Pipelines as makefiles

or snakemake, waf, scons, rake, ninja, ...

Wolfgang Resch

National Institute of Arthritis and Musculoskeletal and Skin Diseases, Laboratory of Molecular Immunogenetics

April 10, 2014

Table of Contents

1 Introduction

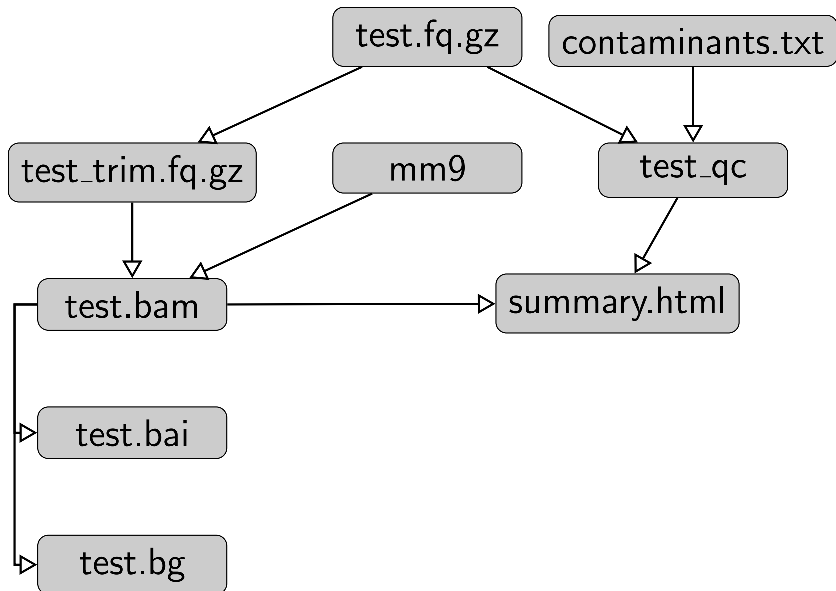
2 Anatomy of a Makefile

3 Parallelizing make

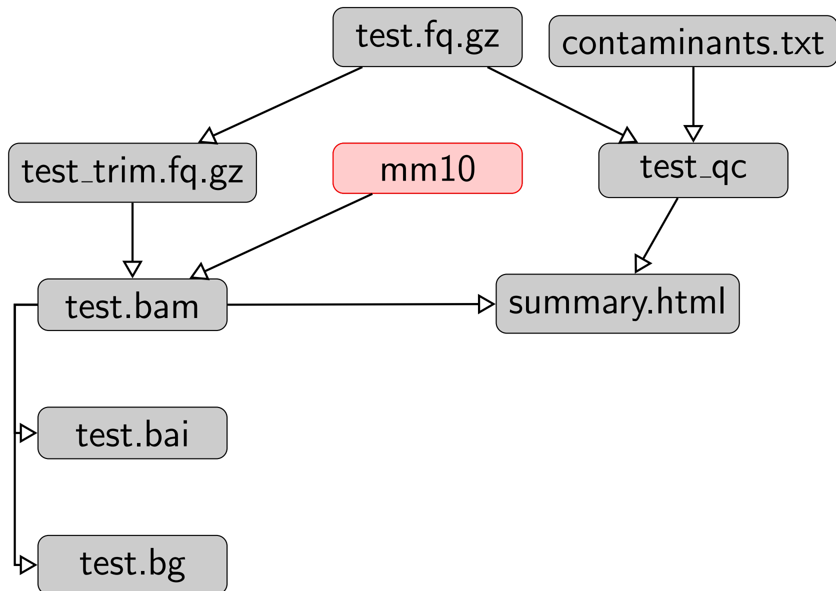
4 A Makefile Generator

5 Alternatives to make

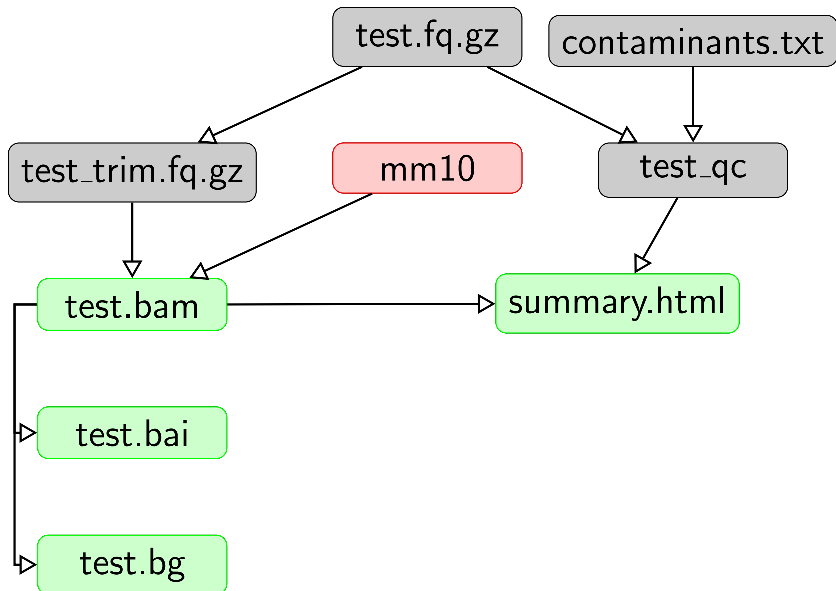
Example pipeline



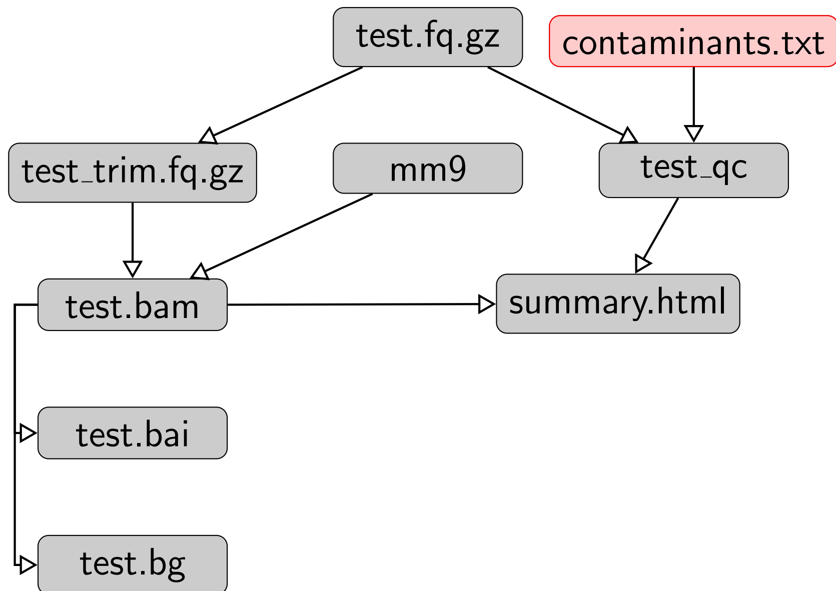
Example pipeline



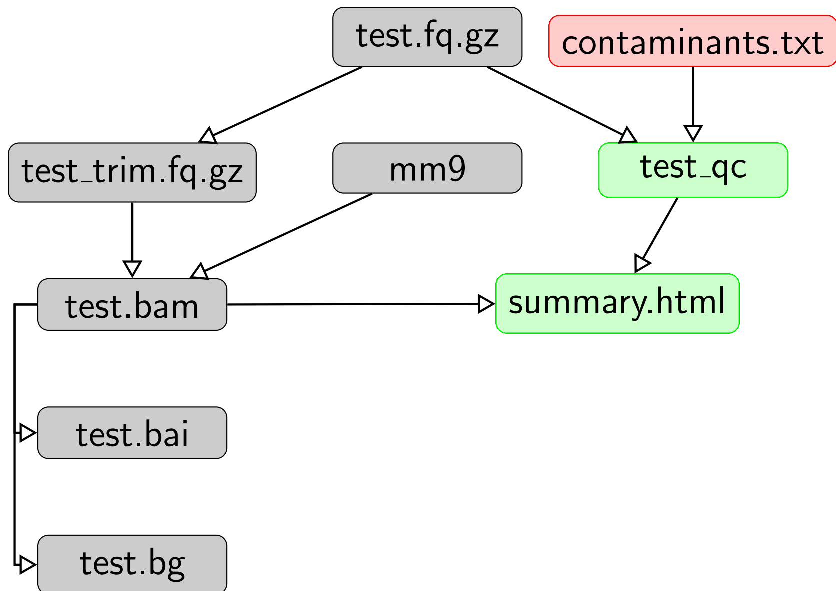
Example pipeline



Example pipeline



Example pipeline



Code organization

- Monolithic script

Code organization

- Monolithic script - always have to regenerate all files.

Code organization

- Monolithic script - always have to regenerate all files.
- Collection of scripts, tied together

Code organization

- Monolithic script - always have to regenerate all files.
- Collection of scripts, tied together - have to keep track of dependencies manually.

Make is a dependency tracking build system

Originally intended for compiling source code, but can be repurposed for other uses.

