Introduction to the awk Language

awk is a general-purpose language for processing text. With **awk**, you can manipulate strings, process records, and generate reports.

awk is named after its creators: A. V. Aho, P. J. Weinberger, and Brian W. Kernighan. Unfortunately, its name suggests that **awk** is awkward — whereas in truth the **awk** language is simple, elegant, and powerful. With it, you can perform many tasks that would otherwise require hours of drudgery.

awk uses a simple syntax. Each statement in an **awk** program contains either or both of two elements: a *pattern* and an *action*. The pattern tells **awk** what lines to select from the input stream; and the action tells **awk** what to do with the selected data.

This tutorial explains how to write **awk** programs. It explains how to describe a pattern to **awk**. It also describes the range of actions that **awk** can perform; these include formatted printing of text, assigning variables, defining arrays, and controlling the flow of data.

Example Files

Before you begin to study **awk**, please take the time to type the following text files that are used by the examples in this tutorial.

The first is some text from Shakespeare. Use the command **cat** to type it into the file **text1**, as follows. Note that **<ctrl-D>** means that you should hold down the **Ctrl** (or **control**) key and simultaneously press ‘D’. Do not type it literally.

```
cat > text1
When, in disgrace with fortune and men’s eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possest,
Desiring this man’s art and that man’s scope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Haply I think on thee - and then my state,
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven’s gate;
For thy sweet love remember’d such wealth brings
That then I scorn to change my state with kings.
<ctrl-D>
```

The second example consists of some of Babe Ruth’s batting statistics, which we will use to demonstrate how **awk** processes tabular input. Type it into file **table1**, as follows:

```
cat > table1
1920    .376   54     158    137
1921    .378   59     177    171
1922    .315   35     94     99
1923    .393   41     151    131
1924    .378   46     143    121
1925    .290   25     61     66
1926    .372   47     139    145
1927    .356   60     158    164
1928    .323   54     163    142
1929    .345   46     121    154
<ctrl-D>
```

The columns give, respectively, the season, the batting average, and the numbers of home runs, runs scored, and runs batted in (RBIs).

The rest of this tutorial presents many examples that use these files. Type them in and run them! In that way, you can get a feel for **awk** into your fingers. Experiment; try some variations on the examples. Don’t be afraid of making mistakes; this is one good way to learn the limits (and the strengths) of a language.

Using awk

awk reads input from the standard input (entered from your terminal or from a file you specify), processes each input line according to a specified **awk** program, and writes output to the standard output. This section explains the structure of an **awk** program and the syntax of **awk** command lines.

Command-line Options

The complete form for the **awk** command line is as follows:

awk [**-y**] [**-F***c*] [**-f** *progfile*] [*prog*] [*file1*] [*file2*] ...

The following describes each element of the command line.

-y Map *patterns* from lower case to both lower-case and upper-case letters. For example, with this option the string **the** would match **the** or **The**.

-F*c* Set the field-separator character to the character *c*. The field-separator character and its uses are described below.

-f *progfile*
Read the **awk** program from *progfile*

prog An **awk** program to execute. If you do not use the **-f** option, you must enter **awk**'s statements on its command line.

Note that if you include **awk**'s program on its command line (instead of in a separate file), you must enclose the program between apostrophes. Otherwise, some of the **awk** statements will be modified by the shell before **awk** ever sees them, which will make a mess of your program. For example:

```
awk 'BEGIN {print "sample output file"}
           {print NR, $0}'
```

(The following sections explain what the stuff between the apostrophes means.) However, if you include the statement within a file that you pass to **awk** via its **-f** option, you must *not* enclose the statements within parentheses; otherwise, **awk** will become very confused. If you were to put the statements in the above program into an **awk** program file, they would appear as follows:

```
BEGIN {print "sample output file"}
 {print NR, $0}
```

file1 file2 ...
The files whose text you wish to process. For example, the command

```
awk '{print NR, $0}' text1
```

prints the contents of **text1**, but precedes each line with a line number.

If you do not name an input file, **awk** processes what it reads from the standard input. For example, the command

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

reads what you type from the keyboard and echoes it preceded with a line number. To exit from this program, type **<ctrl-D>**.

Structure of an awk Program

An **awk** program consists of one or more statements of the form:

pattern { *action* }

Note that **awk** insists that the action be enclosed between braces, so that it can distinguish the action from the pattern.

A program can contain as many statements as you need to accomplish your purposes. When **awk** reads a line of input, it compares that line with the *pattern* in each statement. Each time a line matches *pattern*, **awk** performs the corresponding *action*. **awk** then reads the next line of input.

A statement can specify an *action* without a *pattern*. In this case, **awk** performs the action on every line of input. For example, the program

```
awk '{ print }' text1
```

prints every line of **text1** onto the standard output.

An **awk** program may also specify a pattern without an action. In this case, when an input line matches the pattern, **awk** prints it on the standard output. For example, the command

```
awk 'NR > 0' table1
```

prints all of **table1** onto the standard output. Note that you can use the same pattern in more than one statement. Examples of this will be given below.

awk's method of forming patterns uses *regular expressions* (also called *patterns*), like those used by the COHERENT commands **sed**, **ed**, and **egrep**. Likewise, **awk**'s method of constructing actions is modelled after the C programming language. If you are familiar with regular expressions and with C, you should have no problem learning how to use **awk**. However, if you are not familiar with them, they will be explained in the following sections.

