CS107 Handout 63
Autumn 2000 December 13, 2000

CS107 Final Solution

Problem 1: Lisp Coding

a) Write the predicate function `acceptable` which returns `T` whenever the specified element passes all of the predicates listed in the specified set of predicates. If the predicate list is empty, then you should return `T`. Use `car-cdr` recursion, and do not use `mapcar`.

```
> (acceptable 10 (list #'atom #'numberp #'evenp))
T
> (acceptable 10 '(list #'atom #'numberp #'oddp))
NIL
> (acceptable 'will-pass-nothin '())
T
```

The car-cdr version should look as follows:

```
(defun acceptable (elem predlist)
  (if (null predlist) T
      (and (funcall (car predlist) elem)
           (acceptable elem (cdr predlist)))))
```

Most students nailed this one entirely, so you've all got that car-cdr thing down. Even though it wasn't the correct way to write it, you might acklowledge that the following mapcar version is tres' cute as far as LISP routines go.

```
(defun acceptable (elem predlist)
   (eval (cons 'and
               (mapcar #'(lambda (pred)
                     (funcall pred elem)) predlist))))
```

(Because and is a special form, it's not a true function object and can't be applied or funcalled like normal function objects can be. It was this limitation that actually prompted me to use this for the car-cdr question instead of the first mapcar question.)

b) Write a routine called sort-lists, which takes a list of lists and produces the same list of lists where all sub lists have been sorted according to the specified predicate object. You should use mapcar, lambda, and the built-in sort routine to accomplish this. Do **not** use explicit car-cdr recursion.

```
> (sort-lists '((11 4 6 2) (2 9 4 2 6) (3 2 5 5 1)) #'<)
((2 4 6 11) (2 2 4 6 9) (1 2 3 5 5))
> (sort-lists '((11 4 6 2) (2 9 4 2 6) (3 2 5 5 1)) #'>)
((11 6 4 2) (9 6 4 2 2) (5 5 3 2 1))
> (sort-lists '(("Marcia" "Jan" "Cindy") ("Greg" "Peter" "Bobby")) #'string<)
(("Cindy" "Jan" "Marcia") ("Bobby" "Greg" "Peter"))
```

Everyone nailed this one as well.. this was in place to ensure that you understood the need for an anonymous function being defined within the scope of comp.

```
(defun sort-lists (list-of-lists comp)
  (mapcar #'(lambda (list)
               (sort list comp)) list-of-lists))
```

c) Finally, write a function called `max-width`, which examines a list and returns the length of the widest of all nested lists. You should use `mapcar` and either `apply` or `funcall`; you should **not** use any explicit `car`-`cdr` recursion.

```
> (max-width '(1 2 3 (4 5 6)))
4
> (max-width '(1 2 3 (4 5 6 7 8)))
5
> (max-width '(1 2 3 (4 5) 6 (7 8)))
6
> (max-width '((1 2 3 (4 5 6 7 8)) (9 10 (11 12 (13 14 15) 16))))
5
```

This was one of the more difficult problems on the exam. It required a solid understanding of recursion and mapping.

```
(defun max-width (list)
  (if (atom list) 0
      (max (length list)
           (apply #'max (mapcar #'max-width list)))))
```

There was some confusion about whether or not max took a list of elements or a flat sequence, but we didn't dock students for whatever decision they made, since it required memorization, and that seemed unfair. Basically, we looked for the right recursive breakdown and a proper understand as to how to define the function to call itself using mapcar. The details of apply became important if you needed to invoke a function object.

Problem 2: Going to Disneyland

For this problem, we're going to simulate a family trip to Disneyland.

Attached to the back of the exam is a simulation for a single family trip to Disneyland. Mom has decided that she and her six children have 12 hours to spend in the park. Mom unleashes all of her children to run off and do their own thing for 12 hours while mom sits and enjoys twelve hours of peace and quiet. Each of the children goes on as many rides as possible in the time allowed, but each respects the twelve-hour time limit and finds mom after twelve hours have passed. Once all six children have found mom, mom and kids all go home.

