AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 118

Awk — A Pattern Scanning and Processing Language Programmer's Manual

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger

June 5, 1985

Awk — A Pattern Scanning and Processing Language
Programmer's Manual

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Awk is a programming language that allows many tasks of information retrieval, data processing, and report generation to be specified simply. An *awk* program is a sequence of pattern-action statements that searches a set of files for lines matching any of the specified patterns and executes the action associated with each matching pattern. For example, the pattern

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

is a complete *awk* program that prints all input lines whose first field is the string `name`; the action

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

is a complete program that prints the first and second fields of each input line; and the pattern-action statement

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

is a complete program that prints the second and third fields of each input line whose first field is `address`.

Awk patterns may include arbitrary combinations of regular expressions and comparison operations on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns as well as arithmetic and string expressions; assignments; `if-else`, `while` and `for` statements; function calls; and multiple input and output streams.

This manual describes the version of *awk* released in June, 1985.

June 5, 1985

Awk — A Pattern Scanning and Processing Language
Programmer's Manual

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Basic Awk

Awk is a programming language for information retrieval and data manipulation. Since it was first introduced in 1979, *awk* has become popular even among people with no programming background. This manual begins with the basics of *awk*, and is intended to make it easy for anyone to get started; the rest of the manual describes the complete language and is somewhat less tutorial. For the experienced *awk* user, Appendix A contains a summary of the language; Appendix B contains a synopsis of the new features added to the language in the June, 1985 release.

1.1. Program Structure

The basic operation of *awk* is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions that the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern. Accordingly, an *awk* program is a sequence of pattern-action statements of the form

pattern { *action* }
pattern { *action* }
...

The third program in the abstract,

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

is a typical example, consisting of one pattern-action statement. Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

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

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

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

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

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

1.2. Usage

There are two ways to run an *awk* program. You can type the command

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

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

say

```
awk '{ print $1, $2 }' data1 data2
```

If no files are mentioned on the command line, the *awk* interpreter will read the standard input. Notice that the pattern-action statements are enclosed in single quotes. This protects characters like `$` from being interpreted by the shell and also allows the program to be longer than one line.

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

```
awk -f myprogram optional list of input files
```

Any filename can be used in place of `myprogram`.

1.3. Fields

Awk normally reads its input one line at a time; it splits each line into a sequence of *fields*, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the *awk* programs in this manual, we will use the following file, `countries`. Each line contains the name of a country, its area in thousands of square miles, its population in millions, and the continent where it is, for the ten largest countries in the world. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.)

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

The wide space between fields is a tab in the original input; a single blank separates `North` and `South` from `America`. This file is typical of the kind of data that *awk* is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a line is determined by the *field separator*. Fields are normally separated by sequences of blanks and/or tabs, in which case the first line of `countries` would have 4 fields, the second 5, and so on. It's possible to set the field separator to just tab, so each line would have 4 fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs.

The first field within a line is called `$1`, the second `$2`, and so forth. The entire line is called `$0`.

1.4. Printing

If the pattern in a pattern-action statement is missing, the action is executed for all input lines. The simplest action is to print each line; this can be accomplished by the *awk* program consisting of a single `print` statement:

```
{ print }                                                    (P.1)
```

so the command

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

prints each line of `countries`, thus copying the file to the standard output.

In the remainder of this paper, we shall only show *awk* programs, without the command line that invokes them. Each complete program is identified by `(P.n)` in the right margin; in each

case, the program can be run either by enclosing it in quotes as the first argument of the `awk` command as shown above, or by putting it in a file and invoking `awk` with the `-f` flag, as discussed in Section 1.2. In an example, if no input is mentioned, it is assumed to be the file `countries`.

The `print` statement can be used to print parts of a record; for instance, the program

```
{ print $1, $3 }                                                    (P.2)
```

prints the first and third fields of each input line. Thus

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

produces as output the sequence of lines:

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

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

1.5. Formatted Printing

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

`printf` *format*, $expr_1$, $expr_2$, ... , $expr_n$

which prints the $expr_i$'s according to the specification in the string *format*. For example, the *awk* program

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

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

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

With `printf`, no output separators or newlines are produced automatically; you must create them yourself, which is the purpose of the `\n` in the format specification. Section 4.3 contains a full description of `printf`.

1.6. Built-In Variables

Besides reading the input and splitting it into fields, *awk* counts the number of lines read and the number of fields within the current line; you can use these counts in your *awk* programs. The variable `NR` is the number of the current input line, and `NF` is the number of fields. So the program

```
{ print NR, NF }                                                    (P.4)
```

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

```
{ print NR, $0 }                                                    (P.5)
```

prints each line preceded by its line number.