Records and Fields

awk divides its input into *records*. It divides each record, in turn, into *fields*. Records are separated by a character called the *input-record separator*; likewise, fields are separated by the *input-field separator*. **awk** in effect conceives of its input as a table with an indefinite number of columns.

The newline character is the default input-field separator, so **awk** normally regards each input line as a separate record. The space and the tab characters are the default input-field separator, so white space normally separates fields.

To address a field within a record, use the syntax **$***N*, where *N* is the number of the field within the current record. The pattern **$0** addresses the entire record. Examples of this will be given below. In addition to input record and field separators, **awk** provides output record and field separators, which it prints between output records and fields. The default output-field separator is the newline character; **awk** normally prints each output record as a separate line. The space character is the default output-field separator.

Patterns

This section describes how **awk** interprets the pattern section of a statement.

Special Patterns

To begin, **awk** defines and sets a number of special *pattern*s. You can use these patterns in your program for special purposes. You can also redefine some of these patterns to suit your preferences. The following describes the commonest such special *patterns*, and how they're used:

BEGIN This pattern matches the beginning of the input file. **awk** executes all *actions* associated with this pattern before it begins to read input.

END This pattern matches the end of the input file. **awk** executes all *actions* associated with this pattern after it had read all of its input.

FILENAME
awk sets this pattern to the name of the file that it is currently reading. Should you name more than one input file on the command line, **awk** resets this pattern as it reads each file in turn.

FS Input-field separator. This pattern names the character that **awk** recognizes as the field separator for the records it reads.

NF This pattern gives the number of fields within the current record.

NR This pattern gives the number of the current record within the input stream.

OFS Output-field separator. **awk** sets this pattern to the character that it writes in its output to separate one field from another.

ORS Output-record separator. **awk** sets this pattern to the character that it writes in its output to separate one field from another.

RS Input-record separator. **awk** sets this pattern to the character by which it separates records that it reads.

Arithmetic Relational Expressions

An *operator* marks a task to be within an expression, much as the '+' or '/' within an arithmetic expression indicates that the numbers are to be, respectively, added or divided. You can use **awk**'s operators to:

- Compare a special pattern with a variable, a field, or a constant.
- Assign a value to a variable or to a special pattern.
- Dictate the relationship among two or more expressions.

The first type of operator to be discussed are *arithmetic relational operators*. These compare the input text with an arithmetic value. **awk** recognizes the following arithmetic operators:

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

With these operators, you can compare a field with a constant; compare one field with another; or compare a special pattern with either a field or a constant.

For example, the following **awk** program prints all of the years in which Babe Ruth hit more than 50 home runs:

```
awk '$3 >= 50' table1
```

(As you recall, column 3 in the file **table1** gives the number of home runs.) The program prints the following on your screen:

```
1920    .376    54      158     137
1921    .378    59      177     171
1927    .356    60      158     164
1928    .323    54      163     142
```

The following program, however, shows the years in which Babe Ruth scored more runs than he drove in:

```
awk '$4 > $5 { print $1 }' table1
```

Remember, field 4 in file **table1** gives the number of runs scored, and field 5 gives the number of runs batted in. You should see the following on your screen:

```
1920
1921
1923
1924
1928
```

In the above program, expression

```
{print $1}
```

defines the action to perform, as noted by the fact that expression is enclosed between braces. In this case, the program tells **awk** that if the input record matches the pattern, to print only the first field. However, to print both the season and the number of runs scored, use the following program:

```
awk '$4 > $5 { print $1, $4 }' table1
```

This prints the following:

```
1920 158
1921 177
1923 151
1924 143
1928 163
```

Note that **$1** and **$4** are separated by a comma. The comma tells **awk** to print its default output-field separator between columns. If we had left out the comma, the output would have appeared as follows:

```
1920158
1921177
1923151
1924143
1928163
```

As we noted above, the special pattern **OFS** gives the output-field separator. **awk** by default defines this special pattern to the space character. If we wish to redefine the output-field separator, we can use an operator, plus the special pattern **BEGIN**, as follows:

```
awk 'BEGIN { OFS = ":" }
       $4 > $5 { print $1, $4 }' table1
```

This prints:

```
1920:158
1921:177
1923:151
1924:143
1928:163
```

The first statement

```
BEGIN { OFS = ":"}
```

tells **awk** to set the output-field separator (the special pattern **OFS**) to ':' before it processes any input (as indicated by the special pattern **BEGIN**).

Although we're getting a little ahead of ourselves, note that there's no reason to print the fields in the order in which they appear in the input record. For example, if you wish to print the number of runs scored before the season, use the command:

```
awk 'BEGIN { OFS = ":"}
       $4 > $5 { print $4, $1 }' table1
```

This prints:

```
158:1920
177:1921
151:1923
143:1924
163:1928
```

As you recall, the special pattern **NR** gives the number of the current input record. You can execute an action by comparing this pattern with a constant. For example, the command

```
awk 'NR > 12' text1
```

prints:

```
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

That is, the program prints every line after line 12 in the input file. As you recall, a statement that has a pattern but no action prints the entire record that matches the pattern.

As we saw with the special patterns, some patterns can be defined to be numbers and others to be text. If you compare a number with a string, **awk** by default makes a string comparison. The following example shows how **awk** compares one field to part of the alphabet:

```
awk '$1 <= "C"' text1
```

This program prints:

