CS370 Programming Languages (Zaring)
Spring 2020
Assignment 3
Due by the beginning of lecture on Thursday, March 5

Description:
Write the Scheme procedures described below. Unless told otherwise, you may use only the primitive procedures and expressions used in lecture, the example procedures defined in lecture, and the primitive procedures and expressions used and defined in chapters 1-8 of *TLS*. If you're uncertain about using some procedure (e.g., one you discovered via the DrRacket help system), ask if it's okay to use it: contrary to the usual belief, in this course, it's better to ask for permission before the fact than to ask for forgiveness after the fact. Unless told otherwise in a particular problem, you may define and use auxiliary/helper procedures in your solutions.

When writing operators (i.e., a procedure that returns a funval),

(1) Avoid having the funval returned by the operator depend on any top-level user-defined procedures
(2) Think hard before using `letrec` (although you may find it's necessary in some cases)

For each procedure you write, include a header comment that (at the very least)

- States the purpose of the procedure
- Gives any/all pre-conditions for the procedure (i.e., a description of any special properties the actual parameters must have in order for the procedure to work correctly)
- Gives a big-O statement of the procedure's worst-case asymptotic runtime

- `(curried-binary` *binaryProc*`)`
  Assuming *binaryProc* is a binary procedure, returns a unary procedure that when applied to a value *x* returns a unary procedure that when applied to a value *y* returns the value of *binaryProc* applied to *x* and *y*. For example,

```
(((curried-binary cons) 'a) '(b)) returns (a b)
(((curried-binary +) 10) 2) returns 12
(((curried-binary >) 1) 2) returns #f
```

- `(uncurried-binary` *curriedBinaryProc*`)`
  Assuming *curriedBinaryProc* is a binary procedure that has been curried (perhaps one that was "curried" by `curried-binary`), returns a binary procedure that when applied to a value *x* and a value *y* returns the value of the original uncurried version of *curriedBinaryProc* applied to *x* and *y*. For example,

```
((uncurried-binary (curry-binary cons)) 'a '(b)) returns (a b)
((uncurried-binary (curry-binary +)) 10 2) returns 12
((uncurried-binary (curry-binary >)) 1 2) returns #f
```

- `(adjacent-related-grouped` *related? lis*`)`

  (Reminiscent of the procedure `adjacent-equals-grouped` from Assignment 1.) Assume *lis* is a list and *related?* is a binary predicate (which should be assumed to be O(1)). Returns the list in which all runs of two or more adjacent elements of *lis* for which *related?* returns a non-`#f` value have been grouped into lists. For example,

```
(adjacent-related-grouped eq? '()) returns ()
(adjacent-related-grouped eq? '(a)) returns ((a))
(adjacent-related-grouped eq? '(a b)) returns ((a) (b))
(adjacent-related-grouped eq? '(a b c)) returns ((a) (b) (c))
(adjacent-related-grouped eq? '(a a b c)) returns ((a a) (b) (c))
(adjacent-related-grouped eq? '(a b a a b b c a a a b b)) returns
   ((a) (b) (a a) (b b) (c) (a a a) (b b))
(adjacent-related-grouped eq? '(a b a a b b c a a a b b b c c c d)) returns
   ((a) (b) (a a) (b b) (c) (a a a) (b b b) (c c c) (d))
(adjacent-related-grouped (lambda (x y) (not (eq? x y)))
   '(a b a a b b c a a a b b b c c c d)) returns
   ((a b a) (a b) (b c a) (a) (a b) (b) (b c) (c) (c d))
(adjacent-related-grouped (lambda (x y) #f) '(1 1 2 1 2 3 1 2 3 4)) returns
   ((1) (1) (2) (1) (2) (3) (1) (2) (3) (4))
(adjacent-related-grouped (lambda (x y) #t) '(1 1 2 1 2 3 1 2 3 4)) returns
   ((1 1 2 1 2 3 1 2 3 4))
(adjacent-related-grouped < '(1 1 2 1 2 3 1 2 3 4)) returns
   ((1) (1 2) (1 2 3) (1 2 3 4))
(adjacent-related-grouped <= '(1 1 2 1 2 3 1 2 3 4)) returns
   ((1 1 2) (1 2 3) (1 2 3 4))
(adjacent-related-grouped
 (lambda (x y) (<= (car x) (car y)))
 '((1 a) (1 b) (2 c) (1 d) (2 e) (3 f) (1 g) (2 h) (3 i) (4 j)))
   returns (((1 a) (1 b) (2 c)) ((1 d) (2 e) (3 f))
   ((1 g) (2 h) (3 i) (4 j)))
```

NOTE: For the following procedure, you may not define any top-level or local auxiliary procedures as part of your answer nor may the returned procedure *p* (described below) call any top-level or local user-defined auxiliary procedures (other than, if applicable, *binaryProc*).

- `(rreducer` *binaryProc unaryProc zeroaryProc*`)`

  Assume *binaryProc* is a binary procedure, *unaryProc* is a unary procedure, and *zeroaryProc* is a *thunk* (a procedure of zero parameters). Returns a unary procedure *p* that "reduces" a list using *binaryProc*, treating *binaryProc* as if it's right-associative. That is, applying *p* to a list $(x_1\ x_2\ x_3\ x_4\ x_5)$ would return a value equivalent to the application

  (*binaryProc* x_1 (*binaryProc* x_2 (*binaryProc* x_3 (*binaryProc* x_4 x_5))))

  applying *p* to the list (x_1) would return a value equivalent to the application

  (*unaryProc* x_1)

  and applying *p* to the list `()` would return a value equivalent to the application

  (*zeroaryProc*)

For example,

```
((rreducer - (lambda (x) x) (lambda () 0)) '()) returns 0
((rreducer - (lambda (x) x) (lambda () 0)) '(10)) returns 10
((rreducer - (lambda (x) x) (lambda () 0)) '(10 20 30 40)) returns -20
((rreducer cons (lambda (x) x) (lambda () '())) '()) returns ()
((rreducer cons (lambda (x) x) (lambda () '())) '(a)) returns a
((rreducer cons (lambda (x) x) (lambda () '())) '(a b c d)) returns
   (a b c . d)
```

NOTE: For the following procedure, you may not define any top-level or local auxiliary procedures as part of your answer nor may the returned procedure *p* (described below) call any top-level or local user-defined auxiliary procedures (other than, if applicable, *binaryProc*).

