CSE 341: Homework 2

due 10/11/99

This assignment is to be turned in at the beginning of class on 10/11/99. As usual, all functions you are asked to implement and turn in should be typed and tested with the scheme interpreter. With each function turn in the source code listing as well as sample output for a demonstrative set of tests.

The topics covered in this assignment include: tail-recursion, applicative programming (first class functions), lexical scoping, and tree traversal. Do not use any functions with *side-effects*.

Part I: (Not to be turned in.)

Try these questions out before section.

1. Given the following definitions,

```
(define x 5)
(define y 4)
```

   What would the values of the following scheme expressions?

   (a)
```
(let ((x 3)
      (z (+ x 1))
     )
     (* x y z)
)
```

   (b)
```
(let* ((x 3)
       (z (+ x 1))
      )
      (* x y z)
)
```

   (c)
```
(let* ((z (+ x 1))
       (x 3)
      )
     (* x y z)
)
```

   (d)
```
(list
    (let ((x 3)
          (z (+ x 1))
         )
         (* x y z)
    )
    x y
)
```

2. Given definitions

```
(define x 5)
(define y 4)
(define (foo x)
   (+ x y)
)
(define (bar y)
   (foo y)
)
```

What is the value of

```
(bar 10)
```

3. Write a function `flatten` that takes a *tree* and returns a list that contains the leaves of the tree in left to right order. For Example:

```
(flatten '(1 ((2) 3) (4)))        => (1 2 3 4)
```

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section.

1. Define a tail-recursive version of factorial. Instead of defining a recursive helper function in the global environment, define it locally.
2. Partially evaluating a function by passing its arguments to it one argument at a time is called `currying` after Haskell B. Curry. The idea is that given an n argument function, you can specify one of the arguments and be left with an $n-1$ argument function. Implement the function `curry` that takes as its own arguments a two argument function f and an argument x and returns a new function g that takes only one argument, say y, such that $g(y) = f(x, y)$. This is illustrated with the following scheme code and example:

```
;; assuming f has no side-effects and that
;; x and y are defined and are valid arguments to f
(define g (curry f x))
(equal? (g y) (f x y))                => #t

;; more examples
(define add5 (curry + 5))
(define add2 (curry + 2))
(add5 10)                             => 15
(add5 2)                              => 7
(add2 10)                             => 12
```

3. Define a function `filter` that takes, as arguments, a function `f` and a list l. It returns a new list that only contains elements of `l` for which `f` returns true. For Example:

```
(filter number? '(a b 1 3 c 4 d)       => (1 3 4)
(filter symbol? '(a b 1 3 c 4 d)       => (a b c d)
(filter symbol? '())                   => ()
```

4. Use your `curry` function along with `filter` to re-implement `positive-only` from the first assignment. Do not define any new functions or *lambdas* explicitly. Use only `curry`, `filter`, and builtin scheme functions.

Part III: (Due Monday, 10/11/99. Turn this in.)

Complete the following. You do not have to handle errors in input gracefully. Do not forget to turn in output with your source code.

1. (5 points)

   A polynomial $ax^3 + bx^2 + cx + d$ can easily be evaluated using Horner's Rule as $(x(x(x(a) + b) + c) + d)$. This gives a very natural recursive way to evaluate a polynomial at some point x. Take the first coefficient, multiply by x, and add the next coefficient. Then take this value, multiply by x, and add the next coefficient, etc. Define a tail-recursive function `eval-poly` that takes a point `x` and a list of coefficients `coefs` and evaluates the polynomial at the point `x`. Assume the coefficients are ordered such that, $x^2 + 2x + 3$ is expressed by the list `(1 2 3)`. For example:

   ```
   (eval-poly 1 '(1 2 3))          => 6        ;; 1 + 2 + 3
   (eval-poly 2 '(1 2 3))          => 11       ;; 1*2^2 + 2*2 + 3
   ```

2. (10 points)

   Recall the `sum-tree` function written in class. It can be reorganized to look much like the solution to the `flatten` function in Part I.

   ```
   (define (sum-tree tree)
      (cond ((null? tree) 0)
            ((pair? tree) (+ (sum-tree (car tree)) (sum-tree (cdr tree))))
            (else tree)
      )
   )
   ```

   (a) Abstract the structure of `flatten` and `sum-tree` so that you can implement these both with a simple call to a first-order function `tree-fold`. Write `tree-fold`. Hint: The *identity* function is the function that returns its argument. I.e. `(identity x)` $\rightarrow$ `x`. You will have to define `identity` yourself. It is not in the global environment.

   (b) Re-implement `flatten` and `sum-tree` using `tree-fold`.

3. (10 points) Define a lookup table that can be used to lookup a value associated with a given symbol. Use a binary search tree implementation for your underling data type. Implement the following operations:

`lookup table sym` This looks up a symbol, `sym` in the table `table`. If the symbol found in the table then return the value. If the symbol is not found then return return `'symbol-not-found`.

`insert table sym value` This returns a new table with where a subsequent call to `(lookup table sym)` should return `value`.

`contains table sym` This returns `#t` if `sym` is defined in `table`, `#f` otherwise.

`emptytable` This creates a new empty table and returns it.

Some example usages:

```
(contains (emptytable) 'x)                      => #f
(define t1 (insert (emptytable) 'x 5))
(contains t1 'x)                                => #t
(lookup t1 'x)                                  => 5
(lookup t1 'y)                                  => symbol-not-found

(define t2 (insert (insert t1 'y 10) 'z 9))
(define t3 (insert t2 'foo '(1 2 3)))

(map (curry lookup t3) '(x y z))                => (5 10 9)

(lookup t3 'foo)                                => (1 2 3)
```

Your table should be able to hold any type of data not just numbers.

Hint: To compare two symbols you can convert them to strings using `symbol->string` and then compare them as strings using the `string<?`, `string>?`, etc. relations.