Assignment Guidelines.

- This assignment covers material in Module 8.
- Submission details:
  - Solutions to these questions must be placed in files `a08q1.rkt`, `a08q2.rkt`, `a08q3.rkt`, and `a08q4.rkt`, respectively, and must be completed using Racket Intermediate Student with `lambda`.
  - 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.
  - 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:
  - Read each question carefully for additional restrictions.
  - You may use recursion or higher order functions (`map`, `filter`, `foldr`). You may use any combination of the tools we have learned so far.

- 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. Lists? We don’t need no stinking lists!. Recall that we defined a (listof Int) as follows:

A (listof Int) is either

- '(), or
- (cons v L) where v is an Int and L is a (listof Int).

We then used the built-in functions first and rest to extract parts of the list, and the built-in function cons to make a list one longer.

Here we will create a structure that is able to store arbitrarily long data, just like a list.

For clarity, we will consider only storing Int values, but the same structure would work for any value, just like a list.

For this question, use the following data definition:

```
(define-struct ls (first rest))
;; a Ls is either
;; '(), or
;; (make-ls first rest) where first is an Int and rest is a Ls.
```

Keep in mind that since the structure is named ls, and its fields are named first and rest, you will access these fields using ls-first and ls-rest.

**Exercise**

(a) Length. Write a function (ls-length L) that consumes a Ls and returns the number of values in it. For example,

```scheme
(check-expect (ls-length (make-ls 5 (make-ls 7 (make-ls 11 '())))) 3)
```

(b) Max. Write a function (ls-max L) that consumes a non-empty Ls and returns the largest value. For example,

```scheme
(check-expect (ls-max (make-ls 5 (make-ls 11 (make-ls 7 '())))) 11)
```

2. Summarizing Data.

Write a function (remove-item L item) that returns L with all copies of item removed.

```scheme
(remove-item (list 42 "a" "bee" "sea" "bee" 17) "bee") => (list 42 "a" "sea" 17)
```

Write a function (count-item L item) that counts how many values in L are equal to item.

```scheme
(count-item (list 42 "a" "bee" "sea" "bee" 17) "bee") => 2
```

**Hint** Remember, you may use any of our tools so far. Does one of our higher order functions help a lot?

Recall the following data definition:

```
(define-struct asc (key val))
;; An Asc is a (make-asc Any Any)
;; a Dict (dictionary) is a (listof Asc)
```

Write a function (summarize L) that consumes a (listof Any) and returns a Dict. Each different value from L appears in the answer just once, with the value being the value from L, and the count being the number of times that value appears in L. The items occur in order of first appearance.

For example,

```scheme
(summarize (list "b" "X" "b" "c" "b" "c"))
=> (list (make-asc "b" 3) (make-asc "X" 1) (make-asc "c" 2))
```
3. Keys in Multiple Dictionaries.

Write a function (common-keys D1 D2) that consumes two Dict and returns a (listof Any) containing all the keys which occur in both dictionaries. The keys should stay in the same order they occur in D1.

For example,

```lisp
(common-keys
 (list (make-asc 1 "one") (make-asc 5 "five")
       (make-asc 7 "seven") (make-asc 3 "three"))
 (list (make-asc 3 "trois") (make-asc 8 "huit")
       (make-asc 11 "onze") (make-asc 5 "cinq")))
=> (list 5 3)
```

4. Letter Translation. People have been using encryption to send messages secretly for thousands of years. The Caesar Cipher is named after Julius Caesar, who used it to keep messages secret even if the messenger was captured (or a spy).

The concept is simple: each letter is replaced with another letter. A translation table is a list that indicates which character substitutes for another character.

```lisp
(define-struct sub (src dst))
;; a Sub is a (make-sub Char Char)

(define rot-ate (list
 (make-sub #\a #\t)
 (make-sub #\t #\e)
 (make-sub #\e #\a))

For example, rot-ate is a translation table that replaces #\a with #\t, #\t with #\e, and #\e with #\a.
```

Write a function (translate word cipher), which consumes a Str and a (listof Sub). It returns the Str that results from translating all characters in word using the translations described by cipher. Any characters that occur in word but not in cipher should be included unchanged in the output.

For example,

```lisp
(translate "can this be hard to read?" rot-ate) => "ctn ehis ba htrd eo ratd?"
```