1.7. Simple Patterns

You can select specific lines for printing or other processing with simple patterns. For example, the operator == tests for equality. To print the lines for which the fourth field equals the string `Asia` we can use the program consisting of the single pattern:

```
$4 == "Asia"                                                        (P.6)
```

With the file `countries` as input, this program yields

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

The complete set of comparisons is >, >=, <, <=, == (equal to) and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with more than 100 million population. The program

```
$3 > 100                                                            (P.7)
```

is all that is needed (remember that the third field is the population in millions); it prints all lines in which the third field exceeds 100.

You can also use patterns called "regular expressions" to select lines. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/                                                                (P.8)
```

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

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

We will have a lot more to say about regular expressions in §2.4.

There are two special patterns, `BEGIN` and `END`, that "match" before the first input line has been read and after the last input line has been processed. This program uses `BEGIN` to print a title:

```
BEGIN   { print "Countries of Asia:" }
/Asia/  { print "     ", $1 }                                       (P.9)
```

The output is

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

1.8. Simple Arithmetic

In addition to the built-in variables like `NF` and `NR`, *awk* lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in file `countries`:

```
        { sum = sum + $3 }                                              (P.10)
END     { print "Total population is", sum, "million"
          print "Average population of", NR, "countries is", sum/NR }
```

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

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

1.9. A Handful of Useful "One-liners"

Although *awk* can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you might find useful and/or instructive. They are not explained here, but any new constructs do appear later in this manual.

Print last field of each input line:

```
        { print $NF }                                                   (P.11)
```

Print 10th input line:

```
        NR == 10                                                        (P.12)
```

Print last input line:

```
                { line = $0}
        END     { print line }                                          (P.13)
```

Print input lines that don't have 4 fields:

```
        NF != 4 { print $0, "does not have 4 fields" }                  (P.14)
```

Print input lines with more than 4 fields:

```
        NF > 4                                                          (P.15)
```

Print input lines with last field more than 4:

```
        $NF > 4                                                         (P.16)
```

Print total number of input lines:

```
        END     { print NR }
```

Print total number of fields:

```
                { nf = nf + NF }
        END     { print nf }                                            (P.17)
```

Print total number of input characters:

```
                { nc = nc + length($0) }
        END     { print nc + NR }                                       (P.18)
```

(Adding NR includes in the total the number of newlines.)

Print the total number of lines that contain `Asia`:

```
        /Asia/  { nlines++ }
        END     { print nlines }                                        (P.19)
```

(The statement `nlines++` has the same effect as `nlines = nlines + 1`.)

1.10. Errors

If you make an error in your *awk* program, you will generally get a message like

```
awk: syntax error near source line 2
awk: bailing out near source line 2
```

The first message means that you have made a grammatical error that was finally detected near the line specified; the second indicates that no recovery was possible. Sometimes you will get a little more help about what the error was, for instance a report of missing braces or unbalanced parentheses.

The "bailing out" message means that because of the syntax errors *awk* made no attempt to execute your program. Some errors may be detected when your program is running. For example, if you try to divide a number by zero, *awk* will stop processing and report the input line number and the line number in the program.

2. Patterns

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

2.1. BEGIN and END

The special pattern `BEGIN` matches before the first input record is read, so any statements in the action part of a `BEGIN` are done once before *awk* starts to read its first input file. The pattern `END` matches the end of the input, after the last file has been processed. `BEGIN` and `END` provide a way to gain control for initialization and wrapup.

The field separator is stored in a built-in variable called `FS`. Although `FS` can be reset at any time, usually the only sensible place is in a `BEGIN` section, before any input has been read. For example, the following *awk* program uses `BEGIN` to set the field separator to tab (`\t`) and to put column headings on the output. The second `printf` statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The `END` action prints the totals. Notice that a long line can be continued after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
                "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
        area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }          (P.20)
```

With the file `countries` as input, `(P.20)` produces

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

     TOTAL  30292  2201
```

2.2. Relational Expressions

An *awk* pattern can be any expression involving comparisons between strings of characters or numbers. *Awk* has six relational operators, and two regular expression matching operators ~ (tilde) and !~ that will be discussed in the next section.

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise the operands are compared as strings. (Every value might be either a number or a string; usually *awk* can tell what was intended. The full story is in §3.4.) Thus, the pattern `$3>100` selects lines where the third field exceeds 100, and

TABLE 1. COMPARISON OPERATORS

OPERATOR	MEANING
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

```
$1 >= "S"                                                    (P.21)
```

selects lines that begin with an S, T, U, etc., which in our case are

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

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

```
$1 == $4                                                     (P.22)
```

will compare the first and fourth fields as strings of characters, and with the file `countries` as input, will print the single line for which this test succeeds:

```
Australia        2968    14      Australia
```

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

2.3. Regular Expressions

Awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called *regular expressions*, and are like those in the Unix™ programs *egrep* and *lex*.

The simplest regular expression is a string of characters enclosed in slashes, like

```
/Asia/                                                       (P.23)
```

Program (P.23) prints all input lines that contain any occurrence of the string `Asia`. (If a line contains `Asia` as part of a larger word like `Asian` or `Pan-Asiatic`, it will also be printed.)

If *re* is a regular expression, then the pattern

