Introduction to Unix Shell Scripting

Day 1 - Fundamentals

Thomas Junier, Robin Engler

22-23 May 2024

Bash Fundamentals

Course Structure

1. What shell scripts are and what they are good for
2. How the shell works
3. Building Blocks of Shell Programs
4. Project

Practice

Time for a little warmup:

$\rightarrow$ **Exercise 1.1** - a complex pipeline.

Scripting - the next level

Suppose I wanted to perform the same task (as in Exercise 1.1) on **all other `data/sample_??.dna` files**.

What are my options? (suggestions welcome!)

Scripting - the next level

Suppose I wanted to perform the same task (as in Exercise 1.1) on **all other `data/sample_??.dna` files**.

What are my options? (suggestions welcome!)

- Type the commands again, once per file.
- Use the history mechanism (↑, `!`) - saves a lot of typing.
- Use a loop - saves even more typing.

However:

- Typing the same command multiple times is boring, time-consuming, and worst of all **error-prone**.
- The history is **local** and **limited** — what if I need to do the same task on another machine, or when it's no longer in the history?
- The history repeats commands **exactly** — inconvenient when parameters change (e.g. input files)

$\rightarrow$ What we need is a form of **long-term** storage for our command, and that can handle at least some **variation**.

In other words, a *shell script*.

What are shell scripts?

Shell scripts:

- Are **plain text** files.
- Contain **shell instructions** - just like the ones we type when working interactively.
- Can be **launched**[1] like any other program.
  - input / output
  - command-line arguments
  - ...

[1] Usually, but not necessarily, from the terminal

To elaborate...

- The shell is a full **programming language** (not just a command dispatcher) with constructs like *variables*, *control structures*, and *functions*.
- However, it is *specialized*[2] for **running other programs** (including in **concurrency** or **parallel**), **passing data** between them, and manipulating files. It can also do **simple** computations.
- By contrast, Python & the like deal primarily with numbers, strings, and composite structures made out of these.

[2]The shell can be considered a Domain-Specific Language, or DSL

Practice

Let's try to write our first shell script.

→ **Exercise 1.2** - a simple shell script.

What are shell scripts good for?

Typical Use Case

Automating tasks that we already do (but manually) *in the terminal*.

Two Styles

Shells can handle a given task in one of two ways:

- *Directly* ("pure" style) — the shell performs the task **itself**.
- By *delegation* — the shell **calls** one or more other program(s) that do the task.

→ The script from Exercise 1.2 is an example of the delegation style.

Note that the same program can use both approaches.

Delegation Style: Conductor Metaphor

Verdi conducting *Aida*. Engraving by Adrien Marie, 1881 (public domain)

- The conductor does not play any instrument...
- ... but does play an essential part.[a]
- The shell may do no heavy computation...
- ... but still has a crucial role.

[a] And, of course, Verdi still wrote the whole opera score...

You will likely need Shell scripting Skills

The Curse of Uniqueness

Sooner or later you'll face a task that nobody has programmed for you.

- It may, however, be possible to break it down into parts that *can* be carried out by existing programs.
- In that case, shell scripting is almost certainly part of the solution.

When/How **not** to use Shells?

Shell scripting is **not** recommended if you need:

- Speed (pure style, especially).
- Non-trivial data types, *e.g.* floating-point numbers, any kind of structure beyond 1-D arrays (matrices, trees, database records, dates, ...).
- Functional or object-oriented programming.
- Arbitrary access to file contents.
- Mathematical operations beyond the basics (trigonometry, logs, statistics, etc.).

When/How **not** to use Shells? (continued)

In such cases, it's better to program in a language more adapted to your task (*e.g.* Python, Julia, Rust, etc.).

$\rightarrow$ You can always **call** your program from a shell script (that is, in delegation style).

To sum up

We now know:

- What shell scripting is.
- What it can (and cannot) do for us.
- That it may well prove useful.

$\Rightarrow$ we're ready to start.

Just a Few More Points

Long-term learning Objective

Know enough to learn the rest by yourselves.

Some terminology

Shell, n.: a computer **program** which exposes an *operating system's services* to a human user or other programs (source: Wikipedia)[3]. By extension: any **command interpreter** (Python/R/Julia shell, etc.)

Shell, n.: the programming **language** that a text-based shell implements[4].

