simple awk tutorial

why awk?

awk is small, fast, and simple, unlike, say, perl. awk also has a clean comprehensible C-like input language, unlike, say, perl. And while it can't do everything you can do in perl, it can do most things that are actually text processing, and it's much easier to work with.

what do you do?

In its simplest usage awk is meant for processing column-oriented text data, such as tables, presented to it on standard input. The variables $1, $2, and so forth are the contents of the first, second, etc. column of the current input line. For example, to print the second column of a file, you might use the following simple awk script:

```
        awk < file '{ print $2 }'
```

This means "on every line, print the second field".

To print the second and third columns, you might use

```
        awk < file '{ print $2, $3 }'
```

Input separator

By default awk splits input lines into fields based on whitespace, that is, spaces and tabs. You can change this by using the -F option to awk and supplying another character. For instance, to print the home directories of all users on the system, you might do

```
        awk < /etc/passwd -F: '{ print $6 }'
```

since the password file has fields delimited by colons and the home directory is the 6th field.

Arithmetic

Awk is a weakly typed language; variables can be either strings or numbers, depending on how they're referenced. All numbers are floating-point. So to implement the fahrenheit-to-celsius calculator, you might write

```
        awk '{ print ($1-32)*(5/9) }'
```

which will convert fahrenheit temperatures provided on standard input to celsius until it gets an end-of-file.

The selection of operators is basically the same as in C, although some of C's wilder constructs do not work. String concatenation is accomplished simply by writing two string expressions next to each other. '+' is always addition. Thus

```
        echo 5 4 | awk '{ print $1 + $2 }'
```

prints 9, while

```
        echo 5 4 | awk '{ print $1 $2 }'
```

prints 54. Note that

```
echo 5 4 | awk '{ print $1, $2 }'
```

prints "5 4".

Variables

awk has some built-in variables that are automatically set; $1 and so on are examples of these. The other builtin variables that are useful for beginners are generally NF, which holds the number of fields in the current input line ($NF gives the last field), and $0, which holds the entire current input line.

You can make your own variables, with whatever names you like (except for reserved words in the awk language) just by using them. You do not have to declare variables. Variables that haven't been explicitly set to anything have the value "" as strings and 0 as numbers.

For example, the following code prints the average of all the numbers on each line:

```
awk '{ tot=0; for (i=1; i<=NF; i++) tot += $i; print    tot/NF; }'
```

Note the use of $i to retrieve the i'th variable, and the for loop, which works like in C. The reason `tot` is explicitly initialized at the beginning is that this code is run for every input line, and when starting work on the second line, tot will have the total value from the first line.

Blocks

It might seem silly to do that. Probably, you have only one set of numbers to add up. Why not put each one on its own line? In order to do this you need to be able to print the results when you're done. The way you do this is like this:

```
awk '{ tot += $1; n += 1; }  END { print tot/n; }'
```

Note the use of two different block statements. The second one has `END` in front of it; this means to run the block once after all input has been processed. In fact, in general, you can put all kinds of things in front of a block, and the block will only run if they're satisfied. That is, you can say

```
awk ' $1==0 { print $2 }'
```

which will print the second column for lines of input where the first column is 0. You can also supply regular expressions to match the whole line against:

```
awk ' /^test/ { print $2 }'
```

If you put no expression, the block is run on every line of input. If multiple blocks have conditions that are true, they are **all** run. There is no particularly clean way I know of to get it to run exactly one of a bunch of possible blocks of code.

The block conditions `BEGIN` and `END` are special and are run before processing any input, and after processing all input, respectively.

Other language constructs

As hinted at above, awk supports loop and conditional statements like in C, that is, for, while, do/while, if, and if/else.

printf

awk includes a `printf` statement that works essentially like C printf. This can be used when you want to format output neatly or combine things onto one line in more complex ways (`print` implicitly adds a newline; `printf` doesn't.)

Here's how you strip the first column off:

```
        awk '{ for (i=2; i<=NF; i++) printf "%s ", $i; printf "\n"; }'
```

Note the use of NF to iterate over all the fields and the use of printf to place newlines explicitly.

Combining awk with other tools

The strength of shell scripting is the ability to combine lots of tools together. Some things are most readily done with successive awk processes pipelined together. awk is also often combined with the sed utility, which does regular expression matching and substitution. You can actually do most common sed operations in awk, but it's usually more convenient to use sed.

In combination with sed, sort, and some other useful shell utilities like paste, awk is quite satisfactory for a lot of numerical data processing, as well as the maintenance of simple databases kept in column-tabular formats.

awk is also extremely useful in a certain style of makefile writing for generating pieces of makefile to include.

Do not script in csh. Use sh, or if you must, ksh.

More stuff

This introduction intentionally covers only the absolute basics of awk. awk can do lots of other useful and powerful things. The manual page for GNU awk (gawk) is a reasonably good reference for looking things up once you know the basics.

The only thing seriously lacking in awk that I've yet run into is that there's no way to tell awk not to do buffering on its input and output, which makes it useless for assorted interactive or network-oriented applications it would otherwise be ideally suited for.