CS107 Practice Solution

1) C and code generation

a)
```
R1 =.1 M[SP]      ;; load char (first array element)
R2 = R1 * 12      ;; multiply by sizeof(struct list)
R3 = M[SP + 8]    ;; load value of next field
R4 = R3 + R2      ;; add offset
M[SP + 8] = R4    ;; store back result
```

b) For all ordinary functions with declared arguments, it would be no trouble to reverse the calling conventions. However, C allows variable-argument functions like printf that can have fixed arguments (such as the format string) followed by an arbitrary number of other arguments. The fixed arguments need to be in a guaranteed location so they can be found, since they contain the key to interpreting the variable arguments. If the variable arguments are placed on the stack after the format string, it won't be possible for the callee know how far the offset is to those argument since there is an indeterminate amount of space in-between there.

c)
```
void EnterWordIntoGroup(void *elem, void *clientData)
{
   int length = strlen(*(char **)elem);
   Darray group = *(DArray *)ArrayNth((DArray)clientData, length);
   ArrayAppend(group, elem);
}

 void SummarizeGroup(void *elem, void *clientData)
{
   Darray group = *(DArray *)elem;
   printf("%d of length %d\n", ArrayLength(group),
                     strlen(*(char **)ArrayNth(group, 0));
}
```

Grading

These questions are graded similarly to the midterm. The code generation is usually just a matter of details: getting the correct offsets, no extra or missing indirection, etc. Sometimes there are troubles with the bigger conceptual issues of understanding the difference between a pointer and an array, knowing what exactly a typecast does— these are more serious mistakes. The DArray question comes down to really understanding how to interpret the void*— when you need the &, when you don't, when you need the typecast and what level of indirection it is, etc. Like the midterm, wrong levels of indirection and improper casting were costly mistakes. All of the many hours you spend on C coding this quarter should have taught you to be very, very careful about handling `void*` pointers so that you would never again succumb to their many pitfalls. Take the time to re-read issue #1 on the hw1c feedback handout for "the single most important thing to have learned from the RSG" (and all of homework 1 for that matter).

2) LISP

a) (defun weave (list1 list2)

```
(if (null list1) list2
    (cons (car list1) (weave list2 (cdr list1)))))
```

b) (defun closest-to-average (nums)

```
(let ((avg (/ (apply #'+ nums) (length nums))))
    (most #'(lambda(x y) (< (abs (- x avg)) (abs (- y avg)))) nums)))
```

c) (defun my-mapcar (list fn)

```
(maplist #'(lambda(sublist) (funcall fn (car sublist))) list)
```

d) (defun map-unique-pairs (fn list)

```
(apply #'append
   (maplist #'(lambda(sublist)
                  (mapcar #'(lambda(elem) (funcall fn elem (car sublist))
                     (cdr sublist)) (uniquify list))))
```

Grading

Only a few pieces of syntax, most notably use of ‘ and #’, are considered important, while other syntax glitches (mismatched parens, for example) are usually not graded off unless the syntax is consistently problematic or truly clouds your intention. Solving the problem in an awkward way is usually okay unless the result is really inefficient or unnecessarily convoluted. Things that are important: following the directions about whether to use recursion / mapping / other functions, knowing how to manipulate functions as data, forming and using lambda expressions, understanding the use of the lexical closure, being familiar with the common built-ins from the handouts and lecture, etc.

3) Concurrency

```
bool closed = false;
Semaphore lodgeLock;                    // binary, init to 1
Semaphore skis;                   // general, init to NUM_RENTAL_SKIS

void Clerk(void)
{
    int i;

    ThreadSleep(ALL_DAY);
    SemaphoreWait(lodgeLock); // ensure no one waiting for skis
    closed = true;                  // once closed, no more skis are taken out
    SemaphoreSignal(lodgeLock);
    for (i = 0; i < NUM_RENTAL_SKIS; i++)
      SemaphoreWait(skis);          // grab skis as they return
}



void RA(void)
{
    int i;
    Semaphore numDone = SemaphoreNew(“Count finished”, 0);

    Drive();
    for (i = 0; i < NUM_RESIDENTS; i++)
      ThreadNew(“Resident”, Resident, 1, numDone);
    for (i = 0; i < NUM_RESIDENTS; i++)
      SemaphoreWait(numDone);
    SemaphoreFree(numDone);
    Drive();
}

void Resident(Semaphore done)
{
    while (true) {
      SemaphoreWait(lodgeLock)
      if (closed) break; // quit skiing, lodge is closed
      SemaphoreWait(skis);          // important we do this before releasing
lock
      SemaphoreSignal(lodgeLock);
      OneRun();
      SemaphoreSignal(skis);  // return skis
    }
    SemaphoreSignal(lodgeLock);     // release lock!
    SemaphoreSignal(done);          // tell RA we're back at the van
}
```

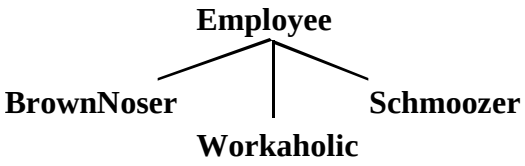
Grading

This question is graded with a focus on the concurrency issues. Whereas errors like an off-by-one on the loop bound, using the wrong size for the van or forgetting to call OneRun are minor or perhaps even no deduction, errors like busy-waiting, race conditions, and accessing global variables without protection are significantly counted. Less drastic errors, like initializing a semaphore to the wrong value or redundant locks, are counted somewhat less seriously. Solving the problem in the most direct and tidy

way is desirable, but an awkward and convoluted version that gets the job done with usually get most of the points as well.