Shell, n.: a **terminal** emulator.

[3] Some shells (*sensu lato*) are graphical; in this course we mean **text shells only**.

[4] Graphical shells are typically not programmable.

Shell scripting, n.: the art of **writing programs** using **a shell language**.

Script, n.: a program, usually in a (high-level), **interpreted** language (*e.g.* Python, R, Perl, Ruby, JavaScript, and (of course) shells, but *not* C, Java, Rust…)

Which Shell?

- We'll use **Bash**, the default shell on most Linux distributions (including WSL); also available on MacOS X[5].
- But there are others[6], both older (Bourne, Korn, ...) and newer (Fish, Nu, ...), simple or feature-rich, traditional or exotic, etc.

Warning: generally, code written for shell X **won't work** in shell Y ⇒ it matters which one we use.

[5] Though the default version is somewhat old

[6] `$ cat /etc/shells` shows you which shells are available on your system.

Course Approach

- The only way to learn programming is, well, to program…
- However, a solid grasp of fundamentals is essential.

We'll start with **fundamentals**, mixing theory and exercises in the (interactive) shell.

Once we have this under our belt, we'll move to a **coding project** where we develop an actual (reasonably) useful script for bioinformatics.

Disclaimer #1

I may simplify things a little for the sake of brevity and/or clarity.

For example, I might say:

> *"Word splitting occurs on unquoted expansions".*

instead of:

> *"Word splitting occurs on the result of unquoted expansions, except in assignments, unless assigning to an array".*

Which would be closer to reality (but still not entirely accurate).

Disclaimer #2

I may occasionally use a term informally before formally defining it.

For example, I have already used the word “command” even though we haven’t discussed commands in the shell grammar yet. This is OK when intuition is enough to understand the point.

Part I - Below the Surface

WARNING: Lots of Theory Ahead!

- For interactive use, a basic knowledge of the shell may well be enough…
- … not so for programming, which makes use of more advanced concepts.
- Arguably most hair-pulling bugs[7] come from our[8] not-so-complete understanding of what the shell actually does with our input.
- **Don't** try to *learn* all this material - rather, be aware of it.
- **Do** follow along in your terminal!

[7] E.g. when you yell at the computer and threaten it with defenestration...

[8] I include myself here, of course

Where does the Magic Happen?

Bash does much more than launch programs:

- the globbing in `ls *.pdf` is done by Bash, not by `ls`.
- the expansion in `echo $PATH` is done by Bash, not by `echo`.
- the redirection in `grep Apis < dna.seq` is done by Bash, not `grep`.

What the Shell Does behind the Scenes

Besides launching programs, a modern shell, among other functionalities:

- Provides variables.
- Can do arithmetic.
- Performs filename globbing (`*.fasta`, etc.).
- Allows redirection of I/O (`>`, `<`, `|`, etc.).
- Implements a history mechanism.
- Supports auto-completion (`<TAB>`).
- Has flow control structures.

$\rightarrow$ Most of the power of Bash comes from the above features.

For Programming

Not all of the above are relevant to programming, however.

We'll focus on:

- Variables and how to use them to store, retrieve, and process data;
- Input and output;
- Control Structures;

because these aspects are the most relevant.

Shell Operation: The Gist

Broadly speaking, the shell does the following:

1. Reads input (terminal, file, ...)
2. Splits the input into *tokens*.
3. Parses tokens into commands.
4. Performs expansions (arithmetic, parameters, etc.).
5. Removes quotes.
6. Performs redirections.
7. Executes the commands.
8. Goes back to pt. 1.

