Fall 2006 Handout 12

A doubly-linked list

A structure cannot contain another copy of the same kind of structure (this would cause an infinite regress), but it can contain a pointer to another copy of the same kind of structure.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/doubly.c

```
 1 /*
 2 Let the user type in positive numbers, not necessarily unique.  Store them in
 3 ascending order in a dynamically allocated doubly linked list.  The smallest
 4 number will be at the head of the list, and the biggest number at the tail.
 5 Then print the list from head to tail and from tail to head.
 6
 7 Each number is stored in a structure called a node_t.  Each node is malloc'ed
 8 separately.  head is the first node, tail is always the last.
 9
10 The next field of each node holds the address of the next node, or is NULL if
11 there is no next node.  The prev field of each node holds the address of the
12 previous node, or is NULL if there is no previous node.  The 0 that the user
13 types in to terminate the input is not stored in the list.
14 */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <limits.h>  /* for INT_MAX */
19
20 typedef struct node_t {
21     int n;                 /* the payload */
22     struct node_t *prev;   /* the address of the previous node */
23     struct node_t *next;   /* the address of the next node */
24 } node_t;
25
26 int main()
27 {
28     node_t head = {      0, NULL, NULL};
29     node_t tail = {INT_MAX, NULL, NULL};
30
31     node_t *p, *q;   /* for looping through the list */
32     int n;           /* each number that the user types in */
33
34     head.next = &tail;
35     tail.prev = &head;
36
37     printf(
38         "Please type positive numbers less than %d.\n"
39         "Press RETURN after each number.\n"
```

 All rights reserved ©2006 Mark Meretzky

```
40          "Type a negative number to delete the corresponding positive number.\n"
41          "Type 0 when done.\n", INT_MAX);
42
43      for (;;) {
44          scanf("%d", &n);
45          if (n == 0) {
46              break;
47          }
48
49          if (n > 0) { /* Insert a new node. */
50              node_t *new = malloc(sizeof(node_t));
51              if (new == NULL) {
52                  fprintf(stderr, "Can't allocate %lu bytes.\n", sizeof(node_t));
53                  return EXIT_FAILURE;
54              }
55              new->n = n;
56
57              /*
58              Search the list to find the insertion point.
59              The insertion point will always be between head and tail.
60              */
61              for (p = &head; (q = p->next) != &tail; p = q) {
62                  if (q->n >= n) {
63                      break;
64                  }
65              }
66
67              /* Insert the new node between the ones that p and q point to. */
68              printf("Found the insertion point.\n");
69              new->next = q;
70              new->prev = p;
71              p->next = new;
72              q->prev = new;
73          }
74
75          else {   /* Delete an existing node. */
76              n = -n;
77
78              /* Let p point to the node to be deleted. */
79              for (p = head.next; p != &tail; p = p->next) {
80                  if (p->n == n) {
81                      goto found;
82                  }
83              }
84              fprintf(stderr, "The number %d is not on the list.\n", n);
85              continue;
86
87              found:;
88              p->prev->next = p->next;
89              p->next->prev = p->prev;
90              free(p);
91          }
92      }
93
```

 All rights reserved ©2006 Mark Meretzky

```
 94     printf("Print the list from head to tail:\n");
 95     for (p = head.next; p != &tail; p = p->next) {
 96         printf("%d\n", p->n);
 97     }
 98
 99     printf("\nPrint the list from tail to head:\n");
100     for (p = tail.prev; p != &head; p = p->prev) {
101         printf("%d\n", p->n);
102     }
103
104     return EXIT_SUCCESS;
105 }
```

The above list is never empty: it always contains at least the two nodes **head** and **tail**, and possibly other nodes between them. The following list may be empty: it does not have the two permanently resident nodes. This makes it more complicated to insert and delete.

