Principles of Programming, Spring 2006
Practice 4

Daejun Park, Heejong Lee
Programming Research Lab.@SNU

March 30, 2006

1. Below is some equivalence predicates. What is the result printed by the interpreter in response to each expression? What is the result when you change `eq?` into `eqv?` or `equal?`

   ```
   (eq? 'a 'a)
   (eq? "a" "a")
   (eq? (cons 1 ()) (cons 1 ()))
   (let ((a (cons 1 ())))
    (eq? a a))
   (let ((p (lambda (x) x)))
    (eq? p p))
   ```

   You will need the following reference. http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Equivalence-Predicates.html

2. Modify your `reverse` procedure of practice 3.2 to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

   ```
   (define x (list (list 1 2) (list 3 4)))

   x
   ;((1 2) (3 4))

   (reverse x)
   ;((3 4) (1 2))

   (deep-reverse x)
   ;((4 3) (2 1))
   ```

3. Define a procedure `square-tree` analogous to the `square-list` procedure of practice 3.4. That is, `square-tree` should behave as follows:

```
(square-tree
  (list 1
        (list 2 (list 3 4) 5)
        (list 6 7)))
;(1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

4. We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `(1 2 3)`, then the set of all subsets is `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

5. Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\cdots (a_n x + a_{n-1}) x + \cdots + a_1) x + a_0$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0. Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) <??>)
              0
              coefficient-sequence))
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
;79
```