ▫

The pattern as specified will match the word *BASIC* anywhere on the line. That is, it might match *BASIC* used in a description. To see if the first field (`$1`) matches the pattern, we write:

```
$1 == "BASIC"
```

The symbol == is a relational operator meaning "equal to" and is used to compare the first field of each line with the string `BASIC`. You could also construct this test using a given regular expression that looks for the acronym at the beginning of the line.

```
$1 ~ /^BASIC/
```

The pattern-matching operator ~ evaluates as true if an expression (`$1`) matches a regular expression. Its opposite, !~, evaluates true if the expression does not match the regular expression.

Although these two examples look very similar, they achieve very different results. The relational operator == evaluates true if the first field is *BASIC* but false if the first field is *BASIC,* (note the comma). The pattern-matching operator ~ locates both occurrences.

Pattern-matching operations must be performed on a regular expression (a string surrounded by slashes). Variables cannot be used inside a regular expression with the exception of shell variables, as shown in the next section. Constants cannot be evaluated using the pattern-matching operator.

▫ Passing Parameters from a Shell Script ▫

Our program is too specific and requires too much typing. We can put the `awk` script in a file and invoke it with the `-f` option. Or we can put the command inside a shell script, named for the function it performs. This shell script should be able to read the first argument from the command line (the name of the acronym) and pass it as a parameter to `awk`. We'll call the shell script `awkronym` and set it up to read a file named `acronyms`. Here's the simplest way to pass an argument into an `awk` procedure:

```
$ cat awkronym
awk '$1 == search' search=$1 acronyms
```

Parameters passed to an `awk` program are specified *after* the program. The search variable is set up to pass the first argument on the command line to the `awk` program. Even this gets confusing, because `$1` inside the `awk` program represents the first field of each input line, while `$1` in the shell represents the first argument supplied on the command line. Here's how this version of the program works:

```
$ awkronym CICS
CICS Customer Information Control System
```

By replacing the search string `BASIC` with a variable (which could be set to the string `CICS` or `BASIC`), we have a program that is fairly generalized.

Notice that we had to test the parameter as a string `$1 == search`). This is because we can't pass the parameter inside a regular expression. Thus, the expressions "`$1 ~ /search/`" or "`$1 ~`

search" will produce syntax errors.

As an aside, let's look at another way to import a shell variable into an awk program that even works inside a regular expression. However, it looks complicated:

```
search=$1
awk '$1 ~ /'"$search"'/' acronyms
```

This program works the same as the prior version (with the exception that the argument is evaluated inside a regular expression). Note that the first line of the script makes the variable assignment before awk is invoked. In the awk program, the shell variable is enclosed within single, then double, quotation marks. These quotes cause the shell to insert the value of $search inside the regular expression before it is interpreted by awk. Therefore, awk never sees the shell variable and evaluates it as a constant string.

You will come upon situations when you wish it were possible to place awk variables within regular expressions. As mentioned in the previous section, pattern matching allows us to search for a variety of occurrences. For instance, a field might also include incidental punctuation marks and would not match a fixed string unless the string included the specific punctuation mark. Perhaps there is some undocumented way of getting an awk variable interpreted inside a regular expression, or maybe there is a convoluted work-around waiting to be figured out.

□ Changing the Field Separator □

The awk program is oriented toward data arranged in fields and records. A record is normally a single line of input, consisting of one or more fields. The field separator is a blank space or tab and the record separator is a newline. For example, here's one record with five fields:

```
CICS Customer Information Control System
```

Field three or $3 is the string *Information*. In our program, we like to be able to print the description as a field. It is obvious that we can't just say `print $2` and get the entire description. But that is what we'd like to be able to do.

This will require that we change the input file using another character (other than a blank) to delimit fields. A tab is frequently used as a field separator. We'll have to insert a tab between the first and second fields:

```
$ cat acronyms
awk Aho, Weinstein & Kernighan
BASIC   Beginner's All-Purpose Symbolic Instruction Code
CICSCustomer Information Control System
COBOL   Common Business Orientated Language
DBMSData Base Management System
GIGOGarbage In, Garbage Out
GIRLGeneralized Information Retrieval Language
```

You can change the field separator from the command line using the -F option:

```
$ awk -F"|-----|" '$1 == search {print $2}' search=$1 acronyms
```

Note that |-----| is entered by typing a double quotation mark, pressing the *TAB* key, and typing a double quotation mark. This makes the tab character (represented in the example as |-----|) the exclusive field separator; spaces no longer serve to separate fields. Now that we've implemented all three enhancements, let's see how the program works:

```
$ awkronym GIGO
Garbage In, Garbage Out
```

▫ System Variables ▫

The awk program defines a number of special variables that can be referenced or reset inside a program. See Table 13-1.

Table 1.1 awk System Variables

System Variable	Meaning
FILENAME	Current filename
FS	Field separator (a blank)
NF	Number of fields in the current record
NR	Number of the current record
OFS	Output field separator (a blank)
ORS	Output record separator (a newline)
RS	Record separator (a newline)

The system variable FS defines the field separator used by awk. You can set FS inside the program as well as from the command line.

