Automation: Filter-based Programming and Scripting

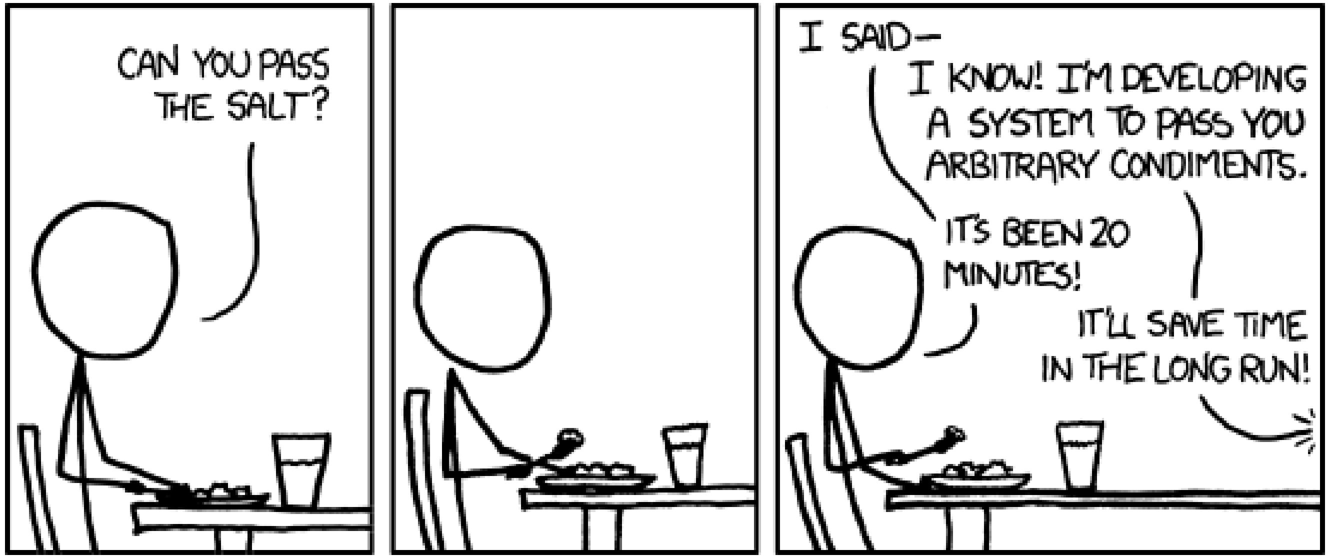
Philipp Reinecke

ReineckeP@cardiff.ac.uk

Goals

- Help you solve common problems that are too boring (or complex) to do manually
- … without re-inventing the wheel
- … without writing a huge program
- … and using whatever tools are already there.
- Examples:
  - Run repeated experiments
  - Sort, analyse, and plot a lot of experiment data
  - Analyse logfiles
  - Quickly prototype an idea

Goals (ctd.)


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

Source: www.xkcd.com, Creative Commons Attribution NonCommercial 2.5 License

- … and know when to choose a better approach.

The Unix Philosophy

Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that **the power of a system comes more from the relationships among programs than from the programs themselves**. **Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.**

Brian Kernighan and Rob Pike (1984)
(emphasis added)

This is the Unix philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

Write programs to handle text streams, because that is a universal interface.

Douglas McIlroy (formatting added)

https://en.wikipedia.org/wiki/Unix_philosophy

Examples

- Create time-stamped archive:

```
tar czf archive-`date | tr “[: ]” -`.tgz directory/
```

- Create remote log of running processes:

```
while (true); do
       date;
       ps aux;
       sleep 1;
done | netcat loghost 5555
```

