

Lecture 9: Make Pattern Matching & Conceptual Integrity

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2004

Pattern Matching, set-up

- Below is a fairly standard makefile.
- What would you do if you want to change your compiler to gcc and add compiler flags such as -g?

```
program: main.o input.o output.o
    g++ $^ -o $@
main.o: main.cpp defs.h
    g++ -c $<
input.o: input.cpp defs.h
    g++ -c $<
output.o: output.cpp defs.h
    g++ -c $<
```

September 20, 2004

© University of Colorado, 2004

2

Pattern Matching, set-up, cont.

- Use Macros of course!

```
CXX = g++
CFLAGS = -c -g
program: main.o input.o output.o
    $(CXX) $^ -o $@
main.o: main.cpp defs.h
    $(CXX) $(CFLAGS) $<
input.o: input.cpp defs.h
    $(CXX) $(CFLAGS) $<
output.o: output.cpp defs.h
    $(CXX) $(CFLAGS) $<
```

September 20, 2004

© University of Colorado, 2004

3

Pattern Matching, example

- Did you notice how in all cases, our rules for compiling each file were exactly the same, except for the file name?

```
main.o: main.cpp defs.h
    $(CXX) $(CFLAGS) $<
input.o: input.cpp defs.h
    $(CXX) $(CFLAGS) $<
output.o: output.cpp defs.h
    $(CXX) $(CFLAGS) $<
```
- Make has a mechanism for capturing these similarities, called pattern matching
 - This is make's **third** abstraction mechanism

September 20, 2004

© University of Colorado, 2004

4

Pattern Matching

- We can capture similarities between rules based on file suffixes
- Thus our rules in the previous examples that took care of compiling files can be expressed as

```
%o: %.cpp
$(CXX) $(CFLAGS) $<
```

- This is not exactly the same, why? Does it matter?

Pattern Matching in Make

- Make supports pattern matching through the presence of the character “%” in rules

```
%.o: %.c
g++ -c $<
```

- If you type “make input.o” the rule becomes

```
input.o: input.c
g++ -c $<
```

- Note: automatic variables are required. Why?

Benefits of Pattern Matching

- Scalability
 - The same rule can apply to thousands (or more) of files
- Compactness
 - Small compact specifications are easier to understand and debug
- These are similar to the benefits of wildcards and regular expressions
 - which should come as no surprise

More on Pattern Matching

- Pattern matching in make is not exactly like wildcards in the shell
 - Pattern matching rules do not try to match every possible file name
 - Instead, they only execute if there is a dependency that needs to be created that matches the rule
- Lets look at an example

Pattern Matching Example

```
program: program.o
    g++ $^ -o $@
%.o: %.c
    g++ -c $<
```

- If you type “make program”, make will look for “program.o”. This matches the “%.o” rule, so make will execute “program.o: program.c”
- You may have other .c files in the directory, but they will not be made into .o files **unless they are specifically mentioned** in the makefile

Pattern Matching Rules as Goals

- As a result, **make cannot run a pattern matching rule, unless it is explicitly told to do so** (via the command line) or in response to a dependency of another rule
- Therefore, if the first rule in a makefile is a pattern matching rule, make **skips over it and looks for the first non-pattern matching rule**
 - (But only when you type “make” at the command line with no other command line arguments)

Pattern Matching Rules as Goals

```
%.o: %.c
    g++ -c $<
program: program.o
    g++ $^ -o $@
```

- Typing “make” for the above makefile, causes the program rule to be executed, the pattern matching rule is ignored (even though it comes first)

Pattern Matching Example, cont.

```
program: program.o
    g++ $^ -o $@
%.o: %.c
    g++ -c $<
```

- Continuing our example, if you typed “make input.o” with this makefile, the pattern matching rule would be used to create “input.o” from an “input.c”
 - even though “input.o” is not explicitly mentioned in the makefile
 - but only if “input.c” exists!



Suffix Rules

- A variation on pattern matching rules are suffix rules. The following two rules are equivalent

```
%o: %c
```

```
g++ -c $<
```

```
.c.o:
```

```
g++ -c $<
```

- Note the reversed order of the suffixes



Implicit Rules

- Make's abstraction mechanisms...
 - Pattern matching rules
 - automatic variables
 - macros
- ...make it possible to have a common set of rules automatically defined by make
 - These rules are called "implicit rules"
 - Make's implicit rules are available in the "reference materials" section of the class website



Implicit Rule Example

- If you create a makefile that contains just the following rule:

```
program: program.o
```

```
$(CC) $(CFLAGS) $^ -o $@
```

- Make will act as if you had also included the following rule

```
.c.o:
```

```
$(CC) $(CFLAGS) -c $<
```



Brooks' Corner: Conceptual Integrity

- Brooks example => Cathedrals
 - Many cathedrals consist of contrasting design ideas
 - The Reims Cathedral was the result of eight generations of builders repressing their own ideas and desires to build a cathedral that embodies the key design elements of the original architect!
- With respect to software
 - Design by too many people results in conceptual disunity of a system which makes the program hard to understand and use.



Conceptual Integrity

- Brooks considers it the most important consideration in system design
 - Better to leave functionality **out** of a system rather than break the conceptual integrity of the design
- Questions
 - How is conceptual integrity achieved?
 - Does conceptual integrity give too much power to system designers?



Function vs. Complexity

- The key test to a system's design is the ratio of functionality to conceptual complexity
 - Ease-of-use is enhanced only if a function provides more power than it takes to learn (and remember!) how to use the function



Function vs. Complexity, cont.

- Neither function or simplicity alone is good enough
 - OS/360 had lots of functionality
 - PDP-10 had lots of simplicity
 - Both reached only half of the target!
 - You must be able to specify your intentions with simplicity and straightforwardness; if your elements are too simple, then complex tasks will not be straightforward to specify!
- Brooks claims that adhering to the notion of conceptual integrity can help you achieve the proper balance of functionality and complexity
 - Ease of use requires unity of design, i.e. conceptual integrity



Architects as Aristocrats

- Conceptual Integrity requires that the design be the product of one mind
- The architect (or surgeon) has ultimate authority (and ultimate responsibility!)
 - Does this imply too much power for the architects?
 - In one sense, yes, but ease-of-use of a system comes from conceptual integrity!
 - In another sense, no, the architect sets the structure of the system, developers can then be creative in how the system is implemented!
 - Indeed, some initial constraints can help focus the creativity since the architect has taken care of the "key" design decisions.