Assignment Guidelines:

- For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including helper functions.
- For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe. Read the Style Guide carefully to ensure that you are following the proper conventions. In addition, your solution must include the definition of constants and helper functions where appropriate.
- 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 on LEARN.
- 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.
- Read each question carefully for restrictions.
- Test data for all questions will always meet the stated assumptions for consumed values.
- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include references to the parameter names of your functions.
- 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.
- Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
- You may post general assignment questions using the discussion groups on Waterloo LEARN. Choose Connect Discussions. Read the guidelines for posting questions. Do NOT post any code as part of your questions.
- Check Markus and your basic test results to ensure that your files were properly submitted. In most cases, solutions that do not pass the basic tests will not receive any correctness marks.
- Read the course web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the Style Guide.
- Your solutions should be placed in files a7qY.rkt, where Y is a value from 1 to 4.

Plagiarism: Read https://www.student.cs.uwaterloo.ca/~cs115/#assignments

Language level: Beginner Student with List Abbreviations
Coverage: Module 06 and 07
Question 1 – Deduplication (Dedup):
Duplicate data costs employers a lot of money. Consider an email server that stores the
e-mails/attachments for a company. When one employee emails their coworkers a photo, the server
stores duplicates of the photo. When the other employees "reply all" to that email, the server stores
even more duplicates. This duplication wastes a high percentage of the server's storage capacity, but
the wasted space can be recovered by removing the duplicates, or "deduping" the data.

Write a function (dedup lst0 lst1) that consumes two sorted lists of numbers and merges them
into one sorted list of numbers. The produced list should not contain any duplicates.

Examples
(dedup empty empty) => empty
(dedup (list 1 2 2 4 5 5 6 6 9)
       (list 1 1 1 2 2 2 5 7 9 9)) => (list 1 2 4 5 6 7 9)

Question 2 – Split:
Write a function (split lst idx) that consumes a list and an index idx, a natural no larger than
the list length. This function produces a list containing exactly two other lists that we'll refer to as first
and second. The first list contains all of the elements (if any) in lst up to and including the element
with index idx. The second list contains all of the elements (if any) in lst after the index idx.

Examples
(split empty 0) => (list empty empty)
(split (list 'A) 0) => (list empty (list 'A))
(split (list 'A) 1) => (list (list 'A) empty)
(split (list 'A "B" #\C) 2) => (list (list 'A "B") (list #\C))

Question 3 – Splice:
DNA is the hereditary material of humans. It encodes our physical traits using a sequence of chemicals:
adenine (A), guanine (G), cytosine (C) and thymine (T). We can represent a DNA sequence using a list
with symbols 'A, 'G, 'C and 'T:

;; A DNA-Sequence is a (listof (anyof 'A 'C 'G 'T))

Suppose that we've identified a sequence of DNA that encodes a favourable trait, e.g., resistance to a
particular disease. DNA splicing allows us to switch an unfavourable sequence for the favourable
sequence.

a. Write a function (splice-end seq0 seq1 idx) that consumes two DNA-Sequence's seq0
   and seq1, and a natural idx which is the index of an element in seq0 and seq1. The function
   produces a DNA-Sequence: all elements in seq0 up to/including the index (if they exist), followed
   by the elements in seq1 after the index (if they exist). Note that idx must be less than or equal to
   the length of the shorter of the two DNA-Sequence's, i.e.,

   idx <= (min (length seq0) (length seq1)).
Examples
(splice-end empty empty 0) => empty
(splice-end (list 'A 'A 'A) (list 'C 'G 'T) 0) => (list 'C 'G 'T)
(splice-end (list 'A 'A 'A) (list 'C 'G 'T) 3) => (list 'A 'A 'A)
(splice-end (list 'A 'A 'A 'A 'A 'A 'A 'A 'A 'A) (list 'C 'C 'C 'G 'T 'C 'C 'C) 2) => (list 'A 'A 'C 'G 'T 'C 'C)

b. Write a function (splice-subseq seq0 seq1 idx0 idx1) that is similar to splice-end, except that it consumes a second natural index idx1. This function produces the same DNA-Sequence as splice-end, except that the sequence ends with the elements of seq0 after idx1. Note that we must have:

idx0 <= idx1 <= (min (length seq0) (length seq1))

