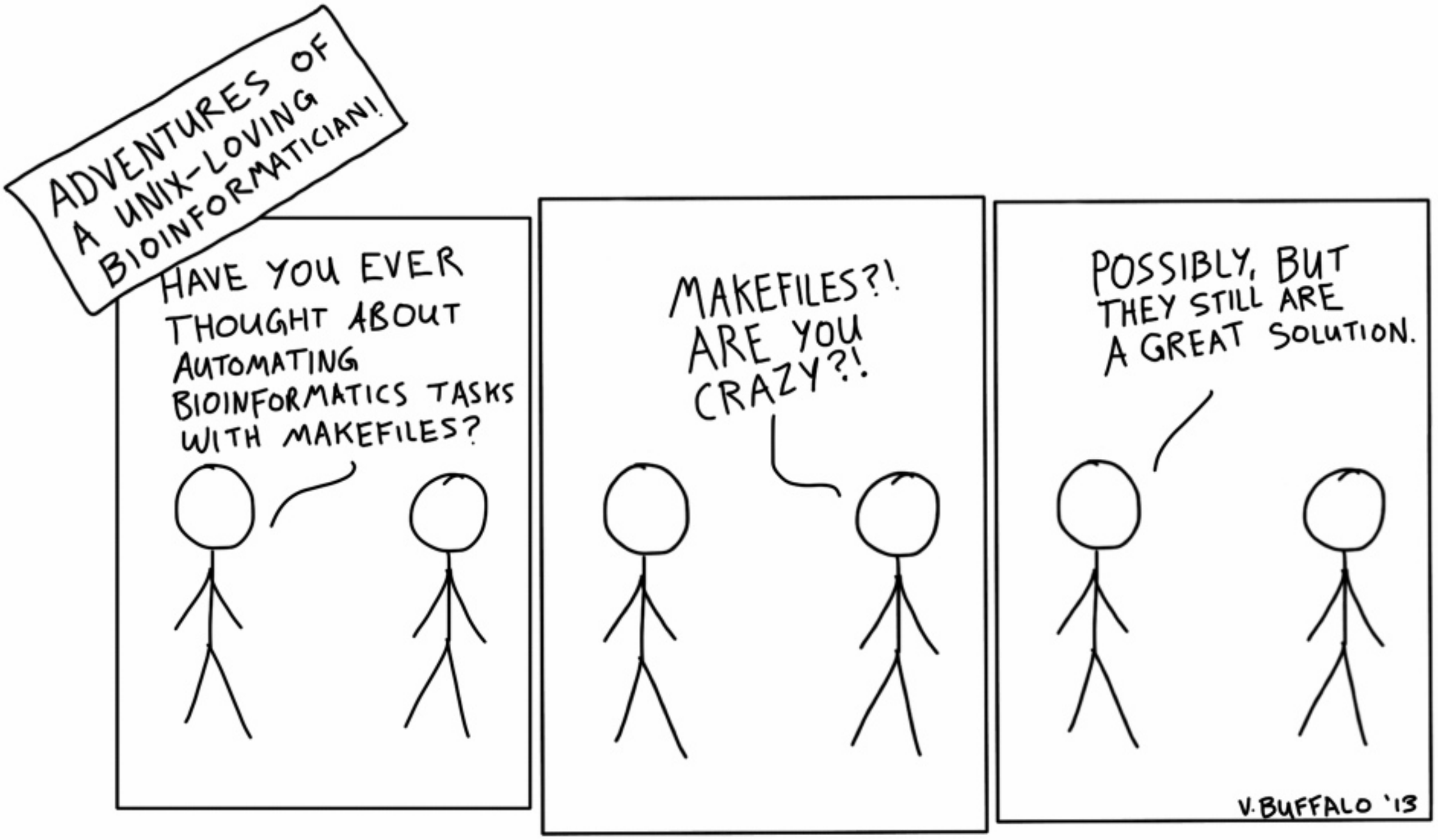
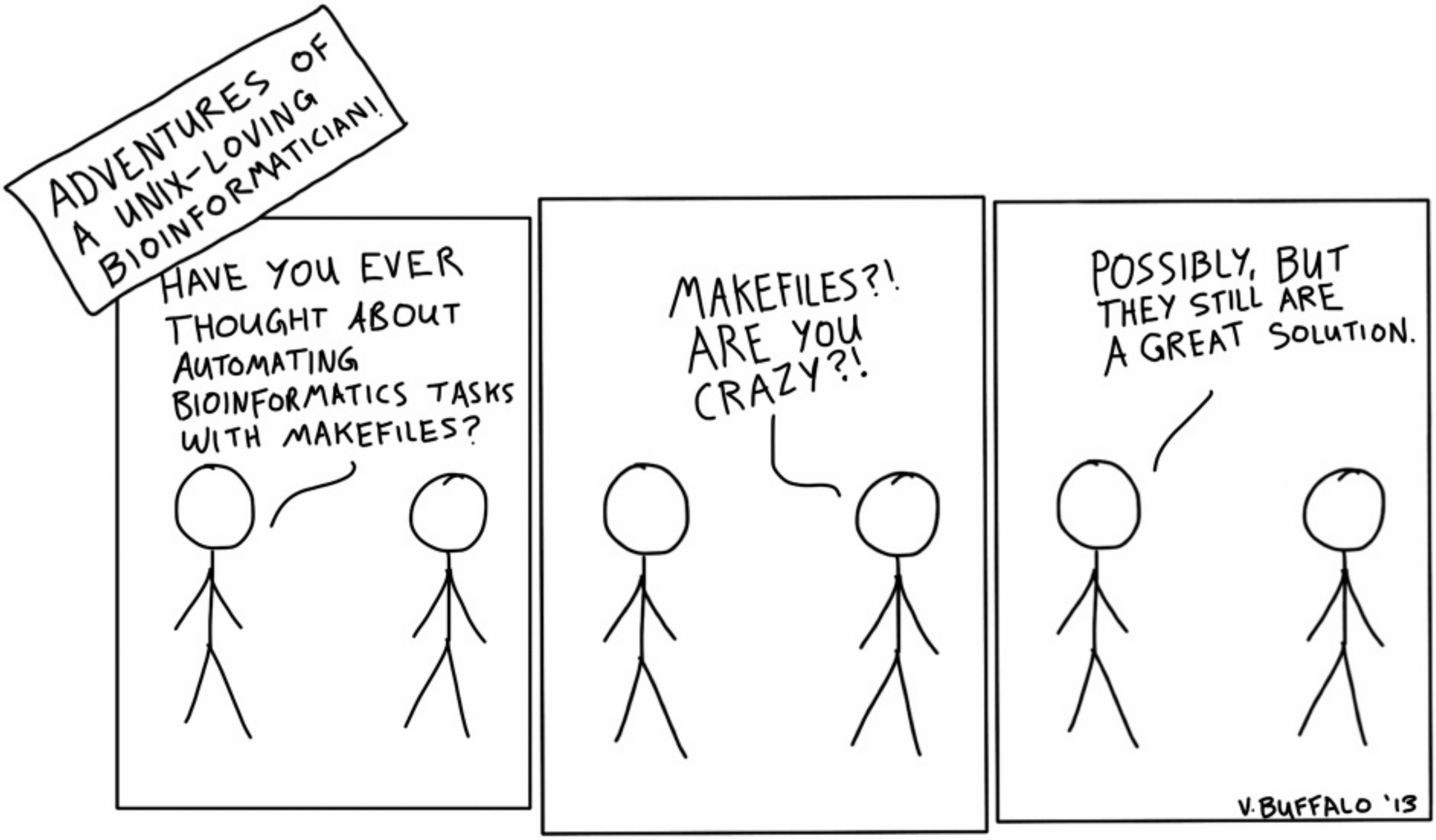
GNU

OR:
HOW
I LEARNED
TO STOP
REINVENTING
THE WHEEL
AND
LOVE
MAKE

Makefiles

Vince Buffalo, 2013


ADVENTURES OF A UNIX-LOVING BIOINFORMATICIAN!
HAVE YOU EVER THOUGHT ABOUT AUTOMATING BIOINFORMATICS TASKS WITH MAKEFILES?
MAKEFILES?! ARE YOU CRAZY?!
POSSIBLY, BUT THEY STILL ARE A GREAT SOLUTION.
V. BUFFALO '13



Part I: Why Makefiles aren't Crazy.

True Facts About Being a Bioinformatician

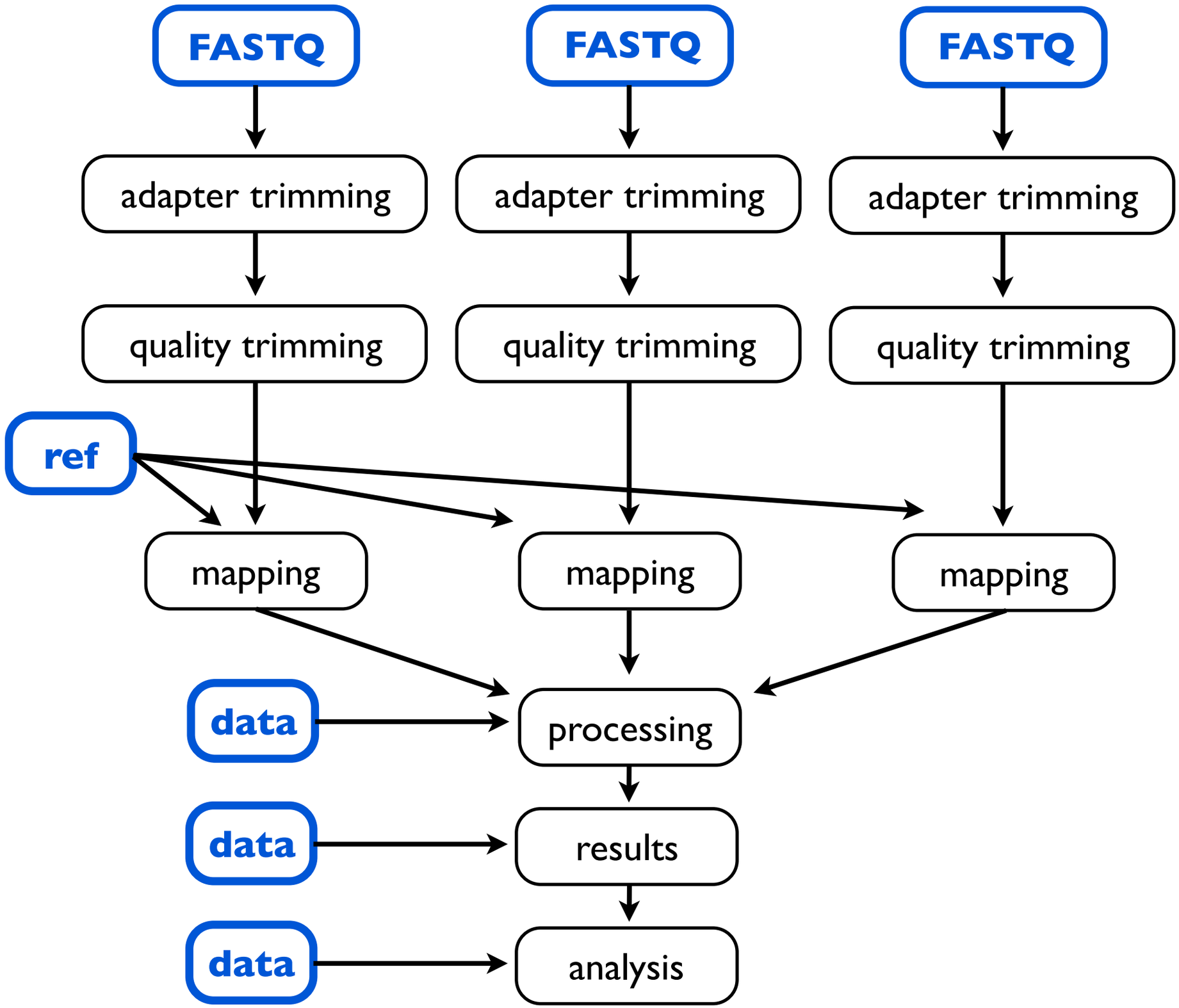
(1) You will almost certainly have to re-run an analysis more than once, possibly with new or changed data.