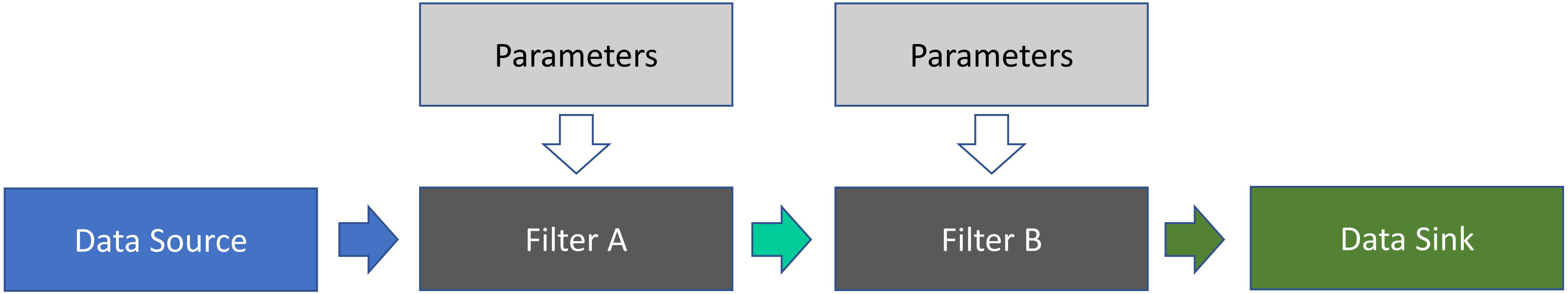
```
netcat –l 5555 > processes.log
```

What we are going to talk about

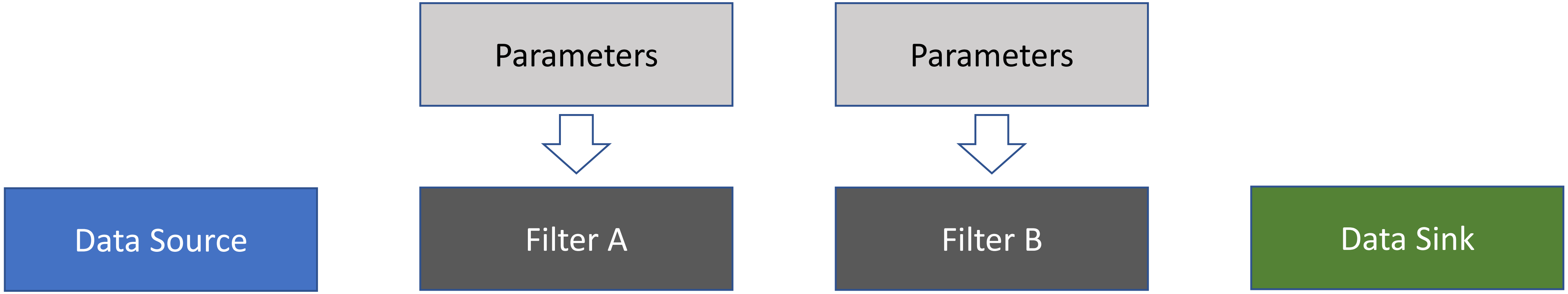
- Choose appropriate tools
- Glue them together
- Gluing:
    - Filters
    - Shell scripts
    - Scripting languages
- Useful tools

Filters

- Filter-based programming:
  - Processing of a stream
  - Input stream -> processing -> output stream
  - Filter is black box, can be controlled by parameters
- Idea from signal processing, e.g. low-pass filters
- Application in our context: Text files, records, lists, etc.
- Also used in other contexts
  - Functional programming – operate on list



Filters in Linux/Unix



Parameters

Parameters

Data Source

Filter A

Filter B

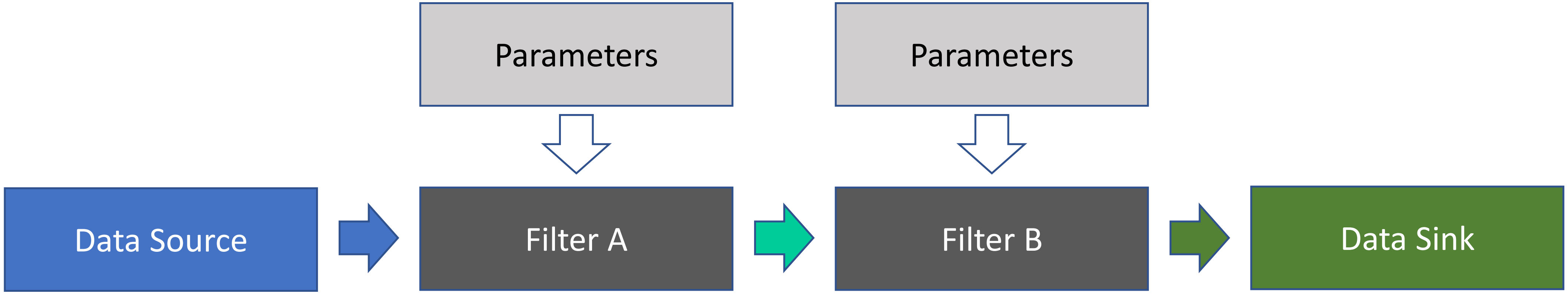
Data Sink

- Start of chain, no input
- “Data at rest”
- Standard input: STDIN (usually keyboard)
- Files
- Network
- Program

