Introduction to Unix Shell Scripting

Day 2 - Project, part I

Thomas Junier, Robin Engler

13-15 June 2023

FASTA to TSV converter script project

The Trouble with Fasta

Here are some operations that one might wish to perform on a FASTA file:

- Count the number of records.
- Count the number of species.
- Find the longest sequence.
- Reject sequences with ambiguous nucleotides (`N`, `Y`, etc.).
- Discard aligned sequences with too many gaps.
- Partition records by species.

With shell tools, the first two are trivial, but **the last four are next to impossible**.

Why ?

- Unix shell tools (`sed`, `awk`, `grep`, etc.) are predominantly *line-oriented*.
- Some bioinformatics formats are line-oriented (*e.g.* GFF, VCF)... but Fasta is not (neither are GenBank, UniProt, ...).
- Converting FASTA to some line-oriented format (*e.g.* CSV) would solve the problem.[1]

[1] The *format* problem, that is - the rest can be left to `grep` and the like.

The project

To be able to perform more operations easily on FASTA file content, we are going to write a **FASTA $\rightarrow$ CSV converter script**.

WARNING

Didactical Script!

The script is meant to *illustrate* concepts, **not** to be efficient.
$\Rightarrow$ We'll write it in pure style. A real-world script would be in delegation style and very different (but mostly useless for teaching).

WARNING

Didactical Script!

The script is meant to *illustrate* concepts, **not** to be efficient.
$\Rightarrow$ We'll write it in pure style. A real-world script would be in delegation style and very different (but mostly useless for teaching).

Indeed, a FASTA $\rightarrow$ CSV converter could be written in one line of `sed`...

```
sed -n '1{h;b};${H;bo};/^>/!{H;b};:o;x;
          s/^>//;s/\n/\t/;s/\n//g;p'
```

Practice

→ **Exercise 2.1** - FASTA to CSV converter: program logic.

Now that we can run scripts, it's time to read in our first line of data: let's have a look at some ways of getting data into our script.

Input

There are many ways of getting data into our script, including:

- Reusing **standard input**.
- **Positional parameters**.
- The `read` and `mapfile` functions.
- Command substitution: `$(...)`.
- Environment variables.

Standard Input (`stdin`)

Commands called by a script inherit its standard input:

```
#!/bin/bash
# reuse_stdin.sh
grep Spo0
# at this point, stdin is used up!
```

```
./reuse_stdin.sh < ../data/Spo0A.msa
```

Here `grep`'s stdin is the same as `reuse_stdin.sh`'s, namely `../data/Spo0A.msa`.

Practice

→ **Exercise 2.2** - Reusing standard input.

Arguments and Positional Parameters

A command's arguments are accessed through special parameters `$1`, `$2`, etc., which hold the first, second, etc. arguments, respectively.

These are called **positional parameters**.

```
./show_args.sh -f -y optarg arg1 arg{2..4}
```

- `$0` holds the name of the script (or function).
- `$@` holds all argument values passed to the script (or function).
- `$#` hold the count of the number of argument values passed to the script (or function).

We can now change our script to:

```
#!/bin/bash
# pos_arg.sh
grep Spo0A "$1"
```

```
./pos_arg.sh ../data/Spo0A.msa
```

Sadly, the script now longer works with the < redirection, because then there is no `$1` (do `set -u` to check). [2]

[2] But there is a way to make a script work in both ways.

Arguments don't have to be filenames

- We have used `$1` to specify a data file (by its name).
- The effect is similar to redirecting `stdin` to that file.
- This is very frequent.
- But positional parameters can of course used for countless other purposes.

Up to here we have been reading whole files. We can have finer control by reading only **one *line* at a time**.

read

The `read` builtin[3] sets variables to values passed on `stdin`. It reads one line, word-splits it, and sets the corresponding identifiers:

```
read name
echo "Hi, $name!"
```

The command fails (returns 1) if it can't read anything.

