Assignment 08
Due Wednesday, July 22 at 10:00 am (no late submissions)

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 many 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. A Strange Recursive Structure. We can store information in many different ways. With lists, we store a `first` item, and the `rest` of the items.

Here we will store the contents of a `Str`, in three parts:

(1) the first character
(2) the middle characters
(3) the last character.

For example, in `"Babbage"`, the parts are `"B"`, `"abbag"`, and `"e"`.

If we store the middle characters using the same kind of structure, we have a recursive data structure that stores the contents of a `Str`.

We will use the following data definition:

```
(define-struct silly-string (first middle last))
;; a SillyStr is either:
;;    * a Str of length 0 or 1, or
;;    * a (make-silly-string Str SillyStr Str)
;; Requires: first and last have length exactly 1.
```

Exercise

Write a function `(sillify s)` that consumes a `Str` and returns the corresponding `SillyStr`.
For example,

```
(check-expect (sillify "a") "a")
(check-expect (sillify "yo") (make-silly-string "y" "" "o"))
(check-expect
 (sillify "Babbage")
 (make-silly-string "B"
                    (make-silly-string "a"
                                       (make-silly-string "b"
                                                          "b"
                                                          "a")
                                       "g")
                    "e"))
```

Hint

The function `(unsilly s)`, which does the opposite of what you need to, is included in the interface file. You may use it to write *some* additional tests. For example,

```
(check-expect
 (unsilly (sillify
           "Those who can imagine anything, can create the impossible."))
 "Those who can imagine anything, can create the impossible.")
```

2. Palindromes. A palindrome is a sequence of characters that is the same forwards as backwards: for example, `"racecar"` is a palindrome, while `"Koenigsegg"` is not.

If the first and last characters are equal, and the middle characters are a palindrome, then the sequence is a palindrome. This corresponds to the structure of a `SillyStr`. So our `SillyStr` data structure is ideally designed for identifying palindromes.

We will not consider lower- and upper-case letters to be the same. So we will not consider `"Hannah"` to be a palindrome, even though `"hannah"` is.

Exercise

Write a function `(palindrome? ss)`.
It consumes a `SillyStr` and returns `#true` if `ss` a palindrome, and `#false` otherwise. For example,

```
(check-expect (palindrome? (make-silly-string "h"
                                              (make-silly-string "e" "l" "l")
                                              "o"))
              #false)
(check-expect (palindrome? (make-silly-string "r"
                                              (make-silly-string "a" "d" "a")
                                              "r"))
              #true)
(check-expect (palindrome? (sillify "racecar")) #true)
(check-expect (palindrome? (sillify "heptapod")) #false)
(check-expect (palindrome? (sillify "able was I ere I saw elba")) #true)
(check-expect (palindrome? (sillify "Hannah")) #false)
(check-expect (palindrome? (sillify "hannah")) #true)
```

! Do not convert to a `Str`. You must directly use the recursive structure of the `SillyStr`.

3. Shopping List. We are going to write some pieces of a program to keep track of ingredients for cooking. For each kind of food that we have, we will use a `Str` store what the ingredient is, and a `Num` to store how much we have of it. (You can think of this as the exact number for things like eggs, and kilograms for everything else. Our program won't care what the units are.)

We describe an `Ingredient` as follows:

```
(define-struct ingredient (name count))
;; An Ingredient is a (make-ingredient Str Num)
;; Requires: count > 0.
```

When we have a list of ingredients, we want to mention each kind only once. So we define the following:

```
;; A FoodList is a (listof Ingredient)
;; Requires: no name appears more than once.
```

For tests I am going to imagine I have the following:

```
(define pantry
  (list (make-ingredient "egg" 12)
        (make-ingredient "flour" 10)
        (make-ingredient "butter" 0.5)
        (make-ingredient "sugar" 2)
        (make-ingredient "rice" 5)
        (make-ingredient "cinnamon" 0.005)
        (make-ingredient "yeast" 0.113)
        (make-ingredient "raisins" 0.4)))
```

I also have some recipes that need certain ingredients. For example,

```
(define cookies
  (list (make-ingredient "flour" 0.3)
        (make-ingredient "sugar" 0.06)
        (make-ingredient "butter" 0.24)))

(define raisin-bread
  (list (make-ingredient "flour" 1)
        (make-ingredient "sugar" 0.1)
        (make-ingredient "yeast" 0.007)
        (make-ingredient "cinnamon" 0.05)
        (make-ingredient "raisins" 1)))
```

Exercise

Write a function `(quantity name foods)`. It consumes a `Str` and a `FoodList`, and returns a `Num` indicating how much of `name` there is in `foods`. For example, something we have in `pantry`:

```
(check-expect (quantity "raisins" pantry) 0.4)
```

And something we have none of:

```
(check-expect (quantity "vinegar" pantry) 0)
```

Now we want to be able to determine what we need to buy.

Exercise

Write a function `(find-missing available recipe)`. It consumes two `FoodList`, and returns a `FoodList` containing all the things from `recipe` for which there is not enough in `available`.
For example, we can't make raisin bread without more cinnamon and raisins. We need 1 kg of raisins but only have 0.4 kg, so we need 0.6 kg more. We need 0.05 kg of cinnamon but only have 0.005 kg, so we need 0.045 kg more.

```
(check-expect (find-missing pantry raisin-bread)
              (list (make-ingredient "cinnamon" 0.045)
                    (make-ingredient "raisins" 0.6)))
```

! Return the items in the same order that they appear in `recipe`.

But we have everything we need to make cookies, so the list of missing things is empty:

```
(check-expect (find-missing pantry cookies) '())
```

4. 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,

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

Exercise

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

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