- Command-line programs
- Typically:
  - Use text input
  - Produce text output
- Only need to know what they do, not how
- Examples:
  - sort, uniq, grep
  - tr, sed, AWK

- End of chain, no output
- “Data at rest”
- Standard output: STDOUT (usually screen)
- Standard error: STDERR (usually screen)
- Files
- Network

Filters in Linux/Unix: Pipework



Input

- Read from file: <
```
sort < file.txt
```
- HERE document: <<
```
sort << EOF
Cardiff
Aberdeen
Bristol
London
EOF
```
- Read from command line: <<<
```
bc <<< 2*3*4
```

Between filters

- Pipe: |
```
sort < file.txt |
uniq |
wc -l
```
- Split: tee
```
sort < file.txt |
tee /dev/stderr
```
- Merge
```
sort < file.txt 2>&1
sort < file.txt 1>&2
```

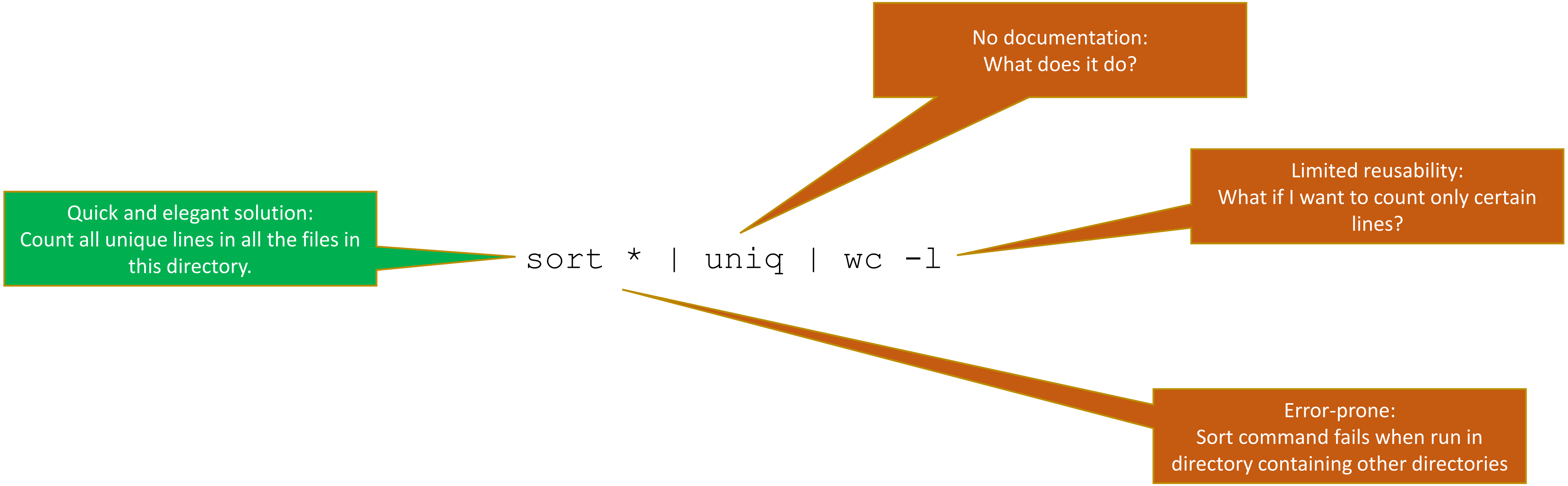
Merge STDERR into STDOUT

Merge STDOUT into STDERR

Output

- New file: >
```
sort < file.txt >
sorted-file.txt
```
- Append to file: >>
```
sort < file.txt >>
sorted-file.txt
```
- Redirect named stream
```
sort < file.txt 1>stdout.txt
sort < file.txt 2>stderr.txt
```

Filters: Advantages and Limitations



```
sort * | uniq | wc -l
```

Quick and elegant solution:
Count all unique lines in all the files in this directory.

