AWK

One of the great things we can do in the shell is embed other programming languages within the body of our scripts. We have seen hints of this with the stream editor `sed`, and the arbitrary precision calculator program `bc`. By using the shell's single quoting mechanism to isolate text from shell expansion, we can freely express other programming languages, provided we have a suitable language interpreter to execute them.

In this adventure, we are going to look at one such program, `awk`.

History

The AWK programming language is truly one of the classic tools used in Unix. It dates back to the very earliest days of the Unix tradition. It was originally developed in the late 1970's at Bell Telephone Laboratories by Alfred Aho, Peter Weinberger, and Brian Kernighan. The name "AWK" comes from the last names of the three authors. It underwent major improvements in 1985 with the release of `nawk` or "new awk." It is that version that we still use today, though it is usually just called `awk`.

Availability

`awk` is a standard program found in most every Linux distribution. Two free/open source versions of the program are in common use. One is called `mawk` (short for Mike's awk, named for its original author, Mike Brennan) and `gawk` (GNU awk). Both versions fully implement the 1985 `nawk` standard as well as add a variety of extensions. For our purposes, either version is fine, since we will be focusing on the traditional `nawk` features. In most distributions, the name `awk` is symbolically linked to either `mawk` or `gawk`.

So, What's It Good For?

Though AWK is fairly general purpose, it is really designed to create *filters*, that is, programs that accept standard input, transform data, and send it to standard output. In particular, AWK is very good at processing *columnar data*. This makes it a good choice for developing report generators, and tools that are used to re-format data. Since it has strong regular expression support, it's good for very small text extraction and reformatting problems, too. Like `sed`, many AWK programs are just one line long.

In recent years, AWK has fallen out of fashion, being supplanted by other, newer, interpreted languages such as *Perl* and *python*, but AWK still has some advantages:

- It's easy to learn. The language is not overly complex and has a syntax much like the C programming language, so learning it will be useful in the future when we study other languages and tools.
- It really excels at a solving certain types of problems.

How It Works

The structure of an AWK program is somewhat unique among programming languages. Programs consist of a series of one or more *pattern* and *action* pairs. Before we get into that though, let's look at what the typical AWK program does.

We already know that the typical AWK program acts as a filter. It reads data from standard input, and outputs filtered data on standard output. It reads data one *record* at a time. By default, a record is a line of text terminated by a newline character. Each time a record is read, AWK automatically separates the record into *fields*. Fields are, again by default, separated by whitespace. Each field is assigned to a variable, which is given a numeric name. Variable $1 is the first field, $2 is the second field, and so on. $0 signifies the entire record. In addition, a variable named NF is set containing the number of fields detected in the record.

Pattern/action pairs are tests and corresponding actions to be performed on each record. If the pattern is true, then the action is performed. When the list of patterns is exhausted, the AWK program reads the next record and the process is repeated.

Let's try a really simple case. We'll filter the output of an `ls` command:

```
me@linuxbox ~ $ ls -l /usr/bin | awk '{print $0}'
```

The AWK program is contained within the single quotes following the `awk` command. Single quotes are important because we do not want the the shell to attempt any expansion on the AWK program, since its syntax has nothing to do with the shell. For example, `$0` represents the value of the entire record the AWK program read on standard input. In AWK, the `$` means "field" and is not a trigger for parameter expansion as it is in the shell.

Our example program consists of a single action with no pattern present. This is allowed and it means that every record matches the pattern. When we run this command, it simply outputs every line of input much like the `cat` command.

If we look at a typical line of output from `ls -l`, we see that it consists of 9 fields, each separated from its neighbor by one or more whitespace characters:

```
-rwxr-xr-x 1 root root 265 Apr 17 2012 zxpdf
```

Let's add a pattern to our program so it will only print lines with more than 9 fields:

```
me@linuxbox ~ $ ls -l /usr/bin | awk 'NF > 9 {print $0}'
```

We now see a list of symbolic links in `/usr/bin` since those directory listings contain more than 9 fields. This pattern will also match entries with file names containing embedded spaces, since they too will have more than 9 fields.

Special Patterns

Patterns in AWK can have many forms. There are conditional expressions like we have just seen. There are also regular expressions, as we would expect. There are two special patterns called BEGIN and END. The BEGIN pattern carries out its corresponding action before the first record is read. This is useful for initializing variables, or printing headers at the beginning of output. Likewise, the END pattern performs its corresponding action after the last record is read from the input file. This is good for outputting summaries once the input has been processed.

