Fall, 1999 Name: _____

My Grading TA: _____ My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *SICP* Sections 1.3.2 through 2.2.1

This test has 8 questions and pages numbered 1 through 9.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 9pt font) of notes. Don't use anything with printing on the other side. No photo-reduction is permitted. You may not share your notes with anyone else during the test. These notes are to be handed in at the end of the test, so please have your name in the upper right hand corner of your notes. Use of other notes or failure to follow these instructions will be considered cheating.

WARNING: you won't have time to learn the material on during the test. Just write down what would be too tedious to remember otherwise.

If you need more space, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Of course, you may write helping procedures or methods whenever you wish. It may be helpful to give a comment that describes what they do, if it's not completely clear from the name.

You may assume that all Java code is in the same package on this test, unless the problem states otherwise.

1. (5 points) Briefly answer the following question. Would there be any advantage to adding something like Scheme's "lambda" to Java? Explain.

2. (5 points) Use the substitution model to find the result of the following Scheme expression.

```
(let ((t (lambda (x y) x))
      (f (lambda (x y) y))
      (i (lambda (x y z) (x y z))))
  (i f 4 (i t 5 6)))
```

3. Briefly answer the following questions.

   (a) (5 points) What properties does Scheme lack for supporting data abstraction?

   (b) (5 points) How could you change Scheme to better support data abstraction?

The following describes, in Scheme terms, a procedure, `foldr`, that you will implement in both Scheme and Java. This procedure is a version of `accumulate` that applies to lists. It works as shown in the following examples.

```
((foldr - 0) (list 300 40 2)) ==> 262
((foldr - 0) (list 40 2)) ==> 38
((foldr + 0) nil) ==> 0
((foldr + 0) (list 300 40 2)) ==> 342
((foldr * 1) (list 300 40 2)) ==> 24000
((foldr * 1) (list 40 2)) ==> 80
((foldr * 1) (list )) ==> 1
((foldr (lambda (x xs) (+ (expt 2 x) xs)) 0) (list 3 2)) ==> 12
```

That is, in general, the following hold for all procedures `combiner` of type `(-> (S T) T)`, values `null-value` of type `T`, values `x` of type `S`, and `xs` of type `(list S)`.

```
((foldr combiner null-value) nil)
          = null-value
((foldr combiner null-value) (cons x xs))
          = (combiner x ((foldr combiner null-value) xs))
```

4. (15 points) Implement the `foldr` procedure in Scheme.

5. (15 points) In Java, implement a public class `Foldr` that solves the previous problem for `combiner` procedures that take and return `double`s. That is, the `combiner` argument will implement the following interface in your Java code.

   For example, the following should output "`sum of myList = 342.0`" when run in Java.

```
public static void main(String [] argv) {
    Cons myList = new Cons(300.0, new Cons(40.0, new Cons(2.0, null)));
    DoubleConsFun myFoldr
        = new Foldr(new DoubleCombinerR() {
                        public double value(double x, double xs) {
                           return x + xs;
                        }
                    },
                   0.0);
    System.out.println("sum of myList = " + myFoldr.value(myList));
}
```

   Notice that the class you are to write must implement the interface `DoubleConsFun` given below. You don't have to write a method named `main`. (The code for the class `lib.Cons` is on the next page.)

```
import lib.Cons;
public interface DoubleConsFun {
    double value(Cons items);
}
```

   So, fill in your code by completing the following below.

```
import lib.Cons;
public class Foldr implements DoubleConsFun {
```

The following is the code for the class `lib.Cons` used in class and on the previous page.

```
// $Id: Cons.java,v 1.5 1999/11/05 19:28:33 leavens Exp $

package lib;

/** cons cells, or pairs, as in Lisp */
public class Cons {
    protected Object fst, snd;

    /** construct a Cons object */
    public Cons(Object x, Object y) {
        fst = x;
        snd = y;
    }

    /** an overload of the constructor, convenient */
    public Cons(double x, Object y) {
        fst = new Double(x);
        snd = y;
    }

    /** return the first element of the pair */
    public Object car() { return fst; }

    /** return the second element of the pair */
    public Object cdr() { return snd; }

    // internal iteration
    /** a reflection of Scheme's map procedure */
    public Cons map(Function body) {
        return new Cons(body.value(fst),
                        (snd == null)
                        ? null
                        : ((Cons)snd).map(body));
    }

    /** a reflection of filter in Scheme */
    public Cons filter(BooleanFunction pred) {
        if (pred.value(fst)) {
            return new Cons(fst,
                            (snd == null)
                            ? null
                            : ((Cons)snd).filter(pred));
        } else {
            return
                ((snd == null)
                 ? null
```

```
                : ((Cons)snd).filter(pred));
        }
    }

    /** return a String representation of the pair */
    public String toString() {
        return "(" + fst + " . " + snd + ")";
    }

    /** return a new pair that shares the components of this */
    public Object clone() {
        return new Cons(fst, snd);
    }

}
```

6. (30 points) Consider the following code in Scheme, which you will implement in Java on the next page.

```
(define (make-date month day year)
  (list day month year))

(define (date-get-day date)
  (car date))

(define (date-get-month date)
  (car (cdr date)))

(define (date-get-year date)
  (car (cdr (cdr date))))

(define (same-year? d1 d2)
  (= (date-get-year d1) (date-get-year d2)))

(define (same-month? d1 d2)
  (and (same-year? d1 d2)
       (= (date-get-month d1) (date-get-month d2))))

(define (earlier? d1 d2)
  (or (< (date-get-year d1) (date-get-year d2))
      (and (same-year? d1 d2)
           (< (date-get-month d1) (date-get-month d2)))
      (and (same-month? d1 d2)
           (< (date-get-day d1) (date-get-day d2)))))

(define (date-print date)
  (display (date-get-month date))
  (display "/")
  (display (date-get-day date))
  (display "/")
  (display (date-get-year date))
  (newline))
```

Please write your Java code on the next page.

Your task is to implement a class `Date` in Java, that corresponds to the Scheme code on the previous page. Part of this problem is for you to choose appropriate names for the methods, following Java conventions, and to choose the appropriate levels of privacy for each Java declaration. Hint: Don't use the class `Cons` or lists. You don't have to write a method named `main`.

7. (10 points) In Scheme, write a procedure, `minimum`, that takes a non-empty list of numbers, `items`, and returns an item that is no greater than any other item in `items`. The following are examples.

```
(minimum (list 3 4 2 4 2)) ==> 2
(minimum (list 2)) ==> 2
(minimum (list 7 5 99 22 5 -1 3)) ==> -1
```

In your solution, you may use the built-in Scheme procedure `min`, but you may only use it with exactly 2 arguments.

8. (10 points) In Scheme, write a procedure, `merge`, that takes two sorted lists of numbers, `first` and `second`, and returns a sorted list containing the numbers from both lists. The following are examples.

```
(merge (list 1 2 8 11 21) (list 3 4 7 12 13 20))
       ==> (1 2 3 4 7 8 11 12 13 20 21)
(merge (list 1 1 2 8 11 21 100) (list 3 3 4 7 11 12 13 20))
       ==> (1 1 2 3 3 4 7 8 11 11 12 13 20 21 100)
(merge (list 1) (list 1 2)) ==> (1 1 2)
(merge nil (list 1 2)) ==> (1 2)
(merge (list 1 2) nil) ==> (1 2)
(merge nil nil) ==> ()
```