CSE 341, Autumn 2006, Assignment 3
Scheme — Symbolic Data Part II; Macros
Due: Wed October 18, 10:00pm

20 points total (5 points each)

This is the second of two Scheme assignments dealing with symbolic data and metacircular interpreters. It uses material in Chapter 4 of the book *Structure and Interpretation of Computer Programs*. The last question is about Scheme macros, rather than the metacircular interpreter.

1. Do exercise 4.11 from SICP.

2. Do exercise 4.13 from SICP.

3. Add a `while` loop to the metacircular interpreter. The syntax is

   ```
   (while test
      expression1
      expression2
      ...
      expressionQ
   )
   ```

   The semantics is what one would expect: evaluate the test. If it's true, evaluate all the expressions, and go back and start the loop again. Continue until the test is false. The value of the entire `while` expression is `false`.

   For example, after evaluating the following

   ```
   (define k 0)
   (define n 1)
   (while (< k 10)
      (set! n (* 2 n))
      (set! k (+ k 1)))
   ```

   k should be 10, and n should be 1024.

   Implement this as a new special form. Hint: in implementing this, you can use `do` in the underlying ordinary Scheme. This is perhaps the world's most complicated loop construct, and can simulate both while loops and for loops (including loops with multiple iteration variables). The specification for the syntax and semantics of `do` loops is given in Section 4.2.4 of the *Revised*^{5} *Report on the Algorithmic Language Scheme*, which is linked from the Scheme page in the course web. If you take this approach, make sure you first understand how `do` operates — write some little test cases and experiment with it in ordinary Scheme. (Part of the purpose of this question is to ask you to read a programming language reference manual, and digest a new construct.)

4. This question is about Scheme macros, not the the metacircular interpreter — put your answer to this question in a different file.

   In ordinary Scheme, define `while` as a macro. Demonstrate that your macro is working correctly on some suitable test cases, and also that it expands correctly. As usual, invoke these from a `run-all` function.

   Hint: for debugging your macro, and also for showing how it expands, DrScheme has a couple of useful functions. `expand-once` takes an S-expression as an argument and partially expands any macros. This

returns something called a "syntax object." Then you can use `syntax-object->datum` to convert the syntax object to an S-expression. For example, if the `my-or` macro is defined as in the lecture notes, evaluating this expression:

```
(syntax-object->datum (expand-once '(my-or (= x 3) (= x 4))))
```

will show you what the `my-or` macro expanded into.

There is another function `expand` that expands all non-primitive syntax. (Unfortunately the results from either of these aren't always totally easy to understand. The `expand` function will not only expand your macros, but also Scheme syntax such as `let`. And `expand-once` may lose some context due to limitations in the expansion mechanism — see the Scheme report. However, this shouldn't be a problem for your `while` macro.)

Extra Credit. (10% max)

Here are several suggestions for extra credit problems.

Experiment with lazy evaluation in Scheme as discussed in Section 4.2 of SICP. Do some of the exercises in that section. In addition, if we are on to the Miranda language by the time you are doing this extra credit problem, try reimplementing some Miranda programs that take advantage of lazy evaluation in the lazy version of Scheme.

Implement the full `do` loop (as defined in the *Revised*^{5} *Report on the Algorithmic Language Scheme*) in the metacircular interpreter.

Finally, another possible extra credit problem is to add yet another kind of loop, a `named-while`. The syntax is:

```
(while name test
   expression1
   expression2
   ...
   expressionQ
)
```

There is also an exit expression: `(exit name)`.

Normally the named while operates just like a standard while loop. However, if an `exit` is encountered, the interpreter looks for a while loop with the same name as specified in the exit, and exits from that loop (and any inner loops if need be). It's an error if there isn't a loop found with the given name.

Unlike the other extensions, you will probably need to implement this using the primitive Scheme constructs.

Turnin: Turn in two files. The first should be your modified metacircular interpreter, including a `run-all` function that runs all of your code and test cases. The second should include your `while` macro, and some test cases that show the expansions of different loops, and the results of evaluating programs using while loops.