Software Project (Lecture 5): Building & Scripting

Wouter Swierstra, Atze Dijkstra

Feb 2016

Last time

Working effectively with git and GitHub.

Today

Automatically building software

Scripting with Bash

Building Software

The premise: we have the source code of a software system and we want to generate an actual product (e.g., executables) out of that.
Problems:

- *Efficiency* Compiling a file is slow. Compiling 10,000 files is even slower
- *Multiple languages* How do we integrate them?
- *Variants* – different flavors, platforms, etc.

Efficiency

We could just specify a script that specifies all the steps necessary to build the system.

```
gcc -c main.c -o main.o
bison parser.y -o parser.c
gcc -c parser.c -o parser.o
gcc main.o parser.o -o program
```

Slow: whenever *anything* changes we rebuild *everything*.

Aside: C/C++ compilation model

- C/C++ sources are compiled into *object files*:

  ```
  gcc -c source.c -o object.o
  ```

- Object files are *linked* together into an *executable*.

  ```
  gcc object1.o object2.o object3.o -o program
  ```

Note: gcc is not only a compiler, but a front-end that invokes other tools (compiler, assembler, linker, etc.) based on the file extensions.

Improving efficiency

Solution: only execute a step when something has changed:

- When `main.c` changes, rebuild `main.o` , then `program`
- When `parser.y` changes, rebuild `parser.c` , then `parser.o`, then `program`.

The dependency graph

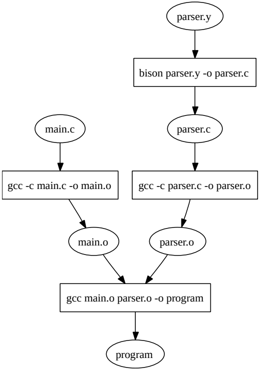

Figure 1: Dependency graph

Make

The most widely used build system is *Make* (Feldman, 1978)

- easy to use
- interop with most compilers or tools
- available on (almost) every operating system

There is probably no large software system in the world today that has not been processed by a version or offspring of MAKE. — Feldman's 2003 ACM Software System Award citation

When to use Make?

Make is useful in any situation where files are generated from other files and have to be updated automatically.

- Software generation (e.g., compilers, linkers, parser generators).
- Documentation (e.g., LaTeX; etc.)
- Maintaining a web site.

Makefiles

A *makefile* contains a list of *rules*. A rule specifies how a list of *targets* (the outputs) can be built from a list of *prerequisites* (inputs) by running a command:

```
target1 target2 ... targetn : dep1 dep2 ... depm
    command
```

Note: `command` must be prefixed with a TAB character, not just any whitespace!

Example

```
main.o: main.c
  gcc -c main.c -o main.o

parser.c: parser.y
  bison parser.y -o parser.c

parser.o: parser.c
  gcc -c parser.c -o parser.o

program: main.o parser.o
  gcc main.o parser.o -o program
```

Make

Make reads a makefile and recursively *updates* targets. A target is considered out-of-date when it doesn't exist or when it's older than its prerequisites.
For example:

- is required but it doesn't exist – rebuild.
- exists and has timestamp 20-8-2002 09:10:23; its prerequisite has timestamp 20-8-2002 09:11:56 – rebuild

Running Make

The first time:

```
$ make program
gcc -c main.c -o main.o
bison parser.y -o parser.c
gcc -c parser.c -o parser.o
gcc main.o parser.o -o program
```

Running Make, again

The second time:

```
$ make program
make: `program' is up to date.
```

Running Make (cont'd)

After editing `parser.y`

```
$ make program
bison parser.y -o parser.c
gcc -c parser.c -o parser.o
gcc main.o parser.o -o program
```

Phony targets

Phony targets are targets that are used to do useful things besides building stuff. The most common ones:

- `default` : the target that Make builds if no target is specified on the command line.
- `clean` : throw away derived files.

```
clean:
  rm -f *.o parser.c program
```

Pattern rules

Pattern rules match certain target/prerequisite pairs so we don't have to write many instances by hand.

```
%.ext1 : %.ext2
    command $< -o $@
```

The special variables `$<` and `$@` refer to the prerequisite and target, respectively.

Example

```
# generic rules
%.o: %.c
  gcc -c $< -o $@

%.c: %.y
  bison $< -o $@

# specific stuff
program: main.o parser.o
  gcc main.o parser.o -o program
```

Variables

Like any shell script, Makefiles may contain variables:

```
OBJS = main.o parser.o
PROG = program
CC = gcc

default: $(PROG)

$(PROG): $(OBJS)
  $(CC) $(OBJS) -o $(PROG)
```

If you need a different C compiler, you will only need to change this in one place.

