A Self-Tutorial

on

AWK

COMPILED BY:

AMIR SALAREE

DEPT. EARTH AND PLANETARY SCIENCES
NORTHWESTERN UNIVERSITY
FALL 2011

Table of Contents

Why learn AWK?

In the past I have covered *grep* and *sed*. This section discusses AWK, another cornerstone of UNIX shell programming. There are three variations of AWK:

> AWK - the original from AT&T
> NAWK - A newer, improved version from AT&T
> GAWK - The Free Software foundation's version

Originally, I didn't plan to discuss NAWK, but several UNIX vendors have replaced AWK with NAWK, and there are several incompatibilities between the two. It would be cruel of me to not warn you about the differences. So I will highlight those when I come to them. It is important to know than all of AWK's features are in NAWK and GAWK. Most, if not all, of NAWK's features are in GAWK. NAWK ships as part of Solaris. GAWK does not. However, many sites on the Internet have the sources freely available. If you user Linux, you have GAWK. But in general, assume that I am talking about the classic AWK unless otherwise noted.

Why is AWK so important? It is an excellent filter and report writer. Many UNIX utilities generates rows and columns of information. AWK is an excellent tool for processing these rows and columns, and is easier to use AWK than most conventional programming languages. It can be considered to be a pseudo-C interpretor, as it understands the same arithmatic operators as C. AWK also has string manipulation functions, so it can search for particular strings and modify the output. AWK also has associative arrays, which are incredible useful, and is a feature most computing languages lack. Associative arrays can make a complex problem a trivial exercise.

I won't exhaustively cover AWK. That is, I will cover the essential parts, and avoid the many variants of AWK. It might be too confusing to discuss three different versions of AWK. I won't cover the GNU version of AWK called "gawk." Similarly, I will not discuss the new AT&T AWK called "nawk." The new AWK comes on the Sun system, and you may find it superior to the old AWK in many ways. In particular, it has better diagnostics, and won't print out the infamous "bailing out near line ..." message the original AWK is prone to do. Instead, "nawk" prints out the line it didn't understand, and highlights the bad parts with arrows. GAWK does this as well, and this really helps a lot. If you find yourself needing a feature that is very difficult or impossible to do in AWK, I suggest you either use NAWK, or GAWK, or convert your AWK script into PERL using the "a2p" conversion program which comes with PERL. PERL is a marvelous language, and I use it all the time, but I do not plan to cover PERL in these tutorials. Having made my intention clear, I can continue with a clear conscience.

Many UNIX utilities have strange names. AWK is one of those utilities. It is not an abbreviation for *awk*ward. In fact, it is an elegant and simple language. The work "AWK" is derived from the initials of the language's three developers: A. Aho, B. W. Kernighan and P. Weinberger.

Basic Structure

The essential organization of an AWK program follows the form:

pattern { action }

The pattern specifies when the action is performed. Like most UNIX utilities, AWK is line oriented. That is, the pattern specifies a test that is performed with each line read as input. If the condition is true, then the action is taken. The default pattern is something that matches every line. This is the blank or null pattern. Two other important patterns are specified by the keywords "BEGIN" and "END." As you might expect, these two words specify actions to be taken before any lines are read, and after the last line is read. The AWK program below:

```
BEGIN { print "START" }
      { print         }
END   { print "STOP"  }
```

adds one line before and one line after the input file. This isn't very useful, but with a simple change, we can make this into a typical AWK program:

```
BEGIN { print "File\tOwner"," }
{ print $8, "\t", $3}
END { print " - DONE -" }
```

I'll improve the script in the next sections, but we'll call it "FileOwner." But let's not put it into a script or file yet. I will cover that part in a bit. Hang on and follow with me so you get the flavor of AWK.

The characters "\t" Indicates a tab character so the output lines up on even boundries. The "$8" and "$3" have a meaning similar to a shell script. Instead of the eighth and third argument, they mean the eighth and third field of the input line. You can think of a field as a column, and the action you specify operates on each line or row read in.

There are two differences between AWK and a shell processing the characters within double quotes. AWK understands special characters follow the "\" character like "t". The Bourne and C UNIX shells do not. Also, unlike the shell (and PERL) AWK does not evaluate variables within strings. To explain, the second line could not be written like this:

```
{print "$8\t$3" }
```

That example would print "$8 $3." Inside the quotes, the dollar sign is not a special character. Outside, it corresponds to a field. What do I mean by the third and eight field? Consider the Solaris "/usr/bin/ls -l" command, which has eight columns of information. The System V version (Similar to the Linux version), "/usr/5bin/ls -l," has 9 columns. The third column is the owner, and the eighth (or nineth) column in the name of the file. This AWK program can be used to process the output of the "ls -l" command, printing out the filename, then the owner, for each file. I'll show you how.

Update: On a linux system, change "$8" to "$9".

One more point about the use of a dollar sign. In scripting languages like Perl and the various shells, a dollar sign means the word following is the name of the variable. Awk is different. The dollar sign means that we are refering to a field or column in the current line. When switching between Perl and AWK you must remener that "$" has a different meaning. So the following piece of code prints two "fields" to standard out. The first field printed is the number "5", the second is the fifth field (or column) on the input line.

```
BEGIN { x=5 }
{ print x, $x}
```

Executing an AWK script

So let's start writing our first AWK script. There are a couple of ways to do this.

Assuming the first script is called "FileOwner," the invocation would be

```
ls -l | FileOwner
```

This might generate the following if there were only two files in the current directory:

```
File Owner


a.file barnett
another.file barnett
- DONE -
```

There are two problems with this script. Both problems are easy to fix, but I'll hold off on this until I cover the basics.

The script itself can be written in many ways. The C shell version would look like this:

```
#!/bin/csh -f
# Linux users have to change $8 to $9
awk '
BEGIN   { print "File\tOwner" } \
                { print $8, "\t", $3}   \
END     { print " - DONE -" } \
'
```

As you can see in the above script, each line of the AWK script must have a backslash if it is not the last line of the script. This is necessary as the C shell doesn't, by default, allow strings I have a long list of complaints about using the C shell.

The Bourne shell (as does most shells) allows quoted strings to span several lines:

```
#!/bin/sh
# Linux users have to change $8 to $9
awk '
BEGIN { print "File\tOwner" }
{ print $8, "\t", $3}
END { print " - DONE -" }
'
```

The third form is to store the commands in a file, and execute

```
awk -f filename
```

Since AWK is also an interpretor, you can save yourself a step and make the file executable by add one line in the beginning of the file:

```
#!/bin/awk -f
BEGIN { print "File\tOwner" }
{ print $8, "\t", $3}
END { print " - DONE -" }
```

Change the permission with the *chmod* command, (i.e. "chmod +x awk_example1.awk"), and the script becomes a new command. Notice the "-f" option following '#!/bin/awk "above, which is also used in the third format where you use AWK to execute the file directly, i.e. "awk -f filename". The "-f" option specifies the AWK file containing the instructions. As you can see, AWK considers lines that start with a "#" to be a comment, just like the shell. To be precise, anything from the "#" to the end of the line is a comment (unless its inside an AWK string. However, I always comment my AWK scripts with the "#" at the start of the line, for reasons I'll discuss later.

Which format should you use? I prefer the last format when possible. It's shorter and simpler. It's also easier to debug problems. If you need to use a shell, and want to avoid using too many files, you can combine them as we did in the first and second example.

Which shell to use with AWK?

The format of AWK is not free-form. You cannot put new line breaks just anywhere. They must go in particular locations. To be precise, in the original AWK you can insert a new line character after the curly braces, and at the end of a command, but not elsewhere. If you wanted to break a long line into two lines at any other place, you had to use a backslash:

```
#!/bin/awk -f
BEGIN { print "File\tOwner" }
{ print $8, "\t", \
$3}
END { print " - DONE -" }
```

The Bourne shell version would be

```
#!/bin/sh
awk '
BEGIN { print "File\tOwner" }
{ print $8, "\t", \
$3}
END { print "done"}
'
```

while the C shell would be

```
#!/bin/csh -f
awk '
BEGIN { print "File\tOwner" }\
{ print $8, "\t", \\
$3}\
END { print "done"}\
'
```

As you can see, this demonstrates how awkward the C shell is when enclosing an AWK script. Not only are back slashes needed for every line, some lines need two. (Note - this is true when using old awk (e.g. on Solaris) because the print statement had to be on one line. Newer AWK's are more flexible where newlines can be added.) Many people will warn you about the C shell. Some of the problems are subtle, and you may never see them. Try to include an AWK or *sed* script within a C shell script, and the back slashes will drive you crazy. This is what convinced me to learn the Bourne shell years ago, when I was starting out. I strongly recommend you use the Bourne shell for any AWK or *sed* script. If you don't use the Bourne shell, then you should learn it. As a minimum, learn how to set variables, which by some strange coincidence is the subject of the next section.

Dynamic Variables

Since you can make a script an AWK executable by mentioning "#!/bin/awk -f" on the first line, including an AWK script inside a shell script isn't needed unless you want to either eliminate the need for an extra file, or if you want to pass a variable to the insides of an AWK script. Since this is a common problem, now is as good a time to explain the technique. I'll do this by showing a simple AWK program that will only print one column. **NOTE: there will be a bug in the first version.** The number of the column will be specified by the first argument. The first version of the program, which we will call "Column," looks like this:

```
#!/bin/sh
#NOTE - this script does not work!
column=$1
awk '{print $column}'
```

A suggested use is:

```
	ls -l | Column 3
```

This would print the third column from the *ls* command, which would be the owner of the file. You can change this into a utility that counts how many files are owned by each user by adding

```
ls -l | Column 3 | uniq -c | sort -nr
```

Only one problem: the script doesn't work. The value of the "column" variable is not seen by AWK. Change "awk" to "echo" to check. You need to turn off the quoting when the variable is seen. This can be done by ending the quoting, and restarting it after the variable:

```
#!/bin/sh
column=$1
awk '{print $'$column'}'
```

This is a very important concept, and throws experienced programmers a curve ball. In many computer languages, a string has a start quote, and end quote, and the contents in between. If you want to include a special character inside the quote, you must prevent the character from having the typical meaning. In the C language, this is down by putting a backslash before the character. In other languages, there is a special combination of characters to to this. In the C and Bourne shell, the quote is just a switch. It turns the interpretation mode on or off. There is really no such concept as "start of string" and "end of string." The quotes toggle a switch inside the interpretor. The quote character is not passed on to the application. This is why there are two pairs of quotes above. Notice there are two dollar signs. The first one is quoted, and is seen by AWK. The second one is not quoted, so the shell evaluates the variable, and replaces "$column" by the value. If you don't understand, either change "awk" to "echo," or change the first line to read "#!/bin/sh -x."

