Practice Final Exam II

1) LISP coding

a) Write the function (`all-but-last list`) which takes a list and returns a new list of the same elements after removing the last one. If given an empty or singleton list, it returns nil.

```
? (all-but-last '(1 2 3 4))
(1 2 3)
? (all-but-last  '((3.5 "hi") (4 1)))
((3.5 "hi"))
? (all-but-last '(a))
NIL
```

Write this function recursively.

```
(defun all-but-last(list)
```

b) Write the function (`prefix pred list`) which takes a list and a one-argument predicate and returns the longest prefix of the list that passes the predicate. The predicate function is expected to take one argument, a list, and returns true if the list is "acceptable", nil otherwise. You should evaluate the predicate at most O(n) times, each time for a successively shorter prefix. If no prefix satisfies the predicate, an empty list is returned.

```
? (prefix #'sortedp '(4 8 1 2 3 4))
(4 8)
? (prefix #'listp '(a b c d))
(a b c d)
? (prefix #'numberp '(10 8 "hi"))
NIL                                    ;; nil because no list is a number!
```

Write this function recursively, using the `all-but-last` function from part (a).

```
(defun prefix(pred list)
```

c) Write the function (`longest pred list`) which takes a list and a predicate function and returns the longest subsequence which passes the predicate. A subsequence can start and end anywhere within the list, but cannot "skip over" elements. If there is a tie for the longest subsequence, your function can return any one of them. If there is no subsequence at all that passes the predicate, return the empty list.

```
? (longest #'sortedp '(5 8 3 4 2 7 12 1))
(2 7 12)
? (longest #'atom '(a b c))
NIL
```

You will want to use the `prefix` function from part (b), as well as the `maplist` built-in and the `most` function from the Lisp assignment:

```
; MAPLIST (fn list)
: Like mapcar, but maps over sublists instead of elements. Invokes fn
; once for each sublist of decreasing length. For example, maplist over
; (a b c) calls fn once passing (a b c), once passing (b c) and once
; passing (c). Like mapcar, results are gathered and returned in a list.

? (maplist #'length '(30 apple 12 "Hi!"))
(4 3 2 1)

; MOST (list comparator)
; Takes a non-empty list and two argument comparator which returns T
; if its first argument was "more than" its second argument.  Returns
; the element in the list which was "most" according to the comparator.

? (most '(3 46 7 -72 6 -8) #'>)
46
```

Do this with mapping, not recursion.

```
(defun longest(pred list)
```

d) Write the function (`find-palindrome list`) that will return the longest palindrome subsequence in list. A palindrome is a sequence that reads the same forward and backwards.

```
? (find-palindrome '(6 10 6 4 5 4 6))
(6 4 5 4 6)
? (find-palindrome '(1 2 3))
(1)                                    ;; could return (2) or (3)
equivalently
```

You will want to use the function from part (c) and the `reverse` built-in.

```
; REVERSE (list)
; Takes list and returns a new list with same elements in reverse order.

? (reverse '(a b c))
(c b a)
```

```
(defun find-palindrome(list)
```

2) Concurrency coding

You are tasked with writing the pizza life cycle simulation. The actors in this simulation are the many hungry students, the manager of the pizza store, and the many pizza delivery drivers. At the beginning of the simulation, one thread is launched for each student and one for the manager.

Each student spends time "studying" which builds up their hunger. Each student wants a number of slices (randomly chosen when student was created) but not an entire pizza (12 slices), so they hook up with other students to form an order. When there is enough interest for a whole pizza, one of the students calls and orders a pizza from the manager. As a special case, the last students to finish studying might not have enough people for a whole pizza, but they order one anyway. It's also possible that a student might spread his slices over more than one pizza order if necessary.

The manager does nothing until a student requests a pizza. It "makes a pizza" and dispatches a new driver to deliver it. A separate thread is spawned for each driver so the manager can immediately go back to manning the phones and making pizzas. A driver thread "drives" to the dorm, hands off the pizza to the hungry students, and drives back to the pizza store.

