Appendix A: Awk Reference Manual

This appendix explains, with examples, the constructs that make up Awk programs. Because it's a description of the complete language, the material is detailed, so we recommend that you skim it, then come back as necessary to check your understanding.

The first section describes patterns. The second section deals with actions: expressions, assignments, and control-flow statements. The remaining sections cover function definitions, output, input, and how Awk programs can call other programs.

Awk programs. The simplest Awk program is a sequence of pattern-action statements:

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

In some statements, the pattern may be missing; in others, the action and its enclosing braces may be missing. After Awk has checked your program to make sure there are no syntactic errors, it reads the input a line at a time, and for each input line, evaluates the patterns in order. For each pattern that matches the current input line, it executes the associated action. A missing pattern matches every input line, so every action with no pattern is performed on each line of input. A pattern-action statement consisting only of a pattern prints each input line matched by the pattern. The terms "input line" and "record" are used synonymously, though Awk also supports multiline records, where a record may contain several lines.

An Awk program is a sequence of pattern-action statements and function definitions. A function definition has the form

`function` *name* (*parameter-list*) { *statements* }

Pattern-action statements and function definitions are separated by newlines or semicolons and can be intermixed.

Statements are separated by newlines or semicolons or both.

The opening brace of an action must be on the same line as the pattern it accompanies; the remainder of the action, including the closing brace, may appear on the following lines.

Blank lines are ignored; they may be inserted before or after any statement to improve the readability of a program. Spaces and tabs may be inserted around operators and operands, again to enhance readability.

A semicolon by itself denotes the empty statement, as does `{}`.

A comment starts with the character `#` and ends at the end of the line, as in

```
{ print $1, $3 }     # name and population
```

Comments may appear at the end of any line.

Backslashes may be used to break statements across multiple lines.

In addition, a statement may be broken without a backslash after a comma, left brace, `&&`, `||`, `do`, `else`, and the closing right parenthesis in an `if`, `for`, or `while` statement.

A long statement may be spread over several lines by inserting a backslash and newline at each break:

```
{ print \
      $1,   # country name
      $2,   # area in thousands of square kilometers
      $3 }  # population in millions
```

As this illustrates, statements may be broken after commas, and a comment may be inserted at the end of each broken line.

In this book we have used several formatting styles, partly to illustrate different ones, and partly to keep programs short. For short programs, format doesn't much matter, but consistency and readability will help to keep longer programs manageable.

Commandlines. An Awk program is usually provided as a single argument on the commandline, or from a file named in a `-f` argument.

```
awk [-Fs] [-v var=value] 'program' optional list of filenames
awk [-Fs] [-v var=value] -f progfile optional list of filenames
```

Multiple `-f` options are allowed; the Awk program is created by combining these files in order. If the filename is `-`, the program is read from the standard input.

The option `-Fs` sets the field separator variable `FS` to *s*.

The option `--csv` causes input to be treated as comma-separated values.

An option of the form `-v` *var*=*value* sets the variable *var* to *value* before the Awk program begins execution. Any number of `-v` arguments are permitted.

The option `--version` prints the version identification of the specific Awk program and terminates.

All options must appear before a literal *program*. The special optional argument `--` marks the end of a list of optional arguments.

Command-line arguments are discussed further in Section A.5.5, below.

The Input File `countries`. As input for many of the Awk programs in the manual, we will use the `countries` file from Section 5.1. Each line contains the name of a country, its population in millions, its area in thousands of square kilometers, and the continent it is in. Values are from 2020; Russia has been arbitrarily placed in Europe. In the file, the four columns are separated by tabs; a single space separates `North` and `South` from `America`.

The file `countries` contains the following lines:

```
Russia      16376       145         Europe
China       9388        1411        Asia
USA         9147        331         North America
Brazil      8358        212         South America
India       2973        1380        Asia
Mexico      1943        128         North America
Indonesia   1811        273         Asia
Ethiopia    1100        114         Africa
Nigeria     910         206         Africa
Pakistan    770         220         Asia
Japan       364         126         Asia
Bangladesh  130         164         Asia
```

For the rest of the manual, the `countries` file is used when no input file is mentioned explicitly.

A.1 Patterns

Patterns control the execution of actions: when a pattern matches an input line, its associated action is executed. This section describes the types of patterns and the conditions under which they match.

Summary of Patterns

1. `BEGIN` { *statements* }
   The *statements* are executed once before any input has been read.
2. `END` { *statements* }
   The *statements* are executed once after all input has been read.
3. *expression* { *statements* }
   The *statements* are executed at each input line where the *expression* is true, that is, nonzero or non-null.
4. */regular expression/* { *statements* }
   The *statements* are executed at each input line that contains a string matched by the *regular expression.*
5. $pattern_1$ `,` $pattern_2$ { *statements* }
   A range pattern matches each input line from a line matched by $pattern_1$ to the next line matched by $pattern_2$, inclusive; the *statements* are executed at each matching line. Both matches can occur on the same line.

`BEGIN` and `END` do not combine with other patterns, but there may be multiple instances. `BEGIN` and `END` always require an action; the *statements* and enclosing braces may be omitted from all other patterns.

A range pattern cannot be part of any other pattern.

A.1.1 BEGIN and END

The BEGIN and END patterns do not match any input lines. Rather, the statements in the BEGIN action are executed after Awk has processed the command line, but before it reads any input; the statements in the END action are executed after all input has been read. BEGIN and END thus provide a way to gain control for initialization and wrapup. BEGIN and END do not combine with other patterns. If there is more than one BEGIN, the associated actions are executed in the order in which they appear in the program; the same is true for multiple END patterns. Although it's not required, we put BEGIN first and END last.

One common use of a BEGIN action is to change the default way that input lines are split into fields. The field separator is controlled by a built-in variable called FS. By default, fields are separated by sequences of spaces and/or tabs; this behavior occurs when FS is set to a space.

If the command-line argument --csv is used, the input is treated as comma-separated values (CSV) format. Input fields are separated by commas, independent of the value of FS. Fields may be quoted with double-quote characters ". Such quoted fields may contain commas and double quotes, which are represented as ""; that is, two adjacent quotes are a literal quote. See Section A.5.2 for more details.

Setting FS to any character other than a space makes that character the field separator. A multi-character field separator is interpreted as a regular expression, as discussed below.

The following program uses the BEGIN action to set the field separator to a tab character (\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 aligned under the column headings. The END action prints the totals. (Variables and expressions are discussed in Section A.2.1.)

```
# print countries with column headers and totals

BEGIN { FS = "\t"   # make tab the field separator
        printf("%12s %6s %5s   %s\n\n",
              "COUNTRY", "AREA", "POP", "CONTINENT")
      }
      { printf("%12s %6d %5d   %s\n", $1, $2, $3, $4)
        area += $2
        pop += $3
      }
END   { printf("\n%12s %6d %5d\n", "TOTAL", area, pop) }
```

With the countries file as input, this program produces

```
   COUNTRY   AREA   POP   CONTINENT

    Russia  16376   145   Europe
     China   9388  1411   Asia
       USA   9147   331   North America
    Brazil   8358   212   South America
     India   2973  1380   Asia
    Mexico   1943   128   North America
 Indonesia   1811   273   Asia
  Ethiopia   1100   114   Africa
   Nigeria    910   206   Africa
  Pakistan    770   220   Asia
     Japan    364   126   Asia
Bangladesh    130   164   Asia

     TOTAL  53270  4710
```

A.1.2 Expression Patterns

Like most programming languages, Awk is rich in expressions for describing numeric computations, but it also has expressions for describing operations on strings. The term *string* means a sequence of zero or more characters represented in UTF-8. These may be stored in variables, or appear literally as string constants like `""`, `"Asia"`, `"にほんで"` and `"☹☺"`.

A *substring* is a contiguous sequence of zero or more characters within a string. The string `""`, which contains no characters, is called the *null* or *empty string*. In every string, the null string appears as a substring of length zero before the first character, between every pair of adjacent characters, and after the last character.

Any expression can be used as an operand of any operator. If an expression has a numeric value but an operator requires a string value, the numeric value is automatically transformed into a string; similarly, a string is converted into a number when an operator requires a numeric value. Type conversions and coercions are discussed in detail in Section A.2.2 below.

Any expression can be used as a pattern. If an expression used as a pattern has a nonzero or nonnull value at the current input line, then the pattern matches that line. The typical expression patterns are those involving comparisons between numbers or strings. A comparison expression contains one of the six relational operators, or one of the two string-matching operators `~` (tilde) and `!~` that will be discussed in the next section. These operators are listed in Table A-1.

TABLE A-1. COMPARISON OPERATORS

OPERATOR	MEANING
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matched by
!~	not matched by

If the pattern is a comparison expression like `NF > 10`, then it matches the current input line when the condition is satisfied, that is, when the number of fields in the line is greater than 10. If the pattern is an arithmetic expression like `NF`, it matches the current input line when its numeric value is nonzero. If the pattern is a string expression, it matches the current input line when the string value of the expression is nonnull.

In a relational comparison, if both operands are numeric, a numeric comparison is made; otherwise, any numeric operand is converted to a string, and then the operands are compared as strings. The strings are compared character by character using UTF-8 ordering. One string is ‘‘less than’’ another if it would appear before the other according to this ordering, for example, `"India" < "Indonesia"` and `"Asia" < "Asian"`. Comparisons are case-sensitive: `"A"` and `"Z"` both precede `"a"`.

The pattern

```
$3/$2 > 0.5
```

selects lines where the value of the third field divided by the second is greater than 0.5, that is, where the population density is greater than 500 people per square kilometer, while

```
$0 >= "M"
```

selects lines that begin with an `M`, `N`, `O`, etc.:

```
Russia      16376       145         Europe
USA         9147        331         North America
Mexico      1943        128         North America
Nigeria     910         206         Africa
Pakistan    770         220         Asia
```

Note that this also matches lines that begin with any character past `M`, which among other things includes any lower-case letter.

Sometimes the type of a comparison operator cannot be determined solely by the syntax of the expression in which it appears. The program

```
$1 < $4
```

could compare the first and fourth fields of each input line either as numbers or as strings. Here, the type of the comparison depends on the values of the fields, and it may vary from line to line. In the `countries` file, the first and fourth fields are always strings, so string comparisons are always made; the output is

```
Brazil      8358        212         South America
Mexico      1943        128         North America
```

As with all comparisons, the comparison is done numerically only if both fields are numbers; this would be the case with

```
$2 < $3
```

on the same data.

Section A.2.2 contains a complete discussion of strings, numbers, expressions, and coercions.

