An Awk Primer

Contents

Articles

References

Article Licenses

A Guided Tour of Awk

Awk Overview

The Awk text-processing language is useful for such tasks as:

- Tallying information from text files and creating reports from the results.
- Adding additional functions to text editors like "vi".
- Translating files from one format to another.
- Creating small databases.
- Performing mathematical operations on files of numeric data.

Awk has two faces: it is a utility for performing simple text-processing tasks, and it is a programming language for performing complex text-processing tasks.

The two faces are really the same, however. Awk uses the same mechanisms for handling any text-processing task, but these mechanisms are flexible enough to allow useful Awk programs to be entered on the command line, or to implement complicated programs containing dozens of lines of Awk statements.

Awk statements comprise a programming language. In fact, Awk is useful for simple, quick-and-dirty computational programming. Anybody who can write a BASIC program can use Awk, although Awk's syntax is different from that of BASIC. Anybody who can write a C program can use Awk with little difficulty, and those who would like to learn C may find Awk a useful stepping stone, with the caution that Awk and C have significant differences beyond their many similarities.

There are, however, things that Awk is not. It is not really well suited for extremely large, complicated tasks. It is also an "interpreted" language -- that is, an Awk program cannot run on its own, it must be executed by the Awk utility itself. That means that it is relatively slow, though it is efficient as interpretive languages go, and that the program can only be used on systems that have Awk. There are translators available that can convert Awk programs into C code for compilation as stand-alone programs, but such translators have to be purchased separately.

One last item before proceeding: What does the name "Awk" mean? Awk actually stands for the names of its authors: "Aho, Weinberger, & Kernighan". Kernighan later noted: "Naming a language after its authors ... shows a certain poverty of imagination." The name is reminiscent of that of an oceanic bird known as an "auk", and so the picture of an auk often shows up on the cover of books on Awk.

Awk Command-Line Examples

Introduction

It is easy to use Awk from the command line to perform simple operations on text files. Suppose I have a file named "coins.txt" that describes a coin collection. Each line in the file contains the following information:

- metal
- weight in ounces
- date minted
- country of origin
- description

The file has the following contents:

```
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
silver  10    1981  USA                ingot
gold     1    1984  Switzerland        ingot
gold     1    1979  RSA                Krugerrand
gold     0.5  1981  RSA                Krugerrand
gold     0.1  1986  PRC                Panda
silver   1    1986  USA                Liberty dollar
gold     0.25 1986  USA                Liberty 5-dollar piece
silver   0.5  1986  USA                Liberty 50-cent piece
silver   1    1987  USA                Constitution dollar
gold     0.25 1987  USA                Constitution 5-dollar piece
gold     1    1988  Canada             Maple Leaf
```

I could then invoke Awk to list all the gold pieces as follows:

```
awk '/gold/' coins.txt
```

This tells Awk to search through the file for lines of text that contain the string "gold", and print them out. The result is:

```
gold     1    1986  USA                American Eagle
gold     1    1908  Austria-Hungary    Franz Josef 100 Korona
gold     1    1984  Switzerland        ingot
gold     1    1979  RSA                Krugerrand
gold     0.5  1981  RSA                Krugerrand
gold     0.1  1986  PRC                Panda
gold     0.25 1986  USA                Liberty 5-dollar piece
gold     0.25 1987  USA                Constitution 5-dollar piece
gold     1    1988  Canada             Maple Leaf
```

Printing the Descriptions

This is all very nice, a critic might say, but any "grep" or "find" utility can do the same thing. True, but Awk is capable of doing much more. For example, suppose I only want to print the description field, and leave all the other text out. I could then change my invocation of Awk to:

```
awk '/gold/ {print $5,$6,$7,$8}' coins.txt
```

This yields:

```
American Eagle
Franz Josef 100 Korona
ingot
Krugerrand
Krugerrand
Panda
Liberty 5-dollar piece
Constitution 5-dollar piece
Maple Leaf
```

Simplest Awk Program

This example demonstrates the simplest general form of an Awk program:

```
awk search pattern { program actions }
```

Awk searches through the input file line-by-line, looking for the search pattern. For each of these lines found, Awk then performs the specified actions. In this example, the action is specified as:

```
{print $5,$6,$7,$8}
```

The purpose of the `print` statement is obvious. The `$5`, `$6`, `$7`, and `$8` are **fields**, or "field variables", which store the words in each line of text by their numeric sequence. `$1`, for example, stores the first word in the line, `$2` has the second, and so on. By default a "word", or **record**, is defined as any string of printing characters separated by spaces.

Based on the structure of "coins.txt" (see above), the field variables are matched to each line of text in the file as follows:

```
metal:        $1
weight:       $2
date:         $3
country:      $4
description:  $5 through $8
```

The program action in this example prints the fields that contain the description. The description field in the file may actually include from one to four fields, but that's not a problem, since "print" simply ignores any undefined fields. The astute reader will notice that the "coins.txt" file is neatly organized so that the only piece of information that contains multiple fields is at the end of the line. This limit can be overcome by changing the field separator, explained later.

Awk's default program action is to print the entire line, which is what "print" does when invoked without parameters. This means that these three examples are the same:

```
awk '/gold/'
awk '/gold/ {print}'
```

```
awk '/gold/ {print $0}'
```

Note that Awk recognizes the field variable `$0` as representing the entire line. This is redundant, but it does have the virtue of making the action more obvious.

Conditionals

Now suppose I want to list all the coins that were minted before 1980. I invoke Awk as follows:

```
awk '{if ($3 < 1980) print $3, "    ",$5,$6,$7,$8}' coins.txt
```

This yields:

```
1908    Franz Josef 100 Korona
1979    Krugerrand
```

This new example adds a few new concepts:

Printing Lines

- If no search pattern is specified, Awk will match *all* lines in the input file, and perform the actions on each one.
- The `print` statement can display custom text (in this case, four spaces) simply by enclosing the text in quotes and adding it to the *parameter list*.
- An `if` statement is used to check for a certain condition, and the `print` statement is executed only if that condition is true.

There's a subtle issue involved here, however. In most computer languages, strings are strings, and numbers are numbers. There are operations that are unique to each, and one must be specifically converted to the other with conversion functions. You don't concatenate numbers, and you don't perform arithmetic operations on strings.

Awk, on the other hand, makes no strong distinction between strings and numbers. In computer-science terms, it isn't a "strongly-typed" language. All data in Awk are regarded as strings, but if that string also happens to represent a number, numeric operations can be performed on it. So we can perform an *arithmetic* comparison on the date field.

`BEGIN` and `END`

The next example prints out how many coins are in the collection:

```
awk 'END {print NR,"coins"}' coins.txt
```

This yields:

```
13 coins
```

The first new item in this example is the `END` statement. To explain this, I have to extend the general form of an Awk program.

Awk Programs

Every Awk program follows this format (each part being optional):

```
awk 'BEGIN { initializations } search pattern 1 { program actions } search pattern 2 { program actions }  ...  END { final actions }' input file
```

The `BEGIN` clause performs any initializations required before Awk starts scanning the input file. The subsequent body of the Awk program consists of a series of search patterns, each with its own program action. Awk scans each line of the input file for each search pattern, and performs the appropriate actions for each string found. Once the file has been scanned, an `END` clause can be used to perform any final actions required.

So, this example doesn't perform any processing on the input lines themselves. All it does is scan through the file and perform a final action: print the number of lines in the file, which is given by the `NR` variable. `NR` stands for "number of records". `NR` is one of Awk's "pre-defined" variables. There are others, for example the variable `NF` gives the number of fields in a line, but a detailed explanation will have to wait for later.

Counting Money

Suppose the current price of gold is $425 per ounce, and I want to figure out the approximate total value of the gold pieces in the coin collection. I invoke Awk as follows:

```
awk '/gold/ {ounces += $2} END {print "value = $" 425*ounces}' coins.txt
```

This yields:

```
value = $2592.5
```

In this example, `ounces` is a variable I defined myself, or a "user-defined" variable. Almost any string of characters can be used as a variable name in Awk, as long as the name doesn't conflict with some string that has a specific meaning to Awk, such as `print` or `NR` or `END`. There is no need to declare the variable, or to initialize it. A variable handled as a string value is initialized to the "null string", meaning that if you try to print it, nothing will be there. A variable handled as a numeric value will be initialized to zero.

So the program action:

```
{ounces += $2}
```

sums the weight of the piece on each matched line into the variable `ounces`. Those who program in C should be familiar with the `+=` operator. Those who don't can be assured that this is just a shorthand way of saying:

```
{ounces = ounces + $2}
```

The final action is to compute and print the value of the gold:

```
END {print "value = $" 425*ounces}
```

The only thing here of interest is that the two print parameters—the literal `value = $` and the expression `425*ounces`—are separated by a space, not a comma. This concatenates the two parameters together on output, without any intervening spaces.

Practice

If you do not give Awk an input file, it will allow you to type input directly to your program. Pressing CTRL-D will quit.

1. Try modifying one of the above programs to calculate and display the total amount (in ounces) of gold and silver, separately but with one program. You will have to use two pairs of pattern/action.
2. Write an Awk program that finds the average weight of all coins minted in the USA.
3. Write an Awk program that reprints its input with line numbers before each line of text.

In the next chapter, we learn how to write Awk programs that are longer than one line.

Awk Program Example

A Large Program

All this is fun, but each of these examples only seems to nibble away at "coins.txt". Why not have Awk figure out *everything* interesting at one time?

