CSE 341: Homework 5

due 03/09/01

In this assignment you will be implementing a Scheme interpreter in Java. To make this a little less tedious we have provided you with an implementation of a `SchemeObject` class hierarchy. Functions for reading in `SchemeObject`s from an `InputStream` have also been provided. This is the same functionality that you implemented in Assignment 4 in Miranda. In section on 03/01/01 we will discuss the code you have been given to start with; however, this does not mean that you should wait until section to get started. Your goal in this assignment is to use the provided code and implement a Scheme interpreter.

Part I: (Not to be turned in.)

Do this before section on 03/01/01.

1. Look over the documentation and the code for the provided `SchemeObject` class and subclasses. `http://www.cs.washington.edu/education/courses/cse341/CurrentQtr/homework/hw5doc.html`
2. Write a Java program that executes *read-print* infinite loop. This repeatedly reads a Scheme object from the standard input and writes it to the standard output. You should use the `SchemeObject.read(InputStream)` function.
3. What classes in the standard Java class Hierarchy would be useful for creating `SchemeEnvironment` class?

Part II: (Not to be turned in.)

These will be discussed in section on 03/01/01.

1. Understand the `public static SchemeObject.read(InputStream)` function.

Part III: (Due Friday, 03/09/01. Turn this in.)

Complete the following. Follow the usual turn in requirements of submitting an electronic copy of your source code and turning in a hardcopy with test cases. For your electronic copy and your hardcopy you should include a `README` file with an explanation of your class structure, a description of how to run your interpreter, and a justification of your design decisions. This `README` will be graded.

Like Assignment 4, this assignment does not tell you exactly what to do. Instead you need to exercise good design choices. Your goal is to write a Scheme interpreter in Java using the partial SchemeObject hierarchy and parser provided (you may modify these). You must support the following special forms: `define`, `set!`, `lambda`, `if`, `cond`, `let`, and `quote`. You must support the following primitive functions: `+`, `-`, `*`, `=`, `<`, `eq?`, `equal?`, `cons`, `car`, `cdr`, `set-car!`, `set-cdr!`, and `list`. You may refer to R^4RS for a detailed specification of what all of these do. You must support the following Scheme data types: functions, integers, reals, strings, booleans, and lists. You do not have to support characters, vectors (arrays), rational numbers, tail-recursive optimization, or variable argument length user defined functions.

Below are some hints that you may choose to follow when completing this assignment.

1. Write class(es) to handle environments and environment frames. Recall that an environment is a list of environment frames and an environment frame is just a lookup table mapping variables (Scheme symbols) to values (Scheme data). You will at least need to implement functionality to perform the following operations with environments:

   - Add/Change a new variable binding to an environment (Note the difference between `set!` and `define`. You will have to implement both of them).
   - Lookup the value of a variable binding.
   - Create a new environment by adding a new environment frame to an existing environment.

   Do yourself a favor and test that this works before you try the next step. Independently testing subcomponents of a complicated program is a very time saving debugging practice.

2. A function in Scheme is a first-class object. Create subclasses of `SchemeObject` for primitive functions `=`, `eq?`, `cons`, `list`, `car`, `cdr`, and `apply`. Hint: make an abstract parent class (that is itself a subclass of `SchemeObject`). To do this, you need to create `SchemeNumberEqualsFunction`, `SchemePointerEqualsFunction`, `SchemeConsFunciton`, `SchemeListFunciton`, `SchemeCarFunction`, `SchemeCdrFunction`, and `SchemeApplyFunction` classes.

3. A big part of interpreting Scheme is manipulating cons-cells. This is a bit clunky in Java because Java is statically typed. You may find yourself performing lots of type casts.

   Create functionality for easily creating `SchemeList`s of length between zero and at least four. I.e. implement the Scheme `list` function in Java. Also create the functionality of mapping a scheme function over a list to create a new list. For example, mapping an instance of `SchemeCarFunction` over a list of lists would give the list containing the first element of each list. These were useful when manipulating cons-cells in Scheme, they will also be useful when manipulating your Scheme cons-cells in Java.

4. At this point, you should be able to create an environment and bind variables to values that are numbers, strings, lists of values, or *primitive* functions.

   Create an environment containing primitive bindings for primitive functions: `+`, `-`, `*`, `list`, `car`, `cdr`, and `apply`.

5. Write the code for evaluating a scheme object in an environment. This is similar to the `eval` function that we wrote in Scheme. Change your *read-print* loop program to have a *read-eval-print* loop. To do this you will need to have the global environment defined. Add special-forms `define` and `quote`.

6. Add the rest of the required special-forms and missing primitive functions. Recall that an easy way to add some special-forms is by using a source-to-source transformation to another special-form.

7. Test out `lambda` and lexical scoping. Can you implement the `factorial` function in your interpreter? How about `curry`?