Fall, 1999 Name: ______________________

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

Com S 342 — Principles of Programming Languages

Test on *SICP* Sections 2.2.2–3 and 2.4–2.5

This test has 6 questions and pages numbered 1 through 11.

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 unnamed package on this test, unless the problem states otherwise.

You may use any of the built-in procedures in Scheme.

1. (10 points) Which of the following features could be added to Scheme to make it easier for programmers to deal with multiple representations of data?

   (a) A graphical user interface builder toolkit, like the Java Swing classes.

   (b) Packages, as in Java.

   (c) Built-in support for type tags.

   (d) An exception handling mechanism.

   (e) While loops.

   In the space below, list the letter(s) of all that would be helpful, and write a sentence or two justification for each.

2. (10 points) Suppose you are managing a software effort for a company that makes computer and video game software. In the software that runs these games, there are a number of operations that all elements of the game support (move, run, shoot, fall, die, sigh), but the new elements are constantly being added to the game (for example, new characters). The software is currently written in C++ in an object-oriented style. One of your programmers thinks it would make maintenance of the software easier if the software was written using data-directed dispatch, with an operation-type table. Do you agree? Explain.

3. This problem explores multiple representations of calendar dates.

   (a) (10 points) In Scheme, write a procedure `make-four-digit-date`, that implements calendar dates in the message-passing style. (You are supposed to know what that means.) The version of `apply-generic` appropriate for message passing is shown below.

   ```
   (define (apply-generic op arg) (arg op))
   ```

   Using this version of `apply-generic`, your procedure should behave as in the following examples.

   ```
   (apply-generic 'year (make-four-digit-date 1999 12 16)) ==> 1999
   (apply-generic 'month (make-four-digit-date 1999 12 16)) ==> 12
   (apply-generic 'day (make-four-digit-date 1999 12 16)) ==> 16
   ```

(b) (15 points) In Scheme, write a curried procedure, `make-windowed-date-gen`, that takes a number, `window-start` that is between 0 to 99, and returns a procedure that takes a year, month, and date and returns a dispatch procedure. In this case the year is specified as a number `yy` that is between 0 and 99; if `yy` is strictly less than `window-start`, it is interpreted as 2000 + `yy`, otherwise it is interpreted as 1900 + `yy`. Using the version of `apply-generic` appropriate for message passing (see the previous page), your procedure should behave as in the following examples.

```
(define make-windowed-date-50 (make-windowed-date-gen 50))
(define make-windowed-date-20 (make-windowed-date-gen 20))

(apply-generic 'year (make-windowed-date-50 99 12 16)) ==> 1999
(apply-generic 'month (make-windowed-date-50 99 12 16)) ==> 12
(apply-generic 'day (make-windowed-date-50 99 12 16)) ==> 16
(apply-generic 'year (make-windowed-date-50 50 12 16)) ==> 1950
(apply-generic 'year (make-windowed-date-50 49 12 16)) ==> 2049
(apply-generic 'year (make-windowed-date-50 20 12 16)) ==> 2020
(apply-generic 'year (make-windowed-date-50 19 12 16)) ==> 2019

(apply-generic 'year (make-windowed-date-20 50 12 16)) ==> 1950
(apply-generic 'year (make-windowed-date-20 49 12 16)) ==> 1949
(apply-generic 'year (make-windowed-date-20 20 12 16)) ==> 1920
(apply-generic 'year (make-windowed-date-20 19 12 16)) ==> 2019
```

4. (15 points) In Scheme, write a “package” for use with data-directed dispatch that encapsulates four-digit dates (as in problem 3a above), and installs them in the operation-type table. (You are supposed to know what this means.) Using the version of `apply-generic` appropriate for data-directed dispatch (which is different than that for message passing), your procedure should behave as in the following examples.

```
(load-from-lib "tagged-data.scm")
(load-from-lib "operation-type-table.scm")
(load "install-four-digit-date-package.scm") ; your code is in this file

(define make-date (get 'make-four-digit-date 'four-digit-date))
(define exam-day (make-date 1999 12 16))

(apply-generic 'year exam-day) ==> 1999
(apply-generic 'month exam-day) ==> 12
(apply-generic 'day exam-day) ==> 16
```

Note that your package is to install operations named: `make-four-digit-date`, `year`, `month`, and `day`.

The following is the code for a few of the classes and interfaces in the package `lib.atomic_expression`, for use in the next problem.

```
// $Id: AtomicExpression.java,v 1.1 1999/11/15 05:26:53 leavens Exp $

package lib.atomic_expression;

/** A reflection of atomic expressions in Scheme.
 *
 * This uses the grammar:
   <pre>
          <atomic-expression T> ::= ()
                   |  <T>
                   |  ( <atomic-expression T> . <atomic-expression T> )

   </pre>
 * @see Nil
 * @see Atom
 * @see Expr
 **/
public interface AtomicExpression {
    Object visit(Visitor v);
}

// $Id: Nil.java,v 1.1 1999/11/15 05:26:53 leavens Exp $

package lib.atomic_expression;

public class Nil implements AtomicExpression {
    public Nil() {
    }
    public Object visit(Visitor v) {
        return v.visitNil(this);
    }
}

// $Id: Atom.java,v 1.1 1999/11/15 05:26:53 leavens Exp $

package lib.atomic_expression;

public class Atom implements AtomicExpression {
    protected Object obj;
    public Atom(Object obj) {
        this.obj = obj;
    }
    public Object getObject() {
        return obj;
    }
    public Object visit(Visitor v) {
        return v.visitAtom(this);
    }
}

// $Id: Expr.java,v 1.1 1999/11/15 05:26:53 leavens Exp $

package lib.atomic_expression;

public class Expr implements AtomicExpression {
    protected AtomicExpression left, right;
    public Expr(AtomicExpression left, AtomicExpression right) {
        this.left = left;
        this.right = right;
    }
    public AtomicExpression getLeft() {
        return left;
    }
    public AtomicExpression getRight() {
        return right;
    }
    public Object visit(Visitor v) {
        return v.visitExpr(this);
    }
}

// $Id: Visitor.java,v 1.1 1999/11/15 05:26:53 leavens Exp $

package lib.atomic_expression;

/** A top-level visitor interface for AtomicExpressions.
 * @see ToString
 * @see CountLeaves
 **/
public interface Visitor {
    /** visit a Nil atomic expression **/
    Object visitNil(Nil n);
    /** visit an Atom atomic expression **/
    Object visitAtom(Atom a);
    /** visit an Expr atomic expression **/
    Object visitExpr(Expr e);
}
```

