Introduction to Unix Shell Scripting

Day 3 - Project, part II

Thomas Junier, Robin Engler

13-15 June 2023

Now we need to take action based on whether the line is a header or not. For this, we need to discuss how to output data.

Output

- `echo`: simple, primitive, doesn't handle `\t` and `\n` by default. Terminates lines with a `\n` unless told not to (`-n`).
- `printf`: formatted printing (see below); `\t` and `\n` work as expected.
- Any programs called will use our script's `stdout` and `stderr`.

`printf` - formatted printing

`printf`[1] takes a *format string* and zero or more arguments. *Placeholders* in the format string are replaced by the corresponding arguments.

```
place='Machu Picchu'; count=3
printf "I went to %s %d times.\n" "$place" "$count"
```

- `s` in `%s` specifies that the replacement is a string.
- `i` in `%i` specifies that the replacement is an integer.

[1] Inherited from the C language, and copied by countless others.

printf - some more examples

```
pi=3.1415926535
printf "The value of Pi is about %.2f\n" "$pi"
count=1000
printf "%s * %i = %e\n" "$count" "$count" \
       $(( count * count ))
```

→ use `echo` for simple tasks, `printf` for anything else.

Practice

Let's use our newly gained knowledge on output formatting to improve our `fasta2tsv` script.

→ **Exercise 3.1** - output formatting.

→ **Exercise 3.2** - Add input argument support to `fasta2tsv`.

A small problem

If you look carefully at our `fasta2tsv.sh` script, we have a redundancy in some of the code: two of the `printf` statements are (almost) identical:

```
read line
printf "%s%s" "${line:1}" "$FIELD_SEPARATOR"

while read line; do
    # Get the 1st char of the current line.
    ...
        printf '\n%s%s' "${line:1}" "$FIELD_SEPARATOR"
    ...
done
```

A small problem (continued)

This is a problem, because if we want to make a change to one of these lines, we will need to remember to do the same change on the other line as well. For example:

- Renaming a variable (say `FIELD_SEPARATOR` to `FIELD_SEP`).
- Optionally keeping the leading >.
- Optionally outputting other formats than CSV/TSV.

The **DRY** principle: **Don't Repeat Yourself**.
This leads us to *functions*.

Functions - Motivating Example

See `./src/func_motiv*.sh`

Functions

A *function* is like a miniature script that can be called from within another script. Calling a function causes its code to be executed, but does not by itself start a new process.

We write functions in order to:

- Re-use code (DRY principle).
- Improve the clarity of the code.
- Avoid creating new processes (they have a cost: see `noop_test.sh`)

Definition

Functions are defined with one of the following forms:

```
my_func() {
  <commands> # function body
}

function my_func {
 <commands> # function body
}
```

Call

A function is called just like a command. Arguments are passed in the same way:

```
my_func arg1 "$value"
```

Positional parameters (`$1`, `$@`, `shift`, etc.) work in the same way as in scripts.

Local Variables

By default, all variables in Bash are **global**: once declared, they can be accessed from anywhere in the script.
Variables defined in a function can[2] be declared as **local**, so that they are only available in the namespace of that function:

```
my_var='a'
my_func() {
  local my_var='b'  # A different variable.
}
```

See `../src/func.sh`.

[2] And usually should

“Returning values” from functions

Unlike in pretty much every other language, shell functions **do not** return values: instead, they return an exit status, again behaving like commands. To pass data to the caller, one can use command substitution:

```
my_func() { echo "Hi!"; }
result=$(my_func)  # The function is called just like any
```

The exit status is that of the last command in the function body, but it can be overridden with the `return` keyword.

Practice

Let's practice using functions to avoid duplication.

→ **Exercise 3.3** - Introduction to functions.

→ **Exercise 3.4** - Removing code duplication in `fasta2tsv`.

More control structures with `case` and `break`

Let's consider the scenario where we want to make the command line interface (CLI) of a script a bit more powerful.

For instance, in our `fasta2tsv.sh` script we might want to have the following optional arguments:

- `-s/--separator`, to allow specifying a custom field separator.
- `-o/--output`, to save the output of the script to a file, rather than printing it to stdout.
- `-h/--help`, to display help for the script.

How should we proceed?

Looping over Arguments

The most obvious solution is to loop over the values passed by the user, and for each value, test whether it is a recognized argument/option:

- If yes, process it and move on to the next value.
- If not, ignore the value or maybe raise an error (depending on how permissive you want the interface to be).

Let's see how we can do this in practice…

Looping over Arguments (continued)

We know about `$1`, `$2`, etc., but what if the number of arguments isn't known in advance (typically because some arguments are optional) ?

- `"$@"` expands to all arguments, one word each.

Note: there are other ways to access all arguments, but `"$@"` is the most useful one here. See `../src/showpp.sh`.

Intuitively, we might do something like:

```
for arg in "$@"; do # or just: for arg; do
  # process $arg -> leaves $@ intact.
done
```

Intuitively, we might do something like:

```
for arg in "$@"; do # or just: for arg; do
  # process $arg -> leaves $@ intact.
done
```

Another possibility is

```
while [[ $@ ]]; do
  # process $1
  shift # "consumes" $1 -> changes $@.
done
```

Processing Arguments

To identify which argument corresponds to which option, we *could* do something like:

```
if [[ "$1" == '-h' ]] ; then ...
elif [[ "$1" == '-s' ]] ; then ...
elif [[ "$1" == '-t' ]] ; then ...
fi
```

But there is a better way…

case keyword

The `case`[3] statement is like a multi-way `if...elif...`:

```
case <word> in
  <pattern1> ) <list1> ;;
  <pattern2> ) <list2> ;;
  <pattern3> ) <list3> ;; # or more...
esac
```

`<word>` is *pattern-matched* (like globbing) against each `<pattern>` in turn, and the first match causes the corresponding `<list>` to be executed. After a match is found, the `case` block exits (i.e. subsequent patterns are not attempted to be matched against).

