Assignment Guidelines.

- This assignment covers material in Modules 6 and 7.
- Submission details:
  - Solutions to these questions must be placed in files a7q1.rkt, a7q2.rkt, and a7q3.rkt, respectively, and must be completed using Racket.
  - Unless otherwise indicated in the question you may use only the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment. A list of functions described in each module of the lecture slides can be found at [https://www.student.cs.uwaterloo.ca/~cs115/built_in](https://www.student.cs.uwaterloo.ca/~cs115/built_in).
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the CS115 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page.
  - For the design recipe, helper functions only require a purpose, a contract and an example.
- Restrictions:
  - You should expect to use recursion on every question.
  - Read each question carefully for additional restrictions.

⚠️ Do not use `map`, `foldr`, `filter`, `length`, `append`, or `range` on this assignment.

- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1. **Faro Shuffle.** A faro shuffle involves splitting a deck of cards into two piles, then taking cards exactly alternating between the two piles.

   Write a function \( (\text{faro-shuffle} \; L1 \; L2) \), where each parameter is a \((\text{listof} \; \text{Any})\). The function should return a list which contains all the items in \( L1 \) and \( L2 \), interleaved. That is, the first item in the result is from \( L1 \), the second is from \( L2 \), then it alternates.

   If \( L1 \) and \( L2 \) are not of equal length, the extra items should be included at the end.

   \[(\text{faro-shuffle} \; (\text{list} \; 1 \; 3 \; 5) \; (\text{list} \; 2 \; 4 \; 6 \; 8 \; 10)) \Rightarrow (\text{list} \; 1 \; 2 \; 3 \; 4 \; 5 \; 6 \; 8 \; 10)\]

2. **Add or Subtract.**

   Write a function \( (\text{add-sub-string} \; s) \). The function consumes a \( \text{Str} \) of even length, where every even-numbered character is either "+" or "-", and odd-valued characters are numeric.

   The function should add up all the digits that follow a "+", and subtract all the digits that follow a "-".

   For example,

   \[ (\text{add-sub-string} \; "+3+4-5") \Rightarrow 2 \]

   \[ (\text{add-sub-string} \; "-5+3+4-6-6") \Rightarrow -10 \]

   The interface file contains the function \( \text{char->nat} \) that converts a numeric \( \text{Char} \) to the corresponding numeric value. You may use this function as part of your solution.

   \[ (\text{char->nat} \; #\4) \Rightarrow 4 \]

   \[ (\text{char->nat} \; #\2) \Rightarrow 2 \]
3. Adding Numbers. This question explores how we can represent arbitrarily big numbers, even if the native numbers have a limited range. Instead of using Nat, we will use the following data definition:

```scheme
;; A Digit is a Nat
;; Requires: the value is less than 10.

To represent multi-digit numbers, we will use a list of digits:

```scheme
;; a DigitList is a (listof Digit)
;; Requires: the last value in the list is not zero.
```

;; if you prefer:

```scheme
;; a DigitList is either:
;; '() or
;; (cons d L) where d is a Digit and L is a digitList.
;; Requires: the last value in the list is not zero.
```

This uses a list to represent a number in base 10. The digits are stored smallest first, so they seem backwards. For example, the number 245 is represented (list 5 4 2). Also note: zero is represented as the empty list, '().

(a) Carry.

Write a function (increment-with-carry digits) that consumes a DigitList, and returns the DigitList representing the number one greater.

```scheme
(increment-with-carry (list 5 4 2)) => (list 6 4 2)
(increment-with-carry (list 9 1)) => (list 0 2)
(increment-with-carry (list 9 9 9)) => (list 0 0 0 1)
```

(b) Recursive addition. Adding two numbers can be viewed as a recursive process.

First consider small numbers with no carry. 5321 + 57 = 5378. This can be viewed as 532 + 5 = 537 followed by 1 + 7 = 8. Recursively compute 532 + 5, giving 537; so the final answer is 537 followed by 8, i.e. 5378.

Now consider an addition with carry. 5346 + 57. Considering the units digits, 6 + 7 = 13. This has a carry; the units digits of the sum will be 3, but the rest of the work is one more than 534 + 5; one more than 539 is 540, so answer is 540 followed by 3, so 5403.

Write a function (add D1 D2) that consumes two DigitList, and returns the DigitList representing their sum.

```scheme
(check-expect (add (list) (list 4)) (list 4)) ; 0 + 4 = 4
(check-expect (add (list 3) (list 4)) (list 7)) ; 3 + 4 = 7
(check-expect (add (list 7) (list 6)) (list 3 1)) ; 7 + 6 = 13
(check-expect (add (list 5 4 2) (list 7 6 5)) (list 2 1 8)) ; 245 + 567 = 812
```

Do not convert your DigitList to a regular Nat. Do the computation using the recursive structure of the DigitList.