Macro Example

- int max (int c, int a, int b) { vs . **#define** *max(c,a,b) (c = ((a>b) ? a:b))*

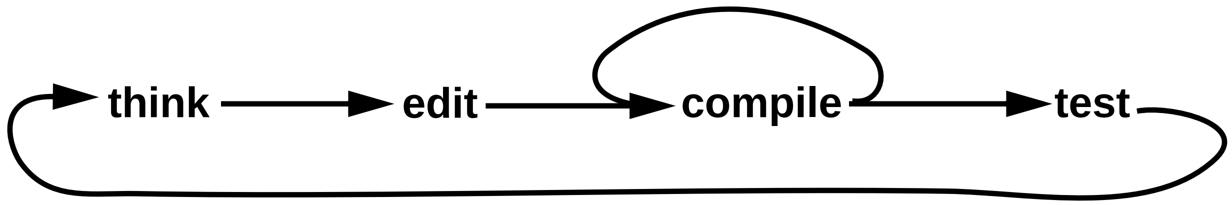
```
      c = (a>b) ? return a : return b;
}
```

- int main () {

```
       int x=3,y=5,z=0;
       max (z,x++,y++);
      printf (“max of x=%d and y=%d is %d\n”, x,y,z);
}
```

Make

- Typical program development cycle



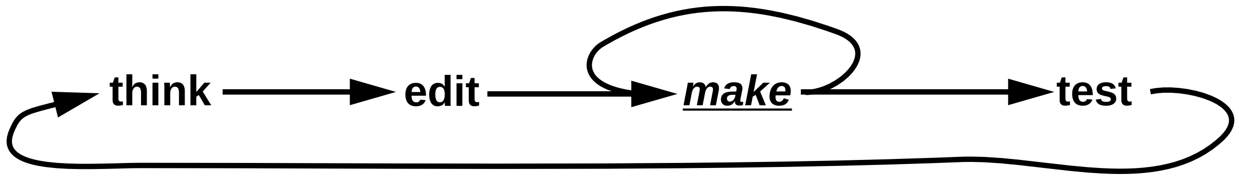
- Potential problems

  edit a file, but forget to compile it

  edit an interface, but forget to compile ***all*** the files that ***depend*** on it

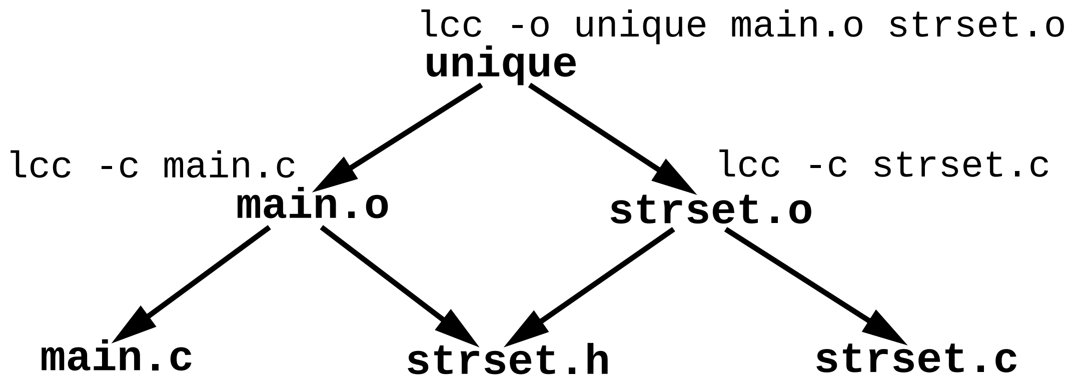
  do more compilation than is necessary

- **`make`** ***automates*** compiling and building a program



Dependency Graphs

- make processes a ***dependency graph***



  each node represents a ***file***

  each node is annotated with the ***command*** that “makes” the file

- To make node ***X***

  make all dependents of ***X*** (those ***modified more recently*** than ***X)***

  update ***X*** using the associated command

  if **`strset.h`** or **`main.c`** is newer than **`main.o`**

  re-make **`main.o`** with “**`lcc -c main.c`**”