Some improvements are needed, however. The Bourne shell has a mechanism to provide a value for a variable if the value isn't set, or is set and the value is an empty string. This is done by using the format:

${*variable*:-*defaultvalue*}

This is shown below, where the default column will be one:

```
#!/bin/sh
column=${1:-1}
awk '{print $'$column'}'
```

We can save a line by combining these two steps:

```
#!/bin/sh
awk '{print $'${1:-1}'}'
```

It is hard to read, but it is compact. There is one other method that can be used. If you execute an AWK command and include on the command line

variable=*value*

this variable will be set when the AWK script starts. An example of this use would be:

```
#!/bin/sh
awk '{print $c}' c=${1:-1}
```

This last variation does not have the problems with quoting the previous example had. You should master the earlier example, however, because you can use it with any script or command. The second method is special to AWK. Modern AWK's have other options as well.

The Essential Syntax of AWK

Earlier I discussed ways to start an AWK script. This section will discuss the various grammatical elements of AWK.

Arithmetic Expressions

There are several arithmetic operators, similar to C. These are the binary operators, which operate on two variables:

AWK Table 1
Binary Operators

Operator	Type	Meaning
+	Arithmetic	Addition
-	Arithmetic	Subtraction
*	Arithmetic	Multiplication
/	Arithmetic	Division
%	Arithmetic	Modulo
<space>	String	Concatenation

Using variables with the value of "7" and "3," AWK returns the following results for each operator when using the print command:

Expression	Result
7+3	10
7-3	4
7*3	21

```
|7/3          2.33333 |
|7%3          1       |
|7 3          73      |
+---------------------+
```

There are a few points to make. The modulus operator finds the remainder after an integer divide. The *print* command output a floating point number on the divide, but an integer for the rest. The string concatenate operator is confusing, since it isn't even visible. Place a space between two variables and the strings are concatenated together. This also shows that numbers are converted automatically into strings when needed. Unlike C, AWK doesn't have "types" of variables. There is one type only, and it can be a string or number. The conversion rules are simple. A number can easily be converted into a string. When a string is converted into a number, AWK will do so. The string "123" will be converted into the number 123. However, the string "123X" will be converted into the number 0. (NAWK will behave differently, and converts the string into integer 123, which is found in the beginning of the string).

Unary arithmetic operators

The "+" and "-" operators can be used before variables and numbers. If X equals 4, then the statement:

```
print -x;
```

will print "-4."

The Autoincrement and Autodecrement Operators

AWK also supports the "++" and "--" operators of C. Both increment or decrement the variables by one. The operator can only be used with a single variable, and can be before or after the variable. The prefix form modifies the value, and then uses the result, while the postfix form gets the results of the variable, and afterwards modifies the variable. As an example, if X has the value of 3, then the AWK statement

```
print x++, " ", ++x;
```

would print the numbers 3 and 5. These operators are also assignment operators, and can be used by themselves on a line:

```
x++;
--y;
```

Assignment Operators

Variables can be assigned new values with the assignment operators. You know about "++" and "--." The other assignment statement is simply:

variable = arithmetic_expression

Certain operators have precedence over others; parenthesis can be used to control grouping. The statement

```
x=1+2*3 4;
```

is the same as

```
x = (1 + (2 * 3)) "4";
```

Both print out "74."

Notice spaces can be added for readability. AWK, like C, has special assignment operators, which combine a calculation with an assignment. Instead of saying

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

you can more concisely say:

```
x+=2;
```

The complete list follows:

AWK Table 2
Assignment Operators

Operator	Meaning
+=	Add result to variable
-=	Subtract result from variable
*=	Multiply variable by result
/=	Divide variable by result
%=	Apply modulo to variable

Conditional expressions

The second type of expression in AWK is the conditional expression. This is used for certain tests, like the *if* or *while*. Boolean conditions evaluate to true or false. In AWK, there is a definite difference between a boolean condition, and an arithmetic expression. You cannot convert a boolean condition to an integer or string. You can, however, use an arithmetic expression as a conditional expression. A value of 0 is false, while anything else is true. Undefined variables has the value of 0. Unlike AWK, NAWK lets you use booleans as integers.

Arithmetic values can also be converted into boolean conditions by using relational operators:

AWK Table 3
Relational Operators

Operator	Meaning
==	Is equal
!=	Is not equal to
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to

These operators are the same as the C operators. They can be used to compare numbers or strings. With respect to strings, lower case letters are greater than upper case letters.

Regular Expressions

Two operators are used to compare strings to regular expressions:

AWK Table 4
Regular Expression Operators

Operator	Meaning
~	Matches
!~	Doesn't match

The order in this case is particular. The regular expression must be enclosed by slashes, and comes after the operator. AWK supports extended regular expressions, so the following are examples of valid tests:

```
word !~ /START/
lawrence_welk ~ /(one|two|three)/
```

And/Or/Not

There are two boolean operators that can be used with conditional expressions. That is, you can combine two conditional expressions with the "or" or "and" operators: "&&" and "||." There is also the unary not operator: "!."

Commands

There are only a few commands in AWK. The list and syntax follows:

if (*conditional*) *statement* [else *statement*]
while (*conditional*) *statement*
for (*expression* ; *conditional* ; *expression*) *statement*
for (*variable* in *array*) *statement*
break
continue
{ [*statement*] ...}
variable=expression
print [*expression-list*] [> *expression*]
printf *format* [, *expression-list*] [> *expression*]
next
exit

At this point, you can use AWK as a language for simple calculations; If you wanted to calculate something, and not read any lines for input, you could use the *BEGIN* keyword discussed earlier, combined with a *exit* command:

```
#!/bin/awk -f
BEGIN {
```

```
# Print the squares from 1 to 10 the first way

        i=1;
        while (i <= 10) {
                printf "The square of ", i, " is ", i*i;
                i = i+1;
        }

# do it again, using more concise code

        for (i=1; i <= 10; i++) {
                printf "The square of ", i, " is ", i*i;
        }

# now end
exit;
}
```

The following asks for a number, and then squares it:

```
#!/bin/awk -f
BEGIN {
    print "type a number";
}
{
    print "The square of ", $1, " is ", $1*$1;
    print "type another number";
}
END {
    print "Done"
}
```

The above isn't a good filter, because it asks for input each time. If you pipe the output of another program into it, you would generate a lot of meaningless prompts.

Here is a filter that you should find useful. It counts lines, totals up the numbers in the first column, and calculates the average. Pipe "wc -c *" into it, and it will count files, and tell you the average number of words per file, as well as the total words and the number of files.

```
#!/bin/awk -f
BEGIN {
# How many lines
    lines=0;
    total=0;
}
{
# this code is executed once for each line
# increase the number of files
    lines++;
# increase the total size, which is field #1
    total+=$1;
}
END {
# end, now output the total
    print lines " lines read";
    print "total is ", total;
    if (lines > 0 ) {
        print "average is ", total/lines;
    } else {
        print "average is 0";
    }
}
```

You can pipe the output of "ls -s" into this filter to count the number of files, the total size, and the average size. There is a slight problem with this script, as it includes the output of "ls" that reports the total. This causes the number of files to be off by one. Changing

```
lines++;
```

to

```
if ($1 != "total" ) lines++;
```

will fix this problem. Note the code which prevents a divide by zero. This is common in well-written scripts. I also initialize the variables to zero. This is not necessary, but it is a good habit.

AWK Built-in Variables

I have mentioned two kinds of variables: positional and user defined. A user defined variable is one you create. A positional variable is not a special variable, but a function triggered by the dollar sign. Therefore

```
print $1;
```

and

```
X=1;
print $X;
```

do the same thing: print the first field on the line. There are two more points about positional variables that are very useful. The variable "$0" refers to the entire line that AWK reads in. That is, if you had eight fields in a line,

```
print $0;
```

is similar to

```
print $1, $2, $3, $4, $5, $6, $7, $8
```

This will change the spacing between the fields; otherwise, they behave the same. You can modify positional variables. The following commands

```
$2="";
print;
```

deletes the second field. If you had four fields, and wanted to print out the second and fourth field, there are two ways. This is the first:

```
#!/bin/awk -f
{
        $1="";
        $3="";
        print;
}
```

and the second

```
#!/bin/awk -f
{
        print $2, $4;
}
```

These perform similarly, but not identically. The number of spaces between the values vary. There are two reasons for this. The actual number of fields does not change. Setting a positional variable to an empty string does not delete the variable. It's still there, but the contents has been deleted. The other reason is the way AWK outputs the entire line. There is a field separator that specifies what character to put between the fields on output. The first example outputs four fields, while the second outputs two. In-between each field is a space. This is easier to explain if the characters between fields could be modified to be made more visible. Well, it can. AWK provides special variables for just that purpose.

FS - The Input Field Separator Variable

AWK can be used to parse many system administration files. However, many of these files do not have whitespace as a separator. as an example, the password file uses colons. You can easily change the field separator character to be a colon using the "-F" command line option. The following command will print out accounts that don't have passwords:

```
awk -F: '{if ($2 == "") print $1 ": no password!"}' </etc/passwd
```

There is a way to do this without the command line option. The variable "FS" can be set like any variable, and has the same function as the "-F" command line option. The following is a script that has the same function as the one above.

```
#!/bin/awk -f
BEGIN {
        FS=":";
}
{
        if ( $2 == "" ) {
                print $1 ": no password!";
        }
}
```

The second form can be used to create a UNIX utility, which I will name "chkpasswd," and executed like this:

```
chkpasswd </etc/passwd
```

The command "chkpasswd -F:" cannot be used, because AWK will never see this argument. All interpreter scripts accept one and only one argument, which is immediately after the "#!/bin/awk" string. In this case, the single argument is "-f." Another difference between the command line option and the internal variable is the ability to set the input field separator to be more than one character. If you specify

```
FS=": ";
```

then AWK will split a line into fields wherever it sees those two characters, in that exact order. You cannot do this on the command line.

There is a third advantage the internal variable has over the command line option: you can change the field separator character as many times as you want while reading a file. Well, at most once for each line. You can even change it depending on the line you read. Suppose you had the following file which contains the numbers 1 through 7 in three different formats. Lines 4 through 6 have colon separated fields, while the others separated by spaces.

```
ONE 1 I
TWO 2 II
#START
THREE:3:III
FOUR:4:IV
FIVE:5:V
#STOP
SIX 6 VI
SEVEN 7 VII
```

The AWK program can easily switch between these formats:

```
#!/bin/awk -f
{
        if ($1 == "#START") {
                FS=":";
        } else if ($1 == "#STOP") {
                FS=" ";
        } else {
                #print the Roman number in column 3
                print $3
        }
}
```

Note the field separator variable retains its value until it is explicitly changed. You don't have to reset it for each line. Sounds simple, right? However, I have a trick question for you. What happens if you change the field separator while reading a line? That is, suppose you had the following line

    One Two:Three:4 Five

and you executed the following script:

```
#!/bin/awk -f
{
        print $2
        FS=":"
        print $2
}
```

What would be printed? "Three" or "Two:Three:4?" Well, the script would print out "Two:Three:4" twice. However, if you deleted the first print statement, it would print out "Three" once! I thought this was very strange at first, but after pulling out some hair, kicking the deck, and yelling at muself and everyone who had anything to do with the development of UNIX, it is intuitively obvious. You just have to be thinking like a professional programmer to realize it is intuitive. I shall explain, and prevent you from causing yourself physical harm.