A compound pattern is an expression that combines other patterns, using parentheses and the logical operators `||` (OR), `&&` (AND), and `!` (NOT). A compound pattern matches the current input line if the expression evaluates to true. The following program uses the AND operator to select all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The program

```
$4 == "Asia" || $4 == "Europe"
```

uses the OR operator to select lines with either Asia or Europe as the fourth field. As we will see in a moment, because the latter query is a test on string values, another way to write it is to use a regular expression with the alternation operator `|`:

```
$4 ~ /^(Asia|Europe)$/
```

Two regular expressions are *equivalent* if they match the same strings. Test your understanding of the precedence rules for regular expressions: Are the two regular expressions `^Asia|Europe$` and `^(Asia|Europe)$` equivalent?

If there are no occurrences of `Asia` or `Europe` in other fields, this pattern could also be written as

```
/Asia/ || /Europe/
```

or even

```
/Asia|Europe/
```

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

A.1.3 Regular Expression Patterns

Awk provides a notation called *regular expressions* for specifying and matching strings of characters. Regular expressions are widely used throughout Unix, where restricted forms of regular expressions use "wild-card characters" for specifying sets of filenames. Regular expressions are also supported by text editors, and today are part of most programming languages, either directly in the syntax (as in Awk) or by libraries (as in Python).

A *regular expression pattern* tests whether a string contains a substring matched by a regular expression. In this section, we will discuss the most basic kinds of regular expressions and show how they appear in patterns; a detailed description of regular expressions follows in the next section.

Summary of Regular Expression Patterns

/regexpr/
Matches when the current input line contains a substring matched by *regexpr*.

expression ~ */regexpr/*
Matches if the string value of *expression* contains a substring matched by *regexpr*.

expression !~ */regexpr/*
Matches if the string value of *expression* does not contain a substring matched by *regexpr*.

Any expression may be used in place of */regexpr/* in the context of ~ and !~. It is evaluated and then interpreted as a regular expression.

The simplest regular expression is a string of letters and numbers, like `Asia`, that matches itself. To turn a regular expression into a string-matching pattern, enclose it in slashes:

```
/Asia/
```

This pattern matches when the current input line contains the substring `Asia`, either as `Asia` by itself or as some part of a larger word like `Asian` or `Pan-Asiatic`. Note that spaces are significant within regular expressions: the string-matching pattern

```
/ Asia /
```

matches only when `Asia` is surrounded by spaces and thus matches no lines in `countries`.

The pattern above is one of three types of string-matching patterns. Its form is a regular expression *r* enclosed in slashes:

/r/

This pattern matches an input line if the line contains a substring matched by *r*.

The other two types of string-matching patterns use an explicit match operator:

expression ~ */r/*
expression !~ */r/*

The match operator ~ means "is matched by" and !~ means "is not matched by." The first pattern matches when the string value of *expression* contains a substring matched by the regular expression *r*; the second pattern matches if there is no such substring.

The left operand of a match operator is often a field: the pattern

```
$4 ~ /Asia/
```

matches all input lines in which the fourth field contains `Asia` as a substring, while

```
$4 !~ /Asia/
```

matches if the fourth field does *not* contain `Asia` anywhere.

Note that the string-matching pattern `/Asia/` is a shorthand for `$0 ~ /Asia/`.

A.1.4 Regular Expressions in Detail

A regular expression is a notation for specifying and matching strings. Like an arithmetic expression, a regular expression is a basic expression or one created by applying operators to

component expressions. To understand the strings matched by a regular expression, we need to understand the strings matched by its components.

Summary of Regular Expressions

The regular expression metacharacters are:

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

A basic regular expression is one of the following:

- a nonmetacharacter, such as `A`, that matches itself.
- an escape sequence that matches a special symbol: e.g., `\t` matches a tab (see Table A-2).
- a quoted metacharacter, such as `\*`, that matches the metacharacter literally.
- `^`, which matches the beginning of a string.
- `$`, which matches the end of a string.
- `.`, which matches any single character.
- a character class: `[ABC]` matches any of the characters `A`, `B`, or `C`.

Character classes may include abbreviations: `[0-9]` matches any single digit, `[A-Za-z]` matches any single letter in either case, `[[:`*class*`:]]` matches any character in the *class*, which may be `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, or `xdigit` (hexadecimal digit).

Character classes may be complemented, to match any character not in the class: `[^0-9]` matches any character except a digit; `[^[:cntrl:]]` matches any non-control character.

These operators combine regular expressions into larger ones.

$r_1 \mid r_2$	alternation: matches any string matched by r_1 or r_2
$r_1 r_2$	concatenation: matches xy where r_1 matches x and r_2 matches y
$r*$	matches zero or more consecutive strings matched by r
$r+$	matches one or more consecutive strings matched by r
$r?$	matches the null string or one string matched by r
$r\{m,n\}$	between m and n instances of r; n is optional
(r)	grouping: matches the same strings as r

The operators are listed in order of increasing precedence. Redundant parentheses in regular expressions may be omitted as long as the precedence of operators is respected.

Metacharacters. Most characters in a regular expression match literal occurrences of themselves in the text, so a regular expression consisting of a single character like a letter or digit is a basic regular expression that matches itself.

However, the regular expression mechanism uses some characters to indicate a meaning other than their literal value. The characters

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

are called *metacharacters* because they have special meanings as discussed below.

To preserve the literal meaning of a metacharacter in a regular expression, precede it by a backslash: the regular expression `\$` matches the character `$`. If a character is preceded by a single `\`, we say that the character is *quoted*.

In a regular expression, an unquoted caret `^` matches the beginning of a string, an unquoted dollar sign `$` matches the end of a string, and an unquoted period `.` matches any single character. Thus,

`^C`	matches a `C` at the beginning of a string; no special meaning elsewhere
`C$`	matches a `C` at the end of a string; no special meaning elsewhere
`^C$`	matches the string consisting of the single character `C`
`^.$`	matches any string containing exactly one character
`^...$`	matches any string containing exactly three characters
`...`	matches any three consecutive characters
`\.$`	matches a period at the end of a string

Character classes. A regular expression consisting of a group of characters enclosed in brackets is called a *character class*; it matches any one of the enclosed characters. For example, `[AEIOU]` matches any of the characters `A`, `E`, `I`, `O`, or `U`.

Ranges of characters can be abbreviated in a character class by using a hyphen. The character immediately to the left of the hyphen defines the beginning of the range; the character immediately to the right defines the end. Thus, `[0-9]` matches any digit, and `[a-zA-Z][0-9]` matches a letter followed by a digit. Without both a left and right operand, a hyphen in a character class denotes itself, so the character classes `[+-]` and `[-+]` match either a `+` or a `-`. The character class `[A-Za-z-]+` matches words that include hyphens.

Ranges of Unicode characters work so long as the range is of manageable size, roughly 256 characters. In general, if the character set in question fits on a single page in the Unicode descriptions at `unicode.org`, a range will work; for example, the character class `[ァ-チ]` matches Japanese Katakana characters.

Special character classes like `[:alpha:]` match any one of a range of characters defined by the local environment, as set by the `LOCALE` shell variable. This enables some language-independent character-class matching. For example, if the locale is set to the value `LC_ALL=fr_FR.UTF-8`, then the regular expression `[[:alpha:]]` matches accented letters like é and à , while in the locale `en_EN`, it does not.

Complemented character classes. A *complemented* character class is one in which the first character after the `[` is a `^`. Such a class matches any character *not* in the group following the caret. Thus, `[^0-9]` matches any character except a digit; `[^a-zA-Z]` matches any character except an upper or lower-case letter.

`^[ABC]`	matches an `A`, `B`, or `C` at the beginning of a string
`^[^ABC]`	matches any character at the beginning of a string, except `A`, `B`, or `C`
`[^ABC]`	matches any character other than an `A`, `B`, or `C`
`^[^a-z]$`	matches any single-character string, except a lower-case letter
`^[^[:lower:]]$`	also matches any single-character string, except a lower-case letter

Inside a character class, all characters have their literal meaning, except for the quoting character `\`, `^` at the beginning, and `-` between two characters. Thus, `[.]` matches a period and `^[^^]` matches any character except a caret at the beginning of a string.

Grouping. Parentheses are used in regular expressions to specify how components are grouped. There are two binary regular expression operators: alternation and concatenation. The alternation operator `|` is used to specify alternatives: if r_1 and r_2 are regular expressions, then r_1 `|` r_2 matches any string matched by r_1 or by r_2.

There is no explicit concatenation operator. If r_1 and r_2 are regular expressions, then `(`r_1`)(`r_2`)` (with no space between `(`r_1`)` and `(`r_2`)`) matches any string of the form xy where r_1 matches x and r_2 matches y. The parentheses around r_1 or r_2 can be omitted if the contained regular expression does not contain the alternation operator. The regular expression

```
(Asian|European|North American) (male|female) (black|blue)bird
```

matches twelve strings ranging from

```
Asian male blackbird
```

to

```
North American female bluebird
```

Repetitions. The symbols `*`, `+`, and `?` are unary operators used to specify repetitions in regular expressions. If r is a regular expression, then (r)`*` matches any string consisting of zero or more consecutive substrings matched by r; (r)`+` matches any string consisting of one or more consecutive substrings matched by r; and (r)`?` matches zero or one instances of r, that is the null string or any string matched by r.

The notation `(`r`){`m`,`n`}` specifies a match of between m and n (inclusive) occurrences of the preceding regular expression; if `,`n is omitted, the pattern matches exactly m occurrences.

If r is a basic regular expression, parentheses can be omitted.

`B*`	matches the null string or `B` or `BB`, and so on
`AB*C`	matches `AC` or `ABC` or `ABBC`, and so on
`AB+C`	matches `ABC` or `ABBC` or `ABBBC`, and so on
`ABB*C`	also matches `ABC` or `ABBC` or `ABBBC`, and so on
`AB?C`	matches `AC` or `ABC`
`[A-Z]+`	matches any string of one or more upper-case letters
`(AB)+C`	matches `ABC`, `ABABC`, `ABABABC`, and so on
`X(AB){1,2}Y`	matches `XABY`, `XABABY`, but not `XABABABY` and so on

In regular expressions, the alternation operator `|` has the lowest precedence, then concatenation, and finally the repetition operators `*`, `+`, `?` and `{}`. As in arithmetic expressions, operators of higher precedence are evaluated before lower ones. These conventions allow parentheses to be omitted: `ab|cd` is the same as `(ab)|(cd)`, and `^ab|cd*e$` is the same as `(^ab)|(c(d*)e$)`.

Escapes in regular expressions and strings. Within regular expressions and strings, Awk uses certain character sequences, called *escape sequences*, to specify characters for which there may be no other notation. For example, `\n` stands for a newline character, which cannot otherwise appear in a string or regular expression; `\b` stands for backspace; `\t` stands for tab; and `\/` represents a slash. Any arbitrary value can be entered with an octal or hexadecimal escape: `\033` and `\0x1b` both represent the ASCII escape character. An arbitrary Unicode character can be entered as `\u`*h...*, where *h...* is a sequence of up to 8 hexadecimal digits that represent a valid Unicode character; for example, the character ☺ is `\u1F642`.

