Assignment 4: Interpreter and More Higher-Order Functions
CS351—Fall 2008

Due 23:59 Sun 2-Nov-2008. Email *one text file* containing *all* your solutions to: `barak+cs351-hw4@cs.nuim.ie`. (This file should be loadable into Scheme, meaning essay question answers should be in comments.)

For the first few questions, assume these are available:

```
(define reduce-r
  (lambda (f x0 lis)
    (if (null? lis)
        x0
        (f (car lis) (reduce-r f x0 (cdr lis))))))

(define reduce-l
  (lambda (f x0 lis)
    (if (null? lis)
        x0
        (reduce-l f (f x0 (car lis)) (cdr lis)))))
```

1. Define the function `comb` such that

   `(cadr (reduce-r comb '(-1e100 ?)  '((`x_0 v_0`) ... (`x_n v_n`))))`

   returns the v_i paired with the maximal x_i, where $x_0, \ldots, x_n$ are numbers $> -10^{100}$. (In the case where the list is empty it should return `?`.)

   Solution:

   ```
   ;;; Take two elements, return the one with the larger car
   (define comb
     (lambda (x y)
       (if (>= (car x) (car y)) x y)))
   ```

2. Define the function `comb-2` such that

   `(reduce-r comb-2 '((-1e100 ?))  '((`x_0 v_0`) ... (`x_n v_n`)))`

   returns a list of two elements, `((`x_i v_i`) (`x_j v_j`))` where x_i and x_j are the two largest numbers in $x_0, \ldots, x_n$, which themselves are, as above, $> -10^{100}$. (In the case where the list is empty it should return `((-1e100 ?))` and in the case where the list has one element it should return a list of two elements: that element and `(-1e100 ?)`.)

   Solution:

   ```
   ;;; Take an item and a list of items, return a list of the
   ;;; items (at most two) from them both with the biggest cars
   (define comb2
     (lambda (x ys)     ; x is new one, ys is list of best so far
       (let ((z (sort (cons x ys)
                      (lambda (a b)
                        (> (car a) (car b))))))
         (if (> (length z) 2)
             (take z 2)
             z))))
   ```

3. What is the difference, as far as users are concerned, between `reduce-r` and `reduce-l`? Show a set of arguments which, when passed to either, give different results.

Solution:

(reduce-l f x (e_1 ... e_n)) $\Rightarrow$ (f (f $\cdots$ (f (f x e_1) e_2)$\cdots$) e_n)

(reduce-r f x (e_1 ... e_n)) $\Rightarrow$ (f e_1 (f e_2 (f $\cdots$ (f e_n x) $\cdots$)))

These can differ if f is not associative, or if x is not an identity element of f.

Example of these differing:

(reduce-l list 'x '(a b c)) $\Rightarrow$ (((x a) b) c)

(reduce-r list 'x '(a b c)) $\Rightarrow$ (a (b (c x)))

4. Add macro expanders for and, or and cond to the meta-circular interpreter defined in class. (The expander for cond only needs to handle clauses of the form (test val) and the terminal clause (else val); you can forget (A => B) clauses.) Be sure these work for the cases of one, or even zero, argument forms. (Note that it is okay to use the same macros in sub-forms of the form being expanded, as long as the expansions ultimately terminate. For instance, you could expand (or A B C) to (let ((v A)) (if v v (or B C))) and rely on subsequent evaluation to expand the let and or forms contained in that output form.

   EXTRA CREDIT: Augment your macro expander for cond to handle (A => B) clauses and also clauses of the form (A) which return A if it is true, and go on to the next clause otherwise.