If you change the field separator **before** you read the line, the change **affects** what you read. If you change it **after** you read the line, it will **not** redefine the variables. You wouldn't want a variable to change on you as a side-effect of another action. A programming language with hidden side effects is broken, and should not be trusted. AWK allows you to redefine the field separator either before or after you read the line, and does the right thing each time. Once you read the variable, the variable will not change unless you change it. Bravo!

To illustrate this further, here is another version of the previous code that changes the field separator dynamically. In this case, AWK does it by examining field "$0," which is the entire line. When the line contains a colon, the field separator is a colon, otherwise, it is a space:

```
#!/bin/awk -f
{
        if ( $0 ~ /:/ ) {
                FS=":";
        } else {
                FS=" ";
        }
        #print the third field, whatever format
        print $3
}
```

This example eliminates the need to have the special "#START" and "#STOP" lines in the input.

OFS - The Output Field Separator Variable

There is an important difference between

    print $2 $3

and

    print $2, $3

The first example prints out one field, and the second prints out two fields. In the first case, the two positional parameters are concatenated together and output without a space. In the second case, AWK prints two fields, and places the output field separator between them. Normally this is a space, but you can change this by modifying the variable "OFS."

If you wanted to copy the password file, but delete the encrypted password, you could use AWK:

```
#!/bin/awk -f
BEGIN {
        FS=":";
        OFS=":";
}
{
        $2="";
        print
}
```

Give this script the password file, and it will delete the password, but leave everything else the same. You can make the output field separator any number of characters. You are not limited to a single character.

NF - The Number of Fields Variable

It is useful to know how many fields are on a line. You may want to have your script change its operation based on the number of fields. As an example, the command "ls -l" may generate eight or nine fields, depending on which version you are executing. The System V version, "/usr/bin/ls -l" generates nine fields, which is equivalent to the Berkeley "/usr/ucb/ls -lg" command. If you wanted to print the owner and filename then the following AWK script would work with either version of "ls:"

```
#!/bin/awk -f
# parse the output of "ls -l"
# print owner and filename
# remember - Berkeley ls -l has 8 fields, System V has 9
{
        if (NF == 8) {
                print $3, $8;
        } else if (NF == 9) {
                print $3, $9;
        }
}
```

Don't forget the variable can be prepended with a "$." This allows you to print the last field of any column

```
#!/bin/awk -f
{ print $NF; }
```

One warning about AWK. There is a limit of 99 fields in a single line. PERL does not have any such limitations.

NR - The Number of Records Variable

Another useful variable is "NR." This tells you the number of records, or the line number. You can use AWK to only examine certain lines. This example prints lines after the first 100 lines, and puts a line number before each line after 100:

```
#!/bin/awk -f
{ if (NR >= 100) {
        print NR, $0;
}
```

RS - The Record Separator Variable

Normally, AWK reads one line at a time, and breaks up the line into fields. You can set the "RS" variable to change AWK's definition of a "line." If you set it to an empty string, then AWK will read the entire file into memory. You can combine this with changing the "FS" variable. This example treats each line as a field, and prints out the second and third line:

```
#!/bin/awk -f
BEGIN {
# change the record separator from newline to nothing
        RS=""
# change the field separator from whitespace to newline
        FS="\n"
}
{
# print the second and third line of the file
        print $2, $3;
}
```

The two lines are printed with a space between. Also this will only work if the input file is less than 100 lines, therefore this technique is limited. You can use it to break words up, one word per line, using this:

```
#!/bin/awk -f
BEGIN {
        RS=" ";
}
{
        print ;
}
```

but this only works if all of the words are separated by a space. If there is a tab or punctuation inside, it would not.

ORS - The Output Record Separator Variable

The default output record separator is a newline, like the input. This can be set to be a newline and carriage return, if you need to generate a text file for a non-UNIX system.

```
#!/bin/awk -f
# this filter adds a carriage return to all lines
# before the newline character
BEGIN {
	ORS="\r\n"
}
{ print }
```

FILENAME - The Current Filename Variable

The last variable known to regular AWK is "FILENAME," which tells you the name of the file being read.

```
#!/bin/awk -f
# reports which file is being read
BEGIN {
	f="";
}
{	if (f != FILENAME) {
		print "reading", FILENAME;
		f=FILENAME;
	}
	print;
}
```

This can be used if several files need to be parsed by AWK. Normally you use standard input to provide AWK with information. You can also specify the filenames on the command line. If the above script was called "testfilter," and if you executed it with

```
testfilter file1 file2 file3
```

It would print out the filename before each change. An alternate way to specify this on the command line is

```
testfilter file1 - file3 <file2
```

In this case, the second file will be called "-," which is the conventional name for standard input. I have used this when I want to put some information before and after a filter operation. The prefix and postfix files special data before and after the real data. By checking the filename, you can parse the information differently. This is also useful to report syntax errors in particular files:

```
#!/bin/awk -f
{
    if (NF == 6) {
        # do the right thing
    } else {
        if (FILENAME == "-" ) {
            print "SYNTAX ERROR, Wrong number of fields,",
            "in STDIN, line #:", NR,  "line: ", $0;
        } else {
            print "SYNTAX ERROR, Wrong number of fields,",
            "Filename: ", FILENAME, "line # ", NR,"line: ", $0;
        }
    }
}
```

Associative Arrays

I have used dozens of different programming languages over the last 20 years, and AWK is the first language I found that has associative arrays. This term may be meaningless to you, but believe me, these arrays are invaluable, and simplify programming enormously. Let me describe a problem, and show you how associative arrays can be used for reduce coding time, giving you more time to explore another stupid problem you don't want to deal with in the first place.

Let's suppose you have a directory overflowing with files, and you want to find out how many files are owned by each user, and perhaps how much disk space each user owns. You really want someone to blame; it's hard to tell who owns what file. A filter that processes the output of *ls* would work:

```
ls -l | filter
```

But this doesn't tell you how much space each user is using. It also doesn't work for a large directory tree. This requires *find* and *xargs*:

```
find . -type f -print | xargs ls -l | filter
```

The third column of "ls" is the username. The filter has to count how many times it sees each user. The typical program would have an array of usernames and another array that counts how many times each username has been seen. The index to both arrays are the same; you use one array to find the index, and the second to keep track of the count. I'll show you one way to do it in AWK--the wrong way:

```
#!/bin/awk -f
# bad example of AWK programming
# this counts how many files each user owns.
BEGIN {
        number_of_users=0;
}
{
# must make sure you only examine lines with 8 or more fields
        if (NF>7) {
                user=0;
# look for the user in our list of users
                for (i=1; i<=number_of_users; i++) {
#               is the user known?
                        if (username[i] == $3) {
# found it - remember where the user is
                                user=i;
                        }
                }
                if (user == 0) {
# found a new user
                        username[++number_of_users]=$3;
                        user=number_of_users;
                }
# increase number of counts
                count[user]++;
        }
}
END {
        for (i=1; i<=number_of_users; i++) {
                print count[i], username[i]
        }
}
```

I don't want you to **read** this script. I told you it's the wrong way to do it. If you were a C programmer, and didn't know AWK, you would probably use a technique like the one above. Here is the same program, except this example that uses AWK's associative arrays. The important point is to notice the difference in size between these two versions:

```
#!/bin/awk -f
{
        username[$3]++;
}
END {
        for (i in username) {
                print username[i], i;
        }
}
```

This is shorter, simpler, and *much* easier to understand--Once you understand exactly what an associative array is. The concept is simple. Instead of using a number to find an entry in an array, use *anything you want*. An associative array in an array whose index is a string. All arrays in AWK are associative. In this case, the index into the array is the third field of the "ls" command, which is the username. If the user is "bin," the main loop increments the count per user by effectively executing

    username["bin"]++;

UNIX guru's may gleefully report that the 8 line AWK script can be replaced by:

```
awk '{print $3}' | sort | uniq -c | sort -nr
```

True, However, this can't count the total disk space for each user. We need to add some more intelligence to the AWK script, and need the right foundation to proceed. There is also a slight bug in the AWK program. If you wanted a "quick and dirty" solution, the above would be fine. If you wanted to make it more robust, you have to handle unusual conditions. If you gave this program an empty file for input, you would get the error:

```
awk: username is not an array
```

Also, if you piped the output of "ls -l" to it, the line that specified the total would increment a non-existing user. There are two techniques used to eliminate this error. The first one only counts valid input:

```
#!/bin/awk -f
{
	if (NF>7) {
		username[$3]++;
	}
}
END {
	for (i in username) {
		print username[i], i;
	}
}
```

This fixes the problem of counting the line with the total. However, it still generates an error when an empty file is read as input. To fix this problem, a common technique is to make sure the array always exists, and has a special marker value which specifies that the entry is invalid. Then when reporting the results, ignore the invalid entry.

```
#!/bin/awk -f
BEGIN {
	username[""]=0;
}
{
	username[$3]++;
}
END {
	for (i in username) {
		if (i != "") {
			print username[i], i;
		}
	}
}
```

This happens to fix the other problem. Apply this technique and you will make your AWK programs more robust and easier for others to use.

Multi-dimensional Arrays

Some people ask if AWK can handle multi-dimensional arrays. It can. However, you don't use conventional two-dimensional arrays. Instead you use associative arrays. (Did I even mention how useful associative arrays are?) Remember, you can put **anything** in the index of an associative array. It requires a different way to think about problems, but once you understand, you won't be able to live without it. All you have to do is to create an index that combines two other indices. Suppose you wanted to effectively execute

```
a[1,2] = y;
```

This is invalid in AWK. However, the following is perfectly fine:

```
a[1 "," 2] = y;
```

Remember: the AWK string concatenation operator is the space. It combines the three strings into the single string "1,2." Then it uses it as an index into the array. That's all there is to it. There is one minor problem with associative arrays, especially if you use the *for* command to output each element: you have no control over the order of output. You can create an algorithm to generate the indices to an associative array, and control the order this way. However, this is difficult to do. Since UNIX provides an excellent sort utility, more programmers separate the information processing from the sorting. I'll show you what I mean.

Example of using AWK's Associative Arrays

I often find myself using certain techniques repeatedly in AWK. This example will demonstrate these techniques, and illustrate the power and elegance of AWK. The program is simple and common. The disk is full. Who's gonna be blamed? I just hope you use this power wisely. Remember, you may be the one who filled up the disk.

Having resolved my moral dilemma, by placing the burden squarely on your shoulders, I will describe the program in detail. I will also discuss several tips you will find useful in large AWK programs. First, initialize all arrays used in a *for* loop. There will be four arrays for this purpose. Initialization is easy:

```
u_count[""]=0;
g_count[""]=0;
ug_count[""]=0;
all_count[""]=0;
```

The second tip is to pick a convention for arrays. Selecting the names of the arrays, and the indices for each array is very important. In a complex program, it can become confusing to remember which array contains what. I suggest you clearly identify the indices and contents of each array. To demonstrate, I will use a "_count" to indicate the number of files, and "_sum" to indicate the sum of the file sizes. In addition, the part before the "_" specifies the index used for the array, which will be either "u" for user, "g" for group, "ug" for the user and group combination, and "all" for the total for all files. In other programs, I have used names like

```
username_to_directory[username]=directory;
```

Follow a convention like this, and it will be hard for you to forget the purpose of each associative array. Even when a quick hack comes back to haunt you three years later. I've been there.

