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. Your goal is to implement a memory cell encoded with functions as data. Recall §2.1.3 of the SICP book, in particular the implementation of functions `cons`, `car`, and `cdr`.

   A memory cell is a function value that expects exactly one argument: a list. According to the contents of the list, the memory cell should perform one of two operations:

   - **Operation set.** The argument is a list with exactly one element. The memory cell must return a new memory cell.
   - **Operation get.** The argument is an empty list. The memory cell returns the value contained in the memory cell.

   The following two definitions can be used to interact with any memory cell:

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

   (a) *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.

   (b) *Implement a read-only cell.* Function `ro-cell` takes a number and return a read-only cell. Operation *set* should not change the stored value and return a cell with the same contained value. Operation *get* returns the initial value.

2. 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`.

3. 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) Implement function `find`.

   (b) 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`.

   (c) 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`.

4. 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.*

5. 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. Your function should only handle functions declarations, definitions, variables, and numbers. Do **not** handle conditionals nor handle booleans.