```
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
```

The statement **$1 <= "C"** selected all records that begin with an ASCII value less than or equal to that of the letter 'C' (0x43) — in this case, both lines that begin with 'A' (0x41). If we ran this example against **table1**, it would print every record in the file. This is because each record begins with the character '1' (0x31), which matches the pattern **$1 <= "C"**.

Finally, you can use a numeric field plus a constant in a comparison statement. For example, the following program prints all of the seasons in which Babe Ruth had at least 100 more runs batted in than home runs:

```
awk '$3 + 100 < $5 {print $1}' table1
```

This prints the following:

```
1921
1927
1929
```

Boolean Combinations of Expressions

awk has a number of operators, called *Boolean* operators, that let you hook together several small expressions into one large, complex expression. **awk** recognizes the following Boolean operators:

- **||** Boolean OR (one expression or the other is true)
- **&&** Boolean AND (both expressions are true)
- **!** Boolean NOT (invert the value of an expression)

(The eponym "Boolean" comes from the English mathematician George Boole.) In a Boolean expression, **awk** evaluates each sub-expression to see if it is true or false; the relationship of sub-expressions (as set by the Boolean operator) then determines whether the entire expression is true or false.

For example, the following program prints all seasons in which Babe Ruth hit between 40 and 50 home runs:

```
awk '$3 >= 40 && $3 <= 50 { print $1, $3 }' table1
```

This prints the following:

```
1923 41
1924 46
1926 47
1929 46
```

In the above program, **awk** printed its output only if the subexpression **$3 >= 40** was true *and* (**&&**) the subexpression **$3 <= 50** was true.

The next example demonstrates the Boolean OR operator. It prints all seasons for which Babe Ruth hit fewer than 40 home runs or more than 50 home runs:

```
awk '$3 < 40 || $3 > 50 { print $1, $3}' table1
```

This example prints the following:

```
1920 54
1921 59
1922 35
1925 25
1927 60
1928 54
```

In this example, **awk** printed its output if the subexpression **$3 < 40** was true *or* (**||**) the subexpression **$3 > 50** was true. Note that the output would also printed if both subexpressions were true (although in this case, this is impossible).

Finally, the Boolean operator '!' negates the truth-value of any expression. For example, the expression **$1 = "And"** is true if the first field in the current record equals "And"; however, the expression **$1 != "And"** is true if the first field does *not* equal "And". For example, the program

```
awk '$1 != "And"' text1
```

prints:

```
When, in disgrace with fortune and men’s eyes,
I all alone beweep my outcast state,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possest,
Desiring this man’s art and that man’s scope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Haply I think on thee - and then my state,
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven’s gate;
For thy sweet love remember’d such wealth brings
That then I scorn to change my state with kings.
```

These are the 12 lines from **text1** that do not begin with “And”.

Note that **awk** evaluates all operators from left to right unless sub-expressions are grouped together with parentheses, as is described in the following section.

Patterns

The previous examples have all matched strings or numbers against predefined fields in each input record. This is fine for manipulating tabular information, like our table of Babe Ruth’s batting statistics, but it is not terribly useful when you are processing free text. Free text is not organized into predefined columns, nor are you likely to know which field (that is, which word) will contain the pattern you’re seeking.

To help you manage free text, **awk** has a pattern-matching facility that resembles those of the editors **ed** and **sed**.

The most common way to search for a pattern is to enclose it between slashes. For example, the program

```
awk ’/and/’ text1
```

prints every line in **text1** that contains the string “and”.

```
When, in disgrace with fortune and men’s eyes,
And look upon myself, and curse my fate,
Desiring this man’s art and that man’s scope,
Haply I think on thee - and then my state,
```

Note that “and” does not have to be a word by itself — it can be a fragment within a word as well. Note, too, that this pattern matches “and” but does not match “And” — but it would if we were to use the **-y** option on the **awk** command line (described above).

You can use Boolean operators to search for more than one string at once. For example, the program

```
awk ’/and/ && /or/’ text1
```

finds every line in **text1** that contains both “and” and “or”. There is only one:

```
When, in disgrace with fortune and men’s eyes,
```

Note that the “or” in this line is embedded in the word “fortune”.

awk can also scan for classes and types of characters. To do so, enclose the characters within brackets and place the bracketed characters between the slashes. For example, the following program looks for every line in **text1** that contains a capital ‘A’ through a capital ‘E’:

```
awk ’/[A-E]/’ text1
```

This prints the following:

```
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
Desiring this man’s art and that man’s scope,
```

In addition, you can use the following special characters for further flexibility:

[ ]	Class of characters
( )	Grouping subexpressions
|	Alternatives among expressions
+	One or more occurrences of the expression
?	Zero or more occurrences of the expression
*****	Zero, one, or more occurrences of the expression
.	Any non-newline character

When adding a special character to a pattern, enclose the special character as well as the rest of the pattern within slashes.

To search for a string that contains one of the special characters, you must precede the character with a backslash. For example, if you are looking for the string "today?", use the following pattern:

```
/today\?/
```

When you need to find an expression in a particular field, not just anywhere in the record, you can use one of these operators:

~	Contains the data in question
!~	Does not contain the data in question

For example, if you need to find the digit '9' in the fourth field of file **table1**, use the following program:

```
awk '$4~/9/ {print $1, $4}' table1
```

This prints the following:

```
1922 94
1926 139
```

As you can see, the above program found every record with a '9' in its fourth field, regardless of whether the '9' came at the beginning of the field or its end. **awk** also recognizes two operators that let you set where a pattern is within a field:

^	Beginning of the record or field
$	End of the record or field

For example, to find every record in **table1** whose fourth field *begins* with a '9', run the following program:

```
awk '$4~/^9/ {print $1, $4}' table1
```

This prints:

```
1922 94
```

Finally, to negate a pattern use the operator **!~**. For example, to print every record in **table1** whose fourth column does *not* begin with a '9', use the following program:

```
awk '$4!~/^9/ {print $1, $4}' table1
```

This prints:

```
1920 158
1921 177
1923 151
1924 143
1925 61
1926 139
1927 158
1928 163
1929 121
```

Ranges of Patterns

You can tell **awk** to perform an action on all records between two patterns. For example, to print all records between the *patterns* **1925** and **1929**, inclusive, enclose the strings in slashes and separate them with a comma, then indicate the **print** action, as follows:

```
awk '/1925/,/1929/ { print }' table1
```

You can also use the special pattern **NR** (or *record number*) to name a range of record numbers. For example, to print records 5 through 10 of file **text1**, use the following program:

```
awk 'NR == 5, NR == 10 { print }' text1
```

Resetting Separators

As noted above, **awk** recognizes certain characters by default to parse its input into records and fields, and to separate its output into records and fields:

FS Input-field separator. By default, this is one or more white-space characters (tabs or spaces).

OFS Output-field separator. By default, this is exactly one space character.

ORS Output-record separator. By default, this is the newline character.

RS Input-record separator. By default, this is the newline character.

By resetting any of these special patterns, you can change how **awk** parses its input or organizes its output. Consider, for example, the command:

```
awk 'BEGIN {ORS = "|"}
        /1920/,/1925/ {print $1, $5}' table1
```

This prints the following:

```
1920 137|1921 171|1922 99|1923 131|1924 121|1925 66|
```

As you can see, this prints the season and the number of runs batted in for the 1920 through 1925 season. However, **awk** uses the pipe character '|' instead of the newline character to separate records. If you wish to change the output-field separator as well as the output-record separator, use the program:

```
awk 'BEGIN {ORS = "|" ;       OFS = ":"}
        /1920/,/1925/ {print $1, $5}' table1
```

This produces:

```
1920:137|1921:171|1922:99|1923:131|1924:121|1925:66|
```

As you can see, **awk** has used the colon ':' instead of a white-space character to separate one field from another.

Note, too, that the semicolon ';' character separates expressions in the action portion of the statement associated with the **BEGIN** pattern. This lets you associate more than one action with a given pattern, so you do not have to repeat that pattern. This is discussed at greater length below.

You can also change the input-record separator from the newline character to something else that you prefer. For example, the following program changes the input-record separator from the newline to the comma:

```
awk 'BEGIN {RS = ","}
        {print $0}' text1
```

This yields the following:

```
When
 in disgrace with fortune and men's eyes

I all alone beweep my outcast state

And trouble deaf heaven with my bootless cries

And look upon myself
 and curse my fate

Wishing me like to one more rich in hope

Featured like him
 like him with friends possest

Desiring this man's art and that man's scope

With what I most enjoy contented least.
Yet in these thoughts myself almost despising

Haply I think on thee - and then my state

Like to the lark at break of day arising
From sullen earth
 sings hymns at heaven's gate;
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

The blank lines resulted from a comma's occurring at the end of a line.

Note that by specifying the null string **(RS="")**, you can make two consecutive newlines the record separator. Note, too, that only one character can be the input-record separator. If you try to reset this separator to a string, **awk** uses the first character in the string as the separator, and ignores the rest.

You can change the input-field separator by redefining **FS**. The default **FS** is **<space>\t** exactly and in that order (where **<space>** is the space character). In this case, **awk** uses its "white-space rule," in which **awk** treats any sequence of spaces and tabs as a single separator. This is the default rule for **FS**. If you set **FS** to anything else, including **\t<space>**, then each separator is separate. For example, the following program changes the input-field separator to the comma and prints the first such field it finds in each line from file **text1**:

```
awk 'BEGIN {FS = ","}
       {print $1}' text1
```

This produces:

```
When
I all alone beweep my outcast state
And trouble deaf heaven with my bootless cries
And look upon myself
Wishing me like to one more rich in hope
Featured like him
Desiring this man's art and that man's scope
With what I most enjoy contented least.
Yet in these thoughts myself almost despising
Haply I think on thee - and then my state
Like to the lark at break of day arising
From sullen earth
For thy sweet love remember'd such wealth brings
That then I scorn to change my state with kings.
```

As you can see, this program prints text up to the first comma in each line. **awk** throws away the comma itself, because the input-field separator is not explicitly printed.

You can define several characters to be input-field separators simultaneously. When you specify several characters within quotation marks, each character becomes a field separator, and all separators have equal precedence. For example, you can specify the letters 'i', 'j', and 'k' to be input-field separators. The following program does this, and prints the first field so defined from each record in file **text1**:

```
awk 'BEGIN  {FS = "ijk"}
       {print $1}' text1
```

This prints:

```
When,
I all alone beweep my outcast state,
And trouble deaf heaven w
And loo
W
Featured l
Des
W
Yet
Haply I th
L
From sullen earth, s
For thy sweet love remember'd such wealth br
That then I scorn to change my state w
```

Note that if you set the input-record separator to a null string, you can use the newline character as the input-field separator. This is a handy way to concatenate clusters of lines into records that you can then manipulate further.