The immediate objection to this idea is that it would be impractical to enter a lot of Awk statements on the command line, but that's easy to fix. The commands can be written into a file, and then Awk can execute the commands from that file.

```
awk -f awk program file name
```

Given the ability to write an Awk program in this way, then what should a "master" "coins.txt" analysis program do? Here's one possible output:

```
Summary Data for Coin Collection:

    Gold pieces:                  nn
    Weight of gold pieces:        nn.nn
    Value of gold pieces:       nnnn.nn

    Silver pieces:                nn
    Weight of silver pieces:      nn.nn
    Value of silver pieces:     nnnn.nn

    Total number of pieces:       nn
    Value of collection:        nnnn.nn
```

The "Master" Program

The following Awk program generates this information:

```
# This is an awk program that summarizes a coin collection.
/gold/    { num_gold++; wt_gold += $2 }                          # Get weight of gold.
/silver/  { num_silver++; wt_silver += $2 }                      # Get weight of silver.
END {
    val_gold = 485 * wt_gold;                                    # Compute value of gold.
    val_silver = 16 * wt_silver;                                 # Compute value of silver.
    total = val_gold + val_silver;

    print "Summary data for coin collection:";
    printf("\n");                                                # Skips to the next line.
    printf("    Gold pieces:\t\t%4i\n", num_gold);
    printf("    Weight of gold pieces:\t%7.2f\n", wt_gold);
    printf("    Value of gold pieces:\t%7.2f\n", val_gold);
    printf("\n");
    printf("    Silver pieces:\t\t%4i\n", num_silver);
    printf("    Weight of silver pieces:\t%7.2f\n", wt_silver);
    printf("    Value of silver pieces:\t%7.2f\n", val_silver);
    printf("\n");
```

```
    printf("    Total number of pieces:\t%4i\n", NR);
    printf("    Value of collection:\t%7.2f\n", total);
}
```

This program has a few interesting features:

- Comments can be inserted in the program by preceding them with a `#`. Awk ignores everything after `#`.
- Note the statements `num_gold++` and `num_silver++`. C programmers should understand the `++` operator; those who are not can be assured that it simply *increments* the specified variable by one. There is also a `--` that *decrements* the variable.
- Multiple statements can be written on the same line by separating them with a semicolon (`;`). Semicolons are optional if there is only one statement on the line.
- Note the use of the `printf` statement, which offers more flexible printing capabilities than the `print` statement.

The `printf` Statement

`printf` has the general syntax:

```
printf("<format_code>", <parameters>)
```

Special Characters:

- `\n` New line
- `\t` Tab (aligned spacing)

Format Codes:

- `%i` or `%d` Integer
- `%f` Floating-point (decimal) number
- `%s` String

The above description of `printf` is oversimplified. There are many more codes and options, which will be discussed later.

There is one format code for each of the parameters in the list. Each format code determines how its corresponding parameter will be printed. For example, the format code `%2d` tells Awk to print a two-digit integer number, and the format code `%7.2f` tells Awk to print a seven-digit floating-point number, including two digits to the right of the decimal point.

Note also that, in this example, each string printed by `printf` ends with a `\n`, which is a code for a *newline* (ASCII line-feed code). Unlike the `print` statement, which automatically advances the output to the next line when it prints a line, `printf` does not automatically advance the output, and by default the next output statement will append its output to the same line. A newline forces the output to skip to the next line.

The tabs created by `\t` align the output to the nearest *tab stop*, usually 8 spaces apart. Tabs are useful for creating tables and neatly aligned outputs.

Running the Program

I stored this program in a file named "summary.awk", and invoked it as follows:

```
awk -f summary.awk coins.txt
```

The output was:

```
Summary data for coin collection:

    Gold pieces:               9
    Weight of gold pieces:         6.10
    Value of gold pieces:      2958.50

    Silver pieces:               4
    Weight of silver pieces:        12.50
    Value of silver pieces:       200.00

    Total number of pieces:        13
    Value of collection:      3158.50
```

Practice

So far you have enough information to make good use of Awk. The next chapter provides a much more complete description of the language.

1. Modify the above program to tally and display the countries from "coins.txt".
2. Write a program that counts and displays the number of blank and non-blank lines (use `NF`).
3. Modify the program from #2 to count the average number of words per line.

Awk Syntax

Awk Invocation and Operation

Awk is invoked as follows:

```
awk -Fch  -f program-file  variables  input-files
```

Each parameter given to Awk is optional.

Field Separator

The first option given to Awk, `-F`, lets you change the *field separator*. Normally, each "word" or **field** in our data files is separated by a space. That can be changed to any one character. Many files are *tab-delimited*, so each data field (such as metal, weight, country, description, etc.) is separated by a tab. Using tabs allows spaces to be included in a field. Other common field separators are colons or semicolons.

For example, the file "/etc/passwd" (on Unix or Linux systems) contains a list of all users, along with some data. The each field is separated with a colon. This example program prints each user's name and ID number:

```
awk -F:  '{ print $1, $3 }' /etc/passwd
```

Notice the colon used as a field separator. This program would not work without it.

Program File

An Awk program has the general form:

```
BEGIN { initializations }  search pattern 1 { program actions } search pattern 2 { program actions } ... END { final actions }
```

Again, each part is optional if it is not needed.

If you type the Awk program into a separate file, use the `-f` option to tell Awk the location and name of that file. For larger, more complex programs, you will definitely want to use a program file. This allows you to put each statement on a separate line, making liberal use of spacing and indentation to improve readability. For short, simple programs, you can type the program directly on the command line.

If the Awk program is written on the command line, it should be enclosed in single quotes instead of double quotes to prevent the shell from interpreting characters within the program as special shell characters. Please remember that the COMMAND.COM shell (for Windows and DOS) does not allow use of single quotes in this way. Naturally, if such interpretation is desired, double quotes can be used. Those special shell characters in the Awk program that the shell should *not* interpret should be preceded with a backslash.

Variables

It is also possible to initialize Awk variables on the command line. This is obviously only useful if the Awk program is stored in a file, or if it is an element in a shell script. Any initial values needed in a script written on the command-line can be written as part of the program text.

Consider the program example in the previous chapter to compute the value of a coin collection. The current prices for silver and gold were embedded in the program, which means that the program would have to be modified every time the price of either metal changed. It would be much simpler to specify the prices when the program is invoked.

The main part of the original program was written as:

```
/gold/    { num_gold++; wt_gold += $2 }
/silver/  { num_silver++; wt_silver += $2 }
END {
    val_gold   = 485 * wt_gold
    val_silver = 16 * wt_silver
    ...
```

The prices of gold and silver could be specified by variables, say, `pg` and `ps`:

```
END {
    val_gold   = pg * wt_gold
    val_silver = ps * wt_silver
    ...
```

The program would be invoked with variable initializations in the command line as follows:

```
awk -f summary.awk pg=485 ps=16 coins.txt
```

This yields the same results as before. Notice that the variable initializations are listed as `pg=485` and `ps=16`, and not `pg = 485` and `ps = 16`; including spaces is not recommended as it might confuse command-line parsing.

Data File(s)

At the end of the command line comes the data file. This is the name of the file that Awk should process with your program, like "coins.txt" in our previous examples.

Multiple data files can also be specified. Awk will scan one after another and generate a continuous output from the contents of the multiple files, as if they were just one long file.

Practice

1. If you haven't already, try running the program from "Field Separator" to list all users. See what happens without the `-F:`. (If you're not using Unix or Linux, sorry; it won't work.)
2. Write an Awk program to convert "coins.txt" (from the previous chapters) into a tab-delimited file. This will require "piping", which varies depending on your system, but you should probably write something like `> tabcoins.txt` to send Awk's output to a new file instead of the screen.
3. Now, rerun "summary.awk" with `-F'\t'`. The single-quotes are needed so that Awk will process "\t" as a tab rather that the two characters "\" and "t". Fields can now contain spaces without harming the output. Try changing some metals to "pure gold" or "98% pure silver" to see that it works.
4. Experiment with some of the other command-line options, like multiple input files.
5. Write a program that acts as a simple calculator. It does not need an input file; let it receive input from the keyboard. The input should be two numbers with an operator (like + or -) in between, all separated by spaces.

Match lines containing these patterns and output the result.

Open this box if you need a hint for #5.

Your program should contain lines like this:

```
$2=="+" { print ($1 + $3) }
```

In the next chapter, you will be introduced to Awk's most notable feature: pattern matching.

Search Patterns (1)

As you already know, Awk goes line-by-line through a file, and for each line that matches the *search pattern*, it executes a block of statements. So far, we have only used very simple search patterns, like `/gold/`, but you will now learn advanced search patterns. The search patterns on this page are called **regular expressions**. A regular expression is a set of rules that can match a particular string of characters.

Simple Patterns

The simplest kind of search pattern that can be specified is a simple string, enclosed in forward-slashes (`/`). For example:

```
/The/
```

This searches for any line that contains the string "The". This will *not* match "the" as Awk is *case-sensitive*, but it will match words like "There" or "Them".

This is the crudest sort of search pattern. Awk defines special characters or *meta-characters* that can be used to make the search more specific. For example, preceding the string with a `^` ("caret") tells Awk to search for the string at the beginning of the input line. For example:

```
/^The/
```

This matches any line that *begins* with the string "The". Lines that contain "The" but don't start with it will not be matched.

Similarly, following the string with a `$` matches any line that ends with search pattern. For example:

```
/The$/
```

Lines that do not end with "The" will not be matched in this example.

But what if we actually want to search the text for a character like `^` or `$`? Simple, we just precede the character with a backslash (`\`). For example:

```
/\$/
```

This will matches any line with a "$" in it.

Alternatives

There are many different meta-characters that can be used to customize the search pattern.

Character Sets

It is possible to specify a set of alternative characters using square brackets (`[]`):

```
/[Tt]he/
```

This example matches the strings "The" and "the". A range of characters can also be specified. For example:

```
/[a-z]/
```

This matches any character from "a" to "z", and:

```
/[a-zA-Z0-9]/
```

This matches any letter or number.

A range of characters can also be excluded by preceding the range with a `^`. This is different from the caret that matches the beginning of a string because it is found inside the square brackets. For example:

```
/^[^a-zA-Z0-9]/
```

This matches any line that *doesn't* start with a letter or digit. You can actually include a caret in a character set by making sure that it is not the first character listed.

Alternation

A vertical bar (`|`) allows regular expressions to be logically OR'ed. For example:

```
/(^Germany)|(^Netherlands)/
```

This matches lines that start with the word "Germany" *or* the word "Netherlands". Notice how parentheses are used to group the two expressions.

Wildcards

The dot (`.`) allows "wildcard" matching, meaning it can be used to specify any arbitrary character. For example:

```
/f.n/
```

This will matches "fan", "fun", "fin", but also "fxn", "f4n", and any other string that has an f, one character, then an n.

Repetition

This use of the dot wildcard should be familiar to UNIX shell users, but Awk interprets the `*` wildcard in a subtly different way. In the UNIX shell, the `*` substitutes for a string of arbitrary characters of any length, including zero, while in Awk the `*` simply matches zero or more repetitions of the *previous* character or expression. For example, "`a*`" would match "a", "aa", "aaa", and so on. That means that ".*" will match *any* string of characters. As a more complicated example,

```
/(ab|c)*/
```

This matches "ab", "abab", "ababab", "c", "cc", "ccc", and even "abc", "ababc", "cabcababc", or any other similar combination.