True Facts About Being a Bioinformatician

(1) You will almost certainly have to re-run an analysis more than once, possibly with new or changed data.

(2) In the future, you (or your collaborators, or PI) will almost certainly have to revisit part of a project and it will look completely cryptic. Your only defense is to document each step.

Bioinformatics Workflows



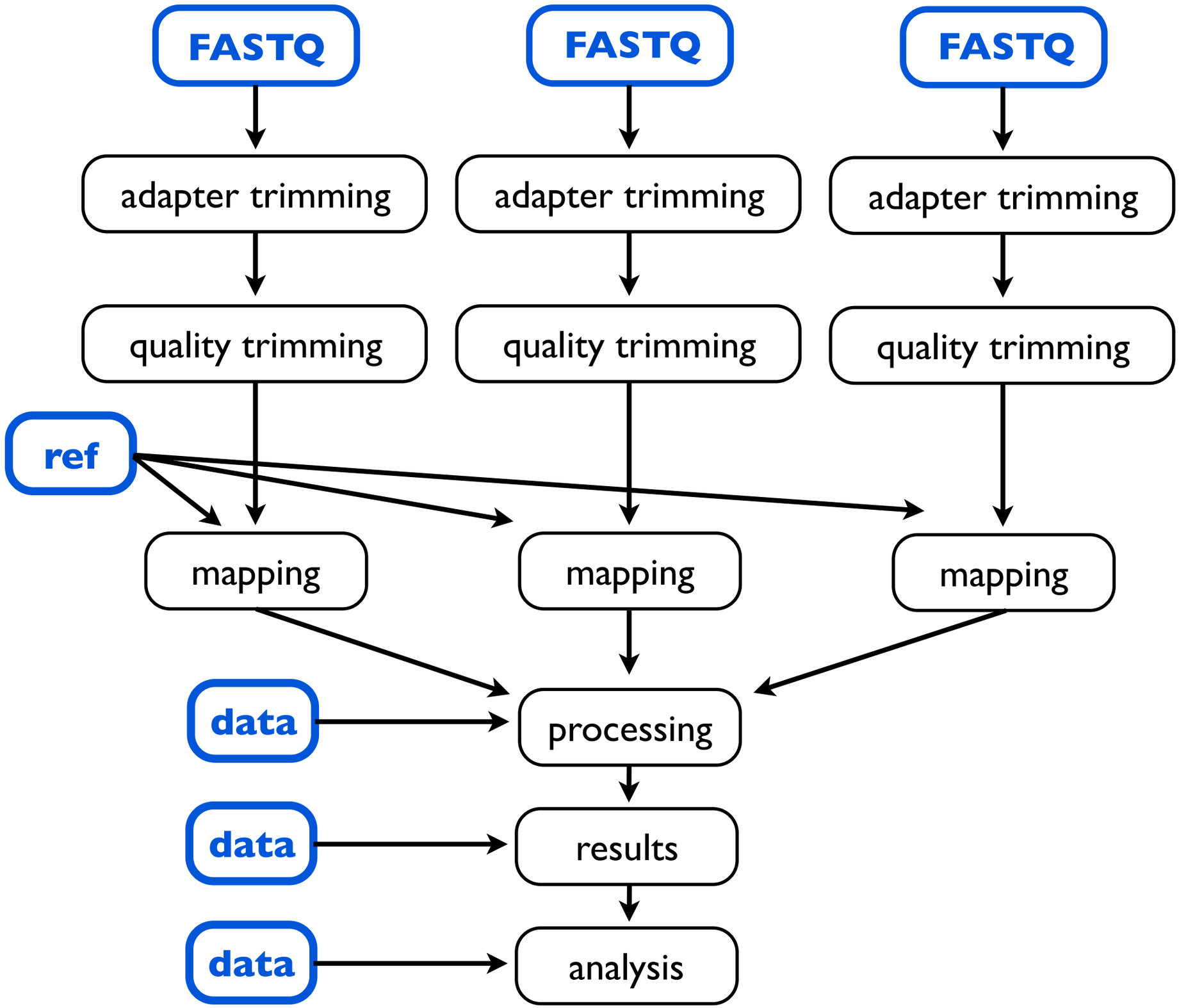
blue: data

Bioinformatics Workflows

Represented by a **Dependency Graph**

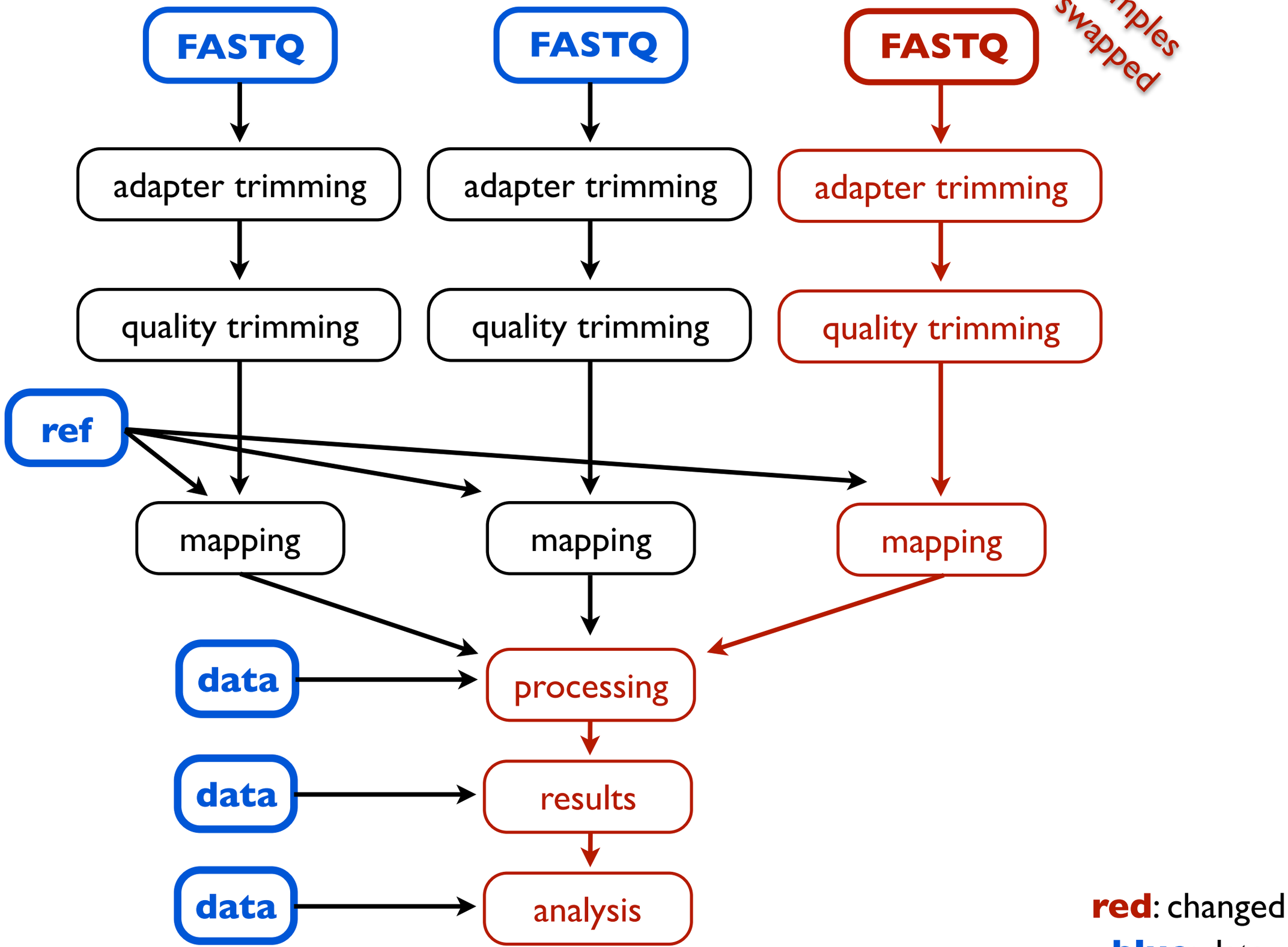
If we have to change some data, only changed parts need to be re-run.

Bioinformatics Workflows



blue: data

Bioinformatics Workflows



Bioinformatics Workflows

Only steps dependent on changed data need to be updated (and this is recursive).