One last point about the **FS** separator. If the white-space rule is not invoked and an assignment is made to a nonexistent field, **awk** can add the proper number of field separators. For example if **FS=":"** and the input line is **a:b**, then the command **$5 = "e"** produces **a:b:::e**. If the white-space rule were in effect, **awk** would add spaces as if each space were a separator, and print a warning message. In short, it would try to produce the sanest result from the error.

Finally, the variable **NR** gives the number of the current record. The next example prints the total number of records in file **text1**:

```
awk 'END {print NR}' text1
```

The output is

```
14
```

which is to be expected, since **text1** is a sonnet.

Actions

The previous section described how to construct a *pattern* for **awk**. For each pattern, there must be a corresponding *action*. So far, the only action shown has been to print output. However, **awk** can perform many varieties of actions. In addition to printing, **awk** can:

- Execute built-in functions
- Redirect output
- Assign variables
- Use fields as variables
- Define arrays
- Use control statements

These actions are discussed in detail in the following sections.

As noted above, each **awk** statement must have an action. If a statement does not include an action, **awk** assumes that the action is **{print}**.

Within each statement, **awk** distinguishes an action from its corresponding pattern by the fact that the action is enclosed within braces. Note that the action section of a statement may include several individual actions; however, each action must be separated from the others by semicolons ';' or newlines.

Some forms of **awk**, such as that provided by the Free Software Foundation (FSF), allow user-defined functions. The FSF version of **awk** is available from the MWC BBS as well as via COHware. Note that your system must have at least two megabytes of RAM to run the FSF version of **awk**.

awk Functions

awk includes the following functions with which you can manipulate input. You can assign a function to any variable or use it in a pattern. The following lists **awk**'s functions. Note that an *argument* can be a variable, a field, a constant, or an expression:

abs(*argument***)**
Return the absolute value of *argument*.

exp(*argument***)**
Return Euler's number *e* (2.178...) to the power of *argument*.

index(*string1***,***string2***)**
Return the position of *string2* within *string1*. If *s2* does not occur in *s1*, **awk** returns zero. This **awk** function resembles the COHERENT C function **index()**.

int(*argument***)**
Return the integer portion of *argument*.

length Return the length, in bytes, of the current record.

length(*argument***)**
Return the length, in bytes, of *argument*.

log(*argument***)**
Return the natural logarithm of *argument*.

print(*argument1 argument2 ... argumentN***)**
Concatenate and print *argument1* through *argumentN*.

print(*argument1***,***argument2***,** *... argumentN***)**
Print *argument1* through *argumentN*. Separate each *argument* with the **OFS** character.

printf(*f***,** *argument1***,** *... argumentN***)**
Format and print strings *argument1* through *argument* in the manner set by the formatting string *f*, which can use **printf()**-style formatting codes.

split(*str***,** *array***,** *fs***)**
Divide the string *str* into fields associated with *array*. The fields are separated by character *fs* or the default field separator.

sprintf(*f***,** *e1***,** *e2***)**
Format strings *e1* and *e2* in the manner set by the formatting string *f*, and return the formatted string. *f* can use **printf()**-style formatting codes.

sqrt(*argument***)**
Return the square root of *argument*.

substr(*str***,** *beg***,** *len***)**
Scan string *str* for position *beg*; if found, print the next *len* characters. If *len* is not included, print from from *beg* to the end of the record.

Printing with awk

Printing is the commonest task you will perform in your **awk** programs. **awk**'s printing functions **printf** and **sprintf** resemble the C functions **printf()** and **sprintf()**; however, there are enough differences to make a close reading of this section worthwhile.

print is the commonest, and simplest, **awk** function. When used without any arguments, **print** prints all of the current record. The following example prints every record in file **text1**:

```
awk '{print}' text1
```

You can print fields in any order you desire. For example, the following program reverses the order of the season and batting-average columns from file **table1**:

```
awk '/1920/,/1925/ { print $2,$1 }' table1
```

The output is as follows:

```
.376 1920
.378 1921
.315 1922
.393 1923
.378 1924
.290 1925
```

Because the field names are separated by a comma, **awk** inserts the **OFS** between the fields when it prints them. If you do not separate field names with commas, **awk** concatenates the fields when it printing them. For example, the program

```
awk '/1920/,/1925/ { print $2 $1 }' table1
```

produces:

```
.3761920
.3781921
.3151922
.3931923
.3781924
.2901925
```

When you use **awk** to process a column of text or numbers, you may wish to specify a consistent format for the output. The statement for formatting a column of numbers follows this *pattern*:

```
{printf "format", expression}
```

where *format* prescribes how to format the output, and *expression* specifies the fields for **awk** to print.

The following table names and defines the most commonly used of **awk**'s format control characters. Each character must be preceded by a percent sign '%' and a number in the form of *n* or *n.m*.