It's important to note that such escape sequences have special meaning only within an Awk *program*; in data, they are just characters. The complete list of escape sequences is shown in Table A-2.

Examples. To finish our discussion of regular expressions, here are some examples of useful string-matching patterns containing regular expressions with unary and binary operators,

TABLE A-2. ESCAPE SEQUENCES

SEQUENCE	MEANING
`\a`	alarm (bell)
`\b`	backspace
`\f`	formfeed
`\n`	newline (line feed)
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\`*ddd*	octal value *ddd*; *ddd* is 1 to 3 digits between 0 and 7
`\x`*hh*	hexadecimal value; *hh* is 1 or 2 hexadecimal digits in upper or lower case
`\u`*h...*	Unicode value; *h...* is up to 8 hexadecimal digits in upper or lower case
`\`*c*	any other character *c* literally, e.g., `\"` for `"` and `\\` for `\`

along with a description of the kinds of input lines they match. Recall that a string-matching pattern */r/* matches the current input line if the line contains at least one substring matched by *r*.

```
/^[0-9]+$/
```
matches any input line that consists of one or more decimal digits

```
/^[0-9][0-9][0-9]$/
```
exactly three digits

```
/^[0-9]{3}$/
```
also exactly three digits

```
/^(\+|-)?[0-9]+\.?[0-9]*$/
```
a decimal number with an optional sign and optional fraction

```
/^[+-]?[0-9]+[.]?[0-9]*$/
```
also a decimal number with an optional sign and optional fraction

```
/^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE][+-]?[0-9]+)?$/
```
a floating point number with optional sign and optional exponent

```
/^[A-Za-z_][A-Za-z_0-9]*$/
```
a letter or underscore followed by any letters, underscores, or digits (e.g., a variable name)

```
/^[A-Za-z]$|^[A-Za-z][0-9]$/
```
a letter or a letter followed by a digit

```
/^[A-Za-z][0-9]?$/
```
also a letter or a letter followed by a digit

Since `+` and `.` are metacharacters, they have to be preceded by backslashes in the fourth example to match literal occurrences. These backslashes are not needed within character classes, so the fifth example shows an alternate way to describe the same numbers.

Any regular expression enclosed in slashes can be used as the right-hand operand of a matching operator: the program

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

prints all lines in which the second field is not a string of digits.

Table A-3 summarizes regular expressions and the strings they match. The operators are listed in order of increasing precedence. Characters are Unicode code points.

TABLE A-3. REGULAR EXPRESSIONS

EXPRESSION	MATCHES
c	the nonmetacharacter c
\c	escape sequence or literal character c
^	beginning of string
$	end of string
.	any character
[$c_1c_2\ldots$]	any character in $c_1c_2\ldots$
[^$c_1c_2\ldots$]	any character not in $c_1c_2\ldots$
[c_1-c_2]	any character in the range beginning with c_1 and ending with c_2
[^c_1-c_2]	any character not in the range c_1 to c_2
r_1 | r_2	any string matched by r_1 or r_2
(r_1)(r_2)	any string xy where r_1 matches x and r_2 matches y; parentheses are not needed around subexpressions with no alternations
(r)*	zero or more consecutive strings matched by r
(r)+	one or more consecutive strings matched by r
(r)?	zero or one string matched by r
(r){m,n}	m through n consecutive strings matched by r; n may be omitted; parentheses are not needed around basic regular expressions
(r)	any string matched by r

A.1.5 Range Patterns

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

pat_1, pat_2

A range pattern matches each line between an occurrence of pat_1 and the next occurrence of pat_2 inclusive; pat_2 may match the same line as pat_1, making the range a single line.

Matching begins whenever the first pattern of a range matches; if no instance of the second pattern is subsequently found, then all lines to the end of the input are matched:

```
/Europe/, /Africa/
```

prints

```
Russia      16376       145         Europe
China       9388        1411        Asia
USA         9147        331         North America
Brazil      8358        212         South America
India       2973        1380        Asia
Mexico      1943        128         North America
Indonesia   1811        273         Asia
Ethiopia    1100        114         Africa
```

FNR is the number of the line just read from the current input file and FILENAME is the filename itself; both are built-in variables. Thus, the program

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

prints the first five lines of each input file with the filename prefixed. Alternatively, this program could be written as

```
FNR <= 5 { print FILENAME ": " $0 }
```

A range pattern cannot be part of any other pattern.

A.2 Actions

In a pattern-action statement, the pattern determines when the action is executed. Sometimes an action is simple: a single print or assignment. Other times, it may be a sequence of several statements separated by newlines or semicolons. This section begins the description of actions by discussing expressions and control-flow statements. The following sections present user-defined functions, and statements for input and output.

Summary of Actions

The statements in actions can include:

expressions, with constants, variables, assignments, function calls, etc.
`print` *expression-list*
`printf(`*format*`,` *expression-list*`)`
`if (`*expression*`)` *statement*
`if (`*expression*`)` *statement* `else` *statement*
`while (`*expression*`)` *statement*
`for (`*expression*`;` *expression*`;` *expression*`)` *statement*
`for (`*variable* `in` *array*`)` *statement*
`do` *statement* `while (`*expression*`)`
`break`
`continue`
`next`
`nextfile`
`exit`
`exit` *expression*
`{` *statements* `}`

A.2.1 Expressions

We begin with expressions, because expressions are the simplest statements, and most other statements are made up of expressions of various kinds. An expression is formed by combining primary expressions and other expressions with operators. The primary expressions are the primitive building blocks: they include constants, variables, array references, function invocations, and various built-ins, like field names.

Our discussion of expressions starts with constants and variables. Then come the operators that can be used to combine expressions. These operators fall into five categories: arithmetic, comparison, logical, conditional, and assignment. The built-in arithmetic and string functions come next, followed at the end of the section by the description of arrays.

Constants. There are two types of constants, string and numeric. A string constant is created by enclosing a sequence of characters in quotation marks, as in `"hello, world"` or `"Asia"` or `""`. String constants may contain the escape sequences listed in Table A-2. A long string can be split into multiple lines with backslashes:

```
s = "a really very long \
string split over two lines"
```

The newline that follows the backslash is removed; it is not part of the string, so the result is equivalent to

```
s = "a really very long string split over two lines"
```

Any spaces at the beginning of the continuation are included.

A numeric constant can be an integer like `1127`, a decimal number like `3.14`, or a number in scientific (exponential) notation like `6.022E+23`. Different representations of the same number have the same numeric value: the numbers `1e6`, `1.00E6`, `10e5`, `0.1e7`, and `1000000` are numerically equal.

All numbers are stored in double-precision floating point, the precision of which is machine dependent, though usually about 15 decimal digits.

Two special numeric values are recognized, `"+nan"` and `"+inf"`, which represent NaN, the "not a number" value, and infinity. These must include an explicit + or − sign both as literals in a program and as data input.

The names are not case sensitive, so `NaN` and `Inf` are also valid.

The `nan` and `inf` values can be generated by arithmetic expressions; for example.

```
$ awk '{print "   " $1/$2}'
1 2
   0.5
1 +nan
   +nan
+nan 1
   +nan
+nan +nan
   +nan
+nan −inf
   +nan
+inf +inf
   −nan
0 +inf
   0
+inf 0
awk: division by zero
 input record number 7, file
 source line number 1
```

Variables. Expressions can contain several kinds of variables: user-defined, built-in, and fields. Variable names are sequences of letters, digits, and underscores that do not begin with a digit; all built-in variables have all-upper-case names.

A variable has a value that is a string or a number or both. Since the type of a variable is not declared, Awk infers the type from context. When necessary, Awk will convert a string value into a numeric one, or vice versa. For example, in

```
$4 == "Asia" { print $1, 1000 * $2 }
```

`$2` is converted into a number if it is not one already, and `$1` and `$4` are converted into strings if they are not already.

An uninitialized variable has the string value `""` (the null string) and the numeric value 0.

Built-In Variables. Table A-4 lists the built-in variables. These variables can be used in all expressions, and may be reset by the user. `FILENAME` is set each time a new file is read. `FNR`, `NF`, and `NR` are set each time a new record is read; additionally, `NF` is reset when `$0` changes or when a new field is created. Conversely, if `NF` changes, `$0` is recomputed when its value is needed. The variables `RLENGTH` and `RSTART` change as a result of invoking the `match` function.

TABLE A-4. BUILT-IN VARIABLES

VARIABLE	MEANING	DEFAULT
`ARGC`	number of command-line arguments, including command name	-
`ARGV`	array of command-line arguments, numbered 0..`ARGC-1`	-
`CONVFMT`	conversion format for numbers	`"%.6g"`
`ENVIRON`	array of shell environment variables	-
`FILENAME`	name of current input file	-
`FNR`	record number in current file	-
`FS`	input field separator	`" "`
`NF`	number of fields in current record	-
`NR`	number of records read so far	-
`OFMT`	output format for numbers	`"%.6g"`
`OFS`	output field separator for `print`	`" "`
`ORS`	output record separator for `print`	`"\n"`
`RLENGTH`	length of string matched by `match` function	-
`RS`	input record separator	`"\n"`
`RSTART`	start of string matched by `match` function	-
`SUBSEP`	subscript separator	`"\034"`

Field Variables. The fields of the current input line are called `$1`, `$2`, through `$NF`; `$0` refers to the whole line. Fields share the properties of other variables — they may be used in arithmetic or string operations, and they may be assigned to. Thus one can divide the second field in each line of `countries` by 1000 to express areas in millions of square kilometers instead of thousands:

```
{ $2 = $2 / 1000; print }
```

One can assign a new string to a field:

```
BEGIN                   { FS = OFS = "\t" }
$4 == "North America"   { $4 = "NA" }
$4 == "South America"   { $4 = "SA" }
                        { print }
```

In this program, the `BEGIN` action sets `FS`, the variable that controls the input field separator, and `OFS`, the output field separator, both to a tab. The `print` statement in the fourth line

prints the value of `$0` after it has been modified by previous assignments. When `$0` is changed by assignment or substitution, `$1`, `$2`, etc., and `NF` will all be recomputed; likewise, when one of `$1`, `$2`, etc., is changed, `$0` is reconstructed using `OFS` to separate fields.

Fields can also be specified by expressions. For example, `$(NF-1)` is the next-to-last field of the current line. The parentheses are needed: `$NF-1` is one less than the numeric value of the last field.

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

```
BEGIN  { FS = OFS = "\t" }
       { $5 = 1000 * $3 / $2; print }
```

Any intervening fields are created when necessary and given null values.