Let's try a more elaborate example. We'll assume for the moment that the directory does not contain any file names with embedded spaces (though this is *never* a safe assumption). We could use the any file names with embedded spaces (though this is *never* a safe assumption). We could use the following script to list symbolic links:

```
#!/bin/bash

# Print a directory report

ls -l /usr/bin | awk '
    BEGIN {
        print "Directory Report"
        print "================"
    }

    NF > 9 {
        print $9, "is a symbolic link to", $NF
    }

    END {
        print "============="
        print "End Of Report"
    }

'
```

In this example, we have 3 pattern/action pairs in our AWK program. The first is a BEGIN pattern and its action that prints the report header. We can spread the action over several lines, though the opening brace "{" of the action must appear on the same line as the pattern.

The second pattern tests the current record to see if it contains more than 9 fields and, if true, the 9th field is printed, followed by some text and the final field in the record. Notice how this was done. The NF variable is preceded by a "$", thus it refers to the NFth field rather than the value of NF itself.

Lastly, we have an END pattern. Its corresponding action prints the "End Of Report" message once all of the lines of input have been read.

Invocation

There are three ways we can run an AWK program. We have already seen how to embed a program in a shell script by enclosing it inside single quotes. The second way is to place the awk script in its own file and call it from the the `awk` program like so:

```
awk -f program_file
```

Lastly, we can use the *shebang* mechanism to make the AWK script a standalone program like a shell script:

```
#!/usr/bin/awk -f

# Print a directory report

BEGIN {
    print "Directory Report"
    print "================"
}

NF > 9 {
    print $9, "is a symbolic link to", $NF
}

END {
    print "============="
    print "End Of Report"
}
```

The Language

Let's take a look at the features and syntax of AWK programs.

Program Format

The formatting rules for AWK programs are pretty simple. Actions consist of one or more statements surrounded by braces ({}) with the starting brace appearing on the same line as the pattern. Blank lines are ignored. Comments begin with a pound sign (#) and may appear at the end of any line. Long statements may be broken into multiple lines using line continuation characters (a backslash followed immediately by a newline). Lists of parameters separated by commas may be broken after any comma. Here is an example:

```
BEGIN { # The action's opening brace must be on same line as the pattern

  # Blank lines are ignored

  # Line continuation characters can be used to break long lines
  print \
    $1, # Parameter lists may be broken by commas
    $2, # Comments can appear at the end of any line
    $3

  # Multiple statements can appear on one line if separated by
  # a semicolon
  print "String 1"; print "String 2"

} # Closing brace for action
```

Patterns

Here are the most common types of patterns used in AWK:

BEGIN and END

As we saw earlier, the BEGIN and END patterns perform actions before the first record is read and after the last record is read, respectively.

relational-expression

Relational expressions are used to test values. For example, we can test for equivalence:

```
$1 == "Fedora"
```

or for relations such as:

```
$3 >= 50
```

It is also possible to perform calculations like:

```
$1 * $2 < 100
```

/regular-expression/

AWK supports extended regular expressions like those supported by `egrep`. Patterns using regular expression can be expressed in two ways. First, we can enclose a regular expression in slashes and a match is attempted on the entire record. If a finer level of control is needed, we can provide an expression containing the string to be matched using the following syntax:

expression ~ /regexp/

For example, if we only wanted to attempt a match on the third field in a record, we could to this:

```
$3 ~ /^[567]/
```

From this, we can think of the "~" as meaning "matches" or "contains", thus we can read the pattern above as "field 3 matches the regular expression `^[567]`".

pattern logical-operator pattern

It is possible to combine patterns together using the logical operators || and &&, meaning OR and AND, respectively. For example, if we want to test a record to see if the first field is a number greater than 100 and the last field is the word "Debit", we can do this:

```
$1 > 100 && $NF == "Debit"
```

! pattern

It is also possible to negate a pattern, so that only records that do not match a specified pattern are selected.

pattern, pattern

Two patterns separated by a comma is called a *range pattern*. With it, once the first pattern is matched, every subsequent record matches until the second pattern is matched. Thus, this type of pattern will select a range of records. Let's imagine that we have a list of records and that the first field in each record contains a sequential record number:

```
0001    field   field   field
0002    field   field   field
0003    field   field   field
```

and so on. And let's say that we want to extract records 0050 through 0100, inclusive. To do so, we could use a range pattern like this:

```
$1 == "0050", $1 == "0100"
```

Fields And Records

The AWK language is so useful because of its ability to automatically separate fields and records. While the default is to separate records by newlines and fields by whitespace, this can be adjusted. The `/etc/passwrd` file, for example, does not separate its fields with whitespace; rather, it uses colons (:). AWK has a built in variable named FS (field separator) that defines the delimiter separating fields in a record. Here is an AWK program that will list the user ID and the user's name from the file:

```
BEGIN { FS = ":" }
{ print $1, $5 }
```

This program has two pattern/action pairs. The first action is performed before the first record is read and sets the input field separator to be the colon character.

and sets the input field separator to be the colon character.