Table of Contents

1 Introduction

2 Anatomy of a Makefile

3 Parallelizing make

4 A Makefile Generator

5 Alternatives to make

Rules

```
1 # comment
2 target: prerequisites
3         command
4         command
```

- Makefile rules explain how a target is created, what files are required to create it, and how it is created.
- Commands are shell commands. By default each is executed in a separate shell.
- Commands have to be indented by an initial **tab** character.
- Targets are recreated if they are older than at least one of the prerequisites.

Example pipeline

```
# -*- mode: makefile; -*-
SHELL := /bin/bash

qc/t_qc: t.fastq.gz contaminats.txt
	fastqc -c contaminants.txt -o temp t.fastq.gz
	mv temp/t_fastqc/fastqc_data.txt qc/t_qc
	rm -rf temp/t_fastqc*

t_trim.fastq.gz: t.fastq.gz
	zcat t.fastq.gz | fq_qual_trim3 | gzip -c > t_trim.fastq.gz

t.bam: t_trim.fastq.gz /path/to/mm9.ebwt
	zcat t_trim.fastq.gz \
	  | bowtie --sam --best --strata --all /path/to/mm9 - \
	  | samtools view -Sb -F4 - \
	  > t.bam \
	&& samtools sort t.bam t.bam \
	&& mv t.bam.bam t.bam

t.bai: t.bam
	samtools index t.bam
```

Calling make

```
1 make          # creates the first target: qc/t_qc
2 make qc/t_qc  # no action
3 make t.bai    # creates test.bai and all it's prereqs
4 make t.bai qc/t_qc
```

- By default, `make` looks for a file called `makefile` or `Makefile`. If filename is different, it has to be provided with the `-f` option.
- If a target is not older than its prerequisites, it will not be recreated.
- Rules can be executed in arbitrary

Variables

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces:

referencing make variables

$(foo) or ${foo}

If shell variables are used in a command, they have to be referenced with '$$' since a single '$' is interpreted as a make variable

Recursively expanded variables

Recursively expanded variables are defined with '=' or 'define'

```
1  foo:
2          echo $(foo)
3
4  foo = $(bar)
5  bar = Really?
```

- `make foo` will echo "Really?" because `$(foo)` expands to `$(bar)`, which in turn expands to "Really?".
- This type of variable stores it value verbatim at definition and then expands at the point of reference.

Recursively expanded variables

- For some versions of make, this is the only type of variable.
- Don't try to do something like '`foo = $(foo) bar`' as it would result in an infinite expansion.

Simple expanded variables

Simple expanded variables are defined with ':='

```
1 all:
2         echo $(foo)
3
4 foo := $(bar)
5 bar := Really?
```

`make foo` will echo " ", an empty string. This is because simply expanded variables are expanded at the point of definition and when `foo` was define, `bar` was not yet set.

Automatic variables

The value of an automatic variable is local to a rule.

- `$@` Name of the target
- `$<` First prerequisite
- `$^` All prerequisites separated by spaces
- `$*` Stem of an implicit rule

Example pipeline

```makefile
# -*- mode: makefile; -*-
SHELL := /bin/bash
BOWTIE_OPT := --sam --best --strata --all
BOWTIE_IDX := /path/to/mm9

qc/t_qc: t.fastq.gz contaminats.txt
	fastqc -c $(word 2,$^) -o temp $<
	mv temp/$(subst .fastq.gz,_fastqc,$<)/fastqc_data.txt $@
	rm -rf temp/$(subst .fastq.gz,_fastqc,$<)*

t_trim.fastq.gz: t.fastq.gz
	zcat $< | fq_qual_trim3 | gzip -c > $@

t.bam: t_trim.fastq.gz $(BOWTIE_IDX).ebwt
	zcat $< \
	  | bowtie $(BOWTIE_OPT) $(BOWTIE_IDX) - \
	  | samtools view -Sb -F4 - \
	  > $@ \
	&& samtools sort $@ $@ \
	&& mv $@.bam $@

t.bai: t.bam
	samtools index $<
```

Target specific variables

Most variables in a makefile are global. Exceptions are automatic variables and target specific variables. For example in

```
1  foo := global
2
3  foo1:
4          echo $(foo)
5
6  foo2: foo = local
7  foo2:
8          echo $(foo)
```

`make foo1` would echo “global” and `make foo2` would echo “local”

Command modifiers

- Commands preceeded with ‘`@`’ are not echoed before execution
- Commands preceeded with ‘`-`’ ignore errors

Functions

A function call looks similar to a variable expansion

```
1     $(functionname arg)
2     $(functionname arg1,arg2,...,argn)
```

and is allowed anywhere a variable reference is allowed. Functions are expanded like variables.

