Gnu Awk - Part 8 (HPR Show 2438)

Dave Morriss

Contents

Introduction

This is the eighth episode of the "Learning Awk" series that b-yeezi and I are doing.

Recap of the last episode

- The `while` loop: tests a condition and performs commands *while* the test returns true
- The `do while` loop: performs commands after the `do`, then tests afterwards, repeating the commands *while* the test is true.
- The `for` loop (type 1): initialises a variable, performs a test, and increments the variable all together, performing commands while the test is true.
- The `for` loop (type 2): sets a variable to successive indices of an array, preforming a collection of commands for each index.

These types of loops were demonstrated by examples in the last episode.

Note that the example for '`do while`' was an infinite loop (perhaps as a test of the alertness of the audience!):

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

    i=2;
    do {
        print "The square of ", i, " is ", i*i;
        i = i + 1
    }
    while (i != 2)

exit;
}
```

The condition in the `while` is always true:

```
The square of  2  is  4
The square of  3  is  9
The square of  4  is  16
The square of  5  is  25
The square of  6  is  36
The square of  7  is  49
The square of  8  is  64
The square of  9  is  81
The square of  10  is  100
...
The square of  1269630  is  1611960336900
The square of  1269631  is  1611962876161
The square of  1269632  is  1611965415424
The square of  1269633  is  1611967954689
The square of  1269634  is  1611970493956
...
```

The variable `i` is set to 2, the `print` is executed, then `i` is set to 3. The test "`i != 2`" is true and will be *ad infinitum*.

Some more statements

We will come back to loops later in this episode, but first this seems like a good point to describe another statement: the `switch` statement.

The `switch` statement

This is specific to `gawk`, and can be disabled if non-GNU `awk`-compatibility is required. The `switch` statement in `gawk` is very similar to the one in `C` and many other languages.

The layout of the `switch` statement is as follows:

`switch` (*expression*) { `case` *value*: *case-body* `default`: *default-body*
}

The '`expression`' part is an expression, which returns a numeric or string result. The '`value`' part after the `case` is a numeric or string constant or a regular expression.

The `expression` is evaluated and the result matched against the case `values` in turn. If there is a match the `case-body` statements are executed. If there is no match the `default-body` statements are executed.

The following example is included as one of the files associated with this show, called `switch_example.awk`:

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

#
# Example of the use of 'switch' in GNU Awk.
#
# Should be run against the data file 'file1.txt' included with the second
# show in the series: http://hackerpublicradio.org/eps/hpr2129/file1.txt
#
NR > 1 {
    printf "The %s is classified as: ",$1

    switch ($1) {
        case "apple":
            print "a fruit, pome"
            break
        case "banana":
        case "grape":
        case "kiwi":
            print "a fruit, berry"
            break
        case "strawberry":
            print "not a true fruit, pseudocarp"
            break
        case "plum":
            print "a fruit, drupe"
            break
        case "pineapple":
            print "a fruit, fused berries (syncarp)"
            break
        case "potato":
            print "a vegetable, tuber"
            break
        default:
            print "[unclassified]"
    }
}
```

The result of running this script against the “fruit” file presented in show 2129 is the following (`switch_example.out`):

```
The apple is classified as: a fruit, pome
The banana is classified as: a fruit, berry
The strawberry is classified as: not a true fruit, pseudocarp
The grape is classified as: a fruit, berry
The apple is classified as: a fruit, pome
The plum is classified as: a fruit, drupe
The kiwi is classified as: a fruit, berry
The potato is classified as: a vegetable, tuber
The pineapple is classified as: a fruit, fused berries (syncarp)
```

What this simple example does is:

- It ignores the first line of the file (a header)
- It prints the first field (the name of a fruit - mostly) in the string “The %s is classified as:”. There is no newline so whatever is printed next is appended to the line.
- It uses the first field in a `switch` statement. Each `case` is an exact match with the contents of the field. If there is a match a `print` statement is used to print out the Botanical classification. If there are no matches then the `default` instance would print “[unclassified]”, but that doesn’t happen in this example.
- All `print` statements are followed by `break`. If this hadn’t been there the next `case` would be executed and so forth. This can be desirable in some instances. See the next section for a discussion of `break`.
- Note that banana, grape and kiwi are all Botanically classified as a berry, so there are three `case` parts associated with one `print`.

The `break` statement

This statement is mainly for “breaking out of” a `for`, `while` or `do-while` loop, though, as we have seen it can interrupt the flow of execution in a `switch` statement also. Outside of these statements `break` has no effect.