The second pair contains only an action and no pattern. This will match every record. The action prints the first and fifth fields from each record.

The FS variable may contain a regular expression, so really powerful methods can be used to separate fields.

Records are normally separated by newlines, but this can be adjusted too. The built-in variable RS (record separator) defines how records are delimited. A common type of record consists of multiple lines of data separated by one or more blank lines. AWK has a shortcut for specifying the record separator in this case. We just define RS to be an empty string:

```
RS = ""
```

Note that when this is done, newlines, in addition to any other specified characters, will always be treated as field separators regardless of how the FS variable is set. When we process multi-line records, we will often want to treat each line as a separate field, so doing this is often desirable:

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

Variables and Data Types

AWK treats data as either a string or a number, depending on its context. This can sometimes become an issue with numbers. AWK will often treat numbers as strings unless something specifically "numeric" is done with them.

We can force AWK to treat a string of digits as a number by performing some arithmetic on it. This is most easily done by adding zero to the number:

```
n = 105 + 0
```

Likewise, we can get AWK to treat a string of digits as a string by concatenating an empty string:

```
s = 105 ""
```

String concatenation in AWK is performed using a space character as an operator - an unusual feature of the language.

Variables are created as they are encountered (no prior declaration is required), just like the shell. Variable names in AWK follow the same rules as the shell. Names may consist of letters, numbers, and underscore characters. Like the shell, the first character of a variable name must not be a number. Variable names are case sensitive.

Built-in Variables

We have already looked at a few of AWK's built-in variables. Here is a list of the most useful ones:

FS - Field separator

This variable contains a regular expression that is used to separate a record into fields. Its initial value separates fields with whitespace. AWK supports a shortcut to return this variable to its original value:

```
FS = " "
```

The value of FS can also be set using the -F option on the command line. For example, we can quickly extract the user name and UID fields from the `/etc/passwd` file like this:

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

NF - Number of fields

This variable updates each time a record is read. We can easily access the last field in the record by referring to $NF.

NR - Record number

This variable increments each time a record is read, thus it contains the total number of records read from the input stream. Using this variable, we could easily simulate a `wc -l` command with:

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

or number the lines in a file with:

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

OFS - Output field separator

This string is used to separate fields when printing output. The default is a single space. Setting this can be handy when reformatting data. For example, we could easily change a table of values to a CSV (comma separated values) file by setting OFS to equal ",". To demonstrate, here is a program that reads our directory listing and outputs a CSV stream:

```
ls -l | awk 'BEGIN {OFS = ","}
NF == 9 {print $1,$2,$3,$4,$5,$6,$7,$8,$9}'
```

We set the pattern to only match input lines containing 9 fields. This eliminates symbolic links and other weird file names from the data to be processed.

Each line of the resulting output would resemble this:

```
-rwxr-xr-x,1,root,root,100984,Jan,11,2015,a2p
```

If we had omitted setting OFS, the print statement would use the default value (a single space):

```
ls -l | awk 'NF == 9 {print $1,$2,$3,$4,$5,$6,$7,$8,$9}'
```

Which would result in each line of output resembling this:

```
-rwxr-xr-x 1 root root 100984 Jan 11 2015 a2p
```

ORS - Output record separator

This is the string used to separate records when printing output. The default is a newline character. We could use this variable to easily double-space a file by setting ORS to equal two newlines:

```
ls -l | awk 'BEGIN {ORS = "\n\n"} {print}'
```

RS - Record separator

When reading input, AWK interprets this string as the end of record marker. The default value is a newline.

FILENAME

If AWK is reading its input from a file specified on the command line, then this variable contains the name of the file.

FNR - File record number

When reading input from a file specified on the command line, AWK sets this variable to the number of the record read from that file.

Arrays

Single-dimensional arrays are supported in AWK. Data contained in array elements may be either numbers or strings. Array indexes may also be either strings (for *associative arrays*) or numbers.

