Table of Contents

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Chapter 7. Writing Scripts for awk

As mentioned in the preface, this book describes POSIX awk; that is, the awk language as specified by the POSIX standard. Before diving into the details, we'll provide a bit of history.

The original awk was a nice little language. It first saw the light of day with Version 7 UNIX, around 1978. It caught on, and people used it for significant programming.

In 1985, the original authors, seeing that awk was being used for more serious programming than they had ever intended, decided to beef up the language. (See Chapter 11, for a description of the original awk, and all the things it did not have when compared to the new one.) The new version was finally released to the world at large in 1987, and it is this version that is still found on SunOS 4.1.x systems.

In 1989, for System V Release 4, awk was updated in some minor ways.[1] This version became the basis for the awk feature list in the POSIX standard. POSIX clarified a number of things about awk, and added the **CONVFMT** variable (to be discussed later in this chapter).

[1] The *-v* option and **tolower( )** and **toupper( )** functions were added, and **srand( )** and **printf** were cleaned up. The details will be presented in this and the following chapters.

As you read the rest of this book, bear in mind that the term **awk** refers to POSIX awk, and not to any particular implementation, whether the original one from Bell Labs, or any of the others discussed in Chapter 11. However, in the few cases where different versions have fundamental differences of behavior, that will be pointed out in the main body of the discussion.

7.1. Playing the Game

To write an awk script, you must become familiar with the rules of the game. The rules can be stated plainly and you will find them described in Appendix B, rather than in this chapter. The goal of this chapter is not to describe the rules but to show you how to play the game. In this way, you will become acquainted with many of the features of the language and see examples that illustrate how scripts actually work. Some people prefer to begin by reading the rules, which is roughly equivalent to learning to use a program from its manual page or learning to speak a language by scanning its rules of grammar—not an easy task. Having a good grasp of the rules, however, is essential once you begin to use awk regularly. But the more you use awk, the faster the rules of the game become second nature. You learn them through trial and error—spending a long time trying to fix a silly syntax error

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

such as a missing space or brace has a magical effect upon long-term memory. Thus, the best way to learn to write scripts is to begin writing them. As you make progress writing scripts, you will no doubt benefit from reading the rules (and rereading them) in Appendix B or the awk manpage or *The AWK Programming Language* book. You can do that later —let's get started now.

7.2. Hello, World

It has become a convention to introduce a programming language by demonstrating the "Hello, world" program. Showing how this program works in awk will demonstrate just how unconventional awk is. In fact, it's necessary to show several different approaches to printing "Hello, world."

In the first example, we create a file named *test* that contains a single line. This example shows a script that contains the **print** statement:

```
$ echo 'this line of data is ignored' > test
$ awk '{ print "Hello, world" }' test
Hello, world
```

This script has only a single action, which is enclosed in braces. That action is to execute the **print** statement for each line of input. In this case, the *test* file contains only a single line; thus, the action occurs once. Note that the input line is read but never output.

Now let's look at another example. Here, we use a file that contains the line "Hello, world."

```
$ cat test2
Hello, world
$ awk '{ print }' test2
Hello, world
```

In this example, "Hello, world" appears in the input file. The same result is achieved because the **print** statement, without arguments, simply outputs each line of input. If there were additional lines of input, they would be output as well.

Both of these examples illustrate that awk is usually input-driven. That is, nothing happens unless there are lines of input on which to act. When you invoke the awk program, it reads the script that you supply, checking the syntax of your instructions. Then awk attempts to execute the instructions for each line of input. Thus, the **print** statement will not be executed unless there is input from the file.

To verify this for yourself, try entering the command line in the first example but omit the filename. You'll find that because awk expects input to come from the keyboard, it will wait until you give it input to process: press RETURN several times, then type an EOF (CTRL-D on most systems) to signal the end of input. For each time that you pressed RETURN, the action that prints "Hello, world" will be executed.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

There is yet another way to write the "Hello, world" message and not have awk wait for input. This method associates the action with the **BEGIN** pattern. The **BEGIN** pattern specifies actions that are performed *before* the first line of input is read.

```
$ awk 'BEGIN { print "Hello, world" }'
Hello, world
```

Awk prints the message, and then exits. If a program has only a **BEGIN** pattern, and no other statements, awk will not process any input files.

7.3. Awk's Programming Model

It's important to understand the basic model that awk offers the programmer. Part of the reason why awk is easier to learn than many programming languages is that it offers such a well-defined and useful model to the programmer.

An awk program consists of what we will call a *main input loop*. A *loop* is a routine that is executed over and over again until some condition exists that terminates it. You don't write this loop, it is given—it exists as the framework within which the code that you do write will be executed. The main input loop in awk is a routine that reads one line of input from a file and makes it available for processing. The actions you write to do the processing assume that there is a line of input available. In another programming language, you would have to create the main input loop as part of your program. It would have to open the input file and read one line at a time. This is not necessarily a lot of work, but it illustrates a basic awk shortcut that makes it easier for you to write your program.

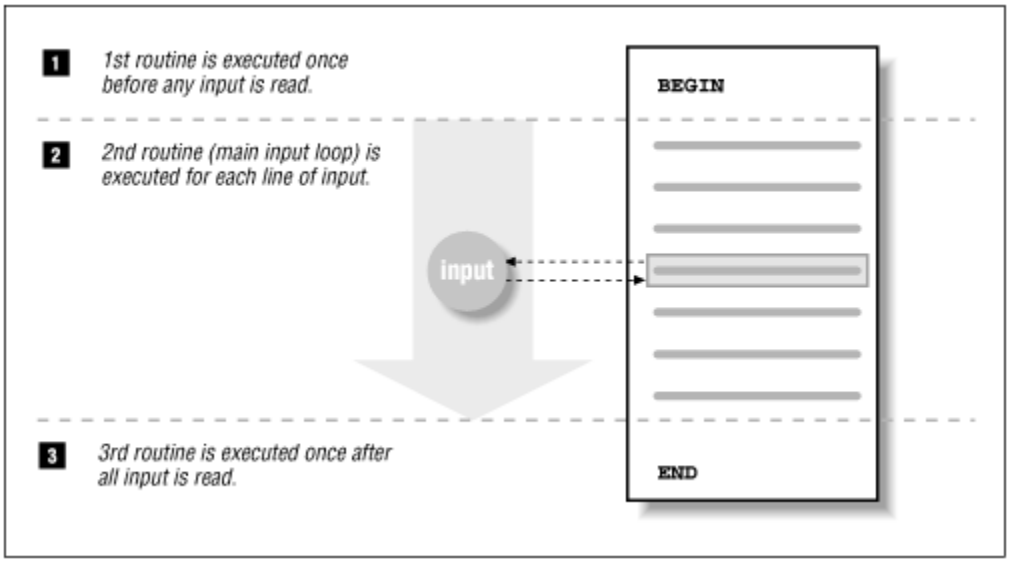
The main input loop is executed as many times as there are lines of input. As you saw in the "Hello, world" examples, this loop does not execute until there is a line of input. It terminates when there is no more input to be read.

Awk allows you to write two special routines that can be executed *before* any input is read and *after* all input is read. These are the procedures associated with the **BEGIN** and **END** rules, respectively. In other words, you can do some preprocessing before the main input loop is ever executed and you can do some postprocessing after the main input loop has terminated. The **BEGIN** and **END** procedures are optional.