```
/re/
```

matches any line that contains a substring specified by the regular expression *re*. To restrict the match to a specific field, use the matching operators ~ (for matches) and !~ (for does not match):

```
$4 ~ /Asia/ { print $1 }                                     (P.24)
```

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

```
$4 !~ /Asia/ { print $1 }                                    (P.25)
```

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

In regular expressions the symbols

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

have special meanings and are called *metacharacters*. For example, the metacharacters ^ and $ match the beginning and end, respectively, of a string, and the metacharacter . matches any single character. Thus,

```
/^.$/                                                    (P.26)
```

will match all lines that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, `/[ABC]/` matches lines containing any one of `A`, `B` or `C` anywhere. Ranges of letters or digits can be abbreviated: `/[a-zA-Z]/` matches any single letter. If the first character after the `[` is a `^`, this complements the class so it matches any character *not* in the set: `/[^a-zA-Z]/` matches any non-letter.

The program

```
$2 !~ /^[0-9]+$/                                         (P.27)
```

prints all lines in which the second field is not a string of one or more digits (`^` for beginning of string, `[0-9]+` for one or more digits, and `$` for end of string). Programs of this nature are often used for data validation.

Parentheses `( )` are used for grouping and `|` is used for alternatives:

```
/(apple|cherry) (pie|tart)/                              (P.28)
```

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

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

```
/a\$/                                                    (P.29)
```

will print all lines containing an `a` followed by a dollar sign.

Awk recognizes the following C escape sequences within regular expressions and strings:

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

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

```
/\t/                                                     (P.30)
```

Awk will interpret any string or variable on the right side of a `~` or `!~` as a regular expression. For example, we could have written program `(P.27)` as

```
BEGIN   { digits = "^[0-9]+$" }
$2 !~ digits                                             (P.31)
```

When a literal quoted string like `"^[0-9]+$"` is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. The reason may seem arcane, but it is merely that one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

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

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

Of course, if the context of a matching operator is

```
x ~ $1
```

then the additional level of backslashes is not needed in the first field.

The precise form of regular expressions and the substrings they match is given in Table 2. The unary operators *, +, and ? have the highest precedence, then concatenation, and then alternation |. All operators are left associative.

TABLE 2. Awk REGULAR EXPRESSIONS

EXPRESSION	MATCHES
c	any non-metacharacter *c*
c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
r_1r_2	r_1 then r_2 (concatenation)
$r_1 \mid r_2$	r_1 or r_2 (alternation)

2.4. Combinations of Patterns

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

```
$4 == "Asia" && $3 > 500                    (P.32)
```

The program

```
$4 == "Asia" || $4 == "Africa"              (P.33)
```

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

```
$4 ~ /^(Asia|Africa)$/                      (P.34)
```

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

2.5. Pattern Ranges

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

pat$_1$, *pat*$_2$ { ... }

In this case, the action is performed for each line between an occurrence of *pat*$_1$ and the next

occurrence of pat_2 (inclusive). As an example, the pattern

```
/Canada/, /Brazil/                                    (P.35)
```

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

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

Similarly, since `FNR` is the number of the current record in the current input file, the program

```
FNR == 1, FNR == 5 { print FILENAME, $0 }             (P.36)
```

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

3. Actions

In a pattern-action statement, the pattern selects input records; the action determines what is to be done with them. Actions frequently are simple print or assignment statements, but may be an arbitrary sequence of statements separated by newlines or semicolons. This section describes the statements that can make up actions.

3.1. Built-in Variables

Table 3 lists the built-in variables that *awk* maintains. Some of these we have already met; others will be used in this and later sections.

TABLE 3. BUILT-IN VARIABLES

VARIABLE	MEANING	DEFAULT
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline

3.2. Arithmetic

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

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }         (P.37)
```

applied to `countries` prints the name of the country and its population density:

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

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

Awk has C-like assignment statements. The simplest form is the assignment statement

v = e

where *v* is a variable or field name, and *e* is an expression. For example, to compute the total population and number of Asian countries, we could write

```
$4 == "Asia"    { pop = pop + $3; n = n + 1 }                    (P.38)
END             { print "population of", n,\
                        "Asian countries in millions is", pop }
```

(A long *awk* statement can also be split across several lines by continuing each line with a \, as in the `END` action of `(P.38)`). Applied to `countries`, `(P.38)` produces

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

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

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

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

The operator += is borrowed from the programming language C. It has the same effect as the longer version — the variable on the left is incremented by the value of the expression on the right — but += is shorter and runs faster. The same is true of the ++ operator, which adds 1 to a variable.