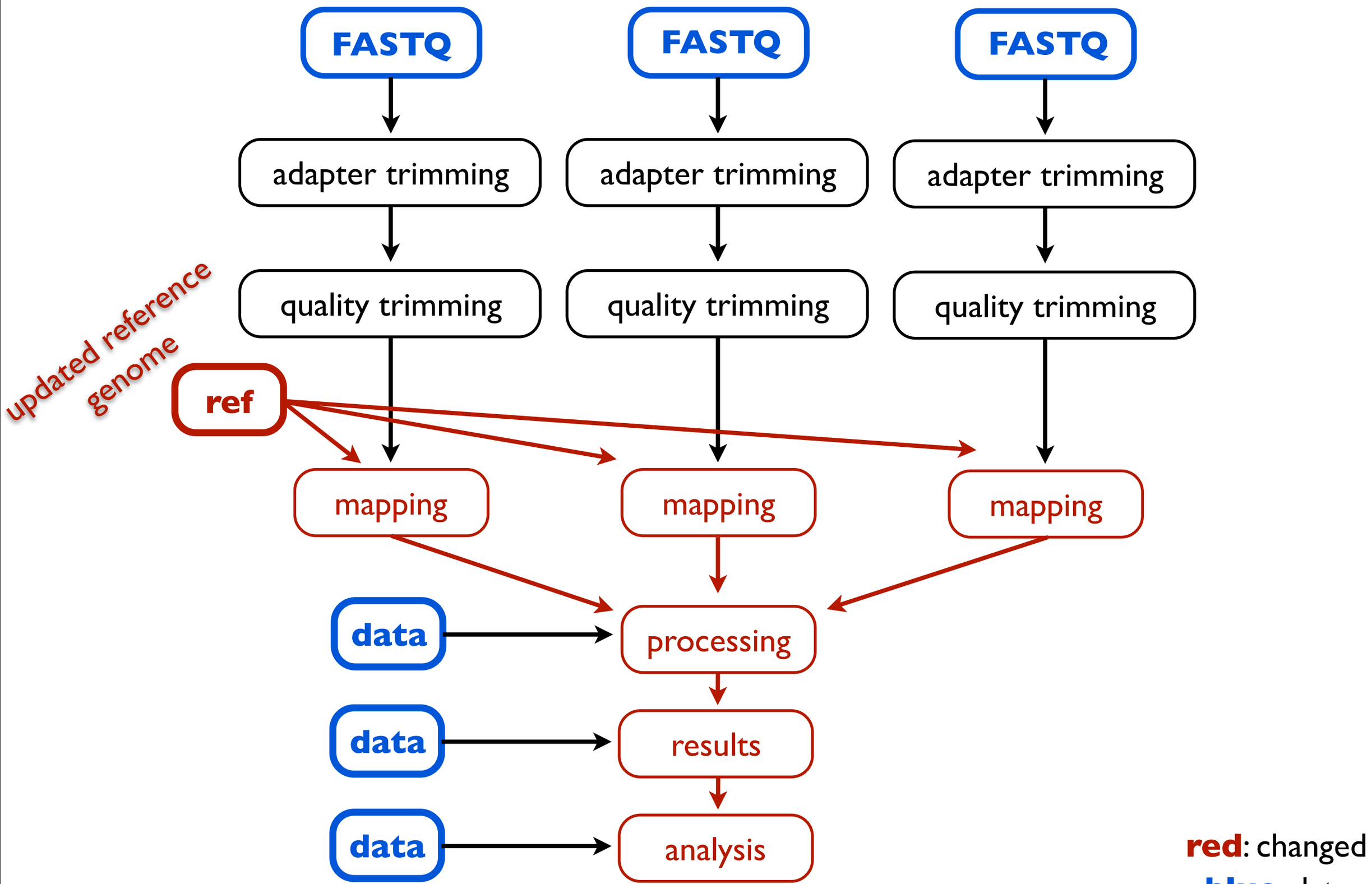
Bioinformatics Workflows

Only steps dependent on changed data need to be updated (and this is recursive).

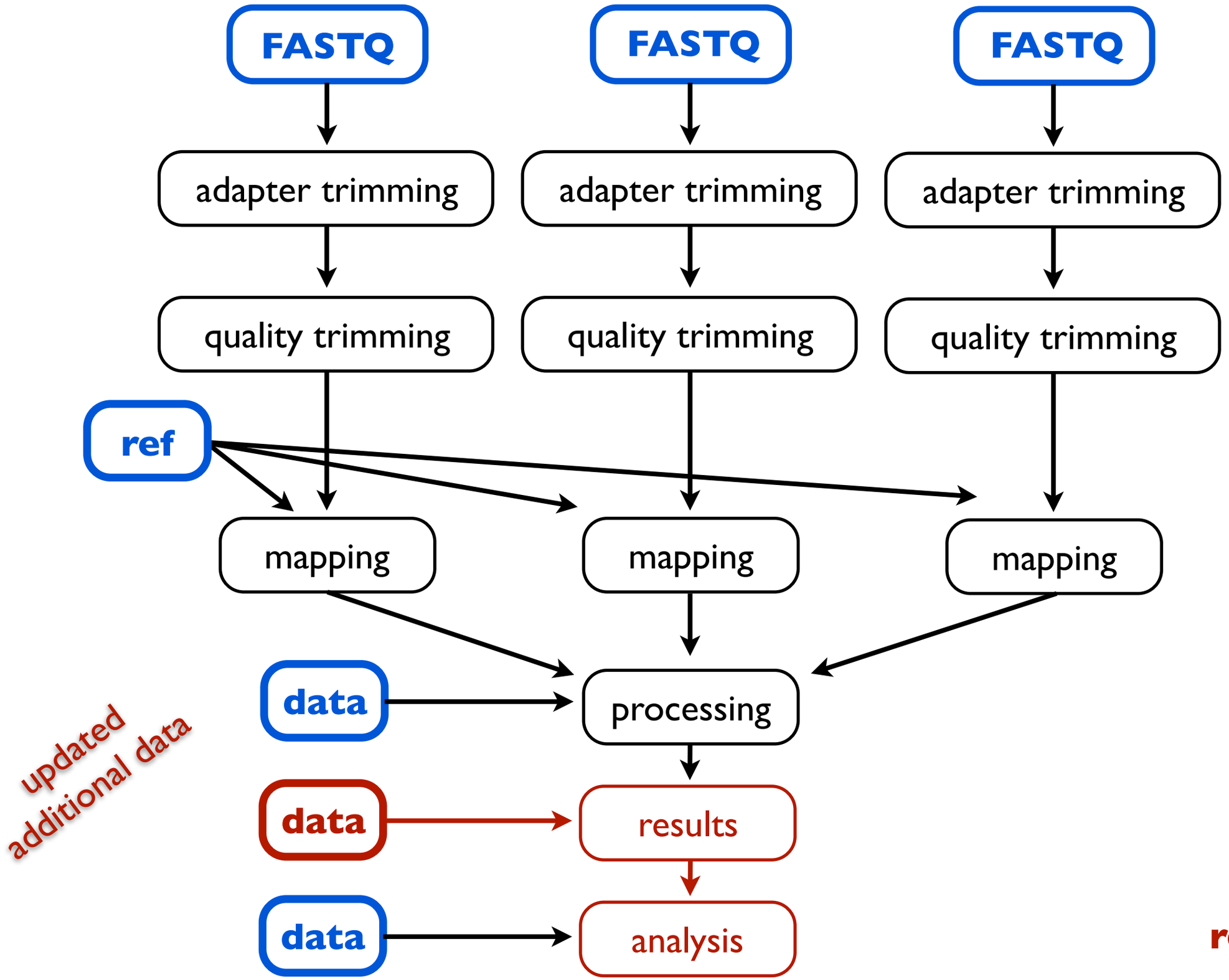
This applies for any **data**:

(1) input sequence data (FASTQ files)
(2) outside data (reference genomes, annotation, etc.)
(3) intermediate data (from between steps)