The number of fields can vary from line to line.

Summary of Expressions

The primary expressions are:

numeric and string constants, variables, fields, function calls, array elements.

These operators combine expressions:

assignment operators = += -= *= /= %= ^=
conditional expression operator ?:
logical operators || (OR), && (AND), ! (NOT)
matching operators ~ and !~
relational operators < <= == != > >=
concatenation (no explicit operator)
arithmetic operators + - * / % ^
unary + and -
increment and decrement operators ++ and -- (prefix and postfix)
parentheses for grouping

Assignment Operators. There are seven assignment operators that can be used in expressions called *assignments*. The simplest assignment is an expression of the form

var = expr

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

```
$4 == "Asia" { pop = pop + $3; n = n + 1 }
END          { print "Total population of the", n,
                     "Asian countries is", pop, "million."
             }
```

Applied to `countries`, the program produces

```
Total population of the 6 Asian countries is 3574 million.
```

The first action contains two assignments, one to accumulate population, and the other to

count countries. The variables are not explicitly initialized, yet everything works properly because each variable is initialized by default to the string value `""` and the numeric value `0`.

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

```
$3 > maxpop  { maxpop = $3; country = $1 }

END          { print "country with largest population:",
                    country, maxpop
             }
```

The result:

```
country with largest population: China 1411
```

Note, however, that this program is correct only when at least one value of `$3` is positive.

The other six assignment operators are `+=`, `-=`, `*=`, `/=`, `%=`, and `^=`. Their meanings are similar: *v op= e* has the same effect as *v = v op e*. The assignment

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

can be written more concisely using the assignment operator `+=`:

```
pop += $3
```

This statement 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 more compact and often clearer. In addition, *v* is evaluated only once so a complicated computation like

```
v[substr($0,index($0,"!")+1)] += 2
```

will run faster.

As another example,

```
{ $2 /= 1000; print }
```

divides the second field by 1000, then prints the line.

An assignment is an expression; its value is the new value of the left side. Thus assignments can be used inside any expression. In the multiple assignment

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

both the field separator and the output field separator are set to tab. Assignment expressions are also common within tests, such as:

```
if ((n = length($0)) > 0) ...
```

though this kind of use can be confusing. Don't forget the parentheses.

Conditional Expression Operator. A conditional expression has the form

$expr_1$ `?` $expr_2$ `:` $expr_3$

First, $expr_1$ is evaluated. If it is true, that is, nonzero or nonnull, the value of the conditional expression is the value of $expr_2$; otherwise, it is the value of $expr_3$. Only one of $expr_2$ and $expr_3$ is evaluated.

The following program uses a conditional expression to print the reciprocal of `$1`, or a warning if `$1` is zero:

```
{ print ($1 != 0 ? 1/$1 : "$1 is zero, line " NR) }
```

As with nested assignments, conditional expressions can be abused to create inscrutable code.

Logical Operators. The logical operators `&&` (AND), `||` (OR), and `!` (NOT) are used to create logical expressions by combining other expressions. A logical expression has the value 1 if it is true and 0 if false. In the evaluation of a logical operator, an operand with a nonzero or nonnull value is treated as true; other values are treated as false. The operands of expressions separated by `&&` or `||` are evaluated from left to right, and evaluation ceases as soon as the value of the complete expression can be determined. This means that in

expr$_1$ `&&` *expr*$_2$

expr$_2$ is not evaluated if *expr*$_1$ is false, while in

expr$_3$ `||` *expr*$_4$

expr$_4$ is not evaluated if *expr*$_3$ is true.

Newlines may be inserted after the `&&` and `||` operators.

The precedence of `&&` is higher than `||`, so an expression like

```
A && B || C && D
```

is parsed as

```
(A && B) || (C && D)
```

Parentheses should be used to make sure such expressions are clear to the reader.

Relational Operators. Relational or comparison expressions are those containing either a relational operator or a regular expression matching operator. The relational operators are <, <=, == (equals), != (not equals), >=, and >. The regular expression matching operators are ~ (is matched by) and !~ (is not matched by).

The value of a comparison expression is 1 if it is true and 0 otherwise. Similarly, the value of a matching expression is 1 if true, 0 if false, so

```
$4 ~ /Asia/
```

is 1 if the fourth field of the current line contains `Asia` as a substring, or 0 if it does not.

Arithmetic Operators. Awk provides the usual `+`, `-`, `*`, `/`, `%`, and `^` arithmetic operators. The `%` operator computes remainders: `x%y` is the remainder when `x` is divided by `y`; its behavior depends on the particular computer if `x` or `y` is negative. The `^` operator is exponentiation: `x^y` is x^y. Note that `^` has a different meaning (bitwise exclusive OR) in C and many other languages.

All arithmetic is done in double-precision floating point, which is usually about 15 decimal digits.

Unary Operators. The unary operators are + and −, with the obvious meanings.

Increment and Decrement Operators. The assignment `n = n + 1` is usually written `++n` or `n++` using the unary increment operator ++, which adds 1 to a variable. The prefix form `++n` increments `n` before delivering its value; the postfix form `n++` increments `n` after delivering its value. This makes a difference when ++ is used in an assignment. If `n` is initially 1, then the assignment `i = ++n` increments `n` and assigns the new value 2 to `i`, while the assignment `i = n++` increments `n` but assigns the old value 1 to `i`. To just increment `n`, however,

there's no difference between `n++` and `++n`. The prefix and postfix decrement operator `--`, which subtracts 1 from a variable, works the same way.

Built-In Arithmetic Functions. The built-in arithmetic functions are shown in Table A-5. These functions can be used as primary expressions in all expressions. In the table, x and y are arbitrary expressions.

TABLE A-5. 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, e^x
`int(`x`)`	integer part of x; truncated towards 0
`log(`x`)`	natural (base e) logarithm of x
`rand()`	random number r, where $0 \le r < 1$
`sin(`x`)`	sine of x, with x in radians
`sqrt(`x`)`	square root of x
`srand(`x`)`	x is new seed for `rand()`; use time of day if x is omitted; return previous seed

Useful constants can be computed with these functions: `atan2(0,-1)` gives π and `exp(1)` gives e, the base of the natural logarithms. To compute the base-10 logarithm of x, use `log(`x`)/log(10)`.

The function `rand()` returns a pseudo-random floating point number greater than or equal to 0 and less than 1. Calling `srand(`x`)` sets the starting seed of the generator from x and returns the previous seed. Calling `srand()` sets the starting point from the time of day. If `srand` is not called, `rand` starts with the same value each time the program is run.

The assignment

```
randint = int(n * rand()) + 1
```

sets `randint` to a random integer between 1 and `n` inclusive, using the `int` function to discard the fractional part. The assignment

```
x = int(x + 0.5)
```

rounds the value of `x` to the nearest integer when `x` is positive.

String Operators. There is only one string operation, concatenation. It has no explicit operator: string expressions are created by writing constants, variables, fields, array elements, function values, and other expressions next to one another. The program

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

prints each line preceded by its line number and a colon, with no spaces. The number `NR` is converted to its string value (and so is `$0` if necessary); then the three strings are concatenated and the result is printed.

Strings as Regular Expressions. So far, in all of our examples of matching expressions, the right-hand operand of `~` and `!~` has been a regular expression enclosed in slashes. But in fact any expression can be used as the right operand of these operators. Awk evaluates the

expression, converts the value to a string if necessary, and interprets the string as a regular expression. For example, the program

```
BEGIN { digits = "^[0-9]+$" }
$2 ~ digits
```

will print all lines in which the second field is a string of digits.

Since expressions can be concatenated, a regular expression can be built up from components. The following program echoes input lines that are valid floating point numbers:

```
BEGIN {
    sign = "[+-]?"
    decimal = "[0-9]+[.]?[0-9]*"
    fraction = "[.][0-9]+"
    exponent = "([eE]" sign "[0-9]+)?"
    number = "^" sign "(" decimal "|" fraction ")" exponent "$"
}
$0 ~ number
```

In a matching expression, a quoted string like `"^[0-9]+$"` can normally be used interchangeably with a regular expression enclosed in slashes, such as `/^[0-9]+$/`. There is one exception, however. If the string in quotes is to match a literal occurrence of a regular expression metacharacter, one extra backslash is needed to protect the protecting backslash itself. That is,

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

and

```
$0 ~ "(\\+|-)[0-9]+"
```

are equivalent.

This behavior may seem arcane, but it arises because one level of protecting backslashes is removed when a quoted string in a program is parsed by Awk. If a backslash is needed in front of a metacharacter to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string. If the right operand of a matching operator is a variable or field, as in

```
x ~ $1
```

then the additional level of backslashes is not needed in the first field because backslashes have no special meaning in data.

As an aside, it's easy to test your understanding of regular expressions interactively: the program

```
$1 ~ $2
```

lets you type in a string and a regular expression; it echoes the line back if the string matches the regular expression.

Built-In String Functions. Awk provides the built-in string functions shown in Table A-6. In this table, *r* represents a regular expression (either as a string or enclosed in slashes), *s* and *t* are string expressions, and *n* and *p* are integers. Strings are represented as UTF-8 characters.

The arguments of a function call are all evaluated before the function is called, but the order of evaluation is unspecified.

TABLE A-6. BUILT-IN STRING FUNCTIONS

FUNCTION	DESCRIPTION
`gsub(`*r*`,`*s*`)`	substitute *s* for *r* globally in `$0`, return number of substitutions made
`gsub(`*r*`,`*s*`,`*t*`)`	substitute *s* for *r* globally in string *t*, return number of substitutions made
`index(`*s*`,`*t*`)`	return first position of string *t* in *s*, or 0 if *t* is not present
`length(`*s*`)`	return number of Unicode characters in *s*; return number of elements if *s* is an array
`match(`*s*`,`*r*`)`	test whether *s* contains a substring matched by *r*; return index or 0; sets `RSTART` and `RLENGTH`
`split(`*s*`,`*a*`)`	split *s* into array *a* on `FS` or as CSV if `--csv` is set, return number of elements in *a*
`split(`*s*`,`*a*`,`*fs*`)`	split *s* into array *a* on field separator *fs*, return number of elements in *a*
`sprintf(`*fmt*`,`*expr-list*`)`	return *expr-list* formatted according to format string *fmt*
`sub(`*r*`,`*s*`)`	substitute *s* for the leftmost longest substring of `$0` matched by *r*; return number of substitutions made
`sub(`*r*`,`*s*`,`*t*`)`	substitute *s* for the leftmost longest substring of *t* matched by *r*; return number of substitutions made
`substr(`*s*`,`*p*`)`	return suffix of *s* starting at position *p*
`substr(`*s*`,`*p*`,`*n*`)`	return substring of *s* of length at most *n* starting at position *p*
`tolower(`*s*`)`	return *s* with upper case ASCII letters mapped to lower case
`toupper(`*s*`)`	return *s* with lower case ASCII letters mapped to upper case

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, so

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

returns 2.