Assigning values to array elements is done like this:

```
a[1] = 5         # Numeric index
a["five"] = 5    # String index
```

Though AWK only supports single dimension arrays (like bash), it also provides a mechanism to simulate multi-dimensional arrays. When assigning an array index, it is possible to use this form to represent more than one dimension:

```
a[j,k] = "foo"
```

When AWK sees this construct, it builds an index consisting of the strings `j` and `k` separated by the contents of the built-in variable SUBSEP. By default, SUBSEP is set to "\034" (character 34 octal, 28 decimal). This ASCII control code is fairly obscure and thus unlikely to appear in ordinary text, so it's pretty safe for AWK to use.

Note that both `mawk` and `gawk` implement language extensions to support multi-dimensional arrays in a more formal way. Consult their respective documentation for details. If a portable method is needed, use the method above rather than the implementation-specific way.

We can delete arrays and array elements this way:

```
delete a[i]      # delete a single element
delete a         # delete array
```

Arithmetic and Logical Expressions

AWK supports a pretty complete set of arithmetic and logical operators:

Operators

Assignment	`= += -= *= /= %= ^= ++ --`
Relational	`< > <= >= == !=`
Arithmetic	`+ - * / % ^`

Operators

Matching	`~ !~`
Array	`in`
Logical	`|| &&`

These operators behave like those in the shell; however, unlike the shell, which is limited to integer arithmetic, AWK arithmetic is floating point. This makes AWK a good way to do more complex arithmatic than the shell alone.

Arithmetic and logical expressions can be used in both patterns and actions. Here's an example that counts the number of lines containing exactly 9 fields:

```
ls -l /usr/bin | awk 'NF == 9 {count++} END {print count}'
```

This AWK program consists of 2 pattern/action pairs. The first one matches lines where the number of fields is equal to 9. The action creates and increments a variable named `count`. Each time a line with exactly 9 fields is encountered in the input stream, `count` is incremented by 1.

The second pair matches when the end of the input stream is reached and the resulting action prints the final value of `count`.

Using this basic form, let's try something a little more useful; a program that calculates the total size of the files in the list:

```
ls -l /usr/bin | awk 'NF >=9 {total += $5} END {print total}'
```

Here is a slight variation (with shortened variable names to make it a little more concise) that calculates the average size of the files:

```
ls -l /usr/bin | awk 'NF >=9 {c++; t += $5} END {print t / c}'
```

Flow Control

AWK has many of the same flow control statements that we've seen previously in the shell (with the notable exception of case, though we can think of an AWK program as one big case statement inside a loop) but the syntax more closely resembles that of the C programming language. Actions in AWK often contain complex logic consisting of various statements and flow control instructions. A statement in this context can be a simple statement like:

```
a = a + 1
```

Or a compound statement enclosed in braces such as:

```
{a = a + 1; b = b * a}
```

if (*expression*) *statement*
if(*expression*) *statement* else *statement*

The *if/then/else* construct in AWK behaves the way we would expect. AWK evaluates an expression in parenthesis and if the result is non-zero, the statement is carried out. We can see this behavior by executing the following commands:

```
awk 'BEGIN {if (1) print "true"; else print "false"}'
awk 'BEGIN {if (0) print "true"; else print "false"}'
```

Relational expressions such as `(a < b)` will also evaluate to 0 or 1.

In the example below, we construct a primitive report generator that counts the number of lines that have been output and, if the number exceeds the length of a page, a formfeed character is output and the line counter is reset:

```
ls -l /usr/bin | awk '
BEGIN {
    line_count = 0
    page_length = 60
}

{
    line_count++
    if (line_count < page_length)
        print
    else {
        print "\f" $0
        line_count = 0
    }
}
'
```

While the above might be the most obvious way to code this, our knowledge of how evaluations are actually performed, allows us to code this example in a slightly more concise way by using some arithmetic:

```
ls -l /usr/bin | awk '
BEGIN {
    page_length = 60
}

{
    if (NR % page_length)
        print
    else
        print "\f" $0
}
'
```

Here we exploit the fact that the page boundaries will always fall on even multiples of the page length. If `page_length` equals 60 then the page boundaries will fall on lines 60, 120, 180, 240, and so on. All we have to do is calculate the remainder (modulo) on the number of lines processed in the input stream (`NR`) divided by the page length and see if the result is zero, and thus an even multiple.

AWK supports an expression that's useful for testing membership in an array:

```
(var in array)
```

where *var* is an index value and *array* is an array variable. Using this expression tests if the index *var* exists in the specified array. This method of testing for array membership avoids the problem of inadvertently creating the index by testing it with methods such as:

```
if (array[var] != "")
```

When the test is attempted this way, the array element `var` is created, since AWK creates variables simply by their use. When the `(var in array)` form is used, no variable is created.

