CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment:	09
Due:	Tuesday, November 26, 2019 9:00 pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	No explicit recursion, see below
Files to submit:	`change-alf.rkt`, `sequences.rkt`, `manipulations.rkt`, `bonus-a09.rkt`
Warmup exercises:	Without using explicit recursion complete HtDP 9.5.2, 9.5.4, and write your own versions of `member?` and `append`.
Practice exercises:	HtDP 21.2.1, 21.2.2, and 21.2.3

For this assignment:

- You may use only the following abstract list functions: `build-list`, `filter`, `map`, `foldr`, and `foldl`.
- You may use any primitive functions, such as mathematical functions, `cons`, `first`, `second`, `third`, `list`, `empty?`, and `equal?`.
- You may use **`cond`**.
- You may use **`lambda`**.
- **You may not use explict recursion for any question.** That is, functions that involve an application of themselves, either directly or via mutual recursion. Use the abstract list functions instead.
- You may **not** use any non-primitive list functions, including `length`, `member?`, `reverse`, `append`, and `list-ref`.
- You may only define non-local constants and helper functions if they will be used to answer multiple parts of the same question. Otherwise, constants and helper functions should be defined locally. Local helper functions do not require purposes, contracts, examples or tests.
- You may use functions defined in earlier parts of a question to help write or test functions later in the same question.

Warning: Question 2b) is quite tricky. If you get stuck on it, complete the rest of the questions and come back to it.

1. (8%) Complete the stepping problems for Module 13a (Lambdas) and Module 13b (Abstract List Functions) at:

   https://www.student.cs.uwaterloo.ca/~cs135/stepping/

2. (24%) Complete Question 3 of Assignment 4 (both parts) following the restrictions described on the first page of this assignment. If you wish, you may cut-and-paste constants, purposes, contracts, tests, and examples for these functions from your Assigment 4 submission. If you wish, you are also permitted to adapt and re-use any parts of the function definitions from your Assignment 4 submission. However, you are encouraged to write additional test cases to ensure thorough testing. Note that the assignment restrictions allow non-local definition of constants, which may be shared by the two parts of this question.

   Hint for part b): Try using `build-list` as part of your solution.

   Place your solution in `change-alf.rkt`.

3. (16%) An *arithmetic sequence* is a sequence of numbers such that the difference between the consecutive terms is constant. For example, the sequence $5, 7, 9, 11, 13, 15, \cdots$ is an arithmetic sequence with consecutive terms differing by 2, because $7-5=9-7=11-9=13-11=15-13=2$.

   A *geometric sequence* is a sequence of numbers where each term after the first is found by multiplying the previous one by a fixed non-zero number called *the common ratio*. For example, the sequence $2, 6, 18, 54, \cdots$ is a geometric sequence with a common ratio of 3 because $2\times 3=6, 6\times 3=18, 18\times 3=54$.

   Either an arithmetic sequence or a geometric sequence can be described by the first two terms of the sequence. Write a function called `sequence` that consumes two numbers (the first two numbers of the potential sequence) and a symbol (either `'arithmetic` or `'geometric`). The function `sequence` should produce a function that consumes a single natural number (the length of the sequence to produce) and produces a list containing the numbers in the sequence. For example:

```
(define f (sequence 5 7 'arithmetic))
(check-expect (f 6) '(5 7 9 11 13 15))
(define g (sequence 2 6 'geometric))
(check-expect (g 4) '(2 6 18 54))
```

   You may assume that the values given will always produce a valid sequence.

   Place your solution in `sequences.rkt`.

4. (32%) The following questions manipulate a single list, a `(listof Any)`, in various ways:

   (a) Write a function `(rotate-right lst)` that moves the element at the end of a list to the front, with all other elements remaining unchanged.

   ```
   (check-expect (rotate-right '(a b c d e f)) '(f a b c d e))
   (check-expect (rotate-right '(a))'(a))
   (check-expect (rotate-right empty) empty)
   ```

   (b) Write a function `(rotate-left lst)` that moves the element at the front of a list to the end, with all other elements remaining unchanged.

   ```
   (check-expect (rotate-left '(a b c d e f)) '(b c d e f a))
   (check-expect (rotate-left '(a))'(a))
   (check-expect (rotate-left empty) empty)
   ```

   (c) Write a function `(prefix n lst)` that produces the first *n* elements of the list. If the length of the list is less than *n*, produces the entire list.

   ```
   (check-expect (prefix 3 '(a b c d e f)) '(a b c))
   (check-expect (prefix 10 '(a b c d e f)) '(a b c d e f))
   (check-expect (prefix 0 '(a b c d e f)) empty)
   (check-expect (prefix 1 '(a b c d e f)) '(a))
   ```

   (d) Write a function `(insert-at position new lst)` that inserts a new element after the specified position in a list, with all preceding elements unchanged, and with all elements after that position shifted to be after the new element. See examples. If the length of the list is less than the specified position, add the new element to the end.

   ```
   (check-expect (insert-at 5 'x '(a b c d e f)) '(a b c d e x f))
   (check-expect (insert-at 0 'x '(a b c d e f)) '(x a b c d e f))
   (check-expect (insert-at 7 'x '(a b c d e f)) '(a b c d e f x))
   (check-expect (insert-at 100 'x '(a b c d e f)) '(a b c d e f x))
   (check-expect (insert-at 0 'x '(a)) '(x a))
   (check-expect (insert-at 1 'x '(a)) '(a x))
   (check-expect (insert-at 100 'x '(a)) '(a x))
   (check-expect (insert-at 0 'x empty) '(x))
   (check-expect (insert-at 1 'x empty) '(x))
   (check-expect (insert-at 100 'x empty) '(x))
   ```

   Place your solution in `manipulations.rkt`.

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

5. **Bonus [5%]**:

   Warning: Part (C) is a serious challenge. You have been warned!

   Place your solution in the file `bonus-a09.rkt`.

   You do not need to include the design recipe for any of these bonus questions. The restrictions on the front of this assignment continue to apply, except as modified by individual parts of the question.

   (a) Write the Racket function `subsets1`, which consumes a list of numbers and produces a list of all of its subsets. For example, `(subsets1 '(1 2))` should produce something like `(list '(1 2) '(1) '(2) '())`. The order of subsets in the list may vary – any complete ordering will be accepted. You can assume the consumed list does not contain any duplicates. Write the function any way you want. (Value: 1%)

   (b) Now write the Racket function `subsets2`, which behaves exactly like `subsets1` but which does not use any explicit recursion or helper functions, as well as following the remaining restrictions on the front of this assignment. Your solution must only be two lines of code, one of which is the function header. Note that if you solve this question, you can also use it as a solution to the previous one — just copy the function and rename the copy `subsets1`. (Value: 1%)

   (c) For the ultimate challenge, write the Racket function `subsets3`. As always, the function produces the list of subsets of a consumed list of numbers. Do not write any helper functions, and do not use any explicit recursion. Do not use any abstract list functions. In fact, use only the following list of Racket functions, constants and special forms: `cons`, `first`, `rest`, `empty?`, `empty`, **`lambda`**, and **`cond`**. You are permitted to use **`define`** exactly once, to define the function itself. (Value: 3%)