CSE 341, Autumn 2008, Assignment 4
Scheme — Metacircular Interpreter
Due: Tues October 28, 10:00pm

30 points total (5 points per question)

You can use up to 2 late days for this assignment — but the assignment needs to be turned in at the latest by 3:00pm on Thursday Oct 30 (so a few hours early). The reason for this is so that we can post the sample solution shortly after 3:00pm, so that students have time to look at it before the midterm if they want.

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 assignments page of the class web; use that as a starting point for this assignment. There is also a set of test cases that you can run. (You'll want to add some additional test cases as well.)

1. Add the following primitive functions to the interpreter: `+ - * / = eq?`. Using these, define a `factorial` function, a `member` function, and a `map` function, and read these into the metacircular interpreter. Demonstrate your functions and your interpreter extensions are working with some suitable test cases.
2. Do exercise 4.4 from SICP. (Install `and` and `or` as new special forms for the evaluator by defining appropriate syntax procedures and evaluation procedures `eval-and` and `eval-or`.)
3. 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 `user-print` function. 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 are examples of this in the test case file already.

   Hints: `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. Write a helper function `display-environment` to display an environment, for use with both `inspect-it` and the function for the next question.
4. Following up on Question 3, write another function `debug-info` with no arguments that prints out the current evaluation argument in a readable way. Again, demonstrate your `debug-info` function on a couple of examples.

   Hint: once you have your `display-environment` function (see Question 3), there is only a small amount additional code for this question. `debug-info` should be implemented as a special form, rather than as a primitive function, since you need to give it the current environment.
5. 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.
6. Do exercise 4.7 from SICP. Hint: similarly, `let*->combination` can be tested separately at first.

Extra Credit. (10% max) Here are three possibilities:

- 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 special form for a `for` loop. (In implementing this, you can use `do` in the underlying ordinary Scheme.)
- The `debug-info` function in Question 4 can be the beginning of a debug capability. 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 may want to study Section 4.1 carefully.

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

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.

After a while, the thrill of repeatedly typing the same program into the read-eval-print loop in your very own metacircular interpreter may wear off. If so, you may want to have copies of the definitions in another window, and copy and paste them into the interpreter. Alternatively, you can define some helper functions that run the interpreter with functions you're interested in. (You'll want something like this in any case for the `runtests` function described in the "turnin" note below.)

Turnin: Turn in two separate files: your modified metacircular interpreter, and a set of tests. For the tests, start with the `interpreter-tests` file and add additional tests as needed. Include your new tests in the `alltests` function at the end of the file, to make it easy to run them.

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.