You can think of an awk script as having potentially three major parts: what happens before, what happens during, and what happens after processing the input. Figure 7.1 shows the relationship of these parts in the flow of control of an awk script.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Figure 7.1. Flow and control in awk scripts



Of these three parts, the main input loop or "what happens during processing" is where most of the work gets done. Inside the main input loop, your instructions are written as a series of pattern/action procedures. A pattern is a rule for testing the input line to determine whether or not the action should be applied to it. The actions, as we shall see, can be quite complex, consisting of statements, functions, and expressions.

The main thing to remember is that each pattern/action procedure sits in the main input loop, which takes care of reading the input line. The procedures that you write will be applied to each input line, one line at a time.

7.4. Pattern Matching

The "Hello, world" program does not demonstrate the power of pattern-matching rules. In this section, we look at a number of small, even trivial examples that nonetheless demonstrate this central feature of awk scripts.

When awk reads an input line, it attempts to match each pattern-matching rule in a script. Only the lines matching the particular pattern are the object of an action. If no action is specified, the line that matches the pattern is printed (executing the **print** statement is the default action). Consider the following script:

```
/^$/ { print "This is a blank line." }
```

This script reads: *if the input line is blank, then print "This is a blank line."* The pattern is written as a regular expression that identifies a blank line. The action, like most of those we've seen so far, contains a single **print** statement.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

If we place this script in a file named *awkscr* and use an input file named *test* that contains three blank lines, then the following command executes the script:

```
$ awk -f awkscr test
This is a blank line.
This is a blank line.
This is a blank line.
```

(From this point on, we'll assume that our scripts are placed in a separate file and invoked using the *-f* command-line option.) The result tells us that there are three blank lines in *test*. This script ignores lines that are not blank.

Let's add several new rules to the script. This script is now going to analyze the input and classify it as an integer, a string, or a blank line.

```
# test for integer, string or empty line.
/[0-9]+/    { print "That is an integer" }
/[A-Za-z]+/ { print "This is a string" }
/^$/        { print "This is a blank line." }
```

The general idea is that if a line of input matches any of these patterns, the associated **print** statement will be executed. The + metacharacter is part of the extended set of regular expression metacharacters and means "one or more." Therefore, a line containing a sequence of one or more digits will be considered an integer. Here's a sample run, taking input from standard input:

```
$ awk -f awkscr
4
That is an integer
t
This is a string
4T
That is an integer
This is a string
RETURN
This is a blank line.
44
That is an integer
CTRL-D
$
```

Note that input "4T" was identified as both an integer and a string. A line can match more than one rule. You can write a stricter rule set to prevent a line from matching more than one rule. You can also write actions that are designed to skip other parts of the script.

We will be exploring the use of pattern-matching rules throughout this chapter.

7.4.1. Describing Your Script

Adding comments as you write the script is a good practice. A comment begins with the "#" character and ends at a newline. Unlike sed, awk allows comments anywhere in the script.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

If you are supplying your awk program on the command line, rather than putting it in a file, do not use a single quote anywhere in your program. The shell would interpret it and become confused.

As we begin writing scripts, we'll use comments to describe the action:

```
#  blank.awk -- Print message for each blank line.
/^$/ { print "This is a blank line." }
```

This comment offers the name of the script, **blank.awk**, and briefly describes what the script does. A particularly useful comment for longer scripts is one that identifies the expected structure of the input file. For instance, in the next section, we are going to look at writing a script that reads a file containing names and phone numbers. The introductory comments for this program should be:

```
# blocklist.awk -- print name and address in block form.
# fields: name, company, street, city, state and zip, phone
```

It is useful to embed this information in the script because the script won't work unless the structure of the input file corresponds to that expected by the person who wrote the script.

7.5. Records and Fields

Awk makes the assumption that its input is structured and not just an endless string of characters. In the simplest case, it takes each input line as a record and each word, separated by spaces or tabs, as a field. (The characters separating the fields are often referred to as *delimiters*.) The following record in the file *names* has three fields, separated by either a space or a tab.

```
John Robinson    666-555-1111
```

Two or more consecutive spaces and/or tabs count as a single delimiter.

7.5.1. Referencing and Separating Fields

Awk allows you to refer to fields in actions using the field operator **$**. This operator is followed by a number or a variable that identifies the position of a field by number. "$1" refers to the first field, "$2" to the second field, and so on. "$0" refers to the entire input record. The following example displays the last name first and the first name second, followed by the phone number.

```
$ awk '{ print $2, $1, $3 }' names
Robinson John 666-555-1111
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

$1 refers to the first name, $2 to the last name, and $3 to the phone number. The commas that separate each argument in the **print** statement cause a space to be output between the values. (Later on, we'll discuss the output field separator (**OFS**), whose value the comma outputs and which is by default a space.) In this example, a single input line forms one record containing three fields: there is a space between the first and last names and a tab between the last name and the phone number. If you wanted to grab the first and last name as a single field, you could set the field separator explicitly so that only tabs are recognized. Then, awk would recognize only two fields in this record.

You can use any expression that evaluates to an integer to refer to a field, not just numbers and variables.

```
$ echo a b c d | awk 'BEGIN { one = 1; two = 2 }
> { print $(one + two) }'
c
```

You can change the field separator with the *-F* option on the command line. It is followed by the delimiter character (either immediately, or separated by whitespace). In the following example, the field separator is changed to a tab.

```
$ awk -F"\t" '{ print $2 }' names
666-555-1111
```

"\t" is an *escape sequence* (discussed below) that represents an actual tab character. It should be surrounded by single or double quotes.

Commas delimit fields in the following two address records.

```
John Robinson,Koren Inc.,978 4th Ave.,Boston,MA 01760,696-0987
Phyllis Chapman,GVE Corp.,34 Sea Drive,Amesbury,MA 01881,879-0900
```

An awk program can print the name and address in block format.

```
# blocklist.awk -- print name and address in block form.
# input file -- name, company, street, city, state and zip, phone
{     print ""    # output blank line
    print $1    # name
    print $2    # company
    print $3    # street
    print $4, $5    # city, state zip
}
```

The first **print** statement specifies an empty string ("") (remember, **print** by itself outputs the current line). This arranges for the records in the report to be separated by blank lines. We can invoke this script and specify that the field separator is a comma using the following command:

```
awk -F, -f blocklist.awk names
```

The following report is produced:

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
John Robinson
Koren Inc.
978 4th Ave.
Boston  MA 01760

Phyllis Chapman
GVE Corp.
34 Sea Drive
Amesbury  MA 01881
```

It is usually a better practice, and more convenient, to specify the field separator in the script itself. The system variable **FS** can be defined to change the field separator. Because this must be done before the first input line is read, we must assign this variable in an action controlled by the **BEGIN** rule.

```
BEGIN { FS = "," }
```

Now let's use it in a script to print out the names and phone numbers.

```
# phonelist.awk -- print name and phone number.
# input file -- name, company, street, city, state and zip, phone

BEGIN { FS = "," }  # comma-delimited fields

{ print $1 ", " $6 }
```

Notice that we use blank lines in the script itself to improve readability. The **print** statement puts a comma followed by a space between the two output fields. This script can be invoked from the command line:

```
$ awk -f phonelist.awk names
John Robinson, 696-0987
Phyllis Chapman, 879-0900
```

This gives you a basic idea of how awk can be used to work with data that has a recognizable structure. This script is designed to print all lines of input, but we could modify the single action by writing a pattern-matching rule that selected only certain names or addresses. So, if we had a large listing of names, we could select only the names of people residing in a particular state. We could write:

```
/MA/ { print $1 ", " $6 }
```

where MA would match the postal state abbreviation for Massachusetts. However, we could possibly match a company name or some other field in which the letters "MA" appeared. We can test a specific field for a match. The tilde (~) operator allows you to test a regular expression against a field.

```
$5 ~ /MA/   { print $1 ", " $6 }
```

You can reverse the meaning of the rule by using bang-tilde (!~).

```
$5 !~ /MA/   { print $1 ", " $6 }
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

This rule would match all those records whose fifth field did not have "MA" in it. A more challenging pattern-matching rule would be one that matches only long-distance phone numbers. The following regular expression looks for an area code.

