CS 335 - Homework 3

Deadline:

Due by 11:59 pm on Friday, February 18

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `3`

Purpose:

To practice writing recursive functions on lists, to practice writing and using higher-order functions, to practice writing and using lambda expressions

Important notes:

- Use the R5RS version of Scheme in DrRacket for these problems.
- Before each Scheme function, include comment lines that include at least the name of the function, and a purpose statement (clearly describing what it expects and what it returns).
- Put a blank line before each function definition and after each function definition -- use additional blank lines are needed for readability.
- Start each function's body on a different line than the function header, indented for readability.
- Use hard returns rather than letting lines of code wrap to the next line.

The Problems:

Problem 1

Start a Scheme file `335hw3.scm` or `335hw3.ss`. The first thing in this file should be comments containing at least:

- your name
- `CS 335 - Homework 3`
- and the last-modified date

Now follow that with a comment containing `Problem 1`, followed by your answers for this problem.

Define a function `eqlist` that expects a single list and returns `#t` (true) if all of the top level elements of the list are equal to each other, otherwise it returns `#f` (false). It should return `#t` if its argument is the empty list or if it contains only 1 item. It must determine this in a recursive fashion.

You may use either the built-in `length` function or your `len` function from Problem 3. And, note that

Scheme supports the boolean operators `and`, `or`, and `not`.

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing eqlist - #t's mean passed:\n")
(equal? (eqlist '('a 'a 'a 'a))
        #t)
(equal? (eqlist '(3 3 3 4))
        #f)
(equal? (eqlist '())
        #t)
(equal? (eqlist '("howdy"))
        #t)
```

Problem 2

Put a comment containing `Problem 2`, followed by your answers for this problem.

Write a function `rev`, your own version of the built-in reverse function, which expects a list and returns the reverse of the list. Only the order of the top-level elements is changed. Your solution must build this in a recursive fashion.

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing rev - #t's mean passed:\n")
(equal? (rev '())
        '())
(equal? (rev '('a))
        '('a))
(equal? (rev '('a 10 20))
        '(20 10 'a))
(equal? (rev '((1 2) "howdy" ('b (4 5)) 7))
        '(7 ('b (4 5)) "howdy" (1 2)))
```

Problem 3

Put a comment containing `Problem 3`, followed by your answers for this problem.

Write a function `tl-replace` that expects two values and a list and produces a list in which all instances of the first given value in the given list are replaced with the second given value. For example,

```
(tl-replace 7 3 '(1 6 7 8 7 7 2))
```

...would return `(1 6 3 8 3 3 2)`.

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing tl-replace - #t's mean passed:\n")
(equal? (tl-replace 7 3 '(1 6 7 8 7 7 2))
        '(1 6 3 8 3 3 2))
(equal? (tl-replace 7 3 '(2 4 6 8 10))
        '(2 4 6 8 10))
(equal? (tl-replace 7 3 '(7 4 7 (7 6)))
        '(3 4 3 (7 6)))
(equal? (tl-replace 7 3 '())
        '())
```

Problem 4

Put a comment containing `Problem 4`, followed by your answers for this problem.

Define a function `delete` that expects a value and a list, and produces a list in which all instances of the given value are removed from the given list, even if it appears within sub-lists of a list. For example,

`(delete 2 '(5 2 9 2))` would return `(5 9)`

`(delete 2 '(5 (2 9) 2))` would return `(5 (9))`

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "testing delete - #t's mean passed:\n")
(equal? (delete 2 '(5 2 9 2))
        '(5 9))
(equal? (delete 2 '(5 (2 9) 2))
        '(5 (9)))
(equal? (delete 2 '(2 (2 4 (6 2 8)) (10 2) 2 7))
        '((4 (6 8)) (10) 7))
(equal? (delete 2 '())
        '())
(equal? (delete 2 '(1 3 5 7 9))
        '(1 3 5 7 9))
```

Problem 5

Put a comment containing `Problem 5`, followed by your answers for this problem.

5 part a

Recall the in-class example `my-map`, a "home-grown" version of the built-in `map` function, which takes a function as a parameter and applies it to each element of a parameter list, returning a list of the results.

Recall, also, the example list from the last homework:

```
(define cs-phone '((tuttle 3381) (burgess 3331)
                   (burroughs 3337) (dixon 3530)
                   (amoussou 3380)))
```

If you wanted to, say, obtain a list of all of the names from `cs-phone`, you could do so very easily using `my-map` or `map`. Include the above definition of `cs-phone`, and also `my-map` if you would like, and write an expression using `cs-phone` and either `my-map` or `map` whose value is the list of names from `cs-phone`.

5 part b

It is a little trickier to write an expression using `cs-phone` and either `my-map` or `map` whose value is the list of phone numbers from `cs-phone`, but it is still quite reasonable. So, write this expression next.

Problem 6

Put a comment containing `Problem 6`, followed by your answers for this problem.

Now try your hand at writing a simple higher-order function, another classic: the function `filter` expects a function, a target, and a list as its parameters. `filter` then applies that function once for each list item, with first argument of that target and second argument a list item, and it returns a list of **ONLY** those elements for which the function returns `#t` (true). Your solution must be recursive.