Typically, if you redefine the field or record separator, it is done as part of a BEGIN procedure. The BEGIN procedure allows you to specify an action that is performed before the first input line is read.

```
BEGIN { FS = "|-----|" }
```

You can also specify actions that are performed after all input is read by defining an END procedure.

The awk command sets the variable NF to the number of fields on the current line. Try running the following awk command on any text file:

```
$ awk '{print $NF}' test
```

If there are five fields in the current record, NF will be set to five; $NF refers to the fifth and last field. Shortly, we'll look at a program, double, that makes good use of this variable.

▫ Looping ▫

The awkronym program can print field two because we restructured the input file and redefined the field separator. Sometimes, this isn't practical, and you need another method to read or print a number of fields for each record. If the field separator is a blank or tab, the two records would have six and five fields, respectively.

```
BASIC Beginner's All-Purpose Symbolic Instruction Code
CICS Customer Information Control System
```

It is not unusual for records to have a variable number of fields. To print all but the first field, our program would require a loop that would be repeated as many times as there are fields remaining. In many awk programs, a loop is a commonly used procedure.

The while statement can be employed to build a loop. For instance, if we want to perform a procedure three times, we keep track of how many times we go through the loop by incrementing a variable at the bottom of the loop, then we check at the top of the loop to see if that variable is greater than 3. Let's take an example in which we print the input line three times.

```
{    i = 1
     while (i <= 3) {
          print
```

```
            ++i
            }
        }
```

Braces are required inside the loop to describe a procedure consisting of more than a single action. Three operators are used in this program: = assigns the value 1 to the variable `i`; <= compares the value of `i` to the constant 3; and ++ increments the variable by 1. The first time the `while` statement is encountered, `i` is equal to 1. Because the expression `i <= 3` is true, the procedure is performed. The last action of the procedure is to increment the variable `i`. The `while` expression is true after the end of the second loop has incremented `i` to 3. However, the end of the third loop increments `i` to 4 and the expression evaluates as false.

A `for` loop serves the same purpose as a `while` loop, but its syntax is more compact and easier to remember and use. Here's how the previous `while` statement is restructured as a `for` loop:

```
for (i = 1; i <= 3; i++)
    print
```

The `for` statement consists of three expressions within parentheses. The first expression, `i = 1`, sets the initial value for the counter variable. The second expression states a condition that is tested at the top of the loop. (The `while` statement tested the condition at the bottom of the loop.) The third expression increments the counter.

Now, to loop through remaining fields on the line, we have to determine how many times we need to execute the loop. The system variable `NF` contains the number of fields on the current input record. If we compare our counter (`i`) against `NF` each time through the loop, we'll be able to tell when all fields have been read:

```
for (i = 1; i <= NF; i++)
```

We will print out each field (`$i`), one to a line. Just to show how `awk` works, we'll print the record and field number before each field.

```
awk '{ for (i = 1; i <= NF; i++)
    print NR":"i, $i } ' $*
```

Notice that the print statement concatenates `NR`, a colon, and `i`. The comma produces an output field separator, which is a blank by default.

This program produces the following results on a sample file:

```
1:1 awk
1:2 Aho,
1:3 Weinstein
1:4 &
1:5 Kernighan
2:1 BASIC
2:2 Beginner's
2:3 All-Purpose
2:4 Symbolic
2:5 Instruction
2:6 Code
```

`Symbolic` is the fourth field of the second record. You might note that the sample file is `acronyms`, the one in which we inserted a tab character between the first and second fields. Because we did not change the default field separator, `awk` interpreted the tab or blank as a field separator. This allows you to write programs in which the special value of the tab is ignored.

Conditional Statements

Now let's change our example so that when given an argument, the program returns the record and field number where that argument appears.

Essentially, we want to test each field to see if it matches the argument; if it does, we want to print the record and field number. We need to introduce another flow control construct, the `if` statement. The `if` statement evaluates an expression — if true, it performs the procedure; if false, it does not.

In the next example, we use the `if` statement to test whether the current field is equal to the argument. If it is, the current record and field number are printed.

```
awk '{ for (i = 1; i <= NF; i++) {
     if ($i == search) {
          print NR":"i
          }
     }
} ' search=$1 acronyms
```

This new procedure prints *2:1* or *3:4* and isn't very useful by itself, but it demonstrates that you can retrieve and test any single field from any record.