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/doubly2.c

```
  1 /*
  2 Let the user type in positive numbers, not necessarily unique.  Store them in
  3 ascending order in a dynamically allocated doubly linked list.  The smallest
  4 number will be at the head of the list, and the biggest number at the tail.
  5 Then print the list from head to tail and from tail to head.
  6
  7 Each number is stored in a structure called a node_t.  Each node is malloc'ed
  8 separately.  The variable head holds the address of the first node, or is NULL
  9 if there are no node's yet.  The variable tail holds the address of the last
 10 node, or is NULL if there are no nodes yet.
 11
 12 The next field of each node holds the address of the next node, or is NULL if
 13 there is no next node.  The prev field of each node holds the address of the
 14 previous node, or is NULL if there is no previous node.  The 0 that the user
 15 types in to terminate the input is not stored in the list.
 16 */
 17
 18 #include <stdio.h>
 19 #include <stdlib.h>
 20
 21 typedef struct node_t {
 22     int n;  /* the number */
 23     struct node_t *prev; /* the address of the previous node */
 24     struct node_t *next; /* the address of the next node */
 25 } node_t;
 26
 27 int main()
 28 {
 29     node_t *head = NULL;
 30     node_t *tail = NULL;
 31     node_t *new; /* the new node that the user typed in */
 32     node_t *p, *q;   /* for looping through the list */
 33     int n;  /* each number that the user types in */
 34
 35     printf(
 36     "Please type positive numbers.\n"
```

 All rights reserved ©2006 Mark Meretzky

```
37      "Press RETURN after each number.\n"
38      "Type a negative number to delete the corresponding positive number.\n"
39      "Type 0 when done.\n"
40      );
41
42      for (;;) {
43          scanf("%d", &n);
44          if (n == 0) {
45              break;
46          }
47
48          if (n > 0) { /* Insert a new node (four cases). */
49              new = malloc(sizeof(node_t));
50              if (new == NULL) {
51                  fprintf(stderr, "Can't allocate %lu bytes.\n", sizeof(node_t));
52                  return EXIT_FAILURE;
53              }
54              new->n = n;
55              if (head == NULL) {
56                  /* Case 1: the list was empty. */
57                  new->next = new->prev = NULL;
58                  head = tail = new;
59              }
60
61              else if (n < head->n) {
62                  /* Case 2: insert new at the head of the list. */
63                  new->next = head;
64                  new->prev = NULL;
65                  head->prev = new;
66                  head = new;
67              }
68
69              else if (new->n >= tail->n) {
70                  /* Case 3: insert new at the tail of the list. */
71                  new->prev = tail;
72                  new->next = NULL;
73                  tail->next = new;
74                  tail = new;
75              }
76
77              else {
78                  /* Case 4: insert new into the interior of the list.
79                  Search the list to find the insertion point. */
80                  for (p = head; (q = p->next) != NULL; p = p->next) {
81                      if (q->n >= n) {
82                          break;
83                      }
84                  }
85
86                  /* Insert new between the nodes that p and q point to. */
87                  new->next = q;
88                  new->prev = p;
89                  p->next = new;
90                  q->prev = new;
```

 All rights reserved ©2006 Mark Meretzky

```
 91                 }
 92             }
 93 
 94             else {   /* Delete an existing node. */
 95                 n = -n;
 96 
 97                 /* Let p point to the node to be deleted. */
 98                 for (p = head; p != NULL; p = p->next) {
 99                     if (p->n == n) {
100                         goto found;
101                     }
102                 }
103                 fprintf(stderr, "The number %d is not on the list.\n", n);
104                 continue;
105 
106                 found:;
107                 if (p == head) { /* Delete the first node. */
108                     head = p->next;
109                 } else {
110                     p->prev->next = p->next;
111                 }
112 
113                 if (p == tail) { /* Delete the last node. */
114                     tail = p->prev;
115                 } else {
116                     p->next->prev = p->prev;
117                 }
118 
119                 free(p);
120             }
121         }
122 
123         printf("Print the list from head to tail:\n");
124         for (p = head; p != NULL; p = p->next) {
125             printf("%d\n", p->n);
126         }
127 
128         printf("\nPrint the list from tail to head:\n");
129         for (p = tail; p != NULL; p = p->prev) {
130             printf("%d\n", p->n);
131         }
132 
133         return EXIT_SUCCESS;
134 }
```

A binary tree: K&R pp. 139–143

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/tree.c`

```
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <string.h>
  4 
  5 typedef struct node_t {
```