Make critique

- *Unsafe*: prerequisites have to be specified by hand. A general software engineering rule is that when any information has to be manually maintained in more than one location, errors are practically guaranteed.
- Lack of abstraction facilities.
- Doesn't scale very well to very large systems.
- Hard to build variants side-by-side (although we can use some file name tricks to do it).

Make critique (2)

- Makefiles tend to get very big (and unreadable) – `automake` is a separate tool for generating Makefiles
- Make cannot handle system-dependent or user-dependent aspects (e.g. where is a certain library installed) – `autoconf`

Make is unsafe

In the file `hello.c`

```
#include "config.h"
...
```

But the Makefile states

```
hello: hello.o
  ...

hello.o: hello.c
  gcc ...
```

If `config.h` changes, `hello.o` will *not* be recompiled!

Lack of scalability

- Make was intended for *small* projects, preferably those that fit into a single directory.
- The most common way to build multi-directory projects is to let Make invoke itself recursively for each subdirectory – Peter Miller, *Recursive Make Considered Harmful*, 1997.

Recursive make

- project/Makefile

```
SUBDIRS=foo bar

all:
	for i in $(SUBDIRS); do make -C $i; done
```

- project/foo/Makefile

```
OBJS = bla.o xyzzy.o

all: libfoo.a

libfoo.a: $(OBJS)
	...
```

Recursive make - II

- `project/bar/Makefile`

```
all: prog

prog: main.o ../foo/libfoo.a
    gcc -o prog main.o -L../foo -lfoo
```

Problem: each makefile in a subdirectory specifies an incomplete DAG. This allows some changes to be missed.

`project/foo/Makefile`:

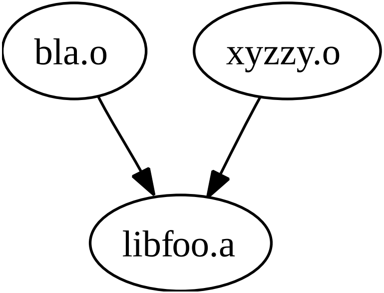

Figure 2: Build

`project/bar/Makefile`:

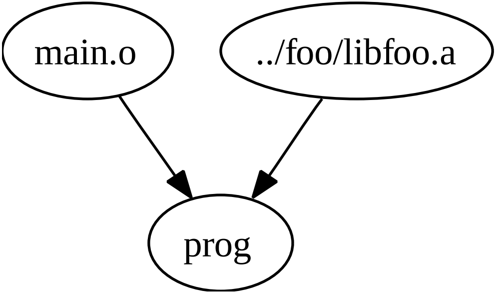

Figure 3: Build

Other issues with recursive make

- You need to specify the order in which directories are traversed
- Dependencies between directories are hard to express and easy to get wrong.
- To be on the safe side, too much is built
- Long build times and hard to parallelize
- Every build needs to be examined

Non-recursive make

```
all: bar/prog

bar/prog: bar/main.o foo/libfoo.a
    gcc -o bar/prog bar/main.o -Lfoo -lfoo

OBJS = foo/bla.o foo/xyzzy.o

foo/libfoo.a: $(OBJS)
    ....
```

Non-recursive make (2)

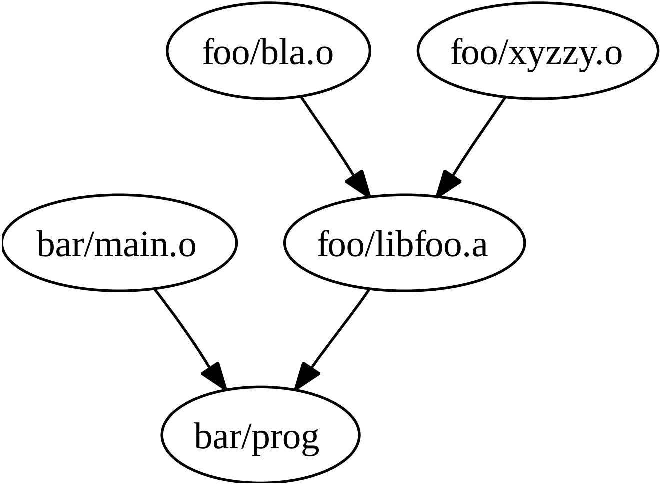

Figure 4: Build

Non-recursive make (3)

This is where we run into the limits of Make:

- We cannot specify different definitions for pattern rules for each directory (a common scenario).
- Similarly, variables are global to an entire Makefile.
- So if we need to compile with different flags, we have to write an explicit rule for each!

Variant builds

We often want to create multiple builds of a system:

- With or without debug info.
- A light or professional build.
- A build for each platform we support.
- ...

This is hard with Make – each build will overwrite the previous one.