To test for array membership in a multi-dimensional array, the following syntax is used:

```
((var1,var2) in array)
```

for (*expression* ; *expression* ; *expression*) *statement*

The *for* loop in AWK closely resembles the corresponding one in the C programming language. It is comprised of 3 expressions. The first expression is usually used to initialize a counter variable, the second defines when the loop is completed, and the third defines how the loop is incremented or advanced at each iteration. Here is a demonstration using a for loop to print fields in reverse order:

```
ls -l | awk '{s = ""; for (i = NF; i > 0; i--) s = s $i OFS; print s}'
```

In this example we create an empty string named `s`, then begin a loop that starts with the number of fields in the current input line (`i = NF`) and counts down (`i--`) until we reach the first field (`i > 0`). Each iteration of the loop causes the current field and the output field separator to be concatenated to the string `s` (`s = s $i OFS`). After the loop concludes, we print the resulting value of string `s`.

for (*var* in *array*) *statement*

AWK has a special flow control statement for traversing the indexes of an array. Here is an example of what it does:

```
awk 'BEGIN {for (i=0; i<10; i++) a[i]="foo"; for (i in a) print i}'
```

In this program, we have a single BEGIN pattern/action that performs the entire exercise without the need for an input stream. We first create an array `a` and add 10 elements, each containing the string "foo". Next, we use `for (i in a)` to loop through all the indexes in the array and print each index. It is important to note that the order of the arrays in memory is *implementation dependent*, meaning that it could be anything, so we cannot rely on the results being in any particular order. We'll look at how to address this problem a little later.

Even without sorted order, this type of loop is useful if we need to process every element in an array. For example, we could delete every element of an array like this:

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

while (*expression*) *statement*
do *statement* while (*expression*)

The *while* and *do* loops in AWK are pretty straightforward. We determine a condition that must be maintained for the loop to continue. We can demonstrate this using our field reversal program (we'll type it out in multiple lines to make the logic easier to follow):

```
ls -l | awk '{
    s = ""
    i = NF
    while (i > 0) {
        s = s $i OFS
        i--
    }
    print s
}'
```

The do loop is similar to the while loop; however the do loop will always execute its statement at least once, whereas the while loop will only execute its statement if the initial condition is met.

break
continue

next

The break, continue, and next keywords are used to "escape" from loops. break and continue behave like their corresponding commands in the shell. continue tells AWK to stop and continue with the next iteration of the current loop. break tells AWK to exit the current loop entirely. The next keyword tells AWK to skip the remainder of the current program and begin processing the next record of input.

exit *expression*

As with the shell, we can tell AWK to exit and provide an optional expression that sets AWK's exit status.

Regular Expressions

Regular expressions in AWK work like those in egrep, a topic we covered in chapter 19 of TLCL. It is important to note that back references are not supported and that some versions of AWK (most notably mawk versions prior to 1.3.4) do not support POSIX character classes.

Regular expressions are most often used in patterns, but they are also used in some of the built-in variables such as FS and RS, and they have various roles in the string functions which we will discuss shortly.

Let's try using some simple regular expressions to tally the different file types in our directory listing (we'll make clever use of an associative array too).

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {t["Regular Files"]++}
$1 ~ /^d/ {t["Directories"]++}
$1 ~ /^l/ {t["Symbolic Links"]++}
END {for (i in t) print i ":\t" t[i]}
'
```

In this program, we use regular expressions to identify the first character of the first field and increment the corresponding element in array t. Since we can use strings as array indexes in AWK, we spell out the file type as the index. This makes printing the results in the END action easy, as we only have to traverse the array with for (i in t) to obtain both the name and the accumulated total for each type.

Output Functions

print *expr1, expr2, expr3,...*

As we have seen, print accepts a comma-separated list of arguments. An argument can be any valid expression; however, if an expression contains a relational operator, the entire argument list must be enclosed in parentheses.

The commas are important, because they tell AWK to separate output items with the output field separator (OFS). If omitted, AWK will interpret the members of the argument list as a single expression of string concatenation.

printf(*format, expr1, expr2, expr3,...*)

In AWK, printf is like the corresponding shell built-in (see TLCL chapter 21 for details). It formats its list of arguments based the contents of a *format string*. Here is an example where we output a list of list of arguments based the contents of a *format string*. Here is an example where we output a list of files and their sizes in kilobytes:

```
ls -l /usr/bin | awk '{printf("%-30s%8.2fK\n", $9, $5 / 1024)}'
```

Writing to Files and Pipelines

In addition to sending output to stdout, we can also send output to files and pipelines.

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {print $0 > "regfiles.txt"}
$1 ~ /^d/ {print $0 > "directories.txt"}
$1 ~ /^l/ {print $0 > "symlinks.txt"}
'
```

Here we see a program that writes separate lists of regular files, directories, and symbolic links.