The third suggestion is to make sure your input is in the correct form. It's generally a good idea to be pessimistic, but I will add a simple but sufficient test in this example.

```
if (NF != 10) {
#       ignore
} else {

etc.
```

I placed the test and error clause up front, so the rest of the code won't be cluttered. AWK doesn't have user defined functions. NAWK, GAWK and PERL do.

The next piece of advice for complex AWK scripts is to define a name for each field used. In this case, we want the user, group and size in disk blocks. We could use the file size in bytes, but the block size corresponds to the blocks on the disk, a more accurate measurement of space. Disk blocks can be found by using "ls -s." This adds a column, so the username becomes the fourth column, etc. Therefore the script will contain:

```
size=$1;
user=$4;
group=$5;
```

This will allow us to easily adapt to changes in input. We could use "$1" throughout the script, but if we changed the number of fields, which the "-s" option does, we'd have to change each field reference. You don't want to go through an AWK script, and change all the "$1" to "$2," and also change the "$2" to "$3" because those are really the "$1" that you just changed to "$2." Of **course** this is confusing. That's why it's a good idea to assign names to the fields. I've been there too.

Next the AWK script will count how many times each combination of users and groups occur. That is, I am going to construct a two-part index that contains the username and groupname. This will let me count up the number of times each user/group combination occurs, and how much disk space is used.

Consider this: how would you calculate the total for just a user, or for just a group? You could rewrite the script. Or you could take the user/group totals, and total them with a second script.

You could do it, but it's not the AWK way to do it. If you had to examine a bazillion files, and it takes a long time to run that script, it would be a waste to repeat this task. It's also inefficient to require two scripts when one can do everything. The proper way to solve this problem is to extract as much information as possible in one pass through the files. Therefore this script will find the number and size for each category:

```
Each user
Each group
Each user/group combination
All users and groups
```

This is why I have 4 arrays to count up the number of files. I don't really need 4 arrays, as I can use the format of the index to determine which array is which. But this does maake the program easier to understand for now. The next tip is subtle, but you will see how useful it is. I mentioned the indices into the array can be anything. If possible, select a format that allows you to merge information from several arrays. I realize this makes no sense right now, but hang in there. All will become clear soon. I will do this by constructing a universal index of the form

```
<user> <group>
```

This index will be used for all arrays. There is a space between the two values. This covers the total for the user/group combination. What about the other three arrays? I will use a "*" to indicate the total for all users or groups. Therefore the index for all files would be "* *" while the index for all of the file owned by user *daemon* would be "daemon *." The heart of the script totals up the number and size of each file, putting the information into the right category. I will use 8 arrays; 4 for file sizes, and 4 for counts:

```
u_count[user " *"]++;
g_count["* " group]++;
ug_count[user " " group]++;
all_count["* *"]++;


u_size[user " *"]+=size;
g_size["* " group]+=size;
ug_size[user " " group]+=size;
all_size["* *"]+=size;
```

This particular universal index will make sorting easier, as you will see. Also important is to sort the information in an order that is useful. You can **try** to force a particular output order in AWK, but why work at this, when it's a one line command for *sort*? The difficult part is finding the right way to sort the information. This script will sort information using the size of the category as the first sort field. The largest total will be the one for all files, so this will be one of the first lines output. However, there may be several ties for the largest number, and care must be used. The second field will be the number of files. This will help break a tie. Still, I want the totals and sub-totals to be listed before the individual user/group combinations. The third and fourth fields will be generated by the index of the array. This is the tricky part I warned you about. The script will output one string, but the *sort* utility will not know this. Instead, it will treat it as two fields. This will unify the results, and information from all 4 arrays will look like one array. The sort of the third and fourth fields will be dictionary order, and not numeric, unlike the first two fields. The "*" was used so these sub-total fields will be listed before the individual user/group combination.

The arrays will be printed using the following format:

```
    for (i in u_count) {
            if (i != "") {
                    print u_size[i], u_count[i], i;
            }
    }
    O
```

I only showed you one array, but all four are printed the same way. That's the essence of the script. The results is sorted, and I converted the space into a tab for cosmetic reasons.

Output of the script

I changed my directory to */usr/ucb*, used the script in that directory. The following is the output:

```
size   count   user   group
3173   81      *      *
3173   81      root   *
2973   75      *      staff
2973   75      root   staff
88     3       *      daemon
88     3       root   daemon
64     2       *      kmem
64     2       root   kmem
48     1       *      tty
48     1       root   tty
```

This says there are 81 files in this directory, which takes up 3173 disk blocks. All of the files are owned by root. 2973 disk blocks belong to group staff. There are 3 files with group daemon, which takes up 88 disk blocks.

As you can see, the first line of information is the total for all users and groups. The second line is the sub-total for the user "root." The third line is the sub-total for the group "staff." Therefore the order of the sort is useful, with the sub-totals before the individual entries. You could write a simple AWK or grep script to obtain information from just one user or one group, and the information will be easy to sort.

There is only one problem. The */usr/ucb* directory on my system only uses 1849 blocks; at least that's what *du* reports. Where's the discrepancy? The script does **not** understand hard links. This may not be a problem on most disks, because many users do not use hard links. Still, it does generate inaccurate results. In this case, the program *vi* is also *e*, *ex*, *edit*, *view*, and 2 other names. The program only exists once, but has 7 names. You can tell because the link count (field 2) reports 7. This causes the file to be counted 7 times, which causes an inaccurate total. The fix is to only count multiple links once. Examining the link count will determine if a file has multiple links. However, how can you prevent counting a link twice? There is an easy solution: all of these files have the same *inode* number. You can find this number with the *-i* option to *ls*. To save memory, we only have to remember the inodes of files that have multiple links. This means we have to add another column to the input, and have to renumber all of the field references. It's a good thing there are only three. Adding a new field will be easy, because I followed my own advice.

The final script should be easy to follow. I have used variations of this hundreds of times and find it demonstrates the power of AWK as well as provide insight to a powerful programming paradigm. AWK solves these types of problems easier than most languages. But you have to use AWK the right way.

Note - this version was written for a Solaris box. You have to verify if *ls* is generating the right number of arguments. The *-g* argument may need to be deleted, and the check for the number of files may have to be modified. **Updated**I added a Linux version below - to be downloaded.

This is a fully working version of the program, that accurately counts disk space, appears below:

```
#!/bin/sh
find . -type f -print | xargs /usr/bin/ls -islg |
awk '
BEGIN {
# initialize all arrays used in for loop
        u_count[""]=0;
        g_count[""]=0;
        ug_count[""]=0;
        all_count[""]=0;
}
{
# validate your input
        if (NF != 11) {
#       ignore
        } else {
# assign field names
                inode=$1;
                size=$2;
                linkcount=$4;
                user=$5;
                group=$6;

# should I count this file?

                doit=0;
                if (linkcount == 1) {
# only one copy - count it
                        doit++;
                } else {
# a hard link - only count first one
                        seen[inode]++;
                        if (seen[inode] == 1) {
                                doit++;
                        }
                }
# if doit is true, then count the file
                if (doit ) {

# total up counts in one pass
# use description array names
# use array index that unifies the arrays

# first the counts for the number of files
```

```
                        u_count[user " *"]++;
                        g_count["* " group]++;
                        ug_count[user " " group]++;
                        all_count["* *"]++;

# then the total disk space used

                        u_size[user " *"]+=size;
                        g_size["* " group]+=size;
                        ug_size[user " " group]+=size;
                        all_size["* *"]+=size;
                }
        }
}
END {
# output in a form that can be sorted
        for (i in u_count) {
                                if (i != "") {
                        print u_size[i], u_count[i], i;
                                }
        }
        for (i in g_count) {
                                if (i != "") {
                        print g_size[i], g_count[i], i;
                                }
        }
        for (i in ug_count) {
                                if (i != "") {
                        print ug_size[i], ug_count[i], i;
                                }
        }
        for (i in all_count) {
                                if (i != "") {
                        print all_size[i], all_count[i], i;
                                }
        }
} ' |
# numeric sort - biggest numbers first
# sort fields 0 and 1 first (sort starts with 0)
# followed by dictionary sort on fields 2 + 3
sort +0nr -2 +2d |
# add header
(echo "size count user group";cat -) |
# convert space to tab - makes it nice output
# the second set of quotes contains a single tab character
tr ' ' '	'
# done - I hope you like it
```

Remember when I said I didn't need to use 4 different arrays? I can use just one. This is more confusing, but more concise

```
#!/bin/sh
find . -type f -print | xargs /usr/bin/ls -islg |
awk '
BEGIN {
# initialize all arrays used in for loop
        count[""]=0;
}
{
# validate your input
        if (NF != 11) {
#       ignore
        } else {
# assign field names
                inode=$1;
                size=$2;
                linkcount=$4;
                user=$5;
                group=$6;

# should I count this file?

                doit=0;
                if (linkcount == 1) {
# only one copy - count it
                        doit++;
                } else {
# a hard link - only count first one
                        seen[inode]++;
                        if (seen[inode] == 1) {
                                doit++;
                        }
                }
# if doit is true, then count the file
                if (doit ) {

# total up counts in one pass
# use description array names
# use array index that unifies the arrays

# first the counts for the number of files

                        count[user " *"]++;
                        count["* " group]++;
                        count[user " " group]++;
                        count["* *"]++;

# then the total disk space used

                        size[user " *"]+=size;
                        size["* " group]+=size;
                        size[user " " group]+=size;
                        size["* *"]+=size;
                }
        }
}
END {
# output in a form that can be sorted
```

```
        for (i in count) {
                if (i != "") {
                        print size[i], count[i], i;
                }
        }
} ' |
# numeric sort - biggest numbers first
# sort fields 0 and 1 first (sort starts with 0)
# followed by dictionary sort on fields 2 + 3
sort +0nr -2 +2d |
# add header
(echo "size count user group";cat -) |
# convert space to tab - makes it nice output
# the second set of quotes contains a single tab character
tr ' ' '	'
# done - I hope you like it
```

Picture Perfect PRINTF Output

So far, I described several simple scripts that provide useful information, in a somewhat ugly output format. Columns might not line up properly, and it is often hard to find patterns or trends without this unity. As you use AWK more, you will be desirous of crisp, clean formatting. To achieve this, you must master the *printf* function.

PRINTF - formatting output

The *printf* is very similar to the C function with the same name. C programmers should have no problem using *printf* function.

Printf has one of these syntactical forms:

```
printf ( format);
printf ( format, arguments...);
printf ( format) >expression;
printf ( format, arguments...) > expression;
```

The parenthesis and semicolon are optional. I only use the first format to be consistent with other nearby *printf* statements. A *print* statement would do the same thing. *Printf* reveals it's real power when formatting commands are used.

