Gnu Awk - Part 4 (HPR Show 2163)

Dave Morriss

Contents

Introduction

This is the fourth episode of the series that b-yeezi and I are doing. These shows are now collected under the series title "Learning Awk".

Recap of the last episode

Logical Operators

We have seen the operators '&&' (*and*) and '||' (*or*). These are also called *Boolean Operators*. There is also one more operator '!' (*not*) which we haven't

yet encountered. These operators allow the construction of *Boolean expressions* which may be quite complex.

If you are used to programming you will expect these operators to have a precedence, just like operators in arithmetic do. We will deal with this subject in more detail later since it is relevant not only in patterns but also in other parts of an Awk program.

The *next* statement

We saw this statement in the last episode and learned that it causes the processing of the current input record to stop. No more patterns are tested against this record and no more actions in the current rule are executed. Note that "*next*" is a statement like "*print*", and can only occur in the action part of a rule. It is also not permitted in *BEGIN* or *END* rules (more of which anon).

The *BEGIN* and *END* rules

The *BEGIN* and *END* elements are special *patterns*, which in conjunction with *actions* enclosed in curly brackets make up *rules* in the same sense that the '*pattern {action}*' sequences we have seen so far are rules. As we saw in the last episode, *BEGIN* rules are run before the main '*pattern {action}*' rules are processed and the input file is (or files are) read, whereas *END* rules run after the input files have been processed.

It is permitted to write more than one *BEGIN* rule and more than one *END* rule. These are just concatenated together in the order they are encountered by Awk.

Awk will complain if either *BEGIN* or *END* is not followed by an *action* since this is meaningless.

Variables, arrays, loops, etc

Learning a programming language is never a linear process, and sometimes reference is made to new features that have not yet been explained. A number of new features were mentioned in passing in the last episode, and we will look at these in more detail in this episode.

Explaining variables

We saw the built-in variables like *NR* and *NF* earlier in the series, and you saw in the last episode that you can create your own variables too. A variable, as in other languages, is simply a named storage area that can hold a value. The

name must consist of letters, digits or the underscore. It may not start with a digit, and there is a difference between upper case and lower case letters ('`sum`', '`Sum`' and '`SUM`' are different variables). Such simple variables which can hold a single value are also called *scalars*.

A variable in Awk may contain a numeric value or a string. Awk deals with the conversion of one to another as appropriate (though sometimes it needs help).

In Awk, unlike many other languages, it is not necessary to initialise variables before using them. All variables start as an empty string which is converted to zero as appropriate.

Variable assignment

Variables are set to values using *assignment* such as:

```
count = 3
```

As you saw in the last episode there are many types of assignment, for example:

```
used += $3
```

This means *increment the contents of variable '`used`' by the contents of field 3.* (There is an assumption here that field 3 contains a numeric value, of course.)

It's a shorthand version of:

```
used = used + $3
```

This means *add the contents of '`used`' to the contents of field 3 and save the result back in '`used`'.*

The first time the variable is incremented its contents are taken to be zero. This is normally bad practice in older and stricter compiled languages, but Awk is more forgiving.

Since we have now started to look at writing arithmetic expressions it is probably a good idea to review what the arithmetic operators are in Awk.

Arithmetic operators

It is important to note that *all numbers in Awk are floating point numbers.* This fact can catch you out in some edge cases, which we will try to highlight as the series progresses.

This list is based on the one from the GNU Awk User's Guide. The operators are listed in order of their precedence, highest to lowest.

x ^ y Exponentiation; x raised to the y power. '2 ^ 3' has the value eight. There is a '`**`' operator but is not standard, and therefore not portable, and will not be used here.

- x Negation

+ x Unary plus; this can be used to force Awk to convert a string to a number.

x * y Multiplication

x / y Division; because all numbers in awk are floating-point numbers, the result is not rounded to an integer – thus '3 / 4' has the value 0.75, where in Bash '`echo $((3/4))`' returns 0.

x % y Remainder after x is divided by y. So '3 % 4' is 3, '5 % 2' is 1, and so on

x + y Addition.

x - y Subtraction.

Assignment operators

As you have seen arithmetic assignment operators (like +=) exist in Awk. These are a shorthand form of more verbose assignments. The following table lists these assignment operators (modified from the GNU Awk User's Guide):

Operator	Effect
variable += increment	Add increment to the value of variable.
variable -= decrement	Subtract decrement from the value of variable.
variable *= coefficient	Multiply the value of variable by coefficient.
variable /= divisor	Divide the value of variable by divisor.
variable %= modulus	Set variable to its remainder by modulus.
variable ^= power	Raise variable to the power power.

