A Make System in Common Lisp

Justin Lee

jlee@cs.utexas.edu

Spring 2005

Overview

- What Make does
- Why Lisp
- Make engine
- Building forms to run the engine

Make Example

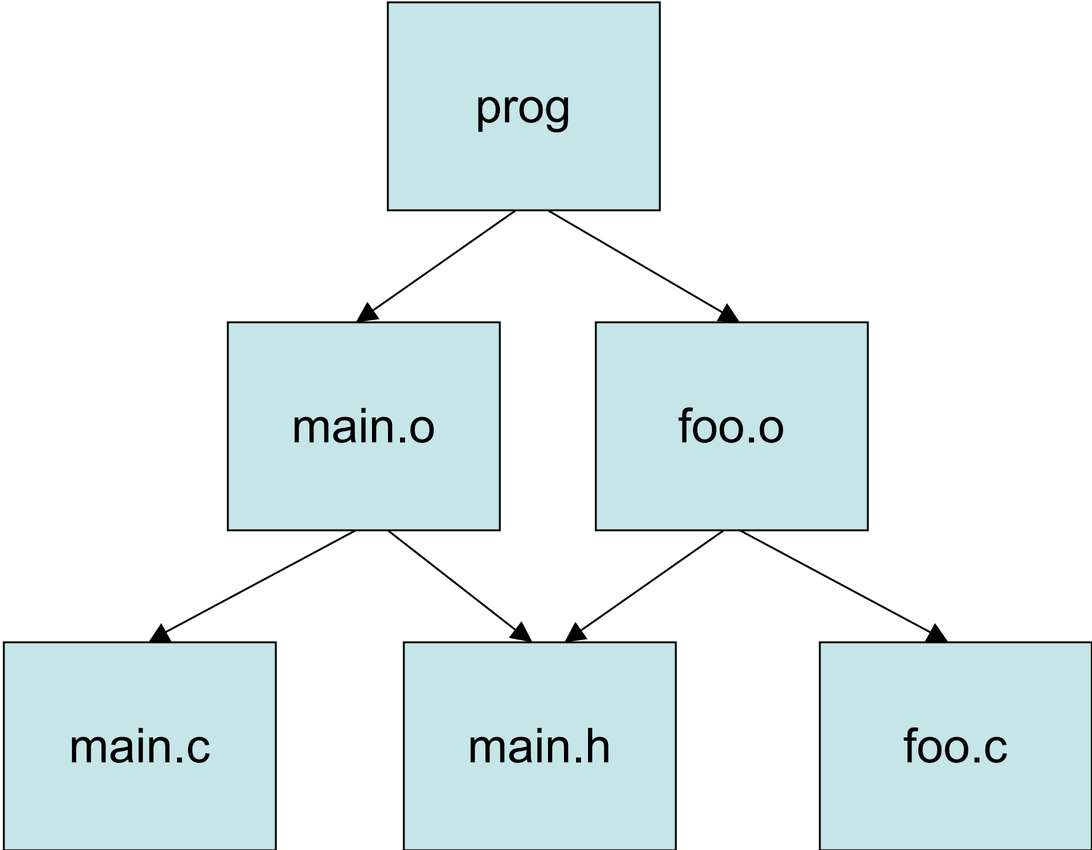
1. target
2. dependency list
3. action list

```
prog: main.o foo.o
    gcc main.o foo.o -o prog

main.o: main.c main.h
    gcc -c main.c

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

Conceptually a Dependency Tree



Running Make

```
$ make prog
gcc -c foo.c
gcc -c main.c
gcc main.o foo.o -o prog
$
```

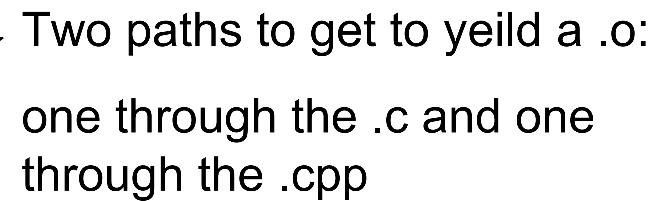
Pattern Rules

```
prog: main.o foo.o
	gcc main.o foo.o -o prog

%.o: %.c %.h
	gcc -c $<
```

Chained Pattern Rules

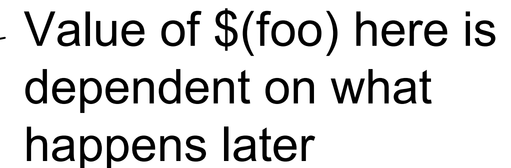
```
%.o: %.cpp %.h
	g++ -c $<
%.o: %.c %.h
	gcc -c $<
yacc/%.c: yacc/%.y
	cd yacc && yacc $<
```



Variable Expansion

- Variables are expanded at different times for target, rule and action parts of a rule