%*n*d	Decimal number
%*n.m*f	Floating-point number
%*n.m*s	String
%%	Literal '%' character

When you use the **printf()** function, you must define the output-record separator within the format string. The following codes are available:

\n	Newline
\t	Tab
\f	Form feed
\r	Carriage return
\"	Quotation mark

For example, the following program prints Babe Ruth's RBIs unformatted:

```
awk '/1920/,/1925/ { print $1, $5 }' table1
```

The output appears as follows:

```
1920 137
1921 171
1922 99
1923 131
1924 121
1925 66
```

As you can see, **awk** right-justifies its output by default. To left-justify the second column, use the following program:

```
awk '/1920/,/1925/ { printf("%d %3d\n", $1, $5) }' table1
```

The output is as follows:

```
1920 137
1921 171
1922  99
1923 131
1924 121
1925  66
```

Note that the '3' in the string **%3d** specifies the minimum number of characters to be displayed. If the size of the number exceeds the space allotted to it, **awk** prints the entire number. A different rule applies when printing strings, as will be shown below.

To print a floating-point number, you must specify the minimum number of digits you wish to appear on either side of the decimal point. For example, the following program gives the average number of RBIs Babe Ruth hit in each game between 1920 and 1925:

```
awk '/1920/,/1925/ { printf("%d %1.2f\n", $1, $5/154.0) }' table1
```

This prints the following:

```
1920 0.89
1921 1.11
1922 0.64
1923 0.85
1924 0.79
1925 0.43
```

Note the following points about the above program:

- To get the average number of runs batted in, we had to divide the total number of RBIs in a season by the number of games in a season (which in the 1920s was 154). **awk** permits you to use a constant to perform arithmetic on a field; this will be discussed in more detail below.
- To force **awk** to produce a floating-point number, the constant had to be in the format of a floating-point number, i.e., "154.0" instead of "154". Dividing an integer by another integer would not have produced what we wanted.

awk rounds its output to match sensitivity you've requested — that is, the number of digits to the right of the decimal point. To see how sensitivity affects output, run the following program:

```
awk '/1920/,/1925/{printf("%1.2f %1.3f %1.4f\n",$5/154.0,$5/154.0,$5/154.0)}'\
table1
```

This prints the following:

```
0.89 0.890 0.8896
1.11 1.110 1.1104
0.64 0.643 0.6429
0.85 0.851 0.8506
0.79 0.786 0.7857
0.43 0.429 0.4286
```

As an aside, the above example also shows that you can break **awk**'s command line across more than one line using a backslash '\' at the end of every line but the last. Note, however, that you *cannot* break an **awk** statement across more than one line, or **awk** will complain about a syntax error.

One last example of floating-point numbers prints Babe Ruth's ratio of runs scored to runs batted in between 1920 and 1925:

```
awk '/1920/,/1925/{x = ($5*1.0) ; printf("%1.3f\n", $4/x)}' table1
```

This produces the following:

```
1.153
1.035
0.949
1.153
1.182
0.924
```

The expression **x = ($5*1.0)** was needed to turn field 5 (the divisor) into a floating-point number, so we could obtain the decimal fraction that we wanted. This is discussed further below, when we discuss how to manipulate constants.

The function **sprintf()** also formats expressions; however, instead of printing its output, it returns it for assignment to a variable. For example, you could rewrite the previous example program to replace the multiplication operation with a call to **sprintf()**:

```
awk '/1920/,/1925/{x = sprintf("%3.1f", $5)
       printf("%1.3f\n", $4/x)}' table1
```

The output is the same as that shown above.

The **%s** formatting string can be used to align text in fields. The digit to the left of the period gives the width of the field; that to the right of the period gives the number of characters to write into the field. Note that if input is larger than the number of characters allotted to it, **awk** truncates the input. For example, the following program aligns on seven-character fields some words from file **text1**:

```
awk '{x=sprintf("%7.5s %7.5s %7.5s %7.5s", $1, $2, $3, $4)
       print x}' text1
```

The output is as follows:

```
 When,      in   disgr    with
     I     all   alone   bewee
   And   troub    deaf   heave
   And    look    upon   mysel
 Wishi      me    like      to
 Featu    like    him,    like
 Desir    this   man's     art
  With    what       I    most
   Yet      in   these   thoug
 Haply       I   think      on
  Like      to     the    lark
  From   sulle   earth   sings
   For     thy   sweet    love
  That    then       I   scorn
```

Note that fields (words) longer than five characters are truncated; and every word is right-justified on a seven-character field.

Redirecting Output

In addition to printing to the standard output, **awk** can redirect the output of an action into a file, or append it onto an existing file. With this feature, you can extract information from a given file and construct new documents. The following example shows an easy way to sift Babe Ruth's statistics into four separate files, for further processing:

```
awk '{  print $1, $2 > "average"
        print $1, $3 > "home.runs"
        print $1, $4 > "runs.scored"
        print $1, $5 > "rbi"}' table1
```

Note like as under the shell, the operator '>' creates the named file if it does not exist, or replaces its contents if it does. To append **awk**'s onto the end of an existing file, use the operator '>>'.

awk can also pipe the output of an action to another program. As under the shell, the operator '|' pipes the output of one process into another process. For example, if it is vital for user **fred** to know Babe Ruth's batting average for 1925, you can mail it to him with the following command:

```
awk '/1925/ {print $1, $2 | "mail fred"}' table1
```

Assignment of Variables

A number of the previous examples assign values to variables. **awk** lets you create variables, perform arithmetic upon them, and otherwise work with them.

