CS146 - Assignment #5: more shell scripting

Introduction

In this assignment, you are going to learn how to write "shell" scripts that have complex embedded awk(1) scripts. That is, the file will technically be a Bourne (or Bash) shell script, but one (or more) of the command-lines inside the shell script will be using awk; in turn, the awk program can be arbitrarily long and complicated. In the end, half or more of the characters in the file may actually be awk code. For example, say we want to evaluate mathematical expressions on the command-line. While Bash can handle the 4 basic operations (add, subtract, multiply, divide):

```
$ echo $(((15+5)/5)
4
```

it knows nothing about basic mathematical functions:

```
$ echo $((cos(0)))   # hoping to get cosine(0), which is 1
bash: cos(0): syntax error in expression
```

Luckily, awk knows about such simple functions:

```
$ (echo 0; echo 1; echo 3.1415) | awk '{print cos($1)}'
1
0.540302
-1
$
```

So if we want to evaluate mathematical functions on the Unix command line, we can write a little shell script around awk, as follows, which shows a script called "fexpr" (meaning "floating point" version of the existing expr(1) program):

```
$ cat fexpr
#!/bin/sh
exec awk "BEGIN{print $*; exit}"
$ fexpr 'cos(0)'
1
$ fexpr 'exp(1)'
2.71828
$ fexpr 'cos(1)^exp(1)' # raise cos(1) to the power of 2.71828
0.187599
$
```

Note: while normally awk executes the program you give it on every input line, in this case we want it to evaluate one expression, and exit. The `BEGIN` tag is always executed by awk before any lines are read. We put "$@" in double quotes, so the Bourne Shell will expand that to everything given on the command line of fexpr; and then we just exit, so awk doesn't sit around waiting for any input lines. To save on the expense of one `fork()` system call, we tell the Bourne shell to `exec` awk rather than forking it as a child process.

`fexpr` demonstrates the power of combining Bourne shell with awk; you might think that it would be easy to have the interpreter file use *only* awk, but the problem is that awk has no easy way to grab all it command line arguments at once, interpret them all together as one expression, and print the resulting value.[1] So instead, we let Bourne do the heavy lifting here and just use $* inside double-quotes, and BAM! we're done. Note also that this is one of the very few cases where we want to use "$*" rather than "$@".

---

[1] **Bonus:** figure out how to do it using awk as the interpreter file rather than Bourne. That means the first line of the file must start with "#!/bin/awk ..."—there could be more on that first line, but I'll let you figure it out. Note *this is hard*, which is why I'm assigning it as a bonus. I'm not ever sure it's *possible*, so don't spend too much time on it.

A: a few bite-size Bourne+awk scripts

1. `cleanPATH [PATH]`: take the contents of the PATH environment variable (or alternately a PATH-like argument as the only command-line argument), return the same list in the same order but with duplicates removed. (When there are duplicates, only the first entry should be returned).

   EXAMPLE: cleanPATH "/bin:/usr/bin:$HOME/bin:/bin" should remove the second "/bin" entry. Take extra care with any trailing colon: if there was one, it should remain; if there wasn't, there shouldn't be one.

2. `revlines [files, or standard input]`. **Purpose**: Read all inputs lines, and then output them in reverse order. If given no filenames, reverse the order of lines given on the standard input. If given a list of files, each file should individually have its lines reversed (but in the same order as given on the command line). BONUS (tiny bonus): do not use a Bourne shell loop over the filenames; instead, look up the awk BEGINFILE and ENDFILE blocks, as well as the FNR built-in variable.

   EXAMPLE: "ls — revlines" should give the same output as "ls -r".

3. `randomizeLines`: like revlines, except output the same lines as input but it *random* order. Use srand() to ensure you don't get the exact same "random" output each time.

4. `newlines`: insert newlines between every "word" of input, in order; remove all other whitespace, so that each word of input appears by itself on a line, with the *only* remaining whitespace output being the newlines. A "word" is defined as any sequence of characters delimited by whitespace on both sides (except of course the very first and last words of the input, which have nothing before and nothing after, respectively).