```
$6 ~ /1?(-|)?\(?[0-9]+\)?(|-)?[0-9]+-[0-9]+/
```

This rule matches any of the following forms:

```
707-724-0000
(707) 724-0000
(707)724-0000
1-707-724-0000
1 707-724-0000
1(707)724-0000
```

The regular expression can be deciphered by breaking down its parts. "1?" means zero or one occurrences of "1". "(-|□)?" looks for either a hyphen or a space in the next position, or nothing at all. "\(?" looks for zero or one left parenthesis; the backslash prevents the interpretation of "(" as the grouping metacharacter. "[0-9]+" looks for one or more digits; note that we took the lazy way out and specified one or more digits rather than exactly three. In the next position, we are looking for an optional right parenthesis, and again, either a space or a hyphen, or nothing at all. Then we look for one or more digits "[0-9]+" followed by a hyphen followed by one or more digits "[0-9]+".

7.5.2. Field Splitting: The Full Story

There are three distinct ways you can have awk separate fields. The first method is to have fields separated by whitespace. To do this, set **FS** equal to a single space. In this case, leading and trailing whitespace (spaces and/or tabs) are stripped from the record, and fields are separated by runs of spaces and/or tabs. Since the default value of **FS** is a single space, this is the way awk normally splits each record into fields.

The second method is to have some other single character separate fields. For example, awk programs for processing the UNIX */etc/passwd* file usually use a ":" as the field separator. When **FS** is any single character, *each* occurrence of that character separates another field. If there are two successive occurrences, the field between them simply has the empty string as its value.

Finally, if you specify more than a single character as the field separator, it will be interpreted as a regular expression. That is, the field separator will be the "leftmost longest non-null and nonoverlapping" substring[3] that matches the regular expression. (The phrase "null string" is technical jargon for what we've been calling the "empty string.") You can see the difference between specifying:

[3] *The AWK Programming Language* [Aho], p. 60.

```
FS = "\t"
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

which causes each tab to be interpreted as a field separator, and:

```
FS = "\t+"
```

which specifies that one or more consecutive tabs separate a field. Using the first specification, the following line would have three fields:

```
abc\t\tdef
```

whereas the second specification would only recognize two fields. Using a regular expression allows you to specify several characters to be used as delimiters:

```
FS = "[':\t]"
```

Any of the three characters in brackets will be interpreted as the field separator.

7.6. Expressions

The use of expressions in which you can store, manipulate, and retrieve data is quite different from anything you can do in sed, yet it is a common feature of most programming languages.

An expression is evaluated and returns a value. An expression consists of any combination of numeric and string constants, variables, operators, functions, and regular expressions. We covered regular expressions in detail in Chapter 2, and they are summarized in Appendix B. Functions will be discussed fully in Chapter 9. In this section, we will look at expressions consisting of constants, variables, and operators.

There are two types of constants: string or numeric (**"red"** or **1**). A string must be quoted in an expression. Strings can make use of the escape sequences listed in Table 7.1.

Table 7.1. Escape Sequences

Sequence	Description
\a	Alert character, usually ASCII BEL character
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
ddd	Character represented as 1 to 3 digit octal value
\x*hex*	Character represented as hexadecimal value[4]
c	Any literal character *c* (e.g., \" for ")[5]

[4] POSIX does not provide "\x", but it is commonly available.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

[5] Like ANSI C, POSIX leaves purposely undefined what you get when you put a backslash before any character not listed in the table. In most awks, you just get that character.

A *variable* is an identifier that references a value. To define a variable, you only have to name it and assign it a value. The name can only contain letters, digits, and underscores, and may not start with a digit. Case distinctions in variable names are important: **Salary** and **salary** are two different variables. Variables are not declared; you do not have to tell awk what *type* of value will be stored in a variable. Each variable has a string value and a numeric value, and awk uses the appropriate value based on the context of the expression. (Strings that do not consist of numbers have a numeric value of 0.) Variables do not have to be initialized; awk automatically initializes them to the empty string, which acts like 0 if used as a number. The following expression assigns a value to **x**:

```
x = 1
```

x is the name of the variable, = is an assignment operator, and 1 is a numeric constant.

The following expression assigns the string "Hello" to the variable **z**:

```
z = "Hello"
```

A space is the string concatenation operator. The expression:

```
z = "Hello" "World"
```

concatenates the two strings and assigns "HelloWorld" to the variable **z**.

The dollar sign ($) operator is used to reference fields. The following expression assigns the value of the first field of the current input record to the variable **w**:

```
w = $1
```

A variety of operators can be used in expressions. Arithmetic operators are listed in Table 7.2.

Table 7.2. Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
^	Exponentiation
**	Exponentiation[6]

[6] This is a common extension. It is not in the POSIX standard, and often not in the system documentation, either. Its use is thus nonportable.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Once a variable has been assigned a value, that value can be referenced using the name of the variable. The following expression adds 1 to the value of **x** and assigns it to the variable **y**:

```
y = x + 1
```

So, evaluate **x**, add 1 to it, and put the result into the variable **y**. The statement:

```
print y
```

prints the value of **y**. If the following sequence of statements appears in a script:

```
x = 1
y = x + 1
print y
```

then the value of **y** is 2.

We could reduce these three statements to two:

```
x = 1
print x + 1
```

Notice, however, that after the **print** statement the value of **x** is still 1. We didn't change the value of **x**; we simply added 1 to it and printed that value. In other words, if a third statement **print x** followed, it would output 1. If, in fact, we wished to accumulate the value in **x**, we could use an assignment operator **+=**. This operator combines two operations; it adds 1 to **x** and assigns the new value to **x**. Table 7.3 lists the assignment operators used in awk expressions.

Table 7.3. Assignment Operators

Operator	Description
++	Add 1 to variable.
--	Subtract 1 from variable.
+=	Assign result of addition.
-=	Assign result of subtraction.
*=	Assign result of multiplication.
/=	Assign result of division.
%=	Assign result of modulo.
^=	Assign result of exponentiation.
**=	Assign result of exponentiation.[7]

[7] As with "**", this is a common extension, which is also nonportable.