The next program, `double`, checks if the first word on a line is a duplicate of the last word on the previous line. We use `double` in proofing documents and it catches a surprisingly common typing mistake.

```
awk '
NF > 0 {
     if ($1 == lastword) {
          print NR ": double " $1
          }
          lastword = $NF
}' $1
```

When the first line of input is read, if the number of fields is greater than 0, then the expression in the `if` statement is evaluated. Because the variable `lastword` has not been set, it evaluates to false. The final action assigns the value of `$NF` to the variable `lastword`. (`$NF` refers to the last field; the value of `NF` is the number of the last field.) When the next input line is read, the first word is compared against the value of `lastword`. If they are the same, a message is printed.

```
double sect1
15: double the
32: double a
```

This version of `double` is based on the program presented by Kernighan and Pike in *The UNIX Programming Environment*. (Writer's Workbench now includes this program.) Kernighan and Pike's program also checks for duplicate words, side-by-side, in the same line. You might try implementing this enhancement, using a `for` loop and checking the current field against the previous field. Another feature of Kernighan and Pike's `double` is that you can run the program on more than one file. To allow for additional files, you can change the shell variable from `$1` to `$*` but the record or line number printed by `NR` will correspond to consecutive input lines. Can you write a procedure to reset `NR` to 0 before reading input from a new file?

Arrays

The `double` program shows us how we can retain data by assigning it to a variable. In `awk`, unlike several other programming languages, variables do not have to be initialized before they are referenced in a program. In the previous program, we evaluated `lastword` at the top, although it was not actually assigned a value until the bottom of the program. The `awk` program initialized the variable, setting it to the null string or 0, depending upon whether the variable is referenced as a string or numeric value.

An array is a variable that allows you to store a list of items or elements. An array is analogous to a restaurant menu. Each item on this menu is numbered:

#1	tuna noodle casserole
#2	roast beef and gravy
#3	pork and beans

One way of ordering roast beef is to say simply "Number 2." Using ordinary variables, you would have had to define a variable `two` and assign it the value *roast beef and gravy*. An array is a way of referencing a group of related values. This might be written:

```
menu[choice]
```

where `menu` is the name of the array and `choice` is the *subscript* used to reference items in the array. Thus, `menu[1]` is equal to *tuna noodle casserole*. In `awk`, you don't have to declare the size of the array; you only have to load the array (before referencing it). If we put our three menu choices on separate lines in a file, we could load the array with the following statement.

```
menu[NR] = $0
```

The variable `NR`, or record number, is used as the subscript for the array. Each input line is read into the next element in the array. We can print an individual element by referring to the value of the subscript (not the variable that set this value).

```
print menu[3]
```

This statement prints the third element in the array, which is *pork and beans*. If we want to refer to all the elements of this array, we can use a special version of the `for` loop. It has the following syntax:

`for (`*element* `in` *array*`)`

This statement can be used to descend the array to print all of the elements:

```
for (choice in menu)
    print menu[choice]
```

Each time through the loop, the variable `choice` is set to the next element in the array. The `menu` array is an example of an array that uses a numeric subscript as an index to the elements.

Now, let's use arrays to increase the functionality of `awkronym`. Our new version will read acronyms from a file and load them into an array; then we'll read a second file and search for the acronyms. Basically, we're reading one input file and defining keywords that we want to search for in other files. A similar program that reads a list of terms in a glossary might show where the words appear in a chapter. Let's see how it works first:

```
$ awkronym sect1
exposure to BASIC programming.
in COBOL and take advantage of a DBMS environment.
in COBOL and take advantage of a DBMS environment .
Of the high-level languages, BASIC is probably
```

Let's look at the program carefully.

```
awk ' {
if ( FILENAME == "acronyms" ) {
    acro_desc[NR] = $1
    next
    }
for ( name in acro_desc )
    for (i = 1; i <= NF; i++)
        if ($i == acro_desc[name] ) {
            print $0
            }
```

```
}' acronyms $*
```

The current filename is stored in the system variable `FILENAME`. The procedure within the first conditional statement is only performed while input is taken from `acronyms`. The `next` statement ends this procedure by reading the next line of input from the file. Thus, the program does not advance beyond this procedure until input is taken from a different file.

The purpose of the first procedure is to assign each acronym (`$1`) to an element of the array `acro_desc`; the record number (`NR`) indexes the array.

In the second half of the program, we start comparing each element in the array to each field of every record. This requires two `for` loops, one to cycle through the array for each input line, and one to read each field on that line for as many times as there are elements in the array. An `if` statement compares the current field to the current element of the array; if they are equal, then the line is printed.

The line is printed each time an acronym is found. In our test example, because there were two acronyms on a single line, the one line is duplicated. To change this, we could add next after the `print` statement.

What if we changed `awkronym` so that it not only scanned the file for the acronym, but printed the acronym with the description as well? If a line refers to *BASIC*, we'd like to add the description (*Beginner's All-Purpose Symbolic Instruction Code*). We can design such a program for use as a *filter* that prints all lines, regardless of whether or not a change has been made. To change the previous version, we simply move the `print` statement outside the conditional statement. However, there are other changes we must make as well. Here's the first part of the new version.

```
awk ' {
    if ( FILENAME == "acronyms" ) {
    split ( $0, fields, "|-----|" )
    acro_desc[fields[1]] = fields[2]
    next
    }
```

The records in `acronyms` use a tab as a field separator. Rather than change the field separator, we use the `split` function (we'll look at the syntax of this function later on) to give us an array named `fields` that has two elements, the name of the acronym and its description. This numeric array is then used in creating an associative array named `acro_desc`. An associative array lets us use a string as a subscript to the elements of an array. That is, given the name of the acronym, we can locate the element corresponding to the description. Thus the expression `acro_desc[GIGO]` will access `Garbage In, Garbage Out`.