Examples

See the associated Awk script called `arithmetic_assignment_operators.awk`:

```
BEGIN{
    x = 42; print "x is",x
    x += 1; print "x += 1 is",x
    x -= 1; print "x -= 1 is",x
    x *= 2; print "x *= 2 is",x
    x /= 2; print "x /= 2 is",x
    x %= 5; print "x %= 5 is",x
    x ^= 4; print "x ^= 4 is",x
}
```

Note that everything here is in a *BEGIN* rule because we don't want to process a file, just run a little Awk program. Note also that semicolons are needed as statement separators when there are multiple statements on a line, but not otherwise.

When run it produces the following output:

```
$ awk -f arithmetic_assignment_operators.awk
x is 42
x += 1 is 43
x -= 1 is 42
x *= 2 is 84
x /= 2 is 42
x %= 5 is 2
x ^= 4 is 16
```

Type conversion

As mentioned earlier, a variable in Awk may contain a numeric value or a string, at any point in time. When converting from a number to a string, then the conversion simply consists of a string version of the number. Converting from a string to a number requires the string to begin with a valid digit sequence.

```
$ awk 'BEGIN{s="9gag.com"; x=s+1; print x}'
10
```

If the string contains no valid number at the start then it is converted to zero.

Awk will convert integer numbers (42), and floating point numbers (4.2), as well as exponential numbers (1E3):

```
$ awk 'BEGIN{ printf "%g %g %g\n","42","4.2","1E3" }'
42 4.2 1000
```

(Note: the '*g*' format-control letter is for printing general numbers)

Increment and decrement operators

In the last episode we saw the use of these operators which increment or decrement the value of a variable by one. There are similar operators in Bash, and these were covered in HPR episode 1951.

The formal definition of these operators is:

++variable Increment variable, returning the new value as the value of the expression.

variable++ Increment variable, returning the old value of variable as the value of the expression.

--variable Decrement variable, returning the new value as the value of the expression. (This expression is like '++variable', but instead of adding, it subtracts.)

variable-- Decrement variable, returning the old value of variable as the value of the expression. (This expression is like 'variable++', but instead of adding, it subtracts.)

We will look at some examples of the use of these operators a little later.

Arrays

As well as the simple (*scalar*) variables we have seen, Awk also provides one-dimensional arrays[1]. These arrays are *associative* (also known as *hashes*).

An array has a name conforming to the rules for scalar variables mentioned earlier. Not surprisingly you cannot name an array the same as a simple variable.

An array is a means of storing multiple values, and these values are referenced by *index* values. Also, unlike most compiled languages, Awk's arrays can be of any length and can be added to at will. They can also be deleted from, but we'll deal with that later.

Given an array `a`, we might store a value in it thus:

```
a[1] = "HPR"
```

Here the array name is `a`, the index is `1` and the contents of `a[1]` is the string "HPR".

If you are familiar with arrays in other languages you might assume that the index `1` is numeric. In fact, in Awk it is converted to a string because all array indices are strings because Awk arrays are not contiguous but are *associative*. Such arrays are indexed by arbitrary string values, making a sort of *look-up table*.

Thus in an example in the last episode we saw:

```
NR != 1 {
    a[$2]++
}
```

Here the Awk script was being used to produce a frequency count of colours in our example file `file1.txt`. Field 2 in this file is the name of a colour, so the meaning of `a[$2]++` is:

> Index the array `a` by the (string) contents of field 2. If the element does not exist create it. Since Awk is very relaxed about initialisation, this array element will be taken to be zero on creation, and will then be incremented to 1. If the element already exists then its previous value will be incremented.

[1] Actually, standard Awk provides a way of treating such arrays as multi-dimensional, and GNU Awk (gawk) provides true *arrays of arrays*, but this is rather advanced and non-portable!

If you were able to look into the resulting array the end result would be:

Index	Contents
brown	2
purple	2
red	2
yellow	2
green	1

So, this shows that there is an array element: `a["brown"]`. Contained in this array element is the number 2 because the colour 'brown' was encountered twice.

Note that we also know that the expression `a[$2]++` achieves the same as the assignment `a[$2]+=1`.

Looping through arrays

In the last episode the concept of looping through an array to print it out was introduced. We saw:

```
for (b in a) {
    print b, a[b]
}
```

As is so often the case with learning to write scripts it is often useful to visit more advanced topics early on, even though the concepts behind them may not yet have have been properly established.