Look at the following example, which counts each blank line in a file.

```
# Count blank lines.
/^$/ {
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
    print x += 1
     }
```

Although we didn't initialize the value of **x**, we can safely assume that its value is 0 up until the first blank line is encountered. The expression "x += 1" is evaluated each time a blank line is matched and the value of **x** is incremented by 1. The **print** statement prints the value returned by the expression. Because we execute the **print** statement for every blank line, we get a running count of blank lines.

There are different ways to write expressions, some more terse than others. The expression "x += 1" is more concise than the following equivalent expression:

```
x = x + 1
```

But neither of these expressions is as terse as the following expression:

```
++x
```

"++" is the increment operator. ("--" is the decrement operator.) Each time the expression is evaluated the value of the variable is incremented by one. The increment and decrement operators can appear on either side of the operand, as *prefix* or *postfix* operators. The position has a different effect.

```
++x    Increment x before returning value (prefix)
x++    Increment x after returning value (postfix)
```

For instance, if our example was written:

```
/^$/ {
    print x++
     }
```

When the first blank line is matched, the expression returns the value "0"; the second blank line returns "1", and so on. If we put the increment operator before **x**, then the first time the expression is evaluated, it will return "1."

Let's implement that expression in our example. In addition, instead of printing a count each time a blank line is matched, we'll accumulate the count as the value of **x** and print only the total number of blank lines. The **END** pattern is the place to put the **print** that displays the value of **x** after the last input line is read.

```
# Count blank lines.
/^$/ {
    ++x
}
END {
    print x
}
```

Let's try it on the sample file that has three blank lines in it.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
$ awk -f awkscr test
3
```

The script outputs the number of blank lines.

7.6.1. Averaging Student Grades

Let's look at another example, one in which we sum a series of student grades and then calculate the average. Here's what the input file looks like:

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

There are five grades following the student's name. Here is the script that will give us each student's average:

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

This script adds together fields 2 through 6 to get the sum total of the five grades. The value of **total** is divided by 5 and assigned to the variable **avg**. ("/" is the operator for division.) The **print** statement outputs the student's name and average. Note that we could have skipped the assignment of **avg** and instead calculated the average as part of the **print** statement, as follows:

```
print $1, total / 5
```

This script shows how easy it is to write programs in awk. Awk parses the input into fields and records. You are spared having to read individual characters and declaring data types. Awk does this for you, automatically.

Let's see a sample run of the script that calculates student averages:

```
$ awk -f grades.awk grades
john 87.4
andrea 86
jasper 85.6
```

7.7. System Variables

There are a number of system or built-in variables defined by awk. Awk has two types of system variables. The first type defines values whose default can be changed, such as the default field and record separators. The second type defines values that can be used in reports or processing, such as the number of fields found in the current record, the count

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

of the current record, and others. These are *automatically* updated by awk; for example, the current record number and input file name.

There are a set of default values that affect the recognition of records and fields on input and their display on output. The system variable **FS** defines the field separator. By default, its value is a single space, which tells awk that any number of spaces and/or tabs separate fields. **FS** can also be set to any single character, or to a regular expression. Earlier, we changed the field separator to a comma in order to read a list of names and addresses.

The output equivalent of **FS** is **OFS**, which is a space by default. We'll see an example of redefining **OFS** shortly.

Awk defines the variable **NF** to be the number of fields for the current input record. Changing the value of **NF** actually has side effects. The interactions that occur when **$0**, the fields, and **NF** are changed is a murky area, particularly when **NF** is decreased.[8] Increasing it creates new (empty) fields, and rebuilds **$0**, with the fields separated by the value of **OFS**. In the case where **NF** is decreased, gawk and mawk rebuild the record, and the fields that were above the new value of **NF** are set equal to the empty string. The Bell Labs awk does not change **$0**.

[8] Unfortunately, the POSIX standard isn't as helpful here as it should be.

Awk also defines **RS**, the record separator, as a newline. **RS** is a bit unusual; it's the only variable where awk only pays attention to the first character of the value.

The output equivalent to **RS** is **ORS**, which is also a newline by default. In the next section, "Working with Multiline Records," we'll show how to change the default record separator. Awk sets the variable **NR** to the number of the current input record. It can be used to number records in a list. The variable **FILENAME** contains the name of the current input file. The variable **FNR** is useful when multiple input files are used as it provides the number of the current record relative to the current input file.

Typically, the field and record separators are defined in the **BEGIN** procedure because you want these values set before the first input line is read. However, you can redefine these values anywhere in the script. In POSIX awk, assigning a new value to **FS** has no effect on the current input line; it only affects the next input line.

Prior to the June 1996 release of Bell Labs awk, versions of awk for UNIX did not follow the POSIX standard in this regard. In those versions, if you have not yet referenced an individual field, and you set the field separator to a different value, the current input line is split

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

into fields using the new value of **FS**. Thus, you should test how your awk behaves, and if at all possible, upgrade to a correct version of awk.

Finally, POSIX added a new variable, **CONVFMT**, which is used to control number-to-string conversions. For example,

```
str = (5.5 + 3.2) " is a nice value"
```

Here, the result of the numeric expression **5.5 + 3.2** (which is 8.7) must be converted to a string before it can be used in the string concatenation. **CONVFMT** controls this conversion. Its default value is **"%.6g"**, which is a **printf**-style format specification for floating-point numbers. Changing **CONVFMT** to **"%d"**, for instance, would cause all numbers to be converted to strings as integers. Prior to the POSIX standard, awk used **OFMT** for this purpose. **OFMT** does the same job, but controlling the conversion of numeric values when using the **print** statement. The POSIX committee wanted to separate the tasks of output conversion from simple string conversion. Note that numbers that are integers are always converted to strings as integers, no matter what the values of **CONVFMT** and **OFMT** may be.

Now let's look at some examples, beginning with the **NR** variable. Here's a revised **print** statement for the script that calculates student averages:

```
print NR ".", $1, avg
```

Running the revised script produces the following output:

```
1. john 87.4
2. andrea 86
3. jasper 85.6
```

After the last line of input is read, **NR** contains the number of input records that were read. It can be used in the **END** action to provide a report summary. Here's a revised version of the **phonelist.awk** script.

```
# phonelist.awk -- print name and phone number.
# input file -- name, company, street, city, state and zip, phone
BEGIN { FS = ", *" }  # comma-delimited fields
{ print $1 ", " $6 }
END {     print ""
    print NR, "records processed." }
```

This program changes the default field separator and uses **NR** to print the total number of records printed. Note that this program uses a regular expression for the value of **FS**. This program produces the following output:

```
John Robinson, 696-0987
Phyllis Chapman, 879-0900
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
2 records processed.
```

The output field separator (**OFS**) is generated when a comma is used to separate the arguments in a **print** statement. You may have wondered what effect the comma has in the following expression:

```
print NR ".", $1, avg
```

By default, the comma causes a space (the default value of **OFS**) to be output. For instance, you could redefine **OFS** to be a tab in a **BEGIN** action. Then the preceding **print** statement would produce the following output:

```
1.      john    87.4
2.      andrea  86
3.      jasper  85.6
```

This is especially useful if the input consists of tab-separated fields and you want to generate the same kind of output. **OFS** can be redefined to be a sequence of characters, such as a comma followed by a space.

Another commonly used system variable is **NF**, which is set to the number of fields for the current record. As we'll see in the next section, you can use **NF** to check that a record has the same number of fields that you expect. You can also use **NF** to reference the last field of each record. Using the "$" field operator and **NF** produces that reference. If there are six fields, then "$NF" is the same as "$6." Given a list of names, such as the following:

```
John Kennedy
Lyndon B. Johnson
Richard Milhouse Nixon
Gerald R. Ford
Jimmy Carter
Ronald Reagan
George Bush
Bill Clinton
```

you will note that the last name is not the same field number for each record. You could print the last name of each President using "$NF."[9]

[9] This scheme breaks down for Martin Van Buren; fortunately, our list contains only recent U.S. presidents.

These are the basic system variables, the ones most commonly used. There are more of them, as listed in Appendix B, and we'll introduce new system variables as needed in the chapters that follow.

7.7.1. Working with Multiline Records

All of our examples have used input files whose records consisted of a single line. In this section, we show how to read a record where each field consists of a single line.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Earlier, we looked at an example of processing a file of names and addresses. Let's suppose that the same data is stored on file in block format. Instead of having all the information on one line, the person's name is on one line, followed by the company's name on the next line and so on. Here's a sample record:

```
John Robinson
Koren Inc.
978 Commonwealth Ave.
Boston
MA 01760
696-0987
```

This record has six fields. A blank line separates each record.

To process this data, we can specify a multiline record by defining the field separator to be a newline, represented as "\n", and set the record separator to the empty string, which stands for a blank line.

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

We can print the first and last fields using the following script:

```
# block.awk - print first and last fields
# $1 = name; $NF = phone number

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

{ print $1, $NF }
```

Here's a sample run:

```
$ awk -f block.awk phones.block
John Robinson 696-0987
Phyllis Chapman 879-0900
Jeffrey Willis 914-636-0000
Alice Gold (707) 724-0000
Bill Gold 1-707-724-0000
```

The two fields are printed on the same line because the default output separator (**OFS**) remains a single space. If you want the fields to be output on separate lines, change **OFS** to a newline. While you're at it, you probably want to preserve the blank line between records, so you must specify the output record separator **ORS** to be two newlines.

```
OFS = "\n"; ORS = "\n\n"
```

7.7.2. Balance the Checkbook

This is a simple application that processes items in your check register. While not necessarily the easiest way to balance the checkbook, it is amazing how quickly you can build something useful with awk.

This program presumes you have entered in a file the following information:

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
1000
125    Market         -125.45
126    Hardware Store  -34.95
127    Video Store      -7.45
128    Book Store      -14.32
129    Gasoline        -16.10
```

The first line contains the beginning balance. Each of the other lines represent information from a single check: the check number, a description of where it was spent, and the amount of the check. The three fields are separated by tabs. Using negative amounts for checks allows positive amounts to represent deposits.

The core task of the script is that it must get the beginning balance and then deduct the amount of each check from that balance. We can provide detail lines for each check to compare against the check register. Finally, we can print the ending balance. Here it is:

```
# checkbook.awk
BEGIN { FS = "\t" }

#1 Expect the first record to have the starting balance.
NR == 1 { print "Beginning Balance: \t" $1
    balance = $1
    next        # get next record and start over
}

#2 Apply to each check record, adding amount from balance.
{    print $1, $2, $3
    print balance += $3   # checks have negative amounts
}
```

Let's run this program and look at the results:

```
$ awk -f checkbook.awk checkbook.test
Beginning Balance:      1000
125 Market -125.45
874.55
126 Hardware Store -34.95
839.6
127 Video Store -7.45
832.15
128 Book Store -14.32
817.83
129 Gasoline -16.10
801.73
```

The report is difficult to read, but later we will learn to fix the format using the **printf** statement. What's important is to confirm that the script is doing what we want. Notice, also, that getting this far takes only a few minutes in awk. In a programming language such as C, it would take you much longer to write this program; for one thing, you might have many more lines of code; and you'd be programming at a much lower level. There are any number of refinements that you'd want to make to this program to improve it, and refining a program takes much longer. The point is that with awk, you are able to isolate and implement the basic functionality quite easily.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

7.8. Relational and Boolean Operators

Relational and Boolean operators allow you to make comparisons between two expressions. The relational operators are found in Table 7.4.

Table 7.4. Relational Operators

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
~	Matches
!~	Does not match

A relational expression can be used in place of a pattern to control a particular action. For instance, if we wanted to limit the records selected for processing to those that have five fields, we could use the following expression:

```
NF == 5
```

This relational expression compares the value of **NF** (the number of fields for each input record) to five. If it is true, the action will be executed; otherwise, it will not.

Make sure you notice that the relational operator "==" ("is equal to") is not the same as the assignment operator "=" ("equals"). It is a common error to use "=" instead of "==" to test for equality.

We can use a relational expression to validate the *phonelist* database before attempting to print out the record.

```
NF == 6 { print $1, $6 }
```

Then only lines with six fields will be printed.

The opposite of "==" is "!=" ("is not equal to"). Similarly, you can compare one expression to another to see if it is greater than (>) or less than (<) or greater than or equal to (>=) or less than or equal to (<=). The expression

```
NR > 1
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

tests whether the number of the current record is greater than 1. As we'll see in the next chapter, relational expressions are typically used in conditional (**if**) statements and are evaluated to determine whether or not a particular statement should be executed.

Regular expressions are usually written enclosed in slashes. These can be thought of as regular expression *constants*, much as **"hello"** is a string constant. We've seen many examples so far:

```
/^$/ { print "This is a blank line." }
```

However, you are not limited to regular expression constants. When used with the relational operators ~ ("match") and **!**~ ("no match"), the right-hand side of the expression can be any awk expression; awk treats it as a string that specifies a regular expression.[10] We've already seen an example of the ~ operator used in a pattern-matching rule for the phone database:

[10] You may also use strings instead of regular expression constants when calling the **match( )**, **split( )**, **sub( )**, and **gsub( )** functions.

```
$5 ~ /MA/    { print $1 ", " $6 }
```

where the value of field 5 is compared against the regular expression "MA."

Since any expression can be used with ~ and **!**~, regular expressions can be supplied through variables. For instance, in the *phonelist* script, we could replace "/MA/" with **state** and have a procedure that defines the value of state.

```
$5 ~ state   { print $1 ", " $6 }
```

This makes the script much more general to use because *a pattern can change dynamically* during execution of the script. For instance, it allows us to get the value of **state** from a command-line parameter. We will talk about passing command-line parameters into a script later in this chapter.

Boolean operators allow you to combine a series of comparisons. They are listed in Table 7.5.

Table 7.5. Boolean Operators

Operator	Description
||	Logical OR
&&	Logical AND
!	Logical NOT

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Given two or more expressions, || specifies that one of them must evaluate to true (non-zero or non-empty) for the whole expression to be true. **&&** specifies that *both* of the expressions must be true to return true.

The following expression:

```
NF == 6 && NR > 1
```

states that the number of fields must be equal to 6 *and* that the number of the record must be greater than 1.

&& has higher precedence than ||. Can you tell how the following expression will be evaluated?

```
NR > 1 && NF >= 2 || $1 ~ /\t/
```

The parentheses in the next example show which expression would be evaluated first based on the rules of precedence.