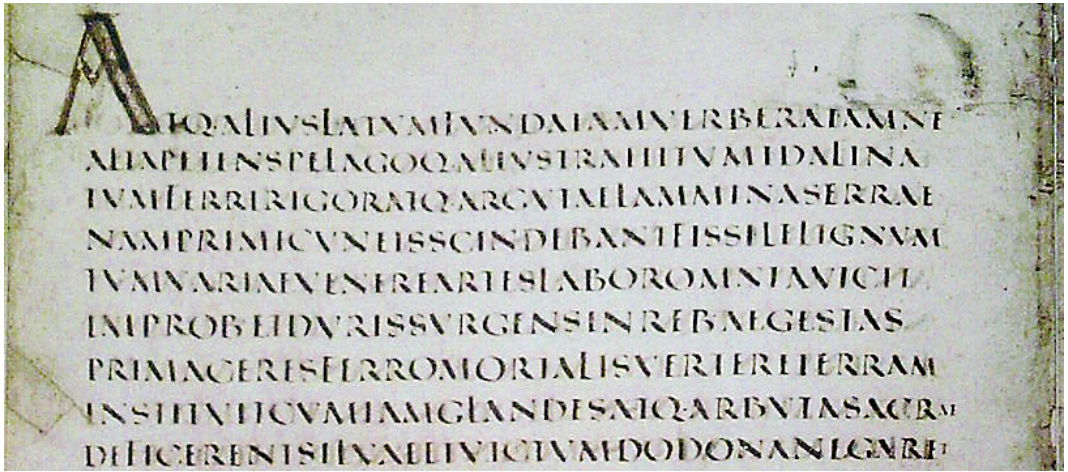
We'll survey 2-7. See the manual for details.

Step 2: Tokenizing

Natural language analogy: recognize words (in speech or text)

- We do this effortlessly, without realizing our brain does any work.
- But this involves significant work: consider foreign languages or hard-to-read texts.

Scriptio continua has no spaces or punctuation… → it's now obvious that recognizing words requires work.



Tokenizing in Bash

Bash (like all programming languages) also needs to tokenize its input.

- The input is split into *tokens* according to special[9] characters called *metacharacters*: whitespace (`space`, `tab`, `newline`) or any of `| & ; ( ) < >`
- Whitespace *separates* tokens (hence is never *part* of tokens).
- Tokens consist wholly of either non-whitespace metacharacters (`|&;()<>`) or non-metacharacters — they're called *operators* and *words*, respectively.

[9] In the sense that they have a special meaning to the shell

The **tokenizing procedure** looks like this:

1. Split on whitespace metacharacters.
2. Split the resulting tokens on metacharacter/non-metacharacter boundaries.

Examples:

```
ls -l | wc -l >> count
ls -l|wc -l>>count
ls-l|wc-l>>count          # -> Error !
```

Lines 1 & 2: 7 tokens (5 words, 2 operators), line 3: 6 tokens (first token is `ls-l`).

[metacharacters: (`space`, `tab`, `newline`) `|` `&` `;` `(` `)` `<` `>`]

Literal Characters

Why don't these commands work?

```
cat data/my file.txt           # WRONG!
echo it is a fact that 3 > 2   # Surprise!
```

Literal Characters

Why don't these commands work?

```
cat data/my file.txt           # WRONG!
echo it is a fact that 3 > 2   # Surprise!
```

... because the space and > are metacharacters, but here we want them to stand for themselves: that is, to be *literal*. For this, we use *quoting*.

Quoting

Quoting removes any special meaning of characters[10]. The main forms are:

- \ (backslash): the next character becomes literal (except at end of line: wrap long commands).
- `''` (single quotes): all characters between `''` become literal (including \, so cannot include `'`).
- `""` (double quotes): all characters between `""` become literal **except** `$`, `` ` ``, and \ (only before `"`, \, `$`, `` ` ``).

[10] There are special characters other than metacharacters, *e.g.* `$`, `*`, `? !`, etc.

Quoting - Showing Arguments

The following function[11] simply shows its arguments one per line[12] (type this in your terminal):

```
function showa() {
  printf "%d args\n" "$#"
  printf "%s\n" "$@"
}

# Can be written in shorter form:
showa(){ printf "%d args\n" "$#"; printf "%s\n" "$@";}
```

[11] We'll study functions on day 2.

[12] Imperfectly, because *e.g.* expansion, redirection etc. still happen normally

Quoting - Examples

```
cat > myfile     # Ok
cat > my file    # WRONG
showa my file
showa my\ file   # 1st form
showa 'my file'  # 2nd form
showa "my file"  # 3rd form
```

Quoting - '' vs ""

The important difference between '' and "" is that the latter allow *expansions* to occur (because $ retains its special status):

```
name=Bond
echo "my name is $name"  # Expansion
# -> my name is Bond

echo 'my name is $name'  # No expansion
# -> my name is $name
```

Practice

→ **Exercise 1.3** - tokenizing and quoting.

