Assignment: 5
Due: Thursday, February 28th, 9:00 pm
Language level: Beginning Student with list abbreviations
Allowed recursion: Pure structural recursion and structural recursion with accumulators
Files to submit: flix.rkt, codes.rkt, tweet.rkt, bonus.rkt
Warmup exercises: HtDP 10.1.4, 10.1.5, 11.2.1, 11.2.2, 11.4.3, 11.5.1, 11.5.2, 11.5.3
Practice exercises: HtDP 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7

• Make sure you read the OFFICIAL A05 post on Piazza for the answers to frequently asked questions.

• Unless stated otherwise, all policies from Assignment 04 carry forward.

• This assignment covers material up to the end of Module 07.

• Of the built-in list functions, you may use only cons, first, second, third, rest, empty?, cons?, list, member?, and length. You may also use equal? to compare two lists.

• If you choose to define a type, provide a complete data definition. You do not need to provide a template, but can have one if you want.

• Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.

• Unless the question specifically says otherwise, you are always permitted to write helper functions. You may use any constants or functions from any part of a question in any other part.

• Solutions will be marked for both correctness [75%] and good style [25%]. Follow the guidelines in the Style Guide.
1. **[5% Correctness]** Perform the assignment 5 questions using the online evaluation “Stepping Problems” tool linked to the course web page and available at


   The instructions are the same as those in assignment 4. If you get stuck, please do not post questions to Piazza asking for the next step, since such questions are considered to be violations of our academic integrity policy. If you are really stuck, then see an ISA or instructor in person.

2. **[20% Correctness]** For this question, you will be writing functions that produce hidden messages from strings. For all parts of this question, you may use `string->list` and `list->string`, but you may **not** use other string functions.

   (a) Write a function `get-char` that consumes a string and a natural number, and produces the character in the string at the position given by the number. Numbering begins with 0. If the given position is out of range (i.e. if you run out of characters), produce the character `#\*`. For example:

   ```scheme
   (get-char "abcdefg" 0) produces #\a
   (get-char "abcdefg" 3) produces #\d
   (get-char "abcdefg" 20) produces #\*
   ```

   **Hint:** You will need to do recursion on both a `Nat` and a `(listof Char)` at the same time. You must adjust your base case(s) and recursive case(s) accordingly.

   (b) Write a function `coded-3-char` that consumes a string and a Decryptor (defined below), and produces a string containing the 3-character “secret message” hidden in the consumed string.

   ```scheme
   ;; A Decryptor is a (list Nat Nat Nat)
   ```

   The first `Nat` in a Decryptor gives the number of characters that should be skipped to get to the first secret character, the second `Nat` gives the number to skip to get to the second secret character, and the third `Nat` gives the number to skip to get to the third. For the second and third `Nats`, skips begin after the previous secret character in the string. For example:

   ```scheme
   (coded-3-char "abcdefg" (list 0 0 0)) produces "abc"
   (coded-3-char "abcdefg" (list 2 1 0)) produces "cef"
   ```

   If the Decryptor skips past the end of the string, then `*` should be produced for every character that does not exist. For example:

   ```scheme
   (coded-3-char "abcdefg" (list 2 10 0)) produces "c**"
   ```
(c) Write a function `enc-possible?` that consumes two strings and produces `true` if the second string may be hidden inside the first string using a Decryptor, and `false` otherwise. You may assume that the second string consumed has length 3 and does not contain `. For example:

- `(enc-possible? "abcdefg" "bdg")` produces `true`
- `(enc-possible? "abcdefg" "abz")` produces `false`
- `(enc-possible? "abcdefg" "bac")` produces `false`

*Hint*: It may be easier to treat the second string as if you do not know its length.