The abbreviated assignment operators are +=, -=, *=, /=, %=, and ^=. Their meanings are similar: *v op= e* has the same effect as *v = v op e*. The increment operators are ++ and --. As in C, they may be used as prefix operators (++x) or postfix (x++). If `x` is 1, then `i=++x` increments `x`, then sets `i` to 2, while `i=x++` sets `i` to 1, then increments `x`. An analogous interpretation applies to prefix and postfix --.

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

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3     { maxpop = $3; country = $1 }
END             { print country, maxpop }                        (P.39)
```

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

Awk provides the built-in arithmetic functions shown in Table 4.

TABLE 4. BUILT-IN ARITHMETIC FUNCTIONS

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

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

3.3. Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in `"abc"` or `"hello, everyone"`. String constants may contain the C escape sequences for special characters listed in §2.3.

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

```
{ print NR ":" $0 }                                    (P.40)
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

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

TABLE 5. BUILT-IN STRING FUNCTIONS

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

The functions `sub` and `gsub` are patterned after the substitute command in the text editor *ed*. The function `gsub(`*r*`,`*s*`,`*t*`)` replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in *ed*, leftmost longest matches are used.) It returns the number of substitutions made. The function `gsub(`*r*`,`*s*`)` is a synonym for `gsub(`*r*`,`*s*`,$0)`. For example, the program

```
{ gsub(/USA/, "United States"); print }                    (P.41)
```

will transcribe its input, replacing occurrences of "USA" by "United States". The sub functions are similar, except that they only replace the first matching substring in the target string.

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

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

returns 2.

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

```
{ print length($0), $0 }                                  (P.42)
```

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

```
length($1) > max        { max = length($1); name = $1 }
END                     { print name }                    (P.43)
```

applied to the file countries prints the longest country name:

```
Australia
```

The function sprintf(*format*, $expr_1$, $expr_2$, ... , $expr_n$) returns (without printing) a string containing $expr_1$, $expr_2$, ..., $expr_n$ formatted according to the printf specifications in the string *format*. Section 4.3 contains a complete specification of the format conventions. Thus, the statement

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

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

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

```
{ $1 = substr($1, 1, 3); print }                          (P.44)
```

on this file to produce

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

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

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

```
        { s = s substr($1, 1, 3) " " }
END     { print s }                                       (P.45)
```

prints

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

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

3.4. Field Variables

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

```
{ $2 /= 1000; print }                                    (P.46)
```

or assign a new string to a field:

```
BEGIN                    { FS = OFS = "\t" }
$4 == "North America"    { $4 = "NA" }
$4 == "South America"    { $4 = "SA" }                   (P.47)
                         { print }
```

The BEGIN action in (P.47) resets the input field separator FS and the output field separator OFS to a tab. Notice that the print in the fourth line of (P.47) prints the value of $0 after it has been modified by previous assignments.

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

A field variable referring to a nonexistent field, e.g., $(NF+1) has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file countries creates a fifth field giving the population density:

```
BEGIN    { FS = OFS = "\t" }
         { $5 = 1000 * $3 / $2; print }                  (P.48)
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

3.5. Number or String?

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

```
pop += $3
```

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

In a string context like

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

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

In an assignment $v = e$ or $v\ op = e$, the type of v becomes the type of e.

In an ambiguous context like

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

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field

variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "`$1 == $2`" will succeed on any pair of the inputs

```
1       1.0     +1      0.1e+1  10E-1   1e2     10e1    001
```

but fail on the inputs

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

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

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

Thus, to force a string comparison between two fields, say

```
$1 "" == $2 ""                                                    (P.49)
```

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

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

3.6. Control Flow Statements

Awk provides `if-else`, `while`, and `for` statements, and statement grouping with braces, as in C.

The `if` statement syntax is

`if (`*expression*`)` *statement*$_1$ `else` *statement*$_2$

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

In the `if` statement, the *expression* is first evaluated. If it is non-zero and non-null, *statement*$_1$ is executed; otherwise *statement*$_2$ is executed. The `else` part is optional.

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

Rewriting the maximum population program (`P.39`) from §3.1 with an `if` statement results in

```
{       if (maxpop < $3) {
                maxpop = $3
                country = $1                                      (P.50)
        }
}
END     { print country, maxpop }
```

The `while` statement is exactly that of C:

`while (`*expression*`)` *statement*

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

```
{       i = 1
        while (i <= NF) {
                print $i                                        (P.51)
                i++
        }
}
```

The `for` statement is like that of C:

`for` (*expression*$_1$; *expression*; *expression*$_2$) *statement*

has the same effect as

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

so

```
{ for (i = 1; i <= NF; i++)  print $i }                         (P.52)
```

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

The `break` statement causes an immediate exit from an enclosing `while` or `for`; the `continue` statement causes the next iteration to begin.

The `next` statement causes *awk* to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The `exit` statement causes the program to behave as if the end of the input had occurred; no more input is read, and the `END` action, if any, is executed. Within the `END` action,

`exit` *expr*

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

3.7. Arrays

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

As an example of a conventional numeric subscript, the statement

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

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

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

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

Array elements may also be named by nonnumeric values, a facility that gives *awk* a capability rather like the associative memory of Snobol tables. For example, the following program accumulates the total population of `Asia` and `Africa` into the associative array `pop`. The `END` action prints the total population of these two continents.

