Question 1. (18 points) When TAs download homework assignments from Catalyst, the files from each user are stored in a separate directory whose name is the user's login name. For this problem, assume that the current directory contains subdirectories and files like the following:

```
xyzzy/hw4.c
xyzzy/test.exe
xyzzy/core
xyzzy/notes.txt
rabbit/stuff.c
rabbit/stuff.exe
rabbit/notes.txt
frodo/homework4.c
frodo/foo.exe
frodo/notes.txt
gumby/project4.c
gumby/core
gumby/notes.txt
```

Give a bash shell command or set of commands to do the following tasks. You must use wildcards (*), loops, or other shell expansions or control constructs to process multiple directories. You may not write out individual commands, one for each directory, using the example names shown above. You do not need to write complete shell scripts – just give the necessary commands.

(a) Delete all of the `core` files and `.exe` files in all of the directories.

```
rm */core */*.exe
```

(b) Each of the `.c` files seems to have a different name. Change the name of each `.c` file to `hw4.c`. You should assume there is exactly one `.c` file in each directory.

```
for d in *
do
   mv $d/*.c $d/hw4.c
done
```

(An industrial-strength answer would need to deal with issues like spaces in directory names, but we did not worry about that when grading the question.)

(c) The command `mail` *user* < *file* will email the contents of *file* to *user*. Give shell command(s) to email the `notes.txt` files to the users whose names are given by the directory names (so, for example, file `rabbit/notes.txt` should be mailed to user `rabbit`.)

```
for d in *
do
   mail $d < $d/notes.txt
done
```

Question 2. (18 points) Write regular expressions that could be used with `grep` or `egrep` to locate words in a file containing a list of words, one word per line (like the one demonstrated in class). Circle `grep` or `egrep` for each problem to indicate whether you are using basic or extended regular expressions.

(a) All words that contain the three letters x, y, and z in that order, possibly with other letters before, after, and/or in between the x, y, and z (possibly including more occurrences of x, y, or z). ("oxygenize" is such a word, "dysoxidize" is not.)

Circle: (`grep`) `egrep`

```
x.*y.*z
```

(b) All words that contain at least two digit characters (0-9) where this is at least one non-digit character between two of the digits. (For example: "30-30" and "2,4-d" are words that should be selected, "11-point", "10th", and "RS232" are not.)

Circle: (`grep`) `egrep`

```
[0-9][^0-9].*[0-9]
```

(c) All words that start and end with the *same* letter that is *not* a vowel. Vowels are the letters a, e, i, o, and u and their upper case versions (A, E, I, O, U). (Examples include "bathtub", "carcinogenic", and "yummy".)

Circle: (`grep`) `egrep`

```
^\([^aeiouAEIOU]\).*\1$
```

It's possible to solve all of these problems using extended regular expressions of course, but the answers here only use basic ones.

Question 3. (6 points) Suppose we have a file `silly.txt` that contains the following lines:

```
eat school pizza
ride farm cow
```

What output is produced when we execute the following command?

```
sed -e 's/\(.*\) \(.*\) \(.*\)/I \1 a \3 @ \2/' silly.txt
```

```
I eat a pizza @ school
I ride a cow @ farm
```

Question 4. (6 points) One problem that users sometimes have with Linux is that there is no undo command to, for example, recover a file that is accidentally deleted. As an attempt to prevent accidental deletions, it is possible to use the `–i` option on `rm`, which causes `rm` to ask the user for permission before deleting each file.

Give a shell command that creates an alias `remove` so that when the user enters a command like `remove a b c d`, the command `rm –i a b c d` is executed instead.

```
alias remove='rm –i'
```

Question 5. (16 points) People write telephone numbers in different ways, for example (206) 555-1212, 206-555-1212, and 206.555.1212. For this problem, create a shell script to read a file that includes such phone numbers and rewrite them in the format aaa.bbb.cccc, like the third example given above.