The first argument to the *printf* function is the format. This is a string, or variable whose value is a string. This string, like all strings, can contain special escape sequences to print control characters.

Escape Sequences

The character "\" is used to "escape" or mark special characters. The list of these characters is in table below:

AWK Table 5
Escape Sequences

Sequence	Description
\a	ASCII bell (NAWK only)
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage Return
\t	Horizontal tab
\v	Vertical tab (NAWK only)
\ddd	Character (1 to 3 octal digits) (NAWK only)
\xdd	Character (hexadecimal) (NAWK only)
	Any character c

It's difficult to explain the differences without being wordy. Hopefully I'll provide enough examples to demonstrate the differences.

With NAWK, you can print three tab characters using these three different representations:

```
printf("\t\11\x9\n");
```

A tab character is decimal 9, octal 11, or hexadecimal 09. See the man page ascii(7) for more information. Similarly, you can print three double-quote characters (decimal 34, hexadecimal 22, or octal 42) using

```
printf("\"\x22\42\n");
```

You should notice a difference between the *printf* function and the *print* function. *Print* terminates the line with the **ORS** character, and divides each field with the **OFS** separator. *Printf* does nothing unless you specify the action. Therefore you will frequently end each line with the newline character "\n," and you must specify the separating characters explicitly.

Format Specifiers

The power of the *printf* statement lies in the format specifiers, which always start with the character "%." The format specifiers are described in table 6:

AWK Table 6

Format Specifiers

Specifier	Meaning
c	ASCII Character
d	Decimal integer
e	Floating Point number (engineering format)
f	Floating Point number (fixed point format)
g	The shorter of e or f, with trailing zeros removed
o	Octal
s	String
x	Hexadecimal
%	Literal %

Again, I'll cover the differences quickly. Table 3 illustrates the differences. The first line states "printf(%c\n",100.0)"" prints a "d."

AWK Table 7

Example of format conversions

Format	Value	Results
%c	100.0	d
%c	"100.0"	1 (NAWK?)
%c	42	"
%d	100.0	100
%e	100.0	1.000000e+02
%f	100.0	100.000000
%g	100.0	100
%o	100.0	144
%s	100.0	100.0
%s	"13f"	13f
%d	"13f"	0 (AWK)
%d	"13f"	13 (NAWK)
%x	100.0	64

This table reveals some differences between AWK and NAWK. When a string with numbers and letters are coverted into an integer, AWK will return a zero, while NAWK will convert as much as possible. The second example, marked with "NAWK?" will return "d" on some earlier versions of NAWK, while later versions will return "1."

Using format specifiers, there is another way to print a double quote with NAWK. This demonstrates Octal, Decimal and Hexadecimal conversion. As you can see, it isn't symmetrical. Decimal conversions are done differently.

```
printf("%s%s%s%c\n", "\"", "\x22", "\42", 34);
```

Between the "%" and the format character can be four optional pieces of information. It helps to visualize these fields as:

```
%<sign><zero><width>.<precision>format
```

I'll discuss each one separately.

Width - specifying minimum field size

If there is a number after the "%," this specifies the minimum number of characters to print. This is the *width* field. Spaces are added so the number of printed characters equal this number. Note that this is the minimum field size. If the field becomes to large, it will grow, so information will not be lost. Spaces are added to the left.

This format allows you to line up columns perfectly. Consider the following format:

```
printf("%st%d\n", s, d);
```

If the string "s" is longer than 8 characters, the columns won't line up. Instead, use

```
printf("%20s%d\n", s, d);
```

As long as the string is less than 20 characters, the number will start on the 21st column. If the string is too long, then the two fields will run together, making it hard to read. You may want to consider placing a single space between the fields, to make sure you will always have one space between the fields. This is very important if you want to pipe the output to another program.

Adding informational headers makes the output more readable. Be aware that changing the format of the data may make it difficult to get the columns aligned perfectly. Consider the following script:

```
#!/usr/bin/awk -f
BEGIN {
	printf("String     Number\n");
}
{
	printf("%10s %6d\n", $1, $2);
}
```

It would be awkward (forgive the choice of words) to add a new column and retain the same alignment.

More complicated formats would require a lot of trial and error. You have to adjust the first *printf* to agree with the second *printf* statement. I suggest

```
#!/usr/bin/awk -f
BEGIN {
        printf("%10s %6sn", "String", "Number");
}
{
        printf("%10s %6d\n", $1, $2);
}
```

or even better

```
#!/usr/bin/awk -f
BEGIN {
        format1 ="%10s %6sn";
        format2 ="%10s %6dn";
        printf(format1, "String", "Number");
}
{
        printf(format2, $1, $2);
}
```

The last example, by using string variables for formatting, allows you to keep all of the formats together. This may not seem like it's very useful, but when you have multiple formats and multiple columns, it's very useful to have a set of templates like the above. If you have to add an extra space to make things line up, it's much easier to find and correct the problem with a set of format strings that are together, and the exact same width. CHainging the first columne from 10 characters to 11 is easy.

Left Justification

The last example places spaces before each field to make sure the minimum field width is met. What do you do if you want the spaces on the right? Add a negative sign before the width:

```
printf("%-10s %-6d\n", $1, $2);
```

This will move the printing characters to the left, with spaces added to the right.

The Field Precision Value

The precision field, which is the number between the decimal and the format character, is more complex. Most people use it with the floating point format (%f), but surprisingly, it can be used with any format character. With the octal, decimal or hexadecimal format, it specifies the minimum number of characters. Zeros are added to met this requirement. With the %e and %f formats, it specifies the number of digits after the decimal point. The %e "e+00" is not included in the precision. The %g format combines the characteristics of the %d and %f formats. The precision specifies the number of digits displayed, before and after the decimal point. The precision field has no effect on the %c field. The %s format has an unusual, but useful effect: it specifies the maximum number of significant characters to print.

If the first number after the "%," or after the "%-," is a zero, then the system adds zeros when padding. This includes all format types, including strings and the %c character format. This means "%010d" and "%.10d" both adds leading zeros, giving a minimum of 10 digits. The format "%10.10d" is therefore redundant. Table 8 gives some examples:

AWK Table 8
Examples of complex formatting

Format	Variable	Results
%c	100	`"d"`
%10c	100	`"         d"`
%010c	100	`"000000000d"`
%d	10	`"10"`
%10d	10	`"        10"`
%10.4d	10.123456789	`"      0010"`
%10.8d	10.123456789	`"  00000010"`
%.8d	10.123456789	`"00000010"`
%010d	10.123456789	`"0000000010"`
%e	987.1234567890	`"9.871235e+02"`
%10.4e	987.1234567890	`"9.8712e+02"`
%10.8e	987.1234567890	`"9.87123457e+02"`
%f	987.1234567890	`"987.123457"`
%10.4f	987.1234567890	`"  987.1235"`
%010.4f	987.1234567890	`"00987.1235"`
%10.8f	987.1234567890	`"987.12345679"`
%g	987.1234567890	`"987.123"`
%10g	987.1234567890	`"   987.123"`
%10.4g	987.1234567890	`"     987.1"`
%010.4g	987.1234567890	`"00000987.1"`

```
|%.8g      987.1234567890   "987.12346"      |
+--------------------------------------------+
|%o        987.1234567890   "1733"           |
|%10o      987.1234567890   "      1733"     |
|%010o     987.1234567890   "0000001733"     |
|%.8o      987.1234567890   "00001733"       |
+--------------------------------------------+
|%s        987.123          "987.123"        |
|%10s      987.123          "   987.123"     |
|%10.4s    987.123          "      987."     |
|%010.8s   987.123          "000987.123"     |
+--------------------------------------------+
|%x        987.1234567890   "3db"            |
|%10x      987.1234567890   "       3db"     |
|%010x     987.1234567890   "00000003db"     |
|%.8x      987.1234567890   "000003db"       |
+--------------------------------------------+
```

There is one more topic needed to complete this lesson on *printf.*

Explicit File output

Instead of sending output to standard output, you can send output to a named file. The format is

```
printf("string\n") > "/tmp/file";
```

You can append to an existing file, by using ">>:"

```
printf("string\n") >> "/tmp/file";
```

Like the shell, the double angle brackets indicates output is **appended** to the file, instead of **written** to an empty file. Appending to the file does not delete the old contents. However, there is a subtle difference between AWK and the shell.

Consider the shell program:

```
#!/bin/sh
while x=`line`
do
        echo got $x >>/tmp/a
        echo got $x >/tmp/b
done
```

This will read standard input, and copy the standard input to files "/tmp/a" and "/tmp/b." File "/tmp/a" will grow larger, as information is always appended to the file. File "/tmp/b," however, will only contain one line. This happens because each time the shell see the ">" or ">>" characters, it opens the file for writing, choosing the truncate/create or appending option at that time.

Now consider the equivalent AWK program:

```
#!/usr/bin/awk -f
{
    print $0 >>"/tmp/a"
    print $0 >"/tmp/b"
}
```

This behaves differently. AWK chooses the create/append option the first time a file is opened for writing. Afterwards, the use of ">" or ">>" is ignored. Unlike the shell, AWK copies all of standard input to file "/tmp/b."

Instead of a string, some versions of AWK allow you to specify an expression:

```
# [note to self] check this one - it might not work
printf("string\n") > FILENAME ".out";
```

The following uses a string concatenation expression to illustrate this:

```
#!/usr/bin/awk -f
END {
    for (i=0;i<30;i++) {
        printf("i=%d\n", i) > "/tmp/a" i;
    }
}
```

This script never finishes, because AWK can have 10 additional files open, and NAWK can have 20. If you find this to be a problem, look into PERL.

I hope this gives you the skill to make your AWK output picture perfect.

AWK Numerical Functions

In previous tutorials, I have shown how useful AWK is in manipulating information, and generating reports. When you add a few functions, AWK becomes even more, mmm, functional.

There are three types of functions: numeric, string and whatever's left. Table9 lists all of the numeric functions:

AWK Table 9

Numeric Functions

Name	Function	Variant
cos	cosine	AWK
exp	Exponent	AWK
int	Integer	AWK

```
|log     Logarithm     AWK         |
|sin     Sine          AWK         |
|sqrt    Square Root   AWK         |
|atan2   Arctangent    NAWK        |
|rand    Random        NAWK        |
|srand   Seed Random   NAWK        |
+----------------------------------+
```

Trigonometric Functions

Oh joy. I bet millions, if not dozens, of my readers have been waiting for me to discuss trigonometry. Personally, I don't use trigonometry much at work, except when I go off on a tangent.

Sorry about that. I don't know what came over me. I don't usually resort to puns. I'll write a note to myself, and after I sine the note, I'll have my boss cosine it.

Now stop that! I hate arguing with myself. I always lose. Thinking about math I learned in the year 2 B.C. (Before Computers) seems to cause flashbacks of high school, pimples, and (shudder) times best left forgotten. The stress of remembering those days must have made me forget the standards I normally set for myself. Besides, no-one appreciates obtuse humor anyway, even if I find acute way to say it.

I better change the subject fast. Combining humor and computers is a very serious matter.