There are other characters that allow matches with repeated characters. A `?` matches exactly zero or one occurrences of the previous regular expression, while a `+` matches one or more occurrences of the previous regular expression. For example:

```
/^[+-]?[0-9]+$/
```

This matches any line that consists only of a (possibly signed) integer number. This is a somewhat confusing example and it is helpful to break it down by parts:

- `^` Find string at beginning of line.
- `[-+]?` Specify possible "-" or "+" sign for number.
- `[0-9]+` Specify at least one digit "0" through "9".
- `$` Specify that the line ends with the number.

Specific Repetition

Though this syntax is defined by the Single Unix Specification many awk don't implement this feature. With GNU Awk up to version 4.0 you need --posix or --re-interval option to enable it.

If a regular expression is to be matched a particular number of times, curly brackets (`{}`) can be used. For example:

```
/f[eo]{2}t/
```

This matches "foot" or "feet".

To specify that the number is a minimum, follow it with a comma.

```
/[0-9]{3,}/
```

This matches a number with at least three digits. It is a much easier way of writing

```
/[0-9][0-9][0-9]+/
```

Two comma-separated number can be placed within the curly brackets to specify a range.

```
/^[0-9]{4,7}$/
```

This matches a line consisting of a number with 4, 5, 6, or 7 digits. Too many or not enough digits, and it won't match.

Practice

Unless you are already familiar with regular expressions, you should try writing a few of your own. To see if you get it right, make a quick Awk program that prints any line matching the expression, and test it out.

1. Write a regular expression that matches postal codes. A U.S. ZIP code consists of 5 digits and an optional hyphen with 4 more digits.
2. Write a regular expression that matches any number, including an optional decimal point followed by more digits.
3. Write a regular expression that finds e-mail addresses. Look for valid characters (letters, numbers, dots, and underscores) followed by an "@", then more valid characters with at least one dot.
4. Write a regular expression that matches phone numbers. It should handle area codes, extension numbers, and optional hyphens and parentheses to group the digits. Make sure it doesn't match phone numbers that are too long or too short.

On the next page, you will learn other kinds of search patterns.

Search Patterns (2)

Fields and Blocks

There is more to Awk's string-searching capabilities. The search can be constrained to a single field within the input line. For example:

```
$1 ~ /^France$/
```

This searches for lines whose first field (`$1`—more on "field variables" later) is the word "France", whereas:

```
$1 !~ /^Norway$/
```

This will search for lines whose first field is *not* the word "Norway".

It is possible to search for an entire series or "block" of consecutive lines in the text, using one search pattern to match the first line in the block and another search pattern to match the last line in the block. For example:

```
/^Ireland/,/^Summary/
```

This matches a block of text whose first line begins with "Ireland" and whose last line begins with "Summary".

Here's how it works: once `/^Ireland/` is matched, all following lines of text will be automatically matched until `/^Summary/` is matched. At that point, the matching stops. If a line beginning with "Summary" is not found, everything after "Ireland" will be matched through the end of the file.

Beyond Regular Expressions

There is no need for the search pattern to be a regular expression. It can be a wide variety of other expressions as well. For example:

```
NR == 10
```

This matches line 10. Lines are numbered beginning with 1. `NR` is, as explained in the overview, a count of the lines searched by Awk, and `==` is the "equality" operator. Similarly:

```
NR == 10,NR == 20
```

This matches lines 10 through 20 in the input file.

Comparison Operators

Awk supports search patterns using a full range of comparison operations:

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

For example,

```
NF == 0
```

This matches all blank lines, or those whose number of fields is zero.

```
$1 == "France"
```

This is a string comparison that matches any line whose first field is the string "France". The astute reader may notice that this example seems to do the same thing as a the previous example:

```
$1 ~ /^France$/
```

In fact, both examples do the same thing, but in the example immediately above the `^` and `$` meta-characters had to be used in the regular expression to specify a match with the entire first field; without them, it would match such strings as "FranceFour", "NewFrance", and so on. The string expression matches only to "France".

Logic Operators

It is also possible to combine several search patterns with the `&&` (AND) and `||` (OR) operators. For example:

```
((NR >= 30) && ($1 == "France")) || ($1 == "Norway")
```

This matches any line past the 30th that begins with "France", or any line that begins with "Norway". If a line begins with "France", but it's before the 30th, it will not match. *All* lines beginning with "Norway" *will* match, however.

One class of pattern-matching that wasn't listed above is performing a numeric comparison on a field variable. It can be done, of course; for example:

```
$1 == 100
```

This matches any line whose first field has a numeric value equal to 100. This is a simple thing to do and it will work fine. However, suppose we want to perform:

```
$1 < 100
```

This will *generally* work fine, but there's a nasty catch to it, which requires some explanation: if the first field of the input can be either a number or a text string, this sort of numeric comparison can give crazy results, matching on some text strings that aren't equivalent to a numeric value.

This is because Awk is a *weakly-typed* language. Its variables can store a number or a string, with Awk performing operations on each appropriately. In the case of the numeric comparison above, if `$1` contains a numeric value, Awk will perform a numeric comparison on it, as expected; but if `$1` contains a text string, Awk will perform a text comparison between the text string in `$1` and the three-letter text string "`100`". This will work fine for a simple test of equality or inequality, since the numeric and string comparisons will give the same results, but it will give unexpected results for a "less than" or "greater than" comparison. Essentially, when comparing strings Awk compares their ASCII values. This is roughly equivalent to an alphabetical ("phone book style") sort. Even still, it's not perfectly alphabetical because uppercase and lowercase letters will not compare properly, and number and punctuation compare in a somewhat arbitrary way.

More about Types

Awk is not broken; it is doing what it is told to do in this case. If this problem comes up, it is possible to add a second test to the comparison to determine if the field contains a numeric value or a text string. This second test has the form:

```
(( $1 + 0 ) == $1 )
```

If `$1` contains a numeric value, the left-hand side of this expression will add 0 to it, and Awk will perform a numeric comparison that will always be true.

If $1 contains a text string that doesn't look like a number, for want of anything better to do Awk will interpret its value as 0. This means the left-hand side of the expression will evaluate to zero; because there is a non-numeric text string in `$1`, Awk will perform a string comparison that will always be false. This leads to a more workable comparison:

```
((( $1 + 0 ) == $1 ) && ( $1 > 100 ))
```

The same test could be modified to check for a text string instead of a numeric value:

```
(( $1 + 0 ) != $1 )
```

It is worthwhile to remember this trickery for the rare occasions it is needed. Weakly-typed languages are convenient, but in some unusual cases they can turn around and bite.

Test It Out

Incidentally, if there's some uncertainty as to how Awk is handling a particular sort of data, it is simple to run tests to find out for sure. For example, I wanted to see if my version of Awk could handle a hexadecimal value as would be specified in C—for example, "0xA8"—and so I simply typed in the following at the command prompt:

```
awk 'BEGIN {tv="0xA8"; print tv,tv+0}'
```

This printed "0xA8 0", which meant Awk thought that the data was strictly a string. This little example consists only of a `BEGIN` clause, allowing an Awk program to be run without specifying an input file. Such "one-liners" are convenient when playing with examples. If you are uncertain about what Awk may be doing, just try a test; it won't break anything.

Practice

1. Write an Awk program that prints any line with less than 5 words, unless it starts with an asterisk.
2. Write an Awk program that prints every line beginning with a number.
3. Write an Awk program that scans a line-numbered text file for errors. It should print out any line that is missing a line number and any line that is numbered incorrectly, along with the actual line number.

On the next page, you'll learn some of the finer points of strings and numbers.

Numbers and Strings

Numbers

Numbers can be expressed in Awk as either decimal integers or floating-point quantities. For example:

- `789`
- `3.141592654`
- `+67`
- `+4.6E3`
- `-34`
- `-2.1e-2`

There is no provision for specifying values in other bases, such as hex or octal; however, as will be shown later, it is possible to output them from Awk in hex or octal format.

Strings

Strings are expressed in double quotes. For example:

- `"All work and no play makes Jack a homicidal maniac!"`
- `"1987A1"`
- `"do re mi fa so la ti do"`

Awk also supports null strings, which are represented by empty quotes: `""`.

Like the C programming language, it is possible in Awk to specify a character by its three-digit octal code (preceded by a backslash).

There are various "special" characters that can be embedded into strings:

- `\n` Newline (line feed)
- `\t` Horizontal tab (aligns to next column of 8 spaces)
- `\b` Backspace
- `\r` Carriage return
- `\f` Form feed (possibly causes a new page, if printing)

A double-quote (`"`) can be embedded in a string by preceding it with a backslash, and a backslash can be embedded in a string by typing it in twice: `\\`. If a backslash is used with other characters (say, `\m`), it is simply treated as a normal character.

Variables

Types and Initialization

As already mentioned, Awk supports both user-defined variables and its own predefined variables. Any identifier beginning with a letter and consisting of alphanumeric characters or underscores (_) can be used as a variable name, provided it does not conflict with Awk's reserved words. Obviously, spaces are not allowed within a variable name; this would create too much confusion. Beware that using a reserved word is a common bug when building Awk programs, so if a program blows up on a seemingly inoffensive word, try changing it to something more unusual and see if the problem goes away.

There is no need to declare variables, and in fact it can't be done, though it is a good idea in an elaborate Awk program to initialize variables in the `BEGIN` clause to make them obvious and to make sure they have proper initial values. Relying on default values is a bad habit in any programming language, although in Awk all variables begin with a value of zero (if used as a number) or an empty string. The fact that variables aren't declared in Awk can also lead to some odd bugs, for example by misspelling the name of a variable and not realizing that this has created a second, different variable that is out of sync with the rest of the program.

Also as mentioned, Awk is weakly typed. Variables have no data type, so they can be used to store either string or numeric values; string operations on variables will give a string result and numeric operations will give a numeric result. If a text string doesn't look like a number, it will simply be regarded as 0 in a numeric operation. Awk can sometimes cause confusion because of this issue, so it is important for the programmer to remember it and avoid possible traps. For example:

- `var = 1776`
- `var = "1776"`

Both examples are the same—they both load the value 1776 into the variable named `var`. This can be treated as a numeric value in calculations in either case, and string operations can be performed on it as well. If `var` is loaded up with a text string of the form:

```
var = "somestring"
```

String operations can be performed on it, but it will evaluate to a 0 in numeric operations. If this example is changed as follows:

```
var = somestring
```

Now, this will always return 0 for both string and numeric operations—because Awk thinks `somestring` without quotes is the name of an uninitialized variable. Incidentally, an uninitialized variable can be tested for a value of 0:

```
var == 0
```

This tests "true" if `var` hasn't been initialized; but, oddly, an attempt to `print` an uninitialized variable gives nothing. For example:

```
print something
```

This simply prints a blank line, whereas:

```
something = 0; print something
```

This prints a "0".

