The interface file for this assignment will include the structure and data definitions for the questions. You are required to write the function headers so they match the name of the function, and the correct number and order of the parameters as described in each question.

Download the interface file from the course Web page to ensure that all structures are defined correctly.

Errors in the names of the required functions as well as the number of parameters of the functions will be identified by the basic tests. Be sure to submit your solutions early in order to avoid losing all correctness marks for such errors.

Do not use reverse.

You must provide the data definition and template in your solutions only when the question specifically indicates that they are required for compound data types described in the question. If you create any additional data type beyond the question description, your program file should include a data definition and template for each additional data type.

If you include a template in your solution, the template should appear in comments.

Unless otherwise indicated by the question you may only 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

For this and all subsequent assignments, you are expected to follow the Style Guide when writing functions from scratch, including helper functions.

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.

You may post general questions regarding the assignment to clarify the purpose of each function using the discussion forum on Waterloo LEARN. Choose Connect -> Discussions. Read the guidelines for posting questions. Do not post any code as part of your questions.

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.

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.

Any string or symbol constant values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic tests results will catch many, but not necessarily all of these types of errors.

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 a07qY.rkt, where Y is a value from 1 to 3.

Language level: Beginning Student with List Abbreviations
Coverage: Modules 6 and 7
Assignment 07  
Due: Wednesday, November 15, 2017 at 10:00 a.m.

1. Encryption is the process of hiding information. For this problem, you will be decoding a secret message that is hidden in a list of strings. Some of the strings in the list contain a character that is part of the secret message. Some of the strings are decoys that will be ignored when decoding the message. To decode the message, you need a key. In this case, the key is a list of natural numbers. Some of the numbers will be used to decode the message, and some of the numbers are decoys that will be ignored when decoding the message.

Write a function called decode that consumes a list of strings and a list of natural numbers and produces a string that represents a decoded message. Your decoded message should appear as a string with all uppercase letters. To decode the message, use the following rules:

- Read a string from the list of strings and a number from the list of numbers.
- If the character in the string at the index position indicated by the number is an alphabetic character (i.e. an uppercase or lowercase letter), then that character is part of the message.
- If the number is a valid index position in the string, but there is a non-alphabetic character at that position in the string, then that string is a decoy and you should skip that string when decoding the message.
- If the number is not a valid index position in the string, then that number is a decoy and you should skip that number when decoding the message.
- The message is completely decoded when you run out of strings or numbers.
- Any given string and any given number are used at most once when decoding the message. In some cases, a string or number may be completely ignored in the decoding.

For example (decode (list "ab g" "of" "#hi?" "C" "as?") (list 3 0 0 5 1 4)) produces "GOCS". This result is determined as follows:

- The letter "g" appears at index position 3, so it is included in the message. Read the next string and number.
- The letter "o" appears at index position 0, so it is included in the message. Read the next string and number.
- There is a "#" at index position 0 of the next string in the list so we skip that string.
- The letter "C" appears at index position 0 of the next string, so it is included in the message. Read the next string and number.
- The index position 5 is invalid in the string "as?" so we skip that number.
- The letter "a" appears at position 1 of the string "as?", so it is included in the message. Try to read the next string and number.
- There are no more strings available to decode, so that is the end of the message.
2. Word wrap is a formatting option where you break text into separate lines to fit into a fixed column width. Write a function called `word-wrap` that consumes a non-empty string representing some text and a natural number, greater than 1, representing a column width. The function should produce a string containing newline characters where the text would need to break to fit in the column. The newline character does not take up space in the column. The last character of the string produced should be the last character of the text consumed.

For example:

```racket
(word-wrap "abcdefg" 5) produces "abcde\nfg"
(word-wrap "University of Waterloo" 8) produces "Universi\nty of Wa\nterloo"
(word-wrap "abcdefghij" 5) produces "abcde\nfg\nj"
```

You may **not** use any string functions in your solution except `string->list` and `list->string`.

Note: There is a constant called `#\n` in Racket. This is a character representing a newline in a string of characters. It appears as `"\n"` within a string. For example, `(string->list "ab\ncd")` produces `(list #\a #\b #\n #\c #\d)`.

3. We have many applications that try to automatically correct our typing. Suppose you have a dictionary where the keys were words, and the associated values are structures that contain information about the popularity of the word in the English language (its rank) and a list of common misspellings of that word. For this question, you should use the following structure and data definitions:

```
;; A **Token** is a non-empty Str that contains only lowercase letters.
(define-struct token (token))
;; A **WordInfo** is a (make-wordinfo Nat (listof Token))
;;  Requires:
;;    rank is a positive integer representing the frequency of the word as it is used in the English language
;;    misspellings is a list of common misspellings of the word

;; An **Entry** is a (list Token WordInfo)
;;  where
;;    the first item is the key to the dictionary
;;    the second item is information associated with the key

;; A **Dictionary** is one of
;;  * empty
;;  * (cons Entry Dictionary)
;;  Requires: keys are unique in the Dictionary
;;            keys cannot appear as entries in any misspellings lists in the Dictionary
```
Write a function called `autocorrect` that consumes a `Token` and a `Dictionary`, and produces suggested corrections. Specifically:

- If the `Token` consumed does not appear as a misspelling anywhere in the dictionary, then the function produces a copy of the original `Token`.
- If the `Token` consumed appears as a misspelling of only one key in the dictionary, then the function produces that key.
- If the `Token` consumed appears as a misspelling of more than one key in the `Dictionary`, then the function produces a `(listof (list Token Nat))`, where the `Token` is a key from the `Dictionary` that contains a matching misspelling and the `Nat` is the rank of that key. The elements of this list may appear in any order, and contain all keys with matching misspellings.

For example, if you defined the following constant

```
(define sample-dict
  (list (list "the" (make-wordinfo 1 (list "teh" "th")))
       (list "there" (make-wordinfo 53 (list "ther" "theer" "th"))))))
```

then

```
(autocorrect "hello" sample-dict) produces "hello"
(autocorrect "theer" sample-dict) produces "there"
(autocorrect "th" sample-dict) produces (list (list "the" 1) (list "there" 53))
  or (list (list "there" 53) (list "the" 1))
```

In your own tests, the expected value will be just one list. In the test, you need to predict the order of that list by knowing how your own function generates its result.