AWK also provides a `>>` operator for appending to files, but since AWK only opens a file once per program execution, the `>` causes AWK to open the file at the beginning of execution and truncate the file to zero length much like we see with the shell. However, once the file is open, it stays open and each subsequent write appends contents to the file. The `>>` operator behaves in the same manner, but when the file is initially opened it is not truncated and all content is appended (i.e., it preserves the contents of an existing file).

AWK also allows output to be sent to pipelines. Consider this program, where we read our directory into an array and then output the entire array:

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {a[$9] = $5}
END {for (i in a)
    {print a[i] "\t" i}
}
'
```

If we run this program, we notice that the array is output in a seemingly random "implementation dependent" order. To correct this, we can pipe the output through `sort`:

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {a[$9] = $5}
END {for (i in a)
    {print a[i] "\t" i | "sort -nr"}
}
'
```

Reading Data

As we have seen, AWK programs most often process data supplied from standard input. However, we can also specify input files on the command line:

```
awk 'program' file...
```

Knowing this, we can, for example, create an AWK program that simulates the `cat` command:

```
awk '{print $0}' file1 file2 file3
```

or `wc`:

```
awk '{chars += length($0); words += NF}
    END {print NR, words, chars + NR}' file1
```

This program has a couple of interesting features. First, it uses the AWK string function `length` to obtain the number of characters in a string. This is one of many string functions that AWK provides, and we will talk more about them in a bit. The second feature is the `chars + NR` expression at the end. This is done because `length($0)` does not count the newline character at the end of each line, so we have to add them to make the character count come out the same as real `wc`.

Even if we don't include filenames on the command line for AWK to input, we can tell AWK to read data from a file specified from within a program. Normally we don't need to do this, but there are some cases where this might be handy. For example, if we wanted to insert one file inside of another, we could use the `getline` function in AWK. Here's an example that adds a header and footer to an existing body text file:

```
awk '
    BEGIN {
        while (getline <"header.txt" > 0) {
            print $0
        }
    }
    {print}
    END {
        while (getline <"footer.txt" > 0) {
            print $0
        }
    }
' < body.txt > finished_file.txt
```

`getline` is quite flexible and can be used in a variety of ways:

getline

In its most basic form, `getline` reads the next record from the current input stream. `$0`, `NF`, `NR`, and `FNR` are set.

getline *var*

Reads the next record from the current input stream and assigns its contents to the variable *var*. *var*, `NR`, and `FNR` are set.

getline <*file*

Reads a record from *file*. `$0` and `NF` are set. It's important to check for errors when reading from files. In the earlier example above, we specified a while loop as follows:

```
while (getline <"header.txt" > 0)
```

As we can see, `getline` is reading from the file `header.txt`, but what does the "> 0" mean? The answer is that, like most functions, `getline` returns a value. A positive value means success, zero means EOF (end of file), and a negative value means some other file-related problem, such as file not found has occurred. If we did not check the return value, we might end up with an infinite loop.

getline *var* <*file*

Reads the next record from *file* and assigns its contents to the variable *var*. Only *var* is set.

command | getline

Reads the next record from the output of *command*. `$0` and `NF` are set. Here is an example where we use AWK to parse the output of the `date` command:

```
awk '
    BEGIN {
        "date" | getline
        print $4
    }
'
```

command | getline *var*

Reads the next record from the output of *command* and assigns its contents to the variable *var*. Only *var* is set.

String Functions

As one would expect, AWK has many functions used to manipulate strings and what's more, many of them support regular expressions. This makes AWK's string handling very powerful.

gsub(*r*, *s*, *t*)

Globally replaces any substring matching regular expression *r* contained within the target string *t* with the string *s*. The target string is optional. If omitted, `$0` is used as the target string. The function returns the number of substitutions made.

index(*s1*, *s2*)

Returns the leftmost position of string *s2* within string *s1*. If *s2* does not appear within *s1*, the function returns 0.

length(*s*)

Returns the number of characters in string *s*.

match(*s*, *r*)

Returns the leftmost position of a substring matching regular expression *r* within string *s*. Returns 0 if no match is found. This function also sets the internal variables `RSTART` and `RLENGTH`.

split(*s*, *a*, *fs*)