We have not yet examined looping and other statements in Awk, but since we want to be able to process entire arrays we need to look at this one now.

In brief, the '`for`' statement provides a way to repeat a given set of statements a number of times. We will look at this statement and the related '`while`' statement later in the series.

This variant of the '`for`' statement allows the processing of arrays. It consists of the following components:

```
for (variable in array)
    body
```

The expression '`(variable in array)`' results in all of the index values in the nominated array being provided, one at a time. While the loop runs the variable is set to successive index values and the body is executed.

The body can consist of a single statement or a group of statements. If a group is used, then curly braces must be used to enclose them.

The order in which array index values are provided is not defined – different Awk version will use different orders. There are extensions within GNU Awk (gawk) which can control this but we will leave this until much later.

So, dealing with our example from last episode, we can modify it as follows (with spelling concessions due to the trans-Atlantic nature of this series):

```
BEGIN {
    FS=","
    OFS=","
    print "color,count"
}
NR != 1 {
    count[$2]+=1
}
END {
    for (color in count)
        print color, count[color]
}
```

This Awk script is available as `color_count.awk`. The array has been renamed from '`a`' to '`count`' because it holds counts (frequencies) of the number of times a colour is encountered. The array is indexed by the names of colours in field 2. When we loop through the array in the *END* rule we use a variable '`color`' to store the latest index. Note that the unnecessary semicolons and curly braces have been removed (to demonstrate that they can be!).

Running the script produces the following output:

```
$ awk -f color_count.awk file1.csv
color,count
brown,2
purple,2
red,2
yellow,2
green,1
```

More built-in variables

In the last episode two more built-in (or predefined) variables were introduced. The first was *FS*, which we have encountered before, though not in such a form. The *FS* variable is set through the *-F* (or *–field-separator*) command-line option, so '`-F ","`' on the command line is the same as the statement `FS = ","` in an Awk script. As we saw, the statement form needs to be in a *BEGIN* rule to be set early enough in the script.

```
$ awk -F "," 'BEGIN{print "FS is",FS}'
FS is ,
```

Of course, *FS* controls the chosen field separator as has been explained earlier in the series.

In the last episode we also saw the *OFS* variable. This does not have a command-line equivalent. This variable, short for *Output Field Separator*, controls the format of the output record produced by the `print` statement. Normally it is set to a single space, so a print statement like the following separates its arguments with a single space:

```
$ awk 'BEGIN{print "Hello","World"}'
Hello World
```

Note that omitting the comma results in the following:

```
$ awk 'BEGIN{print "Hello" "World"}'
HelloWorld
```

This is because Awk has concatenated the two strings before handing them to the `print` statement.

The *OFS* variable can be a string if required:

```
$ awk 'BEGIN{OFS=" blurg "; print "Hello","World"}'
Hello blurg World
```

The contents of *OFS* only affects the behaviour of the `print` statement, not `printf`:

```
$ awk 'BEGIN{OFS="\t"; printf "%s %s\n","Hello","World"}'
Hello World
```

Here the first argument to the `printf` statement, the format string, specifies that two string arguments will be printed followed by a newline. The remaining arguments are the two strings. The contents of *OFS* have no effect on the output.

Summary

This episode covered:

- A recap of the last episode
- Variables: simple or *scalar* variables
- Assignment of values to variables
- Arithmetic operators used in arithmetic expressions
- Assignment operators
- Conversion of strings to numbers and vice versa
- Increment and decrement operators
- Variables: Awk's *associative* arrays (aka *hashes*)
- A brief peek at `for` loops used to scan arrays
- The built-in or predefined variables *FS* and *OFS*

Links

- *GNU Awk User's Guide*: https://www.gnu.org/software/gawk/manual/html_node/index.html
- Previous shows on HPR:
  - "*Gnu Awk - Part 1*": http://hackerpublicradio.org/eps.php?id=2114
  - "*Gnu Awk - Part 2*": http://hackerpublicradio.org/eps.php?id=2129
  - "*Gnu Awk - Part 3*": http://hackerpublicradio.org/eps.php?id=2143
- Arithmetic expansion in Bash "*Some additional Bash Tips*": http://hackerpublicradio.org/eps.php?id=1951
- Resources:
  - Demonstration of arithmetic assignment operators: http://hackerpublicradio.org/eps/hpr2163/arithmetic_assignment_operators.awk
  - Counting frequencies of particular colours: http://hackerpublicradio.org/eps/hpr2163/color_count.awk