The function `match(`*s*`,`*r*`)` finds the leftmost longest substring in the string *s* that is matched by the regular expression *r*. It returns the index where the substring begins or 0 if there is no matching substring. It also sets the built-in variables `RSTART` to this index and `RLENGTH` to the length of the matched substring.

The function `split(`*s*`,`*a*`,`*fs*`)` splits the string *s* into the array *a* according to the separator *fs* and returns the number of elements. It is described after arrays, at the end of this section.

The string 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 value of the expression *format*. 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. Section A.4.3 contains a complete

description of the `printf` format-conversion characters.

The functions `sub` and `gsub` are patterned after the substitute command in the Unix text editor `ed`. The function `sub(`*r*`,`*s*`,`*t*`)` first finds the leftmost longest substring matched by the regular expression *r* in the target string *t*, which must be a variable, field, or array element; it then replaces the substring by the substitution string *s*. As in most text editors, ''leftmost longest'' means that the leftmost match (that is, the first match) is found first, then extended as far as possible.

In the target string `banana`, for example, `anan` is the leftmost longest substring matched by the regular expression `(an)+`. By contrast, the leftmost longest match of `(an)*` is the null string before `b`, which may be surprising when first encountered.

The `sub` function returns the number of substitutions made, which will be zero or one. The function `sub(`*r*`,`*s*`)` is a synonym for `sub(`*r*`,`*s*`,$0)`.

The function `gsub(`*r*`,`*s*`,`*t*`)` is similar, except that it successively replaces the leftmost longest nonoverlapping substrings matched by *r* with *s* in *t*; it returns the number of substitutions made. (The ''`g`'' is for ''global,'' meaning everywhere.) For example, the program

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

will transcribe its input, replacing all occurrences of ''USA'' by ''United States''. (In such examples, when `$0` changes, the fields and `NF` change too.) And

```
b = "banana"
gsub(/ana/, "anda", b)
```

will replace `banana` by `bandana` in `b`; matches are nonoverlapping.

In a substitution performed by either `sub(`*r*`,`*s*`,`*t*`)` or `gsub(`*r*`,`*s*`,`*t*`)`, any occurrence of the character `&` in *s* will be replaced by the substring matched by *r*. Thus

```
b = "banana"
gsub(/a/, "aba", b)
```

replaces `banana` by `babanabanaba` in `b`; so does

```
gsub(/a/, "&b&", b)
```

The special meaning of `&` in the substitution string can be turned off by preceding it with a backslash, as in `\&`.

The function `substr(`*s*`,`*p*`)` returns the suffix of *s* that begins at position *p*. If `substr(`*s*`,`*p*`,`*n*`)` is used, only the first *n* characters of the suffix are returned; if the suffix is shorter than *n*, then the entire suffix is returned. For example, we could abbreviate the country names in `countries` to their first six characters by the program

```
{ $1 = substr($1, 1, 6); print $0 }
```

to produce

```
Russia 16376 145 Europe
China 9388 1411 Asia
USA 9147 331 North America
Brazil 8358 212 South America
India 2973 1380 Asia
Mexico 1943 128 North America
Indone 1811 273 Asia
Ethiop 1100 114 Africa
Nigeri 910 206 Africa
Pakist 770 220 Asia
Japan 364 126 Asia
Bangla 130 164 Asia
```

Setting `$1` (or any other field) forces Awk to recompute `$0` and thus the fields are now separated by a space (the default value of `OFS`), no longer by a tab.

Strings are concatenated merely by writing them one after another in an expression. For example, on the `countries` file,

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

prints

```
China India Indonesia Pakistan Japan Bangladesh
```

by building `s` up a piece at a time starting with an initially empty string. To remove the extra space at the end, you could use

```
print substr(s, 1, length(s)-1)
```

instead of `print s` in the `END` action.

A.2.2 Type Conversions

Each Awk variable and field can potentially hold a string value, a numeric value, or both, at any time. This section sets out the rules for how string and numeric values are treated in assignments, comparisons, expression evaluation, input, and output.

Assignments. When a variable is set by an assignment

```
var = expr
```

its type is set to that of the expression. ("Assignment" includes the assignment operators +=, −=, 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` is set to that of `v2`.

Number or String? The value of an expression may be automatically converted from a number to a string or vice versa, depending on what operation is applied to it. In an arithmetic expression like

```
pop + $3
```

the operands `pop` and `$3` must be numeric, so their values will be forced or *coerced* to numbers if they are not already. Similarly, in the assignment expression

```
pop += $3
```

`pop` and `$3` must be numbers, so after the expression is evaluated, `pop` will be numeric and `$3` will have a numeric value, which may have been computed from its string value if it had one. In a string expression like

```
$1 $2
```

the operands `$1` and `$2` must be strings to be concatenated, so they will be coerced to strings if necessary; if they had numeric values, those will be unchanged.

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

```
$1++
```

implies that `$1` must be coerced to numeric if necessary, and

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

implies that `$1` and `$2` must be coerced to strings if necessary.

Comparisons and coercions. 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 the string values.

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 because `x` is both 0 and `""`. But if `x` is uninitialized,

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

is false because `x` is `""`, which is a string value, not numeric.

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

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

Thus, to force a string comparison between two fields, coerce one field to string:

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

To force a numeric comparison, coerce *both* fields to numeric:

```
$1 + 0 == $2 + 0
```

This works regardless of what the fields contain.

Type inference. In contexts where types cannot be reliably determined, such as

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

the type of each field is determined heuristically 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`) and `$0` for blank lines are treated this way too.

Let us examine the meaning of a comparison like

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

that involves fields. Here, the type of the comparison depends on whether the fields contain numbers or strings, and this can only be determined when the program runs; the type of the comparison may differ from input line to input line. When Awk creates a field at run time, it automatically sets its type to string; in addition, if the field contains a valid number, it also gives the field a numeric type.

For example, the comparison `$1 == $2` will be numeric and succeed if `$1` and `$2` have any of the values

```
1   1.0   +1   1e0   0.1e+1   10E-1   001
```

because all these values are different representations of the numeric value 1. However, this same expression will be a string comparison and hence fail on each of these pairs:

```
0           (null)
0.0         (null)
0           0x
1e5000      1.0e5000
```

In the first three pairs, the second field is not a number. The last pair will be compared as strings on computers where the values are too large to be represented as numbers.

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

Mentioning an array element in an expression causes it to exist, with the values 0 and `""` 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.

This property leads to an elegant program for eliminating duplicate records from an input stream:

```
!a[$0]++  # equivalently, a[$0]++ == 0
```

counts the number of times any particular line appears, but prints only the first occurrence, because that is the only time when the count for the specific array element is zero.

The test

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

determines if `arr[i]` exists without the side effect of creating it.

Number to string conversions. The print statement

```
print $1
```

prints the string value of the first field; thus, the output is identical to the input.

Nonexistent fields and fields that are explicitly null have only the string value `""`; they are not numeric, but when coerced to numbers they acquire the numeric value 0. Array subscripts are always strings; a numeric subscript is converted to its string value.

The numeric value of a string is the value of the longest numeric prefix of the string. Thus

```
BEGIN { print "1E2"+0, "12E"+0, "E12"+0, "1X2Y3"+0 }
```

yields

```
100 12 0 1
```

For printing, the string value of a number is computed by formatting the number with the output format conversion `OFMT`; its default value is `"%.6g"`. Thus

```
BEGIN { print 1E2, 12E-2, E12 "", 1.23456789 }
```

gives

```
100 0.12  1.23457
```

Look carefully at this output: there is an empty field corresponding to the third argument, `E12 ""`.

For other conversions from number to string, the string value of a number is computed by formatting the number with the conversion format conversion `CONVFMT`.

`CONVFMT` controls the conversion of numeric values to strings for concatenation, comparison, and creation of array subscripts. The default value of `CONVFMT` is also `"%.6g"`. The default values of `OFMT` and `CONVFMT` can be changed by assigning them new values. If `CONVFMT` were changed to `"%.2f"`, for example, coerced numbers would be compared with two digits after the decimal point. In both cases, integral values convert to integers regardless of `CONVFMT` and `OFMT`.

Summary of Operators. The operators that can appear in expressions are summarized in Table A-7. Expressions can be created by applying these operators to constants, variables, field names, array elements, function results, and other expressions.

The operators are listed in order of increasing precedence. Operators of higher precedence are evaluated before lower ones; this means, for example, that `*` is evaluated before `+` in an expression. All operators are left associative except the assignment operators, the conditional operator, and exponentiation, which are right associative. Left associativity means that operators of the same precedence are evaluated left to right; thus `3-2-1` is `(3-2)-1`, not `3-(2-1)`.

Since there is no explicit operator for concatenation, it is wise to parenthesize expressions involving other operators in concatenations. Consider the program

```
$1 < 0 { print "abs($1) = " -$1 }
```

The expression following `print` seems to use concatenation, but is actually a subtraction. The programs

```
$1 < 0 { print "abs($1) = " (-$1) }
```

and

```
$1 < 0 { print "abs($1) =", -$1 }
```

both do what was intended.

A.2.3 Control-Flow Statements

Awk provides braces for grouping statements, an `if-else` statement for decision-making, and `while`, `for`, and `do` statements for looping. All of these statements were adopted

TABLE A-7. EXPRESSION OPERATORS

OPERATION	OPERATORS	EXAMPLE	MEANING OF EXAMPLE
assignment	`= += -= *= /= %= ^=`	`x *= 2`	`x = x * 2`
conditional	`?:`	`x ? y : z`	if `x` is true then `y` else `z`
logical OR	`||`	`x || y`	1 if `x` or `y` is true, 0 otherwise
logical AND	`&&`	`x && y`	1 if `x` and `y` are true, 0 otherwise
array membership	`in`	`i in a`	1 if `a[i]` exists, 0 otherwise
matching	`~ !~`	`$1 ~ /x/`	1 if the first field contains an `x`, 0 otherwise
relational	`< <= == != >= >`	`x == y`	1 if `x` is equal to `y`, 0 otherwise
concatenation		`"a" "bc"`	`"abc"`; there is no explicit concatenation operator
add, subtract	`+ -`	`x + y`	sum of `x` and `y`
multiply, divide, mod	`* / %`	`x % y`	remainder of `x` divided by `y`
unary plus and minus	`+ -`	`-x`	negated value of `x`
logical NOT	`!`	`!$1`	1 if `$1` is zero or null, 0 otherwise
exponentiation	`^`	`x ^ y`	x^y
increment, decrement	`++ --`	`++x, x++`	add 1 to `x`
field	`$`	`$i+1`	value of `i`-th field, plus 1
grouping	`( )`	`$(i++)`	return `i`-th field, then increment `i`

Operators are listed in order of increasing precedence.

from C, except for the special form of `for` that iterates over arrays.