```
/Asia/          { pop["Asia"] += $3 }                                    (P.53)
/Africa/        { pop["Africa"] += $3 }
END             { print "Asian population in millions is", pop["Asia"]
                  print "African population in millions is", pop["Africa"] }
```

On `countries`, `(P.53)` generates

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

In program `(P.53)`, if we had used `pop[Asia]` instead of `pop["Asia"]` the expression would have used the value of the variable `Asia` as the subscript, and since the variable is uninitialized, the values would have been accumulated in `pop[""]`.

Suppose our task is to determine the total area in each continent of the file `countries`. Any expression can be used as a subscript in an array reference. Thus

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

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

```
BEGIN   { FS = "\t" }
        { area[$4] += $2 }                                               (P.54)
END     { for (name in area)
                print name, area[name] }
```

Invoked on `countries`, `(P.54)` produces

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

`(P.54)` uses a form of the `for` statement that iterates over all defined subscripts of an array:

```
for (i in array) statement
```

executes *statement* with the variable `i` set in turn to each value of `i` for which `array[i]` has been defined. The loop is executed once for each defined subscript, in a random order. Chaos will result if `i` is altered during the loop.

Awk does not provide multi-dimensional arrays so you cannot write `x[i,j]` or `x[i][j]`. You can, however, create your own subscripts by concatenating row and column values with a suitable separator. For example,

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

creates an array whose subscripts have the form `i,j`, such as `1,1` or `1,2`. (The comma distinguishes a subscript like `1,12` from one like `11,2`.)

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

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

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

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

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

It is also possible to split any string into fields in the elements of an array using the built-in function `split`. The function

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

splits the string `s1:s2:s3` into three fields, using the separator `:` and storing `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]`. The number of fields found, here 3, is returned as the value of `split`. The third argument of `split` is a regular expression to be used as the field separator. If the third argument is missing, `FS` is used as the field separator.

An array element may be deleted with the `delete` statement:

`delete` *arrayname*`[`*subscript*`]`

3.8. User-Defined Functions

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

```
func name(argument-list) {
        statements
}
```

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

```
func fact(n) {
        if (n <= 1)
                return 1
        else                                                    (P.55)
                return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

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

3.9. Comments

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

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

4. Output

The `print` and `printf` statements are the two primary constructs that generate output. The `print` statement is used to generate quick-and-dirty output; `printf` is used for more carefully formatted output.

4.1. Print

The statement

`print` $expr_1$, $expr_2$, ... , $expr_n$

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line use

```
print ""
```

4.2. Output Separators

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

```
BEGIN   { OFS = ":"; ORS = "\n\n" }
        { print $1, $2 }                                    (P.56)
```

Notice that

```
        { print $1 $2 }                                     (P.57)
```

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

4.3. Printf

Awk's `printf` statement is the same as that in C except that the `c` and `*` format specifiers are not supported. The `printf` statement has the general form

`printf` *format*, *expr*$_1$, *expr*$_2$, ... , *expr*$_n$

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Table 6. Each specification begins with a `%`, ends with a letter that determines the conversion, and may include

`-`	left-justify expression in its field
width	pad field to this width as needed; leading 0 pads with zeros
.prec	maximum string width or digits to right of decimal point

TABLE 6. CONVERSION CHARACTERS

CHARACTER	PRINT EXPRESSION AS
`d`	decimal number
`e`	`[-]d.dddddd[+-]dd`
`f`	`[-]ddd.dddddd`
`g`	`e` or `f` conversion, whichever is shorter, with nonsignificant zeros suppressed
`o`	unsigned octal number
`s`	string
`x`	unsigned hexadecimal number
`%`	print a `%`; no argument is converted

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

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

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

4.4. Output into Files

It is possible to print output into files instead of to the standard output. The following program invoked on the file `countries` will print all lines where the population (third field) is bigger than 100 into a file called `bigpop`, and all other lines into `smallpop`:

```
$3 > 100        { print $1, $3 >"bigpop" }
$3 <= 100       { print $1, $3 >"smallpop" }                    (P.58)
```

Notice that the filenames have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. It is important to note that the files are opened once; each successive `print` or `printf` statement adds more data to the corresponding file. If >> is used instead of >, output is appended to the file rather than overwriting its original contents.

4.5. Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of a file. The statement

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

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

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and filenames can come from variables, etc., as well.