Step 3: Parsing Commands

Natural language analogy: recognizing *grammatically correct sentences*. This assumes words have been properly identified:

Utterance	Status
myho vercr afti sful lofe els	words wrong, grammar *undefined*
hovercraft is my eels of full	words ok, grammar wrong
my hovercraft is full of eels	words ok, grammar ok

→ The shell parses *commands* instead of sentences.[13]

[13] For details about the grammar, see its official specification

Examples of commands

```
ls -l > out                # Simple command - familiar
NAME=Hamlet                # Simple command
ls | wc -l                 # Pipeline
mkdir mydir && cd mydir # List

# Compound command
if grep needle < haystack; then echo found; fi
```

Lists

A *list* consists of one or more pipelines separated[14] by any of the following operators:

- `&` – execute job in the background (prompt returns)
- `&&` – execute next command iff previous *succeeds*
- `||` – execute next command iff previous *fails*

```
long_job & very_long_job & # background
ls myfile || touch myfile  # or use -f (see day 2)
grep baggins LotR && echo found || echo not found
```

`&&` and `||` allow simple conditionals (see day 2).

[14] `&` may occur at the end of a command.

Practice

→ **Exercise 1.4** - command parsing.

Step 4: Expansions

After parsing tokens into commands come the **expansions**, *i.e.* the **replacement of expressions with values**, in this order:

1. Brace expansion: `{1..10}`, etc.
2. Left to right:
   - Tilde expansion, *e.g.* `~/Desktop`
   - Parameter expansion: `$USER` & similar
   - Command substitution: `$(date)`, etc.
   - Arithmetic expansion, *e.g.* `$((2+4))`
   - Process substitution: `<(cmd)`
3. Word splitting (different from token splitting!)
4. Filename expansion ("globbing": `*.txt`)

Brace Expansion: `{..}`, `{,}`

Used to generate sets of strings based on:

- Comma-separated strings: `file_{A,B,C}.txt` → `file_A.txt file_B.txt file_C.txt`
- A sequence: `sample_{1..9}` → `sample_1` ... `sample_9`

This is **not** the same as file globbing: the generated strings do not have to be the names of existing files.

Brace Expansion - Examples

```
echo {1..100}  # E.g. in loops (see below).
echo {a..j}    # Works on chars.
echo {10..1}   # Works in reverse.

# Create a project tree (note nesting)
mkdir -p my_project/{src,doc/{mail,ref},data}

# {} at same level -> ~ Cartesian product
echo {A..D}{1..3}
```

Tilde Expansion

This is the well-known replacement of ~ by directories:

Tilde expression	Expansion
`~`	`$HOME`
`~alice`	Alice's home, probably `/home/alice`
`~+`	`$PWD`

... and a few others that address the directory stack; we won't say more about them because they're mostly relevant for interactive use.

Parameter Expansion

An unquoted $ followed by a parameter name is replaced by the parameter's value

```
place=Rovaniemi
echo $place
# -> Rovaniemi

echo "I'm off to $place"
# -> I'm off to Rovaniemi
```

There is **a lot** more to parameter expansion than this. We'll come back to it later on.

Command Substitution

`$(...)` substitutes the output of a command[15]

```
echo "Today is $(date -I)"
dirsize=$(du -s .)
nb_files=$(ls | wc -l)
```

An older form uses *backticks*:

```
echo "it is now `date`"
```

They're the same, but the modern form is easier to *nest*:

```
parent_dirname=$(basename $(dirname $PWD))
```

[15] Not to be confused with arithmetic expansion, `$((...))`.

Arithmetic Expansion

This doesn't work as expected:

```
a=2; b=3; echo $a+$b
```

This does:

```
a=2; b=3; echo $((a+b))  # Note: no $ needed
```

The expression between `$((...))` is evaluated using *shell arithmetic* - more about this later. Do not confuse with `$()`.

Process Substitution

Replace a filename argument with the output of a command.

Example: What items are common to two lists?

```
sort spc-list-1.txt > spc-list-1-sorted.txt
sort spc-list-2.txt > spc-list-2-sorted.txt
comm spc-list-1-sorted.txt spc-list-2-sorted.txt
rm spc-list-1-sorted.txt spc-list-2-sorted.txt
```

With `<(...)`, the comparison can be done *on the fly*:

```
comm <(sort spc-list-1.txt) <(sort spc-list-2.txt)
```

→ No temporary files, possibly faster.