In a loop a `break` statement is often used where it’s not possible to determine the number of iterations of the loop beforehand. Invoking `break` completely terminates the enclosing loop (relevant when there are nested loops, or loops within loops).

The following example (available for download as `divisor.awk`) is from the Gnu Awk manual and shows a method of finding the smallest divisor:

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

# find smallest divisor of num
{
    num = $1

    #
    # Make an infinite loop using the for loop
    #
    for (divisor = 2; ; divisor++) {
        #
        # If the number is divisible by 'divisor' then we're done
        #
        if (num % divisor == 0) {
            printf "Smallest divisor of %d is %d\n", num, divisor
            break
        }

        #
        # If the value of 'divisor' has got too large the number has no
        # divisors and is therefore a prime number
        #
        if (divisor * divisor > num) {
            printf "%d is prime\n", num
            break
        }
    }
}
```

I have added some comments to this script to (hopefully) make it clearer.

Running this in a pipeline with the number presented to it as shown results in the following type of output (`divisor.out`):

```
$ echo 67 | ./divisor.awk
67 is prime
$ echo 69 | ./divisor.awk
Smallest divisor of 69 is 3
```

The `continue` statement

This is similar to `break` in that it is used a `for`, `while` or `do-while` loop. It is **not** relevant in `switch` statements however.

Invoking `continue` skips the rest of the enclosing loop and begins the next cycle.

The following example (available for download as `continue_example.awk`) is from the Gnu Awk manual and demonstrates a possible use of `continue`:

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

#
# Loop, printing numbers from 0-20, except for 5
# (From the GNU Awk User's Guide)
#
BEGIN {
    for (x = 0; x <= 20; x++) {
        if (x == 5)
            continue
        printf "%d ", x
    }
    print ""
}
```

The `next` statement

This statement is not related to loops in the same way as `break` and `continue` but to the main record processing cycle of Awk. The `next` statement causes Awk to stop processing the current input record and go on to the next one.

As we know from earlier episodes in this series, Awk reads records from its input stream and applies rules to them. The `next` statement stops the execution of further rules for the current record, and moves on to the next one.

The following example (available for download as `next_example.awk`) is demonstrates a use of `next`:

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

#
# Ignore the header
#
NR == 1 { next }

#
# If field 2 (colour) is less than 6 characters then save it with its line
# number and skip it
#
length($2) < 6 {
    skip[NR] = $0
    next
}

#
# It's not the header and the colour name is > 6 characters, so print the line
#
{
    print
}

#
# At the end show what was skipped
#
END {
    printf "\nSkipped:\n"
    for (n in skip)
        print n": "skip[n]
}
```

- The script uses **next** in the first rule to avoid the first line of the file (a header).
- The second rule skips lines where the colour name is less than 6 characters long, but it also saves that line in an array called **skip** using the line number as the key (index).
- The third rule prints anything it sees, but it will not be invoked if either rule 1 or rule 2 cause it to be skipped.
- Finally, and `END` rule prints the contents of the array.

Running this with the file we have used many times before, `file1.txt`, results in the following output (`next_example.out`):

```
$ next_example.awk file1.txt
banana     yellow 6
grape      purple 10
plum       purple 2
pineapple  yellow 5

Skipped:
2: apple     red    4
4: strawberry red   3
6: apple     green  8
8: kiwi      brown  4
9: potato    brown  9
```

Links

- *GNU Awk User's Guide*

- Previous shows in this series on HPR:
  - "*Gnu Awk - Part 1*" - episode 2114
  - "*Gnu Awk - Part 2*" - episode 2129
  - "*Gnu Awk - Part 3*" - episode 2143
  - "*Gnu Awk - Part 4*" - episode 2163
  - "*Gnu Awk - Part 5*" - episode 2184
  - "*Gnu Awk - Part 6*" - episode 2238
  - "*Gnu Awk - Part 7*" - episode 2330
- Resources:
  - ePub version of these notes
  - PDF version of these notes
  - Demonstration of the `switch` statement:
    - Script: switch_example.awk
    - Output: switch_example.out
  - Demonstration of the `break` statement:
    - Script: divisor.awk
    - Output: divisor.out
  - Demonstration of the `continue` statement:
    - Script: continue_example.awk
  - Demonstration of the `next` statement:
    - Script: next_example.awk
    - Output: next_example.out