CS 61A Summer 2009 Homework 6

Topic: Metacircular Evaluator and Variants

Lectures: Monday 7/13, Tuesday 7/14, Wednesday 7/15, Thursday 7/16

Reading: Abelson and Sussman: 4.1.1-4.1.6, 4.2.1-4.2.2, 4.3.1-4.3.2

For the metacircular evaluator questions, you may use either the file ~cs61a/lib/mceval-final.scm, which is the lecture version, or the file ~cs61a/lib/mceval.scm, which is the book version.

1. Implement the **special form** `or` for our metacircular evaluator. Remember that `or` evaluates from left to right and stops as soon as it sees a value that isn't false. For example, using the `=61=>` interpreter:

```
> (or nay nay 23 (/ 3 0))
 23
```

2. Abelson and Sussman, 4.6

3. Abelson and Sussman, 4.14, and after you understand the problem, implement the **primitive procedure** `map` for our interpreter.

4. As you know, `set!` is a special form that does not evaluate its first argument. Implement in our interpreter, a **primitive procedure** `eval-n-set!` that evaluates both arguments and act like a `set!` if the first argument evaluates to a word, otherwise it signals an error. For example:

```
=61A=> (define beatles 'john)
  beatles
=61A=> (eval-n-set! (car (cons 'beatles nil)) 'paul)
  okay
=61A=> beatles
  paul
=61A=> (eval-n-set! (list 1 2) 'ringo)
  ERROR: not a symbol
```

`eval-n-set!` only needs to look at and change variables in the global environment. `eval-n-set!` does not need to consider the current environment. Remember that you should not have to modify `mc-eval` to build a new primitive procedure.

Note: What happens if you want `eval-n-set!` to be able to modify variables in the current environment? Can it be done if `eval-n-set!` is a primitive procedure?

5. Modify the metacircular evaluator to allow *type-checking* of arguments to procedures. Here is how the feature should work. When a new procedure is defined, a formal parameter can be either a symbol as usual or else a list of two elements. In this case, the second element is a symbol, the name of the formal parameter. The first element is an expression whose value is a predicate function that the argument must satisfy. That function should return `#t` if the argument is valid. For example, here is a procedure `foo` that has type-checked parameters `num` and `list`:

```
> (define (foo (integer? num) ((lambda (x) (not (null? x))) list))
    (list-ref list num))
FOO
> (foo 3 '(a b c d e))
D
```

```
> (foo 3.5 '(a b c d e))
Error: wrong argument type -- 3.5
> (foo 2 '())
Error: wrong argument type -- ()
```

In this example we define a procedure `foo` with two formal parameters, named `num` and `list`. When `foo` is invoked, the evaluator will check to see that the first actual argument is an integer and that the second actual argument is not empty. The expression whose value is the desired predicate function should be evaluated with respect to `foo`'s defining environment. (Hint: Think about `extend-environment`.)

6. (Optional) Read Abelson and Sussman 4.1.7, do exercise 4.23

Note: The following exercises concern the Lazy Evaluator. You may find the code at ∼`cs61a/lib/lazy.scm`

7. Abelson and Sussman 4.25, 4.26, 4.27

8. (Optional Read Abelson and Sussman 4.2.3, do exercises 4.32, 4.33

Note: The following exercises concern the Nondeterministic Evaluator. You may find the code at ∼`cs61a/lib/vambeval.scm`

9. Abelson and Sussman 4.35, 4.36

10. Write a program in the nondeterministic evaluator to solve Abelson and Sussman, 4.42

Extra for experts:

Part A: Abelson and Sussman, 4.15. This problem isn't very hard but is of critical importance to computer science.

This is known as the **Turing Halting Problem**. It implies that there does not exist any algorithm for debugging programs.

In fact, it implies a lot more! With lots of cleverness, one can use the Halting Paradox to show that there exist theorems in Mathematics that are **unprovable**, that is, no matter how smart you are, it is impossible to prove whether they are true or not.

Part B: For now, use the ideas behind the Halting Problem to prove that it is impossible to define a procedure `(fn-equal? fn1 fn2)` that takes two procedures of one argument and checks *in finite time* whether they have the same output behavior if given the same inputs. A program that runs forever on some input is considered to have no output for that input.

HINT: suppose `fn-equal?` exists and consider the following program:

```
(define (halt fn arg)
   (define (temp-fn x)
      (fn arg)
      x)
   (define (id x) x)
   (fn-equal? temp-fn id))
```

Even more interestingly, one can prove that it is possible to define `fn-equal?` to check the equalities of procedures that **has no loop or recursion**. Thus, procedures that has no loop or recursion is far less complex and powerful. Since we can consider loops as just a form of recursion, we arrive at one of the central principles of theory of computation: **the power of a computer program comes from recursion**. This is known as the *Church's Thesis*.

If you are interested in this topic, I highly recommend that you take *CS172: Computability and Computational Complexity* as some point.