Word Splitting

The **results** of **expansions** that **did not occur within double quotes** then undergo *word splitting*.

```
name='Ursus arctos'
showa '$name' # '': no expansion at all
showa "$name" # "": expansion, NO word splitting
showa $name   # expansion, word splitting
```

Why split on words?

Historical aside

- Early shells had no types other than strings.
- To store a *list* of values, programmers used whitespace:

```
dirs='bin doc new'
# wouldn't work w/o word splitting
mkdir $dirs
```

- Problem: what if file names contain spaces?

Solutions (sort of)

- Bourne shell (`sh`, 1976): word-split unless within `""`.
- Korn shell (`ksh`, 1983): new data type (arrays) for lists, otherwise same behaviour[16]; syntax for "all elements" rather clunky: `"${array[@]}"`
- Bash, v. 2 (1996): same as `ksh`.
- Zsh (`zsh`, 1991): prefer arrays for lists, word-splitting only if required[17], cleaner syntax for all elements.
- Friendly interactive shell (`fish`, 2005): no word splitting — all variables are arrays.

[16] Let's not break existing code

[17] If this breaks existing code, so be it!

IFS - Internal Field Separator

Word splitting uses the characters in the the `IFS` (*Internal Field Separator*[18]) parameter (by default, `<space><tab><newline>`)[19] as word delimiters.

The value of `IFS` can be changed:

```
line='gene_name,seq_len,mol_wt'   # CSV-like
showa $line                       # 1 field
IFS=','; showa $line; unset IFS   # 3 fields
```

[18] Hence the alternative (and better) term *field* splitting.

[19] Contrary to splitting into tokens, which uses whitespace and metacharacters.

IFS - Internal Field Separator (continued)

If `$IFS` is null, no splitting is done. If unset, the above default is used.

→ To reset the value of `IFS`: `unset IFS`.

Filename Expansion ("globbing")

Words that contain **unquoted** `*`, `?`, or `[` are *pattern-matched* against the files in the current directory.[20]

Wildcard	meaning
`?`	any 1 character
`*`	any string, including ''
`[...]`	any character between `[]`[21]

[20] This is done after parameter expansions so that globs can be stored in variables, *e.g.* `glob='*.pdf *.docx'; ls $glob`.

[21] There are predefined classes, *e.g.* for letters, digits, punctuation, etc.

```
# No quotes -> globbing occurs.
ls *.pdf

# Quotes -> no globbing occurs.
ls '*.pdf' "*.pdf" \*.pdf
echo "$glob" # Still quoted -> no globbing

# When a variable is set, quotes not needed
glob=*.pdf
```

In the last example, parameter expansion of `"$glob"` yields `"*.pdf"` (not `*.pdf`), so no globbing occurs. The quotes are removed in the next stage.

Expansions can be Mixed

```
pattern=*.md
echo "Markdown files:" $(ls $pattern)
```

This mixes parameter expansion, command substitution, word splitting, and filename globbing.

Practice

→ **Exercise 1.5** - parameter expansion.

→ **Exercise 1.6** - brace expansion and command substitution.

→ **Exercise 1.7** - process substitution.

Step 5: Quote Removal

After all expansions have been performed, quotes (and backslashes) are removed (unless they are quoted or result from an expansion):

```
echo 'recA' "dnaK"  # "," removed
echo d\'Artagnan    # \ removed
echo "a 'quote"     # ' retained: quoted
q=''"'"; echo "$q"  # middle " retained: expansion
```

This is why quotes usually disappear when we use `echo` or `showa`.

Step 6: Redirection

After all the expansions phase come the *redirections*, (the familiar >, >>, < etc.[22]

```
# Output of ls goes into list.txt (destructive!)
# Use >> to append
ls > list.txt
```

Note: some programs (like `tr`) only read data from STDIN, and hence **require** redirection.

[22] But not |, which is parsed in Step 3.

Step 7: Command Execution

Finally, after all these tokenization, parsing, various expansions, quote removal, and redirection steps, the command is ready to be launched. The shell requests the kernel to do so.