(See where the "filter" aspect comes in? Because only those list elements that pass through the "filter" --- return true for this function and target --- are returned.)

That is,

```
(filter < 3 '(1 2 3 4 5 6 7 8))
```

should return:

`(4 5 6 7 8)` ; the values x for which (< 3 x) is true

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "\ntesting filter - #t's mean passed\n")
(equal? (filter < 3 '(1 3 2 5 16 -3 8 0))
        '(5 16 8))
(equal? (filter > 4 '(1 3 2 5 16 -3 8 0))
        '(1 3 2 -3 0))
(equal? (filter = 27 '(1 3 2 5 16 -3 8 0))
```

```
        '())
(equal? (filter = 27 '())
        '())
```

Problem 7

Put a comment containing `Problem 7`, followed by your answers for this problem.

"fold" functions are also popular examples of higher-order functions. A common type of "fold" function would accept a binary function (that is, a function that expects two parameters), a starting value that is an appropriate argument for that binary function, and then a list of appropriate arguments for that binary function, as its three arguments. It then applies that function in turn to the starting value and the 1st element in the list, then to that result and to the 2nd element in the list, then to *that* result and to the 3rd element in the list, and so on, returning the final result.

That is,

```
(fold + 3 '(1 7 10 2 7 5))
```

should return:

```
35          ; the result of (+ 3 1) -> 4, (+ 4 7) -> 11, (+ 11 10) -> 21,
            ;    (+ 21 2) -> 23, (+ 23 7) -> 30, (+ 30 5) -> 35
```

In this, an important assumption is made: that, if called with an empty list, then `fold` should return the starting value. That is,

```
(fold * 8 '())
```

should return:

```
8
```

And lest you think that all binary functions involve numbers:

```
(fold append '(1) '((2 3) (4) (5 6 7 8)))
```

should return:

```
(1 2 3 4 5 6 7 8)
```

7 part a

Write the `fold` function described above.

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "\ntesting fold - #t's mean true\n")
(= (fold +  3 '(1 7 10 2 7 5))
   35)
(= (fold * 8 '())
   8)
```

```
(equal? (fold append '(1) '((2 3) (4) (5 6 7 8)))
        '(1 2 3 4 5 6 7 8))
```

7 part b

Just as a nifty little application of `fold`...

Consider: what do you get if you call `fold` with the arguments +, 0, and a list of numbers?

Write a function `list-avg` that expects a list of numbers, and produces the average of the numbers in that list, using the function `fold`.

Problem 8

Put a comment containing `Problem 8`, followed by your answers for this problem.

Now, for some practice writing and using lambda expressions.

8 part a

Write a lambda expression that expects a number, and produces the product of 13 and that number. Then, write an expression that uses this lambda expression with the argument 10.

8 part b

Write a lambda expression that expects a list, and if its length is three or more, it returns the 3rd element in that list; otherwise, it returns the empty list.

Now, consider a list such as the following, representing the exam scores for the students in a course:

```
(define exam-grades '((alpha 100 70 60) (beta 75 80 90)
                      (gamma 60 100 74) (theta 98 84 79)))
```

Include the above definition of `exam-grades`, and write an expression that uses this lambda expression in a call to `map` or `my-map` to produce a list of the all of the second exam scores for the students in `exam-grades`.

Problem 9

Put a comment containing `Problem 9`, followed by your answers for this problem.

Write a function `generate-multiplier` that accepts a number, and results in a function that that accepts a number and returns that number multiplied by the given multiplier. That is,

```
((generate-multiplier 3) 7)
```

should return:

```
21
```

```
(map (generate-multiplier 10) '(2 4 6 8 10))
```

should return:

```
(20 40 60 80 100)
```

After your function insert the following (feel free to insert additional testing calls if you'd like):

```
(display "\ntesting generate-multiplier - #t's mean passed\n")
(= ((generate-multiplier 3) 7)
    21)
(equal? (map (generate-multiplier 10) '(2 4 6 8 10))
        '(20 40 60 80 100))
```

Problem 10

Put a comment containing `Problem 10`, followed by your answers for this problem.

Write a function `get-area-funct` which expects a symbol representing a shape, and returns an appropriate function for computing that shape's area (for a 2-d shape) or surface area (for a 3-d shape). Your function must meet at least the following requirements:

- it must handle at least the shapes `'circle`, `'sphere`, and `'cylinder`, although it may also handle additional shapes if you would like
- it must use lambda expressions for the functions it returns
- if given a shape it cannot handle, your function can simply return `#f` (false).

Demonstrate your resulting function with at least the following (tweaking it as needed depending on your value for pi!) (Notice that, for real numbers, you sometimes just test to see if your expected value is "close enough" to the actual value...)

```
(display "\ntesting get-area-funct - #t's mean passed\n")
(< (abs (- ((get-area-funct 'circle) 10) 314.159))  0.001)
(< (abs (- ((get-area-funct 'sphere) 10) 1256.636)) 0.001)
(< (abs (- ((get-area-funct 'cylinder) 10 5) 942.477)) 0.001)
```