When the driver arrives, all hungry students gather around and try to get a slice of the pizza. It doesn't matter to a student if this is the pizza they ordered, all students try to grab slices from any pizza at it arrives.

A student finishes after they have eaten their fill of pizza. A driver finishes after driving back to the store. The pizza manager has to work until all pizzas have been made and all delivery drivers have returned to the store before finishing. The number of pizzas needed is calculated ahead of time and passed to the manager when started, so it knows how many total will need to be made.

Here is the starting main function for the pizza simulation:

```
#define PIZZA_SIZE 12         // there are a dozen slices per pizza
#define NUM_STUDENTS 40

void main(void)
{
   int i, numPizzas, numSlices, total = 0;

   InitThreadPackage(false);

   for (i = 0; i < NUM_STUDENTS; i++)     {     // create all student threads
      numSlices = RandomInteger(1, 4);    // chose num slices for student
      ThreadNew(Student, 1, numSlices, 0, 0, "Student");
      total += numSlices;
   }

   // calculate num entire pizzas needed, add one extra if leftover
   numPizzas = total/PIZZA_SIZE + (total % PIZZA_SIZE == 0 ? 0 : 1);
   ThreadNew(Manager, 1, numPizzas, 0, 0, "Manager");
   RunAllThreads();
}

      // these simulation functions don't do anything, just "fake"
```

```
static void MakePizza(void);  // for Manager, makes one pizza at a time
static void Study(void);            // for Student, to work up some hunger
static void Drive(void);            // for Driver, drive to dorm or back to
store
```

Assume the above three helpers are already written and are thread-safe, you can just call them when you need to. Your job will be to write the `Student`, `Manager`, and `Driver` functions to properly synchronize the activities of the actors. As always, there should be no busy waiting.

Declare global variables and semaphores here. Clearly indicate what are the initial values for each.

Write the `Student`, `Manager`, and `Driver` functions.

3) Java

On the next page, you'll find code for the implementation of a generic Programmer class. Our basic programmer responds to messages to edit a module, test it, and track the frustration level while doing so. A high frustration level is bad, lower (and even negative) is good. You'll note that when a programmer's frustration level is too high, they are prone to swearing while they work. The Programmer also includes a method to sleep to work off some of that stress.

You are going to introduce three new classes based on Programmer: the JavaProgrammer, the LispProgrammer, and the CProgrammer. All three should understand all the same messages as Programmer and have the same general behavior in terms of editing, testing, and sleeping.

However, some actions are different for the various programmers. For example, consider editing: LISP programmers find editing their unreadable programs so unpleasant that each time a LISP programmer edits a module, their frustration level rises an extra point. Whenever a Java or C programmer edits a module, they have to take the extra step of compiling it (print a message "compiling module") before they are done editing.

Although the basic Programmer always has a 50% probability of finding no bugs when testing a module, it differs for the other programmers. For a JavaProgrammer, the probability is much higher, 75% of the time the module has no errors. In LISP, it varies. At any given moment, if you ask a LISP programmer what the probability of having no bugs in a module, the response will be somewhere between 0 and 100%. In C, the probability is proportional to the programmer's experience. The probability of having no bugs starts at zero (when the C programmer has no experience) and each time they test a module, that extra experience increases their probability of having no bugs by 5%. However, no matter how experienced, a C programmer never has higher than 90% probability of no bugs.

The programmers also differ in their testing strategies. Unlike LISP where the interpreter makes it easy to test, in Java and C, the programmer has to first edit a module called "*ModuleName*Test" before they can test. Because C is so prone to memory problems, if a C Programmer tests their module and no bugs are found, before concluding it's perfect, they then run an additional test using Purify (print "purifying module"). Any C program has a 50% chance of having no Purify problems. For JavaProgrammers, it's not enough to test the program once, it has to be tested and pass on all platforms. JavaProgrammers keep a list of the names of all the current platforms. For each test, the JavaProgrammer edits the *ModuleName*Test and prints the name of the platform being tested. Your design should make it possible to easily introduce a new platform and have all Java programmers start including it in test runs right away.