 All rights reserved ©2006 Mark Meretzky

```
 6     char *string;
 7     struct node_t *left;
 8     struct node_t *right;
 9 } node_t;
10
11 node_t *insert(node_t *root, node_t *new);
12 void print(node_t *root);
13 void dismantle(node_t *root);
14
15 int main()
16 {
17     char line[256];
18     node_t *root = NULL; /* The tree is initially empty. */
19
20     printf(
21         "Press RETURN after each line,\n"
22         "control-d after the RETURN after the last line.\n"
23         "The lines will be output in alpahabetical order.\n"
24     );
25
26     while (gets(line) != NULL) {
27         node_t *new = malloc(sizeof(node_t));
28         if (new == NULL) {
29             fprintf(stderr, "Can't malloc memory for node to hold \"%s\".", line);
30             return EXIT_FAILURE;
31         }
32
33         new->string = malloc(strlen(line) + 1);
34         if (new->string == NULL) {
35             fprintf(stderr, "Can't malloc memory for \"%s\".\n", line);
36             return EXIT_FAILURE;
37         }
38         strcpy(new->string, line);
39
40         root = insert(root, new);
41     }
42
43     print(root);
44     dismantle(root);
45     return EXIT_SUCCESS;
46 }
47
48 node_t *insert(node_t *root, node_t *new)
49 {
50     if (root == NULL) {
51         /* Insert the new node into an empty tree. */
52         root = new;
53     }
54
55     else if (strcmp(new->string, root->string) <= 0) {
56         /* Insert the new node to the lower left of the root. */
57         root->left = insert(root->left, new);
58     }
59
```

 All rights reserved ©2006 Mark Meretzky

```
60     else {
61         /* Insert the new node to the lower right of the root. */
62         root->right = insert(root->right, new);
63     }
64
65     return root;
66 }
67
68 void print(node_t *root)     /* in order */
69 {
70     if (root != NULL) {
71         print(root->left);
72         printf("%s\n", root->string);
73         print(root->right);
74     }
75 }
76
77 void dismantle(node_t *root)     /* post order */
78 {
79     if (root != NULL) {
80         dismantle(root->left);
81         dismantle(root->right);
82
83         free(root->string);
84         free(root);
85     }
86 }
```

Suppose that the above linked list and the above binary tree each contain n items. On the average, you would therefore have to make $\frac{n}{2}$ comparisons to find the correct insertion point in the list, but only $\log_2 n$ comparisons to find the correct insertion point in the binary tree.

Make bibliography

See the three digressions on **make** in the textbook on pp. 241–242, 254–256, and 265–266. Page numbers below refer to the 9-page manual page **make**(1). See also **make**(1p) (Posix) and **make**(1u) (Ultrix). Also print (with minus lowercase L sixty)

```
1$ make -p -f /dev/null | pr -l60 -h 'make internal rules' | lpr
```

See also *Managing Projects with make, 2nd ed.,* by Andrew Oram and Steve Talbott; O'Reilly & Associates, 1991; ISBN 0-937175-90-0; **http://www.oreilly.com/catalog/make2/**

Sample C program to demonstrate make

```
/* This file is func.h. */
int f(void);
```

```
/* This file is var.h. */
extern int i;
```

 All rights reserved ©2006 Mark Meretzky

```
/* This file is main.c. */
#include <stdio.h>
#include <stdlib.h>
#include "var.h"
#include "func.h"

int main(int argc, char **argv)
{
    printf("%d\n", i + f());
    return EXIT_SUCCESS;
}
```

```
/* This file is func.c. */
#include "func.h"

int f(void)
{
    return 2;
}
```

```
/* This file is var.c. */
#include "var.h"

int i = 1;
```

```
#!/bin/sh
#Compile and link the above C program.

gcc -c main.c                               #create main.o
gcc -c func.c                               #create func.o
gcc -c var.c                                #create var.o
gcc -o prog main.o func.o var.o             #create prog
```

Which files need to be recreated?