Recap

The main stages of input processing[23]:

2. Tokenizing
3. Parsing into commands
4. Expansions
   - 4.1 `{}`
   - 4.2 `~`, `${}`, `$()`, `$(())`, `<()`
   - 4.3 Word splitting
   - 4.4 Globbing
5. Quote Removal
6. Redirections

[23] The list starts at #2 to keep the numbering of the steps as used above

A Brief Point About Pure and Delegation Styles

- "pure" and "delegation" styles were introduced earlier (slide 11)...
- ... but to make the following points I needed some recently-introduced notions.

Task: count the nucleotides in a sequence file. The next slide shows pure (top) and delegation (bottom) styles:

```
#!/bin/bash
declare -Ai counter
while read line; do
    for ((i=0; i < ${#line}; i++)); do
        counter[${line:i:1}]+=1
    done
done
printf "A: %d\nC: %d\nG: %d\nT: %d\n" \
    ${counter[A]} ${counter[C]} ${counter[G]} ${counter[T]}
```

```
#!/bin/bash
tee >(printf "A: %s\n" $(tr -dc A | wc -c) >&2) \
    >(printf "C: %s\n" $(tr -dc C | wc -c) >&2) \
    >(printf "G: %s\n" $(tr -dc G | wc -c) >&2) \
    | printf "T: %s\n" $(tr -dc T | wc -c) >&2
```

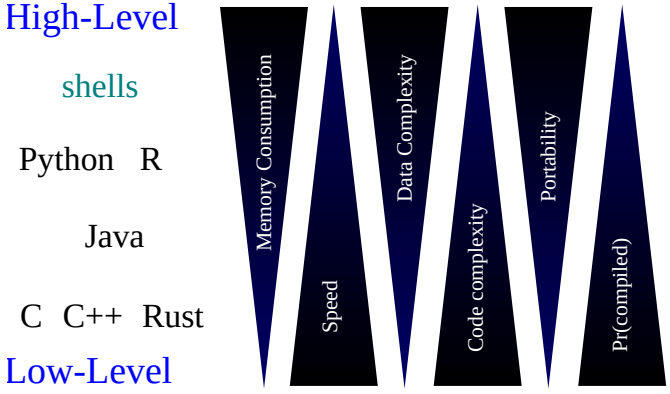
Now let's measure run times, using increasing numbers of lines.

```
$ time ../src/ntcount-pure.sh \
  < ../data/Bacteria_16S_nuc.seq
$ time ../src/ntcount-deleg.sh \
  < ../data/Bacteria_16S_nuc.seq
```

→ Delegation is *usually* the best approach.

Supplementary Slides

Situating Shells among Languages



Note: these are *trends*, not absolutes.

Ok, so what is automation good for?

- Saving time
- Preventing errors
- Ensuring reproducibility
- Avoiding boredom

Nevertheless:

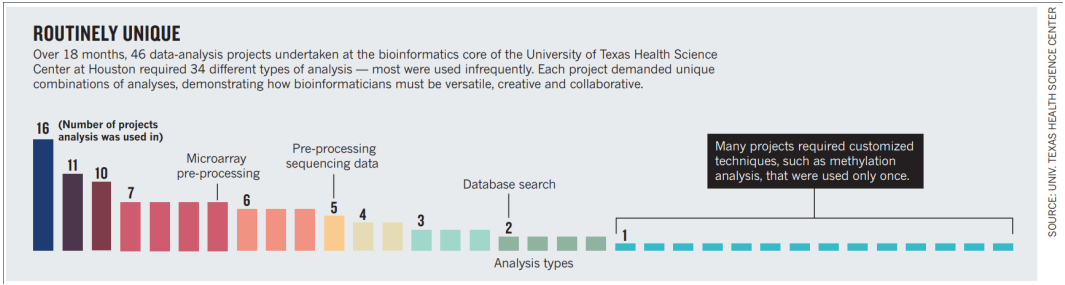
- Interactive tasks *can* be programmed in the shell (e.g. `bashtop`)...
- ... and so can serious computations.
- Whether they *should* be is another question.

Delegation Style - Wall Metaphor

By Pawel Wozniak - Wikimedia Commons

Will I need scripting skills?

Will I need scripting skills?



Chang J., *Nature* **520** (2015)

- 14 techniques ($> 40\%$) used in only one project.
- Most frequent technique used in $< 35\%$ of the projects.
- 79% of techniques used in $< 20\%$ of the projects.