In addition to the editing and testing behaviors, you also need to make the 3 programmers respond to the new method `writeModule(String name)`. This method takes the String name of the module to work on and directs the programmer to edit and test this module until it works. The method returns the number of iterations (i.e. the number of times the programmer had to edit the module) before it worked. During the process, if a programmer's frustration level is above 10 after an unsuccessful attempt, they take a break and sleep to regain some sanity before continuing on.

Your job is to design and implement the three programmer classes as described above. You are free to add any other helper classes and can change or add to the generic Programmer class as well. Note that we are not going to worry about allocation or initialization. You do not have to write any constructors. Where needed you can indicate the starting value for variables.

Your most important design goal is to avoid code duplication. It is recommended you think through the entire design before making any decisions.

```
public class Programmer {

   public void edit(String moduleName)
   {
      System.out.println("Editing " + moduleName);
   }

   public boolean test(String moduleName)
   {
      System.out.println("Testing " + moduleName);
      boolean hasNoBugs = randomChance(.5); // 50% of finding no bugs
      if (hasNoBugs)
         frustrationLevel--;
      else {
         frustrationLevel++;
         if (frustrationLevel > 5) System.out.println("$%@#!");
      }
      return hasNoBugs; // returns whether passed test
   }

   public void sleep()
   {
      System.out.println("Sleeping...");
      frustrationLevel -= 10;
   }

   // This is a utility helper to generate a random boolean with a
   // given probability. If prob is .75, returns true 75% of time, false 25%
   protected boolean randomChance(double probability)
   {
      return (probability >= Math.random());
   }

   protected int frustrationLevel = 0;          // all programmers begin calm
}
```

Make it clear what modifications you want to make to the above Programmer class.You can change the code inside the methods and can add methods, but you cannot change the prototypes (parameters and returns values) of the given methods.

4) C coding (

Write the generic `RemoveAll()` function in C which removes all occurrences of an element from a collection. It takes an array of elements of any type and size and modifies the array in place, shuffling elements over as ones are deleted. It compares elements using a passed-in comparator.

```
// Comparator function takes pointers to two elements, and returns
// (neg,0, pos) to indicate their ordering. negative means (a < b)
typedef int (*CompFn)(const void* a, const void* b);

// elems    = base address of array of elements
// numElems = total number of elements in the array
// elemSize = size of each element in bytes
// cmp      = comparator function for elements
// elem     = pointer to element to remove
void RemoveAll(void *elems, int numElems, int elemSize,
               CompFn cmp, const void *elem)
```

5) Short answer

Each of these questions can be answered fairly briefly— please don't feel compelled to fill every nanoacre with writing. Also note that these questions are purposely not assigned many points, so don't burn a lot time on them.

a) For each of the following three cases, indicate Yes / No if the ++ operation is allowed at compile-time.

```
void bar(int c[]) {
   c++;                 // 1) allowed?
}

void foo() {
   int a[10];
   int *b;

   b = a;
   a++;                 // 2) allowed?
   b++;                 // 3) allowed?
   bar(a);
}
```

b) Give an example of type of error you can make that is caught at compile-time by C yet not caught until runtime in LISP. Identify the language feature(s) responsible for the discrepancy.

c) Give a code fragment that compiles cleanly in both Java and C but will have predictable behavior in Java and unpredictable results in C. Predictable in this case means that it is guaranteed that exactly the same result occurs every time you execute the code. Identify the language feature(s) responsible for the discrepancy.

d) Give one advantage and one disadvantage of Java `synchronized` methods as opposed to direct use of semaphores.

e) You are designing a base Shape class intended to be the superclass for Rectangle, Oval, and Triangle, and are trying to decide how to handle computing the area. Each

shape subclass will have its own definition of the `area` method. For Shape itself you have three options: create an `area` method that returns 0, declare `area` as an abstract method, or leave `area` out of the Shape class and have each subclass declare `area` themselves. Which of the three is the better design decision? Explain what makes that choice better than each of the other two options.