There can be more than one identifiers:

```
read x y z <<< '1 2 3'
# x=1, y=2, y=3
```

[3] A builtin command works like a program but is part of the shell

If there are more words (after splitting) than there are identifiers, the last identifier gets the extra words:

```
read x y z <<< '1 2 3 4 5'
# x: 1, y: 2, z: 3 4 5
```

If there is only one identifier, it captures the whole line:

```
read line
```

If there are fewer words than identifiers, the extra identifiers are set to an empty string.

Practice

→ **Exercise 2.3** - Getting user input from arguments.

This works, but it only reads *one* line - we need to read *all* of them. For this, we need a way of repeating things: a **loop**. This brings us to *control structures*.

Control Structures

Beyond a sequence of simple commands, the script's structure is given by *compound commands*:

- **Loops** (repetition) - shortly
- **Conditionals** (choice) - later
- **Groups** (sequential flow) - not covered

They start and stop with a specific *keyword* (`if...fi`, `{...}`, `while...done`, etc.)

Loops

- Loops are *iterative* **control structures**: they *repeat* commands (called the *body*), typically with minor modifications.
- They operate either:
  - For a **fixed** number of iterations (`for ... in`).
  - Until some **condition** is met (`while`, `until`, `for ((...))`). This condition is expressed as a *test command* (details later). The test succeeds **iff** the command has 0 exit status.

`for` loop - 1st form

```
for <name> in <words> ; do <commands> ;  done
```

Expands `<words>`, and executes `<commands>`, binding `name` to each of the resulting values in turn.

The “;” can be (and often is) replaced with a newline.

`for` loop - Example 1

`for` loops are frequently used with file globs.
The following copies a bunch of `*.fasta` files to `*.fas`:

```
for file in *.fasta ; do
  # body
  cp $file ${file/.fasta/.fas} # more on ${//} later.
done
```

`for` loop - Example 2

Another typical case is with a sequence, often generated by a brace expansion (here `{1..10}` expands to the numbers from 1 to 10):

```
# Compute the squares of numbers from 1 to 10.
for n in {1..10} ; do echo $((n**2)) ;  done
```

(recall that `$((...))` is arithmetic expansion)

`for` loop - 2nd form

Another way to write a `for` loop is using the **C syntax** [4].

```
for ((<start-cmd>; <condition>; <iteration-cmd>)); do
    <list>;
done
```

1. Evaluate `<start-cmd>`.
2. Evaluate `<condition>`; if true execute `<list>`, if not exit loop.
3. Evaluate `<iteration-cmd>` and go back to 2.

The evaluations are done in shell arithmetic.

[4] So named because it was copied from the C language

Example: `for` loop, 2nd form

```
# Powers of 2 smaller than 10000.
for ((p=1; p<10000; p=2*p)); do
  echo $p
done
```

Notes:

- Geometric, not arithmetic progression (as in `{1..10}`).
- Only numeric (no globs, etc.).

`while` loop

In `while` loops, the body is executed **as long as `<test-command>` succeeds**.

```
while <test-command> ; do
  <commands> # body
done
```

`while-example.sh` - print suffixes of a string.

```
#!/bin/bash

# Get user input and assign it to a variable.
input=$1

# This succeeds as long as $input is not empty
while [[ $input ]] ; do
    echo $input
    input="${input:1}" # shorten by 1
done
```

`until` loop

This works like the `while` loop, except that the body is executed as long as the test *fails*.

```
until <conditional> ; do
 <statements>
done
```

The `continue` keyword

All loops (`for`, `while` and `until`) support the use of the `continue` keyword that makes execution of code **skip the remainder of the current loop**.

Example: print all multiples of 7 between 0 and 100.

```
for x in {0..100}; do
  ((x % 7 != 0)) && continue
  echo $x
  # Further operations when x is a multiple of 7
done
```

Practice

→ **Exercise 2.4** - a simple `for` loop.

Iterating over lines in a file

So, how do we iterate on lines of a file?

Iterating over lines in a file

So, how do we iterate on lines of a file?

Reading an entire file, one line at a time.

```
#!/usr/bin/env bash

while read line; do
    # print the line, just to check.
    echo "$line"
done
```

We've just reinvented `cat`! (except that ours is slower...)

Now we need to treat header lines differently from sequence lines. How do we tell them apart?

Now we need to treat header lines differently from sequence lines. How do we tell them apart?