```
foo = $(bar)
bar = foo
prog: $(foo).c
    gcc -c $(foo).c
... more makefile here
```



Variable Types

- Two types: immediate and delayed

```
bar = bar
foo = $(bar)
bar = foo
# foo is “foo”
```

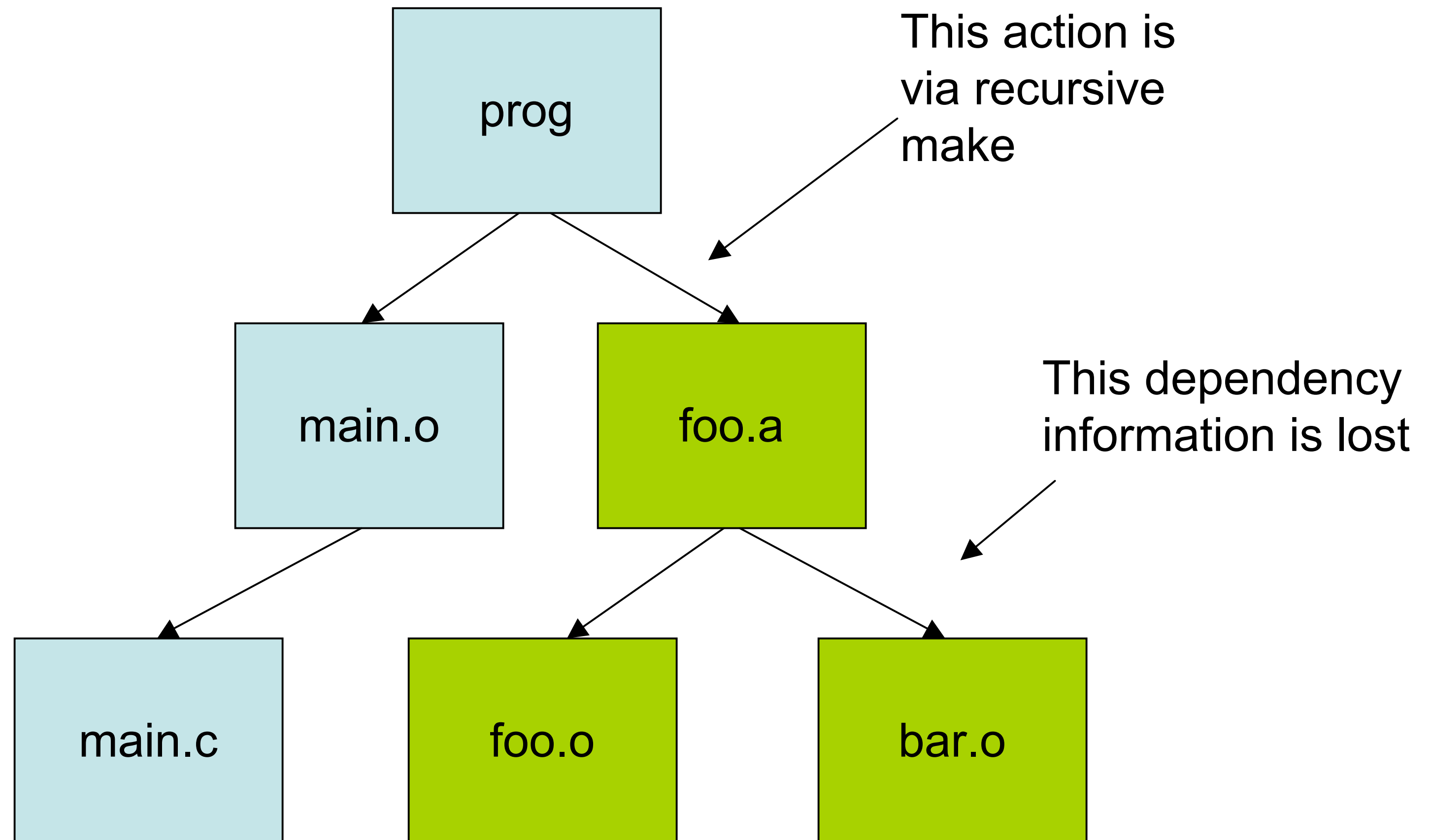
```
bar = bar
foo := $(bar)
bar = foo
# foo is “bar”
```

Existing Module Support

- GNU Make has an include feature
- Raw textual inclusion
- Delayed variable expansion in actions makes modules very painful since included modules can affect rules already defined
- “Recursive Make Considered Harmful”, by Peter Miller

`http://www.pcug.org.au/~millerp/rmch/recu-make-cons-harm.html`

Recursive Make



Multiple Inclusion and Ifeq

- Make supports preprocessor-like conditional constructs - `ifeq endif`
- Literal text inclusion
- To prevent multiple module inclusion have to play games with `ifeq`
- Using `ifeq` to support multiple platforms is painful

Included Modules and Pathnames

- May not know what directory the Makefile module is being run from
- Have carefully to set a “top” variable only once, then parameterize every single path with it
- Platform issues with / vs \
- Basic problem is pathnames are unstructured strings

Upshot

- Litany of problems with bigger make systems
- All of these problems are pragmatic problems, not a matter of expressiveness
- Redesign the language

Little Languages in Lisp

- Lisp is well suited to tree algorithms
- S-exps can be used as syntax