Solution:

```
;;; Added and modified code from lecture file: 10a.scm

(define expand-and
  (lambda (e)
    (cond ((null? (cdr e)) '#t)
          ((null? (cddr e)) (cadr e))
          (else (list 'if
                      (cadr e)
                      (cons 'and (cddr e)))))))

(define expand-or
  (lambda (e)
    (cond ((null? (cdr e)) '#f)
          ((null? (cddr e)) (cadr e))
          (else (list 'let
                      (list (list 'ttt (cadr e)))
                      (list 'if 'ttt 'ttt
                            (cons 'or (cddr e))))))))

(define expand-cond
  (lambda (e)


    (define expand-clauses
      (lambda (clauses)
        (if (null? clauses)
            '(void)
            (expand-clause (car clauses)
                           (expand-clauses (cdr clauses))))))

    (define expand-clause
      (lambda (clause else-code)
        (cond ((null? (cdr clause))
               (expand-clause (list (car clause) '=> '(lambda (x) x))
                              else-code))
```

```
              ((and (= (length clause) 3)
                    (equal? (cadr clause) '=>))
               (list 'let
                     (list (list 'v (car clause)))
                     (list 'if 'v (list (caddr clause) 'v) else-code)))
              ((and (= (length clause) 2)
                    (equal? (car clause) 'else))
               (cadr clause))
              ((= (length clause) 2)
               (list 'if (car clause) (cadr clause) else-code))
              (else (error "unknown cond clause shape" clause)))))

    (expand-clauses (cdr e))))

(define macro-alist
  (list (list 'let expand-let)
        (list 'or expand-or)
        (list 'and expand-and)
        (list 'cond expand-cond)))
```

5. The language PERL has a "tainted" tag that can be attached to values, which is propagated through computations. The intent is to ensure that "insecure" input does not result in security policy violations, by tracking its flow and raising an error if it flows into things like the `exec()` system call without sanitization.

   Make a similar modification to the meta-circular evaluator from the lecture notes. Represent an "untainted" value as currently represented, and a "tainted" value x as the list `(%tainted` x`)`. These are propagated through computations by the following rules: if any argument to a primitive function is tainted then its output is tainted. If the test of an `if` form is tainted then the result returned by the form is *not* necessarily tainted. Add "primitive" functions `taint` and `untaint` with the obvious semantics.

Solution:

```
;;; functions to take a value (possible already tainted) and
;;; make a corresponding tainted value, and to take a tainted
;;; value and make the corresponding untainted value, and to
;;; check if a value is tainted.

(define tainted?
  (lambda (x)
    (and (pair? x)
         (equal? (car x) '%tainted))))

(define taint
  (lambda (x)
    (if (tainted? x) x (list '%tainted x))))

(define untaint
  (lambda (x)
    (if (tainted? x) (cadr x) x)))

;;; Modify my-apply where it applies a primitive function
;;; to strip & propagate taintedness:

;;; OLD:
;;(define my-apply
;;  (lambda (f vals)
```

```
;;    (cond ((procedure? f) (apply f vals))
;;          ...)))

;;; NEW:
;;(define my-apply
;;  (lambda (f vals)
;;    (cond ((procedure? f) (apply-primitive f vals))
;;          ...)))

(define apply-primitive
  (lambda (f vals)
    (let ((y (apply f (map untaint vals))))
      (if (any? tainted vals)
          (taint y)
          y))))

(define any?
  (lambda (p? lis)
    (and (not (null? lis))
         (or (p? (car lis))
             (any? p? (cdr lis))))))
```

6. Make a test suite to test whether your "tainted" code above works. This should be automated, so I can load your code and type `(test-tainted)` and the result will be `#t` if all test cases succeed, and something else if one or more fail. Be sure your test cases have good coverage!

7. *(Optional)* If you encountered any problems with the assignment, or have any comments on it, or other comments or suggestions, I would appreciate hearing them. As practice for actual work, where weekly reports are not unusual, please embody these in a brief report.

Solution:

This is the best class ever. My only suggestion: longer harder assignments. And more of them!

Honor Code: You may discuss these with others, but please write your answers by yourself and without reference to communal notes. In other words, your answers should be *from your own head.*