Examples
(splice-subseq empty empty empty 0 0) => empty
(splice-subseq (list 'A 'A 'A) (list 'C 'G 'T) 0 0) => (list 'A 'A 'A)
(splice-subseq (list 'A 'A 'A) (list 'C 'G 'T) 0 3) => (list 'C 'G 'T)
(splice-subseq (list 'A 'A 'A) (list 'C 'G 'T) 3 3) => (list 'A 'A 'A)
(splice-subseq (list 'A 'A 'A 'A 'A 'A 'A 'A 'A 'A) (list 'C 'C 'C 'G 'T 'C 'C 'C) 2 5) => (list 'A 'A 'C 'G 'T 'A 'A 'A 'A 'A 'A 'A 'A 'A)

c. BONUS. Write a function (splice seq0 seq1 indices) that is similar to splice-subseq, except that it consumes a sorted list of natural indices instead of idx0 and idx1. This function produces the same DNA-Sequence as splice-end, except that the produced sequence alternates between seq0 and seq1 after every index in the indices list. Note that if indices is empty, the function produces seq0; otherwise, we must have

(max indices) <= (min (length seq0) (length seq1))

Examples
(splice empty empty empty) => empty
(splice (list 'A 'A 'A) (list 'C 'G 'T) empty) => (list 'A 'A 'A)
(splice (list 'A 'A 'A) (list 'C 'G 'T) (list 0)) => (list 'A 'A 'A)
(splice (list 'A 'A 'A) (list 'C 'G 'T) (list 0 0)) => (list 'A 'A 'A)
(splice (list 'A 'A 'A) 
(list 'C 'G 'T) 
(list 1 2)) => (list 'A 'G 'A)

(splice (list 'A 'A 'A) 
(list 'C 'G 'T) 
(list 1 2 3)) => (list 'A 'G 'A)

(splice (list 'A 'A 'A) 
(list 'C 'G 'T) 
(list 0 1 2 3)) => (list 'C 'A 'T)

(splice (list 'A 'A 'A 'A 'A 'A 'A 'A 'A 'A) 
(list 'C 'C 'C 'G 'T 'C 'C 'C) 
(list 1 3 6 7)) => (list 'A 'C 'C 'A 'A 'A 'C 'A 'A 'A 'A 'A)

Hint for reusing examples/tests:
- if indices contains exactly one element, then splice is equivalent to splice-end;
- if indices contains exactly two elements, then splice is equivalent to splice-subseq.

Question 4 – Association Lists:
Encryption allows us to disguise a message so that only a friend can read it. One of the simplest ways to disguise a word is to convert it character-by-character to a new word. If our friend knows how we converted the word, they can convert it back.

Suppose we use a variation of AssociationList's to map lowercase character keys to lowercase character values:

;;; A CharAssociation (CA) is a (list Char Char).
;;; A CharAssociationList (CAL) is a (listof CA).
;;; requires: No key appears more than once in CAL.

We want to use this CAL to convert words character by character. For example, if we use

(define mapping 
(list 
(list #\a #\b) 
(list #\b #\c) 
(list #\c #\a) 
(list #\d #\d)))

then the word "cabac" converts to "abcba" because (list #\c #\a) converts "c" to "a", 
(list #\a #\b) converts "a" to "b" and (list #\b #\c) converts "b" to "c", and so on.

a. Write a function (convert cal word) that consumes a CAL and String and produces the same String converted character-for-character using cal. Use the provided lookup function and assume that cal maps all of the characters in word.
Examples
(convert mapping "") => ""
(convert empty "") => ""
(convert mapping "abcd") => "bcad"
(convert (list (list #\a #\a)) "aaa") => "aaa"

b. Once we have a CAL and the convert function, we can disguise words and send them to our friend. To un-disguise the word, our friend must also use convert with a different CAL, one that reverses the original CAL that we used. For example, to reverse words that were converted using mapping, our friend must use the rev-map CAL:

(define rev-map
  (list (list #\a #\c)
       (list #\b #\a)
       (list #\c #\b)
       (list #\d #\d)))

We disguise the word: (convert mapping "abcd") => "bcad"
Our friend un-disguises the word: (convert rev-map "bcad") => "abcd"
Consequently, (convert rev-map (convert mapping "abcd")) => "abcd".

There are two interesting properties of CAL's:
1. Not every CAL has a reverse CAL. Think about why some CAL's have a reverse while others do not. This sounds like a great exam question!
2. Some CAL's are their own reverse CAL. We call these involutions. Involutions offer convenience because our friend can use the same CAL that we use.

Consider the CAL:
(define invol
  (list (list #\a #\z)
       (list #\b #\y)
       (list #\c #\x)
       (list #\x #\c)
       (list #\y #\b)
       (list #\z #\a)))

The example demonstrates (but doesn’t prove!) that invol is an involution:
(convert invol (convert invol "abc")) => "abc"
because (convert invol "abc") => "zyx"
and (convert invol "zyx") => "abc".

Write a predicate (involution? cal) that consumes a CAL and produces true if cal is its own reverse CAL. There are several different approaches that can be used to solve this problem!

Examples
(involution? empty) => true
(involution? mapping) => false
(involution? rev-map) => false
(involution? invol) => true