Homework Assignment 2

Solutions are expected to only use functions from the standard library that was taught. Before using a function from the standard library inquire if you are allowed to use it. The grader is **not testing** that you are using using disallowed code. If a solution uses a disallowed function, the autograder score is voided.

1. (15 points) Your goal is to implement a memory cell encoded with functions as data. A memory cell is a function that takes a list as its only argument. The list may have zero or one elements.

   - If the list has one element, then we are *setting* the value of the memory cell. Setting the value of a memory cell returns a new memory cell.
   - If the list is empty, then we are *getting* the value contained in the memory cell.

   The following definitions abstract these notions:

   ```
   (define (cell-get c) (c (list)))
   (define (cell-set c x) (c (list x)))
   ```

   (a) (7.5 points) Implement a read-write cell. Function `rw-cell` takes a number that initializes the memory cell. Operation *set* returns a new cell with the value given. Operation *get* returns the contents of the cell.

   ```
   (define w1 (rw-cell 10))
   (check-equal? 10 (cell-get w1))
   (define w2 (cell-set w1 20))
   (check-equal? 20 (cell-get w2))
   ```

   (b) (7.5 points) Implement a read-only cell. Function `ro-cell` takes a number and return a read-only cell. Putting a value in a read-only cell should not change the stored value. Getting a value should always return the initial value.

   ```
   (define r1 (ro-cell 10))
   (check-equal? 10 (cell-get r1))
   (define r2 (cell-set r1 20))
   (check-equal? 10 (cell-get r2))
   ```

2. (15 points) Implement a **tail-recursive** function `intersperse` that takes a list `l` and an element `e` and returns a list with the elements in list `l` interspersed with element `e`. *The implementation must only use the list constructors and selectors that we covered in our class.* That is, return a list where we add element `e` between each pair of elements in `l`. For instance,

   ```
   (check-equal? (list 1 0 2 0 3) (intersperse (list 1 2 3) 0))
   ```

3. (15 points) Implement a **tail-recursive** function `find` that takes as arguments a function `predicate` and a list `l` and returns either a pair index-element or `#f`. *The implementation must only use the list constructors and selectors that we covered in our class.* The objective of the function is to find a index-element in a list given some predicate. Function `find` calls function `predicate` over each element of the list until the `predicate` returns true. If `predicate` returns true, then function `find` returns a pair with the zero-based index of the element and the element.

   Function `predicate` takes an integer (the zero based index in the list) and the element we are trying to find.

   (a) (10 points) Implement function `find`.

   ```
   (check-equal? (cons 0 10) (find (lambda (idx elem) #t) (list 10 20 30)))
   (check-equal? #f (find (lambda (idx elem) #f) (list 10 20 30)))
   ```

(b) (2.5 points) Implement function `member` in terms of function `find`. Function `member` takes an element `x` and a list `l` and returns `#t` if the element `x` is in list `l`, otherwise it returns `#f`.

```
(check-true (member 20 (list 10 20 30)))
(check-false (member 40 (list 10 20 30)))
```

(c) (2.5 points) Implement function `index-of` in terms of function `find`. Function `index-of` takes a list `l` and an element `x` and returns the index of the first occurrence of element `x` in list `l`, otherwise it returns `#f`.

```
(check-equal? 1 (index-of (list 10 20 30) 20))
(check-equal? #f (index-of (list 10 20 30) 40))
```

4. (10 points) Implement function `uncurry` which takes as argument a curried function `f` and returns a new function which takes as parameter a list of arguments which are then applied to `f`. *The implementation must only use the list constructors and selectors that we covered in our class.*

```
(define (f x y z w)
  (+ x y z w))
(define g (uncurry (curry f)))
(check-equal? 10 (g (list 1 2 3 4)))
```

5. (45 points) Recall the AST we defined in Lecture 5. Implement function `parse-ast` that takes a datum and yields an element of the AST. You will need as auxiliary functions `real?` and `symbol?` from Racket's standard library and functions `lambda?`, `define-basic?`, and `define-func?` from Homework Assignment 1 (Part II).

   The function takes a datum that is a valid term. Expression do **not** include conditional nor booleans, only functions declarations, definitions, variables, and numbers.

```
(check-equal? (parse-ast 'x) (r:variable 'x))
(check-equal? (parse-ast '10) (r:number 10))
(check-equal?
  (parse-ast '(lambda (x) x))
  (r:lambda (list (r:variable 'x)) (list (r:variable 'x))))
(check-equal?
  (parse-ast '(define (f y) (+ y 10)))
  (r:define
    (r:variable 'f)
    (r:lambda
      (list (r:variable 'y))
      (list (r:apply (r:variable '+) (list (r:variable 'y) (r:number 10)))))))
```