Arrays and Strings

Unlike many other languages, an Awk string variable is *not* represented as a one-dimensional array of characters. However, it is possible to use the `substr()` function to access the characters within a string. More info about arrays and string-handling functions will come later.

Built-In Variables

Awk's built-in variables include the field variables—`$1`, `$2`, `$3`, and so on (`$0` is the entire line)—that break a line of text into individual words or pieces called *fields*. Soon, we will see how slightly more advanced Awk programs can manipulate multi-line data, such as a list of mailing addresses.

Nevertheless, Awk also has several built-in variables. Some of these can be changed by using the assignment operator. For example, writing `FS=":"` will change the field separator to a colon. From that point forward, the field variables will refer to each colon-separated part of the current line.

- `NR`: Keeps a current count of the number of input records. Remember that *records* are usually lines; Awk performs the pattern/action statements once for each record in a file.
- `NF`: Keeps a count of the number of fields within the current input record. Remember that *fields* are space-separated words, by default, but they are essentially the "columns" of data if your input file is formatted like a table. The last field of the input line can be accessed with `$NF`.
- `FILENAME`: Contains the name of the current input file.
- `FS`: Contains the *field separator* character used to divide fields on the input line. The default is "white space", meaning space and tab characters. `FS` can be reassigned to another character (typically in `BEGIN`) to change the field separator.
- `RS`: Stores the current *record separator* character. Since, by default, an input line is the input record, the default record separator character is a newline. By setting `FS` to a newline and `RS` to a blank line (`RS=""`), you can process multi-line data. This would be used for, say, a list of addresses (with each address taking several lines).
- `OFS`: Stores the *output field separator*, which separates the fields when Awk prints them. The default is a blank space. Whenever `print` has several parameters separated with commas, it will print the value of `OFS` in between each parameter.
- `ORS`: Stores the *output record separator*, which separates the output lines when Awk prints them. The default is a newline character. `print` automatically outputs the contents of `ORS` at the end of whatever it is given to print.
- `OFMT`: Stores the format for numeric output. The default format is "`%.6g`", which will be explained when `printf` is discussed.
- `ARGC`: The number of command-line arguments present.
- `ARGV`: The list of command-line arguments.

Changing Variables

By the way, values can be loaded into field variables; they aren't read-only. For example:

```
$2 = "NewText"
```

This changes the second text field in the input line to "NewText". It will not modify the input files; don't worry. This can be used as a trick to perform a modification on the lines of an input file and then simply print the lines using `print` without any parameters.

Again, all variables can be modified, although some of the built-in variables will not produce the expected effect. You can, for instance, change the value of `FILENAME`, but it will not load a new file. Awk simply continues normally, but if you access `FILENAME` the new value will be there. Same for `NR` and `NF`—changing their values will affect your program if it reads those variables, but it won't affect Awk's behavior.

Practice

- Write the address book program. You'll need to set `FS` to a newline and `RS` to a blank line. Your program should read the multi-line input and output it in single-line format.
- Write a program that reads a list of numbers and outputs them in a different format. Each input line should begin with a character such as a comma or hyphen, followed by a space and then up to five numbers (space-separated). Your program should output these numbers with the new separator (given at the beginning of the line) in between the numbers. You'll have to modify `OFS` for each line of input.

Continue to the next page to learn about Awk's powerful *associative arrays*.

Arrays

Introduction

Awk also permits the use of arrays. Those who have programmed before are already familiar with arrays. For those who haven't, an *array* is simply a single variable that holds multiple values, similar to a list. The naming convention is the same as it is for variables, and, as with variables, the array does not have to be declared.

Awk arrays can only have one dimension; the first index is 1. Array elements are identified by an index, contained in square brackets. For example:

```
some_array[1] = "Hello"
some_array[2] = "Everybody"
some_array[3] = "!"
print some_array[1], some_array[2], some_array[3]
```

The number inside the brackets is the *index*. It selects a specific entry, or *element*, within the array, which can then be created, accessed, or modified just like an ordinary variable.

Associative Arrays

This is where people familiar with C-style arrays learn something new. Awk arrays are interesting because they are actually *associative arrays*. The indexes are actually strings, so associative arrays work more like a dictionary than a numbered list. For example, an array could be used to tally the money owed by a set of debtors, as follows:

```
debts["Kim"] = 50
debts["Roberto"] += 70
debts["Vic"] -= 30
print "Vic paid 30 dollars, but still owes", debts["Vic"]
```

There are a few differences between C-style arrays (which behave like a numbered list) and associate arrays (which behave like a dictionary):

C-Style Array	Associative Array
• Fixed length • Integer index • Index starts with 0 • Multi-dimensional	• Variable length • String index • Sparse and no order • Single-dimensional

Awk only has associative arrays, never C-style arrays. This comparison is only for people who have learned arrays from a different programming language.

Array Details

Let's review some of the more specific features of Awk's arrays

Variable Length

An array in Awk can grow and shrink throughout the course of the program. Whenever you access an index that Awk hasn't seen before, the new entry gets created automatically. There is no need to let Awk know how many elements you plan on using.

```
message[1]="Have a nice"
message[2]="day."
print message[1], message[2], message[3]
```

In this example, elements 1 and 2 within the array `message` are created the moment we assigned a value to them. In the last line, `message[3]` is accessed even though it hasn't been created. Awk will just ignore `message[3]` and print an empty string (so nothing appears).

Furthermore, elements can be deleted from an array. The following is an extension to the above example:

```
delete message[2]
message[3]="night."
print message[1], message[2], message[3]
```

Now, `message[2]` no longer exists. It's as if you never gave it a value in the first place, so Awk treats the mentioning of it as an empty string. If you ran both examples together, the result would be:

```
Have a nice day.
Have a nice night.
```

(Notice how the commas within the `print` statement add spaces between the array elements. This can be changed by setting the built-in variable `OFS`.)

Deletion

Some implementation, like gawk or mawk, also let the programmer to delete a whole array rather than individual elements. After

```
delete message
```

the array `message` does not exist anymore.

String Index

You've already seen that Awk arrays use strings to select each element of an array, much like a dictionary.

```
translate["table"] = "mesa"
translate["chair"] = "silla"
translate["good"] = "bueno"
```

However, numbers are perfectly acceptable. As always, Awk simply converts the numbers into a string when necessary.

```
translate[1] = "uno"
translate[5] = "cinco"
```

Things can get tricky, however, when arrays are accessed with decimal numbers.

```
problems[ (1/3) ] = "one third"
```

Could you access this element with `problems[0.333]`? Nope. It depends on the contents of the built-in variable `OFMT`, which tells Awk how to convert numbers into strings. A specific number of decimal places will be converted, the rest thrown away. In general, try to avoid indexes with decimal values, unless you are very careful to use the correct format (which can be changed).

Sparseness and Lack of Order

Awk arrays are *sparse*, meaning that you can have element 1 and element 3 without having element 2. This is obvious—Awk uses string indexes, so it makes no distinction about numbered elements.

More importantly, the elements in an associative array are not stored in any particular order.

There are two useful commands that allow you to check the elements within an array. We will learn more about them in the upcoming chapters, but for now let's look at some examples.

```
if( "Kane" in debts )
    print "Kane owes", debts["Kane"]
```

```
for( person in debts )
    print person, "owes", debts[person]
```

Looking back to the introduction (where a `debts` array was created to associate people's names with an amount of money), we can see that Awk provides some useful commands to access arrays. The first one, `if in`, lets us check if a particular element has been defined, then execute code based on that result. The second one, `for in`, lets us create a temporary variable (called `person` in this example) and repeat a statement for every element within an array.

Play around with these examples to see how Awk doesn't necessarily maintain a specific order within its arrays. Fortunately, this never really turns out to be a problem.

Dimensions

Awk arrays are only *single-dimensional*. That means there is exactly one index. However, there is a built-in variable called `SUBSEP`, which equals `"@"` unless you change it. If you wish to create a *multi-dimensional* array, in which there are two or more indexes for each element, you can separate them with a comma.

```
array["NY", "capital"] = "Albany"
array["NY", "big city"] = "New York City"
array["OR", "capital"] = "Salem"
array["OR", "big city"] = "Portland"
```

These lines of code are exactly equal to:

```
array["NY@capital"] = "Albany"
array["NY@big city"] = "New York City"
array["OR@capital"] = "Salem"
array["OR@big city"] = "Portland"
```

This is just a quick demo of multi-dimensional arrays. As you can see, these aren't really multi-dimensional; rather they are single-dimensional with a special separator. Multi-dimensional arrays won't be explored any further here because there are several technicalities that must be understood. You can still write useful Awk programs without them, but if you are curious about multi-dimensional arrays, feel free to consult your Awk manual (or just play around and see what works).

Practice

1. Update the "coins" program that we wrote in the beginning of this book. Use arrays to keep a tally of the number of coins by country. Display the results along with the summary.
2. Write the debtors program. It should scan a log file that lists transactions like "Jim owes 50" and "Kim paid 30". Using an associative array, keep a running total of all the money that people have borrowed and paid. Make sure that a person can appear several times within the file, and their debt will be updated appropriately. At the `END`, list everyone and their total.
3. Improve the program in #2 to delete a person's name from the array if they have paid everything that they owe. This way, the results won't be cluttered with people who owe zero dollars.

The next page gives a quick review of all the *operators* Awk has to offer.

Operations

Relational Operators

Awk's relational operations have already been discussed. As a quick reminder, here they are:

- `<` Less than
- `<=` Less than or equal to
- `>` Greater than
- `>=` Greater than or equal to
- `==` Equal to
- `!=` Not equal to
- `~` Matches (compares a string to a regular expression)
- `!~` Does not match

Note that, unlike some languages, relational expressions in Awk do *not* return a value. They only evaluate to a true condition or a false condition. That means that a Awk program like this:

```
BEGIN {a=1; print (a==1)}
```

It doesn't print anything at all, and trying to use relational expressions as part of an arithmetic expression causes an error.

Logic Operators

To group together the relational operator into more complex expressions, Awk provides three logic (or *Boolean*) operators:

- `&&` And (reports "true" if both sides are true)
- `||` Or (reports "true" if either side, or both, are true)
- `!` Not (Reverses true/false of the following expression)

Arithmetic Operators

Awk uses the standard four arithmetic operators:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `^` Exponentiation (`**` may also work)
- `%` Remainder

All computations are performed in floating-point. They are performed with the expected order of operations.