   EXAMPLE: "wc -w ¡foo.txt" should give the same output as "newlines foo.txt — wc -l"

5. `revchars arg1 [arg2 ...]` : reverse the order of the *characters* in each command-line argument (in order).

   EXAMPLE: "revchars abc def" gives "cba fed"

   HINT: combine the use of sed(1), newlines, revlines, and a short supplementary awk script (look up the use of "%c" in printf).

6. `top-tasks [username]`: run the top(1) command in dumb terminal mode (`TERM=dumb top -c -b -n 1`), and (a) output all the header lines (uptime, date, RAM, CPU(s), etc) verbatim, along with the column names above the active processes, (2) output each line representing a process owned by username (default: your own username) that is taking more than 0.0% CPU time, and (3) sum the total CPU time and memory usage of *all* the user's processes (not just those displayed). NOTE: This one isn't quite "bite-sized".

   EXAMPLE: Here's the output of my own top-tasks on odin, right now:

```
   odin:~$ top-tasks
   top - 22:09:55 up 57 days, 10:15, 37 users,  load average: 0.72, 0.95, 1.00
   Tasks: 871 total,   1 running, 864 sleeping,   6 stopped,   0 zombie
   %Cpu(s):  0.5 us,  0.9 sy,  0.0 ni, 97.9 id,  0.7 wa,  0.0 hi,  0.0 si,  0.0 st
   GiB Mem :    503.8 total,    424.5 free,      7.5 used,     71.9 buff/cache
   GiB Swap:      2.0 total,      1.9 free,      0.1 used.    493.4 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
126446 wayne     20   0   20772   5884   4416 R  37.5   0.0   0:00.15 top -c -b -n 1
User wayne has load 0.38, Vmem 0.88G, ResMem 0.30G
```

B: `awkcel`: use awk a bit like a spreadsheet

Write a Bourne shell script named `awkcel`. Yes, the name is a blatant rip-off of Excel(tm). It works on tab-separated files, which I recommend have the extension `.tsv`. (This need not be enforced; it will work on any file regardless of name, but the file *does*, in fact, need to be *tab* separated, and every tab is assumed to be a field separator.) Here is the first few lines of my solution:

```
$ cat awkcel
#!/bin/sh
# This is a front-end to awk(1), which allows NAMED COLUMNS to be accessed as variables.
# USAGE: awkcel {any standard awk program} FILENAME
# This script expects exactly two arguments.
# The first argument is an awk program that allows use of variables as described below.
# The second argument is an input file with a constant number of tab-separated columns.
# Any line that starts with a hash ("#") is assumed a comment, and discarded.
# The first (non-comment) line is the HEADER.
# The HEADER line contains tab-separated names of the columns.
# These column names must be valid awk(1) variable names.
# Each subsequent line must have the same number of tab-separated columns as the HEADER.
# COLUMNS SHOULD NOT BE EMPTY: behavior is undefined if any line has two adjacent tabs.
# Upon each line of input, values can be accessed using the name defined in the HEADER.
```

The above should be considered a *specification* of awkcel and must be abided by to get full marks. To enforce awk(1) using tabs to separate columns, use awk's -F option.

So for example, say the file name-studnum.tsv has the following:

```
$ cat name-studnum.tsv
name    studentNum  grade
hayes   00000       80.6
shawna  11111       85.4
dan     22222       83.7
$ awkcel 'studentNum=="00000"{print name, "is the prof"} \
         studentNum!="00000"{ \
              print name, "must be a TA or Reader or Student and got grade", \
              int(grade+0.5)}' name-studnum.tsv
hayes is the prof
shawna must be a TA or Reader or Student and got grade 85
dan must be a TA or Reader or Student 84
$
```

Some example tab-separated files are on openlab in ~wayne/pub/cs146/awkcel. Based on those files, and your implementation of `awkcel`, answer the following questions. Your openlab submission should include all code for `awkcel` itself, and your write-up (in PDF) should show the *user-level* commands you used to answer the below questions.