Place your solutions in the file `codes.rkt`

3. **[25% Correctness]** In this question you will be writing code to keep track of tweets using the following definition:

```
(define-struct tweet (sender text tags))
;; A Tweet is a (make-tweet Str Str (listof Sym))
;; requires: tags does not contain any duplicate tags
```

(a) Write the function `retweet` that consumes a `Tweet`, a string representing a sender, and either a symbol representing an additional tag to add to the tweet, or `false` if no new tag should be added. The function produces a new `Tweet` that has the consumed sender, the same text as the consumed `Tweet`, and has all of the tags of the consumed `Tweet`, as well as the additional tag. The additional tag should be placed last, and should only be placed if it was not already present.

Example:

```
(check-expect (retweet (make-tweet "user1" "test tweet" (list 'a 'b))
                    "user2" 'c)
           (make-tweet "user2" "test tweet" (list 'a 'b 'c))
```

Note: A sender can retweet themselves (that is, the consumed sender is allowed to be the same as the sender of the consumed `Tweet`).

(b) Write the function `find-rt` that consumes a list of `Tweets` and a string, and produces a list of `Tweets` with a text field equal to the consumed string.

Example:

```
(check-expect (find-rt (list (make-tweet "user1" "A" empty)
                          (make-tweet "user2" "B" empty)
                          (make-tweet "user2" "A" empty))
               "A")
               (list (make-tweet "user1" "A" empty)
                      (make-tweet "user2" "A" empty)))
```

The values in the produced list must be in the same relative order as they were in the consumed list.
(c) Write the function `find-tagged` that consumes a list of Tweets and a symbol. The function produces a list of all Tweets that have a tags field that contains the consumed symbol.
Example:

```scheme
(check-expect (find-tagged (list (make-tweet "u1" "C" (list 'a 'b))
                               (make-tweet "u1" "A" (list 'c))
                               (make-tweet "u2" "T" (list 'a 'b))
                               (make-tweet "u3" "!" (list 'a 'c)))
            'c)
           (list (make-tweet "u1" "A" (list 'c))
                  (make-tweet "u3" "!" (list 'a 'c)))))
```

The values in the produced list must be in the same relative order as they were in the consumed list.

(d) Write the function `find-tagged/multi` that consumes a list of Tweets and a list of symbols. The function produces a list of all Tweets that have a tags field that contains all of the symbols in the consumed list of symbols (in any order).
Example:

```scheme
(check-expect (find-tagged/multi (list (make-tweet "u" "C" (list 'a))
                                  (make-tweet "u" "A" (list 'b))
                                  (make-tweet "u" "T"
                                  (list 'c 'a 'b)))
            (list 'a 'b))
           (list (make-tweet "u" "T" (list 'c 'a 'b)))))
```

The values in the produced list must be in the same relative order as they were in the consumed list.

Place your solutions in the file `tweet.rkt`

4. [25% Correctness] Many companies, such as Netflix, use machine learning to make recommendations for their customers. For this question, you will be using a simple machine learning technique to analyze movie preferences and generate movie recommendations.

Movies can be placed into different genres and can often fit into more than one genre. For example, a movie can be a science fiction thriller. For the purposes of this question, there will be exactly eight genres.
Position | Movie Genre
--- | ---
1 | action
2 | animation
3 | comedy
4 | drama
5 | horror
6 | romance
7 | science fiction
8 | thriller

Machine learning represents information, such as movie genres, using feature vectors. In Racket, you will create a Movie-fv (movie feature vector) which will be an eight-element list where each element is a 0 or a 1.

;; A Movie-fv is a (list Nat Nat Nat Nat Nat Nat Nat Nat)
;; requires: each Nat is either 0 or 1

Each position in the list will represent a different genre and a 1 in that position will indicate that the movie can fit in that particular genre and a 0 means it does not. Position 1 in the Movie-fv represents the action genre, position 2 represents animation, and so on as listed above. For example, because it is in the genres romance and comedy, the Movie-fv for Crazy Rich Asians would be (list 0 0 1 0 0 1 0 0). The Movie-fv for The Meg would be (list 1 0 0 0 1 0 1 0) because it is in the genres action, horror and science fiction. Note for a Movie-fv the positions are between 1 and 8 inclusive (not 0 and 7).

