Assignment: 9
   Due: Tuesday, April 2nd, 9:00 pm
Language level: Intermediate Student with Lambda
Allowed recursion: Any and all, see below
Files to submit: directed.rkt, coins.rkt, sequences.rkt, subsets.rkt
Warmup exercises: HtDP 28.1.6, 28.2.1, 30.1.1, 31.3.1, 31.3.3, 31.3.6, 32.3.1
Practice exercises: HtDP 28.1.4, 28.2.2, 28.2.3, 28.2.4, 31.3.7, 32.3.2, 32.3.3

• Make sure you read the OFFICIAL A09 post on Piazza for the answers to frequently asked questions.
• Instructions from A08 carry forward.
• You may use all features of Intermediate Student with Lambda that have been covered in the course, unless stated otherwise in the question.
• Purposes and contracts are not required for lambdas or for locally defined helper functions, but purposes and contracts are required for globally defined helper functions. You can use any purposes and contracts we provide, when we provide them. Examples and test cases are not required for globally defined helper functions, but are encouraged.
• Solutions will be marked for both correctness [80%] and good style [20%]. Follow the guidelines in the Style Guide.
• Your solutions must be entirely your own work.
1. [20%] In Module 12 we provide an definition of directed graphs using adjacency lists:

```scheme
;; A Node is a Sym
;; A Graph is a (listof (list Node (listof Node)))
```

We also develop a function `find-route` that finds a route from an origin node to a destination node in a given directed graph, if one exists. In this question, we solve the related problem of determining if a given route provides a valid path from a origin node to a destination node.

Write a predicate `valid-route?` that takes a route and a directed graph as arguments and indicates if that route represents a path though the directed graph. Remember that a route must contain every node between the origin and the destination. Place your solution in `directed.rkt`.

```scheme
(define a-graph
  '((A (C D E)) (B (E J)) (C ()) (D (F J))
    (E (K)) (F (K H)) (H ()) (J (H)) (K ()))))

(check-expect (valid-route? '(A D J H) a-graph) true)
(check-expect (valid-route? '(A B C) a-graph) false)
(check-expect (valid-route? '(A B C) empty) false)
(check-expect (valid-route? empty a-graph) true)
(check-expect (valid-route? empty empty) true)
(check-expect (valid-route? '(X Y Z) a-graph) false)
```

2. [25%]
Implement the following functions. You may not use explicit recursion, and any helper functions must be anonymous (i.e. you must use `lambda`).

(a) Canadian coins come in five denominations. The 'nickel (worth 0.05), the 'dime (worth 0.10), 'quarter (worth 0.25), the 'loonie (worth 1.00), and the 'toonie (worth 2.00).

Write the function `coin-total` that consumes a list of symbols and produces the total value of all coins named in the list. That is, each 'dime symbol in the list contributes 0.10 to the total. Any symbol that is not a Canadian coin (e.g. 'button) is worth 0.

Example:

```scheme
(check-expect (coin-total '(nickel dime dime quarter button)) 0.50)
;; 5 + 10 + 10 + 25 + 0 = 50 cents
```

(b) Write the function `coin-counts` that consumes a list of symbols, and produces an association list where they keys are Canadian coins (symbols), and the values are the number of occurrences of that coin in the consumed list. The produced association list must have a count for each type of coin, even those not found in the consumed list.

Example:
(check-expect (coin-counts '(nickel dime dime quarter button))
  '((nickel 1) (dime 2) (quarter 1) (loonie 0) (toonie 0)))

Note that the keys must be in exactly the same order shown above.

(c) Other countries have different coins. For example, the fictional nation of Bergawfulstan also has 5 coins. The ‘kaf (worth 2), the ‘riwu (worth 4), the ‘teyne (worth 10), the ‘spok (worth 16), and the ‘dekaf (worth 20).

We can represent this using an association list as follows

(define berg-coins '((kaf 2) (riwu 4) (teyne 10) (spok 16) (dekaf 20)))

Note that the coin names are keys, and must be unique.

Write the function foreign-coin-total that consumes an association list like the above, as well as a list of symbols representing coins, and produces the total value of those coins, using the association list as the value of the coins. The function should treat any symbols not in the association list as having no value, just like in part (a).

Example:

(check-expect (foreign-coin-total berg-coins '(spok riwu spok)) 36)

(d) Extending the idea from part (c), write the function make-coin-totaler that consumes an association list of coin values (as in part b) and produces a function. The produced function consumes a list of symbols and produces their total value (according to the association list consumed).

Example:

(check-expect ((make-coin-totaler berg-coins) '(spok riwu spok)) 36)

Place your solutions in coins.rkt.

3. [35%]

Number sequence tests can provide a psychometric assessment of an individual’s numerical reasoning ability. This is useful if you are, for example, assessing a patient’s recovery after a concussion. Not naming names.

For example, in the following sequences, what’s the next number?

• 2, 4, 6, 8, 10, 12, ?
• 1, -2, 3, -4, 5, -6, ?
• 3, 5, 9, 17, 33, 65, ?
• 1, 1, 2, 3, 5, 8, ?
But since this is CS135, we don’t want you to solve such problems yourself. We want you to write a Racket function to solve them.