```
(NR > 1 && NF >= 2) || $1 ~ /\t/
```

In other words, both of the expressions in parentheses must be true *or* the right hand side must be true. You can use parentheses to override the rules of precedence, as in the following example which specifies that two conditions must be true.

```
NR > 1 && (NF >= 2 || $1 ~ /\t/)
```

The first condition must be true *and* either of two other conditions must be true.

Given an expression that is either true or false, the **!** operator inverts the sense of the expression.

```
! (NR > 1 && NF > 3)
```

This expression is true if the parenthesized expression is false. This operator is most useful with awk's **in** operator to see if an index is not in an array (as we shall see later), although it has other uses as well.

7.8.1. Getting Information About Files

Now we are going to look at a couple of scripts that process the output of a UNIX command, **ls**. The following is a sample of the long listing produced by the command **ls -l**:[11]

[11] Note that on a Berkeley 4.3BSD-derived UNIX system such as Ultrix or SunOS 4.1.x, **ls -l** produces an eight-column report; use **ls -lg** to get the same report format shown here.

```
$ ls -l
-rw-rw-rw-   1 dale     project   6041 Jan  1 12:31 com.tmp
-rwxrwxrwx   1 dale     project   1778 Jan  1 11:55 combine.idx
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
-rw-rw-rw-   1 dale     project   1446 Feb 15 22:32 dang
-rwxrwxrwx   1 dale     project   1202 Jan  2 23:06 format.idx
```

This listing is a report in which data is presented in rows and columns. Each file is presented across a single row. The file listing consists of nine columns. The file's permissions appear in the first column, the size of the file in bytes in the fifth column, and the filename is found in the last column. Because one or more spaces separate the data in columns, we can treat each column as a field.

In our first example, we're going to pipe the output of this command to an awk script that prints selected fields from the file listing. To do this, we'll create a shell script so that we can make the pipe transparent to the user. Thus, the structure of the shell script is:

```
ls -l $* | awk 'script'
```

The $* variable is used by the shell and expands to all arguments passed from the command line. (We could use $1 here, which would pass the first argument, but passing *all* the arguments provides greater flexibility.) These arguments can be the names of files or directories or additional options to the **ls** command. If no arguments are specified, the "$*" will be empty and the current directory will be listed. Thus, the output of the **ls** command will be directed to awk, which will automatically read standard input, since no filenames have been given.

We'd like our awk script to print the size and name of the file. That is, print field 5 ($5) and field 9 ($9).

```
ls -l $* | awk '{
    print $5, "\t", $9
}'
```

If we put the above lines in a file named *fls* and make that file executable, we can enter **fls** as a command.

```
$ fls
6041     com.tmp
1778     combine.idx
1446     dang
1202     format.idx
$ fls com*
6041     com.tmp
1778     combine.idx
```

So what our program does is take the long listing and reduce it to two fields. Now, let's add new functionality to our report by producing some information that the **ls -l** listing does not provide. We add each file's size to a running total, to produce the total number of bytes used by all files in the listing. We can also keep track of the number of files and produce that total. There are two parts to adding this functionality. The first is to accumulate the

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

totals for each input line. We create the variable **sum** to accumulate the size of files and the variable **filenum** to accumulate the number of files in the listing.

```
{
    sum += $5
    ++filenum
    print $5, "\t", $9
}
```

The first expression uses the assignment operator +=. It adds the value of field 5 to the present value of the variable **sum**. The second expression increments the present value of the variable **filenum**. This variable is used as a *counter*, and each time the expression is evaluated, 1 is added to the count.

The action we've written will be applied to all input lines. The totals that are accumulated in this action must be printed after awk has read all the input lines. Therefore, we write an action that is controlled by the **END** rule.

```
END { print "Total: ", sum, "bytes (" filenum " files)" }
```

We can also use the **BEGIN** rule to add column headings to the report.

```
BEGIN { print "BYTES", "\t", "FILE" }
```

Now we can put this script in an executable file named *filesum* and execute it as a single-word command.

```
$ filesum c*
BYTES    FILE
882      ch01
1771     ch03
1987     ch04
6041     com.tmp
1778     combine.idx
Total:  12459 bytes (5 files)
```

What's nice about this command is that it allows you to determine the size of all files in a directory or any group of files.

While the basic mechanism works, there are a few problems to be taken care of. The first problem occurs when you list the entire directory using the **ls -l** command. The listing contains a line that specifies the total number of blocks in the directory. The partial listing (all files beginning with "c") in the previous example does not have this line. But the following line would be included in the output if the full directory was listed:

```
total 555
```

The block total does not interest us because the program displays the total file size in bytes. Currently, **filesum** does not print this line; however, it does read this line and cause the **filenum** counter to be incremented.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

There is also a problem with this script in how it handles subdirectories. Look at the following line from an **ls -l**:

```
drwxrwxrwx   3 dale     project         960 Feb  1 15:47 sed
```

A "d" as the first character in column 1 (file permissions) indicates that the file is a subdirectory. The size of this file (960 bytes) does not indicate the size of files in that subdirectory and therefore, it is slightly misleading to add it to the file size totals. Also, it might be helpful to indicate that it is a directory.

If you want to list the files in subdirectories, supply the *-R* (recursive) option on the command line. It will be passed to the **ls** command. However, the listing is slightly different as it identifies each directory. For instance, to identify the subdirectory *old*, the **ls -lR** listing produces a blank line followed by:

```
./old:
```

Our script ignores that line and a blank line preceding it but nonetheless they increment the file counter. Fortunately, we can devise rules to handle these cases. Let's look at the revised, commented script:

```
ls -l $* | awk '
# filesum: list files and total size in bytes
# input: long listing produced by "ls -l"

#1 output column headers
BEGIN { print "BYTES", "\t", "FILE" }

#2 test for 9 fields; files begin with "-"
NF == 9 && /^-/ {
        sum += $5       # accumulate size of file
        ++filenum       # count number of files
        print $5, "\t", $9       # print size and filename
}

#3 test for 9 fields; directory begins with "d"
NF == 9 && /^d/ {
        print "<dir>", "\t", $9  # print <dir> and name
}

#4 test for ls -lR line ./dir:
$1 ~ /^\..*:$/ {
        print "\t" $0 # print that line preceded by tab
}

#5 once all is done,
END {
    # print total file size and number of files
    print "Total: ", sum, "bytes (" filenum " files)"
}'
```

The rules and their associated actions have been numbered to make it easier to discuss them. The listing produced by **ls -l** contains nine fields for a file. Awk supplies the number of fields for a record in the system variable **NF**. Therefore, rules 2 and 3 test that **NF** is equal to 9. This helps us avoid matching odd blank lines or the line stating the block total. Because we want to handle directories and files differently, we use another pattern to

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

match the first character of the line. In rule 2 we test for "-" in the first position on the line, which indicates a file. The associated action increments the file counter and adds the file size to the previous total. In rule 3, we test for a directory, indicated by "d" as the first character. The associated action prints "<dir>" in place of the file size. Rules 2 and 3 are *compound* expressions, specifying two patterns that are combined using the **&&** operator. Both patterns must be matched for the expression to be true.

Rule 4 tests for the special case produced by the **ls -lR** listing ("./old:"). There are a number of patterns that we can write to match that line, using regular expressions or relational expressions:

```
NF == 1            If the number of fields equals 1 ...
/^\..*:$/        If the line begins with a period followed by any number of
                                                 characters and ends in a colon...
$1 ~ /^\..*:$/        If field 1 matches the regular expression...
```

We used the latter expression because it seems to be the most specific. It employs the match operator (~) to test the first field against a regular expression. The associated action consists of only a **print** statement.

Rule 5 is the **END** pattern and its action is only executed once, printing the sum of file sizes as well as the number of files.

The **filesum** program demonstrates many of the basic constructs used in awk. What's more, it gives you a pretty good idea of the process of developing a program (although syntax errors produced by typos and hasty thinking have been gracefully omitted). If you wish to tinker with this program, you might add a counter for a directories, or a rule that handles symbolic links.

7.9. Formatted Printing

Many of the scripts that we've written so far perform the data processing tasks just fine, but the output has not been formatted properly. That is because there is only so much you can do with the basic **print** statement. And since one of awk's most common functions is to produce reports, it is crucial that we be able to format our reports in an orderly fashion. The **filesum** program performs the arithmetic tasks well but the report lacks an orderly format.

Awk offers an alternative to the **print** statement, **printf**, which is borrowed from the C programming language. The **printf** statement can output a simple string just like the **print** statement.

```
awk 'BEGIN { printf ("Hello, world\n") }'
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

The main difference that you will notice at the outset is that, unlike **print**, **printf** does not automatically supply a newline. You must specify it explicitly as "\n".

The full syntax of the **printf** statement has two parts:

```
printf ( format-expression [, arguments] )
```

The parentheses are optional. The first part is an expression that describes the format specifications; usually this is supplied as a string constant in quotes. The second part is an argument list, such as a list of variable names, that correspond to the format specifications. A format specification is preceded by a percent sign (%) and the specifier is one of the characters shown in Table 7.6. The two main format specifiers are **s** for strings and **d** for decimal integers.[12]

[12] The way **printf** does rounding is discussed in Appendix B.

Table 7.6. Format Specifiers Used in printf

Character	Description
c	ASCII character
d	Decimal integer
i	Decimal integer. (Added in POSIX)
e	Floating-point format ([-]*d.precision***e**[+-]*dd*)
E	Floating-point format ([-]*d.precision***E**[+-]*dd*)
f	Floating-point format ([-]*ddd.precision*)
g	**e** or **f** conversion, whichever is shortest, with trailing zeros removed
G	**E** or **f** conversion, whichever is shortest, with trailing zeros removed
o	Unsigned octal value
s	String
x	Unsigned hexadecimal number. Uses **a-f** for 10 to 15
X	Unsigned hexadecimal number. Uses **A-F** for 10 to 15
%	Literal %

This example uses the **printf** statement to produce the output for rule 2 in the **filesum** program. It outputs a string and a decimal value found in two different fields:

```
printf("%d\t%s\n", $5, $9)
```

The value of **$5** is to be output, followed by a tab (\t) and **$9** and then a newline (\n).[13] For each format specification, you must supply a corresponding argument.

[13] Compare this statement with the **print** statement in the **filesum** program that prints the header line. The **print** statement automatically supplies a newline (the value of **ORS**); when using **printf**, you must supply the newline, it is never automatically provided for you.

This **printf** statement can be used to specify the width and alignment of output fields. A format expression can take three optional modifiers following "%" and preceding the format specifier:

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
%- width . precision format-specifier
```

The *width* of the output field is a numeric value. When you specify a field width, the contents of the field will be right-justified by default. You must specify "-" to get left-justification. Thus, "%-20s" outputs a string left-justified in a field 20 characters wide. If the string is less than 20 characters, the field will be padded with whitespace to fill. In the following examples, a "|" is output to indicate the actual width of the field. The first example right-justifies the text:

```
printf("|%10s|\n", "hello")
```

It produces:

```
|     hello|
```

The next example left-justifies the text:

```
printf("|%-10s|\n", "hello")
```

It produces:

```
|hello     |
```

The *precision* modifier, used for decimal or floating-point values, controls the number of digits that appear to the right of the decimal point. For string values, it controls the maximum number of characters from the string that will be printed. Note that the default precision for the output of numeric values is "%.6g".

You can specify both the *width* and *precision* dynamically, via values in the **printf** or **sprintf** argument list. You do this by specifying asterisks, instead of literal values.

```
printf("%*.*g\n", 5, 3, myvar);
```

In this example, the width is 5, the precision is 3, and the value to print will come from **myvar**.

The default precision used by the **print** statement when outputting numbers can be changed by setting the system variable **OFMT**. For instance, if you are using awk to write reports that contain dollar values, you might prefer to change **OFMT** to "%.2f".

Using the full syntax of the format expression can solve the problem with **filesum** of getting fields and headings properly aligned. One reason we output the file size before the filename was that the fields had a greater chance of aligning themselves if they were output in that order. The solution that **printf** offers us is the ability to fix the width of output fields; therefore, each field begins in the same column.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

Let's rearrange the output fields in the **filesum** report. We want a minimum field width so that the second field begins at the same position. You specify the field width place between the **%** and the conversion specification. "%-15s" specifies a minimum field width of 15 characters in which the value is left-justified. "%10d", without the hyphen, is right-justified, which is what we want for a decimal value.

```
printf("%-15s\t%10d\n", $9, $5)       # print filename and size
```

This will produce a report in which the data is aligned in columns and the numbers are right-justified. Look at how the **printf** statement is used in the **END** action:

```
printf("Total: %d bytes  (%d files)\n", sum, filenum)
```

The column header in the **BEGIN** rule is also changed appropriately. With the use of the **printf** statement, **filesum** now produces the following output:

```
$ filesum g*
FILE                  BYTES
g                        23
gawk                   2237
gawk.mail              1171
gawk.test                74
gawkro                  264
gfilesum                610
grades                   64
grades.awk              231
grepscript                6
Total: 4680 bytes  (9 files)
```

7.10. Passing Parameters Into a Script

One of the more confusing subtleties of programming in awk is passing parameters into a script. A parameter assigns a value to a variable that can be accessed within the awk script. The variable can be set on the command line, after the script and before the filename.

```
awk 'script' var=value inputfile
```

Each parameter must be interpreted as a single argument. Therefore, spaces are not permitted on either side of the equal sign. Multiple parameters can be passed this way. For instance, if you wanted to define the variables **high** and **low** from the command line, you could invoke awk as follows:

```
$ awk -f scriptfile high=100 low=60 datafile
```

Inside the script, these two variables are available and can be accessed as any awk variable. If you were to put this script in a shell script wrapper, then you could pass the shell's command-line arguments as values. (The shell makes available command-line arguments in the positional variables—$1 for the first parameter, $2 for the second, and so on.)[14] For instance, look at the shell script version of the previous command:

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

[14] Careful! Don't confuse the shell's parameters with awk's field variables.