Here is a NAWK script that calculates the trigonometric functions for all degrees between 0 and 360. It also shows why there is no tangent, secant or cosecant function. (They aren't necessary). If you read the script, you will learn of some subtle differences between AWK and NAWK. All this in a thin veneer of demonstrating why we learned trigonometry in the first place. What more can you ask for? Oh, in case you are wondering, I wrote this in the month of December.

```
#!/usr/bin/nawk -f
#
# A smattering of trigonometry...
#
# This AWK script plots the values from 0 to 360
# for the basic trigonometry functions
# but first - a review:
#
# (Note to the editor - the following diagram assumes
# a fixed width font, like Courier.
# otherwise, the diagram  looks very stupid, instead of slightly stupid)
#
# Assume the following right triangle
#
#       Angle Y
#
#       |
#       |
#       |
#     a |     c
#       |
#       |
#       +-------   Angle X
#          b
#
# since the triangle is a right angle, then
#       X+Y=90
#
# Basic Trigonometric Functions. If you know the length
# of 2 sides, and the angles, you can find the length of the third side.
# Also - if you know the length of the sides, you can calculate
# the angles.
#
# The formulas are
#
#       sine(X) = a/c
#       cosine(X) = b/c
#       tangent(X) = a/b
#
# reciprocal functions
#       cotangent(X) = b/a
#       secant(X) = c/b
#       cosecant(X) = c/a
#
# Example 1)
# if an angle is 30, and the hypotenuse (c) is 10, then
#       a = sine(30) * 10 = 5
#       b = cosine(30) * 10 =  8.66
#
# The second example will be more realistic:
#
#       Suppose you are looking for a Christmas tree, and
# while talking to your family, you smack into a tree
# because your head was turned, and your kids were arguing over who
# was going to put the first ornament on the tree.
#
# As you come to, you realize your feet are touching the trunk of the tree,
# and your eyes are 6 feet from the bottom of your frostbitten toes.
# While counting the stars that spin around your head, you also realize
# the top of the tree is located at a 65 degree angle, relative to your eyes.

# You suddenly realize the tree is 12.84 feet high! After all,
#       tangent(65 degrees) * 6 feet = 12.84 feet

# All right, it isn't realistic. Not many people memorize the
# tangent table, or can estimate angles that accurately.
# I was telling the truth about the stars spinning around the head, however.

#
BEGIN {
# assign a value for pi.
        PI=3.14159;
```

```
# select an "Ed Sullivan" number - really really big
        BIG=999999;
# pick two formats
# Keep them close together, so when one column is made larger
# the other column can be adjusted to be the same width
        fmt1="%7s %8s %8s %8s %10s %10s %10s %10sn";
# print out the title of each column
        fmt2="%7d %8.2f %8.2f %8.2f %10.2f %10.2f %10.2f %10.2fn";
# old AWK wants a backslash at the end of the next line
# to continue the print statement
# new AWK allows you to break the line into two, after a comma
        printf(fmt1,"Degrees","Radians","Cosine","Sine",
                "Tangent","Cotangent","Secant", "Cosecant");

        for (i=0;i<=360;i++) {
# convert degrees to radians
                r = i * (PI / 180 );
# in new AWK, the backslashes are optional
# in OLD AWK, they are required
                printf(fmt2, i, r,
# cosine of r
                cos(r),
# sine of r
                sin(r),
#
# I ran into a problem when dividing by zero.
# So I had to test for this case.
#
# old AWK finds the next line too complicated
# I don't mind adding a backslash, but rewriting the
# next three lines seems pointless for a simple lesson.
# This script will only work with new AWK, now - sigh...
# On the plus side,
#   I don't need to add those back slashes anymore
#
# tangent of r
                (cos(r) == 0) ? BIG : sin(r)/cos(r),
# cotangent of r
                (sin(r) == 0) ? BIG : cos(r)/sin(r),
# secant of r
                (cos(r) == 0) ? BIG : 1/cos(r),
# cosecant of r
                (sin(r) == 0) ? BIG : 1/sin(r));
        }
# put an exit here, so that standard input isn't needed.
        exit;
}
```

NAWK also has the arctangent function. This is useful for some graphics work, as

arc tangent(a/b) = angle (in radians)

Therefore if you have the X and Y locations, the arctangent of the ratio will tell you the angle. The *atan2*() function returns a value from negative *pi* to positive *pi*.

Exponents, logs and square roots

The following script uses three other arithmetic functions: *log*, *exp*, and *sqrt*. I wanted to show how these can be used together, so I divided the log of a number by two, which is another way to find a square root. I then compared the value of the exponent of that new log to the built-in square root function. I then calculated the difference between the two, and converted the difference into a positive number.

```
#!/bin/awk -f
# demonstrate use of exp(), log() and sqrt in AWK
# e.g. what is the difference between using logarithms and regular arithmetic
# note - exp and log are natural log functions - not base 10
#
BEGIN {
# what is the about of error that will be reported?
	ERROR=0.000000000001;
# loop a long while
	for (i=1;i<=2147483647;i++) {
# find log of i
		logi=log(i);
# what is square root of i?
# divide the log by 2
		logsquareroot=logi/2;
# convert log of i back
		squareroot=exp(logsquareroot);
# find the difference between the logarithmic calculation
# and the built in calculation
		diff=sqrt(i)-squareroot;
# make difference positive
		if (diff < 0) {
			diff*=-1;
		}
		if (diff > ERROR) {
			printf("%10d, squareroot: %16.8f, error: %16.14f\n",
				i, squareroot, diff);
		}
	}
	exit;
}
```

Yawn. This example isn't too exciting, except to those who enjoy nitpicking. Expect the program to reach 3 million before you see any errors. I'll give you a more exciting sample soon.

Truncating Integers

All version of AWK contain the *int* function. This truncates a number, making it an integer. It can be used to round numbers by adding 0.5:

```
printf("rounding %8.4f gives %8dn", x, int(x+0.5));
```

Random Numbers

NAWK has functions that can generate random numbers. The function *rand* returns a random number between 0 and 1. Here is an example that calculates a million random numbers between 0 and 100, and counts how often each number was used:

```
#!/usr/bin/nawk -f
# old AWK doesn't have rand() and srand()
# only new AWK has them
# how random is the random function?
BEGIN {
#       srand();
        i=0;
        while (i++<1000000) {
                x=int(rand()*100 + 0.5);
                y[x]++;
        }
        for (i=0;i<=100;i++) {
                printf("%dt%d\n",y[i],i);
        }
        exit;
}
```

If you execute this script several times, you will get the exact same results. Experienced programmers know random number generators aren't really random, unless they use special hardware. These numbers are pseudo-random, and calculated using some algorithm. Since the algorithm is fixed, the numbers are repeatable unless the numbers are seeded with a unique value. This is done using the *srand* function above, which is commented out. Typically the random number generator is not given a special seed until the bugs have been worked out of the program. There's nothing more frustrating than a bug that occurs randomly. The *srand* function may be given an argument. If not, it uses the current time and day to generate a seed for the random number generator.

The Lotto script

I promised a more useful script. This may be what you are waiting for. It reads two numbers, and generates a list of random numbers. I call the script "lotto.awk."

```
#!/usr/bin/nawk -f
BEGIN {
# Assume we want 6 random numbers between 1 and 36
# We could get this information by reading standard input,
# but this example will use a fixed set of parameters.
#
# First, initialize the seed
    srand();
# How many numbers are needed?
    NUM=6;
# what is the minimum number
    MIN=1;
# and the maximum?
    MAX=36;
# How many numbers will we find? start with 0
    Number=0;
    while (Number < NUM) {
        r=int(((rand() *(1+MAX-MIN))+MIN));
# have I seen this number before?
        if (array[r] == 0) {
# no, I have not
            Number++;
            array[r]++;
        }
    }

# now output all numbers, in order
    for (i=MIN;i<=MAX;i++) {
# is it marked in the array?
        if (array[i]) {
# yes
            printf("%d ",i);
        }
    }
    printf("\n");
    exit;
}
```

If you do win a lottery, send me a postcard.

String Functions

Besides numeric functions, there are two other types of function: strings and the whatchamacallits. First, a list of the string functions:

AWK Table 10
String Functions

Name	Variant
index(string,search)	AWK, NAWK, GAWK
length(string)	AWK, NAWK, GAWK
split(string,array,separator)	AWK, NAWK, GAWK
substr(string,position)	AWK, NAWK, GAWK
substr(string,position,max)	AWK, NAWK, GAWK
sub(regex,replacement)	NAWK, GAWK
sub(regex,replacement,string)	NAWK, GAWK
gsub(regex,replacement)	NAWK, GAWK
gsub(regex,replacement,string)	NAWK, GAWK
match(string,regex)	NAWK, GAWK
tolower(string)	GAWK
toupper(string)	GAWK

Most people first use AWK to perform simple calculations. Associative arrays and trigonometric functions are somewhat esoteric features, that new users embrace with the eagerness of a chain smoker in a fireworks factory. I suspect most users add some simple string functions to their repertoire once they want to add a little more sophistication to their AWK scripts. I hope this column gives you enough information to inspire your next effort.

There are four string functions in the original AWK: *index*(), *length*(), *split*(), and *substr*(). These functions are quite versatile.

The Length function

What can I say? The *length*() function calculates the length of a string. I often use it to make sure my input is correct. If you wanted to ignore empty lines, check the length of the each line before processing it with

```
if (length($0) > 1) {
. . .
}
```

You can easily use it to print all lines longer than a certain length, etc. The following command centers all lines shorter than 80 characters:

```
#!/bin/awk -f
{
    if (length($0) < 80) {
        prefix = "";
        for (i = 1;i<(80-length($0))/2;i++)
            prefix = prefix " ";
        print prefix $0;
    } else {
        print;
    }
}
```

The Index Function

If you want to search for a special character, the *index*() function will search for specific characters inside a string. To find a comma, the code might look like this:

```
sentence="This is a short, meaningless sentence.";
if (index(sentence, ",") > 0) {
printf("Found a comma in position \%d\n", index(sentence,","));
}
```

The function returns a positive value when the substring is found. The number specified the location of the substring.

If the substring consists of 2 or more characters, all of these characters must be found, in the same order, for a non-zero return value. Like the *length*() function, this is useful for checking for proper input conditions.

The Substr function

The *substr*() function can extract a portion of a string. One common use is to split a string into two parts based on a special character. If you wanted to process some mail addresses, the following code fragment might do the job:

```
#!/bin/awk -f
{
# field 1 is the e-mail address - perhaps
```

```
    if ((x=index($1,"@")) > 0) {
        username = substr($1,1,x-1);
        hostname = substr($1,x+1,length($1));
# the above is the same as
#        hostname = substr($1,x+1);
        printf("username = %s, hostname = %s\n", username, hostname);
    }
}
```

The *substr*() function takes two or three arguments. The first is the string, the second is the position. The optional third argument is the length of the string to extract. If the third argument is missing, the rest of the string is used.

