awk exercise

April 25, 2013

1 Introduction

awkis in between bash and Perl when it comes to complexity and functionality. When something is too difficult to achieve in bash and too simple to write a program in Perl, awkis ideal. Most often awkis used as one liners, but as in bash it's possible to save awkcode in a file for later use. awkis used for parsing text files, like sequence files. All programming snippets should be tried out on your computer.

2 Begin

We start with one liners directly from the terminal. "Hello world!" is written like:

```
awk ' BEGIN {print "Hello world"} '
```

The keyword BEGIN indicates that before anything else is done, the command specified in the curly braces should be executed. There is also an END keyword. The command or commands that follow are executed then everything else has bin processed. Strings have to be quoted by double apostrophes while numbers could be typed unquoted (like in Perl or C++). If we would do some arithmetics, it could look like Like:

```
awk ' BEGIN {print 4+5} '
awk ' BEGIN {print sin(3.14/6)} '
```

awk uses radians and the above corresponds to 60°. There is no built in mathematical constants in awkso if we would like to use a more precise value of π we could use:

```
awk ' BEGIN {print sin(atan2(0, -1)/6)} ' # arctangent of 0/-1
```

awkof course make use of variables:

```
awk ' BEGIN {a=2+3; b=4/8; print a, b} '
```

As in Perl or C++, command lines are separated by semicolons, so we have three commands here. Like in Perl print can take many arguments separated by commas. When assigning a value to a variable awkdoesn't use a $sign in front of the variable name. Also unlike Perl an output separator is inserted between the argument values. We can change this to any string:

```
awk ' BEGIN {OFS="xxxxx"; a=2+3; b=4/8; print a, b} '
awk ' BEGIN {OFS="\n"; a=2+3; b=4/8; print a, b} '
```

OFS stands for output field separator. awkcomes with a few built in variables which will be specified later.

3 Parsing

As mentioned, awkis primarily used for parsing text files. The general syntax is:

```
awk ' condition { action } ' textFile
```

In this case the file textFile is read line by line since the default input separator is newline. The condition is often a pattern that should or should not be matched in the input line. Try these commands on the regions.fas file:

```
awk ' /^>/ {print} ' regions.fas
```

This prints out all the id lines in a fasta file. Here we use the regular expression $\hat{>}$ as pattern which means lines starting ($\hat{()}$ with larger than ($>$) are matched and evaluated as true. Since the statement is true the code in curly braces is executed. Note that awkby default prints the entire lines if no argument is given to print. Each line (record) is saved in a variable called $0 which is overwritten for each new line. So this is equivalent with the above command:

```
awk ' /^>/ {print $0} ' regions.fas
```

If we only want to print the id, but not the length, the command is:

```
awk ' /^>/ {print $1} ' regions.fas
```

The input field separator is by default white space (spaces and tabs). Each line is split on white space and each field is stored in $1, $2 and so on. It is possible to use different pattern and associate them with different actions. Let's say that we want to count the number of sequences from regions 1 (FPV3KKJ**01**) and 2 (FPV3KKJ**02**):

```
awk ' /^>FPV3KKJ01/ {countRegion1++} /^>FPV3KKJ02/ \\
 {countRegion2++} END {print countRegion1 countRegion2} ' \\
 regions.fas
```

Here we introduce the `END` keyword. The commands in curly braces following `END` wioo, as expected, be executed when all previous commands have run. The `END`, like `BEGIN` will only execute once. To make a more readable output we could have written:

```
awk ' /^>FPV3KKJ01/ {countRegion1++} /^>FPV3KKJ02/ \\
 {countRegion2++} END {print "Region 1: "countRegion1"\n \\
 "Region 2:", countRegion2} ' regions.fas
```

If we would like to manipulate the output strings we use the commands `sub` and `gsub`. Let's say we want to output a list of ids (without the larger than and the lengths:

```
awk ' /^>/ {sub(">","",$1); print $1} ' regions.fas
```

In this case, the `sub` command replaces the first occurrence of the string `>` with the empty string `""` in the string found in $1; The `gsub` (global substitution) command replaces every occurrence:

```
awk ' /^[^>]/ {test=gsub(/[AGC]/,"",$1); print $1} ' regions.fas
```

Only T:s are printed. The first regular expression means all lines that don't start with `>`. Remember that the carrot sign in the square brackets (`[^>]`) should be interpreted as exclude the following list of characters (or strings). In this case there is only one character in the list. If we want to calculate the length of a string the `length` command is used. For instance, to print sequence lengths we could use:

```
awk ' /^[^>]/ {print length($0)} ' regions.fas
```

If we want to make the sequence in lowercase we use the `tolower` command:

```
awk ' /^>/ {print} /^[^>]/ {print tolower($0)} ' regions.fas
```

The corresponding command is `tolower`. Another way of writing this is:

```
awk ' /^>/ {print; next} {print tolower($0)} ' regions.fas
```

The `next` command exits the evalution for the current line (record) and the next line is parsed. Note that we don't have a pattern specification before the second curly braces pair. This means that all lines are matched. It should be observed that the patterns don't have to be mutually exclusive. Following is valid:

```
awk ' /^>/ {print} /^>/ {print tolower($1)} ' regions.fas
```

Finally we take a look at the `if` statement. Let's say that we want to retrieve sequences that are between 50 and 100 nucleotides long (including these numbers):

```
awk ' /^>/ {idline=$1; sub(/length=/,"",$2); \\
 if($2>=200 && $2<=250) {valid=1; print} else \\
 {valid=0}; next} {if (valid==1) {print}} ' regions.fas
```

We first save the entire id line in the variable `idline`. The predefined variable $2 will contain a string like: `length=236`. With `sub` we remove this string leaving just a number (sequence length) in variable $2. We then test if this number is greater than or equal to 200 and at the same time (`&&`) lesser than or equal to 250. If so we set the variable `valid` to 1, else we set it to 0. With the `next` statement we move to the next line. The second pair of curly braces doesn't have a pattern so all sequence lines will match. If `valid` is set to 1, we output the sequence line.

4 Built in variables

`awk`comes with some built in variables. Some of them are listed here:

OFS output field separator

ORS output record separator

FS input field separator

RS input record separator

NR Keeps a current count of the number of input records.

NF Keeps a count of the number of fields in an input record.

FILENAME Contains the name of the current input file.

```
Make a text file called listOfItems.txt with only one line:
80:100:60;70:70:100;1:1:2;1000:32:23
```

If we treat this as four records separated by semicolons, and we want to sum the three numbers for each record, and also want to output the record number together with the file name where we retrieved the data from, we could do like:

```
awk ' BEGIN {FS=":";RS=";"} {print NR, $1+$2+$3} \\
 END {print "Data from "FILENAME} ' listOfItems.txt
```

We could also have used an array. But if we have the need of using arrays with for example for loops, it's easier to stick with Perl. Do you want to know more use `man awk` from the command line or search the web.

5 Exercises