Assignment Guidelines.

- This assignment covers material in Module 8.
- Submission details:
  - Solutions to these questions must be placed in files `a8q1.rkt`, `a8q2.rkt`, `a8q3.rkt`, and `a8q4.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:
  - 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. Working with Structures. Examples in this question will use the following list:

```
(define-struct student (name program))
;; a Student is a (make-student Str Str)

;; a (listof Any) for some examples.
(define test-list (list 3
   (make-posn 4 5)
   (make-student "Al Gore" "government")
   17
   (make-student "Bill Gates" "appliedmath")
   "boo"
   (make-posn 7 11)
   (make-student "Conan O'Brien" "history")
   "foo"))
```

(a) Fixed Structs and Fields. Write a function (posn-x-values L) that consumes a (listof Any). The function returns a (listof Num) containing the x fields of all the values in L that are of type Posn. For example,

```
(check-expect (posn-x-values test-list) (list 4 7))
```

(b) Extract Type and and Selector. Write a function (extract-fields L type-predicate selector). It consumes a (listof Any), and two Function arguments. The function will select all the values from L that satisfy type-predicate, and extract the field given by selector. For example,

```
(check-expect (extract-fields test-list posn? posn-x) (list 4 7))
(check-expect (extract-fields test-list student? student-name) (list "Al Gore" "Bill Gates" "Conan O'Brien"))
```

This function will be similar to posn-x-values, but more general.

2. 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) **Summing.** Write a function `(ls-sum L)` that consumes a `ls` and returns the sum of all the values in it. For example,

```
(check-expect (ls-sum (make-ls 7 (make-ls 11 '())))) 18
```

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

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

### 3. Dictionaries.
Recall the following data definitions:

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

;; a Dict is a (listof Asc)
;; Requires: there are no duplicate keys.
```

The following `Dict` are used below for examples:

```
(define test-d1 (list (make-asc "blue" 1)
                      (make-asc "gold" 99)
                      (make-asc "brown" 20)
                      (make-asc "bright cyan" 59)
                      (make-asc "dark gray" 56)))

(define test-d2 (list (make-asc "red" 4)
                      (make-asc "bright cyan" 59)
                      (make-asc "brown" 20)
                      (make-asc "gold" 105)
                      (make-asc "bright white" 63)))
```

**Exercise**

(a) **Searching.** Write a predicate function `(has-key? d k)` that consumes a `Dict` and an `Any`. The function returns `#true` if one of the items in `d` has the key `k`, and `#false` otherwise. For example,

```
(check-expect (has-key? test-d1 "blue") #true)
(check-expect (has-key? test-d1 "green") #false)
```

(b) **Intersection.** Write a function `(intersection d1 d2)` that consumes two `Dict`, and returns the dictionary containing all the associations in `d1` where the key is also used in `d2`. The values need not match. If the same key is associated with different values in `d1` and `d2`, use the value from `d1`. Keep the dictionary in the same order as the items in `d1`. For example,

```
(check-expect (intersection test-d1 test-d2)
              (list (make-asc "gold" 99)
                   (make-asc "brown" 20)
                   (make-asc "bright cyan" 59)))
```