4) Java

The classes described have pretty much just one good arrangement:



Changes to base Employee class:

Change Employee to be an abstract class and change body of `hasTime()` method to:

```
return (numTasksDone < getMaxTasks());
```

Add these methods to the Employee class:

```
public int getMaxTasks() {
   return MaxTasks;
}

public abstract boolean hasInterest(String taskName);

public boolean agreesToDo(Employee asker, String taskName) {
   return hasAbility(taskName) && hasTime() && hasInterest(taskName);
}

public boolean assignOne(Employee asker, String taskName) {
   if (agreesToDo(asker, taskName)) {
      performTask(asker, taskName);
      return true;
   } else
      return false;
}

public void doAllTasks(Employee asker, String tasks[])  {
   for (int i = 0; i < tasks.length; i++)
      assignOne(asker, tasks[i]);
}
}
```

BrownNoser class:

```
public class BrownNoser extends Employee {

   public boolean hasInterest(String taskName) {
      return boss.hasAbility(taskName);
   }

   public boolean agreesToDo(Employee asker, String taskName){
      return (asker == boss) || super.agreesToDo(asker, taskName);
   }
}
```

Workaholic class:

```
public class Workaholic extends Employee {

   public boolean hasInterest(String taskName) {
      return true;
   }

   public int getMaxTasks() {
      return super.getMaxTasks() + numRefusals;
   }

   public void performTask(Employee asker, String taskName) {
      super.performTask(asker, taskName);
      super.performTask(asker, "Checking "); // check over our work
   }

   public boolean agreesToDo(Employee asker, String taskName) {
      if (super.agreesToDo(asker, taskName))
         return true;
      else {
         numRefusals++;
         return false;
      }
   }

   protected int numRefusals;
}
```

Schmoozer class:

```
public class Schmoozer extends Employee {

   public boolean hasInterest(String taskName) {
      amInterested = !amInterested; // invert each time
      return amInterested;
   }

   public void performTask(Employee asker, String taskName) {
      super.performTask(asker, taskName);
      peopleWhoOweMe.addElement(asker);
   }

   public boolean assignOne(Employee asker, String taskName) {
      for (int i = 0; i < peopleWhoOweMe.size(); i++) {
         Employee e = (Employee)peopleWhoOweMe.elementAt(i);
         if (e.assignOne(this, taskName)) { // found a victim!
            performTask(asker, "Delegating");
            peopleWhoOweMe.removeElementAt(i);
            return true;
         }
      }
      return super.assignOne(asker, taskName);
   }

   protected boolean amInterested;
   protected Vector peopleWhoOweMe;
```

```
}
```

Grading

The grading for this question is designed to evaluate your ability to factor the common behavior of the classes using inheritance and avoid repeating code. Getting details of the problem description wrong (e.g. whether a particular task is interesting to a worker) are small to no deduction, depending on how significant that detail is. The bulk of the points are allocated for arranging the operations into small methods that make it easy to override to replace and extend, unifying the code for the various behaviors in common (deciding whether to do a task, the basic skeleton of doAllTasks, etc.), being sure to use super in an inherited method to include parent behavior rather than repeating code, and so on.

5) Short answer

a) C type-checks functional arguments. The function passed to `ArraySearch` needs to match the prototype of an `ArrayCompareFn`. If you pass `strlen`, for example, it doesn't compile. In LISP, passing `#'length` to `look` doesn't get caught until runtime. However, C only checks the surface. If you pass a function that has the correct prototype, yet casts its arguments incorrectly at RT (casts strings to num, e.g.) you get no warning of your mistake. If you pass `#'oddp` function to `look` on a list of strings, LISP gives you a message about type mismatch when the function is invoked on an element that doesn't have the correct RT type.

b) Portability means that the source code can be re-used, but architecture-neutral goes beyond that and assures that there are no architecture-specific features at all and that the compiled code can be re-used. This means you can post a program on the Web, in compiled byte-code form (not source!) and it can be downloaded and run on any Java VM. The disadvantages come in the form of inefficiency because the byte codes can't be optimized for any architecture and the byte codes have to be translated during execution.

c) You get a compile-time error. The compiler only knows that the result is Object and thus will only compile message expressions that are specific to the Object class for it. Even though the element doesn't forget it is a String and does respond to the length message, you cannot compile an expression sending length to it because of Java's CT message type checking.

d) What interference there might be from executing `func1` and `func2` in parallel? To have trouble, there must be contention for some shared resource. They both take the parameter `a`, so is that the problem? In LISP, all parameters to a function are read-only and any results in the function are constructed by creating new data, not by changing any state. For example, in pure LISP (what we used) there isn't even an assignment statement! C parameters are passed by value and copies of a are made for the two functions, so for any ordinary variable this won't be trouble, but what if `a` is a pointer— now we have a problem! Even ignoring the parameter issue, C allows for other function side-effects (accessing global variables, printing, etc.) that LISP generally does not, so again C has potential for contention that doesn't exist in LISP.

Grading

You don't need to write a book to answer these questions, just a sentence or two with the pertinent facts is all that is required. You do need to support your point however, and that is usually better done with a specific example than a long paragraph of flowing prose. These questions are typically not assigned many points, so it usually not worth spending a lot of time giving more than one example or over-explaining yourself.