```
awk -f scriptfile "high=$1" "low=$2" datafile
```

If this shell script were named **awket**, it could be invoked as:

```
$ awket 100 60
```

"100" would be $1 and passed as the value assigned to the variable **high**.

In addition, environment variables or the output of a command can be passed as the value of a variable. Here are two examples:

```
awk '{ ... }' directory=$cwd file1 ...
awk '{ ... }' directory=`pwd` file1 ...
```

"$cwd" returns the value of the variable **cwd**, the current working directory (**csh** only). The second example uses backquotes to execute the **pwd** command and assign its result to the variable **directory** (this is more portable).

You can also use command-line parameters to define system variables, as in the following example:

```
$ awk '{ print NR, $0 }' OFS='. ' names
1. Tom 656-5789
2. Dale 653-2133
3. Mary 543-1122
4. Joe 543-2211
```

The output field separator is redefined to be a period followed by a space.

An important restriction on command-line parameters is that they are not available in the **BEGIN** procedure. That is, they are not available until *after* the first line of input is read. Why? Well, here's the confusing part. A parameter passed from the command line is treated as though it were a filename. The assignment does not occur until the parameter, if it were a filename, is actually evaluated.

Look at the following script that sets a variable *n* as a command-line parameter.

```
awk  'BEGIN { print n }
{
if (n == 1) print "Reading the first file"
if (n == 2) print "Reading the second file"
}' n=1 test n=2 test2
```

There are four command-line parameters: "n=1," "test," "n=2," and "test2". Now, if you remember that a **BEGIN** procedure is "what we do before processing input," you'll understand why the reference to *n* in the **BEGIN** procedure returns nothing. So the **print** statement will print a blank line. If the first parameter were a file and not a variable assignment, the file would not be opened until the **BEGIN** procedure had been executed.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

The variable *n* is given an initial value of 1 from the first parameter. The second parameter supplies the name of the file. Thus, for each line in *test*, the conditional "n == 1" will be true. After the input is exhausted from *test*, the third parameter is evaluated, and it sets **n** to 2. Finally, the fourth parameter supplies the name of a second file. Now the conditional "n == 2" in the main procedure will be true.

One consequence of the way parameters are evaluated is that you cannot use the **BEGIN** procedure to test or verify parameters that are supplied on the command line. They are available only after a line of input has been read. You can get around this limitation by composing the rule "NR == 1" and using its procedure to verify the assignment. Another way is to test the command-line parameters in the shell script before invoking awk.

POSIX awk provides a solution to the problem of defining parameters before any input is read. The *-v* option[15] specifies variable assignments that you want to take place before executing the **BEGIN** procedure (i.e., before the first line of input is read.) The *-v* option must be specified before a command-line script. For instance, the following command uses the *-v* option to set the record separator for multiline records.

[15] The *-v* option was not part of the original (1987) version of nawk (still used on SunOS 4.1.x systems and some System V Release 3.x systems). It was added in 1989 after Brian Kernighan of Bell Labs, the GNU awk authors, and the authors of MKS awk agreed on a way to set variables on the command line that would be available inside the **BEGIN** block. It is now part of the POSIX specification for awk.

```
$ awk -F"\n" -v RS="" '{ print }' phones.block
```

A separate *-v* option is required for each variable assignment that is passed to the program.

Awk also provides the system variables **ARGC** and **ARGV**, which will be familiar to C programmers. Because this requires an understanding of arrays, we will discuss this feature in Chapter 8.

7.11. Information Retrieval

An awk program can be used to retrieve information from a database, the database basically being any kind of text file. The more structured the text file, the easier it is to work with, although the structure might be no more than a line consisting of individual words.

The list of acronyms below is a simple database.

```
$ cat acronyms
BASIC    Beginner's All-Purpose Symbolic Instruction Code
CICS    Customer Information Control System
COBOL    Common Business Oriented Language
DBMS    Data Base Management System
GIGO    Garbage In, Garbage Out
GIRL     Generalized Information Retrieval Language
```

A tab is used as the field separator. We're going to look at a program that takes an acronym as input and displays the appropriate line from the database as output. (In the next chapter,

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

we're going to look at two other programs that use the acronym database. One program reads the list of acronyms and then finds occurrences of these acronyms in another file. The other program locates the first occurrence of these acronyms in a text file and inserts the description of the acronym.)

The shell script that we develop is named **acro**. It takes the first argument from the command line (the name of the acronym) and passes it to the awk script. The **acro** script follows:

```
$ cat acro
#! /bin/sh
# assign shell's $1 to awk search variable
awk '$1 == search' search=$1 acronyms
```

The first argument specified on the shell command line ($1) is assigned to the variable named **search**; this variable is passed as a parameter into the awk program. Parameters passed to an awk program are specified *after* the script section. (This gets somewhat 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.)

The example below demonstrates how this program can be used to find a particular acronym on our list.

```
$ acro CICS
CICS Customer Information Control System
```

Notice that we tested the parameter as a string (**$1 == search**). We could also have written this as a regular expression match (**$1 ~ search**).

7.11.1. Finding a Glitch

A net posting was once forwarded to one of us because it contained a problem that could be solved using awk. Here's the original posting by Emmett Hogan:

```
I have been trying to rewrite a sed/tr/fgrep script that we use quite
a bit here in Perl, but have thus far been unsuccessful...hence this
posting.  Having never written anything in perl, and not wishing to
wait for the Nutshell Perl Book, I figured I'd tap the knowledge of this
group.

Basically, we have several files which have the format:

item    info line 1
    info line 2
      .
      .
      .
    info line n

Where each info line refers to the item and is indented by either
spaces or tabs.  Each item "block" is separated by a blank line.

What I need to do, is to be able to type:

info glitch filename
```


sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01

```
Where info is the name of the perl script, glitch is what I want to
find out about, and filename is the name of the file with the
information in it.  The catch is that I need it to print the entire
"block" if it finds glitch anywhere in the file, i.e.:

machine         Sun 3/75
        8 meg memory
        Prone to memory glitches
        more info
        more info

would get printed if you looked for "glitch" along with any other
"blocks" which contained the word glitch.

Currently we are using the following script:

#!/bin/csh -f
#
sed '/^ /\!s/^/@/' $2 | tr '\012@' '@\012' | fgrep -i $1 | tr '@' '\012'

Which is in a word....SLOW.

I am sure Perl can do it faster, better, etc...but I cannot figure it out.

Any, and all, help is greatly appreciated.


Thanks in advance,
Emmett

-------------------------------------------------------------------
Emmett Hogan              Computer Science Lab, SRI International
```

The problem yielded a solution based on awk. You may want to try to tackle the problem yourself before reading any further. The solution relies on awk's multiline record capability and requires that you be able to pass the search string as a command-line parameter.

Here's the **info** script using awk: [16]

[16] Remember that you need an **awk** that provides POSIX semantics for this to work. It may be named **awk**, **nawk**, or even something else! Check your local system documentation.

```
awk 'BEGIN { FS = "\n"; RS = "" }
$0 ~ search { print $0 }' search=$1 $2
```

Given a test file with multiple entries, **info** was tested to see if it could find the word "glitch."

```
$ info glitch glitch.test
machine         Sun 3/75
        8 meg memory
        Prone to memory glitches
        more info
        more info
```

In the next chapter, we look at conditional and looping constructs, and arrays.

sed & awk, 2nd Edition By Arnold Robbins, Dale Dougherty ISBN: 1-56592-225-5
Publisher: O'Reilly Media, Inc.
Print Publication Date: 1997/03/01