Makefiles

- **`makefile`** or **`Makefile`** **specifies the dependency graph of make**

  targets: dependents
  commands

```
unique:    main.o strset.o
           lcc -o unique main.o strset.o

main.o:    main.c strset.h
           lcc -c main.c

strset.o:  strset.c strset.h
           lcc -c strset.c
```

- To invoke **`make`**

  `make` ***targets*** ...

```
make strset.o
make unique
```

- With no arguments, **`make`** makes the ***<u>first</u>*** target listed in **`makefile`**

```
% make
lcc -c main.c
lcc -c strset.c
lcc -o unique main.o strset.o
```

```
% touch strset.c
% make strset.o
lcc -c strset.c
% make
lcc -o unique main.o strset.o
```

Built-ins and Macros

- **`make`** contains ***built-in*** dependencies and commands

  a “**`.o`**” file is assumed from a “**`.c`**” file by the C compiler

```
unique:     main.o strset.o
            lcc -o unique main.o strset.o

main.o strset.o: strset.h
```

- **`make`** has a simple ***macro*** facility; macros communicate with built-in commands and simplify **`makefile`**s

```
CC=lcc -A
CFLAGS=-g
LDFLAGS=-g
STRSET=strset0
OBJS=main.o $(STRSET).o

a.out: $(OBJS)
       $(CC) $(LDFLAGS) $(OBJS)

$(OBJS):  strset.h
```

```
% make -n
lcc -A -g -c main.c
lcc -A -g -c strset0.c
lcc -A -g main.o strset0.o
% make -n STRSET=strset1
lcc -A -g -c main.c
lcc -A -g -c strset1.c
lcc -A -g main.o strset1.o
% setenv STRSET strset1
% make -e
lcc -A -g -c main.c
lcc -A -g -c strset1.c
lcc -A -g main.o strset1.o
%
```

Dummy Targets, Prefixes, and Built-in Macros

- “Dummy” targets for common command sequences

```
install: a.out
         cp a.out unique
         strip unique

clean:
         -rm *.o core

clobber: clean
         rm -f a.out unique
```

“-” prefix ignores errors

make clean removes “**.o**” and **core** files

- Dummy targets can be created if only for their modification time

```
FILES=main.c strset.h strset0 strset1.c
...
print:   $(FILES)
         @enscript $?
         @touch print
```

$? macro expands into “younger” dependents

“@” prefix suppresses command echoing

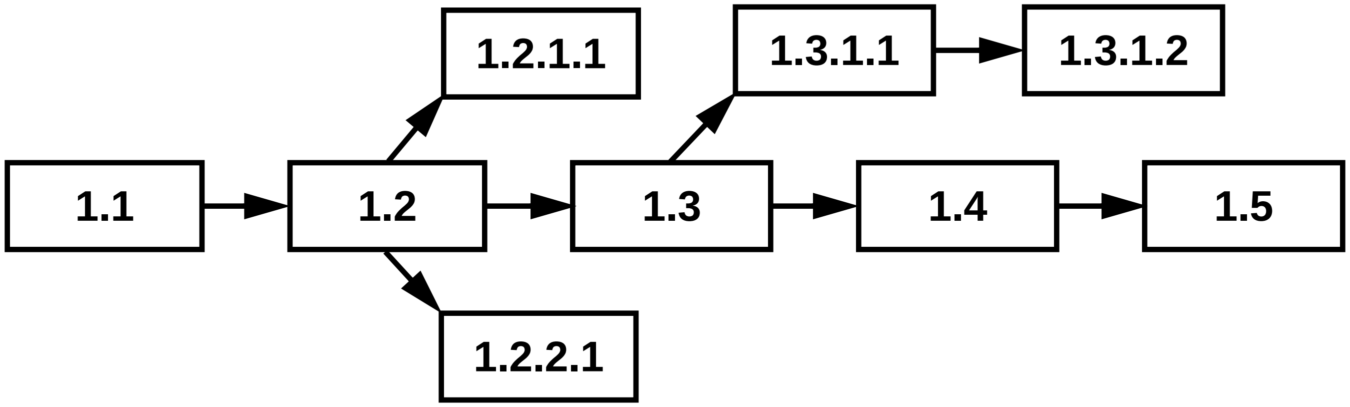
- Use dummy targets for all “program maintenance” tasks

```
clean      install    print
release    submit     test
```

