CS370 Programming Languages (Zaring)
Spring 2020
Assignment 2
Due by the beginning of lecture on Thursday, February 27

Description:

Write the Scheme procedures described below, along with any auxiliary procedures you feel you need to write. 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 1-7 of *TLS* (you won't need any higher-order procedures yet). 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.

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 asymptotic runtime (i.e., a statement of how many steps the procedure takes to produce its answer, stated in terms of proportionality to the size(s) of the actual parameter(s))

- `(same-structure` *x y*`)`

  Returns `#t` iff *x* and *y* (arbitrary Scheme values) have the same structure at all levels and returns `#f` otherwise. *x* and *y* have the same structure iff wherever *x* contains an atom, *y* also contains an atom (any old atom: it need not be the same atom that appeared in *x*), and wherever *x* contains a list, *y* also contains a list. For example,

```
(same-structure? 'a 'b) returns #t
(same-structure? 'a '()) returns #f
(same-structure? '() '()) returns #t
(same-structure? '(a) '()) returns #f
(same-structure? '(a) '(123)) returns #t
(same-structure? '(a b) '(c)) returns #f
(same-structure? '(a (b (c))) '(1 (2 (3)))) returns #t
(same-structure? '(a (b (c))) '(((1) 2) 3)) returns #f
```

- `(elide-length` *lis+ n*`)`

  lis+ is an arbitrarily-complicated list, and *n* is a non-negative integer. `elide-length` returns a list like *lis+*, but where any/all sublists of *lis+* contain only the first *n* (a non-negative integer) elements from *lis+*. If *lis+* (or a sublist) contains more than *n* elements, a list containing just the first *n* elements from *lis+* (or that sublist) followed by the symbol `...` (three periods in a row, which constitutes a legal symbol in Scheme) is returned. For example,

```
(elide-length '() 0) returns ()
(elide-length '() 1) returns ()
(elide-length '(a) 0) returns (...)
(elide-length '(a) 1) returns (a)
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 0) returns
   (...)
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 1) returns
   ((a ...) ...)
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 2) returns
   ((a b ...) (g h ...) ...)
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 3) returns
   ((a b c ...) (g h i ...) (l m n ...) ...)
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 4) returns
   ((a b c d ...) (g h i j ...) (l m n o) (p q r))
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 5) returns
   ((a b c d e ...) (g h i j k) (l m n o) (p q r))
(elide-length '((a b c d e f) (g h i j k) (l m n o) (p q r)) 6) returns
   ((a b c d e f) (g h i j k) (l m n o) (p q r))
```

- `(elide-depth` *lis+ n*`)`

  lis+ is an arbitrarily-complicated list, and *n* is a non-negative integer. Does for depth what `elide-length` does for length. That is, `elide-depth` returns a list in which one can't see inside any lists occurring at or below depth *n*, those elements having been replaced by the symbol `&` (which constitutes a legal symbol in Scheme). For example,

```
(elide-depth '() 0) returns &
(elide-depth '() 1) returns ()
(elide-depth '() 2) returns ()
(elide-depth '(a) 0) returns &
(elide-depth '(a) 1) returns (a)
(elide-depth '(a) 2) returns (a)
(elide-depth '(() (a) ((a)) (((a)))) 0) returns &
(elide-depth '(() (a) ((a)) (((a)))) 1) returns (& & & &)
(elide-depth '(() (a) ((a)) (((a)))) 2) returns (() (a) (&) (&))
(elide-depth '(() (a) ((a)) (((a)))) 3) returns (() (a) ((a)) ((&)))
(elide-depth '(() (a) ((a)) (((a)))) 4) returns (() (a) ((a)) (((a))))
```

- `(perms` *loa*`)`

  loa is a list of atoms. `perms` returns the list (in any order) of all possible permutations of the atoms in *loa*. For example,

```
(perms '()) returns (())
(perms '(a)) returns ((a))
(perms '(a b)) returns ((a b) (b a))
(perms '(a b c)) returns ((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))
```

- `(subst-every-other` *old new los+*`)`
  Returns 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*, irrespective of the (sub)list containing *old*. For example

```
(subst-every-other 'a 'b '()) returns ()
(subst-every-other 'a 'b '(a)) returns (b)
(subst-every-other 'a 'b '(a a)) returns (b a)
(subst-every-other 'a 'b '(a a a a a a)) returns (b a b a b a)
(subst-every-other 'a 'b '(a (a (a a) a) a)) returns (b (a (b a) b) a)
(subst-every-other 'a 'b '(a (a (a (a (a (a ()))))))) returns
   (b (a (b (a (b (a ()))))))
(subst-every-other 'a 'x '(a (b (a (c (a (d ()))))))) returns
   (x (b (a (c (x (d ()))))))
```

Strategy:
This is a second exercise in writing Scheme procedures. Use the techniques for designing *-recursive procedures exemplified in *TLS* and in lecture. You may also find some of the binding-forms presented in lecture helpful.

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 `assign02.rkt`. If you use any of the example procedures from lecture (e.g., `rac`), include those definitions at the very end of `assign02.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 `assign02.rkt`. Format your listing (using landscape orientation, smaller fonts, etc.) to avoid illegible line-wrapping in your listing.
- Your file `assign02.rkt` submitted using the `Assignment 2` item on the `Assignments` page of the CS370 Katie course