Functions

Gnu make provides a number of useful functions for

- text: `subst`, `patsubst`
- lists: `filter`, `filter-out`, `word`, `wordlist`
- file names: `dir`, `notdir`, `suffix`, `prefix`, `basename`, `addsuffix`, `addprefix`, `join`, `wildcard`, `realpath`, `abspath`
- conditionals: `if`, `or`, `and`
- looping: `foreach`
- functions: `call` for defining new functions
- reporting: `error`, `warning`, `info`
- shelling out: `shell`

Phony targets

A phony target does not actually create it's target as a file. Phony targets can be used to execute tasks that don't create files (e.g. cleanup actions) and to simplify execution of groups of targets (e.g. a phony target depending on a number of real targets).

Example pipeline

```
1   # -*- mode: makefile; -*-
2   SHELL := /bin/bash
3   BOWTIE_OPT := --sam --best --strata --all
4   BOWTIE_IDX := /path/to/mm9
5
6   .PHONY: all help
7
8   all: t.bai qc/t_qc
9   help:
10          @echo "List of targets: ..."
11
12  qc/t_qc: tmp = $(subst .fastq.gz,_fastqc,$<)
13  qc/t_qc: t.fastq.gz contaminats.txt
14          fastqc -c $(word 2,$^) -o temp $<
15          mv temp/$(tmp)/fastqc_data.txt $@
16          rm -rf temp/$(tmp)*
17
18  t_trim.fastq.gz: t.fastq.gz
19          zcat $< | fq_qual_trim3 | gzip -c > $@
20
21  t.bam: tmp = $(shell mktemp temp/XXX)
22  t.bam: t_trim.fastq.gz $(BOWTIE_IDX).ebwt
23          zcat $< \
24            | bowtie $(BOWTIE_OPT) $(BOWTIE_IDX) - \
25            | samtools view -Sb -F4 - \
```

Implicit rules

When a file is listed as a dependency for a target, Make will search for an explicit rule to generate/refresh the dependency. If no explicit rule is found, Make will look for an implicit rule. There are built-in implicit rules (e.g. compile and link C source code), but for most our purposes implicit rules have to be created by writing pattern rules.

Pattern rules

A pattern resembles an ordinary rule, but allows the inclusion of exactly one % character in the target name, where the '%' character acts as a wildcard matching any non-empty string. Any '%' characters in the prerequisites match the same string.
So, for example bam files can be indexed with

```
%.bam.bai: %.bam
        samtools index $<
```

Pattern rules

As mentioned before, concrete rules have a higher precedence than implicit rules. For example, given

```
1   %.bar:
2           @echo "implicit rule %.bar is executing" > $@
3   foo.bar:
4           @echo "explicit ruls foo.bar is executing" > $@
5   fnord: foo.bar
6           cat $< && rm $<
7   ifnord: goo.bar
8           cat $< && rm $<
```

`make fnord` will execute the more specific `foo.bar` rule whereas `make ifnord` will execute the implicit rule `goo.bar` based on the pattern rule `%.bar`

Processes as prerequisites

Now the pipeline looks pretty complicated and has the disadvantage that any changes to the commands do not trigger rebuilding of targets. Both these problems can be overcome by moving commands into external scripts and including the scripts as one of the prerequisites.

Example pipeline

```
1  # -*- mode: makefile; -*-
2  SHELL := /bin/bash
3  BOWTIE_OPT := --sam --best --strata --all
4  BOWTIE_IDX := /path/to/mm9
5
6  .PHONY: all help
7
8  all: t.bai qc/t_qc
9  help:
10         @echo "List of targets: ..."
11
12 qc/t_qc: run_fastqc.sh t.fastq.gz contaminats.txt
13         $^ $@
14
15 t_trim.fastq.gz: trim_3.sh t.fastq.gz
16         $^ $@
17
18 t.bam: tmp = $(shell mktemp temp/XXX)
19 t.bam: align_chipseq.sh t_trim.fastq.gz $(BOWTIE_IDX).ebwt
20         $^ $@ $(BOWTIE_OPT)
21
22 %.bam.bai: %.bam
23         samtools index $<
```

Table of Contents

1 Introduction

2 Anatomy of a Makefile

3 Parallelizing make

4 A Makefile Generator

5 Alternatives to make

Table of Contents

1 Introduction

2 Anatomy of a Makefile

3 Parallelizing make

4 A Makefile Generator

5 Alternatives to make

Table of Contents

1 Introduction

2 Anatomy of a Makefile

3 Parallelizing make

4 A Makefile Generator

5 Alternatives to make