“Hidden” Shell Scripts

Besides your analysis pipelines, you may also need to write some shell code for a number of tasks, such as:

- Build systems (Make, Snakemake).
- HPC scheduling jobs.
- Workflow management systems (Nextflow).
- Containers (Dockerfile).
- Test systems (Bats).

With spaces and punctuation, it's much better:

> *atque alius latum funda iam uerberat amnem*
> *alta petens, pelagoque alius trahit umida lina.*
> *tum ferri rigor atque argutae lammina serrae*
> (*nam primi cuneis scindebant fissile lignum*),
> *tum uariae uenere artes. labor omnia uicit*
> *improbus et duris urgens in rebus egestas.*
> *prima Ceres ferro mortalis uertere terram*
> *instituit, cum iam glandes atque arbuta sacrae*
> *deficerent siluae et uictum Dodona negaret.*

Publius Vergilius Maro, *Georgica*, Liber I, 141-149

This, very broadly speaking, is **tokenizing**.

Rush

(if time permits)

To illustrate that, let's try the most bare-bones shell of all:

- demo: `rush` - the **R**ather **U**seless **SH**ell.

Command	Result
`ls -l`	ok
`exit`, Ctrl-D	ok
`ls *.pdf`	doesn't work
`grep Bash description.md > result`	doesn't work
`ls | wc -l`	doesn't work
`cd`	doesn't work
`echo $PATH`	doesn't work
completion, history, arrows	don't work
`peak=Matterhorn; echo $peak`	don't work

Compound Commands

More on this later, but just to fix ideas:

```
if [ ! -d mydir ]; then mkdir mydir; fi
```

This is a *conditional statement* built around the simple commands `[ ! -d mydir ]` and `mkdir mydir`.

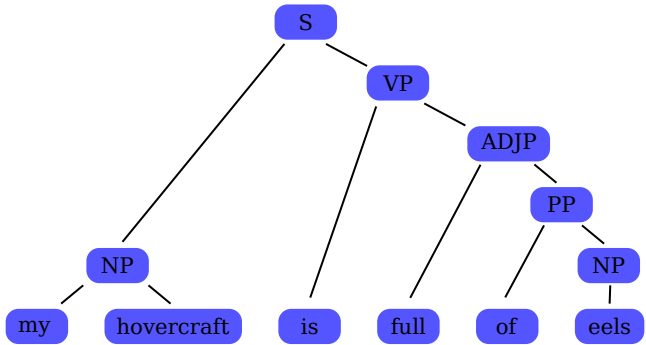
→ Compound commands start and end with *matching keywords* (`if` ... `fi`, `{` ... `}`, etc.)

In a Nutshell

Compound commands are made of lists, which are made of pipelines, which are made of simple commands... except that a compound command is *itself* a simple command: the grammar is *recursive*.

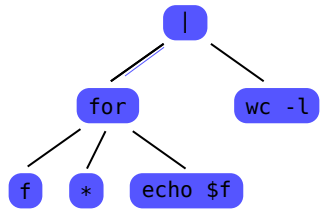
```
for f in *; do echo $f; done | wc -l
<--   compound command    -->
<--    simple command     -->   <-- spl cmd -->
<--         p  i  p  e  l  i  n  e          -->
```

Parsing a sentence converts a linear sequence of words into a tree-like structure:



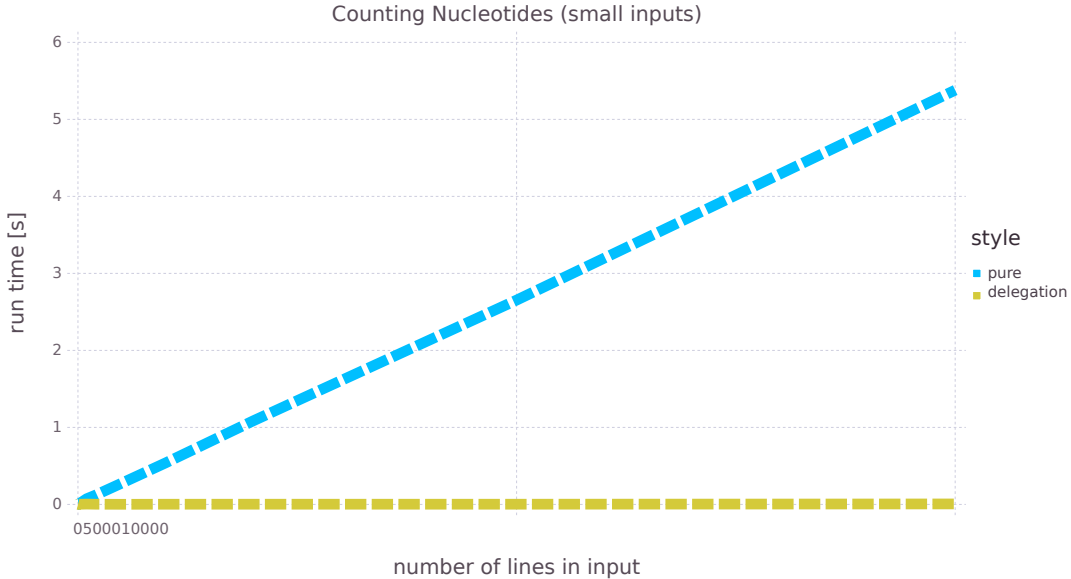
The tree's structure is constrained by the grammar of English.

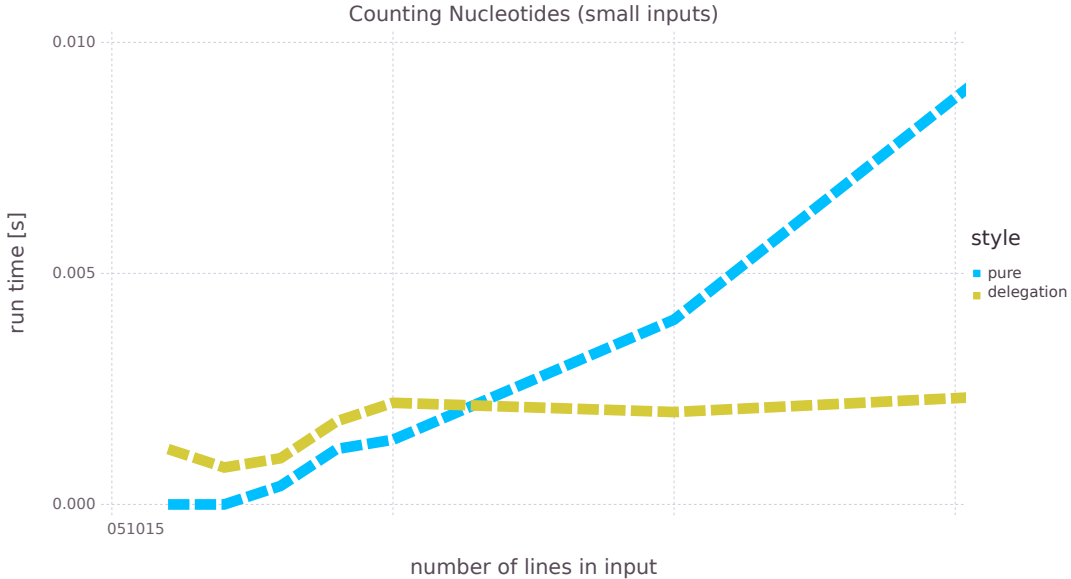
The same goes for Bash:[24]



If a command is syntactically wrong, **it does not parse**.

[24] And all other programming languages, for that matter


Counting Nucleotides (small inputs)
run time [s]
0
1
2
3
4
5
6
0500010000
number of lines in input
style
pure
delegation


Counting Nucleotides (small inputs)
0.010
0.005
0.000
run time [s]
0
5
10
15
number of lines in input
style
pure
delegation

Interpretation

- External programs are typically (much) faster than Bash.
- *Launching* an external program has a **cost** (overhead) .
- The "delegation" style launches programs, the "pure" style doesn't.
- The extra speed more than compensates for the overhead, *except for very small inputs* and/or numerous tasks.

Here Documents: <<

```
cat <<END
# Everything up to END goes to the input of cat;
# The end token can be any word, not just END
# Quoting prevents expansion.
END
```

Useful to store some multiline output within the script - see `src/welcome.sh`.

Here Strings: <<<

Useful for small inputs that can fit on the command line, *e.g.* measuring the length of strings:

```
# /!\ Includes newline!
wc -l <<< CATCGACATGCA
```