```
vector<vector<int> > f;
f.push_back(vector<int>());
f[0][0] = 1;
f[0][1] = 2;
…

'((1 2 3) (4 5 6) (7 8 9))
```

Little Languages in Lisp 2

- Functions can be constructed at runtime
  - Use this to emulate the late bind of variables
- Macros offer endless opportunities to clean up the syntax
- Stable and supported tools and libraries

Engine

- Most of the engineering work so far has been into the main build procedure
- Late bound variables are tricky, but necessary for pattern rules
- Chained patterns are subtle to implement properly
- Cl-ppcre regular expression library has been invaluable

Basic operations

- Turn a make-like pattern into a regex
- Basic file system operations (timestamps)
- Matching a target and substituting the result into the dependencies and actions
- Convert a string with variable references into a lambda
- Recursive decision on which pattern is applicable
  - Specificity concerns
- Recursive build procedure

Late bindings

A s-exp make rule for building object files in a lib directory from a c file:

```
((“lib/%.o”)
 (“lib/%.c”)
 (“gcc -c $[deps] -o $[target]”))
```

Could just use an environment-passing style a-list with variables in it.

But…

But what about this?

```
((“lib/%.o”)
 (“lib/%.c” “lib/%.h”)
 (“gcc -c $[(car deps)] -o $[target]”))
```

Instead of having a special variable for the first dependency, allow arbitrary lisp expressions.

The action string then compiles to:

```
(lambda ()
  (declare (special (deps target)))
  (strcat “gcc -c ” (car deps) “ -o ” target))
```

This must be generated at runtime to avoid macro hell.

Building

Is there a direct rule for the target?
  then recurse on each dependency
    if any ts is newer than target
      execute actions, return ts
    else return 0
Is there any matching and buildable pattern?
  same as above
Does the file exist on disk? Return ts
Else error

Finding an applicable pattern

- It isn’t enough to find a matching pattern
- The dependencies must be buildable too
- Example:

```
((“%.o”) (“%.c”) (gcc “$[deps]”))
((“%.o”) (“%.cpp”) (gcc “$[deps]”))
```

- If the c or cpp file is generated from another program (e.g. yacc) then we must execute this procedure recursively until we get direct rules or files on disk

Concrete Syntax

- Makefile can be as simple as:

```
(compile-tree '((("foo.o") ("foo.c"…
                 rule2
                 pattern-rule1))
```

- But we may want to build something more like a real language. Example which installs a case-sensitive reader-macro:

```
(defrule foo.o (foo.c foo.h)
  (gcc -c $target -o $(car deps)))
```

- Most usages of variables are taken care of since we are embedded inside Lisp
- We can use the full power of Lisp functions and second-order constructs to keep our Makefiles crisply written

“Programmable Programming Language”

- Make syntax does not allow % substitution in actions
- We could write a macro that transforms a percent sign into some horrible lisp expression to calculate the string from the target
- Lets us write:

```
(transform-percent
 ((“%.o”)
  (“%.c”)
  (“gcc –c %.c –o %.o”))))
```

Things to do

- Pathnames are still unstructured strings
  - Better to have some notation (possible logical pathstrings from CL) to describe them before producing concrete strings
- Module system
  - Needs some way of defining toplevel constructs like “clean” and “debug” only once
  - Cooperate with the pathname system so that pathnames can always be written relative to the location of the Makefile

More things to do

- A tool for automatically generating header dependencies in C/C++
- Cross platform functions for generating actions for gcc, Microsoft and Intel compilers (use CLOS)
- Refactor the main algorithm so it can be modified – e.g. use SHA1 comparisons against a database instead of timestamps