Now let's look at the second half of the program:

```
for ( name in acro_desc )
    for (i = 1 ; i <= NF; i++)
        if ( $i == name) {
            $i = $i " ("acro_desc[name]")"
        }
print $0
```

Just like the previous version, we loop through the elements of the array and the fields for each record. At the heart of this section is the conditional statement that tests if the current field (`$i`) is equal to the subscript of the array (`name`). If the value of the field and the subscript are equal, we concatenate the field and the array element. In addition, we place the description in parentheses.

It should be clear why we make the comparison between `$i` and `name`, and not `acro_desc[name]`; the latter refers to an element, while the former refers to the subscript, the name of the acronym.

If the current field (`$i`) equals `BASIC` and the index of the array (`name`) is the string `BASIC`, then the value of the field is set to:

```
BASIC  (Beginner's All-Purpose Symbolic Instruction Code)
```

For this program to be practical, the description should be inserted for the first occurrence of an acronym, not each time. (After we've inserted the description of the acronym, we don't need the description any more.) We could redefine that element in the array after we've used it.

```
acro_desc[name] = name
```

In this instance, we simply make the element equal to the subscript. Thus, `acro_desc[BASIC]` is equal to *Beginner's All-Purpose Symbolic Instruction Code* at the beginning of the procedure, and equal to *BASIC* if a match has been made. There are two places where we test the element against the subscript with the expression "`(acro_desc[name] ! = name)`." The first place is after the `for` loop has read in a new element from `acro_desc`; a conditional statement ensures that we don't scan the next input record for an acronym that has already been found. The second place is when we test `$i` to see if it matches name; this test ensures that we don't make another match for the same acronym on that line.

```
if ( $i == name && acro_desc[name] ! = name)
```

This conditional statement evaluates a compound expression. The `&&` (and) boolean operator states a condition that both expressions have to be true for the compound expression to be true.

Another problem that we can anticipate is that we might produce lines that exceed 80 characters. After all, the descriptions are quite long. We can find out how many characters are in a string, using a built-in `awk` function, `length`. For instance, to evaluate the length of the current input record, we specify:

```
length($0)
```

The value of a function can be assigned to a variable or put inside an expression and evaluated.

```
if ( length($0) > 70 ) {
     if (i > 2 )
          $i = "\n" $i
     if (i+1 < NF )
          $(i+1) = "\n" $(i+1)
     }
```

The length of the current input record is evaluated after the description has been concatenated. If it is greater than 70 characters, then two conditions test where to put the newline. The first procedure concatenates a newline and the current field; thus we only want to perform this action when we are not near the beginning of a line (field greater than 2). The second procedure concatenates the newline and the next field `(i+1)` so that we check that we are not near the end of the line. The newline precedes the field in each of these operations. Putting it at the end of the field would result in a new line that begins with a space output with the next field.

Another way to handle the line break, perhaps more efficiently, is to use the `length` function to return a value for each field. By accumulating that value, we could specify a line break when a new field causes the total to exceed a certain number. We'll look at arithmetic operations in a later section.

Here's the full version of `awkronyms`:

```
awk ' {
     if ( FILENAME == "acronyms" ) {
     split ( $0, fields, "|-----|" )
     acro_desc[fields[1]]=fields[2]
     next
     }
for ( name in acro_desc )
     if (acro_desc[name] ! = name)
     for (i = 1; i <= NF; i++)
          if ($i == name && acro_desc[name] ! = name) {
          $i = $i " ("acro_desc[name]")"
          acro_desc[name] = name
          if (length ($0) > 70) {
               if (i > 2)
                    $i = "\n" $i
               if (i+1 < NF)
                    $(i+1) = "\n" $(i+1)
          }
     }
print $0
}' acronyms $*
```

And here's one proof that it works:

```
$ cat sect1
Most users of microcomputers have had some
exposure to BASIC programming.
Many data-processing applications are written
in COBOL and take advantage of a DBMS environment.
C, the language of the UNIX environment,
is used by systems programmers.
Of the high-level languages, BASIC is probably
the easiest to learn, and C is the most difficult.
Nonetheless, you will find the fundamental programming
constructs common to most languages.

$ awkronym sect1
Most users of microcomputers have had some
exposure to
BASIC (Beginner's All-Purpose Symbolic Instruction Code)
programming.
Many data-processing applications are
written in COBOL (Common Business Orientated Language)
and take advantage of a
DBMS (Data Base Management System) environment.
C, the language of the UNIX environment,
is used by systems programmers.
Of the high-level languages, BASIC is probably
the easiest to learn, and C is the most difficult.
Nonetheless, you will find the fundamental programming
constructs common to most languages.
```

Notice that the second reference to `BASIC` has not been changed. There are other features we might add to this program. For instance, we could use `awk`'s pattern-matching capabilities so that we don't make the change on lines containing macros, or on lines within pairs of certain macros, such as `.DS`/`.DE`.

