Makefiles for Fun and Profit*

Boris Veytsman

June 7, 2001

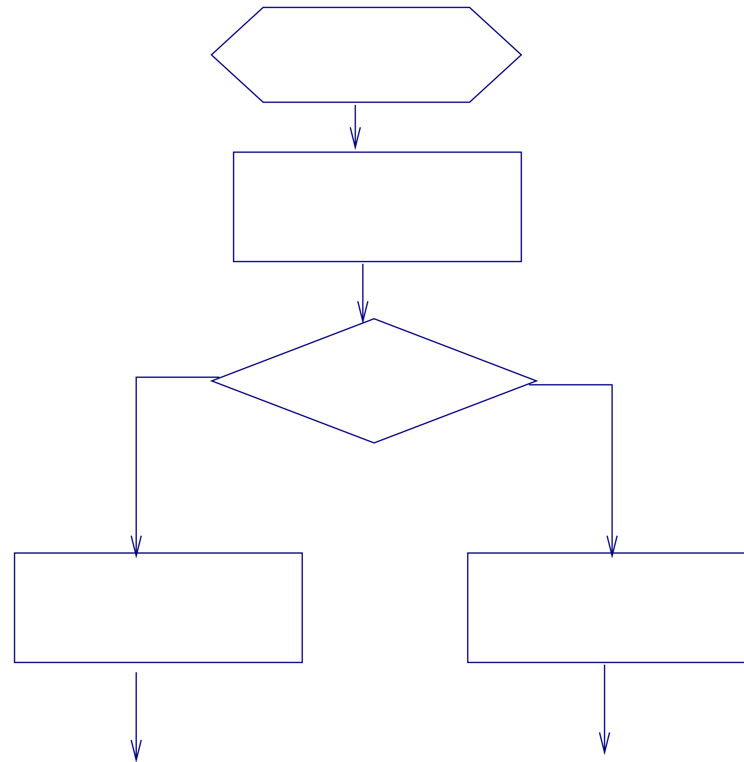
*This document contains lecture notes for informal Unix seminar for ITT AES employees (Reston, VA). No information in this document is either endorsed by or attributable to ITT. This document contains no ITT Privileged/Proprietary Information.

Why Makefiles?

A language that doesn't affect the way you think about programming is not worth knowing

ITT Industries
Engineered for life

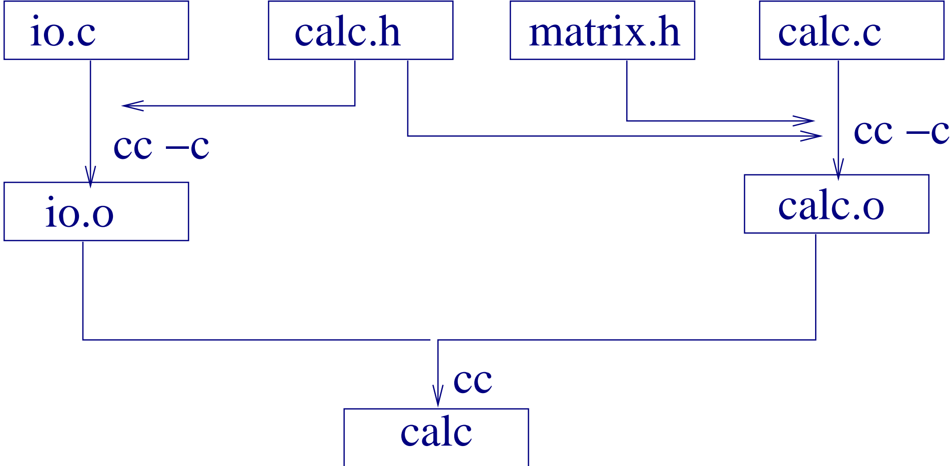
A common flowchart:



A tree branching *down*

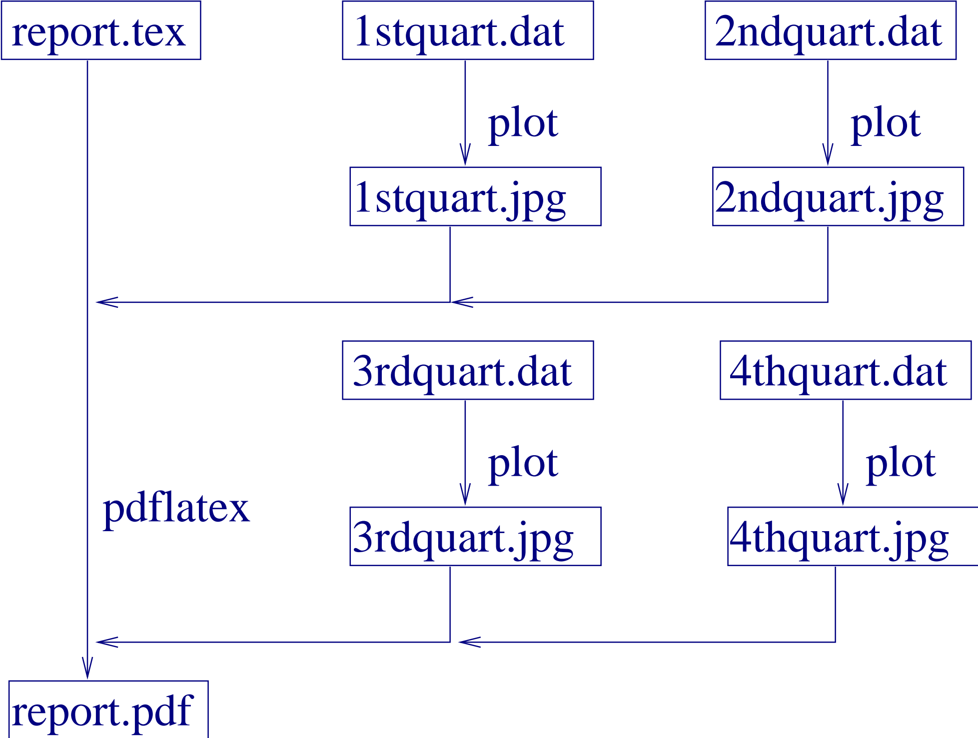
ITT Industries
Engineered for life

A programming flowchart:



A tree branching *up*!

A report making flowchart:



ITT Industries
Engineered for life

Make Algorithm

> The algorithm to do that is extremely nasty. You might want to mug someone with it. *M. Devine, Computer Science 340*

Targets: what you want to make

Prerequisite: what you need

Dependency: prerequisites for a given target

Rule: how to make a target

1. Traverse the tree up, finding prerequisites, their prerequisites, etc.

2. Traverse the tree down, making targets *unless* they are newer than prerequisites.

Timing is important:

```
touch source.c
```

will trigger a rebuild

ITT Industries
Engineered for life

Which Make?

> The chat program is in public domain. This is not the GNU public license. If it breaks then you get to keep both pieces. *Copyright notice for the chat program*

BSD make: Broken. Do not use

SysV make: Functional

GNU make: A gift from heaven. Versatile, intelligent, caring and loving partner for your work needs

A Simple Makefile

Today's robots are very primitive, capable of understanding only a few simple instructions such as "go left", "go right", and "build car". *John Sladek*

Dependency line:

```
target: prerequisite prerequisite ...
```

Rule line:

```
[TAB]   command
```

Do not forget TABs!

Usage:

```
make [target] [-f file]
```

```
all: calc

calc: io.o calc.o
        cc io.o calc.o -o calc

io.o: io.c calc.h
        cc -c io.c

calc.o: calc.c calc.h matrix.h
        cc -c calc.c

clean:
        rm -f calc calc.o io.o core
```

1. Too much typing here
2. Easy to make a mistake
3. The structure is not evident

Variables

> One man's constant is another man's variable. *A.J. Perlis*

1. Look like shell variables—but braces are *obligatory*

2. Actually you *can* use shell variables!

ITT Industries
Engineered for life

```
SRCS = calc.c io.c
OBJS = calc.o io.o

all: calc

calc: ${OBJS}
        cc ${OBJS} -o calc

io.o: io.c calc.h
        cc -c io.c

calc.o: calc.c calc.h matrix.h
        cc -c calc.c

clean:
        rm -f calc ${OBJS} core
```

ITT Industries
Engineered for life

Substitution and Suffix Rules

An amendment to a motion may be amended, but an amendment to an amendment to a motion may not be amended. However, a substitute for an amendment to and amendment to a motion may be adopted and the substitute may be amended. *The Montana legislature's contribution to the English language*

ITT Industries
Engineered for life

```
SRCS = calc.c io.c
OBJS = ${SRCS:%.c=%.o}

.SUFFIXES: .c .o

.c.o:
	cc -c $< -o $@

all: calc

calc: ${OBJS}
	cc ${OBJS} -o calc

io.o: calc.h

calc.o: calc.h matrix.h

clean:
	rm -f calc ${OBJS} core
```

Some variables:

$<: What you have

$@: What you want

GNU make has much better *pattern* mechanism:

```
%o: %.c
        cc -c $< -o $@
```

ITT Industries
Engineered for life

Why Do You Not Need Suffix Rules

Computers are not intelligent. They only think they are.