Suggestions ?

Now we need to treat header lines differently from sequence lines. How do we tell them apart?

Suggestions ?

$\rightarrow$ We look at the first character of the line (> for headers, residue symbol for sequence lines). Time for a deeper look at *parameters*.

Parameters

- A *parameter* (also called a *variable*) is just a name for a value.
- Values are essentially just strings, although
  - They can be interpreted numerically
  - They can be stored in indexed structures (arrays)
- Variables are the way a script stores values for later use (often a transformation or decision).
- There are multiple ways of setting them, and many more ways of retrieving their values (including handy operations).

Setting

The simplest way is to use =, the *assignment* operator:

```
answer=42              # no need to declare
human='Homo sapiens' # quotes!
date=$(date); shell_msg="I use $SHELL"
species=$human         # no word splitting
answer="nuts!"         # can reassign freely
```

- There is **no whitespace** around the equals sign!
- Identifiers contain letters, digits, or underscores; they cannot not start with a digit.
- Word splitting is disabled (except for arrays - see below).

Getting

The basic form for getting the value of variable `var` is `${var}`. As we have seen, this occurs during the *parameter expansion* step.

```
place=London  # set value
echo ${place} # get value: ${...}
echo $place   # short form
```

Beware of the short form! Say we have filenames like `<spc>-18S.dna`: etc.

```
# Count nb lines in mammals
for spc in cat dog gnu hog rat; do
  wc -l $spc_18S.dna # WRONG!
  wc -l ${spc}_18S.dna
done
```

Can you guess what the rule is?

`set -u` guards against unset variables[5].

[5] See Appendix I (day 3 slides).

Unset and Null

- A *null* variable has the empty string for a value.
- An *unset* variable does not exist at all (and it's usually a mistake to try to use its value[6]).
- A variable can be deleted with `unset`.

```
place=Seoul     # non-empty
place=''        # empty
place=          # same as place=''
unset place     # (NOT $place!) - deleted
```

[6]Think of `NULL`, `null`, `nil`, `None`, or `Nothing` in your favorite language.

Defaults for Null and Unset

The "default" operator `:-` substitutes a value if a parameter is unset or null.

```
unset notset
echo $notset
echo ${notset:-default}
null=
echo $null
echo ${null:-default}
```

Variants

- `:=` also *sets* the variable.
- Both have forms (`-`, `=`) that only check for unset (not null).

Notes:

- There must be **no whitespace** around `-`, `:-`, etc..
- The right-hand side is *expanded*:

```
unset var; msg="Unset var!"; echo ${var-$msg}
```

Constants

You can prevent a parameter from changing values

```
declare -r PI=3.14159
PI=4096
# -> bash: PI: readonly variable
```

Readonly variables can't be `unset`.

Type

By default, Bash's *simple* values are **strings**:[7] In certain contexts, Bash treats them as numbers:

```
a=20; b=30; echo $a+$b       # string
a=20; b=30; echo $(($a+$b)) # arithmetic
declare -i n                 # arithmetic
```

[7] I.e., it doesn't know about numbers (let alone different kinds of them!), dates, characters – IOW it's a (very) weakly typed language.

String Operations I

```
genus=Harpagofututor
${#genus}          # length -> 14
${genus:7}         # substring from 7 (0-based)
${genus:0:7}       # substring of length 7 from 0
${genus: -5:4}     # SPACE is required (:-)
a=7; b=14
${genus:a:b}       # arithmetic evaluation
```

String Operations II

```
prog=BLASTN
${prog/N/P}        # substitute 1st match
DNA=cgatgtattcag
RNA=${DNA//t/u}    # idem, all matches (t->u)
img=figure1.jpeg
${img%jp*g}svg     # delete pattern at end
ext=.png
${img/.jp?g/$ext} # expansion happens
```

Many more variants (`man bash`).

Arrays

- *indexed* arrays (or just "arrays" for short) store lists of values referred to by a nonnegative integer. They work like the (1D) arrays of other languages, e.g. `weights[7]` could be the 7th element[8] in an array of weight values.
- *associative* arrays store key-value pairs, like Python's dictionaries or Ruby/Perl's hashes, e.g. `nb_reads['rec_A']`[9] for the number of reads mapping to the *recA* gene.

