Practice Final Solution II

1) LISP coding

a)

```
(defun all-but-last (list)
  (if (or (null list) (null (cdr list))) nil
      (cons (car list) (all-but-last (cdr list)))))
```

b)

```
(defun prefix(pred list)
  (cond ((null list) nil)
        ((funcall pred list) list)
        (T (prefix pred (all-but-last list)))))
```

c)

```
(defun longest(pred list)
  (most (maplist #'(lambda(sublist) (prefix pred sublist)) list)
        #'(lambda(x y) (> (length x) (length y)))))
```

d)

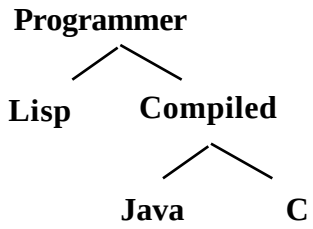
```
(defun find-palindrome(list)
  (longest #'(lambda(sublist) (equal sublist (reverse sublist))) list))
```

2) Concurrency coding

```
int numToOrder = 0, numStudentsLeft = NUM_STUDENTS;
Semaphore numLock;                                  // binary, init to 1
Semaphore orderPizza,slicesAvail, driverReturned;     // general, init to 0


void Manager(int numPizzas)
{
   int i;

   for (i = 0; i < numPizzas; i++) {
      SemaphoreWait(orderPizza);          // wait for rendezvous from Student
      MakePizza();
      ThreadNew(Driver, 0, 0, 0, 0, "Driver");
   }

   for (i = 0; i < numPizzas; i++)  // numPizzas same as numDrivers...
      SemaphoreWait(driverReturned);      // use semaphore to count
}


void Driver()
{
   int i;

   Drive();
   for (i = 0; i < PIZZA_SIZE; i++)
      SemaphoreSignal(slicesAvail); // signal once per slice
   Drive();
   SemaphoreSignal(driverReturned);
}



void Student(int numSlices)
{
   int i;

   Study();

   SemaphoreWait(numLock);          // globals accessed under binary lock
   numToOrder += numSlices;   // add into total
   numStudentsLeft--;
         // loop since last person may need to order two pizzas in rare case
   while (numToOrder >= PIZZA_SIZE || (numStudentsLeft == 0 && numToOrder > 0))
   {
      SemaphoreSignal(orderPizza);
      numToOrder -= PIZZA_SIZE;
   }
   SemaphoreSignal(numLock);

   for (i = 0; i < numSlices; i++)
      SemaphoreWait(slicesAvail);   // wait for each slice
}
```

3) Java coding

Obviously all the three programmers should be subclassed from Programmer, but the commonalties of Java and C programmers leads to an intermediate Compiled Programmer class as well:



Changes to base Programmer class:

Change line 2 of test method to read:

```
boolean hasNoBugs = randomChance(noBugProbability());
```

Add these methods:

```
protected double noBugProbability()
{
   return 0.5;
}

public int writeModule(String moduleName)
{
   int numTries = 0;

   while (true) {       // keep going until it works
      numTries++;
      edit(moduleName);
      if (test(moduleName)) break;  // we passed!
      if (frustrationLevel > 10)
         sleep();
   }
   return numTries;
}
```

LispProgrammer class:

```
public class LispProgrammer extends Programmer {

   public double noBugProbability()
   {
      return Math.random();
   }

   public void edit(String moduleName)
   {
      super.edit(moduleName);
      frustrationLevel++;
   }
}
```

CompiledProgrammer class:

```
public class CompiledProgrammer extends Programmer {

   protected void compile(String moduleName)
   {
      System.out.println("Compiling " + moduleName);
   }

   public void edit(String moduleName)
   {
      super.edit(moduleName);
      compile(moduleName);
   }

   public boolean test(String moduleName)
   {
      edit(moduleName + "Test");
      return super.test(moduleName);
   }
}
```