Beyond Make

Some requirements for a modern build system

- Correctness and safeness (full dependencies, derive dependencies automatically)
- Efficiency, i.e., *incremental building*
- Tracking of derived files
- Support for variant builds, cross compilation, and parallel builds.
- Separation of tool specification and system model

Incremental building

A build should never do unneccessary work – recompiling unmodified files. Should the compiler track dependencies?

- Pro: compiler has accurate dependency knowledge.
- Con: most tools don't do this (some Java compilers do).
- Con: dependency checking mechanism has to be implemented in every tool.

Full dependencies?

How to determine full dependencies?
Solution 1: for each tool, write another tool that yields the list of dependencies (e.g., `gcc -MM` for `gcc`).
Problem: we rely on the existence of such a tool or need to write one by hand.

Full dependencies?

Solution 2: find out dependencies automatically by logging file system calls; i.e., all files opened by the tool are dependencies.
Problem: non-portable. Under Linux we can use the `ptrace` system call, under Mac OS `ktrace`, etc. This is the approach taken by Vesta and ClearCase to ensure safe and repeatable builds.

Are safe incremental builds possible?

Not really.

Assumption of most build systems: tools are pure functions that only depend on its arguments and on the contents of files. But:

- A tool can do impure things, such as let its result depend on the current time or by consulting the network.
- Tracking *all* the filesystem dependencies is also hard; e.g, what to do with directory access? (If a compiler accesses a directory, then changes to that directory should trigger recompilation).

For "reasonable" tools, the assumption works reasonably well.

Other build systems

- Many languages have their own build system/package manager: Rake (Ruby), Cabal/Shake (Haskell), Ant (Java), ...
- Many modern IDEs (such as Eclipse or Visual Studio) come with integrated build management – typically configured through a GUI or XML document.
- Many proprietary systems such as Vesta or ClearCase.

Shell scripting

Automation

Imagine you need

- to move around generated files,
- add a standard header template to your HTML documentation,
- or create scheduled backups of your software.

All of these tasks are easy to do by hand.
But it would be nice to automate them.

Scripting languages

Scripting languages are programming languages designed to facilitate the automation of such tasks.

Typically these tasks can be done by a human – but this does not scale well.

These examples aren't difficult calculations or applications.

The focus is on *convenience* and *automation*.

Scripting languages

- Python
- Lua
- Perl
- AppleScript
- Bash
- ...

Bash

Bash is a Unix shell, that allows you to execute commands from the terminal:

```
$ echo Hello world!
Hello world!
```

Many servers and machines can be scripted using Bash.

Bash: redirection

You can redirect output or input to/from files:

```
$ echo 'Hello world!' > README.md
$ cat README.md
Hello world!
$ echo 'Hello again!' >> README.md
Hello world!
Hello again!
```

Note: > writes to a file; >> appends to a file.

Creating and running scripts

```
$ cat hello.sh
#!/bin/bash
echo Hello!
$ chmod u+x hello.sh
$ ./hello.sh
Hello!
```

Using variables

Many system dependent variables are predefined in your environment.

```
$ env
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
...

$ echo Hello $USER
Hello wouter
```

Variable conventions

Variables start with a dollar sign; use single quotes to prevent variable substitution.

```
$ echo 'Hello $USER'
Hello $USER
```

You can define your own variables using export:

```
$export five=5
$ echo $five
5
```

Command expansion

```
$ export now=`date`
$ echo $now
Fri Feb 27 15:19:54 CET 2015
```

Enclose commands in backticks to evaluate them.

Simple control flow

Bash supports all kinds of simple control flow constructs, such as if-then-elses and loops.
Check if a file exists:

```
#!/bin/bash
if [ -f /var/log/messages ]
  then
    echo "/var/log/messages exists."
fi
```

Simple control flow

Or to iterate over all the files in a directory

```
#!/bin/bash
for i in `ls myproject/*.xml`; do
  echo The file $i has extension.xml
done
```

Bash

Combined with the available tools on Unix systems, bash lets you script all kinds of OS operations:

- `sed` for find/replace operations
- `awk` for extracting data
- `cd`, `cp`, `mv`, `rm` for file manipulation
- ...

Drawbacks

Bash scripts, like Makefiles, can quickly become complex and hard to maintain.

- There is very little type safety and almost no static checking – run it and hope for the best;
- It is easy to delete, overwrite, or lose data.

But still

- It can be incredibly useful to automate mundane tasks.

Request

Source of these slides can be found in on GitHub.

`https://github.com/wouter-swierstra/SoftwareProject`

I'm looking for tips to give next year's students:

- Useful shell scripts, how to set up the UU git server, mirroring GitHub and UU repositories, . . .

Pull requests welcome!

Next time

Presentations about your risks, architecture, and planning