A single statement can always be replaced by a list of statements enclosed in braces. The statements in the list are separated by newlines or semicolons. Newlines may be inserted after any left brace and before any right brace.

The `if-else` statement has the form

```
if (expression)
    statement1
else
    statement2
```

The `else` *statement*$_2$ is optional. Newlines are optional after the right parenthesis, after *statement*$_1$, and after the keyword `else`. If `else` appears on the same line as *statement*$_1$, however, then a semicolon must terminate *statement*$_1$ if it is a single statement.

In an `if-else` statement, the test *expression* is evaluated first. If it is true, that is, either nonzero or nonnull, *statement*$_1$ is executed. If *expression* is false, that is, either zero or null, and `else` *statement*$_2$ is present, then *statement*$_2$ is executed.

Summary of Control-Flow Statements

`{` *statements* `}`
statement grouping
`if (`*expression*`)` *statement*
if *expression* is true, execute *statement*
`if (`*expression*`)` $statement_1$ `else` $statement_2$
if *expression* is true, execute $statement_1$ otherwise execute $statement_2$
`while (`*expression*`)` *statement*
if *expression* is true, execute *statement*, then repeat
`for (`$expression_1$`;` $expression_2$`;` $expression_3$`)` *statement*
equivalent to $expression_1$`; while (`$expression_2$`) {` *statement*`;` $expression_3$ `}`
`for (`*variable* `in` *array*`)` *statement*
execute *statement* with *variable* set to each subscript in *array* in turn, in unspecified order
`do` *statement* `while (`*expression*`)`
execute *statement*; if *expression* is true, repeat
`break`
immediately leave innermost enclosing `while`, `for`, or `do`; illegal outside of loops
`continue`
start next iteration of innermost enclosing `while`, `for`, or `do`; illegal outside of loops
`return` *expression*
return from function with value *expression* if present.
`next`
start next iteration of main input loop; illegal inside function definition
`nextfile`
start next iteration of main input loop with the next input file; illegal inside function definition
`exit`
`exit` *expression*
go immediately to the `END` action; if within the `END` action, exit program entirely. Return *expression* as program status, or zero if there is no *expression*.

To eliminate any ambiguity, each `else` is associated with the closest previous unassociated `if`. For example, in the statement

```
if (e1) if (e2) s=1; else s=2
```

the `else` is associated with the second `if`. (The semicolon after `s=1` is required, because the `else` appears on the same line.)

The `while` statement repeatedly executes a statement while a condition is true:

```
while (expression)
    statement
```

In this loop, *expression* is evaluated; if it is true, *statement* is executed and *expression* is tested again. The cycle repeats as long as *expression* is true, that is, until the *expression* becomes false. For example, this program prints all input fields, one per line:

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

The loop stops when `i` reaches `NF+1`, and that is its value after the loop exits.

The `for` statement is a more general form of `while`:

```
for (expression1; expression2; expression3)
     statement
```

The `for` statement has the same effect as

```
expression1
while (expression2) {
     statement
     expression3
}
```

so

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

does the same loop over the fields as the `while` example above. In the `for` statement, all three expressions are optional. If $expression_2$ is missing, the condition is taken to be always true, so `for(;;)` is an infinite loop.

An alternate version of the `for` statement that loops over array subscripts is described in Section A.2.5 below.

The `do` statement has the form

```
do
     statement
while (expression)
```

Newlines are optional after the keyword `do` and after *statement*. If `while` appears on the same line as *statement*, however, then *statement* must be terminated by a semicolon if it is a single statement. The `do` loop executes *statement* once, then repeats *statement* as long as *expression* is true. It differs from the `while` and `for` in a critical way: its test for completion is at the bottom instead of the top, so it always goes through the loop at least once.

There are two statements for modifying how loops cycle: `break` and `continue`. The `break` statement causes an exit from the immediately enclosing `while`, `for`, or `do`. The `continue` statement causes the next iteration to begin; it causes execution to go to the test expression in the `while` and `do`, and to $expression_3$ in the `for` statement. Both `break` and `continue` are illegal outside of loops.

The `return` statement returns from a function, optionally with a value.

The `next`, `nextfile`, and `exit` statements control the outer loop that reads the input lines in an Awk program. The `next` statement causes Awk to fetch the next input line and begin matching patterns starting from the first pattern-action statement.

The `nextfile` statement causes Awk to close the current input file and begin processing the next input file if there is one.

In an `END` action, the `exit` statement causes the program to terminate immediately. In any other action, it causes the program to behave as if the end of the input had occurred; no more input is read, and the `END` actions, if any, are executed.

If an `exit` statement includes an expression:

```
exit expr
```

it causes Awk to return the value of *expr* as its exit status unless overridden by a subsequent error or `exit`. If there is no *expr*, the exit status is zero. In some operating systems, including Unix, the exit status may be tested by the program that invoked Awk.

A.2.4 Empty Statement

A semicolon by itself denotes the empty statement. In the following program, the body of the `for` loop is an empty statement.

```
BEGIN { FS = "\t" }
      { for (i = 1; i <= NF && $i != ""; i++)
            ;
        if (i <= NF)
            print
      }
```

The program prints all lines that contain an empty field.

A.2.5 Arrays

Awk provides one-dimensional arrays for storing strings and numbers. Arrays and array elements need not be declared, nor is there any need to specify how many elements an array has or will have. Like variables, array elements spring into existence by being mentioned; at birth, they have the numeric value 0 and the string value `""`.

As a simple example, the statement

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

assigns the current input line to element `NR` of the array `x`. In fact, it is often easiest to read the entire input into an array, then process it in any convenient order. For example, this variant of the program from Section 1.7 prints its input in reverse line order:

```
    { x[NR] = $0 }
END { for (i = NR; i > 0; i--) print x[i] }
```

The first action stores each input line in the array `x`, using the line number as a subscript; the real work is done in the `END` statement.

The characteristic that sets Awk arrays apart from those in most other languages is that subscripts are strings. This gives Awk a key-value capability like Python's dictionary data structure, or hash tables in Java or JavaScript, or maps in other languages. Arrays in Awk are called *associative arrays*, a terminology that predates dictionary and hash table.

The following program accumulates the populations of `Asia` and `Africa` in the array `pop`. The `END` action prints the total populations of these two continents.

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

On `countries`, this program generates

```
Asian population 3574 million
African population 320 million
```

Note that the subscripts are the string constants `"Asia"` and `"Africa"`. If we had written `pop[Asia]` instead of `pop["Asia"]`, the expression would have used the value of the variable `Asia` as the subscript, and because that variable is uninitialized, the values would have been accumulated in `pop[""]`.

This example doesn't really need an associative array since there are only two elements, both named explicitly. Suppose instead that our task is to determine the total population for each continent. Associative arrays are ideally suited for this kind of aggregation. Any expression can be used as a subscript in an array reference, so

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

uses the string in the fourth field of the current input line to index the array `pop` and in that entry accumulates the value of the third field:

```
BEGIN { FS = "\t" }
      { pop[$4] += $3 }
END   { for (name in pop)
            print name, pop[name]
      }
```

The subscripts of the array `pop` are the continent names; the values are the accumulated populations. This code works regardless of the number of continents; the output from the `countries` file is

```
Africa 320
South America 212
North America 459
Asia 3574
Europe 145
```

The last program used the `for` statement that loops over all subscripts of an array:

`for (`*variable* `in` *array*`)`
statement

This loop executes *statement* with *variable* set in turn to each different subscript in the array. The order in which the subscripts are considered is implementation dependent. Results are unpredictable if elements are deleted or if new elements are added to the array by *statement*.

You can determine whether a particular subscript occurs in an array with the expression

subscript `in` *A*

This expression has the value 1 if *A*`[`*subscript*`]` already exists, and 0 otherwise. Thus, to test whether `Africa` is a subscript of the array `pop` you can say

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

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

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

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

The `delete` Statement. An array element may be deleted with

`delete` *array*`[`*subscript*`]`

For example, this loop removes all the elements from the array `pop`:

```
for (i in pop)
    delete pop[i]
```

The statement

`delete` *array*

deletes the entire array, and thus `delete pop` is equivalent to the loop above.

The `split` Function. The function `split(`*str*`,`*arr*`,`*fs*`)` splits the string value of *str* into fields and stores them in the array *arr*; *str* is unchanged. The number of fields produced is returned as the value of `split`. The string value of the third argument, *fs*, determines the field separator. If there is no third argument, if `--csv` is set, splitting is done as for CSV; otherwise `FS` is used. The string *fs* may be a regular expression. The rules are as for input field splitting, which is discussed in Section A.5.1. The function call

```
split("7/4/76", arr, "/")
```

splits the string `7/4/76` into three fields using `/` as the separator; it stores `7` in `arr["1"]`, `4` in `arr["2"]`, and `76` in `arr["3"]`.

If the source string is empty, the number of elements is always zero and the array is not set.

As a final special case, if the *fs* argument is the empty string `""`, the string *str* is split into individual characters, one character per array element.

Strings are versatile array subscripts, but the behavior of numeric subscripts as strings may sometimes appear counterintuitive. Since the string values of `1` and `"1"` are the same, `arr[1]` is the same as `arr["1"]`. But `"01"` is not the same string as `"1"` and the string `"10"` comes before the string `"2"`.

Multidimensional Arrays. Awk does not support multidimensional arrays directly but it provides a simulation using one-dimensional arrays. If you write multidimensional subscripts like `[i,j]` or `[s,p,q,r]`, Awk concatenates the components of the subscripts (with a separator between them) to synthesize a single subscript out of the multiple subscripts that you write. For example,

```
for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
        arr[i,j] = 0
```

creates an array of 100 elements whose subscripts appear to have the form `1,1`, `1,2`, and so

on. Internally, however, these subscripts are stored as strings of the form `1 SUBSEP 1`, `1 SUBSEP 2`, and so on. The built-in variable `SUBSEP` contains the value of the subscript-component separator; its default value is not a comma but the ASCII file separator character `"\034"` or `"\x1C"`, a value that is unlikely to appear in normal text.

The test for array membership with multidimensional subscripts uses a parenthesized list of subscripts, such as

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

To loop over such an array, however, you would write

```
for (k in arr) ...
```

and use `split(k,x,SUBSEP)` if access to the individual subscript components is needed.

Array elements cannot themselves be arrays.

