CS370 Programming Languages (Zaring)
Spring 2020
Exam 1
Due by noon on Friday, March 20; NO LATE EXAMS ACCEPTED

By taking Exam 1, you are agreeing to abide by the terms given below and, as always, to the terms of the Luther College Honor Code. Be sure to read this set of instructions completely and carefully.

This is a take-home exam. You may use your own CS370 course notes, *TLS*, your own graded assignments, any materials I've sent you (including sample solutions, examples/slides from lectures, etc.), and DrRacket. You may use nothing else whatsoever: no other books, no web sites of any kind, no other students' notes, no materials from any past courses, no other applications, etc.

You may not communicate with anyone other than the instructor about any aspect of the exam, including, but not limited to, asking/telling which problems someone is working on or has worked on, asking/telling how many questions someone has left to answer, asking/telling what section(s) of the textbook have been useful, etc. You're completely on your own.

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

- The purpose of the procedure
- 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)
- A big-O statement of the procedure's worst-case asymptotic runtime

Hand in

- This exam paper, with your name written at the top
- Any/all paper containing written answers (if any)
- Any/all files you produce for your answers, submitted using the `Exam 1` item on the `Exams` page of the CS370 Katie course.

Unless specifically told otherwise in Scheme programming questions,

- You need not comment your code (other than to provide the aforementioned header comments)
- You should assume that users always provide actual parameter values that meet the stated specifications for the problem (i.e., users provide lists where lists are required, numbers where numbers are required, positive numbers where positive numbers are required, etc.)
- You may write auxiliary procedures when asked to "write procedure X so that ..."
- You should use all good programming practices (the most appropriate sorts of expressions, etc.), good style (mnemonic names, etc.), and appropriate programming techniques (higher-order procedures, the success-fail style, etc.)
- You may not use imperative features of Scheme other than `define`
- You may not require/import any modules/libraries aside from the usual `tls.ss` module.
- You should prefer algorithms that avoid needless expense with regard to big-O bounds

There are no obscure, built-in Scheme features that will magically solve a question for you, so don't waste time scouring HelpDesk looking for such things. All the questions can be answered with material seen and used many times in this course.

Question 1 (40 points)

Write the procedure `(deepen-right` *lis*`)` where *lis* is a list. `deepen-right` returns a list containing the same elements as *lis*, but nested deeper and deeper as you go from right to left. For example,

`(deepen-right '())` returns `()`
`(deepen-right '(a))` returns `(a)`
`(deepen-right '(a b))` returns `(a (b))`
`(deepen-right '(a b c))` returns `(a (b (c)))`
`(deepen-right '(a b c d))` returns `(a (b (c (d))))`

Save your code in the file `q1.rkt` and submit that file using the `Exam 1` item on the CS370 Katie course.

Question 2 (45 points)

Write the procedure `(without-nulls` *lis*+`)` that removes any and all null lists from *lis*+ (an arbitrarily-complex list), ultimately leaving the result with no null lists whatsoever, noting that the removal of more-nested nulls can create less-nested nulls. For example:

`(without-nulls '())` returns `()`
`(without-nulls '(a b c d))` returns `(a b c d)`
`(without-nulls '(a b () c () d ()))` returns `(a b c d)`
`(without-nulls '((a b ()) (c () d ())))` returns `((a b) (c d))`
`(without-nulls '((a b ()) (()) (c (() ()) d ())))` returns `((a b) (c d))`
`(without-nulls '(() (() ()) ((() ()) ()) (() (() ((((((()))))))))))` returns `()`

Save your code in the file `q2.rkt` and submit that file using the `Exam 1` item on the CS370 Katie course.

Question 3 (45 points)

Write the procedure `(fringe-positions` *target los*+`)` that returns a list of the position numbers with respect to a zero-origin, left-to-right numbering of all the symbols in *los*+ (an arbitrarily-complex list of symbols) at which the *target* (a symbol) occurs. For example,

`(fringe-positions 'x '())` returns `()`
`(fringe-positions 'x '(a b c d))` returns `()`
`(fringe-positions 'x '(a x b x c x d))` returns `(1 3 5)`
`(fringe-positions 'x '(a (x (b (x) c) x) d))` returns `(1 3 5)`
`(fringe-positions 'x '(((((((a) x) b) x) c) x) d))` returns `(1 3 5)`
`(fringe-positions 'x '(((((((x) x) x) x) x) x) x))` returns `(0 1 2 3 4 5 6)`

Suggestion: Consider using the success-fail style with success procedures that have more than one formal.

Save your code in the file `q3.rkt` and submit that file using the `Exam 1` item on the CS370 Katie course.

Question 4 (40 points)

Write the operator `(searcher` *target*`)` that returns a unary predicate that takes an arbitrarily-complex list of symbols and returns `#t` if *target* (a symbol) occurs anywhere inside that list and returns `#f` otherwise. For example,

`((searcher 'x) '())` returns `#f`
`((searcher 'x) '(a b c d))` returns `#f`
`((searcher 'x) '(a b x d))` returns `#t`
`((searcher 'x) '((() () a) () (b (x)) ((d))))` returns `#t`
`((searcher 'y) '((() () a) () (b (x)) ((d))))` returns `#f`

Special restrictions for this question:

- The predicate returned by `searcher` can't depend on any top-level user-defined procedures other than, perhaps, `searcher`.
- You may not use `letrec` or the Y combinator to create local auxiliary procedures.

Save your code in the file `q4.rkt` and submit that file using the `Exam 1` item on the CS370 Katie course.

The following question is surprisingly subtle and tricky. You should probably save it for last.

Question 5 (30 points)

Write the procedure `(convolution` *lisa lisb* `.` *etc*`)` where *lisa* and *lisb* are lists of the same length, and *etc* is a list of any additional actual parameters you may wish to supply. `convolution` returns the *convolution* of the elements of *lisa* with the elements of *lisb*; that is, if *lisa* is $(x_1\ x_2\ \ldots\ x_n)$ and *lisb* is $(y_1\ y_2\ \ldots\ y_n)$, `convolution` returns the list $((x_1\ y_n)\ \ (x_2\ y_{n-1})\ \ \ldots\ \ (x_n\ y_1))$. For example,

`(convolution '() '() ...)` should return `()`
`(convolution '(a) '(1) ...)` should return `((a 1))`
`(convolution '(a b) '(1 2) ...)` should return `((a 2) (b 1))`
`(convolution '(a b c d) '(1 2 3 4) ...)` should return `((a 4) (b 3) (c 2) (d 1))`

Special restrictions for this question:

- `convolution` must operate in time O(|*lisa*|+|*lisb*|). Further, to keep the big-O constant of proportionality as low as possible, you may make only a single pass over the items in *lisa* and *lisb*. (For example, you may not `cdr` your way down either *lisa* or *lisb* twice, you make not make a duplicate of either *lisa* or *lisb* and then `cdr` your way down the copy, and so on.)
- The only pre-defined procedures you may use are `car`, `cdr`, `cons`, `eq?`, `atom?`, and `null?`.
- You may not `define` any auxiliary procedures nor use `letrec` or the Y combinator to create any local auxiliary procedures.
- The values of any/all actual parameters supplied via *etc* may not have values that depend on the specific values of *lisa* or *lisb*. (For example, you couldn't supply a third parameter whose value was always the reverse of *lisa* or *lisb*.)

Save your code in the file `q5.rkt` and submit that file using the `Exam 1` item on the CS370 Katie course.