Using C code only if it's clearer to do so, discuss all of the synchronization and thread communication issues present in this problem. Mention what variables need to be passed from the `Mom` thread to each of her children threads, and discuss all semaphores needed, what threads wait and signal them, and what their initial values need to be set to. You may present your answer in any format, provided it is clear and conveys as much detail as necessary to convince me you understand concurrency. You have this and the next two pages for your discussion. Take at least 10 minutes just to analyze the sample output before you start writing.

Use the following function prototype and the specified main program to help catalyze your discussion. The `Mom` and `main` functions below are part of the simulation code used to generate the sample output.

```
static void Mom(int numChildren, int numHours);

int main(int argc, char **argv)
{
  InitThreadPackage(false);
  ThreadNew("Brady", Mom, 2, 6, 10);
  RunAllThreads();
  return 0;
}
```

Six points out of ten are allocated for proposing a reasonable solution-- including the mother executing numChildren SemaphoreWaits on a global semaphore that is signalled once by each of her children at the end of the day and some way for the kids to know the day is over. Two points are allocated for either presenting a correct code implementation or writing a coherent, non-wrong essay describing your implementation. Two additional points are allocated for a cogent, correct explanation of your solution. These last two points mostly boiled down to what scheme you proposed for the children realizing that their day was over. If you chose to have a boolean signalled by the mother, you needed to explain whether a semaphore was or was not required to protect the boolean. The actual answer is no-- n threads reading and one thread using an atomic write operation means no data corruption was possible. Many students proposed a semaphore to prevent kids from sneaking in one extra ride before the day was over-- however, the only way you could guarantee this was to have the semaphore protect not merely reading the boolean, but also the process of taking a ride. This is an unacceptable level of serialization. It was equally valid to proposed the mother pass a number of hours to the children, with each child calling `time` after each ride to determine when to return.

Problem 3: Java Runtime

You are given the following full Java class definitions:

```
public abstract class Gershwin {

   public abstract void piano();
   public void harmony() { System.out.println("Gershwin.harmony");
                           rhythm(); }
   public void rhythm() { System.out.println("Gershwin.rhythm"); }
}

public abstract class Copland extends Gershwin {

   public void motive() { System.out.println("Copland.motive"); }
   public void cadence() { System.out.println("Copland.cadence");
                           motive(); }
}

public abstract class Barber extends Gershwin {

   public void piano() { System.out.println("Barber.piano"); }
   public abstract void motive();
   public void rhythm() { System.out.println("Barber.rhythm");
                          motive(); }
}

public class Bartok extends Barber {

   public void piano() { System.out.println("Bartok.piano");
                         harmony(); }
   public void motive() { System.out.println("Bartok.motive"); }
   public void harmony() { System.out.println("Bartok.harmony");
                           super.harmony(); }
}
```

Consider the following method declared to take a `Gershwin` as a parameter:

```
void play(Gershwin tune)
{
   tune.piano();
}
```

What are the possible types of objects that `tune` may be referencing at runtime? For each possibility, trace through a call to `play` and print the expected output.

Well, the keyword abstract is a dead giveaway—only Bartok objects can be created at runtime, so tune would always reference a Bartok object (unless there are addition subclasses of Gershwin beyond the four presented here.) Since all methods were instance and non-static, the most specific version of any one method would be invoked on behalf of the object. That means:

```
Bartok.piano
Bartok.harmony
Gershwin.harmony
Barber.rhythm
Bartok.motive
```

Problem 4: Object Oriented Design

For this problem you will write classes to model donuts. Any cop'll tell you that there are really only three types of donuts worth mentioning: glazed donuts, sprinkled donuts, and twisted donuts.