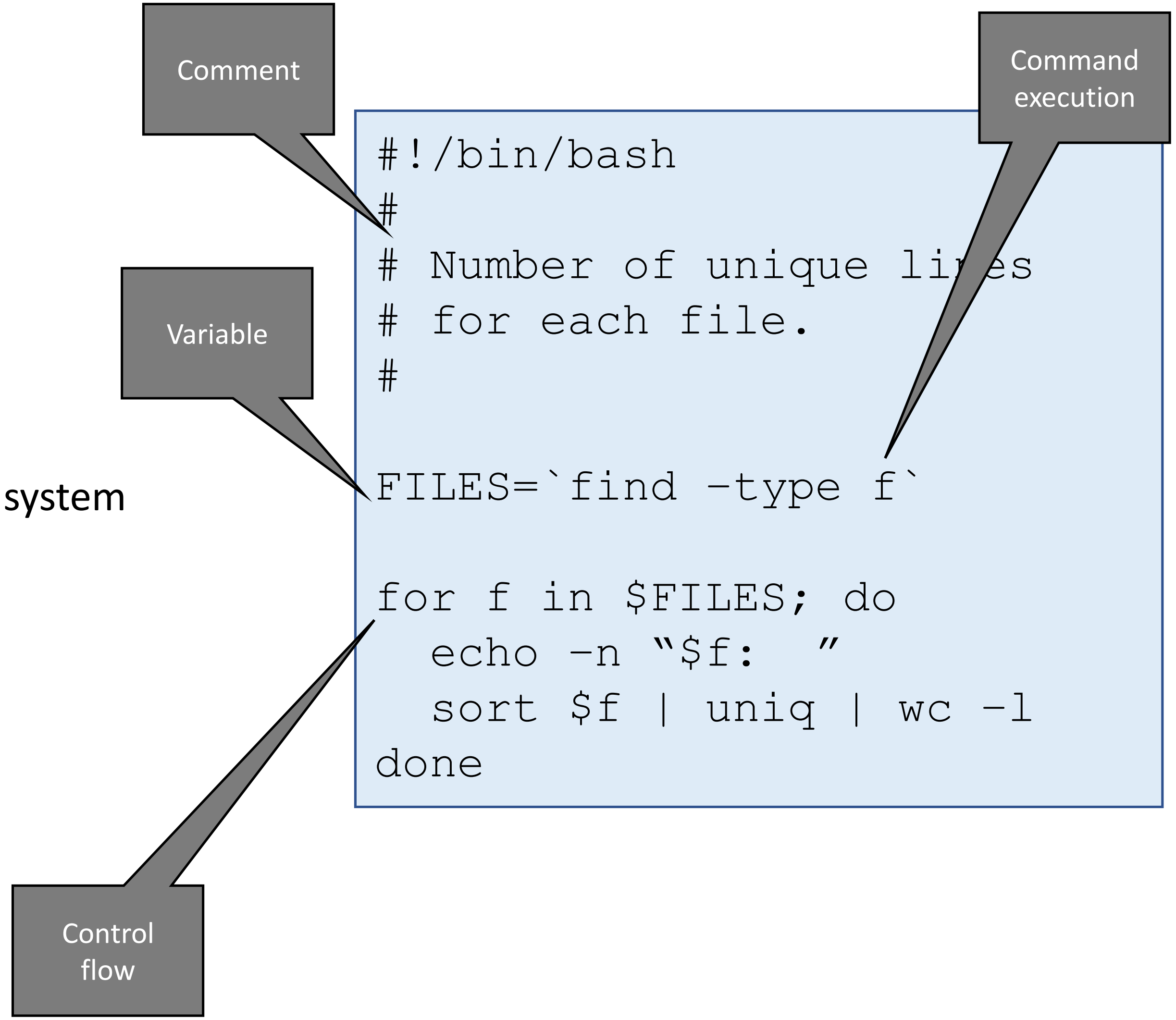
No documentation:
What does it do?

Limited reusability:
What if I want to count only certain lines?