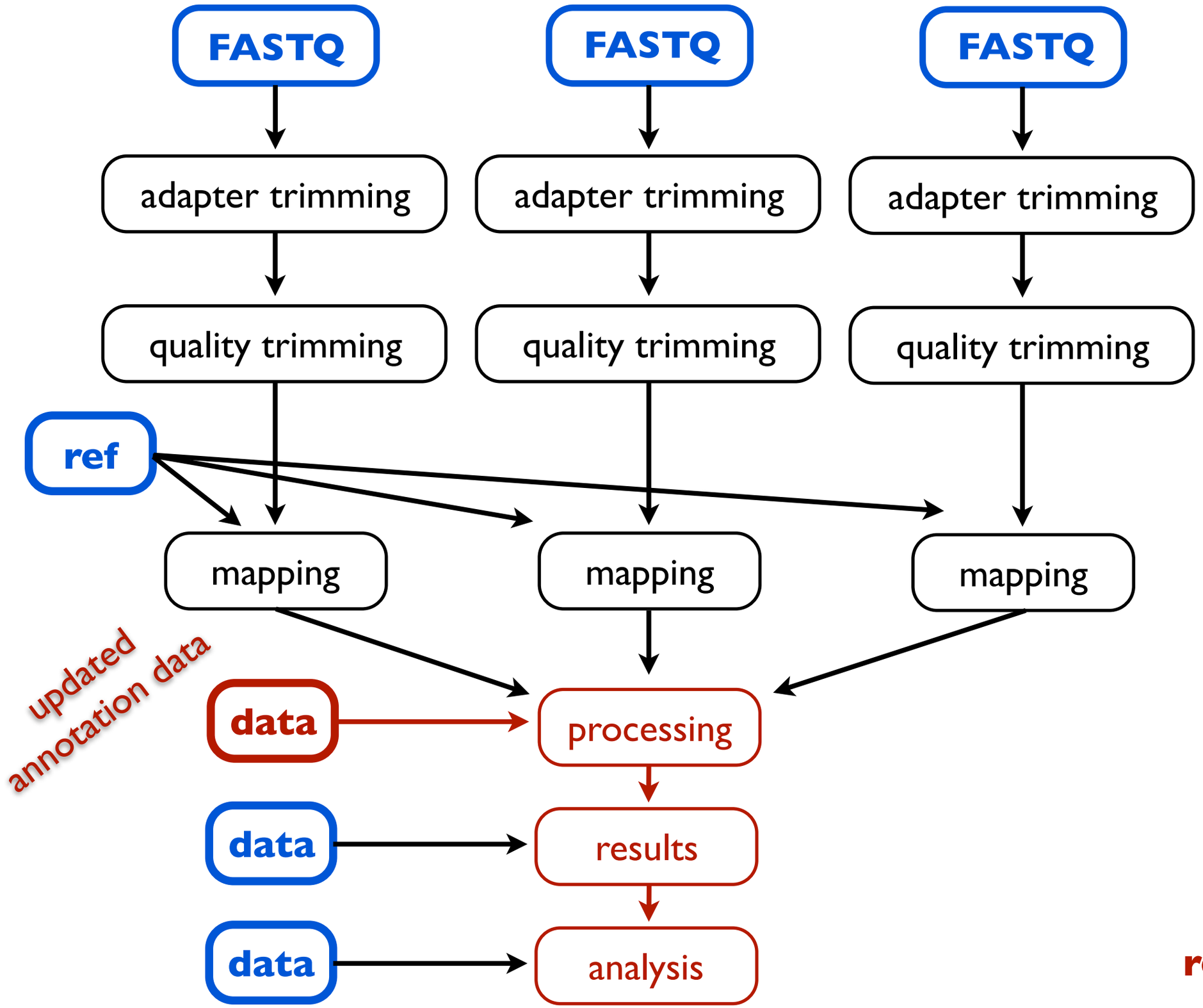
Bioinformatics Workflows



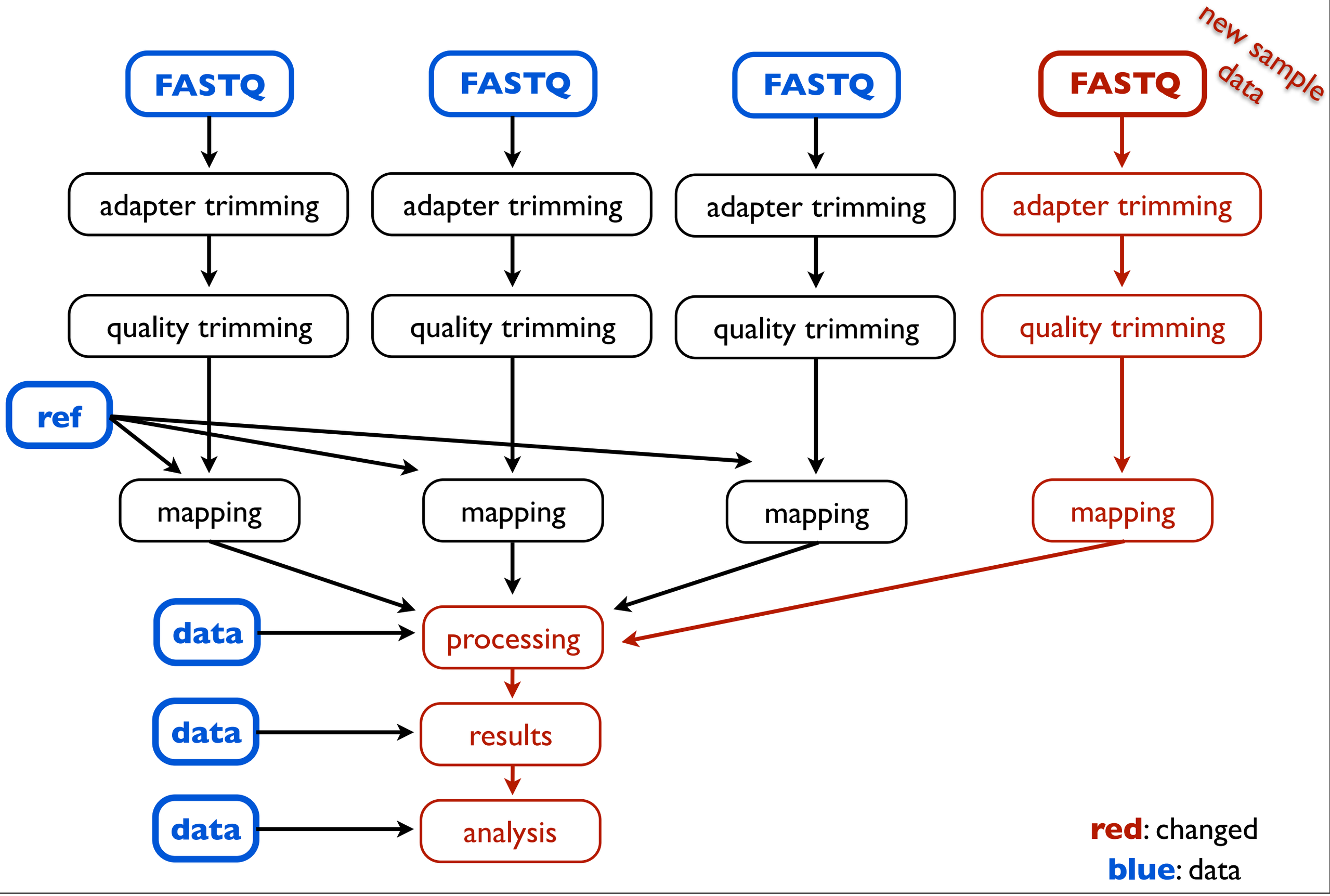
Bioinformatics Workflows



Bioinformatics Workflows



Bioinformatics Workflows



Automating Workflows

Wrapping a workflow in a **script** is optimal because:

(1) It can be re-run many times.

(2) Scripts facilitate documenting each step.

(3) Scripts can be shared with collaborators, re-run to reproduce analysis, and collaboratively edited.

What Scripting Language?

- Bash (or other shell)

- Python/Perl

What Scripting Language?

Bash (or other shell) scripts

Advantages

- Easy to call command line bioinformatics programs.

- Quick for simple tasks.

- Portability is usually easy.

Disadvantages

- Parallelization is hard.

- Syntax is a little old and hard for complex pipelines.

- Long scripts can be hard to organize/read.

- Not many libraries.

What Scripting Language?

Python/Perl

Advantages

- Powerful languages.

- Rich libraries.

- Great for complex pipelines, as modules, object orientation, and functions can help organize code.

Disadvantages

- Parallelization is *still* hard.

- Calling and interfacing with command line bioinformatics programs requires more overheard than Bash.

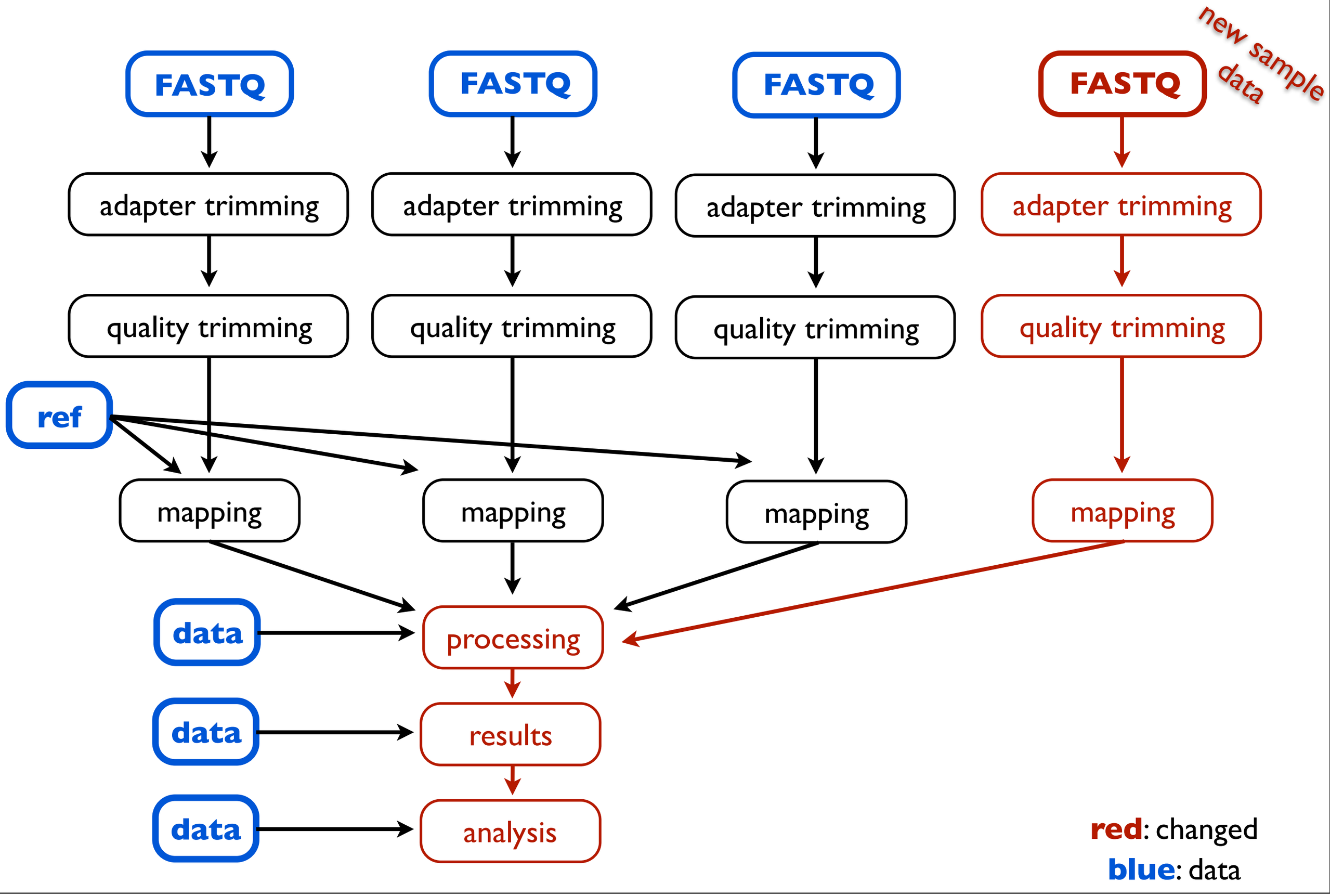
- Requires a lot of custom code to represent complex workflows.

What Scripting Language?

Scripting Languages: The Core Problem

Regardless of the scripting language, implementing the dependency structure to prevent re-running unnecessary steps can be a pain.

Bioinformatics Workflows

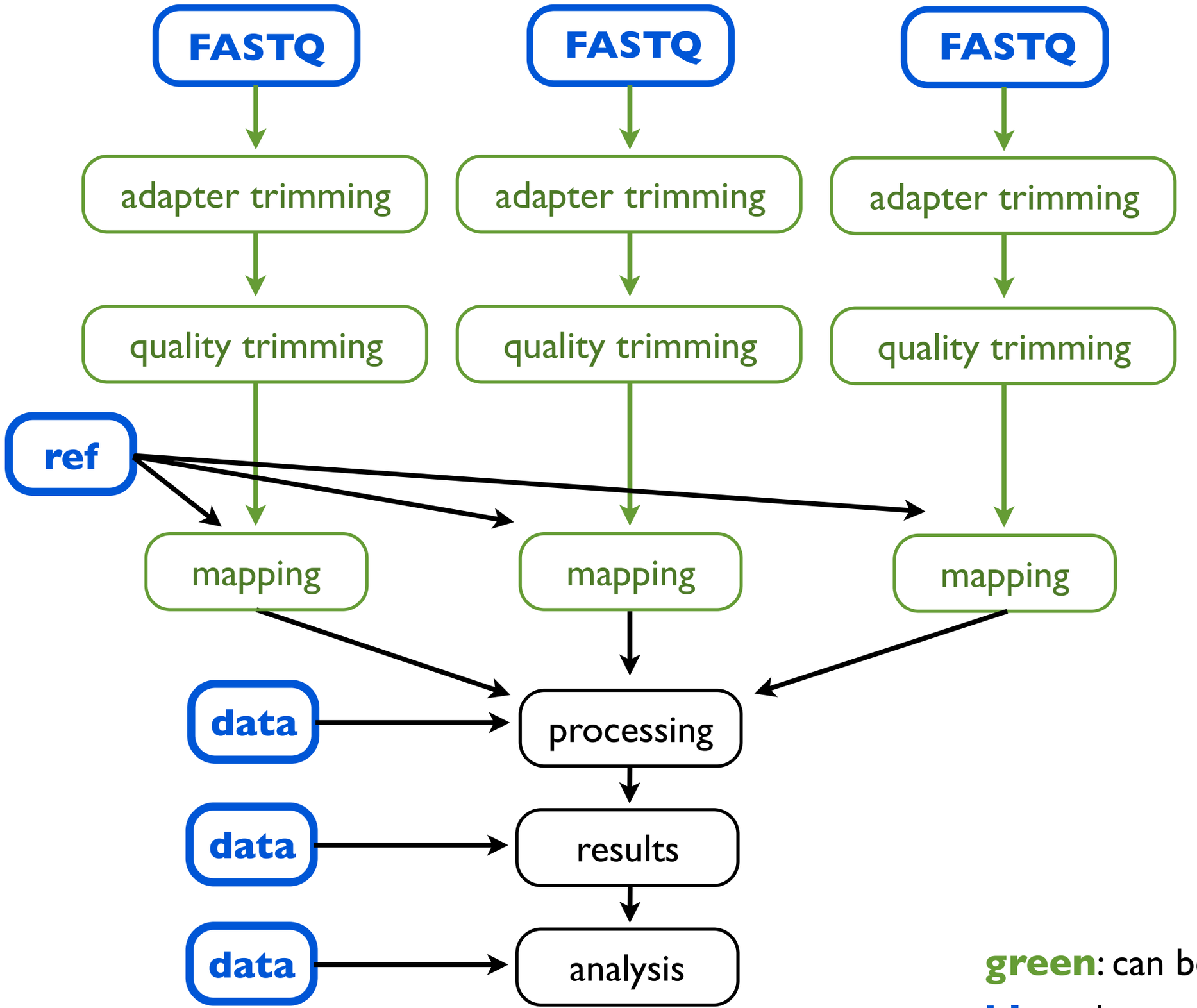


What Scripting Language?