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

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (c in pop)                                        (P.59)
                print c ":" pop[c] | "sort" }
```

Invoked on the file `countries` (`P.59`) yields

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

In all of these `print` statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named `sort`), but they are created and opened only once in the entire run.

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

5. Input

There are several ways of providing the input data to an *awk* program *P*. The most common arrangement is to put the data into a file, say `awkdata`, and then execute

```
awk 'P' awkdata
```

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

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

Egrep will quickly find the lines containing `Asia` and pass them on to the *awk* program for subsequent processing.

5.1. Input Separators

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

```
    field1      field2
  field1
field1
```

When the field separator is a tab, however, leading blanks are *not* discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable `FS`. For example,

```
awk 'BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" } ...'
```

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

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

behaves the same as the previous example. Regular expressions used as field separators will not match null strings.

5.2. Multi-Line Records

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

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

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

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

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. Sections 5.3 and 6.2 show other examples of processing multi-line records.

5.3. The getline Function

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

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

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with START and ends with a line beginning with STOP. The following *awk* program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

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

Once the line containing STOP is encountered, the record can be processed from the data in the f array:

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

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

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

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

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

```
getline <"file"
```

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

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

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

It is also possible to pipe the output of another command directly into `getline`. For example, the statement

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

executes who and pipes its output into getline. Each iteration of the while loop reads one more line and increments the variable n, so after the while loop terminates, n contains a count of the number of users. Similarly, the statement

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

pipes the output of date into the variable d, thus setting d to the current date.

Table 7 summarizes the getline function.

TABLE 7. GETLINE FUNCTION

FORM	SETS
getline	**$0, NF, NR, FNR**
getline *var*	*var*, **NR, FNR**
getline <*file*	**$0, NF**
getline *var* <*file*	*var*
cmd **| getline**	**$0, NF**
cmd **| getline** *var*	*var*

5.4. Command-line Arguments

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

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

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

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

var=value

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

6. Cooperation with the Rest of the World

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

6.1. The system Function

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

For example, the program

```
$1 == "#include"  { gsub(/[<>"]/, "", $2); system("cat " $2) }        (P.60)
```

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

6.2. Cooperation with the Shell

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

```
awk '{ print $1 }' ...
```

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

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

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

Records are separated by a single blank line.

We want to search the address list by issuing commands like

```
addr Emlin
```

That is easily done by a program of the form

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

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

There are several ways to do this. One way is to create a file called `addr` that contains

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

The quotes are critical here: the *awk* program is only one argument, even though there are two sets of quotes, because quotes do not nest. The `$1` is outside the quotes, visible to the shell, which therefore replaces it by the pattern `Emlin` when the command `addr Emlin` is invoked.†

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

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

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

A third way to implement `addr` is to use `ARGV` to pass the regular expression to an *awk*

† On a Unix system, `addr` can be made executable by changing its mode with the command: `chmod +x addr`.

program that explicitly reads through the address list with `getline`:

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

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

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

7. Generating Reports

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

```
Africa:
        Sudan           19
        Algeria         18

Asia:
        China          866
        India          637
        USSR           262

Australia:
        Australia       14

North America:
        USA            219
        Canada          24

South America:
        Brazil         116
        Argentina       26
```

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

```
BEGIN   { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END     { for (cc in pop)                                          (P.61)
                print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for the sort command deserve special mention. The `-t:` argument tells `sort` to use : as its field separator. The `+0 -1` arguments make the first field the primary sort key. In general, $+i\ -j$ makes fields $i+1$, $i+2$, ..., j the sort key. If $-j$ is omitted, the fields from $i+1$ to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (**n** is for numeric, **r** for reverse order). The Unix Programmer's Manual contains a complete description of the **sort** command. Invoked on the file **countries**, **(P.61)** produces as output

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

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

```
BEGIN   { FS = ":" }
{       if ($1 != prev) {
                print "\n" $1 ":"
                prev = $1                                        (P.62)
        }
        printf "\t%-10s %6d\n", $2, $3
}
```

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

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

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

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

8. Additional Examples

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

1. *Word frequencies*. Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```
        { for (w = 1; w <= NF; w++) count[$w]++ }                (P.63)
END     { for (w in count) print count[w], w | "sort -nr" }
```

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

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

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

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

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

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

4. *Shell facility.* The following *awk* program simulates (crudely) the history facility of the Unix system shell. A line containing only = re-executes the last command executed. A line beginning with = *cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
                system(x[NR] = x[NR-1])
            else
                for (i = NR-1; i > 0; i--)                    (P.64)
                        if (x[i] ~ $2) {
                                system(x[NR] = x[i])
                                break
                        }
            next }

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

5. *Form-letter generation.* The following program generates form letters, using a template stored in a file called `form.letter`:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

and replacement text of this form:

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

The `BEGIN` action stores the template in the array `template`; the remaining action cycles through the input data, using `gsub` to replace template fields of the form `$n` with the corresponding data fields.