Another version of this program could trademark certain terms or phrases in a document. For instance, you'd want to locate the first occurrence of UNIX and place `\(rg` after it.

□ awk Applications □

A shell program is an excellent way to gather data interactively and write it into a file in a format that can be read by `awk`. We're going to be looking at a series of programs for maintaining a project log. A shell script collects the name of a project and the number of hours worked on the project. An `awk` program totals the hours for each project and prints a report.

The file `day` is the shell script for collecting information and appending it to a file named `daily` in the user's home directory.

```
$ cat /usr/local/bin/day
case $# in
0) echo "Project: \c"; read proj; echo "Hours: \c"; read hrs;;
1) proj=$1; echo "Hours: \c"; read hrs;;
2) proj=$1; hrs=$2;;
esac
set `who am i`; name=$1; month=$3; day=$4;
echo $name"\t"$month $day"\t"$hrs"\t"$proj>>$HOME/daily
```

The `case` statement checks how many arguments are entered on the command line. If an argument is missing, the user is prompted to enter a value. Prompting is done through a pair of statements: `echo` and `read`. The `echo` command displays the prompt on the user's terminal; `\c` suppresses the carriage return at the end of the prompt. The `read` command waits for user input, terminated by a carriage return, and assigns it to a variable. Thus, the variables `proj` and `hrs` are defined by the end of the case statement.

The `set` command can be used to divide the output of a command into separate arguments (`$1`, `$2`, `$3 ...`). By executing the command `who am i` from within `set`, we supply the user's name and the day's date automatically. The `echo` command is used to write the information to the file. There are four fields, separated by tabs. (In the Bourne shell, the escape sequence `\t` produces a tab; you must use quotation marks to keep the backslash from being stripped off by the shell.)

Here's what `daily` contains for one user at the end of a week:

```
$ cat /usr/fred/daily
fred  Aug 4  7     Course Development
fred  Aug 5  4     Training class
fred  Aug 5  4     Programmer's Guide
fred  Aug 6  2     Administrative
fred  Aug 6  6     Text-processing book
fred  Aug 7  4     Course Development
fred  Aug 7  4     Text-processing book
fred  Aug 8  4     Training class
fred  Aug 8  3     Programmer's Guide
```

There are nine records in this file. Obviously, our input program does not enforce consistency in naming projects by the user.

Given this input, we'd like an `awk` program that reports the total number of hours for the week and gives us a breakdown of hours by project. At first pass, we need only be concerned with reading fields three and four. We can total the number of hours by accumulating the value of the third field.

```
total += $3
```

The += operator performs two functions: it adds `$3` to the current value of `total` and then assigns this value to `total`. It is the same as the statement:

```
total = total + $3
```

We can use an associative array to accumulate hours (`$3`) by project (`$4`).

```
hours[$4] += $3
```

Each time a record is read, the value of the third field is added to the accumulated value of `project[$4]`.

We don't want to print anything until all input records have been read. An `END` procedure prints the accumulated results. Here's the first version of `tot`:

```
awk '
    BEGIN { FS="|-----|" }
{
        total += $3
        hours[$4] += $3
}
    END {
        for (project in hours)
        print project, hours[project]
        print
        print "Total Hours:" , total
} ' $HOME/daily
```

Let's test the program:

```
$ tot
Course Development 11
Administrative 2
Programmer's Guide 7
Training class 8
Text-processing book 10

Total Hours:     38
```

The program performs the arithmetic tasks well, but the report lacks an orderly format. It would help to change the output field separator (OFS) to a tab. But the variable lengths of the project names prevent the project hours from being aligned in a single column. The `awk` program offers an alternative print statement, `printf`, which is borrowed from the C programming language.

Formatted Print Statements

The `printf` statement has two parts: the first is a quoted expression that describes the format specifications; the second is a sequence of arguments such as variable names. The two main format specifications are `%s` for strings and `%d` for decimals. (There are additional specifications for octal, hexadecimal, and non-integer numbers.) Unlike the regular `print` statement, `printf` does not automatically supply a newline. This can be specified as `\n`. A tab is specified as `\t`.

A simple `printf` statement containing string and decimal specifications is:

```
printf "%s\t%d\n" , project, hours[project ]
```

First project is output, then a tab (`\t`), the number of hours, and a newline (`\n`). For each format specification, you must supply a corresponding argument.