Scripting Languages: Parallelization

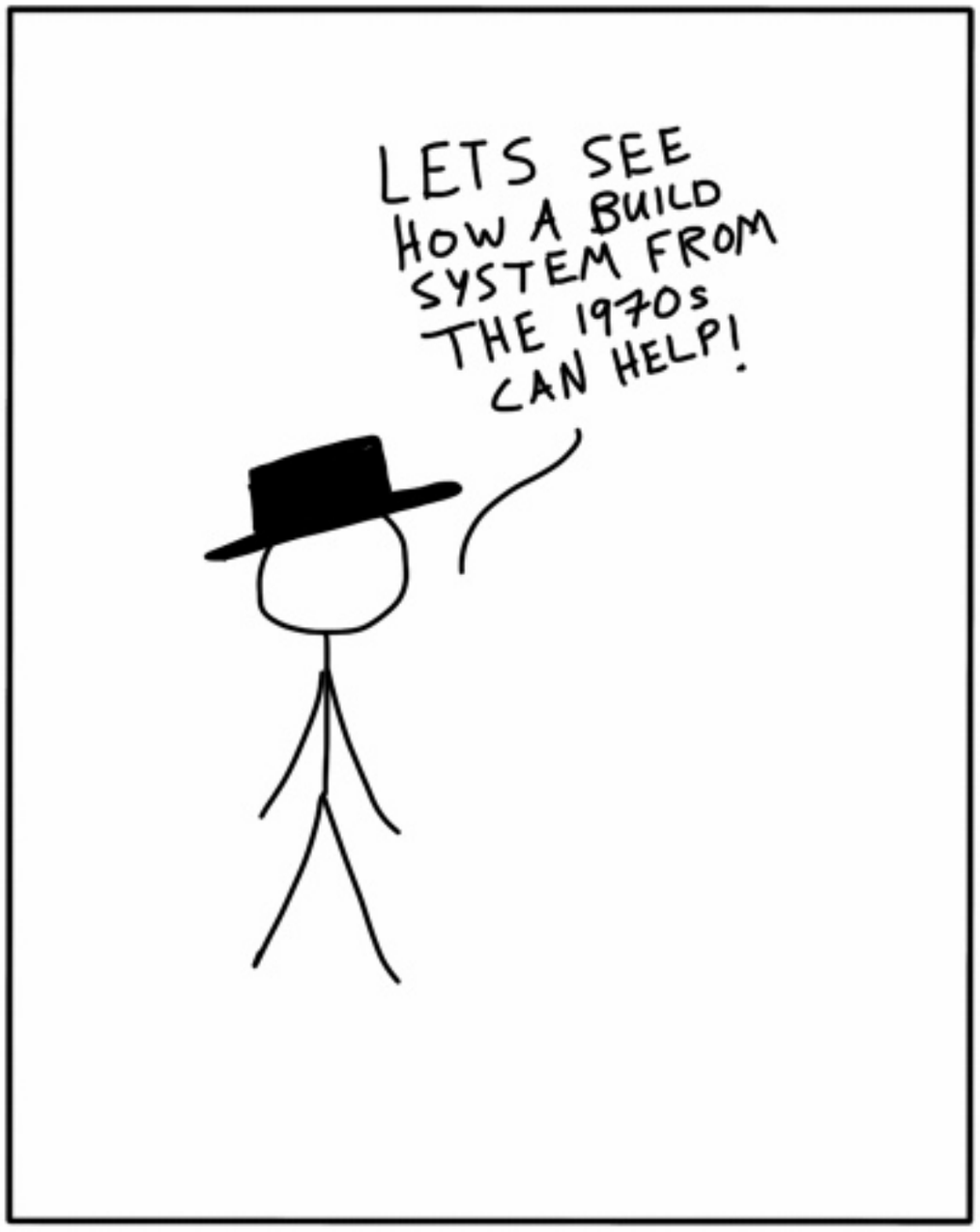
Additionally, parallelizing workflows can be a pain.

Bioinformatics Workflows



green: can be run in parallel

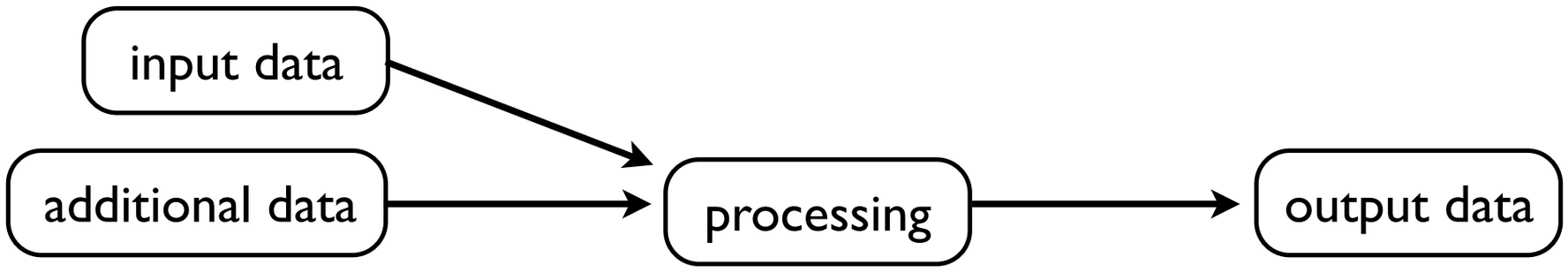
blue: data



Part II: Makefiles in Bioinformatics

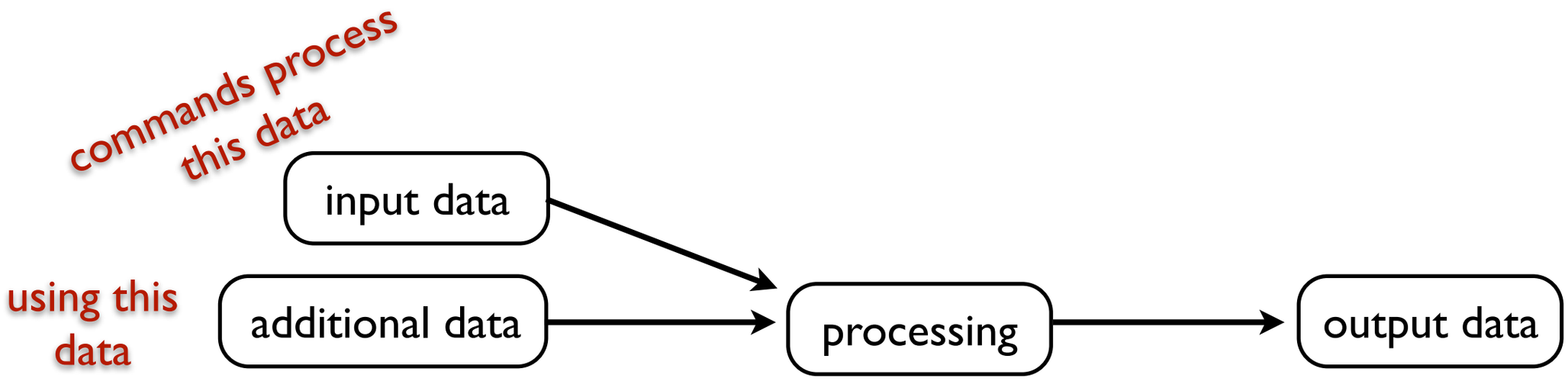
Bioinformatics Workflows

Abstracting Each Step



Bioinformatics Workflows

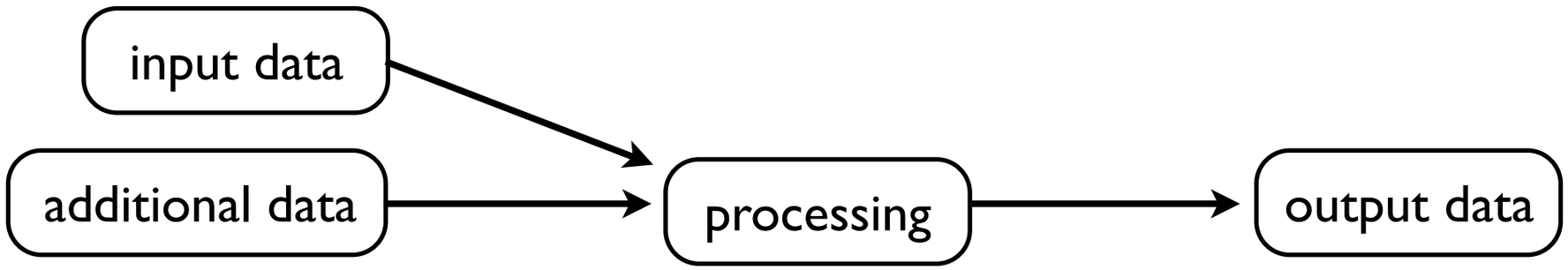
Abstracting Each Step



e.g. bioinformatics commands

Makefiles are Workflows

Makefiles Have a Declarative Format

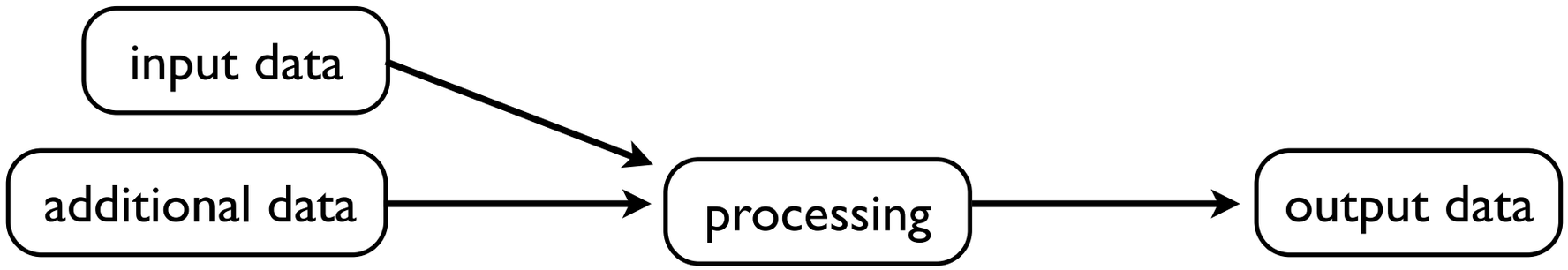


Makefile Rule

```
target: prerequisites
   recipe
```

Makefiles are Workflows

Makefiles Have a Declarative Format



Makefile Rule

```
target: prerequisites
   recipe
```

Add this to a file (usually called `Makefile`) and run with the command `make`

Makefiles are Workflows

Makefiles Have a Declarative Format

Makefile Rule

```
target: prerequisites
   recipe
```

Makefiles are Workflows

Makefiles Have a Declarative Format


```
target: prerequisites
    recipe
```

We tell Make all the rules, how they connect, and it figures out how to run your workflow.