CProgrammer class:

```
public class CProgrammer extends CompiledProgrammer {

   protected boolean purify(String moduleName)
   {
      System.out.println("Purifying " + moduleName);
      return randomChance(.50);
   }

   public boolean test(String moduleName)
   {
      experience++;
         // if passed execution test, also run Purify test
      return (super.test(moduleName) && purify(moduleName));
   }

   public double noBugProbability()
   {
      return Math.min(experience*.05, .90);
   }

   protected int experience = 0;
}
```

JavaProgrammer class:

```
public class JavaProgrammer extends CompiledProgrammer {

   public boolean test(String moduleName)
   {
      for (int i = 0; i < platforms.size(); i++) {
         System.out.println("Testing on platform " + platforms.elementAt(i));
         if (!super.test(moduleName))
            return false;
      }
      return true;
   }

   public double noBugProbability()
   {
      return .75;
   }

   static protected Vector platforms = new Vector(); // string names
}
```

4) C coding

```
void RemoveAll(void *elems, int numElems, int elemSize,
               CompFn cmp, const void *elem)
{
   int i;
   void *cur;

   for (i = 0; i < numElems; i++) {
      cur = (char *)elems + i*elemSize;
      if (cmp(cur, elem) == 0) {
         memmove(cur, (char *)cur + elemSize, elemSize*(numElems - (i+1)));
         numElems--;
         i--;           // back up to catch element just shifted down
   }
}



void Foo()
{
   char *colors[] = {"blue", "red", "green"};
   char *red = "red";

   RemoveAll(colors, 3, sizeof(char *), CompareStrings, &red);
}


int CompareStrings(const void* str1, const void* str2)
{
   return strcmp(*(char **)str1, *(char **)str2);
}
```

5) Short answer

a) Both 1 and 3 are fine, but 2 is not, since an array name is not an assignable L-value.

b) Ask yourself: what sort of errors does C catch at compile-time? Most of them are concerned with type-checking: wrong number/type of parameters passed to a function, mismatched types in expressions, calling a function which doesn't exist, etc. For most of these, LISP will allow you to defun code which makes these types of errors, but only when the code is actually executed does it spot the error using the runtime type information The language feature here is C's required type declaration and static type-checking posed against Lisp's untyped nature, parameter flexibility and dynamic runtime safety. Which would you rather have? Why do you have to choose? Why can't there be both?

c) There are many examples. Some have to do with Java's safety features: accessing an array element out of bounds or attempting an improper cast throws a runtime exception in Java, but can crash or produce junk in C. Some have to do with Java's commitment to portability: expressions that depend on the size of primitive types, values of uninitialized variables, or order of expression evaluation have well-defined results in Java and implementation-dependent behavior in C. And Java's built-in concurrency guarantees Java's standard printing method is protected under mutual exclusion, unlike C, so multi-threaded code that prints can have garbled results in C, but not in Java.

d) Synchronized methods are less error-prone. They provide a straightforward mechanism for simple binary lock around a method, avoiding errors such as using the wrong semaphore, reversed signal/wait, and unmatched signal/wait that can easily be introduced when using semaphores directly. As a downside, there may be reduced efficiency because the entire method is blocked, not just the critical code section within, and only one lock is shared by the whole class, so two synchronized methods that touch completely different parts of the object data will be unnecessarily serialized. Synchronized methods also are just binary locks, and don't do the general resource counting and message-passing of semaphores (although those mechanisms can be constructed with some work).

e) The better solution is declaring `area` as an abstract method in Shape. Declaring it `abstract` forces the subclasses to provide their own implementation, which is appropriate since each has its own particular version of area computation.

If we drop `area` completely from Shape, then we have not made it part of the Shape interface at all. It is no longer possible to exploit RT polymorphism and call `area` on each member of a Vector of "Shapes" since compile-time type checking requires that Shape must have an `area` method. It doesn't matter if every known subclass of Shape implements `area` either.

If Shape defines a bogus implementation that returns 0, it is possible for the subclasser to forget to override the method. Everything compiles and runs, but the default implementation is used, generating erroneous results and making it difficult to sort out what happened. Declaring it abstract creates an explicit hole that makes it more obvious what the subclass's responsibility is and enforces their part of the inheritance "contract".