CS370 Programming Languages (Zaring)
Spring 2020
Assignment 1
Due by noon on Friday, February 21

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 1-3 of *TLS* (you won't need any numbers, star-recursion, or 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))

Numeric operations are not necessary, desirable, or permitted when writing any of the following procedures or any auxiliary/helper procedures you write.

- `(zip` *lisa lisb*`)`
  Assume that *lisa* and *lisb* are lists of the same length or that *lisa* is exactly one element longer than *lisb* (and the procedure deals with either situation). Returns the list that results from "zipping" together *lisa* and *lisb*. For example

  `(zip '() '())` returns `()`
  `(zip '(a) '())` returns `(a)`
  `(zip '(a) '(b))` returns `(a b)`
  `(zip '(a b c) '(d e f))` returns `(a d b e c f)`
  `(zip '(a b c d) '(e f g))` returns `(a e b f c g d)`

- `(unzip` *lis*`)`
  Assume that *lis* is a list. Returns the pair of lists that results from "unzipping" *lis* (placing the last element of *lis* into the first element of the result, if *lis* has an odd number of elements). For example,

  `(unzip '())` returns `(() ())`
  `(unzip '(a))` returns `((a) ())`
  `(unzip '(a b))` returns `((a) (b))`
  `(unzip '(a b c d e f))` returns `((a c e) (b d f))`
  `(unzip '(a b c d e f g))` returns `((a c e g) (b d f))`

- `(all-tails` *lis*`)`
  Assume that *lis* is a list. Returns the list of all tails of *lis*, in order from longest to shortest. For example,

  `(all-tails '())` returns `(())`
  `(all-tails '(a))` returns `((a) ())`
  `(all-tails '(a b))` returns `((a b) (b) ())`
  `(all-tails '(a b c))` returns `((a b c) (b c) (c) ())`

- `(all-heads` *lis*`)`
  Assume that *lis* is a list. Returns the list of all heads of *lis*, in order from shortest to longest. For example,

  `(all-heads '())` returns `(())`
  `(all-heads '(a))` returns `(() (a))`
  `(all-heads '(a b))` returns `(() (a) (a b))`
  `(all-heads '(a b c))` returns `(() (a) (a b) (a b c))`

- `(without-adjacent-duplicates` *los*`)`
  Assume *los* is a list of symbols. Returns the list in which all runs of two or more adjacent duplicate symbols have been eliminated and replaced by a single occurrence of that symbol. For example,

  `(without-adjacent-duplicates '())` returns `()`
  `(without-adjacent-duplicates '(a))` returns `(a)`
  `(without-adjacent-duplicates '(a b))` returns `(a b)`
  `(without-adjacent-duplicates '(a a b))` returns `(a b)`
  `(without-adjacent-duplicates '(a a b b b))` returns `(a b)`
  `(without-adjacent-duplicates '(a a b b b a))` returns `(a b a)`
  `(without-adjacent-duplicates '(a b a a c c a a a d d d))` returns
  `(a b a c a d)`

`(adjacent-equals-grouped` *los*`)`
Assume *los* is a list of symbols. Returns the list in which all runs of adjacent equal symbols have been grouped into lists. For example,

`(adjacent-equals-grouped '())` returns `()`
`(adjacent-equals-grouped '(a))` returns `((a))`
`(adjacent-equals-grouped '(a b))` returns `((a) (b))`
`(adjacent-equals-grouped '(a b c))` returns `((a) (b) (c))`
`(adjacent-equals-grouped '(a a b c))` returns `((a a) (b) (c))`
`(adjacent-equals-grouped '(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-equals-grouped '(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))`

Strategy:
This is a first exercise in writing Scheme procedures. Use the techniques for designing recursive procedures exemplified in *TLS*.

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 `assign01.rkt`. If you use any of the example procedures from lecture (e.g., `rac`), include those definitions at the very end of `assign01.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.

Finding Errors:

You'll find the following techniques helpful for finding errors/debugging:

Using the `trace` library

After the `#lang Scheme` line at the top of your program (in the definitions pane), insert the line

```
(require racket/trace)
```

Then, at the end of your program (in the definitions pane), insert the line

(`trace` sym_1 sym_2 ... sym_n)

where sym_1, sym_2, ... , sym_n are the names of the procedures you wish to "trace". Having done this, any/all calls to any of the traced procedures will display what the values of the formal parameters are for that call, and any/all returns from any of the traced procedures will display the value that's being returned.

Using the DrRacket Debugger

Type a sample call to the procedure you wish to test at the end of the definitions pane (<u>not</u> in the execution pane, where you'd usually type it). Then, click on the `Debug` button near the upper-right corner of the DrRacket window. You'll then be able single-step through your code, just as you would in a Python, Java, etc. debugger, using the `Pause`, `Go`, `Step`, `Over`, and `Out` buttons, seeing the vales returned at each step of the evaluation of your procedures. Click on the `Stop` button to return to normal operation.

Search for "Graphical Debugging Interface" in the DrRacket documentation for more details.

What to Hand in:

- A printed listing of `assign01.rkt`. Format your listing (using landscape orientation, smaller fonts, etc.) to avoid illegible line-wrapping in your listing.
- Your file `assign01.rkt` submitted using the `Assignment 1` item on the `Assignments` page of the CS370 Katie course