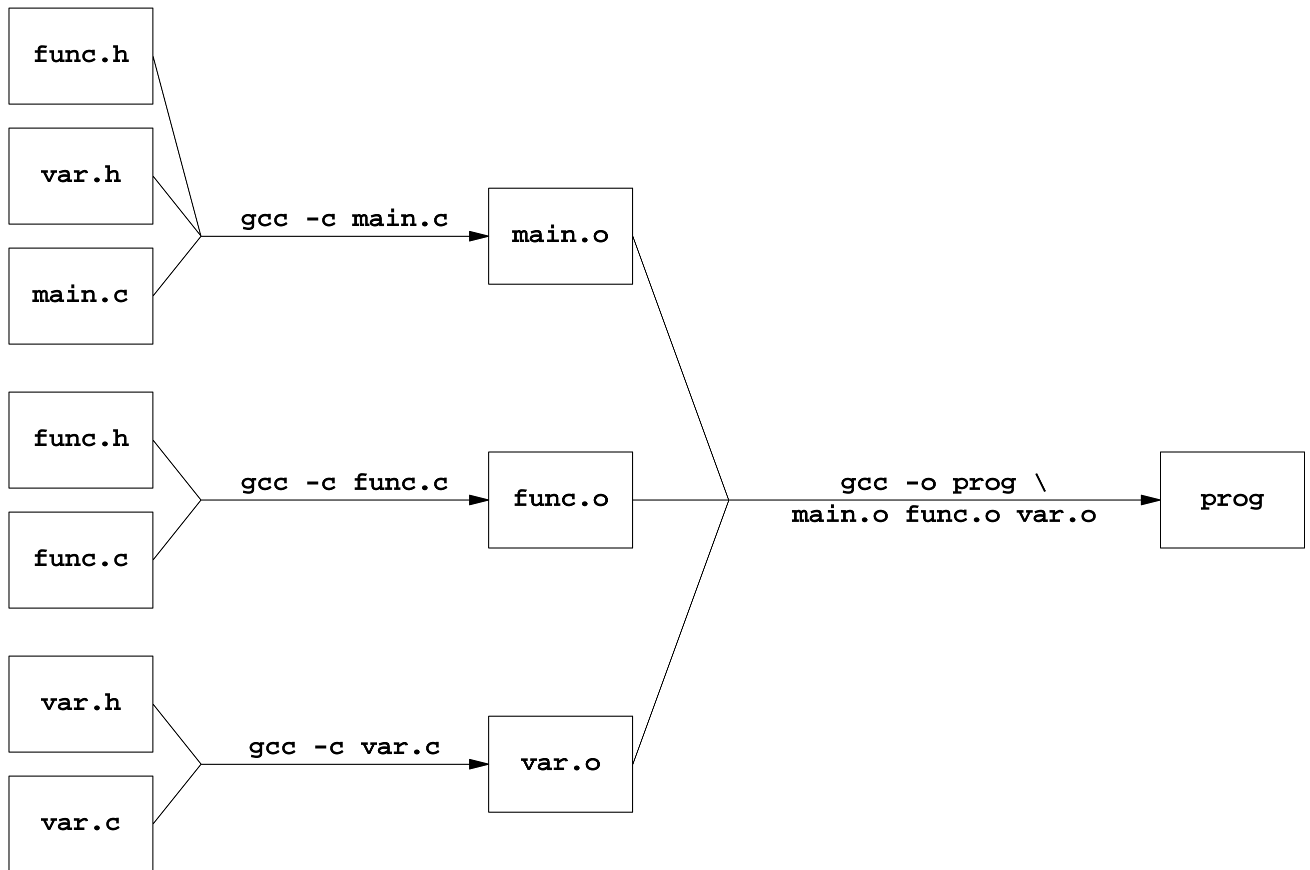
A **.o** file needs to be recompiled if it is older than the corresponding **.c** file or any of the **.h** files that it **#include**'s. An executable file needs to be relinked if it is older than any of the **.o** files it comprises.

```
1$ cd $m46/make
2$ ls -l | tail +2
-rw-r--r--   1 mm64     users      73 Dec  8 13:21 func.c
-rw-r--r--   1 mm64     users      40 Dec  8 13:24 func.h
-rw-r--r--   1 mm64     users     496 Dec  8 13:25 func.o
-rw-r--r--   1 mm64     users     136 Dec  8 13:22 main.c
-rw-r--r--   1 mm64     users    1764 Dec  8 13:25 main.o
-rwxr-xr-x   1 mm64     users   40688 Dec  8 13:26 prog
-rw-r--r--   1 mm64     users      55 Dec  8 13:27 var.c
-rw-r--r--   1 mm64     users      40 Dec  8 13:24 var.h
-rw-r--r--   1 mm64     users     428 Dec  8 13:25 var.o
```