The substr function can be used in many non-obvious ways. As an example, it can be used to convert upper case letters to lower case.

```
#!/usr/bin/awk -f
# convert upper case letters to lower case
BEGIN {
    LC="abcdefghijklmnopqrstuvwxyz";
    UC="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
}
{
    out="";
# look at each character
    for(i=1;i<=length($0);i++) {
# get the character to be checked
        char=substr($0,i,1);
# is it an upper case letter?
        j=index(UC,char);
            if (j > 0 ) {
# found it
                out = out substr(LC,j,1);
            } else {
                out = out char;
            }
    }
    printf("%s\n", out);
}
```

GAWK's Tolower and Toupper function

GAWK has the *toupper*() and *tolower*() functions, for convenient conversions of case. These functions take strings, so you can reduce the above script to a single line:

```
#!/usr/local/bin/gawk -f
{
    print tolower($0);
}
```

The Split function

Another way to split up a string is to use the *split*() function. It takes three arguments: the string, an array, and the separator. The function returns the number of pieces found. Here is an example:

```
#!/usr/bin/awk -f
BEGIN {
# this script breaks up the sentence into words, using
# a space as the character separating the words
    string="This is a string, is it not?";
    search=" ";
    n=split(string,array,search);
    for (i=1;i<=n;i++) {
        printf("Word[%d]=%s\n",i,array[i]);
    }
    exit;
}
```

The third argument is typically a single character. If a longer string is used, only the first letter is used as a separator.

NAWK's string functions

NAWK (and GAWK) have additional string functions, which add a primitive SED-like functionality: *sub*(), *match*(), and *gsub*().

Sub() performs a string substitution, like *sed*. To replace "old" with "new" in a string, use

    sub(/old/, "new", string)

If the third argument is missing, $0 is assumed to be string searched. The function returns 1 if a substitution occurs, and 0 if not. If no slashes are given in the first argument, the first argument is assumed to be a variable containing a regular expression. The *sub*() only changes the first occurrence. The *gsub*() function is similar to the **g** option in *sed*: all occurrence are converted, and not just the first. That is, if the patter occurs more than once per line (or string), the substitution will be performed once for each found pattern. The following script:

```
#!/usr/bin/nawk -f
BEGIN {
    string = "Another sample of an example sentence";
    pattern="[Aa]n";
    if (gsub(pattern,"AN",string)) {
        printf("Substitution occurred: %s\n", string);
    }

    exit;
}
```

print the following when executed:

```
Substitution occurred: ANother sample of AN example sentence
```

As you can see, the pattern can be a regular expression.

The Match function

As the above demonstrates, the *sub*() and *gsub*() returns a positive value if a match is found. However, it has a side-effect of changing the string tested. If you don't wish this, you can copy the string to another variable, and test the spare variable. NAWK also provides the *match*() function. If *match*() finds the regular expression, it sets two special variables that indicate where the regular expression begins and ends. Here is an example that does this:

```
#!/usr/bin/nawk -f
# demonstrate the match function

BEGIN {
    regex="[a-zA-Z0-9]+";
}
{
    if (match($0,regex)) {
#           RSTART is where the pattern starts
#           RLENGTH is the length of the pattern
            before = substr($0,1,RSTART-1);
            pattern = substr($0,RSTART,RLENGTH);
            after = substr($0,RSTART+RLENGTH);
            printf("%s<%s>%s\n", before, pattern, after);
    }
}
```

Lastly, there are the whatchamacallit functions. I could use the word "miscellaneous," but it's too hard to spell. Darn it, I had to look it up anyway.

AWK Table 11
Miscellaneous Functions

Name	Variant
getline	AWK, NAWK, GAWK
getline <file	NAWK, GAWK
getline variable	NAWK, GAWK
getline variable <file	NAWK, GAWK
"command" | getline	NAWK, GAWK
"command" | getline variable	NAWK, GAWK
system(command)	NAWK, GAWK
close(command)	NAWK, GAWK
systime()	GAWK
strftime(string)	GAWK
strftime(string, timestamp)	GAWK

The System function

NAWK has a function *system*() that can execute any program. It returns the exit status of the program.

```
    if (system("/bin/rm junk") != 0)
    print "command didn't work";
```

The command can be a string, so you can dynamically create commands based on input. Note that the output isn't sent to the NAWK program. You could send it to a file, and open that file for reading. There is another solution, however.

The Getline function

AWK has a command that allows you to force a new line. It doesn't take any arguments. It returns a 1, if successful, a 0 if end-of-file is reached, and a -1 if an error occurs. As a side effect, the line containing the input changes. This next script filters the input, and if a backslash occurs at the end of the line, it reads the next line in, eliminating the backslash as well as the need for it.

```
#!/usr/bin/awk -f
# look for a  as the last character.
# if found, read the next line and append
{
    line = $0;
    while (substr(line,length(line),1) == "\\") {
```

```
# chop off the last character
        line = substr(line,1,length(line)-1);
        i=getline;
        if (i > 0) {
            line = line $0;
        } else {
            printf("missing continuation on line %d\n", NR);
        }
    }
    print line;
}
```

Instead of reading into the standard variables, you can specify the variable to set:

```
getline a_line
print a_line;
```

NAWK and GAWK allow the *getline* function to be given an optional filename or string containing a filename. An example of a primitive file preprocessor, that looks for lines of the format

```
#include filename
```

and substitutes that line for the contents of the file:

```
#!/usr/bin/nawk -f
{
# a primitive include preprocessor
    if (($1 == "#include") && (NF == 2)) {
# found the name of the file
        filename = $2;
        while (i = getline < filename ) {
            print;
        }
    } else {
        print;
    }
}
```

NAWK's getline can also read from a pipe. If you have a program that generates single line, you can use

```
"command" | getline;
print $0;
```

or

```
"command" | getline abc;
print abc;
```

If you have more than one line, you can loop through the results:

```
    while ("command" | getline) {
        cmd[i++] = $0;
    }
    for (i in cmd) {
        printf("%s=%s\n", i, cmd[i]);
    }
```

Only one pipe can be open at a time. If you want to open another pipe, you must execute

```
    close("command");
```

This is necessary even if the end of file is reached.

The systime function

The *systime*() function returns the current time of day as the number of seconds since Midnight, January 1, 1970. It is useful for measuring how long portions of your GAWK code takes to execute.

```
#!/usr/local/bin/gawk -f
# how long does it take to do a few loops?
BEGIN {
    LOOPS=100;
# do the test twice
    start=systime();
    for (i=0;i<LOOPS;i++) {
    }
    end = systime();
# calculate how long it takes to do a dummy test
    do_nothing = end-start;
# now do the test again with the *IMPORTANT* code inside
    start=systime();
    for (i=0;i<LOOPS;i++) {
# How long does this take?
        while ("date" | getline) {
            date = $0;
        }
        close("date");
    }
    end = systime();
    newtime = (end - start) - do_nothing;

    if (newtime <= 0) {
        printf("%d loops were not enough to test, increase it\n",
            LOOPS);
        exit;
    } else {
        printf("%d loops took %6.4f seconds to execute\n",
            LOOPS, newtime);
        printf("That's %10.8f seconds per loop\n",
            (newtime)/LOOPS);
# since the clock has an accuracy of +/- one second, what is the error
        printf("accuracy of this measurement = %6.2f%%\n",
            (1/(newtime))*100);
    }
    exit;
}
```

The Strftime function

GAWK has a special function for creating strings based on the current time. It's based on the *strftime*(3c) function. If you are familiar with the "+" formats of the *date*(1) command, you have a good head-start on understanding what the *strftime* command is used for. The *systime*() function returns the current date in seconds. Not very useful if you want to create a string based on the time. While you could convert the seconds into days, months, years, etc., it would be easier to execute "date" and pipe the results into a string. (See the previous script for an example). GAWK has another solution that eliminates the need for an external program.

The function takes one or two arguments. The first argument is a string that specified the format. This string contains regular characters and special characters. Special characters start with a backslash or the percent character. The backslash characters with the backslash prefix are the same I covered earlier. In addition, the *strftime*() function defines dozens of combinations, all of which start with "%." The following table lists these special sequences:

AWK Table 12
GAWK's strftime formats

%a	The locale's abbreviated weekday name
%A	The locale's full weekday name
%b	The locale's abbreviated month name
%B	The locale's full month name
%c	The locale's "appropriate" date and time representation
%d	The day of the month as a decimal number (01--31)
%H	The hour (24-hour clock) as a decimal number (00--23)
%I	The hour (12-hour clock) as a decimal number (01--12)
%j	The day of the year as a decimal number (001--366)
%m	The month as a decimal number (01--12)
%M	The minute as a decimal number (00--59)
%p	The locale's equivalent of the AM/PM
%S	The second as a decimal number (00--61).
%U	The week number of the year (Sunday is first day of week)
%w	The weekday as a decimal number (0--6). Sunday is day 0
%W	The week number of the year (Monday is first day of week)
%x	The locale's "appropriate" date representation

%X	The locale's "appropriate" time representation
%y	The year without century as a decimal number (00--99)
%Y	The year with century as a decimal number
%Z	The time zone name or abbreviation
%%	A literal %.

Depending on your operating system, and installation, you may also have the following formats:

AWK Table 13
Optional GAWK strftime formats

%D	Equivalent to specifying %m/%d/%y
%e	The day of the month, padded with a blank if it is only one digit
%h	Equivalent to %b, above
%n	A newline character (ASCII LF)
%r	Equivalent to specifying %I:%M:%S %p
%R	Equivalent to specifying %H:%M
%T	Equivalent to specifying %H:%M:%S
%t	A TAB character
%k	The hour as a decimal number (0-23)
%l	The hour (12-hour clock) as a decimal number (1-12)
%C	The century, as a number between 00 and 99
%u	is replaced by the weekday as a decimal number [Monday == 1]
%V	is replaced by the week number of the year (using ISO 8601)
%v	The date in VMS format (e.g. 20-JUN-1991)

One useful format is

```
strftime("%y_%m_%d_%H_%M_%S")
```

This constructs a string that contains the year, month, day, hour, minute and second in a format that allows convenient sorting. If you ran this at noon on Christmas, 1994, it would generate the string

```
94_12_25_12_00_00
```

Here is the GAWK equivalent of the date command:

```
#! /usr/local/bin/gawk -f
#

BEGIN {
    format = "%a %b %e %H:%M:%S %Z %Y";
    print strftime(format);
}
```

You will note that there is no exit command in the begin statement. If I was using AWK, an exit statement is necessary. Otherwise, it would never terminate. If there is no action defined for each line read, NAWK and GAWK do not need an exit statement.