[3] Very much like `switch` in C/C++/Java/Perl, or `case` in Ruby, or `match` in Python.

case - continued

Usually written on more than 1 line per clause:

```
case $arg in
    '-s' )
        # do something
        ;;
    '*' ) # default - matches anything
        ;;
esac
```

Note: Word splitting is disabled between `case` and `in`.

Breaking out of Loops

break allows an early exit from a loop:

```
# Print the first line that contains some string.
while read line; do
  if [[ $line == *GAATTC* ]]; then
    printf "%s\n" "$line"
    break
  fi
done
```

And now we've reinvented `grep -m 1`!

To break out of *n* nested loops, use **break n**.

Practice - Exercise 3.5

Ok, now we know everything we need to parse options, let's put this knowledge into practice to improve the CLI of our `fasta2tsv.sh` script.

→ **Exercise 3.5** - Final touches to `fasta2tsv.sh`

A small problem

We can now specify the separator: the script can separate with tabs (the default) or anything else. But see what happens with CSV:

```
./fasta2tsv-stage-13.sh -t -s ',' \
  < ../data/Spo0A.msa | csvlook
```

Why might that be?

A small problem

We can now specify the separator: the script can separate with tabs (the default) or anything else. But see what happens with CSV:

```
./fasta2tsv-stage-13.sh -t -s ',' \
  < ../data/Spo0A.msa | csvlook
```

Why might that be?

→ The header contains commas.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.
- **Warn** - succeed, but emit a warning message.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.
- **Warn** - succeed, but emit a warning message.
- **Attempt to fix** the header.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.
- **Warn** - succeed, but emit a warning message.
- **Attempt to fix** the header.
- **Ignore** it.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.
- **Warn** - succeed, but emit a warning message.
- **Attempt to fix** the header.
- **Ignore** it.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

Dealing with Separators

We can deal with the problem in the following ways[4]:

- **Fail** - output a failure message, and return a nonzero exit code.
- **Warn** - succeed, but emit a warning message.
- **Attempt to fix** the header.
- **Ignore** it.

Let's first try to at least warn the user of the problem.

[4]The way the script deals with the problem could itself be a parameter, *e.g.* warn by default, but be more strict or lax according to some option.

`fasta2tsv.sh`: Stage 14

The check for separators will have to be done in two places in the code, so it's a good candidate for a function. The same goes for emitting a warning, if needed.

(see `../src/fasta2tsv-stage-14.sh`).

Further Ideas

The `fasta2tsv.sh` script is now functional and can be used in real life. We'll stop here, but here are some ideas for further improvement:

- Instead of *assuming* that the first line is a header, check that it is (and take action if it isn't).
- Deal with quotes.

Name References

Consider computing the *union* vs. the *intersection* of two arrays:

```
a1=(A B C)
a2=(D E F)
u=$("${a1[@}}" "${a2[@]}")
```

Conclusion

- We have seen how the shell operates, including tokenizing, parsing, the various forms of expansions, conditionals and arithmetic.
- We have used this knowledge (and more) to write a script that converts Fasta to CSV. It's not efficient, but it's easy to understand.
- You should now be able to start writing your own scripts.

May You Solve Interesting Problems

For some people, myself included, the satisfaction of solving a problem is the difference between work and drudgery. [...] Besides, once I accomplish my task, I congratulate myself on being clever. I feel like I have done a little bit of magic and spared myself some dull labor.

Dale Dougherty and Arnold Robbins, *sed & awk* (2010)

Learning shell scripting - Resources

- `man bash` - always handy, if a bit terse.
- `help` - for Bash builtins etc.
- The Bash website and especially the Bash Manual (not the same as `man bash`!).
- The Advanced Bash Scripting Guide.
- The Bash Cheat Sheet.
- The Bash Programming Reference. is another cheatsheet, more specialized towards programming.

- Shellcheck - static analysis tool
- Bash Cookbook
- Bash Idioms

Thank You for your Attention

Appendix I: Protecting against Errors

Unset variables

Unset variables can cause hard-to-find bugs, and they can even be dangerous:

```
prefix=tmp
rm $prefix*      # -> rm tmp* (+- ok)

rm $prefiy*      # oops! typo...
# Result: rm *   -> I just deleted all files in my director

set -u           # or shot -o -s nounset
rm $prefiy
# -> bash: prefiy: unbound variable
```

NOTE no quotes around `$prefix*` (why?)

Mutable Variables

To prevent accidentally modifying variables, use `declare -r`.

Appendix II: Solutions involving CSV

Longest Sequence

```
awk -F '\t' '{print length($2), $1}' \
  < sample_00.csv | sort -k1,1n | tail -1
```

or, in pure Bash:

```
while IFS=$'\t' read header sequence; do
    echo ${#sequence} ${header%%;*}
done  < sample_00.csv | sort -k1,1n | tail -1
```

Discard Sequences with too many Gaps

```
AA=ACDEFGHIKLMNPQRSTVWY  # amino acids

while IFS=$'\t' read hdr seq; do
  gaps=${seq//[$AA]/}
  ((${#gaps} < 50)) && echo "$hdr ${#gaps}"
done < Spo0A.tsv
```

Sort sequences by Genus

```
../src/fasta2tsv-stage-12.sh < sample_00.dna \
  | awk -F'\t' '{
  match($1, /g:.*,/);
  genus=substr($1, RSTART+2, RLENGTH-3);
  fprintf ">%s\n%s\n", $1, $2 >genus".fas"
}'
```