Error-prone:
Sort command fails when run in directory containing other directories

Shell Scripts

- Little programs in text files
- Usually not compiled
- Executed by the Unix shell (e.g. sh, bash, csh)
- Usually imperative programming style
- The Unix shell:
  - Command-line environment for interaction with the system
  - Many different shells: sh, bash, csh, …
- Functionality:
  - Variables
  - Command execution
  - Expansion
  - Control-flow
  - Comments



```
#!/bin/bash
#
# Number of unique lines
# for each file.
#

FILES=`find -type f`

for f in $FILES; do
  echo -n “$f:  ”
  sort $f | uniq | wc -l
done
```

Shell: Expansion

- The shell replaces many parts of each line before execution
- Filename expansion: Replaced by filenames matching pattern
  - Any sequence of any length: *
  - Any single character: ?
  - Character from class: []
- Brace expansion: Replaced by all combinations
  - List: `preamble{x,y,z}postscript`
  - Range: `preamble{x..y[..incr]}postscript`
- Parameter: Replaced by variable value
  - Value: `${NAME}` or `$NAME`
  - Value without prefix: `${NAME#prefix}`
  - Value without suffix: `${NAME%suffix}`
- Command: Replaced by command output
  `$(command)` or `` `command` ``
- Arithmetic: Replaced by output of expression
  `$((expression))`
  - Simple arithmetics – integers only
- Order of Expansions:
  - Brace
  - Parameter
  - Arithmetic
  - Command
  - Filename

```
> ls
bob.txt bib.txt bob.doc bob1.txt bub.txt
> echo *.txt
bob.txt bib.txt bob1.txt bub.txt
> echo bob?.txt
bob1.txt
> echo b[o,i]b.txt
bob.txt bib.txt

> echo B{i,o,u,a}b
Bib Bob Bub Bab
> echo Bob-{1..10..2}
Bob-1 Bob-3 Bob-5 Bob-7 Bob-9

> VARIABLE="Bob the Builder"
> echo $VARIABLE
Bob the Builder
> echo ${VARIABLE#Bob}
the Builder
> echo ${VARIABLE%the Builder}
Bob

> FILES=`ls *.txt`
> echo $FILES
bob.txt bib.txt bob1.txt bub.txt

> echo $((304/2))
152
```

Shell: Control flow

- FOR loop: Iterate over all values in space-separated LIST

```
for P in LIST; do
      …
done
```

- IF/THEN/ELSE branch: Conditional execution

```
if CONDITION; then
      …
else
      …
fi
```

- WHILE loop: Repeat while CONDITION is true

```
while CONDITION; do
      …
done
```

- UNTIL loop: Repeat until CONDITION is true

```
until CONDITION; do
      …
done
```

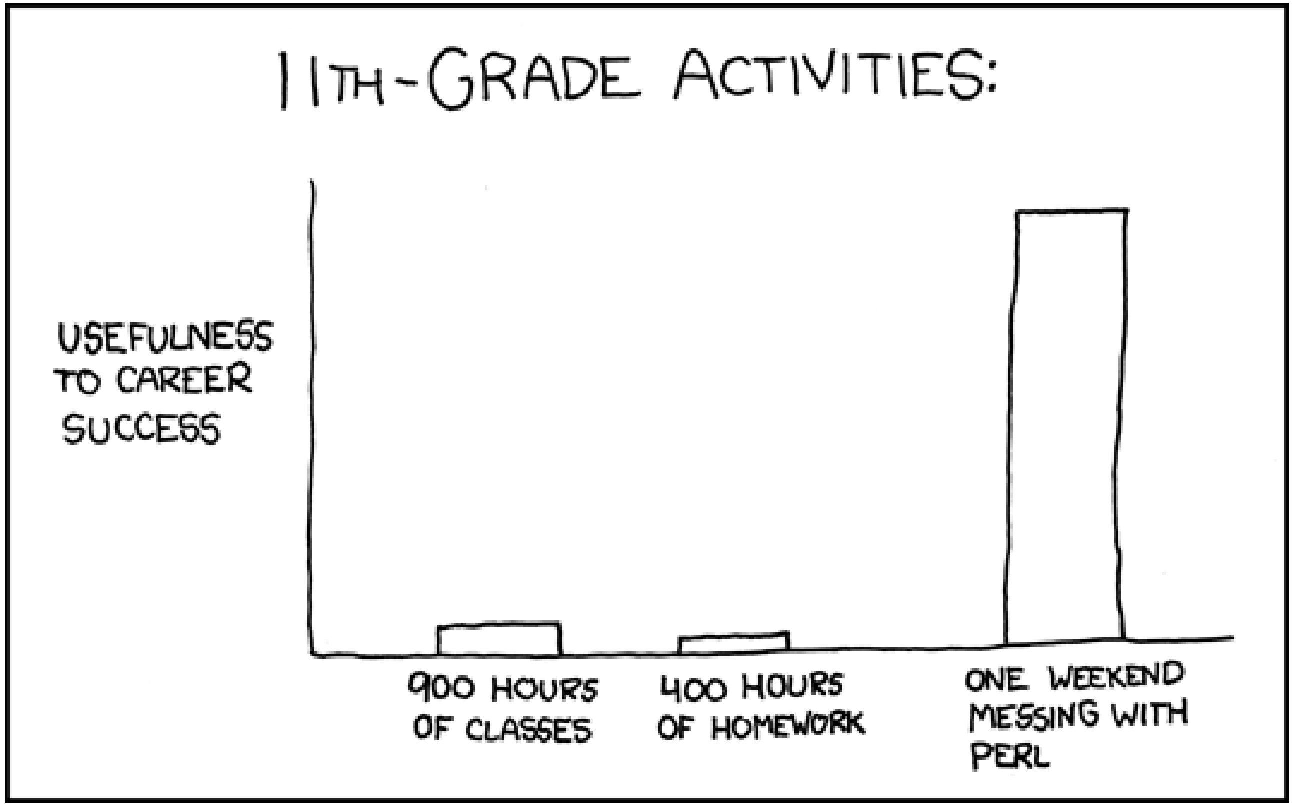
```
> for f in $FILES; do echo $f; done
bob.txt
bib.txt
bob1.txt
bub.txt
> if [ -e bob.txt ]; then echo Yes; else echo No; fi
Yes
```

```
> while [ -e bob.txt ]; do echo Yes; sleep 1; done
Yes
Yes
Yes
...

> until [ -e bub.doc ]; do echo No; sleep 1; done
No
No
No
...
```

