CSE 341, Winter 2010, Assignment 4
Scheme — Metacircular Interpreter
Due: Friday Feb 5, 10:00pm

45 points total (5 points for Question 1; 8 points each for Questions 2–6)

You can use up to 6 late days as you want for this assignment (but remember the advice to hoard them until the end of the quarter if possible).

This Scheme assignment concerns symbolic data and metacircular interpreters. It uses material from Chapter 4 through Section 4.1.6 of the book *Structure and Interpretation of Computer Programs*. The full text is available online (linked from the Scheme page); there is also a copy on reserve in the Engineering Library. A modified version of the metacircular interpreter is linked from the Scheme page of the class web; use that as a starting point for this assignment. There is also a set of unit tests as a starting point — you'll need to add some additional unit tests as well.

1. Add the following primitive functions to the interpreter: `+ - * / < = > eq? write display`. All of these should have the same meaning as the corresponding functions in the underlying DrScheme. Note that these should just be listed with the other primitive functions, and not hard-coded into the main `cond` in `mc-eval` — look at how `cons` is handled.

   Using these, define a `factorial` function, a `member` function, and a recursive `sum` function, and read these into the metacircular interpreter. Demonstrate your functions and your interpreter extensions are working with some suitable unit tests — there are unit tests you can copy for the primitives; you'll need to write additional tests for `factorial`, `member`, and `sum`. You can also add more primitive functions if you'd like.

2. One of the benefits of having a metacircular interpreter available is that it then becomes a platform for adding debugging facilities, or experimenting with new language constructs and even semantics. As a simple example of this, add an `inspect-it` function that takes any object `x` as its argument (in the metacircular interpreter). If `x` is a function, print out its parameters, body, and its environment in a readable way; otherwise just use the existing `write` function. You don't need to have a unit test for the `inspect-it` function — instead, in your unit test file include code that demonstrate your `inspect-it` function on a couple of examples, including an ordinary top-level function with nothing special in its closure, and another function that is closed over a new environment (not the top-level environment). There is code to do this in the sample test file. (See the extra credit options however regarding unit testing `inspect-it`.)

   Hint: `inspect-it` is a function that you'll need to bind in your global environment in the metacircular interpreter to a new primitive that you write in ordinary Scheme. The file `sample-output` (linked from the assignments page) has some example output. It's fine if your output varies, as long as it displays the needed information.

3. Do exercise 4.6 from SICP. Hint: test your `let->combination` function separately at first, to make sure it's working correctly. For example, if you execute `(let->combination '(let ((a 3)) (+ a 44)))` you should get back the resulting lambda expression.

4. Do exercise 4.7 from SICP. Hint: similarly, `let*->combination` can be tested separately at first.

5. Implement "while" loops in your metacircular interpreter. The syntax should be

   `(while <test> <expr1> <expr2> ...)`

   For example, the following code prints out the numbers from 1 to 9:

```
(define n 1)
(while (< n 10)
  (write n)
  (set! n (+ 1 n)))
```

Note that this example won't run in standard Scheme, since standard Scheme doesn't have a while loop. (It does have a more complex `do` special form though.)

You'll need to come up with unit tests for `while`.

6. Implement "for" loops in your metacircular interpreter. The syntax should be

   `(for (<var> <list>) <expr1> <expr2> ...)`

   The semantics are that `<var>` is bound successively to each element in `list` and the expressions are evaluated. `var` is freshly declared and has scope only including the expressions. If you assign to `var` in the middle of the loop (bad style, but allowed) it gets the new value for the remainder of the expressions, but then is set to whatever the next value should be on the next iteration.

   Here are two straightforward examples:

```
(for (x '(clam squid))
  (display x)
  (display " "))

(define (sum s)
  (let ((total 0))
    (for (i s)
      (set! total (+ i total)))
    total))
```

   The first expression prints `clam squid`. The second defines an iterative version of a function to sum the numbers in a list.

   Finally, the following example demonstrates that `for` has a local binding for the loop variable, and what happens when you assign to the loop variable.

```
(define y 100)
(for (y '(1 2 3))
  (display "y before the assignment: ")
  (display y)
  (set! y 20)
  (display "\n y after the assignment: ")
  (display y))
(display "\n y after the loop exits: ")
(display y)
```

   This should print:

```
y before the assignment: 1
y after the assignment: 20
y before the assignment: 2
y after the assignment: 20
y before the assignment: 3
y after the assignment: 20
y after the loop exits: 100
```

You'll need to come up with unit tests for `for`.

Turnin: Turn in two separate files: your modified metacircular interpreter, and your modified unit tests. Please name these `interpreter.scm` and `interpreter-tests.scm`.

Extra Credit. (10% max) Here are some possibilities. Please ask if you have an idea for something else you'd like to try!

- Define a new special form `define-struct`, like the one in Pretty Big Scheme. Do this as a derived expression that constructs a list `(begin ....)` that defines a constructor function, a test function, and access functions for each field. (You can just define read-only structs, and omit setters.) For example, given an expression `(define-struct point (x y))`, your `(begin ....)` list should include special forms to define `make-point`, `point?`, `point-x`, and `point-y`. Represent points as a tagged list, e.g. `(make-point 10 20)` should return `(point 10 20)`.
- It is often more difficult to write unit tests for interactive programs than non-interactive ones, and the `inspect-it` function is no exception. Devise a suitable unit test for this. (Hint: the Scheme `display` function allows an additional parameter, which says where to send the output. You could then test whether the output is the exact expected string, or better, search for important parts of it.)
- A usability problem with the current metacircular interpreter is that an error in your interpreted code bounces you back to ordinary Scheme. Change this so that an error in your interpreted code prints out an error message and resumes the interpreter with the existing environment. (To do this, change the calls to `error` in the interpreter to raise an exception, and catch this exception and restart.)
- Add a function `break` that adds a breakpoint at the given point in the code. This should then allow the user to type in and evaluate expressions in the current environment, and then to resume execution.

Other Hints

There isn't that much code to write for these questions — but with metacircular evaluators it's easy to confuse the code that the evaluator is interpreting, the code for the evaluator itself, and the Scheme primitives called by the evaluator. So you should study Section 4.1 carefully.

To start the interpreter, load it and type: `(run)`

In the version of Scheme understood by the evaluator, you need to enter `'()` for the empty list, as in R5RS — the shortcut `()` won't work.

Before you start on the questions, try starting up the interpreter and type in a few simple expressions to make sure it's working OK for you. Also try the `interpreter-tests.scm` file.

The metacircular interpreter doesn't include an error handling system — if you hit an error, you'll bounce back to ordinary Scheme. Try starting up the interpreter, and intentionally make an error, and restart the read-eval-print loop, to make sure you can recognize whether you are in the metacircular interpreter or ordinary Scheme.

Use the starter unit test file and additional unit test file in whatever way is convenient for you — one option is to start with the starter tests, and then copy in additional tests and add new ones of your own as you implement the various additions to the interpreter.

Assessment: Your solutions should be:

- Correct
- Tastefully commented
- In good style, including indentation and line breaks
- Well tested

The metacircular interpreter includes some side effects, but they are used sparingly. Your additional code shouldn't need any further side effects.