For each of these problems, there will be a partially known integer sequence

\[ n_0, n_1, n_2, \ldots n_i, \ldots \]

where \( n_i = f(i) \) for some unknown function \( f(i) \). Given the first few numbers in the sequence, typically 5 or 6 of them, we want you to guess the function \( f(i) \). Since there might be more than one function that generates the first few numbers, we will provide some guidance along the way. Place your solutions in \texttt{sequences.rkt}.

(a) Let’s start by checking if a proposed solution, assuming we have one, is actually correct. Write a predicate \texttt{solution?} as follows:

\[
\text{;; (solution? \textit{f} \textit{lon}) checks that \textit{f} generates the values in}
\text{;; the list of integers \textit{lon}, where \textit{f} takes a position in}
\text{;; the list as an argument, starting at 0.}
\text{;; solution?: (Nat -> Int) (listof Int) -> Bool}
\]

\[
\text{(check-expect (solution? \((\lambda (i) (* 2 (+ i 1)))\) \((\text{list 2 4 6 8 10 12})\)) true)}
\]
\[
\text{(check-expect (solution? \((\lambda (i) (\text{list 0 1 2 3 4 5}))\) \((\text{true})\)))
\text{(check-expect (solution? \((\lambda (i) (\text{false})\)) \((\text{false})\))})
\]

We suggest you use \texttt{build-list} to answer this part. It’s not mandatory, but it’s good practice!

(b) Now let’s guess that the solution for a given list might be a quadratic polynomial of the form \( f(i) = ai^2 + bi + c \) and produce a function to generate this solution. For this type of guess, if you’re given a list with exactly three numbers there will be exactly one possible solution. If you’re given a list with more than three numbers, you can ignore all but the first three numbers for the purposes of generating the guess. We will check the guess in part c. If you’re given a list with fewer than three numbers, just generate any of the possible solutions; you won’t be wrong. For example, if you’re given ‘\((1 2)\) then \( f(i) = 0i^2 + 1i + 1 \) will work, as will \( f(i) = 0.5i^2 + 0.5i + 1 \) (in fact there are infinite quadratics that begin with 1, 2). If you’re given an empty list, guess \( f(i) = 0 \). Write a function \texttt{guess-quadratic}, as follows:

\[
\text{;; (guess-quadratic \textit{lon}) guesses that the number sequence \textit{lon} has a}
\text{;; quadratic solution and produces the guess.}
\text{;; guess-quadratic: (listof Int) -> (Nat -> Int)}
\]

\[
\text{(define test-a (list 2 4 6 8 10 12))}
\text{(check-expect (solution? (guess-quadratic test-a) test-a) true)}
\text{(define test-b (list 1 11 33 67 113))}
\text{(check-expect (solution? (guess-quadratic test-b) test-b) true)}
\]
(define test-c (list 2 4 8 16 32 64))
(check-expect (solution? (guess-quadratic test-c) test-c) false)

(c) Using the function solution? from part a) and the function guess-quadratic from part b) as helpers, write a function try-quadratic, which guesses that the solution might be quadratic and if so returns the solution. If not, it should return the empty list, so that we know to try solutions based on other patterns.

;; (try-quadratic lon) tries solving the number sequence lon with ;; a quadratic. It produces the solution if it works and empty ;; if it doesn’t.
;; try-quadratic: (listof Int) -> (anyof (Nat -> Int) empty)

(check-expect ((try-quadratic test-a) 6) 14)
(check-expect ((try-quadratic test-a) 1000000) 2000002)
(check-expect ((try-quadratic test-b) 5) 171)
(check-expect (try-quadratic test-c) empty)

(d) Repeat step b) and c) above for a recursive pattern of the form

\[ f(0) = n_0, \]
\[ f(1) = n_1, \text{ and} \]
\[ f(i) = an_{i-1} + bn_{i-2}, \text{ where } i > 1. \]

Write functions called guess-recursive and try-recursive. Don’t forget to write purposes and contracts for these functions. While our test cases will generally use lists with length greater than 3, you still need to do something reasonable with shorter lists, like you did in part b). Like guess-quadratic, the function guess-recursive should always return a function, which will then be checked by try-recursive. Be very very careful to avoid division by zero.

(define test-d (list 1 1 2 3 5 8 13))
(check-expect ((try-recursive test-d) 7) 21)
(check-expect ((try-recursive test-d) 100) 573147844013817084101)
(define test-e (list 0 1 1 2 3 5 8))
(check-expect ((try-recursive test-e) 7) 13)
(define test-f (list 1 1 3 5 11 21 43))
(check-expect ((try-recursive test-f) 7) 85)
(define test-g (list 2 0 0 0 0 0))
(check-expect ((try-recursive test-g) 0) 2)
(check-expect ((try-recursive test-g) 100) 0)
(check-expect (try-recursive test-b) empty)

(e) Finally, write a function solve that tries both the quadratic and recursive patterns, in that order, and returns a solution, if one exists. Otherwise, return the empty list. Don’t forget to write a purpose and contract for this function.

(check-expect ((solve test-a) 6) 14)
Of course we could try other patterns, but we’ll leave that for now.

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.

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

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

Place your solution in the file `subsets.rkt`.

You do not need to include the design recipe for any of these bonus questions.

(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. You must rely on abstract list functions and `lambda` (and potentially standard list functions like `cons`, `first`, `rest`, `append`, etc.). 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 (i.e., your function cannot call itself by name). 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%)