Simple Makefiles

Makefiles can be very simple, or incredibly complex (check out some next time you're compiling software).

```
seqlens.txt: input.fa
  bioawk -cfastx '{print $$name"\t"length($$seq)}' input.fa > seqlens.txt
```

Note we have to escape variables

Simple Makefiles

Why the redundancy? We're specifying dependencies so Make can track input.

```
# in Makefile:

seqlens.txt: input.fa
   bioawk -cfastx '{print $$name"\t"length($$seq)}' input.fa > seqlens.txt

# in shell:

$ make
bioawk -cfastx '{print $name"\t"length($seq)}' input.fa > seqlens.txt

$ make
make: `seqlens.txt' is up to date.

$ touch input.fa
$ make
bioawk -cfastx '{print $name"\t"length($seq)}' input.fa > seqlens.txt
```

Simple Makefiles

Why the redundancy? We're specifying dependencies so Make can track input.

```
# in Makefile:

seqlens.txt: input.fa
   bioawk -cfastx '{print $$name"\t"length($$seq)}' input.fa > seqlens.txt

# in shell:

$ make
bioawk -cfastx '{print $name"\t"length($seq)}' input.fa > seqlens.txt

$ make
make: `seqlens.txt' is up to date.

$ touch input.fa
$ make
bioawk -cfastx '{print $name"\t"length($seq)}' input.fa > seqlens.txt
```

Makefile uses modification time

Simple Makefiles

We can use Make's special variables to refer to the target and prerequisite file.

```
# in Makefile:

seqlens.txt: input.fa
   bioawk -cfastx '{print $$name"\t"length($$seq)}' $^ > $@
```

$<: first prerequisite

$@: target name

$^: all prerequisites

There are **a lot** more:
http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

Simple Makefiles

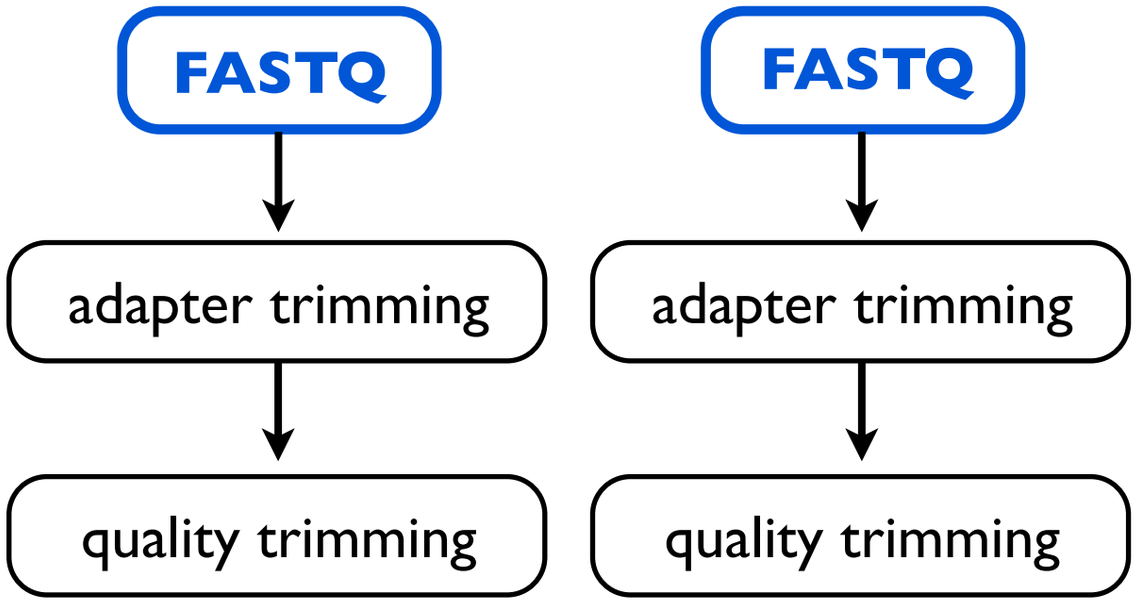
Variables can also be used to make it easier to replace values (recommended!).

```
# in Makefile:

SEQ_FILE = input.fa
LENS_FILE = seqlens.txt

$(LENS_FILE): $(SEQ_FILE)
   bioawk -cfastx '{print $$name"\t"length($$seq)}' $^ > $@
```

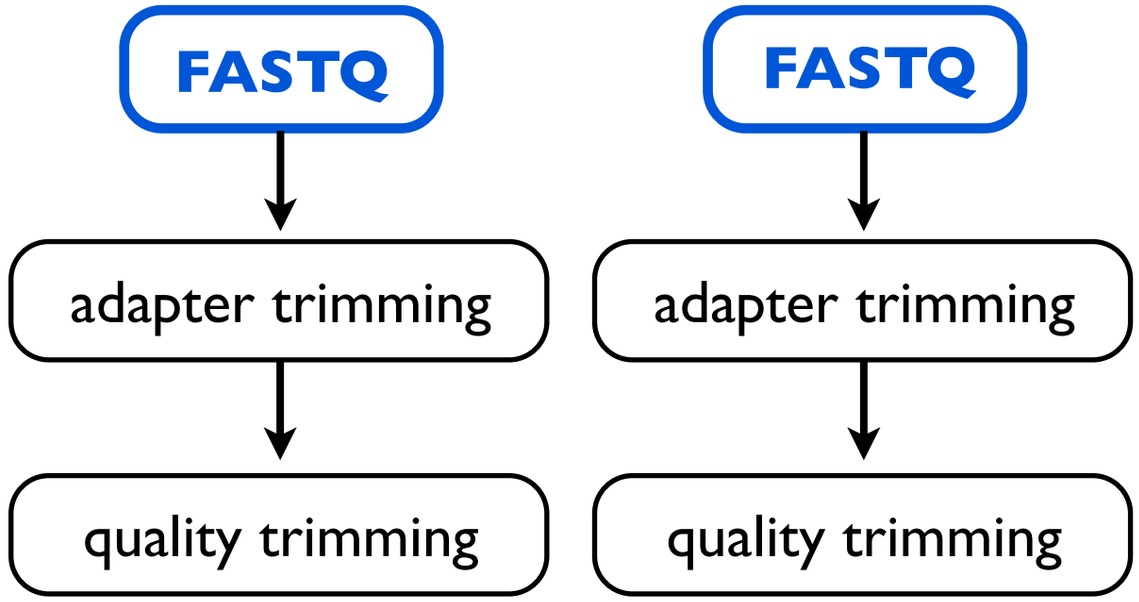
Parallel Make

What about making Make run in parallel?

Parallel Make

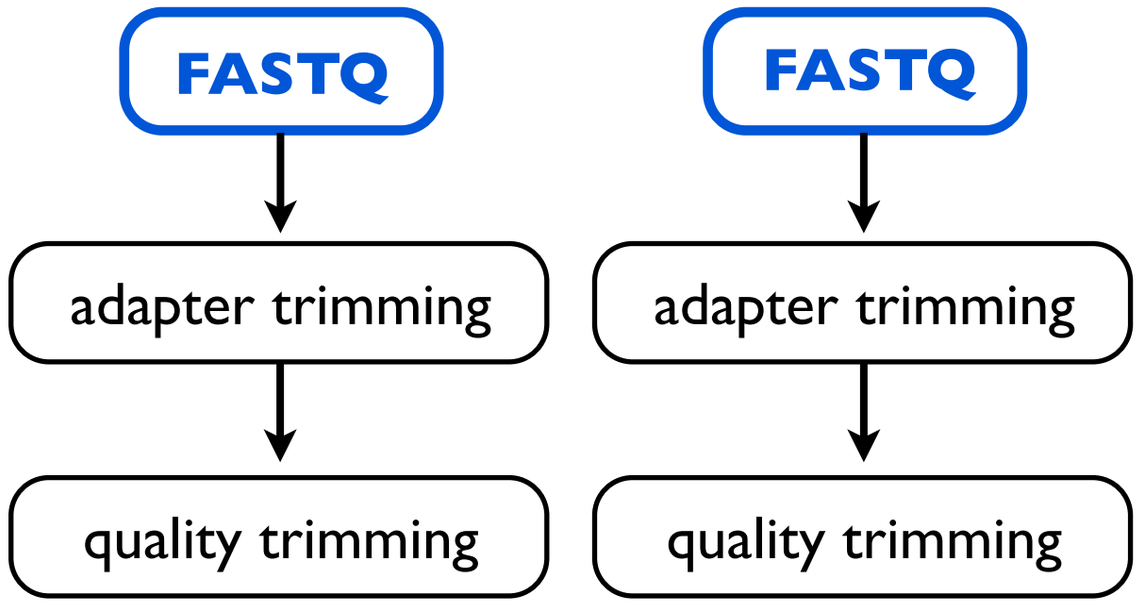


Parallel Make



Can be done in parallel!

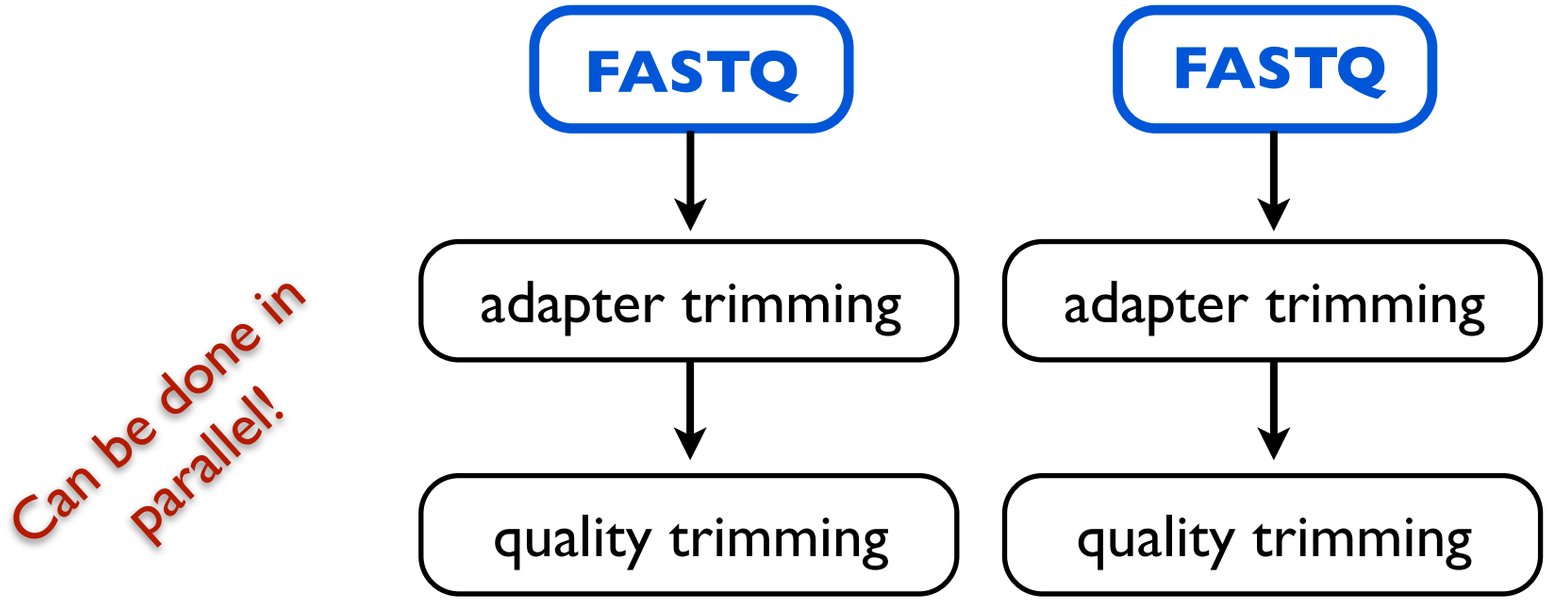
Parallel Make



Can be done in parallel!