 All rights reserved ©2006 Mark Meretzky

```
3$ ls -lt | tail +2
-rw-r--r--   1 mm64      users    55 Dec  8 13:27 var.c
-rwxr-xr-x   1 mm64      users 40688 Dec  8 13:26 prog
-rw-r--r--   1 mm64      users   428 Dec  8 13:25 var.o
-rw-r--r--   1 mm64      users  1764 Dec  8 13:25 main.o
-rw-r--r--   1 mm64      users   496 Dec  8 13:25 func.o
-rw-r--r--   1 mm64      users    40 Dec  8 13:24 var.h
-rw-r--r--   1 mm64      users    40 Dec  8 13:24 func.h
-rw-r--r--   1 mm64      users   136 Dec  8 13:22 main.c
-rw-r--r--   1 mm64      users    73 Dec  8 13:21 func.c
```

The dependency tree



A simple makefile

Put all of the **.c** and **.h** files of your C program (except for the **.h** files in the **/usr/include** directory) into one directory, together with the file named **makefile** shown below. Since **makefile** is not a shellscript, do not start it with **#!** or turn on its three **x** bits.

The file before the colon is the *target.* The files after the colon are the *dependents.* A line with a colon and the line(s) indented below it constitute a *rule.* Do not indent the line with the target and dependency files, but indent the line(s) below them that tell how to create the target file. Skip an empty line between rules.

 All rights reserved ©2006 Mark Meretzky

In **makefile**, all indentation must be by EXACTLY ONE TAB CHARACTER. Do not indent with blanks. This is the only place in Unix where the difference between blanks and tabs is significant. The *Unix Haters Handbook* (by Simon Garfinkel, Daniel Weise, and Steven Strassmann, with a foreward by Dennis Ritchie; IDG Books, 1994; ISBN 1-56884-203-1), p. 185, says "According to legend, Stu Feldman [the creator of **make**] didn't fix **make**'s syntax, after he realized that the syntax was broken, because he already had 10 users."

The following **makefile** contains four rules. A rule is executed if the target doesn't exist, or if the target is older than any of its dependents, or if there are no dependents listed to the right of the colon.

```
prog: main.o var.o func.o
	cc -o prog main.o var.o func.o

main.o: main.c var.h func.h
	cc -c main.c

var.o: var.c var.h
	cc -c var.c

func.o: func.c func.h
	cc -c func.c
```

You can create any of the targets in the **makefile**. **make** will do all the compiling and linking that is necessary, and no more.

1$ cd *to the directory that contains the* **.h** *and* **.c** *files and the* **makefile**
2$ make prog *create* **prog**
3$ make main.o *create* **main.o**
4$ make var.o *create* **var.o**
5$ make func.o *create* **func.o**

If you give **make** no command line argument, you will create the first target in the **makefile** by default. That's why the target at the root of the tree is listed first in the **makefile**. The other targets can be listed in any order.

6$ make *create* **prog**
7$ make *Nothing happens the second time.*
'prog' is up to date.

▼ Homework 12.1: play with make

The files **func.h**, **var.h**, **main.c**, **func.c**, **var.c**, and **makefile** are in the directory **$m46/make**. Copy them to a directory named **$HOME/prog**. Then

1$ cd $HOME/prog
2$ make
3$ ls -l *Look at the new files created by the* **make** *command.*
4$ prog *Run the program created by the* **make** *command.*

Then run the **make** command again and verify that no additional compilation or linking takes place.

Now edit one of the **.c** files, and verify that **make** compiles only the edited file and then relinks the executable. Then edit one of the **.h** files and verify that **make** compiles every **.c** file that **#include**'s the **.h** file (but no other **.c** file) and then relinks the executable.