- **Don’t *overuse* dummy targets and macros**

Version-Control Tools

- Software systems ***evolve*** — they advance in steps or ***versions***
  - repair bugs
  - add performance improvements and new features
  - add versions for other platforms (SPARC, ALPHA, x86, ...)
- Might have to retrieve ***old*** versions
- ***Version-control*** tools help maintain versions of programs, or any files
- ***Revision trees***



Why Revision Control

- Store and retrieve multiple versions of a file
- Maintain a history and log of changes
- Resolve access conflicts
- Maintain a tree with separate paths
  - can merge paths as well
- Control releases and their status
- Reduce storage

<u>R</u>evision <u>C</u>ontrol <u>S</u>ystem

- "Checking in" a file creates a new version, including the initial version

  `ci main.c`

  creates the version file **`main.c,v`** that holds **`main.c`** as version 1.1

  deletes **`main.c`**

- "Checking out" a file retrieves a copy of the latest version

  `co main.c` checks out a ***<u>read-only</u>*** copy
  `lcc -c main.c`

  `co -l main.c` checks out a ***<u>read/write</u>*** copy, ***<u>locks</u>*** **`main.c,v`**
  `emacs main.c`
  `lcc -c main.c`
  `ci main.c` checks in new **`main.c`** as version 1.2

- Options specify explicit versions for **`co`** and **`ci`**

  `co -r1.2 main.c` checks out a read-only copy of version 1.2 **`main.c`**
  `co -l1.2 main.c` checks out a read/write copy of version 1.2 **`main.c`**
  `ci -r2 main.c` checks in a new "release" of **`main.c`**

Branching

- Branching occurs to fix bugs, enhance old versions, ...
-
  $$\begin{tikzcd}\boxed{1.1}\end{tikzcd}$$

  ```
  ci main.c
  co -l main.c; emacs; ... ; ci main.c
  ```

  $$\begin{tikzcd}\boxed{1.1} \arrow[r] & \boxed{1.2}\end{tikzcd}$$

  ```
  co -l main.c; emacs; ... ; ci main.c
  ```

  $$\begin{tikzcd}\boxed{1.1} \arrow[r] & \boxed{1.2} \arrow[r] & \boxed{1.3}\end{tikzcd}$$

  ```
  co -l main.c; emacs; ... ; ci -r2 main.c
  ```

  $$\begin{tikzcd}\boxed{1.1} \arrow[r] & \boxed{1.2} \arrow[r] & \boxed{1.3} \arrow[r] & \boxed{2.1}\end{tikzcd}$$

  ```
  co -l main.c; emacs; ... ; ci main.c
  ```

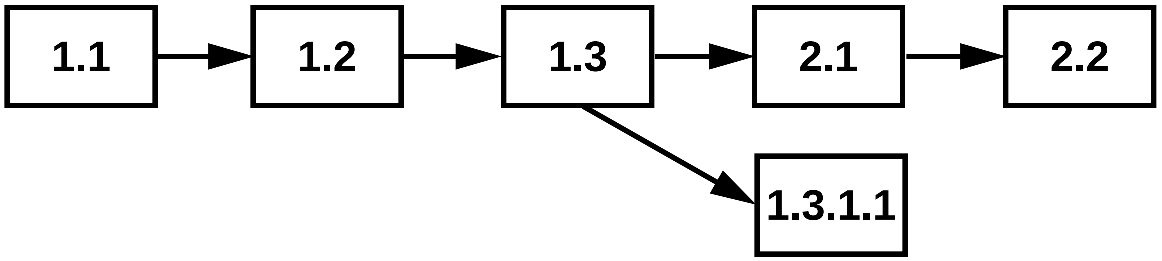
  $$\begin{tikzcd}\boxed{1.1} \arrow[r] & \boxed{1.2} \arrow[r] & \boxed{1.3} \arrow[r] & \boxed{2.1} \arrow[r] & \boxed{2.2}\end{tikzcd}$$