Unfortunately, such a simple statement does not solve our formatting problem. Here are sample lines that it produces:

```
Course Development 11
Administrative  2
Programmer's Guide 7
```

We need to specify a minimum field width so that the tab begins at the same position. The `printf` statement allows you to place this specification between the `%` and the conversion specification. You would use `%-20s` to specify a minimum field width of 20 characters in which the value is left justified. Without the minus sign, the value would be right justified, which is what we want for a decimal value.

```
END {
    for (project in hours)
    printf "%-20s\t%2d\n", project, hours[project]
    printf " \n\tTotal Hours:\t%2d\n", total
    }
```

Notice that literals, such as the string `Total Hours`, are placed in the first part, with the format specification.

Just as we use the `END` procedure to print the report, we can include a `BEGIN` procedure to print a header for the report:

```
BEGIN { FS="|-----|"
    printf "%20s%s\n\n", "PROJECT    ",  " HOURS"
    }
```

This shows an alternative way to handle strings. The following formatted report is displayed:

```
        PROJECT       HOURS

Course Development       11
Administrative            2
Programmer's Guide        7
Training class            8
Text-processing book     10

        Total Hours:     38
```

Defensive Techniques

After you have accomplished the basic task of a program — and the code at this point is fairly easy to understand — it is often a good idea to surround this core with "defensive" procedures designed to trap inconsistent input records and prevent the program from failing. For instance, in the `tot` program, we might want to check that the number of hours is greater than 0 and that the project description is not null for each input record. We can use a conditional expression, using the logical operator `&&`.

```
$3 > 0 && $4 ! = "" {
                        procedure
                    }
```

Both conditions must be true for the procedure to be executed. The logical operator `&&` signifies that if both conditions are true, the expression is true.

Another aspect of incorporating defensive techniques is error handling. In other words, what do we want to have happen after the program detects an error? The previous condition is set up so that if the procedure is not executed, the next line of input is read. In this example the program keeps going, but in other cases you might want the program to print an error message and halt if such an error is encountered.

However, a distinction between "professional" and "amateur" programmers might be useful. We are definitely in the latter camp, and we do not always feel compelled to write 100% user-proof programs. For one thing, defensive programming is quite time consuming and frequently tedious. Second, an amateur is at liberty to write programs that perform the way he or she expects them to; a professional has to write for an audience and must account for their expectations. Consider the possible uses and users of any program you write.

awk and nroff/troff

It is fairly easy to have an `awk` program generate the necessary codes for form reports. For instance, we enhanced the `tot` program to produce a `troff`-formatted report:

```
awk ' BEGIN { FS = "|-----|"
print ".ce"
print ".B "
print "PROJECT ACTIVITY REPORT"
print ".R"
print ".sp 2"
     }
NR == 1 {
     begday = $2
     }
$3 > 0 && $4 != "" {
          hours[$4] += $3
          total += $3
          endday = $2
          logname = $1
     }
     END {
printf "Writer: %s\n", logname
print ".sp"
printf "Period: %s to %s\n", begday, endday
print ".sp"
printf "%20s%s\n\n", "PROJECT    ", " HOURS"
print ".sp"
print ".nf"
print ".na"
     for (project in hours)
printf "%-20s\t%2d\n", project, hours[project]
print ".sp"
```

```
printf "Total Hours: \t %2d\n", total
print ".sp"
}'  $HOME/daily
```

We incorporated one additional procedure in this version to determine the weekly period. The start date of the week is taken from the first record (`NR == 1`). The last record provides the final day of the week.

As you can see, `awk` doesn't mind if you mix `print` and `printf` statements. The regular `print` command is more convenient for specifying literals, such as formatting codes, because the newline is automatically provided. Because this program writes to standard output, you could pipe the output directly to `nroff/troff`.

You can use `awk` to generate input to `tbl` and other `troff` preprocessors such as `pic`.

Multiline Records

In this section, we are going to take a look at a set of programs for order tracking. We developed these programs to help operate a small, mail-order publishing business. These programs could be easily adapted to track documents in a technical publications department.

Once again, we used a shell program, `take.orders`, for data entry. The program has two purposes: The first is to enter the customer's name and mailing address for later use in building a mailing list. The second is to display seven titles and prompt the user to enter the title number, the number of copies, and the price per copy. The data collected for the mailing list and the customer order are written to separate files.

Two sample customer order records follow:

```
Charlotte Smith
P.O  N61331 87 Y 045    Date: 03/14/87
#1 3  7.50
#2 3  7.50
#3 1  7.50
#4 1  7.50
#7 1  7.50

Martin S. Rossi
P.O  NONE   Date: 03/14/87
#1 2  7.50
#2 5  6.75
```

These are multiline records, that is, a newline is used as the field separator. A blank line separates individual records. For most programs, this will require that we redefine the default field separator and record separator. The field separator becomes a newline, and the record separator is null.