Increments

There are increment and decrement operators:

- `++` Increment
- `--` Decrement

The position of these operators with respect to the variable they operate on is important. If ++ *precedes* a variable, that variable is incremented *before* it is used in some other operation. For example:

```
BEGIN {x=3; print ++x}
```

This will print 4. If ++ *follows* a variable, that variable is incremented *after* it is used in some other operation. For example:

```
BEGIN {x=3; print x++}
```

This will print 3, but `x` will equal four from that point on. Similar remarks apply to `--`. Of course, if the variable being incremented or decremented is not part of some other operation at that time, it makes no difference where the operator is placed.

Compound Assignments

Awk also allows the following shorthand operations for modifying the value of a variable:

```
x += 2
x = x + 2
```

```
x -= 2
x = x - 2
```

You get the idea. This shortcut is available for all of the arithmetic operations (`+= -= *= /= ^= %=`).

Concatenation

There is only one unique string operation: concatenation. Two strings can be easily concatenated by placing them consecutively on the same line. Only a space needs to separate them. For example:

```
BEGIN {string = "Super" "power"; print string}
```

This prints:

```
Superpower
```

The strings can be concatenated even if they are variables. This produces the same result as above:

```
BEGIN {a = "Super"; b = "power"; print (a b)}
```

The parentheses might not be necessary, but they are often used to make sure that the concatenation is interpreted correctly.

The Conditional

There is an interesting operator called the *conditional operator*. It has two parts. Look at this example:

```
print ( price > 500 ? "too expensive" : "cheap" )
```

This will print either "too expensive" or "cheap" depending on the value of `price`. The condition before the question mark is evaluated. If true, the first statement is executed, and if false the second is executed. The statements are separated by a colon.

Standard Functions

Below is the list of Awk functions. Arguments which can be omitted are in square brackets.

Numerical functions

Numerical functions work with numbers. All of them return a number and have only numerical parameters, or no parameters at all.

- **int(x)** returns x rounded towards zero. For example, int(-3.9) returns -3, while int(3.9) returns 3.
- **sqrt(x)** returns $\sqrt{x}$.
- **exp(x)** returns e^x.
- **log(x)** returns natural logarithm of x.
- **sin(x)** returns $\sin x$, in radians.
- **cos(x)** returns $\cos x$, in radians.
- **atan2(y,x)** is similar to the same function in C or C++, see below for more information.
- **rand()** returns a pseudo-random number in the [0,1) interval (that is, it is at least 0 and less then 1). If the same program runs more than once, some implementations (i. e. GNU Awk 3.1.8) produce the same series of random numbers, while others (i. e. mawk 1.3.3) each time produce a different series.
- **srand([x])** sets x as a random number seed. Without parameters, it uses time of day to set a seed. It returns the previous seed.

atan2

atan2(y,x) returns the angle α , in radians, such that:

- $-\pi < \alpha \leq \pi$
- $x = \sqrt{x^2 + y^2} \cos \alpha$
- $y = \sqrt{x^2 + y^2} \sin \alpha$

The formulas are

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ \frac{\pi}{2} \operatorname{sgn} x & x = 0 \end{cases}$$

String functions

String functions work with strings. All of them have at least one string parameter, which sometimes can be omitted. For most of them, all parameters are strings and/or regular expressions. Most of them return a string.

Note that in Awk strings, characters are numbered from 1. For example, in the string "cat", the character number 1 is "c", the character number 2 is "a", the character number 3 is "t".

Below, **s** and **t** are strings, **regexp** is a regular expression.

- **length([s])** returns the number of characters in s (in $0 by default).
- **split(s, A [,regexp])** splits s into array A of fields, using regexp (FS by default) as a delimiter. If regexp is empty ("" or //), some implementations (i. e. gawk) split it to characters, others (i. e. mawk 1.3.3) return array of one element, which contains the whole string s. Returns the number of fields.
- **sprintf(format [,expression, ..., expression])** - formats the expressions similar to C and C++ function sprintf, returns the result. See wikipedia article for more information.
- **gsub(regexp, s [,t])** - in t ($0 by default), substitutes all matches of regexp by s. Returns the number of substitutions.
- **sub(regexp, s [,t])** - in t ($0 by default), substitutes the first match of regexp by s. If there is no match, does nothing and returns 0, otherwise returns 1.
  - In sub() and gsub(), **&** in the string s means the whole matched text. Use **\&** for the literal **&** . Note that **\&** should be typed as **\\&** in order to avoid the backslash escape in Awk strings.
- **index(s, t)** - returns the index of the first occurrence of t in s, or 0 if s does not contain t. Example: index("hahaha", "ah") returns 2, while index("hahaha", "foo") returns 0.
- **match(s, regexp)** - like index, but seeks a regular expression rather than a string. Also, sets RSTART to the return value, RLENGTH to the length of the matched substring, or -1 if no match. If empty string is matched, RSTART is set to the index of the first character after the match (length(s)+1 if the match is at the end), and RLENGTH is set to 0.
- **tolower(s)** - returns the copy of s with uppercase characters turned to lowercase.
- **toupper(s)** - returns the copy of s with lowercase characters turned to uppercase.

GNU Awk extensions

String function

gensub(regexp, s, h [, t]) replaces the h-th match of regexp by s in the string t ($0 by default). For example, gensub(/o/, "O", 3, t) replaces the third "o" by "O" in t.

- Unlike sub() and gsub(), it returns the result, while the string t remains unchanged.
- If h is a string starting with **g** or **G**, replaces all matches.
- Like in sub() and gsub(), **&** in the string s means the whole matched text. Use **\&** for the literal **&** .
  - **\&** should be typed as **\\&** in order to avoid the backslash escape in awk strings.
- Unlike sub() and gsub(), **\0** in the string s means the same as **&**, while **\1** ... **\9** mean 1-st ... 9-th parenthesized subexpression.
  - Similarly to above, **\0** ... **\9** should be typed as **\\0** ... **\\9** for the same reason.

Examples:

- **print(gensub(/o/, "O", 3, "cooperation"))** prints **cooperatiOn**
- **print(gensub(/o/, "O", "g", "cooperation"))** prints **cOOperatiOn**
- **print(gensub(/o+/, "(&)", "g", "cooperation"))** prints **c(oo)perati(o)n**
- **print(gensub(/(o+)(p+)/, "<[\\1](\\2)>", "g", "I oppose any cooperation")** prints **I <[o](pp)>ose any c<[oo](p)>eration**

Array functions

Below, A and B are arrays.

- **length(A)** returns the length of A.
- **asort(A[,B])** - if B is not given, sorts A. The indices of A are replaced by sequential integers starting with 1. If B is given, copies A to B, then sorts B as above, while A remains unchanged. Returns the length of A.
- **asorti(A[,B])** - if B is not given, discard values of A and sorts its indices. The sorted indices become the new values, and sequential integers starting with 1 become the new indices. Like in the previous case, if B is given, copies A to B, then sorts B's indices as above, while A remains unchanged. Returns the length of A.

Other functions

GNU Awk also has:

- time functions
- bit manipulation functions
- internationalization functions.

See the man page (man gawk) for more information.

Control Structures

Awk supports control structures similar to those used in C, including:

```
   if ... else    while    for
```

The syntax of "if ... else" is:

```
   if (<condition>) <action 1> [else <action 2>]
```

The "else" clause is optional. The "condition" can be any expression discussed in the section on pattern matching, including matches with regular expressions. For example, consider the following Awk program:

```
{if ($1=="green") print "GO";    else if ($1=="yellow") print "SLOW DOWN";    else if ($1=="red") print "STOP";    else print "SAY WHAT?";}
```

By the way, for test purposes this program can be invoked as:

```
   echo "red" | awk -f pgm.txt
```

-- where "pgm.txt" is a text file containing the program.

The "action" clauses can consist of multiple statements, contained by curly brackets ("{}").

The syntax for "while" is:

```
   while (<condition>) <action>
```

The "action" is performed as long the "condition" tests true, and the

"condition" is tested before each iteration. The conditions are the same as for the "if ... else" construct. For example, since by default an Awk variable has a value of 0, the following Awk program could print the numbers from 1 to 20:

```
   BEGIN {while(++x<=20) print x}
```

- The "for" loop is more flexible. It has the syntax:

```
   for (<initial action>;<condition>;<end-of-loop action>) <action>
```

For example, the following "for" loop prints the numbers 10 through 20 in increments of 2:

```
   BEGIN {for (i=10; i<=20; i+=2) print i}
```

This is equivalent to:

```
   i=10    while (i<=20) {       print i;       i+=2;}
```

The C programming language has a similar "for" construct, with an interesting feature in that multiple actions can be taken in both the initialization and end-of-loop actions, simply by separating the actions with a comma. Most implementations of Awk, unfortunately, do not support this feature.

The "for" loop has an alternate syntax, used when scanning through an array:

```
   for (<variable> in <array>) <action>
```

Given the example used earlier:

```
my_string = "joe:frank:harry:bill:bob:sil";    split(my_string, names, ":");
```

-- then the names could be printed with the following statement:

```
   for (idx in names) print idx, names[idx];
```

This yields:

```
   2 frank    3 harry    4 bill    5 bob    6 sil    1 joe
```

Notice that the names are *not* printed in the proper order. One of the characteristics of this type of "for" loop is that the array is not scanned in a predictable order.

- Awk defines four unconditional control statements: "break", "continue", "next", and "exit". "Break" and "continue" are strictly associated with the

"while" and "for" loops:

- break: Causes a jump out of the loop.
- continue: Forces the next iteration of the loop.

"Next" and "exit" control Awk's input scanning:

- next: Causes Awk to immediately get another line of input and begin scanning it from the first match statement.
- exit: Causes Awk to end reading its input and execute END operations, if any are specified.

