AWK is a standard tool on every POSIX-compliant UNIX system. It's like flex/lex, from the command-line, perfect for text-processing tasks and other scripting needs. It has a C-like syntax, but without mandatory semicolons (although, you should use them anyway, because they are required when you're writing one-liners, something AWK excels at), manual memory management, or static typing. It excels at text processing. You can call to it from a shell script, or you can use it as a stand-alone scripting language.

Why use AWK instead of Perl? Readability. AWK is easier to read than Perl. For simple text-processing scripts, particularly ones that read files line by line and split on delimiters, AWK is probably the right tool for the job.

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

# Comments are like this


# AWK programs consist of a collection of patterns and actions.
pattern1 { action; } # just like lex
pattern2 { action; }

# There is an implied loop and AWK automatically reads and parses each
# record of each file supplied. Each record is split by the FS delimiter,
# which defaults to white-space (multiple spaces,tabs count as one)
# You can assign FS either on the command line (-F C) or in your BEGIN
# pattern

# One of the special patterns is BEGIN. The BEGIN pattern is true
# BEFORE any of the files are read. The END pattern is true after
# an End-of-file from the last file (or standard-in if no files specified)
# There is also an output field separator (OFS) that you can assign, which
# defaults to a single space

BEGIN {

    # BEGIN will run at the beginning of the program. It's where you put all
    # the preliminary set-up code, before you process any text files. If you
    # have no text files, then think of BEGIN as the main entry point.

    # Variables are global. Just set them or use them, no need to declare.
    count = 0;

    # Operators just like in C and friends
    a = count + 1;
    b = count - 1;
    c = count * 1;
    d = count / 1; # integer division
    e = count % 1; # modulus
    f = count ^ 1; # exponentiation

    a += 1;
    b -= 1;
```

```
c *= 1;
d /= 1;
e %= 1;
f ^= 1;

# Incrementing and decrementing by one
a++;
b--;

# As a prefix operator, it returns the incremented value
++a;
--b;

# Notice, also, no punctuation such as semicolons to terminate statements

# Control statements
if (count == 0)
    print "Starting with count of 0";
else
    print "Huh?";

# Or you could use the ternary operator
print (count == 0) ? "Starting with count of 0" : "Huh?";

# Blocks consisting of multiple lines use braces
while (a < 10) {
    print "String concatenation is done" " with a series" " of"
        " space-separated strings";
    print a;

    a++;
}

for (i = 0; i < 10; i++)
    print "Good ol' for loop";

# As for comparisons, they're the standards:
# a < b   # Less than
# a <= b  # Less than or equal
# a != b  # Not equal
# a == b  # Equal
# a > b   # Greater than
# a >= b  # Greater than or equal

# Logical operators as well
# a && b  # AND
# a || b  # OR

# In addition, there's the super useful regular expression match
if ("foo" ~ "^fo+$")
    print "Fooey!";
    if ("boo" !~ "^fo+$")
        print "Boo!";

    # Arrays
    arr[0] = "foo";
    arr[1] = "bar";

    # You can also initialize an array with the built-in function split()

    n = split("foo:bar:baz", arr, ":");

    # You also have associative arrays (actually, they're all associative arrays)
    assoc["foo"] = "bar";
    assoc["bar"] = "baz";

    # And multi-dimensional arrays, with some limitations I won't mention here
    multidim[0,0] = "foo";
    multidim[0,1] = "bar";
    multidim[1,0] = "baz";
    multidim[1,1] = "boo";

    # You can test for array membership
    if ("foo" in assoc)
        print "Fooey!";

    # You can also use the 'in' operator to traverse the keys of an array
    for (key in assoc)
        print assoc[key];

    # The command line is in a special array called ARGV
    for (argnum in ARGV)
        print ARGV[argnum];

    # You can remove elements of an array
    # This is particularly useful to prevent AWK from assuming the arguments
    # are files for it to process
    delete ARGV[1];

    # The number of command line arguments is in a variable called ARGC
    print ARGC;

    # AWK has several built-in functions. They fall into three categories. I'll
    # demonstrate each of them in their own functions, defined later.

    return_value = arithmetic_functions(a, b, c);
    string_functions();
    io_functions();
}

# Here's how you define a function
function arithmetic_functions(a, b, c,     d) {
    # Probably the most annoying part of AWK is that there are no local
    # variables. Everything is global. For short scripts, this is fine, even
    # useful, but for longer scripts, this can be a problem.

    # There is a work-around (ahem, hack). Function arguments are local to the
    # function, and AWK allows you to define more function arguments than it
    # needs. So just stick local variable in the function declaration, like I
    # did above. As a convention, stick in some extra whitespace to distinguish
    # between actual function parameters and local variables. In this example,
    # a, b, and c are actual parameters, while d is merely a local variable.

    # Now, to demonstrate the arithmetic functions

    # Most AWK implementations have some standard trig functions
    d = sin(a);
    d = cos(a);
    d = atan2(b, a); # arc tangent of b / a

    # And logarithmic stuff
    d = exp(a);
    d = log(a);

    # Square root
    d = sqrt(a);

    # Truncate floating point to integer
    d = int(5.34); # d => 5

    # Random numbers
    srand(); # Supply a seed as an argument. By default, it uses the time of day
    d = rand(); # Random number between 0 and 1.

    # Here's how to return a value
    return d;
}

function string_functions(    localvar, arr) {

    # AWK, being a string-processing language, has several string-related
    # functions, many of which rely heavily on regular expressions.