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

Let's write a simple program that multiplies the number of copies by the price. We want to ignore the first two lines of each record, which supply the customer's name, a purchase order number, and the date of the order. We only want to read the lines that specify a title. There are a few ways to do this. With `awk`'s pattern-matching capabilities, we could select lines beginning with a hash (`#`) and treat them as individual records, with fields separated by spaces.

```
awk ' /^#/ {
        amount = $2 * $3
        printf "%s %6.2f\n", $0, amount
        next
    }
{print}' $*
```

The main procedure only affects lines that match the pattern. It multiplies the second field by the third field, assigning the value to the variable `amount`. The `printf` conversion `%f` prints a floating-point number; `6.2` specifies a minimum field width of 6 and a precision of 2. Precision is the number of digits to the right of the decimal point; the default for `%f` is 6. We print the current record along with the value of the variable `amount`. If a line is printed within this procedure, the next line is read from standard input. Lines not matching the pattern are simply passed through. Let's look at how `addem` works:

```
$ addem orders
Charlotte Smith
P.O  N61331 87 Y 045     Date: 03/14/87
#1 3  7.50  22.50
#2 3  7.50  22.50
#3 1  7.50   7.50
#4 1  7.50   7.50
#7 1  7.50   7.50

Martin S. Rossi
P.O  NONE   Date: 03/14/87
#1 2  7.50  15.00
#2 5  6.75  33.75
```

Now, let's design a program that reads multiline records and accumulates order information for a report. This report should display the total number of copies and the total amount for each title. We also want totals reflecting all copies ordered and the sum of all orders.

We know that we will not be using the information in the first two fields of each record. However, each record has a variable number of fields, depending upon how many titles have been ordered. First, we check that the input record has at least three fields. Then a `for` loop reads all of the fields beginning with the third field:

```
NF >= 3 {
for (i = 3; i <= NF; ++i)
```

In database terms, each field has a value and each value can be further broken up into subvalues. That is, if the value of a field in a multiline record is a single line, subvalues are the words on that line. You have already seen the `split` function used to break up an input record; now we'll see it used to subdivide a field. The `split` function loads any *string* into an *array*, using a specified character as the subvalue separator.

`split(`*string*`, `*array*`, `*separator*`)`

The default subvalue separator is a blank. The `split` function returns the number of elements loaded into the array. The *string* can be a literal (in quotation marks) or a variable. For instance, let's digress a minute and look at an isolated use of `split`. Here's a person's name and title with each part separated by a comma:

```
title="George Travers, Research/Development, Alcuin Inc."
```

We can use split to divide this string and print it on three lines.

```
need = split (title, name, ",")
    print ".ne ", need
    for (part in name)
        print name[part]
```

This procedure prints each part on a separate line. The number of elements in the array (3) is saved in the variable `need`. This variable is passed as an argument to an `.ne` request, which tells `troff` to make sure there are at least three lines available at the bottom of the page before outputting the first line.

The `awk` program has twelve built-in functions, as shown in Table 13-2. Four of these are specialized arithmetic functions for cosine, sine, logarithm, and square root. The rest of these functions manipulate strings. (You have already seen how the `length` function works.) See Appendix A for the syntax of these functions.

Going back to our report generator, we need to split each field into subvalues. The variable `$i` will supply the value of the current field, subdivided as elements in the array `order`.

```
sv = split ($i, order)
```

```
    if (sv =-- 3) {
        procedure
    }
    else print "Incomplete Record"
```

Table 1.2 awk Built-in Functions

Function	Description
cos	Cosine
exp	Exponent
getline	Read input line
index	Return position of substring in string
int	Integer
length	Length of string
log	Logarithm
sin	Sine
split	Subdivide string into array
sprintf	Format string like printf
sqrt	Square root
substr	Substring extraction

The number of elements returned by the function is saved in the `sv` variable. This allows us to test that there are three subvalues. If there are not, the `else` statement is executed, printing the error message to the screen.

Next, we assign the individual elements of the array to a specific variable. This is mainly to make it easier to remember what each element represents.

```
title = order[1]
copies = order[2]
price = order[3]
```

Then a group of arithmetic operations are performed on these values.

```
amount = copies * price
total_vol += copies
total_amt += amount
vol[title] += copies
amt[title] += amount
```

These values are accumulated until the last input record is read. The `END` procedure prints the report.

Here's the complete program:

```
awk ' BEGIN { FS = "\n"; RS = "" }
NF >= 3 {
for (i = 3; i <= NF; ++i ) {
    sv = split ($i, order)
        if (sv == 3) {
            title = order[1]
            copies = order[2]
            price = order[3]
            amount = copies * price
            total_vol += copies
            total_amt += amount
            vol[title] += copies
            amt[title] += amount
            }
        else print " Incomplete Record"
}
    END {
        printf "%5s\t%10s\t%6s\n\n", "TITLE", \
        "COPIES SOLD", "TOTAL"
```

```
        for (title in vol)
    printf "%5s\t%10d\t$%7.2f\n", title, vol[title], \
        amt[title]
        "printf" " "%s\n", "-------------"
    printf "\t%s%4d\t$%7.2f\n", "Total " , total-vol, total_amt}'
    $*
```

In awk, arrays are one dimensional; a two-dimensional array stores two elements indexed by the same subscript. You can get a pseudo two-dimensional array in awk by defining two arrays that have the same subscript. We only need one for loop to read both arrays.