Output with print and printf

The simplest output statement is the by-now familiar "print" statement. There's not too much to it:

- "Print" by itself prints the input line.
- "Print" with one argument prints the argument.
- "Print" with multiple arguments prints all the arguments, separated by spaces (or other specified OFS) when the arguments are separated by commas, or concatenated when the arguments are separated by spaces.
- The "printf()" (formatted print) function is much more flexible, and trickier. It has the syntax:

```
   printf(<string>,<expression list>)
```

The "string" can be a normal string of characters:

```
printf("Hi, there!")
```

This prints "Hi, there!" to the display, just like "print" would, with one slight difference: the cursor remains at the end of the text, instead of skipping to the next line, as it would with "print". A "newline" code ("\n") has to be added to force "printf()" to skip to the next line:

```
printf("Hi, there!\n")
```

So far, "printf()" looks like a step backward from "print", and for do dumb things like this, it is. However, "printf()" is useful when precise control over the appearance of the output is required. The trick is that the string can contain format or "conversion" codes to control the results of the expressions in the expression list. For example, the following program:

```
BEGIN {x = 35; printf("x = %d decimal, %x hex, %o octal.\n",x,x,x)}
```

-- prints:

```
   x = 35 decimal, 23 hex, 43 octal.
```

The format codes in this example include: "%d" (specifying decimal output), "%x" (specifying hexadecimal output), and "%o" (specifying octal output). The "printf()" function substitutes the three variables in the expression list for these format codes on output.

- The format codes are highly flexible and their use can be a bit confusing. The "d" format code prints a number in decimal format. The output is an integer, even if the number is a real, like 3.14159. Trying to print a string with this format code results in a "0" output. For example:

```
   x = 35;     printf("x = %d\n",x)       yields:  x = 35    x =
3.1415; printf("x = %d\n",x)       yields:  x = 3    x = "TEST";
printf("x = %d\n",x)       yields:  x = 0
```

- The "o" format code prints a number in octal format. Other than that, this format code behaves exactly as does the "%d" format specifier. For example:

```
   x = 255; printf("x = %o\n",x)          yields:  x = 377
```

- The "x" format code prints a number in hexadecimal format. Other than that, this format code behaves exactly as does the "%d" format specifier. For example:

```
x = 197; printf("x = %x\n",x)          yields:  x = c5
```

- The "c" format code prints a character, given its numeric code. For example, the following statement outputs all the printable characters:

```
   BEGIN {for (ch=32; ch<128; ch++) printf("%c   %c\n",ch,ch+128)}
```

- The "s" format code prints a string. For example:

```
   x = "jive"; printf("string = %s\n",x)  yields:  string = jive
```

- The "e" format code prints a number in exponential format, in the default format:

```
   [-]D.DDDDDDe[+/-]DDD
```

For example:

```
   x = 3.1415; printf("x = %e\n",x)       yields:  x = 3.141500e+000
```

- The "f" format code prints a number in floating-point format, in the default format:

```
   [-]D.DDDDDD
```

For example:

```
   x = 3.1415; printf("x = %f\n",x)       yields:  f = 3.141500
```

- The "g" format code prints a number in exponential or floating-point format, whichever is shortest.
- A numeric string may be inserted between the "%" and the format code to specify greater control over the output format. For example:

```
%3d    %5.2f    %08s    %-8.4s
```

This works as follows:

- The integer part of the number specifies the minimum "width", or number of spaces, the output will use, though the output may exceed that width if it is too long to fit.
- The fractional part of the number specifies either, for a string, the maximum number of characters to be printed; or, for floating-point formats, the number of digits to be printed to the right of the decimal point.
- A leading "-" specifies left-justified output. The default is right-justified output.

- A leading "0" specifies that the output be padded with leading zeroes to fill up the output field. The default is spaces.

For example, consider the output of a string:

```
   x = "Baryshnikov"    printf("[%3s]\n",x)          yields:
[Baryshnikov]    printf("[%16s]\n",x)         yields:       [
Baryshnikov]    printf("[%-16s]\n",x)        yields:       [Baryshnikov
     ]    printf("[%.3s]\n",x)         yields:       [Bar]
printf("[%16.3s]\n",x)       yields:       [             Bar]
printf("[%-16.3s]\n",x)      yields:       [Bar             ]
printf("[%016s]\n",x)        yields:       [00000Baryshnikov]
printf("[%-016s]\n",x)       yields:       [Baryshnikov     ]
```

-- or an integer:

```
   x = 312    printf("[%2d]\n",x)          yields:       [312]
printf("[%8d]\n",x)          yields:       [     312]
printf("[%-8d]\n",x)         yields:       [312     ]
printf("[%.1d]\n",x)         yields:       [312]
printf("[%08d]\n",x)         yields:       [00000312]
printf("[%-08d]\n",x)        yields:       [312     ]
```

-- or a floating-point number:

```
   x = 251.67309    printf("[%2f]\n",x)          yields:
[251.67309]    printf("[%16f]\n",x)         yields:       [
251.67309]    printf("[%-16f]\n",x)        yields:       [251.67309
      ]    printf("[%.3f]\n",x)         yields:       [251.673]
printf("[%16.3f]\n",x)       yields:       [        251.673]
printf("[%016.3f]\n",x)      yields:       [00000000251.673]
```

A Digression: The sprintf Function

While "sprintf()" is a string function, it was not discussed with the other string functions, since its syntax is virtually identical to that of "printf()". In fact, "sprintf()" acts in exactly the same way as "printf()", except that "sprintf()" assigns its output to a variable, not standard output. For example:

```
BEGIN {var = sprintf("[%8.3f]",3.141592654); print var}
```

-- yields:

```
   [   3.142]
```

Output Redirection and Pipes

The output-redirection operator ">" can be used in Awk output statements. For example:

```
   print 3 > "tfile"
```

-- creates a file named "tfile" containing the number "3". If "tfile"

already exists, its contents are overwritten. The "append" redirection operator (">>") can be used in exactly the same way. For example:

```
   print 4 >> "tfile"
```

-- tacks the number "4" to the end of "tfile". If "tfile" doesn't exist, it is created and the number "4" is appended to it.

Output redirection can be used with "printf" as well. For example:

```
   BEGIN {for (x=1; x<=50; ++x) {printf("%3d\n",x) >> "tfile"}}
```

-- dumps the numbers from 1 to 50 into "tfile".

- The output can also be "piped" into another utility with the "|" ("pipe") operator. As a trivial example, we could pipe output to the "tr" ("translate") utility to convert it to upper-case:

```
   print "This is a test!" | "tr [a-z] [A-Z]"
```

This yields:

```
   THIS IS A TEST!
```

Awk Examples, NAWK & Awk Quick Reference

Using Awk from the Command Line

The Awk programming language was designed to be simple but powerful. It allows a user to perform relatively sophisticated text-manipulation operations through Awk programs written on the command line.

For example, suppose I want to turn a document with single-spacing into a document with double-spacing. I could easily do that with the following Awk program:

```
   awk '{print ; print ""}' infile > outfile
```

Notice how single-quotes (' ') are used to allow using double-quotes (" ") within the Awk expression. This "hides" special characters from the shell. We could also do this as follows:

```
awk "{print ; print \"\"}" infile > outfile
```

-- but the single-quote method is simpler.

This program does what it supposed to, but it also doubles every blank line in the input file, which leaves a lot of empty space in the output. That's easy to fix, just tell Awk to print an extra blank line if the current line is *not* blank:

```
awk '{print ; if (NF != 0) print ""}' infile > outfile
```

- One of the problems with Awk is that it is ingenious enough to make a user want to tinker with it, and use it for tasks for which it isn't really appropriate. For example, we could use Awk to count the number of lines in a file:

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

-- but this is dumb, because the "wc (word count)" utility gives the same answer with less bother: "Use the right tool for the job."

Awk is the right tool for slightly more complicated tasks. Once I had a file containing an email distribution list. The email addresses of various different groups were placed on consecutive lines in the file, with the different groups separated by blank lines. If I wanted to quickly and reliably determine how many people were on the distribution list, I couldn't use "wc", since, it counts blank lines, but Awk handled it easily:

```
   awk 'NF != 0 {++count} END {print count}' list
```

- Another problem I ran into was determining the average size of a number of files. I was creating a set of bitmaps with a scanner and storing them on a disk. The disk started getting full and I was curious to know just how many more bitmaps I could store on the disk.

I could obtain the file sizes in bytes using "wc -c" or the "list" utility ("ls -l" or "ll"). A few tests showed that "ll" was faster. Since "ll"

lists the file size in the fifth field, all I had to do was sum up the fifth field and divide by NR. There was one slight problem, however: the first line of the output of "ll" listed the total number of sectors used, and had to be skipped.

No problem. I simply entered:

```
   ll | awk 'NR!=1 {s+=$5} END {print "Average: " s/(NR-1)}'
```

This gave me the average as about 40 KB per file.

- Awk is useful for performing simple iterative computations for which a more sophisticated language like C might prove overkill. Consider the Fibonacci sequence:

```
   1 1 2 3 5 8 13 21 34 ...
```

Each element in the sequence is constructed by adding the two previous elements together, with the first two elements defined as both "1". It's a discrete formula for exponential growth. It is very easy to use Awk to generate this sequence:

```
   awk 'BEGIN {a=1;b=1; while(++x<=10){print a; t=a;a=a+b;b=t}; exit}'
```

This generates the following output data:

```
   1    2    3    5    8    13    21    34    55    89
```

Awk Program Files

Sometimes an Awk program needs to be used repeatedly. In that case, it's simple to execute the Awk program from a shell script. For example, consider an Awk script to print each word in a file on a separate line. This could be done with a script named "words" containing:

```
   awk '{c=split($0, s); for(n=1; n<=c; ++n) print s[n] }' $1
```

"Words" could them be made executable (using "chmod +x words") and the resulting shell "program" invoked just like any other command. For example,

"words" could be invoked from the "vi" text editor as follows:

```
   :%!words
```

This would turn all the text into a list of single words.

- For another example, consider the double-spacing program mentioned previously. This could be slightly changed to accept standard input, using a "-" as described earlier, then copied into a file named "double":

```
awk '{print; if (NF != 0) print ""}' -
```

-- and then could be invoked from "vi" to double-space all the text in the editor.

- The next step would be to also allow "double" to perform the reverse operation: To take a double-spaced file and return it to single-spaced, using the option:

```
   undouble
```

The first part of the task is, of course, to design a way of stripping out the extra blank lines, without destroying the spacing of the original single-spaced file by taking out *all* the blank lines. The simplest approach would be to delete every other blank line in a continuous block of such blank lines. This won't necessarily preserve the original spacing, but it will preserve spacing in some form.

