CSE 341, Autumn 2006, Assignment 2
Scheme — Symbolic Data Part I
Due: Wed October 11, 10:00pm

35 points total

This is the first of two Scheme assignments dealing with symbolic data and metacircular interpreters. It uses materials in Chapters 2 and 4 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.

Question 1 involves extending the symbolic differentiation program in Section 2.3.2 of SICP. (This program is also linked from the sample code in the Scheme section of the course web.) This should be relatively easy. The remaining questions are based on the metacircular evaluator described in Section 4.1 of SICP; again, the code for the evaluator is also linked from the sample code section of the class web.

1. First, load the symbolic differentiation program into Scheme and try it, to make sure it is working OK and that you understand it. Now add the following extensions, by allowing the expressions to be differentiated to include:

   - the difference operator
   - sin and cos
   - raising an expression to an integer power

   Difference should be really easy — use sum as a model. The rules for the others are as follows:

   $$\frac{d(\sin u)}{dx} = \cos u(\frac{du}{dx})$$

   $$\frac{d(\cos u)}{dx} = -\sin u(\frac{du}{dx})$$

   $$\frac{d(u^n)}{dx} = nu^{n-1}(\frac{du}{dx})$$

   For simplification build in the rules that anything to the 0 power is 1, and anything to the power 1 is itself. Your code for this question should be written entirely in a functional style — no side effects.

   Demonstrate your code works correctly on a well-chosen range of examples.

2. For this question and the remaining ones, we'll be extending the Scheme metacircular interpreter discussed in Section 4.1. Add the following primitive functions to the interpreter: `+ - * =`. Using these, define a recursive `factorial` 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.

3. 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`.)

4. 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 a "trace" facility to the metacircular interpreter. To do this, implement functions `trace` and `untrace`. Each of these takes a single argument, namely a function to trace or

stop tracing. When a function is being traced, whenever it is called, print out an informative message giving the name of the function and the values of the arguments. Print another message when returning, giving the return value. The exact format of the trace information is left for you to design. (Also see the extra credit part of this assignment for additional posibilities.) Hint: note that `trace` is a function that you'll need to bind in your global environment in the metacircular interpreter, probably to a new "primitive" that you write in ordinary Scheme.

5. Following up on Question 4, write another function `inspect` that takes a function as its argument (in the metacircular interpreter), and that prints out its environment in a readable way. Demonstrate your `inspect` 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).

   As an example of closing a function over a new environemnt, consider the following code:

   ```
   (define (addit n)
     (lambda (k) (+ n k)))

   (define addtwo (addit 2))
   ```

   `addtwo` is closed over an environment that includes n (bound to 2).

6. Do exercise 4.6 from SICP.

7. Do exercise 4.7 from SICP.

Extra Credit. (10% max — the 10% would be only for a really snazzy graphical version) Add additional capabilities to your `trace` function. A simple addition would be indenting successive recursive calls in the trace, to help the user see the levels. A complex one would be a graphical version of the tracer, which uses graphics to show the call stack and so forth. (You'll need to figure out how to do graphics in DrScheme ... there is a way.)

There will be other extra credit opportunities with Part II of this homework.

Hints. Hopefully Question 1 will be straightforward, and will serve as a warmup to Scheme programs that manipulate other programs as data.

The other questions are all about the Scheme metacircular interpreter. 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:

```
(define the-global-environment (setup-environment))
(driver-loop)
```

(Or copy and paste these from the comments at the end of the file.)

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.

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, for example:

```
(define (testmap)
  (let* ((mapdef '(define (map f s)
                    (if (null? s)
                        '()
                        (cons (f (car s)) (map f (cdr s))))))
         (test1 '(map car '( (a b) (c d e))))
         (test2 '(map car '()))
         (env (setup-environment)))
    (display "defining map as follows: \n")
    (display mapdef)
    (newline)
    (eval mapdef env)           ; this defines map in the metacircular interpreter
    (display "testing map \n")
    (display test1)             ; this displays the test case test1
    (display "   =>   ")
    (display (eval test1 env))  ; this displays the result of evaluating test1
    (newline)
    (display test2)             ; this displays the test case test2
    (display "   =>   ")
    (display (eval test2 env))  ; this displays the result of evaluating test2
    (newline)
  ))
```

(You'll want something like this eventually for the `run-all` function described in the "turnin" note below.)

Turnin: Turn in two separate files: the symbolic differentiator, and your modified metacircular interpeter. For each, include a `run-all` function that runs all of your code and test cases.

Assessment: Your solutions should be:

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

Your code for Question 1 should be written entirely in a functional style — no side effects. You may need to have some side effects in some of the other code, but use them sparingly and only when necessary.