The addemup file, an order report generator, produces the following output:

```
$ addemup orders.today
TITLE  COPIES SOLD   TOTAL

   #1           5  $  37.50
   #2           8  $  56.25
   #3           1  $   7.50
   #4           1  $   7.50
   #7           1  $   7.50
-------------
        Total  16  $ 116.25
```

After you solve a programming problem, you will find that you can re-use that approach in other programs. For instance, the method used in the awkronym program to load acronyms into an array could be applied in the current example to read the book titles from a file and print them in the report. Similarly, you can use variations of the same program to print different reports. The construction of the next program is similar to the previous program. Yet the content of the report is quite different.

```
awk ' BEGIN { FS = "\n" ; RS = ""
printf "%-15s\t%10s\t%6s\n\n", "CUSTOMER", "COPIES SOLD", \
    "TOTAL"
    }
NF >= 3 {
customer = $1
    total_vol = 0
    total_amt = 0
for (i = 3; i <= NF; ++i) {
    split ($i, order)
    title = order[1]
    copies = order[2]
    price = order[3]
    amount = copies * price
    total-vol += copies
    total-amt += amount
}
printf "\t%s%4d\t$%7.2f\n", "Total ", total_vol, total_amt}'
}' $*
```

In this program, named summary, we print totals for each customer order. Notice that the variables total_vol and total_amt are reset to 0 whenever a new record is read. In the previous program, these values accumulated from one record to the next.

The summary program, reading a multiline record, produces a report that 1ists each record on a single line:

```
$ summary orders
CUSTOMER     COPIES SOLD   TOTAL

J. Andrews            7   $  52.50
John Peterson         4   $  30.00
Charlotte Miller     11   $  82.50
Dan Aspromonte      105   $ 787.50
Valerie S. Rossi      4   $  30.00
Timothy P. Justice    4   $  30.00
```

```
Emma  Fleming        25   $ 187.50
Antonio Pollan        5   $  37.50
Hugh Blair           15   $ 112.50
```

▫ Testing Programs ▫

Part of writing a program is designing one or more test cases. Usually this means creating a sample input file. It is a good idea to test a program at various stages of development. Each time you add a function, test it. For instance, if you implement a conditional procedure, test that the procedure is executed when the expression is true; test what happens when it is false. Program testing involves making sure that the syntax is correct *and* that the problem has been solved.

When `awk` encounters syntax errors it will tell you that it is "bailing out." Usually it will print the line number associated with the error. Syntax errors can be caused by a variety of mistakes, such as forgetting to quote strings or to close a procedure with a brace. Sometimes, it can be as minor as an extra blank space. The `awk` program's error messages are seldom helpful, and a persistent effort is often required to uncover the fault.

You might even see a UNIX system error message, such as the dreadful declaration:

```
Segmentation fault-core dumped.
```

Not to worry. Although your program has failed badly, you have not caused an earthquake or a meltdown. An image of "core" memory at the time of the error is saved or dumped in a file named `core`. Advanced programmers can use a debugging program to examine this image and determine where in memory the fault occurred. We just delete `core` and re-examine our code.

Again, check each construct as you add it to the program. If you wait until you have a large program, and it fails, you will often have difficulty finding the error. Not only that, but you are likely to make unnecessary changes, fixing what's not broken in an attempt to find out what is.

Checking that you have solved the problem you set out to tackle is a much larger issue. After you begin testing your program on larger samples, you will undoubtedly uncover "exceptions," otherwise known as *bugs*. In testing the `awkronym` program, we discovered an exception where an acronym appeared as the last word in the sentence. It was not "found" because of the period ending the sentence. That is, `awk` found that `BASIC` and `BASIC.` were not equal. This would not be a problem if we could test the search string as a regular expression but we have to test the array variable as a literal string.

Programming is chiefly pragmatic in its aims. You must judge whether or not specific problems merit writing a program or if certain exceptions are important enough to adapt the general program to account for them. Sometimes, in large public programs as well as small private ones, bugs just become part of the program's known behavior, which the user is left to cope with as best as he or she can. The bug found in `awkronym` is a common enough problem, so it is necessary to implement a fix.

The fix for the `awkronym` bug does not involve `awk` at all. We run a `sed` script before the `awkronym` program to separate punctuation marks from any word. It converts a punctuation mark to a field containing garbage characters. Another script processes the `awkronym` output and strips out these garbage characters. The example below shows how both scripts are used as bookends for the `awkronym` program.

```
sed 's/\ (..*\)\([.,!;]\)/\1 @@@\2/g' $* |
awk ' {
    program lines
}' acronyms - |
sed 's/ @@@\([.,!;]\)/\1/g'
```