The input file is formatted as follows: each line consists of a name, an address, a phone number, and a city. The four fields are separated by colons (`:`), and there are no other colons in the file (i.e., every occurrence of '`:`' is used to separate fields). Telephone numbers only appear in the above three formats. An example input file is:

```
Bill Gates:1 Infinite Loop:(425) 524-1234:Redmond
Bonnie Dunbar:Museum of Flight:206.764.5720:Seattle
Ichiro:Seattle Mariners:206-346-4000:Seattle
```

The shell script should have one argument giving the input file name and should use `sed` to write a copy of the input file on `stdout` with all of the phone numbers rewritten in the aaa.bbb.cccc format. The script should print "missing file name" and exit with a non-zero exit status if there is not exactly one argument. Otherwise you can assume that there are no errors – the data file exists, is properly formatted, the phone numbers are in one of the three formats given above, etc. You may also assume that the name, address, and city fields do not contain phone numbers.

Hint: You might find more than one editing operation and/or a pipeline to be useful. Then again, you might not.

There are, of course, many ways to answer the problem. We gave credit to any answer that properly edited the phone numbers as specified. An industrial-strength version of this would need to check that the phone numbers were, in fact, in the 3rd field of each line but we let that go.

Here's one brute-force answer that would work. The \ at the end of each line indicates a long line that doesn't fit on the page, not a line break:

```
#!/bin/bash
if [$# -ne 1]
then
   echo missing file name
   exit 1
fi
sed –e 's/(\([0-9][0-9][0-9]\)) \([0-9][0-9][0-9]\)-\
          \([0-9][0-9][0-9][0-9]\)/\1.\2.\3/'       \
      -e 's/\([0-9][0-9][0-9]\)-\([0-9][0-9][0-9]\)-\
          \([0-9][0-9][0-9][0-9]\)/\1.\2.\3/'   $1
```

Question 6. (16 points) Consider the following C program.

```
#include <stdio.h>

void foo(int *a, int *b) {
  *a = *b;
  *b = 17;
  *a = *a + 1;
}

int main() {
  int i, j;
  int *x;
  int *y;
  x = &i;
  y = &j;
  *x = 7;
  *y = 0;
  printf("x = %d, y = %d\n", *x, *y);
  foo(x,y);
  printf("x = %d, y = %d\n", *x, *y);
  return 0;
}
```

(a) What output does this program produce when it is executed? (It does execute successfully.) Feel free to draw diagrams showing memory to help answer the question and to help us award partial credit if needed.

```
x = 7, y = 0

x = 1, y = 17
```

(b) Now suppose we change function `foo` by rewriting its first statement as follows:

```
void foo(int *a, int *b) {
  a = b;
  *b = 17;
  *a = *a + 1;
}
```

Now what output does the program produce when it is executed?

```
x = 7, y = 0

x = 7, y = 18
```

Question 7. (20 points) (The small C programming exercise.) A useful Unix utility is the program `head` that prints the first few lines of its input file. For this problem, complete the program below so it will read the file whose name is given on the command line and print the first 5 lines of the file to `stdout`. The program can be written as a single main function, although you may define additional functions if you wish. You may assume that no input line has more than `MAX_LINE_SIZE` characters, including the trailing '\0'. You do not need to check for errors – assume that the input file exists and can be read successfully. Be sure your code works properly if the input file contains fewer than 5 lines. Hints: `FILE * fopen(char* filename, char* mode);` `char * fgets(char *buffer, int max_len, FILE *stream);`

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE_SIZE 1000  /* max # characters per line */

/* print the first 5 lines of the file named on the command */

int main(int argc, char** argv) {
   int nlines_printed;           /* # lines printed so far */
   char line[MAX_LINE_SIZE];     /* current input line */
   FILE * f;                     /* input file */
   f = fopen(argv[1], "r");
   nlines_printed = 0;
   while (nlines_printed < 5 && fgets(line, MAX_LINE_SIZE, f)) {
      fputs(line, stdout);   /* printf("%s", line) also works */
      nlines_printed++;
   }
   fclose(f);    /* good practice, but no deduction if not done */

   return 0;
}
```