Shell scripts vs. scripting languages

- Shell scripts:
  - Centred around the capabilities of the shell
  - Clumsy for complex tasks, especially with complex data and control flows
  - Clumsy for maths
  - Excellent for re-use of filters
- Scripting languages:
  - Historically, often derived from the shell
  - Can be overkill for easier tasks – use shell instead
  - Excellent for automating complex tasks
  - Many different languages: Perl, Python, Ruby, PowerShell (on Windows)



Source: www.xkcd.com, Creative Commons Attribution NonCommercial 2.5 License

Tim Toady

TIMTOWTDI

There Is More Than One Way To Do It

Perl: Variables

- Data types:
  - Scalars
  - Arrays of scalars
  - Hashes of scalars
- Variable type denoted by “funny characters”
- Multiple variables with same name (but different type) can exist
- Variables start to exist when accessed
- Types are cast as needed: Numeric-String-Numeric…
- Scalar: A single value
  - Funny character: `$`
- Array of scalars: Ordered list of values, indexed by integer
  - Funny characters: `@` and `[]`
  - Access: `$array[$index]`
  - Index of last element in array: `$#array`
- Hash of scalars: Unordered list of values, indexed by strings
  - Funny characters: `%` and `{}`
  - Stores key-value pairs
  - Access value for key: `$hash{$key}`
  - List of keys or values: `keys(%hash)` or `values(%hash)`

```
$x = “123”;
$y = 5 + $x;

@array = (1, 2, 5);
print $array[1];
print $#array;


%hash = ( “x” => 1, “y” => 2 );
print $hash{“x”};

print $hash{“z”};

print keys(%hash);

print values(%hash);
```

Perl: Operators

- Perl has all the usual operators, e.g. assignment, arithmetic, etc.
- Some special operators:
    - String concatenation: `.`
    - Short-hand assignment: `<operator>=`
        - `$x <operator>$y` is the same as `$x = $x operator $y`
    - Range: `x..y` or `x…y`
        - Produces list with values ranging from `x` to `y`

Caution: Not the whole truth. See p.103f in the Camel book, 3rd edition.

```
$x=5;

$x=5*3;

$s = “abc”;
$s = “abc” . “def”;

$x += 3;
$s .= “ghi”;

@a = 1..100
@s = ‘a’..`z’;
```

Scripting: Control

- Perl has the typical control structures of a programming language
- Loops:
  - `for ($x=1; $x<=25; $x++) {}`
  - `foreach $x (1...25) {}`
  - `while ($x <= 25) { $x++; }`
  - `until ($x == 25) { $x++; }`
  - Breaking out: `next/last`
- Branching:
  - `if ($x == 3) { } else {}`
  - `if ($x == 3) { } elsif { }`
  - `unless ($x == 3) { }`
  - `print “a” if $x == 3;`
  - `print “b” unless $x == 3;`