```
BEGIN { FS = "|"
        while (getline <"form.letter")
                line[++n] = $0
}
{       for (i = 1; i <= n; i++) {
                s = line[i]
                for (j = 1; j <= NF; j++)
                        gsub("\\$"j, $j, s)
                print s
        }
}
```

6. *Random sentences*. Our final problem is to generate random sentences, given a grammar. Given input like

```
S -> NP VP
NP -> AL N
NP -> N
N -> John
N -> Mary
AL -> A
AL -> A AL
A -> Wee
A -> Little
VP -> V AvL
V -> runs
V -> walks
AvL -> Av
AvL -> ML Av
Av -> quickly
Av -> slowly
ML -> M
ML -> ML M
M -> very
gen S
```

it will generate sentences like

```
John runs quickly
Wee Little Mary runs quickly
Mary runs very very slowly
```

The following program presents a fairly naive approach: each left-hand side is remembered in an associative array, along with the components of its right-hand side. When a `gen` command occurs, a random instance of that left-hand side is expanded recursively.

```
{       if ($1 == "gen") {
                gen($2)
                print ""
        } else if ($2 == "->") {
                i = ++lhsct[$1]
                rhsct[$1 "," i] = NF-2
                for (j = 3; j <= NF; j++)
                        rhslist[$1 "," i "," j-2] = $j
        } else
                print "Unrecognized command: " $0
}
```

```
func gen(sym, i, j) {   # i and j are local variables
        if (sym in lhsct) {
                i = int(lhsct[sym] * rand()) + 1
                for (j = 1; j <= rhsct[sym "," i]; j++)
                        gen(rhslist[sym "," i "," j])
        } else
                printf "%s ", sym
}
```

Notice the use of extra arguments in the list of parameters for `gen`; they serve as local variables for that specific instance of the function.

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

Further Reading

A technical discussion of the design of *awk* may be found in *Awk — a pattern scanning and processing language*, by A. V. Aho, B. W. Kernighan and P. J. Weinberger, which appeared in *Software Practice and Experience*, April 1979.

Much of the syntax of *awk* is derived from C, described in *The C Programming Language*, by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978).

The function `printf` is described in the C book, and also in Section 2 of *The Unix Programmer's Manual*. The programs *ed*, *sed*, *egrep*, and *lex* are also described there, with an explanation of regular expressions.

The Unix Programming Environment, by B. W. Kernighan and R. Pike (Prentice-Hall, 1984) contains a large number of *awk* examples, including illustrations of cooperation with *sed* and the shell. Jon Bentley's *Programming*Pearls columns in the June and July 1985 issues of *CACM* contain a wide variety of other *awk* examples.

Acknowledgements

We are indebted to Jon Bentley, Lorinda Cherry, Marion Harris, Teresa Alice Hommel, Rob Pike, Chris Van Wyk, and Vic Vyssotsky for valuable comments on drafts of this manual.

Appendix A: Awk Summary

Command-line

```
awk 'program' filenames
awk -f program-file filenames
awk -Fs          set field separator to string s; -Ft sets separator to tab
```

Patterns

```
BEGIN
END
/regular expression/
relational expression
pattern && pattern
pattern || pattern
(pattern)
!pattern
pattern, pattern
func name(parameter list) { statement }
```

Control-flow statements

