The Make language

Recipes and the shell

Each line of a recipe (that is, each line starting with a tab) is interpreted by `/bin/sh`, which is some kind of POSIX shell (but not always Bash).

But it's not quite like a shell script:

- Each line runs in its own shell invocation
  - Shell state, like variables and the working directory, reset with each line.
- Make does its own preprocessing of each line:
  - Strips the leading tab
  - Interprets the character `$` as a Make variable substitution (see next slide); for a literal `$`, use `$$`
  - Allows *line continuations* by ending a line with a backslash

The standard shell quoting rules still apply! Use quotes whenever an argument might have spaces, even if it's a Make variable!

```
# For shell variables, "$"
# must be escaped as "$$":
myname:
  echo "$$USER" >myname

# Doesn't work! Working
# directory resets each line.
cat-passwd-broken:
  cd /etc/
  cat passwd

# Works; all one shell.
cat-passwd-1:
  cd /etc/ ; cat passwd

cat-passwd-2:
  cd /etc/ ; \
  cat passwd

.PHONY: cat-passwd-broken \
  cat-passwd-1 cat-passwd-2
```

The two phases of `make`

Although Make parses variable and rule definitions sequentially (the *read-in phase*), it doesn't actually *run* any recipes (the *target-update phase*) until it's parsed everything and decided which rules are out-of-date.

- Read-in phase
  - Reads Makefile(s)
  - Internalizes all variables and rules
  - Expands simply-expanded (`:=`) variables
  - Builds a dependency graph of targets and prerequisites
- Target-update phase
  - Determines which targets need to be updated
  - Runs recipes necessary to update the targets
  - Expands recursively-expanded (`=`) variables used in recipes

Variables

Two types of variable assignment, each of which appear on a line of their own:

- ***Simply expanded***: `NAME := val`
  - Variable references and function calls in `val` happen when the variable is *defined* (read-in phase).
  - Works like assignments in nearly every other language.
  - When in doubt, use these.
- ***Recursively expanded***: `NAME = val`
  - Variable references and function calls in val happen when the variable is *used* (target-update phase for recipe usage).
  - Allows complex templates to be stored in a variable and used with many different values.
  - Can lead to infinite recursion if you're not careful.

Expansions can occur anywhere, and use `$(VAR)` syntax.

```
EXE := main

$(EXE): main.o log.o
  gcc main.o log.o -o $(EXE)

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

log.o: log.c log.h
  gcc -c log.c

.PHONY: clean
clean:
    rm -rf *.o $(EXE)
```

Automatic variables

These variables can only be used inside recipes and are automatically set based on the rule the recipe belongs to.

The manual has a full list, but you'll see these three most frequently:

$@: Name of the target of the rule.

$^: Names of all the prerequisites, separated by spaces.

$<: Name of the first prerequisite

```
EXE := main

$(EXE): main.o log.o
  gcc $^ -o $@

main.o: main.c log.h
  gcc -c $< -o $@

log.o: log.c log.h
  gcc -c $< -o $@

.PHONY: clean
clean:
    rm -rf *.o $(EXE)
```

Simple vs recursive expansion

```
LOG_TARGET = @echo "Current target: $@" ; echo "Prerequisites: $^"

CFLAGS := -Wall
CFLAGS := $(CFLAGS) -Werror

main: main.o log.o
  $(LOG_TARGET)
  gcc $(CFLAGS) $^ -o $@

main.o: main.c log.h
  $(LOG_TARGET)
  gcc $(CFLAGS) -c $< -o $@

log.o: log.c log.h
  $(LOG_TARGET)
  gcc $(CFLAGS) -c $< -o $@
```

Functions

Make has many built-in *functions*, which allow you to process text directly from within Make.

You'll usually use functions when specifying variables, targets, or prerequisites: inside a recipe itself, it's almost always better to use the shell to do any text processing you need.

Function invocations take one or more comma-separated arguments, which is what distinguishes them from variable expansions. They otherwise have identical `$()` syntax.

Functions are are (in general) evaluated the same way as variables are expanded—at the time they're used in a rule or a `:=` assignment.

```
# world
x := $(word 2,hello world)

# hi world
x := $(subst \
    hello,hi,hello world)

# src/foo.c src/bar.c
objs := bld/foo.o bld/bar.o
x := $(patsubst \
    bld/%.o,src/%.c,$(objs))

# foo.c bar.c
srcs := foo.c bar.c bar.h
x := $(filter %.c,$(srcs))
```

Make variables and functions are textual

Like the shell, Make has no concept of data types. Everything is text, and Make's minimal escaping and tolerance for spaces can lead to some pretty odd constructs. The following is taken verbatim from the GNU Make manual:

```
comma:= ,

empty:=
space:= $(empty) $(empty)

foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now ‘a,b,c’.
```

Pattern rules

```
%.o: %.c %.h
```

Add the missing header dependency to the implicit rule

```
%.txt:
    echo "Hello I am $@" > $@
```