- What if you would like to fix and enhance version 1.3?

Branching, cont'd

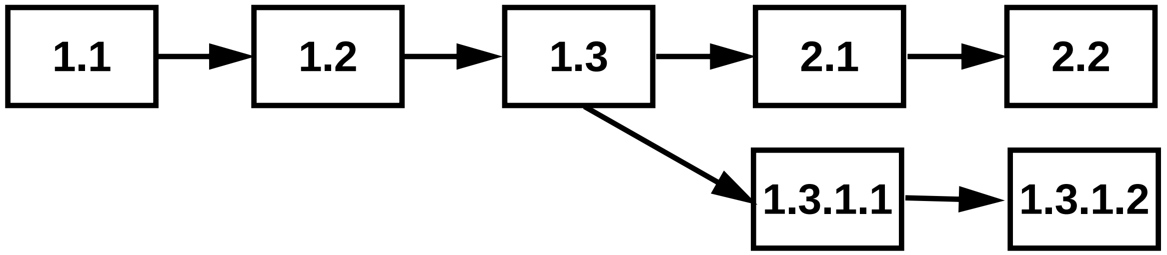
- Create a ***branch*** at version 1.3

```
co -l1.3 main.c; emacs; ... ; ci -r1.3.1 main.c
```



- Extra revision number in 1.3.1.***1*** allows for subsequent revisions

```
co -l1.3.1 main.c; emacs; ... ; ci -r1.3.1 main.c
```



- See RCS man pages for information on more options, commands, ...

Using RCS with Make

- Using RCS with **`make`**

  `*.c` depends on **`*.c,v`**

  ```
  main.c:    main.c,v
             co main.c
  ```

  RCS automatically looks in the directory **`RCS`** for **`,v`** files

  ```
  main.c:    RCS/main.c,v
             co main.c
  ```

  “**`make clobber`**” should remove **`.c`** files

  ```
  clobber:   clean
             rm -f wf main.c parse.c table.c
  ```

  or, if **`rcsclean`** is available

  ```
  clobber:   clean
             rm -f wf; rcsclean *.[ch]
  ```

- **Revised program development cycle**

$$\begin{tikzcd}
\underline{\textbf{\textit{check out}}} \arrow[r] & \textbf{think} \arrow[r] & \textbf{edit} \arrow[r] & \textbf{make} \arrow[r] & \textbf{test} \arrow[r] \arrow[lll, bend left] & \underline{\textbf{\textit{check in}}}
\end{tikzcd}$$

RCS Implementation

- Revisions are stored in the version file in ***differential form***

  if **`main.c`** has the revision tree

$$\begin{tikzcd} \boxed{1.1} \arrow[r] & \boxed{1.2} \arrow[r] & \boxed{1.3} \end{tikzcd}$$

`main.c,v` holds	all of version 1.3
	edit script to convert 1.3 to 1.2
	edit script to convert 1.2 to 1.1

- RCS revisions are ***backward deltas*** . Why?

- Other systems,such as SCCS use ***forward deltas***

version file holds	all of version 1.1
	edit script to convert 1.1 to 1.2
	edit script to convert 1.2 to 1.3

- Deltas are computed with ***"diff"***

```
diff -e main.old main.c
```

  generates **`ed`** commands to edit **`main.old`** into **`main.c`**

  see Section 5.9 in Kernighan and Pike, The UNIX Programming Environment