If you provide a second argument to the *strftime*() function, it uses that argument as the timestamp, instead of the current system's time. This is useful for calculating future times. The following script calculates the time one week after the current time:

```
#!/usr/local/bin/gawk -f
BEGIN {
#  get current time
   ts = systime();
# the time is in seconds, so
   one_day = 24 * 60 * 60;
   next_week = ts + (7 * one_day);
   format = "%a %b %e %H:%M:%S %Z %Y";
   print strftime(format, next_week);
   exit;
}
```

User Defined Functions

Finally, NAWK and GAWK support user defined functions. This function demonstrates a way to print error messages, including the filename and line number, if appropriate:

```
#!/usr/bin/nawk -f
{
    if (NF != 4) {
        error("Expected 4 fields");
    } else {
        print;
    }
}
function error ( message ) {
    if (FILENAME != "-") {
        printf("%s: ", FILENAME) > "/dev/tty";
    }
    printf("line # %d, %s, line: %s\n", NR, message, $0) >> "/dev/tty";
}
```

AWK patterns

In my first tutorial on AWK, I described the AWK statement as having the form

pattern {commands}

I have only used two patterns so far: the special words *BEGIN* and *END*. Other patterns are possible, yet I haven't used any. There are several reasons for this. The first is that these patterns aren't necessary.

You can duplicate them using an *if* statement. Therefore this is an "advanced feature." Patterns, or perhaps the better word is conditions, tend to make an AWK program obscure to a beginner. You can think of them as an advanced topic, one that should be attempted after becoming familiar with the basics.

A pattern or condition is simply an abbreviated test. If the condition is true, the action is performed. All relational tests can be used as a pattern. The "head -10" command, which prints the first 10 lines and stops, can be duplicated with

```
{if (NR <= 10 ) {print}}
```

Changing the *if* statement to a condition shortens the code:

```
NR <= 10 {print}
```

Besides relational tests, you can also use containment tests, i. e. do strings contain regular expressions? Printing all lines that contain the word "special" can be written as

```
{if ($0 ~ /special/) {print}}
```

or more briefly

```
$0 ~ /special/ {print}
```

This type of test is so common, the authors of AWK allow a third, shorter format:

```
/special/ {print}
```

These tests can be combined with the AND (*&&*) and OR (||) commands, as well as the NOT (*!*) operator. Parenthesis can also be added if you are in doubt, or to make your intention clear.

The following condition prints the line if it contains the word "whole" or columns 1 and 2 contain "part1" and "part2" respectively.

```
($0 ~ /whole/) || (($1 ~ /part1/) && ($2 ~ /part2/)) {print}
```

This can be shortened to

```
/whole/ || $1 ~ /part1/ && $2 ~ /part2/ {print}
```

There is one case where adding parenthesis hurts. The condition

```
/whole/ {print}
```

works, but

```
(/whole/) {print}
```

does not. If parenthesis are used, it is necessary to explicitly specify the test:

```
($0 ~ /whole) {print}
```

A murky situation arises when a simple variable is used as a condition. Since the variable *NF* specifies the number of fields on a line, one might think the statement

```
NF {print}
```

would print all lines with one of more fields. This is an illegal command for AWK, because AWK does not accept variables as conditions. To prevent a syntax error, I had to change it to

```
NF != 0 {print}
```

I expected NAWK to work, but on some SunOS systems it refused to print any lines at all. On newer Solaris systems it did behave properly. Again, changing it to the longer form worked for all variations. GAWK, like the newer version of NAWK, worked properly. After this experience, I decided to leave other, exotic variations alone. Clearly this is unexplored territory. I could write a script that prints the first 20 lines, except if there were exactly three fields, unless it was line 10, by using

```
NF == 3 ? NR == 10 : NR < 20 { print }
```

But I won't. Obscurity, like puns, is often unappreciated.

There is one more common and useful pattern I have not yet described. It is the comma separated pattern. A common example has the form:

```
/start/,/stop/ {print}
```

This form defines, in one line, the condition to turn the action on, and the condition to turn the action off. That is, when a line containing "start" is seen, it is printed. Every line afterwards is also printed, until a line containing "stop" is seen. This one is also printed, but the line after, and all following lines, are not printed. This triggering on and off can be repeated many times. The equivalent code, using the *if* command, is:

```
{
  if ($0 ~ /start/) {
    triggered=1;
  }
  if (triggered) {
     print;
     if ($0 ~ /stop/) {
        triggered=0;
     }
  }
}
```

The conditions do not have to be regular expressions. Relational tests can also be used. The following prints all lines between 20 and 40:

```
(NR==20),(NR==40) {print}
```

You can mix relational and containment tests. The following prints every line until a "stop" is seen:

```
(NR==1),/stop/ {print}
```

There is one more area of confusion about patterns: each one is independent of the others. You can have several patterns in a script; none influence the other patterns. If the following script is executed:

```
NR==10 {print}
(NR==5),(NR==15) {print}
/xxx/ {print}
(NR==1),/NeVerMatchThiS/ {print}
```

and the input file's line 10 contains "xxx," it would be printed 4 times, as each condition is true. You can think of each condition as cumulative. The exception is the special **BEGIN** and **END** conditions. In the original AWK, you can only have one of each. In NAWK and GAWK, you can have several BEGIN or END actions.

Formatting AWK programs

Many readers have questions my style of AWK programming. In particular, they ask me why I include code like this:

```
# Print column 3
print $3;
```

when I could use

```
print $3 # print column 3
```

After all, they reason, the semicolon is unnecessary, and comments do not have to start on the first column. This is true. Still I avoid this. Years ago, when I started writing AWK programs, I would find myself confused when the nesting of conditions were too deep. If I moved a complex *if* statement inside another *if* statement, my alignment of braces became incorrect. It can be very difficult to repair this condition, especially with large scripts. Nowadays I use *emacs* to do the formatting for me, but 10 years ago I didn't have this option. My solution was to use the program *cb*, which is a "C beautifier." By including optional semicolons, and starting comments on the first column of the line, I could send my AWK script through this filter, and properly align all of the code.

Environment Variables

I've described the 7 special variables in AWK, and briefly mentioned some others NAWK and GAWK. The complete list follows:

AWK Table 14

Variable	AWK	NAWK	GAWK

FS	Yes	Yes	Yes
NF	Yes	Yes	Yes
RS	Yes	Yes	Yes
NR	Yes	Yes	Yes
FILENAME	Yes	Yes	Yes
OFS	Yes	Yes	Yes
ORS	Yes	Yes	Yes
ARGC		Yes	Yes
ARGV		Yes	Yes
ARGIND			Yes
FNR		Yes	Yes
OFMT		Yes	Yes
RSTART		Yes	Yes
RLENGTH		Yes	Yes
SUBSEP		Yes	Yes
ENVIRON			Yes
IGNORECASE			Yes
CONVFMT			Yes
ERRNO			Yes
FIELDWIDTHS			Yes

Since I've already discussed many of these, I'll only cover those that I missed earlier.

ARGC - Number or arguments (NAWK/GAWK)

The variable **ARGC** specifies the number of arguments on the command line. It always has the value of one or more, as it counts its own program name as the first argument. If you specify an AWK filename using the "-f" option, it is not counted as an argument. If you include a variable on the command line using the form below, NAWK does not count as an argument.

```
nawk -f file.awk x=17
```

GAWK does, but that is because GAWK requires the "-v" option before each assignment:

```
gawk -f file.awk -v x=17
```

GAWK variables initialized this way do not affect the **ARGC** variable.

ARGV - Array of arguments (NAWK/GAWK)

The **ARGV** array is the list of arguments (or files) passed as command line arguments.

ARGIND - Argument Index (GAWK only)

The **ARGIND** specifies the current index into the **ARGV** array, and therefore **ARGV[ARGIND]** is always the current filename. Therefore

```
FILENAME == ARGV[ARGIND]
```

is always true. It can be modified, allowing you to skip over files, etc.

FNR (NAWK/GAWK)

The **FNR** variable contains the number of lines read, but is reset for each file read. The **NR** variable accumulates for all files read. Therefore if you execute an awk script with two files as arguments, with each containing 10 lines:

```
nawk '{print NR}' file file2
nawk '{print FNR}' file file2
```

the first program would print the numbers 1 through 20, while the second would print the numbers 1 through 10 twice, once for each file.

OFMT (NAWK/GAWK)

The **OFMT** variable Specifies the default format for numbers. The default value is "%.6g."

RSTART, RLENGTH and match (NAWK/GAWK)

I've already mentioned the **RSTART** and **RLENGTH** variables. After the *match*() function is called, these variables contain the location in the string of the search pattern. **RLENGTH** contains the length of this match.

SUBSEP - Multi-dimensional array separator (NAWK/GAWK)

Earlier I described how you can construct multi-dimensional arrays in AWK. These are constructed by concatenating two indexes together with a special character between them. If I use an ampersand as the special character, I can access the value at location X, Y by the reference

```
array[ X "&" Y ]
```

NAWK (and GAWK) has this feature built in. That is, you can specify the array element

```
array[X,Y]
```

It automatically constructs the string, placing a special character between the indexes. This character is the non-printing character "034." You can control the value of this character, to make sure your strings do not contain the same character.

ENVIRON - environment variables (GAWK only)

The **ENVIRON** array contains the environment variables the current process. You can print your current search path using

```
print ENVIRON["PATH"]
```

IGNORECASE (GAWK only)

The **IGNORECASE** variable is normally zero. If you set it to non-zero, then all pattern matches ignore case. Therefore the following is equivalent to "grep -i match:"

```
BEGIN {IGNORECASE=1;}
```

```
/match/ {print}
```

CONVFMT - conversion format (GAWK only)

The **CONVFMT** variable is used to specify the format when converting a number to a string. The default value is "%.6g." One way to truncate integers is to convert an integer to a string, and convert the string to an integer - modifying the actions with **CONVFMT**:

```
a = 12;
b = a "";
CONVFMT = "%2.2f";
c = a "";
```

Variables **b** and **c** are both strings, but the first one will have the value "12.00" while the second will have the value "12."

ERRNO - system errors (GAWK only)

The **ERRNO** variable describes the error, as a string, after a call to the **getline** command fails.

FIELDWIDTHS - fixed width fields (GAWK only)

The **FIELDWIDTHS** variable is used when processing fixed width input. If you wanted to read a file that had 3 columns of data; the first one is 5 characters wide, the second 4, and the third 7, you could use *substr* to split the line apart. The technique, using **FIELDWIDTHS**, would be:

```
BEGIN {FIELDWIDTHS="5 4 7";}
{ printf("The three fields are %s %s %s\n", $1, $2, $3);}
```

AWK, NAWK, GAWK, or PERL

This concludes my series of tutorials on AWK and the NAWK and GAWK variants. I originally intended to avoid extensive coverage of NAWK and GAWK. I am a PERL user myself, and still believe PERL is worth learning, yet I don't intend to cover this very complex language. Understanding AWK is very useful when you start to learn PERL, and a part of me felt that teaching you extensive NAWK and GAWK features might encourage you to bypass PERL.

I found myself going into more depth than I planned, and I hope you found this useful. I found out a lot myself, especially when I discussed topics not covered in other books. Which reminds me of some closing advice: if you don't understand how something works, experiment and see what you can discover. Good luck, and happy AWKing...

REFERENCES & RECOMMENDATIONS:

Some other Unix shell tutorials can be found at:

http://www.grymoire.com/Unix

Other shell tutorials can be found at

http://www.shelldorado.com/links/index.html#tutorials

http://cfaj.freeshell.org/shell/

From:

http://www.grymoire.com/Unix/Awk.html