5. (20 points) In Java, using the package `lib.atomic_expression` shown on the previous pages, write a class `SubstVisitor` that implements the interface `lib.atomic_expression.Visitor`. The constructor of `SubstVisitor` takes two non-null objects, `newObj` and `oldObj`; when a `SubstVisitor` is run on an object, it replaces all occurrences of `oldObj` with `newObj`. Occurrences are determined by calling the `.equals` method. The following are examples.

```
new Nil().visit(new SubstVisitor("D.C.", "Washington"))
  = new Nil()

new Atom(new Integer(3)).visit(new SubstVisitor("D.C.", "Washington"))
  = new Atom(new Integer(3))

new Atom("Washington").visit(new SubstVisitor("D.C.", "Washington"))
  = new Atom("D.C.")

new Atom("Paris").visit(new SubstVisitor("D.C.", "Washington"))
 = new Atom("Paris")

new Expr(new Atom("Washington"),
         new Expr(new Atom("London"),
                  new Nil())).visit(new SubstVisitor("D.C.", "Washington"))
 = new Expr(new Atom("D.C."),
            new Expr(new Atom("London"),
                     new Nil()))

new Expr(new Expr(new Atom("Washington"),
                  new Expr(new Atom(new Integer(342)), new Nil())),
         new Expr(new Expr(new Atom("Washington"),
                           new Expr(new Atom("London"), new Nil())),
                  new Nil())).visit(new SubstVisitor("D.C.", "Washington"))
 = new Expr(new Expr(new Atom("D.C."),
                     new Expr(new Atom(new Integer(342)), new Nil())),
            new Expr(new Expr(new Atom("D.C."),
                              new Expr(new Atom("London"), new Nil())),
                     new Nil()))

new Expr(new Atom("Boston"),
         new Expr(new Atom("Boston Harbor"),
                  new Atom("Boston Common")))
             .visit(new SubstVisitor("Ames", "Boston"))
 = new Expr(new Atom("Ames"),
            new Expr(new Atom("Boston Harbor"),
                     new Atom("Boston Common")))
```

Be sure to include the appropriate `import` directive in your code. You don't have to write a `main` method. There is space for your answer on the next page.

// (space for your answer to the problem on the previous page.)

The following is the code for two interfaces from the library that you will use in the next problem.

```
// $Id: Iterator.java,v 1.3 1999/12/16 16:26:53 leavens Exp $

package lib;

public interface Iterator {
    /** is there a current element? */
    boolean hasMore();

    /** move to the next element, provided hasMore() is true */
    void advance();

    /** get the current element, provided hasMore() is true */
    Object getElement();
}
```

```
// $Id: Thunk.java,v 1.1 1999/11/15 05:25:02 leavens Exp $

package lib;

public interface Thunk {
    Object value();
}
```

6. (20 points) In Java, write a class `EqualSeq` that implements the `lib.Thunk` interface (see the previous page). The public constructor for `EqualSeq` takes two `lib.Iterator` arguments. You can assume that these are not null. Your code should *not* iterate over the Iterator arguments until the `value` method is called. When this method is called, it determines whether the two sequences have the same length and if the corresponding elements are equal (using `.equals` to compare the elements); if so, it returns the `Boolean` object representing true (`Boolean.TRUE`) and returns `Boolean.FALSE` otherwise. The following are examples.

```
new EqualSeq(new ConsIterator(new Cons(2.0, null)),
         new ConsIterator(new Cons(2.0, null))).value()
    = Boolean.TRUE
new EqualSeq(new ConsIterator(new Cons(2.0, null)),
         new ConsIterator(new Cons(42.0, null))).value()
    = Boolean.FALSE
new EqualSeq(new ConsIterator(new Cons(2.0, null)),
         new ConsIterator(new Cons(2.0, new Cons(1.0, null)))).value()
    = Boolean.FALSE
new EqualSeq(new ConsIterator(new Cons(2.0, new Cons(1.0,
                                            new Cons(6.0, null))))
         new ConsIterator(new Cons(2.0, new Cons(1.0, null)))).value()
    = Boolean.FALSE
new EqualSeq(new ConsIterator(new Cons(3.0, new Cons(42.0, null))),
         new ConsIterator(new Cons(3.0, new Cons(1.0, null)))).value()
    = Boolean.FALSE
```