An **awk** variable can be a string or a number, depending upon the context. Unlike C, **awk** does not require that you declare a variable. By default, variables are set to the null string (numeric value zero) on start-up of the **awk** program. To set the variable **x** to the numeric value one, you can use the assignment operator '=':

```
x = 1
```

To set x to the string **ted** also use the assignment operator:

```
x = "ted"
```

When the context demands it, **awk** converts strings to numbers or numbers to strings. For example, the statement

```
x = "3"
```

initializes to **x** to the string "3". When an expression contains an arithmetic operator such as the '-', **awk** interprets the expression as numeric. (Alphabetic strings evaluate to zero.) Therefore, the expression

```
x = "3" - "1"
```

assigns the numeric value two to variable **x** not the string “2”.

When the operator is included within the quotation marks, **awk** treats the operator as a character in the string. In the following example

```
x = "3 - 1"
```

initializes **x** to the string “3 - 1”.

A number of examples in the previous section showed you how to perform arithmetic on fields. The following table gives **awk**’s arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Add and assign value
-=	Subtract and assign value
***=**	Multiply and assign value
/=	Divide and assign value
%=	Divide modulo and assign value

Variables are often used with increment operators. For example, the following program computes the average number of home runs Babe Ruth hit each season during the 1920s:

```
awk '   { x += $3 }
        END { y = (NR * 1.0)
                printf("Average for %d years: %2.3f.\n", NR, x/y) }' table1
```

The output is:

```
Average for 10 years: 46.700.
```

Field Variables

awk lets fields receive assignments, be used in arithmetic, and be manipulated in string operations. One task that has not yet been demonstrated is using a variable to address a field. For example, the following program prints the **NR**th field (word) from the first seven lines in file **text1**:

```
awk 'NR < 8 {print NR, $(NR)}' text1
```

The output is:

```
1 When,
2 all
3 deaf
4 myself,
5 one
6 with
7 man's
```

Control Statements

awk has seven defined control statements. This section explains them and gives examples of their use.

if (*condition***)** *action1* **[else** *action2* **]**

If *condition* is true, then execute *action1*. If the optional **else** clause is present and *condition* is false, then execute *action2*.

The following program keeps running totals of Babe Ruth’s RBIs, for both the years where his runs scored exceeded his RBIs and the years where they did not:

```
awk '{ if ( $4 > $5 )
            gyear++
       else
            lyear++
     }
     END { printf("Scored exceed RBIs: %d years.0, gyear)
           printf("Scored not exceed RBIs: %d years.0, lyear)
     }' table1
```

This produces:

```
Scored exceed RBIs: 5 years.
Scored not exceed RBIs: 5 years.
```

Note that if more than one action is associated with an **if** or **else** statement, you must enclose the statements between braces. If you use braces with both the **if** and **else** statements, note that the beginning and closing braces *must* appear on the same line as the **else** statement. For example:

```
if (expr) {
       stuff
       stuff
} else {
       stuff
       stuff
}
```

while (*condition***)** *action*

The **while** statement executes *action* as long as *condition* is true. For example, the following program counts the number of times the word **the** appears in file **text1**. The **while** loop uses a variable to examine every word in every line:

```
awk ' { i = 1
        while (i <= NF ) {
               if ($i == "the") j++
               i++
        }
      }
END   { printf ("The word \"the\" occurs %d times.\n", j) }' text1
```

The result, as follows, shows Shakespeare's economy of language:

```
The word "the" occurs 1 times.
```

By the way, note that if a control statement has more than one statement in its action section, enclose the action section between braces. If you do not, **awk** will behave erratically or exit with a syntax error.

for(*initial* **;** *end* **;** *iteration* **)** *action*
for(*variable* **in** *array* **)** *action*

awk's **for** statement closely resembles the **for** statement in the C language. The statement *initial* defines actions to be performed before the loop begins; this is usually used to initialize variables, especially counters. The statement *end* defines when the loop is to end. The statement *iteration* defines one or more actions that are performed on every iteration of the loop; usually this is used to increment counters. Finally, *action* can be one or more statements that are executed on every iteration of the loop. *action* need not be present, in which case only the action defined in the *iteration* portion of the **for** statement is executed. **for** is in fact just an elaboration of the **while** statement, but adjusted to make it a little easier to use. The following example writes the previous example, but replaces the **while** loop with a **for** mechanism:

```
awk ' { for (i = 1 ; i <= NF ; i++)
            if ($i == "the") j++
      }
END   { printf ("The word \"the\" occurs %d times.\n", j) }' text1
```

The output is the same as the previous example, but the syntax is neater and easier to read.

The second form of the **for** loop examines the contents of an array. It is described in the following section, which introduces arrays.

break The statement **break** immediately interrupts a **while** or **for** loop. For example, the following program is the same as the previous example, but counts only the first occurrence of the word **the** in each line of **text1**. Thus, it counts the number of lines in **text1** that contain **the**:

```
awk '{ for (i = 1 ; i <= NF ; i++) {
            if ($i == "the") {
                   j++
               break
            }
         }
      }
END   {printf ("The word \"the\" occurs in %d lines.\n", j)}' text1
```

continue
The statement **continue** immediately begins the next iteration of the nearest **while** or **for** loop. For example, the following program prints all of Babe Ruth's statistics — runs scored, runs batted, and home runs — in which he had more than 59 in one year:

```
awk ' { for (i = 3 ; i <= NF ; i++)
            if ($i <= 59)
               continue
            else
               printf("%d, column %d: %d\n", $1, i, $i)
      } ' table1
```

