Homework Assignment 2

Note 1: 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. If a solution uses a disallowed function, the autograder score is voided.
Note 2: only upload file `hw2.rkt`, as dependencies are available in server.

1. Your goal is to implement a counter encoded with "functions as data." Read §2.1.3 of the SICP book, in particular the implementation of functions `cons`, `car`, and `cdr`.

   (a) *Implement the constructor of counter.* A *counter* is a function-value that expects exactly one argument, a symbol. The internal state of a counter is: an accumulated value called `accum`, and a function called `grow`, which is used to increment `accum`. Depending on the symbol, the counter should return one of two values.

   - When symbol is `'inc` then: Return a new counter whose accumulated value results from applying `grow` to the accumulated value `accum`.
   - When symbol is `'get` **then:** Return the accumulated value `accum`.
   - Otherwise, return `(void)`.

   The constructor of a counter: Function `(counter accum grow)` takes the initial accumulated value `accum` and the function `grow`. The result is a counter that behaves as explained above. The function should **not** assume the contents of the counter to be numeric.

   (b) *Implement the constructor of an adder.* An *adder* is a function-value that expects exactly one argument, a symbol. The internal state of an adder is a counter called `super`.

   - When symbol is `'inc` then: Return a new *adder* with an internal state that results from incrementing `super` twice.
   - When symbol is `'get` then: Return the accumulated value of `super`.
   - Otherwise, return `(void)`.

   The constructor of an adder: Function `(adder super)` takes a counter `super` and returns a new adder. The function should **not** assume the contents of the counter to be numeric.

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 7. Implement function `parse-ast` that takes a datum and yields an element of the AST. You will have access to auxiliary functions `real?` and `symbol?` from Racket's standard library and functions `lambda?`, `define-basic?`, and `define-func?` from *your* file `hw1.rkt` 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.

   Note: you can request a partial solution of `hw1.rkt` if you forfeit the ability to resubmit Homework Assignment 1.