Input/output

- C-like input/output functions exist
- Output: print – like C
    - `print EXPR` prints to STDOUT
    - `print FILEHANDLE EXPR` prints to FILEHANDLE
    - File handles: STDOUT, STDERR, or your own
- File access:
    - open FILEHANDLE, EXPR opens file and assigns it to FILEHANDLE
    - Funny characters in EXPR:
        - “<name”: opens for reading only
        - “>name”: opens for writing only
        - “>>name”: opens for appending
        - “|program”: pipes output to program
        - “program|”: reads program’s output
- Reading a line from a file: <FILEHANDLE>
- Command-line
    - @ARGV: Command-line arguments
    - <> operator: Command-line arguments (if any), then STDIN

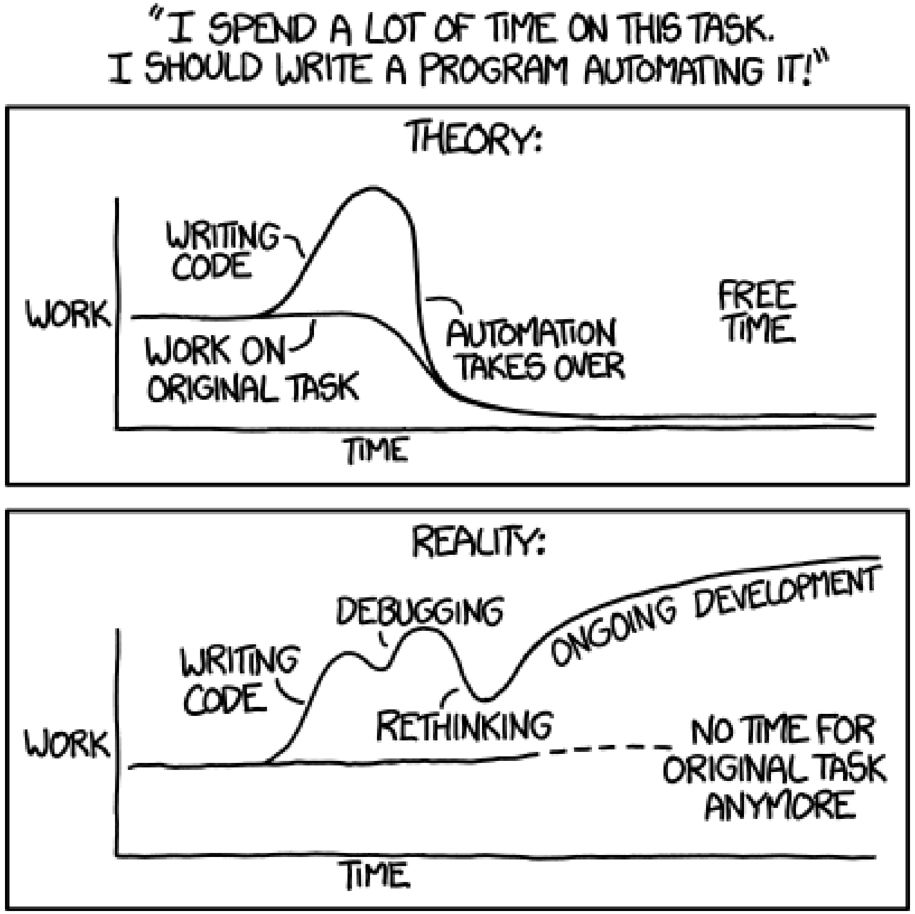
Best thing about Perl: Regular expressions