Splits string *s* into fields and stores each field in an element of array *a*. Fields are split according to field separator *fs*. For example, if we wanted to break a phone number such as 800-555-1212 into 3 fields separated by the "-" character, we could do this:

```
phone="800-555-1212"
split(phone, fields, "-")
```

After doing so, the array `fields` will contain the following elements:

```
fields[1] = "800"
fields[2] = "555"
```

```
fields[3] = "1212"
```

sprintf(*fmt*, *exprs*)

This function behaves like `printf`, except instead of outputting a formatted string, it returns a formatted string containing the list of expressions to the caller. Use this function to assign a formatted string to a variable:

```
area_code = "800"
exchange = "555"
number = "1212"
phone_number = sprintf("(%s) %s-%s", area_code, exchange, number)
```

sub(*r*, *s*, *t*)

Behaves like `gsub`, except only the first leftmost replacement is made. Like `gsub`, the target string *t* is optional. If omitted, `$0` is used as the target string.

substr(*s*, *p*, *l*)

Returns the substring contained within string *s* starting at position *p* with length *l*.

Arithmetic Functions

AWK has the usual set of arithmetic functions. A word of caution about math in AWK: it has limitations in terms of both number size and precision of floating point operations. This is particularly true of `mawk`. For tasks involving extensive calculation, `gawk` would be preferred. The `gawk` documentation provides a good discussion of the issues involved.

atan2(*y*, *x*)

Returns the arctangent of *y/x* in radians.

cos(*x*)

Returns the cosine of *x*, with *x* in radians.

exp(*x*)

Returns the exponential of *x*, that is e^*x*.

int(*x*)

Returns the integer portion of *x*. For example if *x* = 1.9, 1 is returned.

log(*x*)

Returns the natural logarithm of *x*. *x* must be positive.

rand()

Returns a random floating point value *n* such that 0 <= *n* < 1. This is a value between 0 and 1 where a value of 0 is possible but not 1. In AWK, random numbers always follow the same sequence of values unless the seed for the random number generator is first set using the `srand()` function (see below).

sin(*x*)

Returns the sine of *x*, with *x* in radians.

sqrt(*x*)

Returns the square root of *x*.

srand(*x*)

Sets the seed for the random number generator to *x*. If *x* is omitted, then the time of day is used as the seed. To generate a random integer in the range of 1 to *n*, we can use code like this:

```
srand()
# Generate a random integer between 1 and 6 inclusive
dice_roll = int(6 * rand()) + 1
```

User Defined Functions

In addition to the built-in string and arithmetic functions, AWK supports user-defined functions much like the shell. The mechanism for passing parameters is different, and more like traditional languages such as C.

Defining a function

A typical function definition looks like this:

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

We use the keyword `function` followed by the name of the function to be defined. The name must be immediately followed by the opening left parenthesis of the parameter list. The parameter list may contain zero or more comma-separated parameters. A brace delimited code block follows with one or more statements. To specify what is returned by the function, the `return` statement is used, followed by an expression containing the value to be returned. If we were to convert our previous dice rolling example into a function, it would look like this:

```
function dice_roll() {
    return int(6 * rand()) + 1
}
```

Further, if we wanted to generalize our function to support different possible maximum values, we could code this:

```
function rand_integer(max) {
    return int(max * rand()) + 1
}
```

and then change `dice_roll` to make use of our generalized function:

```
function dice_roll() {
    return rand_integer(6)
}
```

Passing parameters to functions

As we saw in the example above, we pass parameters to the function, and they are operated upon within the body of the function. Parameters fall into two general classes. First, there are the *scalar variables*, such as strings and numbers. Second are the arrays. This distinction is important in AWK because of the way that parameters are passed to functions. Scalar variables are *passed by value*, meaning that a copy of the variable is created and given to the function. This means that scalar variables act as local variables within the function and are destroyed once the function exits. Array variables, on the other hand, are *passed by reference* meaning that a pointer to the array's starting position in memory is passed to the function. This means that the array is not treated as a local variable and that any change made to the array persists once the program exits the function. This concept of passed by value versus passed by reference shows up in a lot of programming languages so it's important to understand it.

Local variables

One interesting limitation of AWK is that we cannot declare local variables within the body of a function. There is a workaround for this problem. We can add variables to the parameter list. Since all scalar variables in the parameter list are passed by value, they will be treated as if they are local variables. This does not apply to arrays, since they are always passed by reference. Unlike many other languages, AWK does not enforce the parameter list, thus we can add parameters that are not used by the caller of the function. In most other languages, the number and type of parameters passed during a function call must match the parameter list specified by the function's declaration.