a. All donuts track their current integer weight (in ounces), which can be arbitrarily large but can not be less than 0. Some of these donuts get very big. Sprinkled donuts, and only sprinkled donuts, also track the current integer count of sprinkles, which can be arbitrarily large but never falls below `0`.
b. When sent the `bite` message, all donuts react by screaming (`System.out.println("Ouch!)`) and decrementing their weight by one ounce (never going below 0.) Twisted donuts, being twisted and all, whisper to themselves that they secretly like being bitten (`System.out.println("(Ooo.. I liked that.)").`). Sprinkled donuts, by the way, lose the proportional number of sprinkles as well, and lament the loss out loud (`System.out.println("Bye, sprinkles.")`).
c. When sent the `dunk` message, a donut gasps for oxygen (`System.out.println("Gasp!")`) Both twisted and sprinkled donuts absorb enough coffee to double their weight and complain (`System.out.println("I feel fat.")`) about it, but glazed donuts can't absorb and are therefore unaffected by the dunk. Sprinkled donuts actually lose half their sprinkles (`System.out.println("I feel exposed.")`) **before** the coffee can even be absorbed.

Design a class hierarchy to model the donuts. Draw a little tree of your hierarchy. List the instance variables and methods (including their types/prototypes) for each class. This drawing will serve as your definition so include all the necessary type information. You will not need to deal with initialization or constructors. You may assume that all of you instance variables have been correctly set before your code runs, so long as it's clear from your class drawing what the correct value for each is. We will assume that all members are modified as `protected` so you do not need to deal with that either. Draw your tree below:

```
     class Donut (abstract)

        Methods:
           public void bite();
           public void dunk();
           protected void absorb();
                                   \
        Variables:                   \
         / protected int weight;       \
        /              \                 \
       v                \                  v
class Glazed                                    class Sprinkled

   Methods:                                        Methods:
      protected void absorb();                        public void bite();
                                                      protected void absorb();

                                                   Variables:
                                                      protected int numSprinkles;

                              \
                               v
                       class Twisted

                          Methods:
                             public void bite();
```

The criteria we used to correct your design and code

- Dedicate 12 points out of 20 to the structuring of the tree. 12/12 solutions will either be like mine above, or perhaps create an intermediate `abstract` class called `Absorbable` (which is also just dandy.) For any hierarchy that will clearly force the duplication of code, you should comment where code will need to be duplicated. This may require your looking at the implementation to see exactly what they did and pretend to foresee the code duplication from the tree. Grade according to the following bucket scheme:
  - 12/12 maximum code factoring, ideal solution
  - 10/12 very strong design, with one minor flub that could be excused in most cases
  - 6/12 has everything, but is clearly not what we wanted
  - 2/12 all functionality is repeated in subclass method implementation, and no attempt was made in any way to reuse code.

Now provide the code to implement the `bite` and `dunk` methods and any support methods. Use this and the rest of the exam to provide your answers. Maximize code consolidation by placing as much code and programming logic in your base class as possible.

```
public abstract class Donut {

   public void bite()
   {
      System.out.println("Ouch!");
      if (weight > 0) weight--;
   }

   public void dunk()
   {
      System.out.println("Gasp!");
      absorb();
   }

   protected void absorb()
   {
      weight *= 2;
      System.out.prinln("I'm feeling fat.");
   }
}

public class Glazed extends Donut {

   protected void absorb() {}

}

public class Sprinkled extends Donut {

   public void bite()
   {
      super.bite();
      if (weight >= 0) numSprinkles *= (weight)/(weight + 1);
      System.out.println("Bye, sprinkles.");
   }

   protected void absorb()
   {
      numSprinkles /= 2;
      super.absorb();
   }
}

public class Twisted extends Donut {

   public void bite()
   {
      super.bite();
      System.out.println("Ooo.. I liked that.");
   }
}
```

Criteria

Dedicate 8 points of the 20 to this, where 4 points is dedicated to correctness (and the fact that all functionality is accounted for), and 4 points is dedicated to the proper use of `super` to refer to code that would have been inherited had they not been overriding. Don't be so concerned about the order of `println`'s, as I told students that I didn't care so much if they needed that flexibility to maximize code. Also understand that many of them might not use `super` but instead call some `protected` helper methods implemented at the parent class level.