Next Week

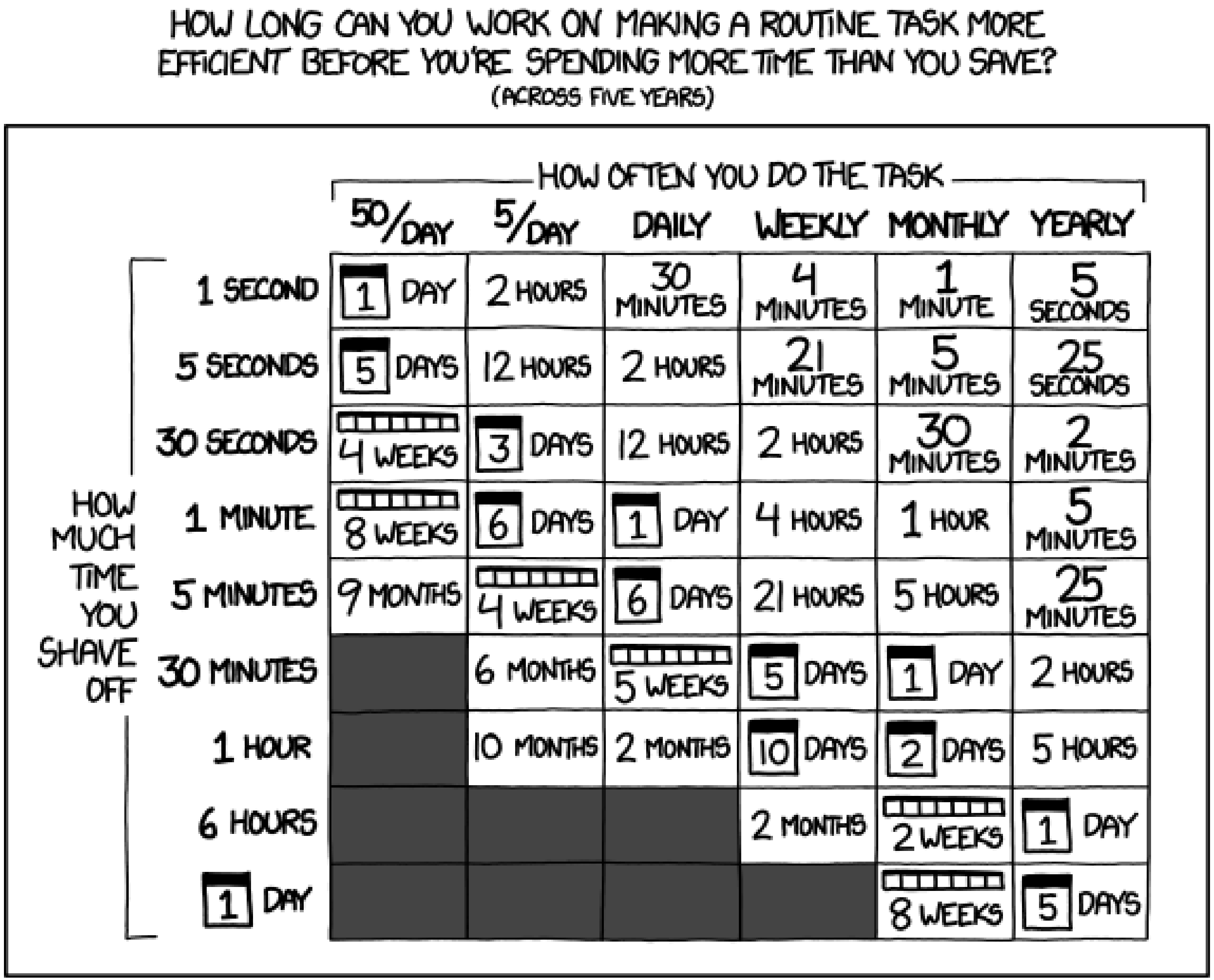
That is the glue, now for the tools

- `tee`
- `cat`
- `echo`
- `tail, head`
- `netcat`
- `wget, curl`
- `grep`
- `uniq`
- `sort`
- `wc`
- `bc`
- `ls, find, rm, mv, touch`
- `Gnuplot, R`
- `man`

- `tr`
  - `tr SET1 SET2`
  - Translate from one set to the other
- `sed`
  - `sed /expr1/expr2/`
  - Replace `expr1` by `expr2`
- `awk`
  - `awk { code }`
  - Execute `code` for each line
  - Useful for extracting columns:
    `awk { print $1 $3; }`



Source: www.xkcd.com, Creative Commons Attribution NonCommercial 2.5 License



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK					
	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
1 DAY	■	■	■	■	8 WEEKS	5 DAYS

Source: www.xkcd.com, Creative Commons Attribution NonCommercial 2.5 License

Conclusion

- Useful for automation
  - Filters
  - Shell-scripts
  - Scripting languages (Perl)
  - Useful tools
- Make sure you pick the right tool for the job
- Recommended reading:
  - William Shotts: LinuxCommand.org (last accessed 2019-01-28)
  - Larry Wall, Tom Christiansen, & Jon Orwant: Programming Perl (3rd edition)

- Note: All cartoons from www.xkcd.com, Creative Commons Attribution NonCommercial 2.5 License.