[8] Or the 8th, depending on the language

[9] Again, the syntax is language-dependent.

Indexed Arrays

```
ary=(1 two 'Hey there') # create whole array
ary[3]=foo              # set individual element
ary+=(bar)              # append
unset ary[1]            # delete element
unset ary               # delete array
echo ${ary[0]}          # 0-based
declare -p ary          # inspect array
```

Accessing All Array Elements

Using * or @ as index refers to all array elements, but with subtle differences depending on quoting:[10]

```
names=('Bilbo Baggins' Beorn Gollum)
showa ${names[@]}   # (or *): 4 arguments!
showa "${names[@]}" # 3 arguments
showa "${names[*]}" # 1 argument
IFS=','; echo "${names[*]}"; unset IFS
```

The # operator yields the number of elements

```
echo ${#names[@]} # (or *)
```

[10] And also on whether or not word splitting occurs.

Iterating over an Array

```
for e in "${array[@]}"; do ... ; done
```

or

```
for ((i=0; i < ${#ary[*]}; i++)); do ... done
```

Example of Array Usage

Cf. `../src/pascal.sh`

Arrays and Word Splitting

Word splitting is *NOT disabled* when creating arrays:

```
elements='A B "C D"'
array=($elements) # 4 elements!
```

Neither is file globbing:

```
pdf_glob='*.pdf'; echo "$pdf_glob"
PDF_S=$pdf_glob ; echo "$PDF_S"
PDF_A=($pdf_glob); declare -p PDF_A
```

Array Caveats

- If no index is given, 0 is assumed:

```
names=(Frodo Lobelia Arwen)
echo $names # = ${names[0]} -> Frodo
```

- Arrays can't be assigned as values:

```
# Try to make a copy of `names`
lotr_names=names # WRONG - string assignment
lotr_names=$names # WRONG - see above
lotr_names=(${lotr_names[@]}) # OK
```

Associative Arrays

Associative arrays *must* be `declared` as such:

```
declare -A aar
aar[key1]=val1
declare -A aar=(K1 V1 K2 V2)
echo ${aar[key1]}
```

The behaviour is otherwise very similar to that of indexed arrays.

Note: `*` and `@` work on the *values*, to expand the *keys* use a `!`: `${!aar[*]}`, etc.

Associative Array Caveats

The order in which associative array elements are expanded is **unpredictable**. In particular, there is *no guarantee* that values (or keys) will be listed in the order they were added to the array.

Practice

Back to our **FASTA to TSV converter script**!

After this long survey of parameters, we are now equipped to formulate an expression that will extract the first character of a line, to check whether it is a header line or not.

$\rightarrow$ **Exercise 2.5** - FASTA to CSV converter: reading the input file line by line.

Now that we have the first character of each line, we need to test if it's '>' or something else.

This is going to be a long excursion into testing.

Conditionals

Conditionals are *branching* control structures. They enable the script to **choose what to do** between two or more possibilities.

The main conditional constructs are:

- `if` - yes-or-no decisions (possibly nested), based on a *test command*
- `case` - multi-way decision, based on pattern matching

if

The basic idea:

```
if <test-command> ; then
   <statements> # iff test-command returns 0
fi
```

See `check_user.sh` for an example.

Before we look in detail at test commands, we need to see the full version:

```
if <test-command> ; then
   <statements>
elif <other-test-commands> ; then
   <other-statements>
else
   <default-commands>;
fi
```

- There can be 0 or more `elif` clauses.
- There can be 0 or 1 `else` clause.

A shortcut

Due to how lists work, an `if` clause can be[11] (and often is) shortened to one of the following forms:

```
<test-command> && <success-command>
<test-command> || <failure-command>
<test-command> && <success-command> || \
  <failure-command>
grep $USER /etc/passwd && echo found || \
echo 'NOT found'
```

[11] Strictly speaking, the `<cmd> && ... || ...` is not equivalent to `if <cmd> then ... else ... fi`

Now we can look at test commands.