```
if (expr) statement [else statement]
if (subscript in array) statement [else statement]
while (expr) statement
for (expr; expr; expr) statement
for (var in array) statement
break
continue
next
exit [expr]
function-name(expr, expr, ...)
return [expr]
```

Input-output

`close(`*filename*`)`	close file
`getline`	set `$0` from next input record; set `NF`, `NR`, `FNR`
`getline <`*file*	set `$0` from next record of *file*; set `NF`
`getline` *var*	set *var* from next input record; set `NR`, `FNR`
`getline` *var* `<`*file*	set *var* from next record of *file*
`print`	print current record
`print` *expr-list*	print expressions
`print` *expr-list* `>`*file*	print expressions on *file*
`printf` *fmt, expr-list*	format and print
`printf` *fmt, expr-list* `>`*file*	format and print on *file*
`system(`*cmd-line*`)`	execute command *cmd-line*, return status

In `print` and `printf` above, `>>`*file* appends to the *file*, and `|` *command* writes on a pipe. Similarly, *command* `|` `getline` pipes into `getline`. `getline` returns 0 on end of file, and −1 on error.

String functions

`gsub(`*r*`,`*s*`,`*t*`)`	substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use `$0`
`index(`*s*`,`*t*`)`	return index of string *t* in string *s*, or 0 if not present
`length(`*s*`)`	return length of string *s*
`split(`*s*`,`*a*`,`*r*`)`	split string *s* into array *a* on regular expression *r*, return number of fields if *r* omitted, `FS` is used in its place
`sprintf(`*fmt*`, `*expr-list*`)`	print *expr-list* according to *fmt*, return resulting string
`sub(`*r*`,`*s*`,`*t*`)`	like `gsub` except only the first matching substring is replaced
`substr(`*s*`,`*i*`,`*n*`)`	return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

Arithmetic functions

`atan2(`*y*`,`*x*`)`	arctangent of *y*/*x* in radians
`cos(`*expr*`)`	cosine (angle in radians)
`exp(`*expr*`)`	exponential
`int(`*expr*`)`	truncate to integer
`log(`*expr*`)`	natural logarithm
`rand()`	random number between 0 and 1
`sin(`*expr*`)`	sine (angle in radians)
`sqrt(`*expr*`)`	square root
`srand(`*expr*`)`	new seed for random number generator; use time of day if no *expr*

Operators (increasing precedence)

`= += -= *= /= %= ^=`	assignment
`||`	logical OR
`&&`	logical AND
`~ !~`	regular expression match, negated match
`< <= > >= != ==`	relationals
blank	string concatenation
`+ -`	add, subtract
`* / %`	multiply, divide, mod
`+ - !`	unary plus, unary minus, logical negation
`^`	exponentiation (`**` is a synonym)
`++ --`	increment, decrement (prefix and postfix)
`$`	field

Regular expressions (increasing precedence)

c	matches non-metacharacter *c*
`\`*c*	matches literal character *c*
`.`	matches any character but newline
`^`	matches beginning of line or string
`$`	matches end of line or string
`[`*abc...*`]`	character class matches any of *abc...*
`[^`*abc...*`]`	negated class matches any but *abc...* and newline
r1`|`*r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r`+`	matches one or more *r*'s
r`*`	matches zero or more *r*'s
r`?`	matches zero or one *r*'s
`(`*r*`)`	grouping: matches *r*

Built-in variables

`ARGC`	number of command-line arguments
`ARGV`	array of command-line arguments (0..`ARGC`-1)
`FILENAME`	name of current input file
`FNR`	input record number in current file
`FS`	input field separator (default blank)
`NF`	number of fields in current input record
`NR`	input record number since beginning
`OFMT`	output format for numbers (default `%.6g`)
`OFS`	output field separator (default blank)
`ORS`	output record separator (default newline)
`RS`	input record separator (default newline)

Limits

Any particular implementation of *awk* enforces some limits. Here are typical values:

100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per `printf` string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the local machine, e.g., 1e−38..1e+38

Initialization, comparison, and type coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

```
var = expr
```

its type is set to that of the expression. ("Assignment" includes `+=`, `-=`, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(i.e., concatenation with a null string).

Uninitialized variables have the numeric value 0 and the string value `""`. Accordingly, if `x` is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But note that

```
if (x == "0") ...
```

is false.

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that `$1` is to be numeric, and

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

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

In contexts where types cannot be reliably determined, e.g.,

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

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value `""`; they are not numeric. Non-existent fields (i.e., fields past `NF`) are treated this way too.

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

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

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

causes it to exist with the value `""` and thus the `if` is satisfied. The special construction

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

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

Appendix B: A Summary of New Features

This appendix summarizes the new features that have been added to *awk* for the June, 1985 release.

Regular expressions may be created dynamically and stored in variables. The field separator `FS` may be a regular expression, as may the third argument of `split()`.

Functions have been added. The declaration is

```
func name(arglist) { body }
```

Scalar arguments are passed by value, arrays by reference. Within the body, parameters are locals; all other variables are global.

```
return expr
```

returns a value to the caller; a plain `return` returns without a value, as does falling off the end.

`getline` for multiple input sources:

```
getline
```

sets `$0`, `NR`, `FNR`, `NF` from the next input record.

```
getline x
```

sets `x` from next input record, sets `NR` and `FNR`, but *not* `$0` and `NF`.

```
getline <"file"
```

sets `$0` from `file`, sets `NF`, but not `NR` or `FNR`.

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

sets `x` from `file`; it has no effect on `$0`, `NR`, `NF`, etc.

```
"command" | getline
```

is like `getline <"file"`, and

```
"command" | getline x
```

is like `getline x <"file"`.

Command-line arguments are accessible, in `ARGV[0]` ... `ARGV[ARGC-1]`. These may be altered or augmented at will; the remaining non-null arguments are used as the normal filenames.

New built-in functions include

```
close(filename)
rand(), srand(expr)
sin(expr), cos(expr), atan2(expr,expr)
sub(reg,repl,target), gsub(reg,repl,target)
system(command-line)
```

The exponentiation operator `^` and the corresponding assignment operator `^=` have been added.

The condition

```
i in array
```

tests whether `array` has a subscript of value `i` without creating it.

The `delete` statement deletes an array element.

The variable `FNR` is the record number in the current input file; the test `FNR==1` succeeds at the first record of each new file.

C string escapes like `\f`, `\b`, `\r`, and `\123` work as in C.

`BEGIN`, `END` and `func` declarations may be intermixed with other patterns in any order.

Source lines are now continued after commas, `||` and `&&`; other contexts still require an explicit `\`.

Limited Warranty

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, the Awk Development Task Force assumes no responsibility for their use by the recipient. Further, the Task Force assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

Index