This produces the following:

```
1920, column 4: 158
1920, column 5: 137
1921, column 4: 177
1921, column 5: 171
1922, column 4: 94
1922, column 5: 99
...
```

next The statement **next** forces **awk** abort the processing of the current record and skip to the next input record. Processing of the new input record begins with the first pattern, just as if the processing of the previous record had concluded normally. To demonstrate this, the following program skips all records in file **text1** that have an odd number of fields (words):

```
awk ' { if (NF % 2 == 0) next }
      { print $0 } ' text1
```

This produces:

```
I all alone beweep my outcast state,
Wishing me like to one more rich in hope,
With what I most enjoy contented least.
Yet in these thoughts myself almost despising,
Like to the lark at break of day arising
```

exit Finally, the control statement **exit** forces the **awk** program to skip all remaining input and execute the *actions* at the **END** pattern, if any. For example, the following program prints the year in which Babe Ruth hit his 300th home run:

```
awk ' { i =  $1 }
      (j += $3) >= 300 { exit }
      END {print "Babe Ruth hit his 300th homer in", i "."}' table1
```

This produces:

```
Babe Ruth hit his 300th homer in 1926.
```

Arrays

awk has a powerful feature for managing arrays. Unlike C, **awk** automatically manages the size of an array, so you do not have to declare the array's size ahead of time. Also, unlike C, **awk** lets you address each element within an array by a label, not just by its offset within the array. This lets you generate arrays "on the fly," which can be very useful in transforming many varieties of data.

To declare an array, simply name it within a statement. **awk** recognizes as an array every variable that is followed by brackets '[ ]'. To initialize a row within an array, you must define its value and name its label. A label can be either a number or a string. A value, too, can be a number or a string; if the value is a number, then you can perform arithmetic upon it, as will be shown in a following example.

Initializing an Array

To demonstrate how an array works, use the line editor **ed** to add a line of text to the beginning of file **table1**. Type the following; please note that the token **<tab>** means that you should type a tab character:

```
ed table1
1i
Year<tab>BA<tab>HRs<tab>Scored<tab>RBIs
.
wq
```

This change writes a header into **table1** that names each column. Now, we can read these labels into an array and use them to describe Babe Ruth's statistics. For example, the following prints a summary of Babe Ruth's statistics for the year 1926:

```
awk '   NR == 1 { for (i=1; i <= NF; i++) header [i] = $i }
        $1 == 1926 {
                for (i=1; i <= NF; i++)
                        print header[i] ":\t", $i
        } ' table1
```

This produces:

```
Year:           1926
BA:             .372
HRs:            47
Scored:         139
RBIs:           145
```

The statement

```
NR == 1 { for (i=1; i <= NF; i++) header [i] = $i }
```

reads the first line in **table1**, which contains the column headers, and uses the headers to initialize the array **header**. Each row is labeled with the contents of the variable **i**.

The loop

```
for (i=1; i <= NF; i++)
        print header[i] ":\t", $i
```

prints the contents of **header**. Because we labeled each row within **header** with a number, we can use a numeric loop to read its contents.

The for() Statement With Arrays

In the previous example, each element in the array was labeled with a number. This permitted us to read the array with an ordinary **for** statement, which sets and increments a numeric variable. However, the rows within an array can be labeled with strings, instead of numbers. To read the contents of such an array, you must use a special form of the **for** statement, as follows:

for (*offset* **in** *array* **)**

array names the array in question. *offset* is a variable that you name at the time of contructing the **for** statement. You can use the value of *offset* in any subordinate printing actions.

The following program demonstrates this new form of **for**, and (incidentally) to demonstrate the power of **awk**'s array-handling feature. It builds an array of each unique word in the file **text1**, and notes the number of times that word occurs within the file:

```
awk '   { for (i = 1 ; i <= NF ; i++)
                words [$i]++ }
        END { for (entry in words)
                print entry ":", words[entry] }' text1 | sort
```

This prints:

```
-: 1
And: 2
Desiring: 1
Featured: 1
For: 1
From: 1
Haply: 1
I: 4
Like: 1
That: 1
When,: 1
Wishing: 1
With: 1
Yet: 1
all: 1
almost: 1
...
```

As you can imagine, a similar program in C would require many more lines of code. However, a few features of this program are worth noting.

First, the expression

```
{ for (i = 1 ; i <= NF ; i++)
       words [$i]++ }
```

declares the array **words** Every time **awk** encounters a new field (word), it automatically adds another entry to the array, and labels that entry with the word. No work on your part is needed for this to happen. The '++' operator increments the value of the appropriate entry within **words**. Because we did not initialize the entry, it implicitly contains a number.

The expression

```
{ for (entry in words)
       print entry ":", words[entry] }
```

walks through the array **words**. **awk** initializes the variable **entry** to the label for each row in **words**; the **print** statement then prints **entry** and the contents of that row in the array — in this case, the number of times the row appears in our input file.

Finally, we piped the output of this program to the command **sort** to print the words in alphabetical order.

For More Information

This tutorial just gives a brief introduction to the power of **awk**. To explore the language in depth, see *sed & awk* by Dale Dougherty (Sebastopol, Calif, O'Reilly & Associates, Inc., 1985). This book, however, describes a more complex version of **awk** than that provided with COHERENT.

The Lexicon's article on **awk** gives a quick summary of its features and options.