Homework Assignment 3

Any automatically graded answer may be manually graded by the instructor. Submissions are expected to only use functions taught in the course. If a submission uses a disallowed function, that exercise can get zero points. Excluding promises, *all functions that mutate values are disallowed* (mutable functions usually have a `!` in their name).

Promises

1. (30 points) Let a *promise list* be a promise that contains either `empty`, or a pair whose left element is the head of the promise list, and whose right element is the tail of the promise list, which is therefore a promise list. The goal of this exercise is to develop a library for manipulating promise lists. Note that function `(promise? p)` returns `#t` if, and only if value `p` is a promise, otherwise it returns `#f`.

   (a) (2 points) Define variable `p:empty` that is bound to the empty promise list.

   ```
   (check-equal? empty (force p:empty))
   ```

   (b) (3 points) Implement function `(p:empty? l)` that returns `#t` if, and only if, variable `l` is a promise to a list. **Note that each promise is its unique object, so comparison always *fails*.** For instance, `(equal? (delay 1) (delay 1))` evaluates to `#f`. Thus, simply `l` against promise `p:empty` is incorrect.

   ```
   (check-true (p:empty? p:empty))
   (check-false (p:empty? 10))
   ```

   (c) (10 points) **Manually graded.** Explain if it is possible to implement a function `(p:cons x l)` that constructs a new promise list such that `x` is the head of the resulting promise list, `l` is the tail of the promise list, and `x` is *not* evaluated. If you answered that it is possible, then implement `(p:cons x l)` and write a test-case that illustrates its usage. If you answered that it is impossible, then explain how to encode such a function. *Your answer must be written as a comment in the solution file that you submit.*

   (d) (2.5 points) Implement function `(p:first l)` that obtains the head of a promise list.

   ```
   (check-equal? (p:first (delay (cons 1 p:empty))) 1)
   ```

   (e) (2.5 points) Implement function `(p:rest l)` takes a promise list and returns the tail of that promise list.

   ```
   (check-equal? (p:rest (delay (cons 1 p:empty))) p:empty)
   ```

   (f) (10 points) Implement function `(p:append l r)` that concatenates two promise lists `l` and `r`. Recall the implementation of `append` in Lecture 6. Feel free to use the non-tail recursive version.

2. (20 points) Recall the Binary Search Tree (BST) we implemented in Exercise 3 of Homework Assignment 1. Function `bst->list` flattens a BST and yields a sorted list of the members of the BST.

   ```
   (define (bst->list self)
     (cond [(empty? self) self]
           [else
            (append
              (bst->list (tree-left self))
              (cons (tree-value self)
                    (bst->list (tree-right self))))]))
   ```

   (a) (10 points) Implement function `(bst->p:list l)` that returns an ordered promise list of the contents of `t`, by following the implementation of function `bst->list`.

(b) (10 points) **Manually graded.** Give an example of a situation in which lazy evaluation outperforms eager evaluation. Use a function that manipulates promise lists to showcase your argument, *e.g.*, functions `(p:append x y)` or `(bst->p:list l)`. *Your answer must be written as a comment in the solution file that you submit.*

Infinite Streams

3. (10 points) Implement the notion of accumulator for infinite streams.[1] Given a stream `s` defined as

   ```
   e0   e1   e2  ...
   ```

   Function `(stream-foldl f a s)`

   ```
     a   (f e0 a)   (f e1 (f e0 a))   (f e2 (f e1 (f e0 a)))   ...
   ```

   ```
   (define s (stream-foldl cons empty (naturals)))
   (check-equal? (stream-get s) empty)
   (check-equal? (stream-get (stream-next s)) (list 0))
   (check-equal? (stream-get (stream-next (stream-next s))) (list 1 0))
   (check-equal? (stream-get (stream-next (stream-next (stream-next s)))) (list 2 1 0)))
   ```

4. (10 points) Implement a function that advances an infinite stream a given number of steps. Given a stream `s` defined as

   ```
   e0   e1   e2   e3   e4   e5   ...
   ```

   Function `(stream-skip 3 s)`

   ```
   e3   e4   e5  ...
   ```

   ```
   (define s (stream-skip 10 (naturals)))
   (check-equal? (stream-get s) 10)
   (check-equal? (stream-get (stream-next s)) 11)
   (check-equal? (stream-get (stream-next (stream-next s))) 12)
   ```

Evaluating expressions

5. (30 points) Extend functions `r:eval-exp` with support for booleans.

   (a) (2 points) Implement a data structure `r:bool` (using a `struct`) with a single field called `value` that holds a boolean. Recall Lecture 5.

   (b) (3 points) Extend the evaluation function to support boolean values.

   ```
   (check-equal? (r:eval-exp (r:bool #t)) #t)
   (check-equal? (r:eval-exp (r:bool #f)) #f)
   ```

   (c) (10 points) Extend the evaluation to support binary-operation `and`. The semantics of `and` **must** match Racket's operator `and`. Recall that Racket's `and` is not a variable to a function, but a special construct, so its usage differs from function `+`, for instance.

   ```
   (check-true (r:eval-exp (r:apply (r:variable 'and) (list (r:bool #t) (r:bool #t)))))
   (check-false (r:eval-exp (r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f)))))
   ```

[1] Recall that `foldl` is the accumulator for lists and was taught in Lecture 6.

(d) (10 points) Extend function + to support multiple-arguments (including zero arguments).

```
(check-equal?
  (r:eval-exp
    (r:apply (r:variable '+)
      (list (r:number 1) (r:number 2) (r:number 3))))
  6)
```

(e) (5 points) Extend primitive `and` to support multiple-arguments (including zero arguments).

```
(check-equal?
  (r:eval-exp
    (r:apply (r:variable 'and)
      (list (r:bool #t) (r:number 2) (r:number 3))))
  3)
```