Test Commands

Test commands are the main ingredient of `if...then` conditionals and `while/until` loops. They can be:

- a *list* - the test succeeds iff the list itself succeeds (returns 0);
- a *conditional expression* between `[[` and `]]` - the test succeeds if the expression is true, the expression involves *strings* (including filenames);[12]
- an *arithmetic expression* between `((` and `))` - the test succeeds iff the expression is **nonzero**.

[12] An older form for conditional expression used `[...]` or `test`.

Why can 0 signal both success and failure?

Historical aside

- Unix: (many) more ways to fail than to succeed $\rightarrow$ 0 for success, > 0 for various kinds of errors.
- Early shells: crude Boolean and arithmetic expressions (if at all).
- C language: Boolean algebra with 0 for false and nonzero for true.

$\Rightarrow$ When (or "if") arithmetic and Boolean expressions were added to shells, they were strongly influenced by C.[a]

[a] As was the `for (( ; ; ))` loop.

Commands as Tests

Quite simply: check if the command succeeds. We have already seen this in `check_user.sh`.

Conditional Expressions

The expression between `[[` and `]]` is a *conditional expression*. It may involve the following:

- file attributes (*e.g.* existence, permissions, ...)
- variable properties (*e.g.* set or unset, integer, ...)
- string properties (*e.g* length)
- string comparison (equality, order, etc.)
- numeric comparison

Word splitting is **disabled** between `[[ ]]`, so `""` have no effect.

File Properties

They take the form `-c filename`, where the character `c` denotes a file property. For example, to check if a file exists, use `-e`:

```
if [[ -e file.txt ]]; then echo "exists!"; fi
# short form
[[ -e file.txt ]] && echo "exists!"
```

Many properties can be tested in this way (see next slide).

A few file property test operators

operator	true if
`-d`	file is a directory
`-f`	file exists and is a regular file (*e.g.*, not a dir)
`-r`	file is readable
`-w`	file is writable
`-x`	file is executable
`-s`	file exists and has a size greater than 0

There are also a few file *comparison* operators, such as `f1 -nt f2` which is true iff `f1` is newer than `f2`. Obviously, they expect *two* arguments.

String Conditionals

The following operate on strings

Operator	True if
`s`	s is a non-empty string
`s1 == s2`	s1 and s2 are equal **strings**
`s1 = s2`	idem (note spaces!)
`s1 != s2`	s1 and s2 are different
`s1 < s2`	s1 comes before s2 **alphabetically**
`s1 > s2`	s1 comes after s2 **alphabetically**

Note that the above operators interpret their operands as *strings* (also, > and < have an entirely different meaning here from I/O redirection)

```
[[ 2+2 == 4 ]]  # false!
[[ 10 < 2 ]]    # true!
```

`[[...]]` *can* compare numbers[13] (we'll see this in a moment).

[13] Though perhaps it shouldn't be used for that.

Pattern Matching

If unquoted, the right-hand argument of a == or != is treated as a pattern (“glob”):

```
[[ abc == ?bc ]]   # true
[[ abc == "?bc" ]] # false (quotes)
gene=lacZ; [[ $gene == lac? ]]
```

Note that the glob pattern isn’t matched against files, but against the **left-hand argument**.

Matching Regular Expressions

The =~ operator matches against POSIX regular expressions[14]

```
[[ abc =~ .bc ]]  # true
[[ abc =~ abc? ]] # true
[[ abc == abc? ]] # false (glob)
```

[14] à la grep, Perl, sed, and so many others... each with its own dialect, all different from globs :-(

Practice

Back to our **FASTA to TSV converter script**!

→ **Exercise 2.6** - FASTA to CSV converter: testing for header lines.