Instead of editing a **.c** file, give the **touch** command to let you experiment more rapidly.

```
5$ ls -l main.c
-rw-r--r--   1 abc1234  users        136 Dec  8 13:22 main.c
```

 All rights reserved ©2006 Mark Meretzky

```
6$ touch main.c                        faster than vi main.c
7$ ls -l
-rw-r--r--   1 abc1234  users        136 Dec  8 13:29 main.c
```

You can confuse **make** by saying

```
8$ vi main.c                 This would normally cause make to recompile main.c.
9$ touch main.o
10$ touch prog
11$ make                                         Does make recompile main.c?
```

{ } around a shell variable: p. 148

```
1$ lpq -Pth_hp4si_1
2$ lpq -Ped_hp4si_1
```

```
#!/bin/sh
#Print the queue for each Hewlett Packard 4Si laser printer.
#Without the {}, the echo would print the wrong variable.

for p in th ed
do
    echo ${p}_hp4si_1:
    lpq -P${p}_hp4si_1:
done

exit 0
```

Put parentheses or curly braces around the name of every **make** macro if the name is more than one character long.

A makefile with macros

$@** and **$* are internal macros, i.e., variables to which **make** gives different values automatically in each rule where they are used. See p. 5 in **make**(1). **$@** is the name of the target file, and **$*** is the name of the target file with the suffix removed (i.e., the basename of the target file). Contrary to what the **man** says, **$*** can be used outside of suffix rules. You can use **$*** only in an indented line, not in a colon line.

CC, **CFLAGS**, and **OBJS** are non-internal macros; see p. 255 in the textbook and pp. 3–5 in **make**(1). A non-internal macro can have any name you want, but please pick names that agree with those output by the **make -p** command above.

 All rights reserved ©2006 Mark Meretzky

```
#makefile for the above C program, using macros.

CC = gcc
CFLAGS = -O      #optimization
OBJS = main.o func.o var.o

prog: $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS)

main.o: main.c func.h var.h
	$(CC) $(CFLAGS) -c $*.c

func.o: func.c func.h
	$(CC) $(CFLAGS) -c $*.c

var.o: var.c var.h
	$(CC) $(CFLAGS) -c $*.c
```

▼ Homework 12.2: create a makefile for moon

```
1$ cd
2$ mkdir moon
3$ cd moon
4$ cp $m46/moon/moon*.[ch] .                  Handout 1, p. 14
5$ chmod 644 moon*.[ch]                       if you plan to edit these files
```

Create a **makefile** for **moon** in your **$HOME/moon** directory. Hand in the **makefile**. Use the internal macros **$@** and **$***. Create the macros **CC**, **CFLAGS**, and **OBJS**. Leave the **CFLAGS** macro empty if you don't want optimization:

```
CFLAGS =
```

Also create a macro named **LOADLIBES** to hold the **-lm** option of **gcc** that **moon** requires. Write **$(LOADLIBES)** at the end of the line that contains **-o $@**.
▲

Other ways to create a make macro: p. 3.

Instead of defining a macro in your **makefile,** you can pass a command line argument to **make**:

```
1$ make CC=gcc                                No space around the equal sign.
```

or set an environment variable:

```
2$ setenv CC gcc          do this in your .login file.
3$ env | more             see the names and values of all your environment variables
4$ make
```

If you a macro without defining it anywhere, **make** will use the default definition you printed out with the **make -p** command. These rules also include the defaults for any rules you left out of your **makefile**. For example, if you don't say how to create a **.o** file from a **.c** file, **make** will use the **.c.o** default rule displayed by the **make -p** command.

A makefile for an archive

An archive is made out of **.o** files, just as a **.o** file is made out of a **.c** file. The **.o** files inside of the archive **libppm.a** are named **libppm.a(ppm_inheader.o)**, **libppm.a(ppm_outheader.o)**, etc. Use backslashes to divide a long colon statement into separate lines.

 All rights reserved ©2006 Mark Meretzky

The internal macro **`$?`** in the following **`makefile`** holds the names of all the **`.o`** files in the library that need to be recompiled (i.e., that are older than the corresponding **`.c`** files); see p. 3. The macro **`$(?:.o=.c)`** is **`$?`** with the **`.o`**'s at the end of each word changed to **`.c`**'s; see p. 4.