- `(lreducer` *binaryProc unaryProc zeroaryProc*`)`
  Like `rreducer`, but instead returns a unary procedure *p* that "reduces" a list using *binaryProc*, treating *binaryProc* as if it's <u>left</u>-associative. For example,

```
((lreducer - (lambda (x) x) (lambda () 0)) '()) returns 0
((lreducer - (lambda (x) x) (lambda () 0)) '(10)) returns 10
((lreducer - (lambda (x) x) (lambda () 0)) '(10 20 30 40)) returns -80
((lreducer cons (lambda (x) x) (lambda () '())) '()) returns ()
((lreducer cons (lambda (x) x) (lambda () '())) '(a)) returns a
((lreducer cons (lambda (x) x) (lambda () '())) '(a b c d)) returns
   (((a . b) . c) . d)
```

- `(subst-every-other-sf` *old new los+ succeed*`)`
  Works like `subst-every-other` from Assignment 2, but is written using the success-fail style; however, since failure can't really occur in this case, no failure thunk is needed. `subst-every-other-sf` succeeds with two values: a list like the arbitrarily-complex list of symbols *los+*, but with every other occurrence (as read from left to right, starting with the leftmost occurrence) of the symbol *old* replaced by an occurrence of the symbol *new* and a Boolean value that's `#f` if the last occurrence of *old* wasn't replaced by *new* but `#t` if the last occurrence of *old* was replaced by *new*. Since *succeed* succeeds with two values, all the success procedures will be binary procedures. For example,

```
(subst-every-other-sf 'a 'b '() (lambda (result replaced) result)) returns ()
(subst-every-other-sf 'a 'b '() (lambda (result replaced) replaced)) returns
   #f
(subst-every-other-sf 'a 'b '(a) (lambda (result replaced) result)) returns
   (b)
(subst-every-other-sf 'a 'b '(a) (lambda (result replaced) replaced)) returns
   #t
(subst-every-other-sf 'a 'b '(a a) (lambda (result replaced) result)) returns
   (b a)
(subst-every-other-sf 'a 'b '(a a) (lambda (result replaced) replaced))
   returns #f
(subst-every-other-sf 'a 'b '(a a a a a a) (lambda (result replaced)
   result)) returns (b a b a b a)
(subst-every-other-sf 'a 'b '(a a a a a a) (lambda (result replaced)
   replaced)) returns #f
```

`(subst-every-other-sf 'a 'b '(a a a a a a a) (lambda (result replaced) result))` returns `(b a b a b a b)`
`(subst-every-other-sf 'a 'b '(a a a a a a a) (lambda (result replaced) replaced))` returns `#t`
`(subst-every-other-sf 'a 'b '(a (a (a a) a) a) (lambda (result replaced) result))` returns `(b (a (b a) b) a)`
`(subst-every-other-sf 'a 'b '(a (a (a a) a) a) (lambda (result replaced) replaced))` returns `#f`
`(subst-every-other-sf 'a 'b '(a (a (a (a (a (a ()))))))) (lambda (result replaced) result))` returns `(b (a (b (a (b (a ())))))))`
`(subst-every-other-sf 'a 'b '(a (a (a (a (a (a ()))))))) (lambda (result replaced) replaced))` returns `#f`
`(subst-every-other-sf 'a 'x '(a (b (a (c (a (d ()))))))) (lambda (result replaced) result))` returns `(x (b (a (c (x (d ())))))))`
`(subst-every-other-sf 'a 'x '(a (b (a (c (a (d ()))))))) (lambda (result replaced) replaced))` returns `#t`

Strategy:
This is a first exercise in writing Scheme higher-order procedures. Use the techniques for designing higher-order procedures exemplified in *TLS* and in lecture.

You'll be graded on program correctness, style (including choice of identifiers/symbols), and documentation (including the required header comment described at the beginning of this handout), just as you have been in your earlier computer science courses.

Avoid major big-O inefficiencies where it's possible to do so without seriously obfuscating your code. Some of the big-O reckoning might take a bit of thought. Big-O bounds should be stated in terms of the properties of the parameters (e.g., the length of a list) rather than in terms of undefined variables (e.g., *n*). Assume that `car`, `cdr`, `cons`, `eq?`, `atom?`, and `null?` work in time O(1).

Bundle up all your procedure definitions into a single file named `assign03.rkt`. If you use any of the example procedures from lecture (e.g., `rac`), include those definitions at the very end of `assign03.rkt`. If you don't finish a problem, and the associated procedure definitions aren't syntactically correct, please comment out those incomplete definitions and place a note at the top of your file indicating which definitions have been commented out.

What to Hand in:
- A printed listing of `assign03.rkt`. Format your listing (using landscape orientation, smaller fonts, etc.) to avoid illegible line-wrapping in your listing.
- Your file `assign03.rkt` submitted using the `Assignment 3` item on the `Assignments` page of the CS370 Katie course