Numeric Conditionals

The following operators also operate on strings, but *treat their operands numerically*[15]:

Operator	Meaning (numerically)
`s1 -eq s2`	$s1 = s2$
`s1 -ne s2`	$s1 \neq s2$
`s1 -lt s2`	$s1 < s2$
`s1 -le s2`	$s1 \leq s2$
`s1 -gt s2`	$s1 > s2$
`s1 -ge s2`	$s1 \geq s2$

[15] Using shell arithmetic, more on this shortly

This way things make more sense:

```
[[ 2+2 -eq 4 ]] # true
[[ 10 -lt 2 ]]  # false!
```

Expansion Happens

I have given examples mostly with literals for simplicity's sake, but (most) expansions are entirely possible in test commands:

```
[[ -x "$HOME" ]] # better not be false
[[ "$SHELL" == /usr/bin/bash ]]
[[ /usr/bin/echo == $(which echo) ]]
a=1; b=1; [[ $a -eq $b ]]
```

Logical Operations

The expressions can be connected with the logical operators `(...)`, `!`, `&&`, and `||` (decreasing order of precedence).

```
a=2; [[ a == a && ! (b == b) ]]
```

Note that the *logical* operators `&&`, `||`and `!` are **not the same** as the *control* operators `&&`, `||`and `!`.

- Comparing numbers with `[[...]]` is cumbersome (to say the least).
- Fortunately, there is a another way of doing numerical comparisons.

Arithmetic Tests

In this test, the expression between `((` and `))` is evaluated using the rules of shell arithmetic (more on this shortly). The test succeeds if the numerical result is **not** zero (contrary to command exit status).

```
(( 2+2 == 4 )) # true
(( 10 < 2 ))   # false!
a=2; ((a+a == a*a))
```

Shell Arithmetic

1. Parameters are expanded ($ not needed) and treated as numbers[16]
2. The resulting expression is evaluated numerically (operators on next slide)

- Shell arithmetic is used in arithmetic expansion, arithmetic conditionals, and a few other contexts.
- Bash can only do *integer* arithmetic! (more on this later)

[16] As far as possible

Table 4: Main Arithmetic Operators, by decreasing precedence - `()` override.

operator	function
`!`	logical negation
`**`	exponentiation
`*`, `/`, `%`	multiplication, division, remainder
`+`, `-`	addition, subtraction
`<`, `>`, `<=`, `>=`	comparison
`==`, `!=`	logical equality and inequality
`&&`	logical AND
`||`	logical OR
`=`, `+=`, `*=`, ...	assignment

Shell Arithmetic - Examples

To illustrate shell arithmetic, we can simply use arithmetic expansion

```
a=2; b=5
echo $((a**3)) # spaces matter less than in [[...]]
echo $((a ** 3))
echo $((a**3 % 3))
echo $((a < b))
```

Recall that in this context 1 represents true and 0 represents false (contrary to command exit status).

No floating-point?

```
echo $((5/2))       # wtf?
echo $((5.0/2.0))   # fails!
```

→ use an external program[17]

```
echo "scale=1; $b/$a" | bc
bc <<< "$scale=1; $b/$a"
result=$(bc <<< "$scale=1; $b/$a") && echo $result
```

[17] In the same way you use `sed`, `awk` etc. when the shell's string functions are too limited

Appendix I: Sample Formats

General Feature Format (GFF)

```
##gff-version 3.1.26
##sequence-region ctg123 1 1497228
ctg123 . gene 1000 9000 . + . ID=gene00001;Name=EDEN
ctg123 . mRNA 1050 9000 . + . ID=mRNA00001;Parent=...
ctg123 . mRNA 1050 9000 . + . ID=mRNA00002;Parent=...
ctg123 . mRNA 1300 9000 . + . ID=mRNA00003;Parent=...
ctg123 . exon 1300 1500 . + . ID=exon00001;Parent=...
ctg123 . exon 1050 1500 . + . ID=exon00002;Parent=...
ctg123 . exon 3000 3902 . + . ID=exon00003;Parent=...
```

Variant Call Format (VCF)

```
##fileformat=VCFv4.3
##source=myImputationProgramV3.1
...
#CHROM POS     ID        REF ALT    QUAL FILTER INFO ...
20     14370   rs6054257 G   A      29   PASS   NS=3;...
20     17330   .         T   A      3    q10    NS=3;...
20     1110696 rs6040355 A   G,T    67   PASS   NS=2;...
20     1230237 .         T   .      47   PASS   NS=3;...
20     1234567 microsat1 GTC G,GTCT 50   PASS   NS=3;...
```