A.3 User-Defined Functions

In addition to built-in functions, an Awk program can contain user-defined functions. Such a function is defined by a statement of the form

```
function name(parameter-list) {
    statements
}
```

A function definition can occur anywhere a pattern-action statement can. Thus, the general form of an Awk program is a sequence of pattern-action statements and function definitions separated by newlines or semicolons.

In a function definition, newlines are optional after the left brace and before the right brace of the function body. The parameter list is a sequence of zero or more variable names separated by commas; within the body of the function these variables refer to the arguments with which the function was called.

The body of a function definition may contain a `return` statement that returns control and perhaps a value to the caller:

```
return expression
```

The *expression* is optional, and so is the `return` statement itself, but the return value is `""` and 0 if no *expression* is provided. If the last statement executed is not a `return` (“falling off the end of the function”) the return value is also `""` and 0.

For example, this function computes the maximum of its arguments:

```
function max(m, n) {
    return m > n ? m : n
}
```

The variables `m` and `n` are local to the function `max`; they are unrelated to any other variables of the same names elsewhere in the program.

A user-defined function can be used in any expression in any pattern-action statement or the body of any function definition. Each use is a *call* of the function.

For example, the `max` function might be called like this:

```
{ print max($1, max($2, $3)) }  # print maximum of $1, $2, $3

function max(m, n) {
    return m > n ? m : n
}
```

There cannot be any spaces between the function name and the left parenthesis of the argument list when the function is called.

If a user-defined function is called in the body of its own definition, that function is said to be *recursive*.

When a function is called with an argument like `$1`, which is just an ordinary variable, the function is given a copy of the value of the variable, so the function manipulates the copy, not the variable itself. This means that the function cannot affect the value of the variable outside the function. (The jargon is that such variables, called "scalars," are passed "by value.") Arrays are not copied, however, so it is possible for the function to alter array elements or create new ones. (This is called passing "by reference.") The name of a function may not be used as a parameter, global array, or scalar.

To repeat, within a function definition, the parameters are local variables — they last only as long as the function is executing, and they are unrelated to variables of the same name elsewhere in the program. But *all other variables are global*; if a variable is not named in the parameter list, it is visible and accessible throughout the program.

This means that the only way to provide local variables for the private use of a function is to include them at the end of the parameter list in the function definition. Any variable in the parameter list for which no actual parameter is supplied in a call is a local variable, with null initial value. This is not a good design but it at least provides the necessary facility. We insert several spaces between the arguments and the local variables so they can be more easily distinguished. Omitting a local variable from this list is a common source of bugs.

A.4 Output

The `print` and `printf` statements generate output. The `print` statement is used for simple output; `printf` is used when careful formatting is required. Output from `print` and `printf` can be directed into files and pipes as well as to the terminal. These statements can be used in any mixture; the output comes out in the order in which it is generated.

A.4.1 The `print` Statement

The `print` statement has two equivalent forms:

```
print expr1, expr2, ... , exprn
print(expr1, expr2, ... , exprn)
```

Both forms print 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

Summary of Output Statements

`print`
print `$0` on standard output
`print` *expression*`,` *expression*`, ...`
print *expression*s, separated by `OFS`, terminated by `ORS`
`print` *expression*`,` *expression*`, ...` `>` *filename*
print on file *filename* instead of standard output
`print` *expression*`,` *expression*`, ...` `>>` *filename*
append to file *filename* instead of overwriting previous contents
`print` *expression*`,` *expression*`, ...` `|` *command*
print to standard input of *command*
`printf(`*format*`,` *expression*`,` *expression*`, ...)`
`printf(`*format*`,` *expression*`,` *expression*`, ...)` `>` *filename*
`printf(`*format*`,` *expression*`,` *expression*`, ...)` `>>` *filename*
`printf(`*format*`,` *expression*`,` *expression*`, ...)` `|` *command*
`printf` statements are like `print` but the first argument specifies the output format
`close(`*filename*`)`, `close(`*command*`)`
break connection between `print` and *filename* or *command*
`fflush(`*filename*`)`, `fflush(`*command*`)`
force out any buffered output of *filename* or *command*

If an expression in the argument list of a `print` or `printf` statement contains a relational operator, either the expression or the argument list must be enclosed in parentheses. Pipes may not be available on non-Unix systems.

```
print $0
```

To print a blank line, that is, a line with only a newline, use

```
print ""
```

The second form of the `print` statement encloses the argument list in parentheses, as in

```
print($1 ":", $2)
```

Both forms of the `print` statement generate the same output but, as we will see, parentheses may be necessary for arguments containing relational operators.

A.4.2 Output Separators

The output field separator and output record separator are stored in the built-in variables `OFS` and `ORS`. Initially, `OFS` is set to a single space 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 line with a colon between the fields and two newlines after the second field:

```
BEGIN   { OFS = ":"; ORS = "\n\n" }
        { print $1, $2 }
```

By contrast,

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

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

A.4.3 The `printf` Statement

The `printf` statement generates formatted output. It is similar to that in C, though width qualifiers like `h` and `l` have no effect.

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

The *format* argument is always required; it is an expression whose string value contains both literal text to be printed and specifications of how the expressions in the argument list are to be formatted, as in Table A-8. Each specification begins with `%`, ends with a character that determines the conversion, and may include modifiers:

`-`	left-justify expression in its field
`+`	always print a sign
`0`	pad with zeros instead of spaces
width	pad result to this width as needed; leading `0` pads with zeros
`.`*prec*	maximum string width, or digits to right of decimal point

If a `*` appears in a specification, it is replaced by the numeric value of the next argument, so widths and precisions can be provided dynamically.

TABLE A-8. PRINTF FORMAT-CONTROL CHARACTERS

CHARACTER	PRINT EXPRESSION AS
`c`	single UTF-8 character (code point)
`d` or `i`	decimal integer
`e` or `E`	`[-]d.dddddde[+-]dd` or `[-]d.ddddddE[+-]dd`
`f`	`[-]ddd.dddddd`
`g` or `G`	`e` or `f` conversion, whichever is shorter, with nonsignificant zeros suppressed
`o`	unsigned octal number
`u`	unsigned integer
`s`	string
`x` or `X`	unsigned hexadecimal number
`%`	print a `%`; no argument is consumed

Table A-9 contains some examples of specifications, data, and the corresponding output. Output produced by `printf` does not contain any newlines unless you put them in explicitly.

A.4.4 Output into Files

The redirection operators > and >> are used to put output into files instead of the standard output. The following program will put the first and third fields of all input lines into two files: `bigpop` if the third field is greater than 1000, and `smallpop` otherwise:

```
$3 > 1000   { print $1, $3 >"bigpop" }
$3 <= 1000  { print $1, $3 >"smallpop" }
```

TABLE A-9. EXAMPLES OF PRINTF SPECIFICATIONS

fmt	$1	printf(fmt, $1)
%c	97	a
%d	97.5	97
%5d	97.5	97
%e	97.5	9.750000e+01
%f	97.5	97.500000
%7.2f	97.5	97.50
%g	97.5	97.5
%.6g	97.5	97.5
%o	97	141
%06o	97	000141
%x	97	61
|%s|	January	|January|
|%10s|	January	| January|
|%-10s|	January	|January |
|%.3s|	January	|Jan|
|%10.3s|	January	| Jan|
|%-10.3s|	January	|Jan |
%%	January	%

Notice that the filenames have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. Filenames can be variables or expressions as well:

```
{ print($1, $3) > ($3 > 1000 ? "bigpop" : "smallpop") }
```

does the same job, and the program

```
{ print > $1 }
```

puts each input line into a file named by its first field.

In `print` and `printf` statements, if an expression in the argument list contains a relational operator, then either that expression or the argument list needs to be parenthesized. This rule eliminates any potential ambiguity arising from the redirection operator >. In

```
{ print $1, $2 > $3 }
```

> is the redirection operator, and hence not part of the second expression, so the values of the first two fields are written to the file named in the third field. If you want the second expression to include the > operator, use parentheses:

```
{ print $1, ($2 > $3) }
```

It is also important to note that a redirection operator opens a file only once; each successive `print` or `printf` statement adds more data to the open file. When the redirection operator > is used, the file is initially cleared before any output is written to it. If >> is used instead of >, the file is not cleared when opened; output is appended after the original contents of the file.

There are three special filenames for pre-defined input and output streams: `"/dev/stdin"`, `"/dev/stdout"`, and `"/dev/stderr"` represent the standard input, standard output, and standard error streams of the program. The name `"-"` may also be used for the standard input.

A.4.5 Output into Pipes

It is also possible to direct output into a pipe instead of a file on systems that support pipes. The statement

```
print | command
```

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

Suppose we want to create a list of continent-population pairs, sorted in reverse numeric order by population. The 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. The `END` action prints each continent name and its population, and pipes this output into a suitable `sort` command.

```
# print continents and populations, sorted by population

BEGIN { FS = "\t" }
      { pop[$4] += $3 }
END   { for (c in pop)
          printf("%15s\t%6d\n", c, pop[c]) | "sort -t'\t' -k2 -rn"
      }
```

This yields

```
           Asia          3574
  North America           459
         Africa           320
  South America           212
         Europe           145
```

Another use for a pipe is writing onto the standard error file on Unix systems; output written there appears on the user's terminal instead of the standard output. There are several older idioms for writing on the standard error:

```
print message | "cat 1>&2"          # redirect cat output to stderr
system("echo '" message "' 1>&2")   # redirect echo output to stderr
print message > "/dev/tty"          # write directly on terminal
```

but the easiest idiom with newer versions of Awk is instead to write to `/dev/stderr`.

Although most of our examples show literal strings enclosed in quotes, command lines and filenames can be specified by any expression. In `print` statements involving redirection of output, the files or pipes are identified by their names; that is, the pipe in the program above is literally named

```
sort -t'\t' -k2 -rn
```

Normally, a file or pipe is created and opened only once during the run of a program. If the file or pipe is explicitly closed and then reused, it will be reopened.

A.4.6 Closing Files and Pipes

The statement `close(`*expr*`)` closes a file or pipe denoted by *expr*; the string value of *expr* must be exactly the same as the string used to create the file or pipe in the first place. Thus

```
close("sort -t'\t' -k2 -rn")
```

closes the sort pipe opened above.

`close` is necessary if you intend to write a file, then read it later in the same program. There are also system-defined limits on the number of files and pipes that can be open at the same time.

`close` is a function; it returns the value returned by the underlying `fclose` function or exit status for a pipeline.

The `fflush` function forces out any output that has been collected for a file or pipe; `fflush()` or `fflush("")` flush all output files and pipes.

A.5 Input

There are several ways of providing input to an Awk program. It's obviously possible to just type input at the keyboard, but the most common arrangement is to put input data in a file, say `data`, and then type

```
awk 'program' data
```

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 `grep` selects input lines containing a specified regular expression, and is part of many Unix programmers' muscle memory. They might instinctively type

```
grep Asia countries | awk 'program'
```

to use `grep` to find the lines containing `Asia` and pass them on to Awk for subsequent processing.