The method for achieving this is also simple, and involves using a variable named "skip". This variable is set to "1" every time a blank line is skipped, to tell the Awk program NOT to skip the next one. The scheme is as follows:

```
   BEGIN {set skip to 0}    scan the input:       if skip == 0    if
line is blank                          skip = 1
else                          print the line                        get
```

```
next line of input       if skip == 1    print the line
                       skip = 0                        get next line of
input
```

This translates directly into the following Awk program:

```
BEGIN      {skip = 0}    skip == 0  {if (NF == 0)
{skip = 1}                  else                  {print};
                 next}    skip == 1  {print;                 skip = 0;
                next}
```

This program could be placed in a separate file, named, say, "undouble.awk", with the shell script "undouble" written as:

```
   awk -f undouble.awk
```

It could also be embedded directly in the shell script, using single-quotes to enclose the program and backslashes ("\") to allow for multiple lines:

```
awk 'BEGIN      {skip = 0} \         skip == 0  {if (NF == 0)
                       {skip = 1}  \                    else
                       {print};  \                    next} \
skip == 1  {print; \                     skip = 0; \
                     next}'
```

Remember that when "\" is used to embed an Awk program in a script file, the program appears as *one* line to Awk. A semicolon *must* be used to separate commands.

For a more sophisticated example, I have a problem that when I write text documents, occasionally I'll somehow end up with the same word typed in twice: "And the result was also also that ... " Such duplicate words are hard to spot on proofreading, but it is straightforward to write an Awk program to do the job, scanning through a text file to find duplicate; printing the duplicate word and the line it is found on if a duplicate is found; or otherwise printing "no duplicates found".

```
BEGIN { dups=0; w="xy-zzy" }          { for( n=1; n<=NF; n++)
                { if ( w == $n ) { print w, "::", $0 ; dups = 1 } ; w =
 $n }          }     END   { if (dups == 0) print "No duplicates
found." }
```

The "w" variable stores each word in the file, comparing it to the next word in the file; w is initialized to "xy-zzy" since that is unlikely to be a word in the file. The "dup" variable is initialized to 0 and set to 1 if a duplicate is found; if it's still 0 at the end of the end, the program prints the "no duplicate found" message. As with the previous example, we could put this into a separate file or embed it into a script file.

- These last examples use variables to allow an Awk program to keep track of what it has been doing. Awk, as repeatedly mentioned, operates in a cycle: get a line, process it, get the next line, process it, and so on; to have an Awk program remember things between cycles, it needs to leave a little message for itself in a variable.

For example, say we want to match on a line whose first field has the value 1,000 -- but then print the *next* line. We could do that as follows:

```
   BEGIN        {flag = 0}    $1 == 1000   {flag = 1;
                   next}    flag == 1    {print;                   flag
 = 0;                  next}
```

This program sets a variable named "flag" when it finds a line starting with 1,000, and then goes and gets the next line of input. The next line of input is printed, and then "flag" is cleared so the line after that won't be printed.

If we wanted to print the next *five* lines, we could do that in much the same way using a variable named, say, "counter":

```
   BEGIN         {counter = 0}    $1 == 1000    {counter = 5;
                   next}    counter > 0   {print;
                    counter--;                   next}
```

This program initializes a variable named "counter" to 5 when it finds a line starting with 1,000; for each of the following 5 lines of input, it prints them and decrements "counter" until it is zero.

This approach can be taken to as great a level of elaboration as needed. Suppose we have a list of, say, five different actions on five lines of input, to be taken after matching a line of input; we can then create a variable named, say, "state", that stores which item in the list to perform next. The scheme is generally as follows:

```
   BEGIN {set state to 0}    scan the input:       if match        set
state to 1                       get next line of input       if state
== 1   do the first thing in the list                        state = 2
                       get next line of input       if state == 2   do
the second thing in the list                        state = 3
                       get next line of input       if state == 3   do
the third thing in the list                        state = 4
                       get next line of input       if state == 4   do
the fourth thing in the list                        state = 5
                       get next line of input       if state == 5   do
the fifth (and last) thing in the list                        state = 0
                       get next line of input
```

This is called a "state machine". In this case, it's performing a simple list of actions, but the same approach could also be used to perform a more complicated branching sequence of actions, such as we might have in a flowchart instead of a simple list.

We could assign state numbers to the blocks in the flowchart and then use if-then tests for the decision-making blocks to set the state variable to indicate which of the alternate actions should be performed next. However, few Awk programs require such complexities, and going into more elaborate examples here would probably be more confusing than it's worth. The essential thing to remember is that an awk program can leave messages for itself in a variable on one line-scan cycle to tell it what to do on later line-scan cycles.

A Note on Awk in Shell Scripts

Awk is an excellent tool for building UNIX/Linux shell scripts, but there are potential pitfalls. Say we have a scriptfile named "testscript", and it takes two filenames as parameters:

```
testscript myfile1 myfile2
```

If we're executing Awk commands from a file, handling the two filenames isn't very difficult. We can initialize variables on the command line as follows:

```
cat $1 $2 | awk -f testscript.awk f1=$1 f2=$2 > tmpfile
```

The Awk program will use two variables, "f1" and "f2", that are initialized from the script command line variables "$1" and "$2".

Where this measure gets obnoxious is when we are specifying Awk commands directly, which is preferable if possible since it reduces the number of files needed to implement a script. The problem is that "$1" and "$2" have different meanings to the scriptfile and to Awk. To the scriptfile, they are command-line parameters, but to Awk they indicate text fields in the input.

The handling of these variables depends on how Awk print fields are defined -- either enclosed in double-quotes (" ") or in single-quotes (' '). If we invoke Awk as follows:

```
awk "{ print \"This is a test: \" $1 }" $1
```

-- we won't get *anything* printed for the "$1" variable. If we instead use single-quotes to ensure that the scriptfile leaves the Awk positional variables alone, we can insert scriptfile variables by initializing them to variables on the command line:

```
awk '{ print "This is a test: " $1 " / parm2 = " f  }' f=$2 < $1
```

This provides the first field in "myfile1" as the first parameter and the name of "myfile2" as the second parameter.

Remember that Awk is relatively slow and clumsy and should not be regarded as the default tool for all scriptfile jobs. We can use "cat" to append to files, "head" and "tail" to cut off a given number of lines of text from the front or back of a file, "grep" or "fgrep" to find lines in a particular file, and "sed" to do search-replaces on the stream in the file.

Nawk

The original version of Awk was developed in 1977. It was optimized for throwing together "one-liners" or short, quick-and-dirty programs. However, some users liked Awk so much that they used it for much more complicated tasks. To quote the language's authors: "Our first reaction to a program that didn't fit on one page was shock and amazement." Some users regarded Awk as their primary programming tool, and many had in fact learned programming using Awk.

After the authors got over their initial consternation, they decided to accept the fact, and enhance Awk to make it a better general-purpose programming tool. The new version of Awk was released in 1985. The new version is often, if not always, known as Nawk ("New Awk") to distinguish it from the old one.

- Nawk incorporates several major improvements. The most important improvement is that users can define their own functions. For example, the following Nawk program implements the "signum" function:

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

    function signum(n) {
       if (n<0)
       return -1
       else if (n==0) return 0
       else
           return 1}
```

Function declarations can be placed in a program wherever a match-action clause can. All parameters are local to the function. Local variables can be defined inside the function.

- A second improvement is a new function, "getline", that allows input from files other than those specified in the command line at invocation (as well as input from pipes). "Getline" can be used in a number of ways:

```
getline                    Loads $0 from current input.
   getline myvar              Loads "myvar" from current input.
   getline myfile             Loads $0 from "myfile".
   getline myvar myfile       Loads "myvar" from "myfile".
   command | getline          Loads $0 from output of "command".
   command | getline myvar    Loads "myvar" from output of "command".
```

- A related function, "close", allows a file to be closed so it can be read from the beginning again:

```
   close("myfile")
```

- A new function, "system", allows Awk programs to invoke system commands:

```
   system("rm myfile")
```

- Command-line parameters can be interpreted using two new predefined variables, ARGC and ARGV, a mechanism instantly familiar to C programmers. ARGC ("argument count") gives the number of command-line elements, and ARGV ("argument vector") is an array whose entries store the elements individually.
- There is a new conditional-assignment expression, known as "?:", which is used as follows:

```
   status = (condition == "green")? "go" : "stop"
```

This translates to:

```
if (condition=="green") {status = "go"} else {status = "stop"}
```

This construct should also be familiar to C programmers.

- There are new math functions, such as trig and random-number functions:

```
sin(x)          Sine, with x in radians.
cos(x)          Cosine, with x in radians.
atan2(y,z)      Arctangent of y/x, in range -PI to PI.
rand()          Random number, with 0 <= number < 1.
srand()         Seed for random-number generator.
```

- There are new string functions, such as match and substitution functions:
  - match(<target string>,<search string>)
    Search the target string for the search string; return 0 if no match, return starting index of search string if match. Also sets built-in variable RSTART to the starting index, and sets built-in variable RLENGTH to the matched string's length.
  - sub(<regular expression>,<replacement string>)
    Search for first match of regular expression in $0 and substitute replacement string. This function returns the number of substitutions made, as do the other substitution functions.
  - sub(<regular expression>,<replacement string>,<target string>)
    Search for first match of regular expression in target string and substitute replacement string.
  - gsub(<regular expression>,<replacement string>)
    Search for all matches of regular expression in $0 and substitute replacement string.
  - sub(<regular expression>,<replacement string>,<target string>)
    Search for all matches of regular expression in target string and substitute replacement string.
- There is a mechanism for handling multidimensional arrays. For example, the following program creates and prints a matrix, and then prints the transposition of the matrix:

```
BEGIN {count = 1;
   for (row = 1; row <= 5; ++row) {
     for (col = 1; col <= 3; ++col) {
       printf("%4d",count);
       array[row,col] = count++; }
     printf("\n"); }
   printf("\n");

   for (col = 1; col <= 3; ++col) {
      for (row = 1; row <= 5; ++row) {
         printf("%4d",array[row,col]); }
      printf("\n"); }
   exit; }
```

This yields:

```
1   2   3    4   5   6    7   8   9    10  11  12    13  14  15

1   4   7  10  13    2   5   8  11  14    3   6   9  12  15
```

Nawk also includes a new "delete" function, which deletes array elements:

```
delete(array[count])
```

- Characters can be expressed as octal codes. "\033", for example, can be used to define an "escape" character.
- A new built-in variable, FNR, keeps track of the record number of the current file, as opposed to NR, which keeps track of the record number of the current line of input, regardless of how many files have contributed to that input. Its behavior is otherwise exactly identical to that of NR.
- While Nawk does have useful refinements, they are generally intended to support the development of complicated programs. My feeling is that Nawk represents overkill for all but the most dedicated Awk users, and in any case would require a substantial document of its own to do its capabilities justice. Those who would like to know more about Nawk are encouraged to read THE AWK PROGRAMMING LANGUAGE by Aho / Weinberger / Kernighan. This short, terse, detailed book outlines the capabilities of Nawk and provides sophisticated examples of its use.

Awk Quick Reference Guide

This final section provides a convenient lookup reference for Awk programming.

- Invoking Awk:

```
   awk [-F<ch>] {pgm} | {-f <pgm file>} [<vars>] [-|<data file>]
```

-- where:

```
   ch:          Field-separator character.    pgm:         Awk
command-line program.    pgm file:    File containing an Awk program.
    vars:        Awk variable initializations.    data file:   Input
data file.
```

- General form of Awk program:

```
   BEGIN              {<initializations>}     <search pattern
1> {<program actions>}     <search pattern 2>
{<program actions>}     ...    END                {<final
actions>}
```

- Search patterns:

```
   /<string>/     Search for string.    /^<string>/    Search for string at beginning of line.    /<string>$/    Search for string at end of line.
```

The search can be constrained to particular fields:

```
   $<field> ~ /<string>/   Search for string in specified field.    $<field> !~ /<string>/  Search for string \Inot\i in specified field.
```

Strings can be ORed in a search:

```
   /(<string1>)|(<string2>)/
```

The search can be for an entire range of lines, bounded by two strings:

```
   /<string1>/,/<string2>/
```

The search can be for any condition, such as line number, and can use the following comparison operators:

```
   == != < > <= >=
```

Different conditions can be ORed with "||" or ANDed with "&&".