1. The file `orthologs.tsv` contains lists of proteins (encoded by genes) that are identical across various species; the technical term is that two proteins that perform the same function, regardless of species, are called "orthologs". On any given line, all proteins listed are orthologs of each other, and any species with an underscore "_" as an entry has no such ortholog.
   (a) Which pair of species has the greatest number of orthologs? Which pair has the least?
   (b) Create a table that lists, for every pair of species in that file, the number of orthologs they have. (You can use a shell script to iterate over the pairs, but for any given pair you should use `awkcel` to compute the number of orthologs for that pair.)

2. The file historical.2011.txt contains the closing prices for a few hundred medium-to-large cap stock market symbols for most trading days of 2011, in date order. (Empty entries contain a single underscore.) Once you think your `awkcel` is working, use it to answer the following questions.
   - (a) What were the closing prices of Amazon (AMZN) and Microsoft (MSFT) on July 1st, 2011? How about the 5th? Did they go up or down over the July 4th holiday?
   - (b) What was Amazon's worst day of 2011? (ie., worst 24-hour, close-to-close loss, in raw dollars, not percent)?
   - (c) Among all the companies in that file, which one suffered the worst single-day loss, as a percent of its price? Name the company, and the date. (Don't include any days surrounding days where the price is not listed, ie., is an underscore.)

BONUS: Try to write a program like `awkcel` in Python. How does its speed compare to `awkcel`? If you can make your Python version faster than *my* version of `awkcel` (which is just Bourne shell wrapped around awk), you automatically pass this course and get a 10% bonus on this assignment.

More questions for awkcel, this time about Spiral Galaxies

I have added more files to the directory `/home/wayne/pub/cs146/awkcel`. Note I've also added a shell script called `hawk`, which stands for "Hayes awk". It is simply a one-line shell scipt that runs normal `awk` but with a large number of new functions defined. Those functions are in the file `misc.awk` in the same directory. Feel free to peruse `misc.awk` for many examples of `awk` syntax. You are welcome to have your `awkcel` also read `misc.awk` (or better, just execute `hawk` rather than normal `awk`.

I have also added one very large (about 2.1GB) file called `SDSS+GZ1+SpArcFiRe+SFR.tsv`. It is an actual data file containing 341 columns and about 750,000 lines. Each line contains data for one galaxy. It is actual, real-life data on exactly 765,367 galaxies. The data are partly from the *Sloan Digital Sky Survey* (`http://sdss.org`), partly from the *Galaxy Zoo Project* (`http://zoo1.galaxyzoo.org`) and partly from *SpArcFiRe* (`http://sparcfire.ics.uci.edu`), a spiral galaxy analysis program written by one of my Ph.D. students a few years ago.

You don't need to know much about galaxies to do this assignment, but your `awkcel` needs to be pretty efficient to answer the following questions, since it's going to take quite awhile for `awk` to read through a 2.1GB file containing over 750,000 lines of 341 columns each. The only thing you need to know about galaxies is that some galaxies have spiral arms (for example the one that's on the background at Galaxy Zoo, or the one on our SpArcFiRe home page), and some are elliptical (ie., they have no spiral arm structure). Among spiral galaxies, some wind "clockwise", and others "anti-clockwise".

Among the columns in `SDSS+GZ1+SpArcFiRe+SFR.tsv`, two are named P_CW and P_ACW. They record the probability that the arms wind clockwise, and anti-clockwise, respectively. Using `awkcel`, define a variable P_SP=P_CW+P_ACW; this value tells us the probability that the galaxy shows observable spiral arm structure, regardless of the winding direction. Answer the following questions:
How many galaxies have P_SP exactly equal to 1? How about exactly equal to 0?
How many galaxies have P_SP > 0.5?
Print out a histogram of mean totalArcLength as a function of P_SP in increments of 0.1, giving 11 bins 0 through 10. Below is the exact `awkcel` command line to do this. (I've paragraphed it nicely but it could all go on one command line if you want.)

```
awkcel '{
    P_SP=P_CW+P_ACW;
    bin=int(10*P_SP);
    n[bin]++;
    numArcs[bin]+=totalArcLength;
}
END{
for(i=0;i<=10;i++)
    print i, n[i], numArcs[i]/n[i];
}' SDSS+GZ1+SpArcFiRe+SFR.tsv
```