Each indented line counts as a separate shellscript. To write a multi-line command such as **`if-then-else-fi`**, you must therefore use backslashes.

```
#!/bin/sh
#What goes wrong if you omit the semicolon?

grep word file
who

grep word file; who
```

```
#!/bin/sh
#What goes wrong if you omit the semicolons?

if grep -q word file
then
	who
fi

if grep -q word file; then who; fi
```

```
#This file is $m46/ppm/src/makefile.

CFLAGS = -I/home/m/mm64/46/ppm/include

libppm.a: \
	libppm.a(ppm_inheader.o) \
	libppm.a(ppm_outheader.o) \
	libppm.a(ppm_negative.o)
	$(CC) $(CFLAGS) -c $(?:.o=.c)
	if [ -f $@ ];\
	then\
		ar rsv $@ $?;\
	else\
		ar crsv $@ $?;\
	fi
	rm $?

#Disable the default rule for creating a .a file out of .c files to
#allow the above rule to be used instead.
.c.a:;
```

A taller tree

The above tree diagram had only three levels: the root, the leaves, and one level in between. For tasks with more steps, the tree may be much taller.

A file written by human beings is called *source code.* Not all **`.c`** and **`.h`** files are source code: some are written by programs. For example, human beings write **`.y`** files and **`.l`** files and feed them to **`yacc`** and **`lex`**:

 All rights reserved ©2006 Mark Meretzky

```
1$ yacc hoc.y          create y.tab.c: pp. 233–287
2$ lex lexer.l         create lex.yy.c: pp. 256–258
```

The resulting files **y.tab.c** and **lex.yy.c** must then be compiled into **.o** files. Here is a **makefile** for a tree with four levels:

```
OBJS = main.o y.tab.o lex.yy.o

prog: $(OBJS)
	$(CC) -o $@ $(OBJS) $(LOADLIBES)

main.o: main.c
	$(CC) $(CFLAGS) -c $*.c

y.tab.o: y.tab.c
	$(CC) $(CFLAGS) -c $*.c

lex.yy.o: lex.yy.c
	$(CC) $(CFLAGS) -c $*.c

y.tab.c: hoc.y
	$(YACC) $(YFLAGS) hoc.y

lex.yy.c: lexer.l
	$(LEX) $(LFLAGS) lexer.l
```

If the source files **hoc.y** and **lexer.l** were put under the protection of RCS, then **make** would need an additional preliminary step to get these files from RCS. The tree would then have five levels:

```
hoc.y:
	co hoc.y

lex.l:
	co lex.l
```

A forest

A *forest* is two or more trees. A **makefile** may contain a forest instead of a single tree. The root of one tree will usually be the executable file we want to create. The roots of the other trees may be also be files that we want to create, but more often are merely names for groups of commands that we want to execute.

For example, there is no file named **cleanup**, nor will there ever be. Type **make cleanup** to lead **make** to believe that we want to create a file named **cleanup**. **make** will then execute the indented **rm** command, believing that this will create **cleanup**.

 All rights reserved ©2006 Mark Meretzky

```
OBJS = main.o file1.o
SOURCES = prog.h main.c file1.c

prog: $(OBJS)                          #The root of the first tree
	$(CC) $(CFLAGS) -o $@ $(OBJS)

main.o: main.c
	$(CC) $(CFLAGS) -c $*.c

file1.o: file1.c
	$(CC) $(CFLAGS) -c $*.c

#Remove all files that are not source code.
cleanup:
	rm prog $OBJS

print:
	pr -l60 $(SOURCES) | lpr     #minus lowercase L sixty

test:
	prog < test.data > test.out
	if cmp -s correct.out test.out;\
	then\
		rm test.out;\
		strip prog;\
		mv prog /usr/local/bin;\
	else\
		echo 'Failed the test.';\
	fi
```

```
1$ make
2$ make main.o
3$ make test
4$ make cleanup
5$ make print
```

Print only the files of which you have no up-to-date printout: p. 265

Add the following rule to the end of the first **makefile** in this handout.

```
print: func.h var.h main.c func.c var.c
	pr -l60 $? | lpr
	touch print
```

The first time you say **make print**, the indented commands will be executed because the file **print** does not exist. The macro **$?** will hold the names of all the dependents of this rule. The **touch** command will then create **print**.

Every subsequent time you say **make print**, the macro **$?** will hold the names of only those dependents that are newer than the file **print**. The indented command will print only the files that have been edited since the last time you said **make print**.

Use the same technique to back up only the files that have been modified since the last time they were backed up.

□

 All rights reserved ©2006 Mark Meretzky