Use `"-"` or `/dev/stdin` on the command line to read the standard input in the middle of a list of files.

Note that literal escaped characters like `\n` or `\007` are not interpreted nor are they in any way special when they appear in an input stream; they are just literal byte sequences. The only interpretation on input is that apparently-numeric values like scientific notation and explicitly signed instances of `nan` and `inf` will be stored with a numeric value as well as a string value.

A.5.1 Input Separators

The default value of the built-in variable `FS` is `" "`, that is, a single space. When `FS` has this specific value, input fields are separated by spaces and/or tabs, and leading spaces and tabs are discarded, so each of the following lines has the same first field:

```
field1
  field1
    field1  field2
```

When `FS` has any other value, leading spaces and tabs are *not* discarded.

The field separator can be changed by assigning a string to the built-in variable `FS`. If the string is longer than one character, it is taken to be a regular expression. The leftmost longest nonnull and nonoverlapping substrings matched by that regular expression become the field separators in the current input line. For example,

```
BEGIN { FS = "[ \t]+" }
```

makes every string consisting of spaces and tabs into a field separator.

When `FS` is set to a single character other than space, that literal character becomes the field separator. This convention makes it easy to use regular expression metacharacters as field separators:

```
FS = "|"
```

makes `|` a field separator. But note that something indirect like

```
FS = "[ ]"
```

is required to set the field separator to a single space.

`FS` can also be set on the command line with the `-F` argument. The command line

```
awk -F'[ \t]+' 'program'
```

sets the field separator to the same strings as the `BEGIN` action shown above.

Finally, if the `--csv` argument is used, fields are treated as comma-separated values, and the value of `FS` is irrelevant.

A.5.2 CSV Input

Comma-separated values, or CSV, is a widely used format for spreadsheet data. As we said earlier, CSV is not rigorously defined, but generally any field that contains a comma or a double quote (`"`) must be surrounded by double quotes. Any field may be surrounded by quotes, whether it contains commas and quotes or not. An empty field is just `""`, and a quote within a field is represented by a doubled quote, as in `""","""`, which represents `","`.

Input records are terminated by an unquoted newline, perhaps preceded by a carriage return (`\r`) for files that originated on Windows. Input fields in CSV files may contain embedded newline characters. A quoted `\r\n` is converted to `\n`. A quoted `\r` or `\n` is left alone.

A.5.3 Multiline Records

By default, records are separated by newlines, so the terms "line" and "record" are normally synonymous. The default record separator can be changed, however, by assigning a new value to the built-in record-separator variable `RS`.

If `RS` is set to the null string, as in

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

then records are separated by one or more blank lines and each record can therefore occupy several lines. Setting `RS` back to newline with the assignment `RS = "\n"` restores the default behavior. With multiline records, no matter what value `FS` has, newline is always one of the field separators. Input fields may not contain newlines unless the `--csv` option has been used.

A common way to process multiline records is to use

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

to set the record separator to one or more blank lines and the field separator to a newline alone; each line is thus a separate field. Section 4.4 contains more discussion of how to handle multiline records.

The RS variable can be a regular expression, so it's possible to separate records by text strings more complicated than just a single character. For example, in a well-structured HTML document, individual paragraphs might be separated by <p> tags. By setting RS to <[Pp]>, an input file could be split into records that were each one HTML paragraph.

A.5.4 The `getline` Function

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

The expression getline x reads the next record into the variable x and increments NR and FNR. No splitting is done; NF is not set.

The expression

```
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 expression

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

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

If the filename is "−", the standard input is read; the filename "/dev/stdin" is equivalent.

Table A-10 summarizes the forms of the getline function. The value of each expression is the value returned by getline.

TABLE A-10. GETLINE FUNCTION

EXPRESSION	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*

As an example, this program copies its input to its output, except that each line like

```
#include "filename"
```

is replaced by the contents of the file `filename`.

```
# include - replace #include "f" by contents of file f

/^#include/ {
    gsub(/"/, "", $2)
    while (getline x <$2 > 0)
        print x
    close(x)
    next
}
{ print }
```

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

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

executes the Unix program `who` (once only) and pipes its output into `getline`. The output of `who` is a list of the users logged in. Each iteration of the `while` loop reads one more line from this list and increments the variable `n`, so after the `while` loop terminates, `n` contains a count of the number of users. Similarly, the expression

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

pipes the output of the `date` command into the variable `d`, thus setting `d` to the current date. Again, input pipes may not be available on non-Unix systems.

In all cases involving `getline`, you should be aware of the possibility of an error return if the file can't be accessed. Although it's appealing to write

```
while (getline <"file") ...          # Dangerous
```

that's an infinite loop if `file` doesn't exist, because with a nonexistent file `getline` returns −1, a nonzero value that is interpreted as true. The preferred way is

```
while (getline <"file" > 0) ...      # Safe
```

Here the loop will be executed only when `getline` returns 1, which it does for each input line it reads.

A.5.5 Command-Line Arguments and Variable Assignments

As we have seen, an Awk command line can have several forms:

```
awk 'program' f1 f2 ...
awk -f progfile f1 f2 ...
awk -Fsep 'program' f1 f2 ...
awk -Fsep -f progfile f1 f2 ...
awk --csv f1 f2 ...
awk -v var=value f1 f2 ...
awk --version
```

In these command lines, `f1`, `f2`, etc., are command-line arguments that normally represent filenames; the name `"-"` may be used for the standard input. The argument `--csv` enables CSV input processing.

The special argument `--` can be used to end the list of options.

If a filename has the form *var=value*, however, it is treated as an assignment of *value* to the Awk variable *var*, performed when that argument would otherwise be accessed as a file. This type of assignment allows variables to be changed before and after a file is read.

The command-line arguments are available to the Awk program in a built-in array called `ARGV`. The value of the built-in variable `ARGC` is one more than the number of arguments. With the command line

```
awk -f progfile a v=1 b
```

`ARGC` has the value 4, `ARGV[0]` contains `awk`, `ARGV[1]` contains `a`, `ARGV[2]` contains `v=1`, and `ARGV[3]` contains `b`. `ARGC` is one more than the number of arguments because `awk`, the name of the command, is counted as argument zero, as it is in C programs. If the Awk program appears on the command line, however, the program is not treated as an argument, nor is `-f` *filename* or any `-F` option. For example, with the command line

```
awk -F'\t' '$3 > 100' countries
```

`ARGC` is 2 , `ARGV[0]` is `awk` and `ARGV[1]` is `countries`.

The following program echoes its command-line arguments (and a spurious space):

```
# echo - print command-line arguments

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

Notice that everything happens in the `BEGIN` action: because there are no other pattern-action statements, the arguments are never treated as filenames, and no input is read.

Another program using command-line arguments is `seq`, which generates sequences of integers:

```
# seq - print sequences of integers
#   input:  arguments q, p q, or p q r;  q >= p; r > 0
#   output: integers 1 to q, p to q, or p to q in steps of r

BEGIN {
    if (ARGC == 2)
        for (i = 1; i <= ARGV[1]; i++)
            print i
    else if (ARGC == 3)
        for (i = ARGV[1]; i <= ARGV[2]; i++)
            print i
    else if (ARGC == 4)
        for (i = ARGV[1]; i <= ARGV[2]; i += ARGV[3])
            print i
}
```

The commands

```
awk -f seq 10
awk -f seq 1 10
awk -f seq 1 10 1
```

all generate the integers one through ten.

The arguments in ARGV may be modified or added to, and ARGC may be altered. As each input file ends, Awk treats the next nonnull element of ARGV (up through the current value of ARGC−1) as the name of the next input file. Thus setting an element of ARGV to null means that it will not be treated as an input file.

Increasing ARGC and adding elements to ARGV cause more filenames to be processed.

A.6 Interaction with Other Programs

This section describes some of the ways in which Awk programs can cooperate with other commands. The discussion applies primarily to the Unix operating system; the examples here may fail or work differently on non-Unix systems.

A.6.1 The `system` Function

The built-in function `system(`*expression*`)` executes the command given by the string value of *expression*. The value returned by `system` is the status returned by the command that was executed, as with `close`.

For example, we can build another version of the file-inclusion program of Section A.5.4 like this:

```
$1 == "#include" {
   gsub(/"/, "", $2)
   system("cat " $2)
   next
}

{ print }
```

If the first field is `#include`, quotes are removed, and the Unix command `cat` is called to print the file named in the second field. Other lines are just copied.

A.6.2 Making a Shell Command from an Awk Program

In all of the examples so far, the Awk program was in a file and fetched with the `-f` flag, or it appeared on the command line enclosed in single quotes, like this:

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

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

Both methods of invoking the Awk program require some typing. To reduce the number of keystrokes, we might want to put both the command and the program into an executable file, and invoke the command by typing just the name of the file.

Suppose we want to create a command `field1` that will print the first field of each line of input. This is easy: we put

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

into the file `field1`, and make the file executable by typing the Unix command

```
$ chmod +x field1
```

We can now print the first field of each line of a set of files by typing

```
field1 filenames ...
```

Now, consider writing a more general command `field` that will print an arbitrary combination of fields from each line of its input; in other words, the command

```
field n1 n2 ... file1 file2 ...
```

will print the specified fields in the specified order. How do we get the value of each n_i into the Awk program each time it is run and how do we distinguish the n_i's from the filename arguments?

There are several ways to do this if one is adept in shell programming. The simplest way that uses only Awk, however, is to scan through the built-in array `ARGV` to process the n_i's, resetting each such argument to the null string so that it is not treated as a filename.

```
# field - print named fields of each input line
#   usage:  field n n n ... file file file ...

awk '
BEGIN {
    for (i = 1; ARGV[i] ~ /^[0-9]+$/; i++) { # collect numbers
        fld[++nf] = ARGV[i]
        ARGV[i] = ""
    }
    if (i >= ARGC)    # no file names so force stdin
        ARGV[ARGC++] = "-"
}

{   for (i = 1; i <= nf; i++)
        printf("%s%s", $fld[i], i < nf ? " " : "\n")
}
' $*
```

This version can deal with either standard input or a list of filename arguments, and with any number of fields in any order.

A.7 Summary

As we said earlier, this is a long manual, packed with details, and you are dedicated indeed if you have read every word to get here. You will find that it pays to go back and reread sections from time to time, either to see precisely how something works, or because one of the examples suggests a construction that you might not have tried before.

Awk, like any language, is best learned by experience and practice, so we encourage you to write your own programs. They don't have to be big or complicated — you can usually learn how some feature works or test some crucial point with only a couple of lines of code, and you can just type in data to see how the program behaves.