CSE 490c – Autumn 2003
Midterm 1

Please do not read beyond this cover page until told to start.

1. *[2 points]*
   Suppose the current working directory has three files, named "a", "a*a", and "aa" (without the quotes). Give a shell command that will delete **only** file "a*a".

   *rm "a*a"*
   or *rm 'a*a'*
   or *rm a*a*

2. *[2 points]*
   Give a common use of the ~/.profile file.

   To initialize environment variables (e.g., $PATH).

3. *[2 points]*
   What is the lifetime of an environment variable?

   From the time it is first used to the end of the process, or until it is explicitly undefined, whichever comes first. (Sub-processes do NOT use the same environment variable – they have separate environments, initialized to be copies of the environments of their parents.)

4. *[2 points]*
   Suppose you download the source to some open source application from the web. You look at the contents of a file called `hash.h`, and the first line is
   ```
   #ifndef HASH_H
   ```
   With 99% likelihood, what is the next line of that file?

   #define HASH_H

5. *[3 points]*
   Suppose you download the source to an application written in C and find that it has 104 source (.c) files, all in the same directory. Reading the code you find that there is a global variable, `root`. You want to find all uses of that variable, so you issue the following shell command:
   ```
   $> grep root *.c
   ```
   Give a significant reason why the output of this command may show many lines that are not what you were looking for (the uses of the global variable `root`). (If you're worried about strings like `myRoot`, note that `grep -w root *.c` fixes that problem, and so it's not a "significant" problem.)

   grep knows nothing about scope. There may be local variables named root, and grep will print lines using them even though they are not references to the global variable.

6. [6 points]
While the three shell commands below all result in exactly the same output on the screen, I claim that they result in different execution sequences on the CPU.

```
$> cat testfile | grep shoe          # 1
$> grep shoe testfile                # 2
$> grep shoe <testfile               # 3
```

For **each of the three**, explain what happens in its particular instance that does not happen in the other two.

(a) What is the **shell** doing in #1 that it is not doing in #2 or #3?
The shell is forking two processes: cat and grep.

(b) What is the C program implementation of **grep** doing in #2 that it doesn't do in #1 or #3?
The string "testfile" is passed to grep as in argv. grep must open that file for reading. (In the other two, it just reads standard in.)

(c) What is the **shell** doing in #3 that it doesn't do in #1 or #2?
The shell opens file testfile and then forks grep in a way that grep's standard input is attached to the file named "testfile." (Note: the shell does NOT pass the CONTENTS of testfile to grep.)

5. [6 points]
Consider the utilities find, awk, and sed. For each, give an example of a situation where using it would be convenient, but the other two would not.

(a) find
To locate one or more copies of a file with some particular name, with the search starting in some directory and continuing recursively over all subdirectories.

(b) awk

To print just the third column of a file containing a multi-column table.

(c) sed

To change all occurrences in some file of "__/__/19__" to "__/__/20__".

6. *[6 points]*
Suppose the current working directory contains (only) files named:

```
     main.c      a.c      a.h      Makefile
```

The contents of `Makefile` are:

```
myApp.exe:  main.o a.o
       gcc main.o a.o -o myApp.exe

main.o : main.c a.h globalDefs.h

a.o: a.c a.h globalDefs.h

%.o : %.c
       gcc -c $<

%.h :
       cp ../$@ .
```

Give, in a plausible order, the (shell) commands executed if the user simply types `make` to the shell. (Reminder: `$<` means the name of the dependency file in the rule, and `$@` means the name of the target file matching the rule.)

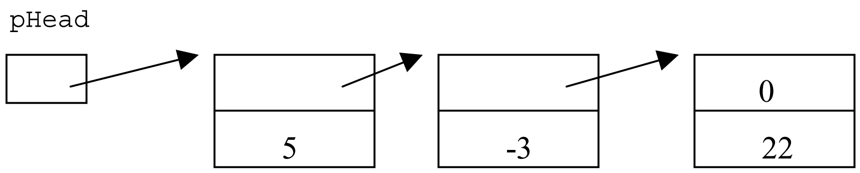
cp ../globalDefs.h .
gcc –c main.c
gcc –c a.c
gcc main.o a.o –o myApp.exe

7. *[10 points]*
Suppose the following occurs in a C program:

```
struct NodeType {
       struct NodeType*    pNext;
             int           value;
};
typedef  struct NodeType  Node;

Node* pHead;
```

Suppose at some point in execution the current state of the program can be drawn like this:



That is, `pHead` points to a `Node`. That `Node`'s `pNext` field points to another `Node`, and its `value` field holds some integer. This "linked list" continues until there is a `Node` whose `pNext` field is null (equal to 0).

a. Write some C code that will set `Node*` variable `pLast` to the address of the last `Node` in the list. **Your code should work for all lists consisting of at least one `Node`**, not just the one shown here. You can assume that (a) `pHead` is **not** null, and (b) `pLast` and `pHead` are both in the scope of the code you're writing.

```
pLast = pHead;
while ( pLast->pNext != NULL ) {
   pLast = pLast->pNext;
}
```

or

```
for (pLast=pHead; pLast->pNext; pLast = pLast->pNext) ;
```

b. Now assuming that `pLast` points to the last `Node` in the list, write additional code that adds a new `Node` to the end of the list. (You must create the `Node` as part of your code.) The new `Node` should have a value field equal to 100.

pLast->pNext = (Node)malloc(sizeof(Node));*
pLast = pLast->pNext;
pLast->pNext = NULL;
pLast->value = 100;

8. *[4 points]*
Consider the following shell script:

```
#! /bin/bash

for f in *.c; do
    if [ ! -e `echo $f | sed 's/.c$/.h/'` ]; then
       echo $f
    fi
done
```

Describe in English what it does.

For each file named XXXX.c in the current directory, print the file name if there is no file named XXXX.h in that directory.