```
   [<charlist or range>]   Match on any character in list or
range.    [^<charlist or range>]  Match on any character not in
list or range.    .                       Match any single character.
    *                      Match 0 or more occurrences of preceding
string.    ?                       Match 0 or 1 occurrences of
preceding string.    +                       Match 1 or more
occurrences of preceding string.
```

If a metacharacter is part of the search string, it can be "escaped" by preceding it with a "\".

- Special characters:

```
   \n     Newline (line feed).
```

```
     Backspace.    \r     Carriage return.    \f     Form feed. A "\" can be embedded in a string by entering it twice: "\\".
```

- Built-in variables:

```
   $0; $1,$2,$3,...  Field variables.    NR                Number of
records (lines).    NF                Number of fields.    FILENAME
          Current input filename.    FS                Field separator
character (default: " ").    RS                Record separator
character (default: "\n").    OFS                Output field separator
(default: " ").    ORS                Output record separator (default:
"\n").    OFMT              Output format (default: "%.6g").
```

- Arithmetic operations:

```
+   Addition.    -   Subtraction.    *   Multiplication.    /   Division.    %   Mod.    ++  Increment.    --  Decrement.
```

Shorthand assignments:

```
   x += 2  -- is the same as:  x = x + 2    x -= 2  -- is the same as:
  x = x - 2    x *= 2  -- is the same as:  x = x * 2    x /= 2  -- is
the same as:  x = x / 2    x %= 2  -- is the same as:  x = x % 2
```

- The only unique string operation is concatenation, which is performed simply by listing two strings connected by a blank space.
- Arithmetic functions:

```
   sqrt()     Square root.    log()      Base \Ie\i log.    exp()      Power of \Ie\i.    int()      Integer part of argument.
```

- String functions:
- length()

```
  Length of string.
```

- substr(<string>,<start of substring>,<max length of substring>)

```
  Get substring.
```

- split(<string>,<array>,[<field separator>])

```
  Split string into array, with initial array index being 1.
```

- index(<target string>,<search string>)

```
  Find index of search string in target string.
```

- sprintf()

```
  Perform formatted print into string.
```

- Control structures:

```
   if (<condition>) <action 1> [else <action 2>]    while (<condition>) <action>

   for (<initial action>;<condition>;<end-of-loop action>) <action>
```

Scanning through an associative array with "for":

```
   for (<variable> in <array>) <action>
```

Unconditional control statements:

```
   break       Break out of "while" or "for" loop.    continue
Perform next iteration of "while" or "for" loop.    next        Get and
 scan next line of input.    exit        Finish reading input and
perform END statements.
```

- Print:

```
  print <i1>, <i2>, ...   Print items separated by OFS; end with newline.    print <i1> <i2> ...    Print items concatenated; end with newline.
```

- Printf():

General format:

```
   printf(<string with format codes>,[<parameters>])
```

Newlines must be explicitly specified with a "\n".

General form of format code:

```
   %[<number>]<format code>
```

The optional "number" can consist of:

- A leading "-" for left-justified output.
- An integer part that specifies the minimum output width. (A leading "0"

```
  causes the output to be padded with zeroes.)
```

- A fractional part that specifies either the maximum number of characters to be printed (for a string), or the number of digits to be printed to the right of the decimal point (for floating-point formats).

The format codes are:

```
   d    Prints a number in decimal format.    o    Prints a number in
octal format.    x    Prints a number in hexadecimal format.    c
Prints a character, given its numeric code.    s    Prints a string.
    e    Prints a number in exponential format.    f    Prints a number
 in floating-point format.    g    Prints a number in exponential or
floating-point format.
```

- Awk can perform output redirection (using ">" and ">>") and piping (using "|") from both "print" and "printf".

Revision History

```
   v1.0   / 11 mar 90 / gvg    v1.1   / 29 nov 94 / gvg / Cosmetic
rewrite.     v1.2   / 12 oct 95 / gvg / Web rewrite, added stuff on
shell scripts.    v1.3   / 15 jan 99 / gvg / Minor cosmetic update.
v1.0.4 / 01 jan 02 / gvg / Minor cosmetic update.    v1.0.5 / 01 jan 04
 / gvg / Minor cosmetic update.    v1.0.6 / 01 may 04 / gvg / Added
comments on state variables.    v1.0.7 / 01 jun 04 / gvg / Added
comments on numeric / string comparisons.    v1.0.8 / 01 jul 04 / gvg /
 Corrected an obnoxious typo error.    v1.0.9 / 01 oct 04 / gvg /
Corrected another tweaky error.    v1.1.0 / 01 sep 06 / gvg / Minor
update, changed title to "Primer".    v1.1.1 / 01 aug 08 / gvg / Minor
cosmetic update.
```

Resources and Licensing

Resources

Further Reading

- The AWK Manual [1]: A very detailed website that explains everything there is to know about Awk.

References

[1] http://people.cs.uu.nl/piet/docs/nawk/nawk_toc.html

Licensing

Licensing

The original version of this book was written by Greg Goebel, and is available at http://www.vectorsite.net/tsawk.html.This original version has been released by its author into the public domain.

Newer revisions here on Wikibooks have been licensed under the following license:

Article Sources and Contributors

Awk Overview *Source*: https://en.wikibooks.org/w/index.php?oldid=1273434 *Contributors*: Whiteknight

Awk Command-Line Examples *Source*: https://en.wikibooks.org/w/index.php?oldid=2430783 *Contributors*: NipplesMeCool, Whiteknight, Xania, 3 anonymous edits

Awk Program Example *Source*: https://en.wikibooks.org/w/index.php?oldid=1488018 *Contributors*: NipplesMeCool, Whiteknight

Awk Invocation and Operation *Source*: https://en.wikibooks.org/w/index.php?oldid=2671509 *Contributors*: NipplesMeCool, Recent Runes, Whiteknight, Xania, 1 anonymous edits

Search Patterns (1) *Source*: https://en.wikibooks.org/w/index.php?oldid=2294382 *Contributors*: NipplesMeCool, Whiteknight, 2 anonymous edits

Search Patterns (2) *Source*: https://en.wikibooks.org/w/index.php?oldid=2053373 *Contributors*: Avicennasis, NipplesMeCool, Whiteknight

Numbers and Strings *Source*: https://en.wikibooks.org/w/index.php?oldid=2294384 *Contributors*: NipplesMeCool, Whiteknight, 1 anonymous edits

Variables *Source*: https://en.wikibooks.org/w/index.php?oldid=2362484 *Contributors*: Miriam e, NipplesMeCool, Vipulksoni, Whiteknight

Arrays *Source*: https://en.wikibooks.org/w/index.php?oldid=2624732 *Contributors*: NipplesMeCool, Urod, Whiteknight

Operations *Source*: https://en.wikibooks.org/w/index.php?oldid=1495506 *Contributors*: NipplesMeCool, Whiteknight

Standard Functions *Source*: https://en.wikibooks.org/w/index.php?oldid=2667132 *Contributors*: Urod, Whiteknight, 3 anonymous edits

Control Structures *Source*: https://en.wikibooks.org/w/index.php?oldid=2610082 *Contributors*: Whiteknight, 1 anonymous edits

Output with print and printf *Source*: https://en.wikibooks.org/w/index.php?oldid=2498649 *Contributors*: Whiteknight, 1 anonymous edits

A Digression: The sprintf Function *Source*: https://en.wikibooks.org/w/index.php?oldid=1273447 *Contributors*: Whiteknight

Output Redirection and Pipes *Source*: https://en.wikibooks.org/w/index.php?oldid=1273448 *Contributors*: Whiteknight

Using Awk from the Command Line *Source*: https://en.wikibooks.org/w/index.php?oldid=1273449 *Contributors*: Whiteknight

Awk Program Files *Source*: https://en.wikibooks.org/w/index.php?oldid=1273450 *Contributors*: Whiteknight

A Note on Awk in Shell Scripts *Source*: https://en.wikibooks.org/w/index.php?oldid=1273451 *Contributors*: Whiteknight

Nawk *Source*: https://en.wikibooks.org/w/index.php?oldid=2611951 *Contributors*: Goldenburg111, Whiteknight, 3 anonymous edits

Awk Quick Reference Guide *Source*: https://en.wikibooks.org/w/index.php?oldid=1273453 *Contributors*: Whiteknight

Revision History *Source*: https://en.wikibooks.org/w/index.php?oldid=1273454 *Contributors*: Whiteknight

Resources *Source*: https://en.wikibooks.org/w/index.php?oldid=1495225 *Contributors*: NipplesMeCool, Whiteknight

Licensing *Source*: https://en.wikibooks.org/w/index.php?oldid=2053371 *Contributors*: Avicennasis, Whiteknight

Image Sources, Licenses and Contributors

File:Heckert GNU white.svg *Source*: https://en.wikibooks.org/w/index.php?title=File:Heckert_GNU_white.svg *License*: Creative Commons Attribution-Sharealike 2.0 *Contributors*: Aurelio A. Heckert <aurium@gmail.com>

License