By convention, additional parameters used as local variables in the function are preceded by additional spaces in the parameter list like so:

```
function my_funct(param1, param2, param3,    local1, local2)
```

These additional spaces have no meaning to the language, they are there for the benefit of the human reading the code.

Let's try some short AWK programs on some numbers. First we need some data. Here's a little AWK program that produces a table of random integers:

```
# random_table.awk - generate table of random numbers

function rand_integer(max) {
    return int(max * rand()) + 1
}

BEGIN {
    srand()
    for (i = 0; i < 100; i++) {
        for (j = 0; j < 5; j++) {
            printf("    %5d", rand_integer(99999))
        }
        printf("\n", "")
    }
}
```

If we store this in a file, we can run it like so:

```
me@linuxbox ~ $ awk -f random_table.awk > random_table.dat
```

And it should produce a file containing 100 rows of 5 columns of random integers.

Convert file into CSV format

One of AWK's many strengths is file format conversion. Here we will convert our neatly arranged columns of numbers into a CSV (comma separated values) file.

```
awk 'BEGIN {OFS=","} {print $1,$2,$3,$4,$5}' random_table.dat
```

This is a very easy conversion. All we need to do is change the output field separator (`OFS`) and then print all of the individual fields. While it is very easy to write a CSV file, reading one can be tricky. In some cases, applications that write CSV files (including many popular spreadsheet programs) will create lines like this:

```
word1, "word2a, word2b", word3
```

Notice the embedded comma in the second field. This throws the simple AWK solution (`FS=","`) out the window. Parsing this kind of file can be done (`gawk`, in fact has a language extension for this problem), but it's not pretty. It is best to avoid trying to read this type of file.

Convert file into TSV format

A frequently available alternative to the CSV file is the TSV (tab separated value) file. This file format uses tab characters as the field separators:

```
awk 'BEGIN {OFS="\t"} {print $1,$2,$3,$4,$5}' random_table.dat
```

Again, writing these files is easy to do. We just set the output field separator to a tab character. In regards to reading, most applications that write CSV files can also write TSV files. Using TSV files avoids the embedded comma problem we often see when attempting to read CSV files.

Print the total for each row

If all we need to do is some simple addition, this is easily done:

```
awk '
    {
        t = $1 + $2 + $3 + $4 + $5
        printf("%s = %6d\n", $0, t)
    }
' random_table.dat
```

Print the total for each column

Adding up the column is pretty easy, too. In this example, we use a loop and array to maintain running totals for each of the five columns in our data file:

```
awk '
    {
        for (i = 1; i <= 5; i++) {
            t[i] += $i
        }

        print
    }
    END {
        print "  ==="
        for (i = 1; i <= 5; i++) {
            printf("  %7d", t[i])
        }
        printf("\n", "")
     }
' random_table.dat
```

Print the minimum and maximum value in column 1

```
awk '
    BEGIN {min = 99999}
    $1 > max {max = $1}
    $1 < min {min = $1}
    END {print "Max =", max, "Min =", min}
' random_table.dat
```

One Last Example

For our last example, we'll create a program that processes a list of pathnames and extracts the extension from each file name to keep a tally of how many files have that extension:

```
# file_types.awk - sorted list of file name extensions and counts

BEGIN {FS = "."}

{types[$NF]++}

END {
    for (i in types) {
        printf("%6d %s\n", types[i], i) | "sort -nr"
    }
}
```

To find the 10 most popular file extensions in our home directory, we can use the program like this:

```
find ~ -name "*.*" | awk -f file_types.awk | head
```

Summing Up

We really have to admire what an elegant and useful tool the authors of AWK created during the early days of Unix. So useful that its utility continues to this day. We have given AWK a brief examination in this adventure. Feel free to explore further by delving deeper into the documentation of the various AWK implementations. Also, searching the web for "AWK one-liners" will reveal many useful and clever tricks possible with AWK.

Further Reading

- The `nawk` man page provides a good reference for the baseline version of AWK. An online version is available at http://linux.die.net/man/1/nawk
- Another `nawk` manual derived from the original `gawk` manual (see below) can be found at http://www.staff.science.uu.nl/~oostr102/docs/nawk/nawk_toc.html
- Many useful AWK programs are just one line long. Eric Pement has compiled an extensive list: http://www.pement.org/awk/awk1line.txt

- In addition to its man page, `gawk` has its own book titled Gawk: Effective AWK Programming available at: `https://www.gnu.org/software/gawk/manual/`
- Peteris Krumins has a nice blog post listing a variety of helpful tips for AWK users: `http://www.catonmat.net/blog/ten-awk-tips-tricks-and-pitfalls/`