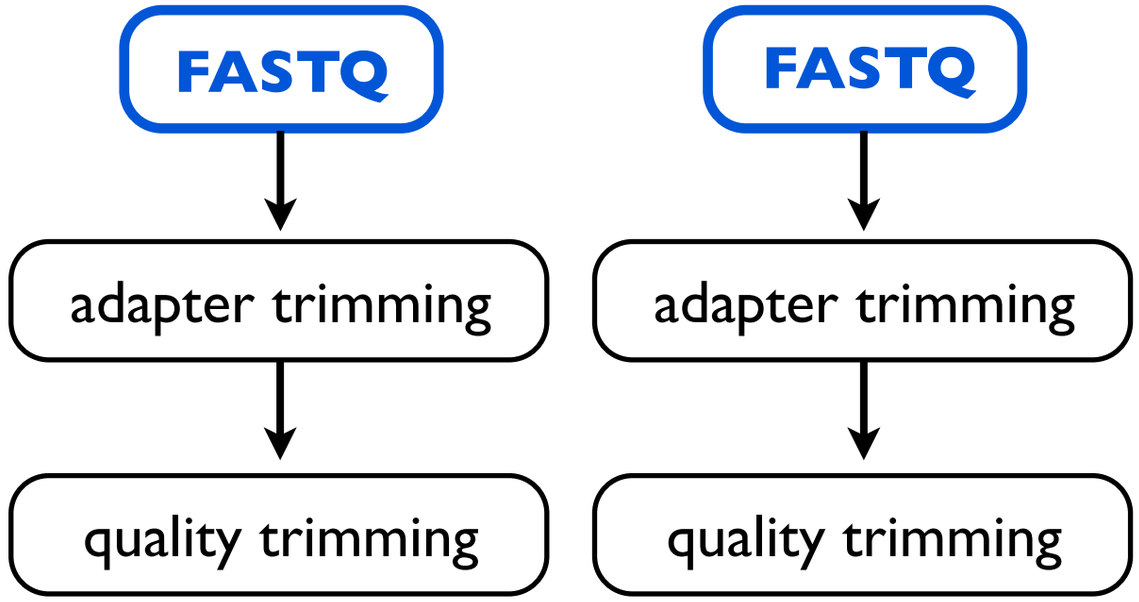
```
# in Makefile:

SEQ_FILE_1 = in-1.fq
SEQ_FILE_2 = in-2.fq
ADAPTER_FILE = adapters.fa

SEQ_FILE_TRIM_1 = in-trimmed-1.fq
SEQ_FILE_TRIM_2 = in-trimmed-2.fq

# Files we want out:
all: SEQ_FILE_TRIM_1 SEQ_FILE_TRIM_2

$(SEQ_FILE_TRIM_1): $(SEQ_FILE_1) $(ADAPTER_FILE)
   scythe -a $(ADAPTER_FILE) -p 0.3 $< | seqtk trimfq - > $@

$(SEQ_FILE_TRIM_2): $(SEQ_FILE_2) $(ADAPTER_FILE)
   scythe -a $(ADAPTER_FILE) -p 0.3 $< | seqtk trimfq - > $@
```

Parallel Make




```
# in shell:

$ make all
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
```

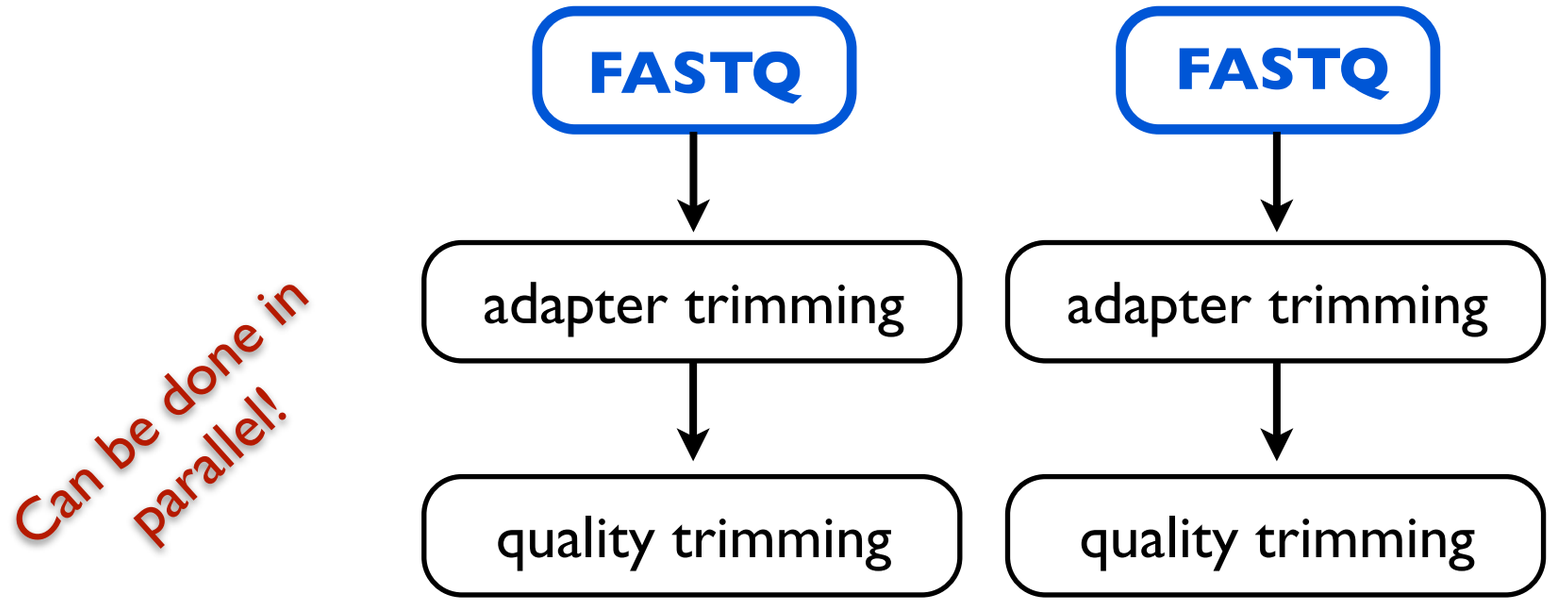
Parallel Make



Can be done in parallel!

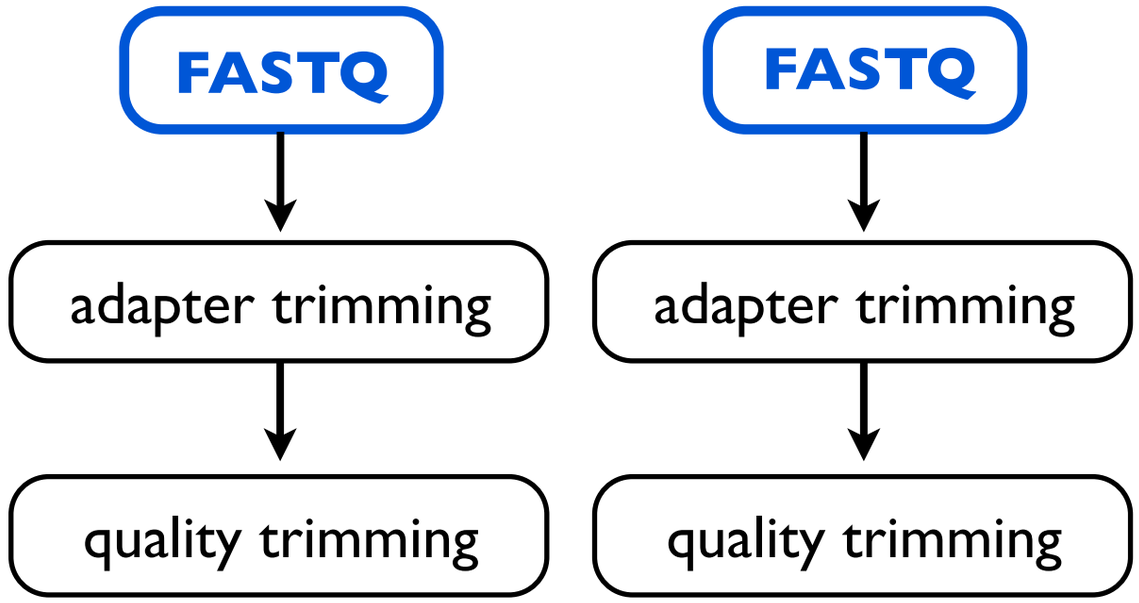
```
# parallel operation:

$ make -j 2 all
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
```

Parallel Make




```
# parallel operation:

$ make -j 2 all
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq

# is it faster?
$ time (make -j 2 all)
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
( make -j 2 all; )  0.47s user 0.03s system 211% cpu 0.241 total

$ time (make all)
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
( make all; )  0.44s user 0.03s system 106% cpu 0.435 total
```

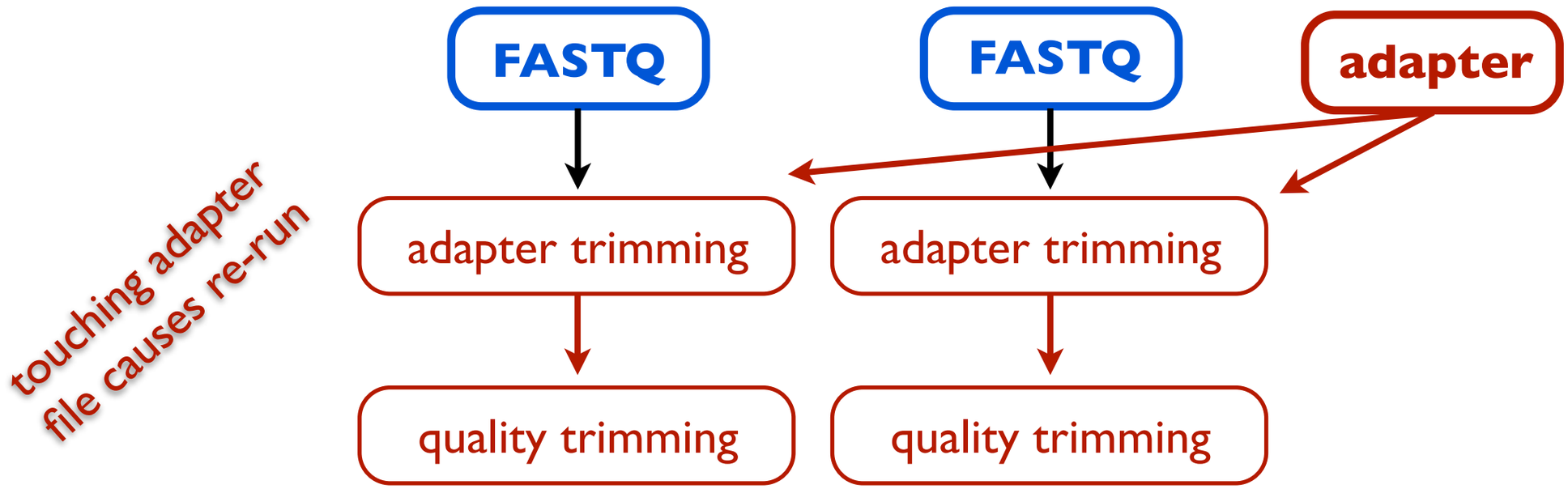
Parallel Make



Can be done in parallel!

```
# parallel operation:

$ make -j 2 all
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq

# is it faster?
$ time (make -j 2 all)
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
( make -j 2 all; )  0.47s user 0.03s system 211% cpu 0.241 total

$ time (make all)
scythe -a adapters.fa -p 0.3 in-1.fq 2> scythe-1.stats | seqtk trimfq - > in-trimmed-1.fq
scythe -a adapters.fa -p 0.3 in-2.fq 2> scythe-2.stats | seqtk trimfq - > in-trimmed-2.fq
( make all; )  0.44s user 0.03s system 106% cpu 0.435 total
```