A good *make* knows a lot itself! (about 400 rules)

```
SRCS = calc.c io.c
OBJS = ${SRCS:%.c=%.o}

all: calc

calc: ${OBJS}

io.o: calc.h

calc.o: calc.h matrix.h

clean:
	rm -f calc ${OBJS} core
```

Dependencies and makedepend

System-independent, adj.: Works equally poorly on all systems.

Objects depend on header files. How to automatically account for this?

Solution 1: Use variables:

```
HEADERS = calc.h matrix.h
${OBJS}: ${HEADERS}
```

Problem: we *overshoot* here! No fine control

Solution 2: use makedepend (or cc -M)

```
makedepend calc.c io.c
```

Makedepend adds to the Makefile a lot of lines:

```
# DO NOT DELETE

calc.o: calc.h matrix.h
io.o: calc.h /usr/include/stdio.h /usr/include/features.h
io.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
io.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stddef.h
io.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stdarg.h
io.o: /usr/include/bits/types.h /usr/include/libio.h /usr/include/_G_config.h
io.o: /usr/include/bits/stdio_lim.h
```

Recursive scanning of all #include!

```
SRCS = calc.c io.c
OBJS = ${SRCS:%.c=%.o}

all: calc

calc: ${OBJS}

depend:
	makedepend ${CFLAGS} ${SRCS}

clean:
	rm -f calc ${OBJS} core

[automatically generated stuff]
```

Version with cc -M:

```
SRCS = calc.c io.c
OBJS = ${SRCS:%.c=%.o}

all: calc

calc: ${OBJS}

depend: ${SRCS}
        ${CC} -M ${CFLAGS} ${SRCS} > depend

clean:
        rm -f calc ${OBJS} core

include depend
```

Standard Variables

The primary purpose of the DATA statement is to give names to constants; instead of referring to π as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of π change. *FORTRAN manual for Xerox Computers*

ITT Industries
Engineered for life

1. Can be set in Makefile:

```
CC = gcc
LFLAGS = -lm
```

2. Can be set in the shell:

```
setenv CC gcc
make all
```

Some useful variables:

ASFLAGS: Options for assembler

AS: Assembler

CFLAGS: Options for C

CC: C compiler

LDFLAGS Options for linker

LD: Linker

LDLIBS: Libraries

1. How to compile everything -O3 flag?
2. How to compile everything *but* fragile.c with -O3 flag?

How Does Make Interpret Commands

An interpretation I satisfies a sentence in the table language if and only if each entry in the table designates the value of the function designated by the function constant in the upper-left corner applied to the objects designated by the corresponding row and column labels. *Genesereth & Nilsson, "Logical Foundations of Artificial Intelligence"*

ITT Industries
Engineered for life

Each line is executed in a subshell (imagine logout & login after each line).

This will *not* work:

```
cd sources
do_something
```

This will:

```
cd sources; do_something
```

```
cd sources;\
do_something
```

Which shell?

1. Some versions—always */bin/sh*
2. Some versions—the value of *$SHELL*

Good practices:

1. Write for */bin/sh*
2. Put in the Makefile

```
SHELL = /bin/sh
```

Use *$$* for shell *$*: make will eat one *$*

ITT Industries
Engineered for life

Errors stop execution! If you want to continue despite errors:

Global solution: use make -i (ignore all errors)

Fine-grained solution: use - in the Makefile:

```
install: all
        - ${INSTALL} -m 644 ${MAN} ${MANDIR}
        ${INSTALL}  ${PROGRAM} ${BINDIR}
```

ITT Industries
Engineered for life

Multiple Directories and Recursive Make

To iterate is human, to recurse, divine. *Robert Heller*

Files in many directories? Use *VPATH*:

```
VPATH = ./src:/usr/src:/projects/src
```

ITT Industries
Engineered for life

Another solution: recursive make

```
DIRS = io calcs
all:
        for x in ${DIRS}; do \
          cd $$x; \
          make all;\
          cd ..;\
        done
```

Why do we need double $ and backslashes here?

The Last Example

Don't try to have the last word—you might get it. *Lazarus Long*

```
LK = borisv@capital.lk.net:/home/borisv/public_html/unix
RSYNC = rsync -v -e ssh
GIFS = logohome_trans.gif
HTMLS = index.html \
        2_manpages/position.html \
        2_manpages/position.pod
PDFS = 1_introduction/1_introduction.pdf\
       2_manpages/2_manpages.pdf\
       3_files/3_files.pdf\
       4_dirs/4_dirs.pdf \
       5_makefiles/5_makefiles.pdf


SRCS = $(GIFS) $(HTMLS) $(PDFS)


all: $(SRCS)
       ${RSYNC}  $(SRCS) $(LK)
```