    # Search and replace, first instance (sub) or all instances (gsub)
    # Both return number of matches replaced
    localvar = "fooooobar";
    sub("fo+", "Meet me at the ", localvar); # localvar => "Meet me at the bar"
    gsub("e", ".", localvar); # localvar => "m..t m. at th. bar"

    # Search for a string that matches a regular expression
    # index() does the same thing, but doesn't allow a regular expression
    match(localvar, "t"); # => 4, since the 't' is the fourth character
    # Split on a delimiter
    n = split("foo-bar-baz", arr, "-"); # a[1] = "foo"; a[2] = "bar"; a[3] = "baz"; n =

    # Other useful stuff
    sprintf("%s %d %d %d", "Testing", 1, 2, 3); # => "Testing 1 2 3"
    substr("foobar", 2, 3); # => "oob"
    substr("foobar", 4); # => "bar"
    length("foo"); # => 3
    tolower("FOO"); # => "foo"
    toupper("foo"); # => "FOO"
}

function io_functions(    localvar) {

    # You've already seen print
    print "Hello world";

    # There's also printf
    printf("%s %d %d %d\n", "Testing", 1, 2, 3);

    # AWK doesn't have file handles, per se. It will automatically open a file
    # handle for you when you use something that needs one. The string you used
    # for this can be treated as a file handle, for purposes of I/O. This makes
    # it feel sort of like shell scripting, but to get the same output, the string
    # must match exactly, so use a variable:

    outfile = "/tmp/foobar.txt";

    print "foobar" > outfile;

    # Now the string outfile is a file handle. You can close it:
    close(outfile);

    # Here's how you run something in the shell
    system("echo foobar"); # => prints foobar

    # Reads a line from standard input and stores in localvar
    getline localvar;

    # Reads a line from a pipe (again, use a string so you close it properly)
    cmd = "echo foobar";
    cmd | getline localvar; # localvar => "foobar"
    close(cmd);

    # Reads a line from a file and stores in localvar
    infile = "/tmp/foobar.txt";
    getline localvar < infile;
    close(infile);
}

# As I said at the beginning, AWK programs consist of a collection of patterns
# and actions. You've already seen the BEGIN pattern. Other
# patterns are used only if you're processing lines from files or standard
# input.
#
# When you pass arguments to AWK, they are treated as file names to process.
# It will process them all, in order. Think of it like an implicit for loop,
# iterating over the lines in these files. these patterns and actions are like
# switch statements inside the loop.

/^fo+bar$/ {

    # This action will execute for every line that matches the regular
    # expression, /^fo+bar$/, and will be skipped for any line that fails to
    # match it. Let's just print the line:

    print;

    # Whoa, no argument! That's because print has a default argument: $0.
    # $0 is the name of the current line being processed. It is created
    # automatically for you.

    # You can probably guess there are other $ variables. Every line is
    # implicitly split before every action is called, much like the shell
    # does. And, like the shell, each field can be access with a dollar sign

    # This will print the second and fourth fields in the line
    print $2, $4;

    # AWK automatically defines many other variables to help you inspect and
    # process each line. The most important one is NF

    # Prints the number of fields on this line
    print NF;

    # Print the last field on this line
    print $NF;
}

# Every pattern is actually a true/false test. The regular expression in the
# last pattern is also a true/false test, but part of it was hidden. If you
# don't give it a string to test, it will assume $0, the line that it's
# currently processing. Thus, the complete version of it is this:

$0 ~ /^fo+bar$/ {
    print "Equivalent to the last pattern";
}

a > 0 {
    # This will execute once for each line, as long as a is positive
}
```

```
# You get the idea. Processing text files, reading in a line at a time, and
# doing something with it, particularly splitting on a delimiter, is so common
# in UNIX that AWK is a scripting language that does all of it for you, without
# you needing to ask. All you have to do is write the patterns and actions
# based on what you expect of the input, and what you want to do with it.

# Here's a quick example of a simple script, the sort of thing AWK is perfect
# for. It will read a name from standard input and then will print the average
# age of everyone with that first name. Let's say you supply as an argument the
# name of a this data file:
#
# Bob Jones 32
# Jane Doe 22
# Steve Stevens 83
# Bob Smith 29
# Bob Barker 72
#
# Here's the script:

BEGIN {

    # First, ask the user for the name
    print "What name would you like the average age for?";

    # Get a line from standard input, not from files on the command line
    getline name < "/dev/stdin";
}

# Now, match every line whose first field is the given name
$1 == name {

    # Inside here, we have access to a number of useful variables, already
    # pre-loaded for us:
    # $0 is the entire line
    # $3 is the third field, the age, which is what we're interested in here
    # NF is the number of fields, which should be 3
    # NR is the number of records (lines) seen so far
    # FILENAME is the name of the file being processed
    # FS is the field separator being used, which is " " here
    # ...etc. There are plenty more, documented in the man page.

    # Keep track of a running total and how many lines matched
    sum += $3;
    nlines++;
}

# Another special pattern is called END. It will run after processing all the
# text files. Unlike BEGIN, it will only run if you've given it input to
# process. It will run after all the files have been read and processed
# according to the rules and actions you've provided. The purpose of it is
# usually to output some kind of final report, or do something with the
# aggregate of the data you've accumulated over the course of the script.

END {
    if (nlines)
        print "The average age for " name " is " sum / nlines;
}
```

Further Reading:

- Awk tutorial
- Awk man page
- The GNU Awk User's Guide GNU Awk is found on most Linux systems.
- AWK one-liner collection
- Awk alpinelinux wiki a technical summary and list of "gotchas" (places where different implementations may behave in different or unexpected ways).
- basic libraries for awk