Parallel Make




```
# in Makefile:

SEQ_FILE_1 = in-1.fq
SEQ_FILE_2 = in-2.fq
ADAPTER_FILE = adapters.fa

SEQ_FILE_TRIM_1 = in-trimmed-1.fq
SEQ_FILE_TRIM_2 = in-trimmed-2.fq

# Files we want out:
all: SEQ_FILE_TRIM_1 SEQ_FILE_TRIM_2

$(SEQ_FILE_TRIM_1): $(SEQ_FILE_1) $(ADAPTER_FILE)
   scythe -a $(ADAPTER_FILE) -p 0.3 $< | seqtk trimfq - > $@

$(SEQ_FILE_TRIM_2): $(SEQ_FILE_2) $(ADAPTER_FILE)
   scythe -a $(ADAPTER_FILE) -p 0.3 $< | seqtk trimfq - > $@
```

Makefile Tricks

- all failing commands stop Make (nice!)
- to ignore failing lines, preface with a dash (–)
- `@echo` can be used to echo within Make
- use `clean` targets to clean up work
- specify non-file targets with `.PHONY`

```
# in Makefile:
.PHONY: clean

hello-world.txt:
   -/bin/false
   @echo "prints hello world to file"
   echo "hello world" > $@

clean:
   rm -f hello-world.txt

# in shell:
$ make
false
make: [hello-world.txt] Error 1 (ignored)
prints hello world to file
echo "hello world" > hello-world.txt
```

Makefile Tricks

Make, like someone that just learned

```
grep -c > input.fa # bad
```

can clobber your files. In both cases,
make data read-only.

Pattern Matching

Part III: deeper into the Make wormhole.

Pattern Matching

Programmers hate repeating themselves, which we did in the Makefile earlier.

Pattern Matching

Programmers hate repeating themselves, which we did in the Makefile earlier.

Many C greybeards realized they were always doing the same things to turn `.c` files into `.o` files.

In bioinformatics we do the same stuff to turn `.fq` files into `.vcf` files*

Pattern Matching

Programmers hate repeating themselves, which we did in the Makefile earlier.

Many C greybeards realized they were always doing the same things to turn `.c` files into `.o` files.

In bioinformatics we do the same stuff to turn `.fq` files into `.vcf` files*

Pattern matching rules match a prerequisite file and turn it into another file.

Pattern Matching

Programmers hate repeating themselves, which we did in the Makefile earlier.

Many C greybeards realized they were always doing the same things to turn `.c` files into `.o` files.

In bioinformatics we do the same stuff to turn `.fq` files into `.vcf` files*

(*assuming no populations)

Pattern Matching

```
# in Makefile:
REF = ref.fa
IN_FILE = sample.fq

all: $(patsubst %.fq, %.vcf, $(IN_FILE))

# since indexing the genome with bwa index does create files, this
# could be those files, but I'm keeping it simple
.PHONY: index clean

index: $(REF)
	bwa index -a is $(REF)

# don't use this in real life, just a *toy* example!
%.vcf: %.fq $(REF)
	bwa mem $(REF) $< | samtools view -Sb - | samtools mpileup -uf $(REF) - | bcftools
view - > $@

clean:
	rm -f *.vcf

clean-index:
	# be careful with clean
	rm -f $(REF).*


# there's a bug here, can you find it?
```

Pattern Matching

```
# in Makefile:
REF = ref.fa
IN_FILE = sample.fq

all: $(patsubst %.fq, %.vcf, $(IN_FILE))

# since indexing the genome with bwa index does create files, this
# could be those files, but I'm keeping it simple
.PHONY: index clean

index: $(REF)
	bwa index -a is $(REF)

# don't use this in real life, just a *toy* example!
%.vcf: %.fq $(REF)
	bwa mem $(REF) $< | samtools view -Sb - | samtools mpileup -uf $(REF) - | bcftools
view - > $@

clean:
	rm -f *.vcf

clean-index:
	# be careful with clean
	rm -f $(REF).*


# there's a bug here, can you find it?
```

Pattern Matching

```
# in shell:
$ make clean-index clean index all
# be careful with clean
rm -f ref.fa.*
rm -f *.vcf
bwa index -a is ref.fa
[bwa_index] Pack FASTA... 0.00 sec
[bwa_index] Construct BWT for the packed sequence...
[bwa_index] 0.00 seconds elapse.
[bwa_index] Update BWT... 0.00 sec
[bwa_index] Pack forward-only FASTA... 0.00 sec
[bwa_index] Construct SA from BWT and Occ... 0.00 sec
[main] Version: 0.7.3a-r367
[main] CMD: bwa index -a is ref.fa
[main] Real time: 0.002 sec; CPU: 0.003 sec
bwa mem ref.fa sample.fq | samtools view -Sb - | samtools mpileup -uf ref.fa - | bcftools
view - > sample.vcf
[fai_load] build FASTA index.
[bam_header_read] EOF marker is absent. The input is probably truncated.
[M::main_mem] read 1 sequences (74 bp)...
[main] Version: 0.7.3a-r367
[main] CMD: bwa mem ref.fa sample.fq
[main] Real time: 0.001 sec; CPU: 0.002 sec
[samopen] SAM header is present: 1 sequences.
[mpileup] 1 samples in 1 input files
<mpileup> Set max per-file depth to 8000
```

Makefile Gotchas

Make will be hard to learn at first, because like SQL, it's a **declarative language** and even if you're used to programming in Python, Perl, C, etc (mostly imperative languages), it's a different class of language.

Makefile Gotchas

Make will be hard to learn at first, because like SQL, it's a **declarative language** and even if you're used to programming in Python, Perl, C, etc (mostly imperative languages), it's a different class of language.

It is also is an old language, and a bit clunky. But it's still standard and (at least in my opinion) better than alternatives.

Makefile Gotchas

Problem: Files may not need an update, so Make may not run.

Solution: Use a clean target to remove files to force them to re-run.

Makefile Gotchas

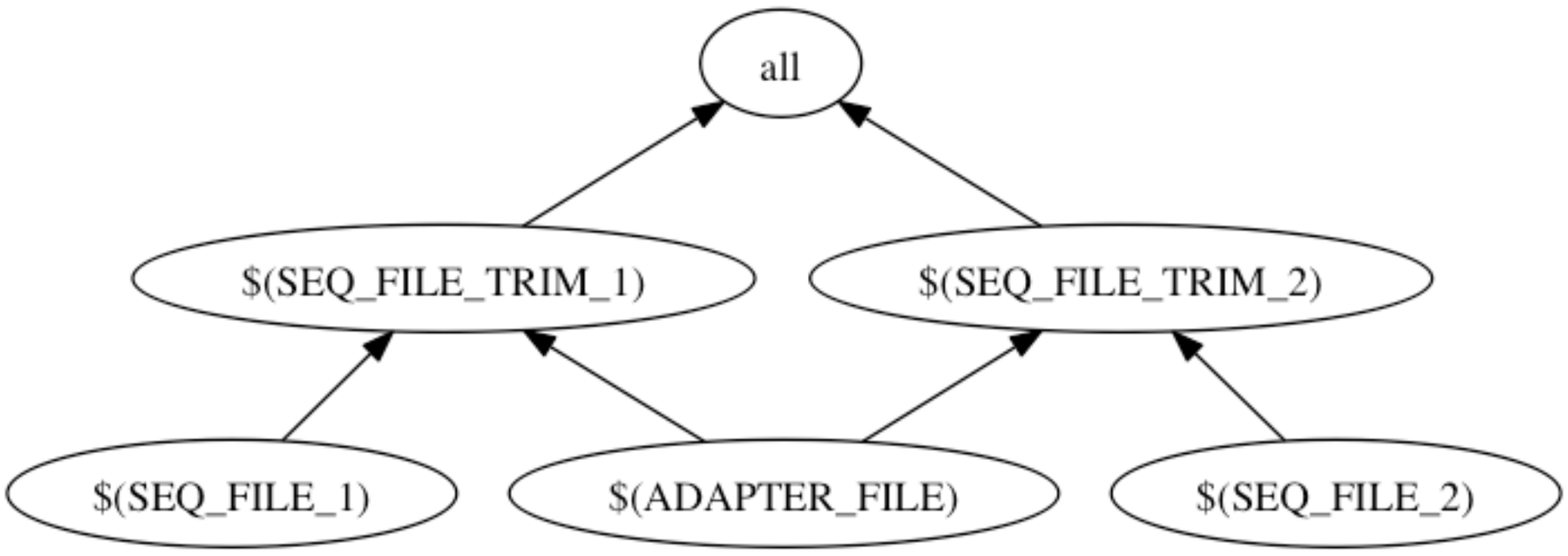
Problem: Makefile doesn't run all rules.

Solution: Check all rules are connected. Check that first rule (or `all`, by convention) generates the last file you need, and has all others as prerequisites.

Makefile Gotchas

There are tools to look at Makefiles, like makefile2dot.

```
python makefile2dot.py < Makefile | dot -Tpng > out.png && open out.png
```



https://github.com/vak/makefile2dot

(arrows point to dependency here)

Makefile Gotchas

Problem: Make is doing something strange.

Solution: Keep is simple. Use `make -n -d` (`-n`, or `--dry-run` doesn't run anything and `-d` turns on debugging information).

Makefile Gotchas

Problem: You have lots of sub-directories to work with.

Solution: Keep is simple. I like to create all files I want at the top, using text functions like

`$(patsubst ... )`,

`$(filter ... )`, **and** `$(notdir ... )`

https://www.gnu.org/software/make/manual/html_node/File-Name-Functions.html
https://www.gnu.org/software/make/manual/html_node/Text-Functions.html

Makefile Gotchas

Remember, use a tab character, not spaces for recipes.

Makefile's Problems

Make is a little clunky of a language, but a hell of a tool. Other systems exist, but I find Make the simplest and most widely supported.

Even if you hate Make, it's hard to argue with the fact that **declarative languages** are the best way of doing this.