CSE 341 Midterm

(Open Book, Open Notes – 50 points total)

Autumn 1999

Name: ______________________________

Section: ______________________________

1. (4 points) Given the following Scheme code:

```
(define bird-name "Tweety")
(define worm "Wormie")

(define (bird worm) (string-append bird-name " eats " worm))
(define (go bird-name worm) (bird worm))
```

What is the result of:

```
(go "Big-bird" "Little-worm")
```

```
guile> (go "Big-bird" "Little-worm")
"Tweety eats Little-worm"
```

2. (4 points) In Java, why is it illegal to call a method from a static class function? For example, the following does not compile:

```
public class Foo
{
   public static void printBar()
   {
      System.out.println(bar());
   }

   public String bar()
   {
      return("bar");
   }
}
```

It is illegal to call a method from a static function because there is no `this` object. All methods have an associated `this` object. Class functions do not.

3. (6 points) Which of the following must be implemented as special-forms in Scheme?

   (a) `define` Yes

   (b) `list` No

   (c) `set!` Yes

   (d) `set-cdr!` No

   (e) `map` No

   (f) `if` Yes

4. (4 points) Like Java, Smalltalk does not allow multiple inheritance. Smalltalk also does not provide an *interface* construct. Give a good reason for why is this might be the case?

   Smalltalk is dynamically typed. You can send any smalltalk object any method you want. Interfaces in Java only provide a mechamism within the Java static type system for you to keep objects that aren't related to each other in the same variable.

5. (5 points) Write a Scheme function `(repeat item k)` that will create a list with `item` repeated `k` times. For Example:

```
(repeat 'a 3)        => (a a a)
```

   Is your function tail-recursive?

```
(define (repeat item k)
   (if (> 0 k)
       '()
       (cons item (repeat item (- k 1)))
   )
)
```

   This is not tail-recursive (perhaps your implementation is?).

6. (8 points) True/False:

   (a) ___False___ Scheme is a pure-functional language.

   (b) ___False___ Imperative languages do not allow side-effects.

   (c) ___True___ In Scheme, `if` could be written as a source-to-source transformation to `cond`.

   (d) ___True___ In Scheme, `cond` could be written as a source-to-source transformation to `if`.

   (e) ___True___ In Java, two threads are not both allowed to be in the same `synchronized` function at once.

   (f) ___False___ In Java, `final` classes cannot be instantiated.

   (g) ___False___ Overloading is when a subclass defines its own version of a parent class's method.

   (h) ___False___ Java functions are first-class.

7. (10 points) Given the following Java class definitions, answer the questions on the next page.

```
public abstract class DoubleFunction
{
   public abstract double apply(double val);

   public double[] map(double[] vals)
   {
      double[] out = new double[vals.length];
      for (int i=0; i < vals.length; i++)
         out[i] = apply(vals[i]);
      return out;
   }
}

public class InverseFunction extends DoubleFunction
{
   public double apply(double val)
      { return 1.0/val; }
}

public class SinFunction extends DoubleFunction
{
   public double apply(double val)
      { return Math.sin(val); }
}
```

Come up with a way to create a `DoubleFunction` that is the composition of two other `DoubleFunctions`. That is, given any two `DoubleFunctions`, `d1` and `d2`, you want to create `d3` to be their composition: `d3.apply(x) == d1.apply(d2.apply(x))`

One possible solution is:

```
public class CompositeFunction extends DoubleFunction
{
   DoubleFunction d1,d2;
   public CompositeFunction(DoubleFunction d1, DoubleFunction d2)
      { this.d1 = d1; this.d2 = d2; }

   public double apply(double val)
      { return d1.apply(d2.apply(val)); }
}
```

Give a code fragment that demonstrates how you would use your composition mechanism to map $1/\sin x$ over the array `[1.0, 2.0, 3.0]`.

```
double[] array = {1.0, 2.0, 3.0};

DoubleFunction f = new CompositeFunction(
                          new InverseFunction(),
                          new SinFunction());

double[] answer = f.map(array);
```

8. (9 points) Recall our own definition of *cons cells* from class using `lambda`s:

```
(define (my-cons the-car the-cdr)
   (lambda (which)
      (cond ((eq? which 'car) the-car)
            ((eq? which 'cdr) the-cdr)
            (else (error "cons cell usage error"))
      )
   )
)

(define (my-car the-pair)
   (the-pair 'car)
)

(define (my-cdr the-pair)
   (the-pair 'cdr)
)
```

Define `my-set-car!` and `my-set-cdr!`. You may make any modifications to the above definitions that you need (but you may not use primitives `car`, `cdr`, `cons`, `set-car!`, or `set-cdr!`).

First modify `my-cons`, `my-car`, and `my-cdr` as follows:

```
(define (my-cons the-car the-cdr)
   (lambda (which arg)
      (cond ((eq? which 'car) the-car)
            ((eq? which 'cdr) the-cdr)
            ((eq? which 'set-cdr!) (set! the-cdr arg))
            ((eq? which 'set-car!) (set! the-car arg))
            (else (error "cons cell usage error"))
      )
   )
)

(define (my-car the-pair)
   (the-pair 'car 'junk)
)

(define (my-cdr the-pair)
   (the-pair 'cdr 'junk)
)
```

(Continued on next page)

Then add `my-set-car!` and `my-set-cdr!` as:

```
(define (my-set-car! the-pair val)
   (the-pair 'set-car! val)
)

(define (my-set-cdr! the-pair val)
   (the-pair 'set-cdr! val)
)
```