For each of the following questions, your solutions will always be tested with valid input.

(a) One method of generating recommendations for a particular customer is to track which movies they liked or did not like by having customers rate movies they have seen and using that information to suggest additional movies they might enjoy. The movie rating will be encoded as 1 for liked and -1 for did not like.

;; A Rating is an Int
;; requires: Int is either -1 or 1

Create a function called find-preference which consumes a (listof Rating) and a (listof Movie-fv) for a particular client. The two lists are the same length and the $i^{th}$ element of the first list corresponds to the rating of the movie whose Movie-fv is the $i^{th}$ element of the second list.

The function find-preference should produce a Pref-v (i.e. a preference vector), which will be an eight-element list that consists of the sum of the scores for each of the eight genres.

;; A Pref-v is a (list Int Int Int Int Int Int Int Int)

If the client watched three animation movies and liked the first two and disliked the last one, their total score for the animation component of the Pref-v would be $2 - 1 = 1$. If
the first two were not dramas and the the last one was, their total score for the drama component would be -1.

Here is the contract for find-preference

;; find-preference: (listof Rating) (listof Movie-fv) -> Pref-v
;; requires: both lists are the same size
;; both lists are non-empty

and below are some examples. If find-preference is given the following arguments, i.e. the client liked the movie,

(find-preference (list 1)
 (list (list 1 1 0 0 0 0 0))
)

then resulting Pref-v would be (list 1 1 0 0 0 0 0). If the client did not like the movie, i.e.

(find-preference (list -1)
 (list (list 1 1 0 0 0 0 0))
)

then resulting Pref-v would be (list -1 -1 -1 0 0 0 0 0). Finally if find-preference is given the following arguments, i.e. the client liked the first two movies and did not like the last one,

(find-preference (list 1 1 -1)
 (list (list 1 1 0 0 0 0 0)
 (list 1 1 1 0 0 0 0)
 (list 0 1 1 1 0 0 0))
)

then resulting Pref-v would be (list 2 1 0 -1 0 0 0 0).

The order of the components in the Pref-v is in the exact same order as the Movie-fv.

Hint: In order to implement find-preference create two helper functions to process the Movie-fvs. The first would add two lists of integers in a pairwise fashion, e.g.

(add-list (list 1 2 3) (list -1 0 1)) = (list 0 2 4).

The second helper function would negate a list. See negate-list in Module 05 slides 42–44.

(b) Create a function called suggestions. One of its parameters will be a (listof Movie) where Movie has the following structure.

(define-struct movie (title genres))
;; A Movie is a (make-movie Str Movie-fv)

The contract for suggestions is

;; suggestions: Pref-v (listof Movie) -> Str
;; requires: (listof Movie) is non-empty

For each Movie in the (listof Movie), take the dot product (see Module 06 slide 68) of the Pref-v and the Movie-fv to get a score. Report the title of the movie with the highest score.
For example, if \texttt{suggestions} had the following arguments

\begin{verbatim}
(suggestions
 (list 2 1 0 -1 0 0 0)
 (list (make-movie "The Meg" (list 1 0 0 0 1 0 1 0))
   (make-movie "Smallfoot" (list 0 1 1 0 0 0 0 0))
   (make-movie "A Star is Born" (list 0 0 0 1 0 1 0 0))))
\end{verbatim}

the movie with the highest score would be \textit{The Meg} with a score of 2 so the output would be \texttt{"The Meg"}.

There is the possibility that there will be a tie. For testing purposes, we will only test using datasets that do not result in ties, so you can break ties arbitrarily (i.e. it does not matter how you handle ties).

Place your solutions in the file \texttt{flix.rkt}

5. **Bonus Question (5%)**

Recall the definition of a Decryptor from Question 2:

\begin{verbatim}
;; A Decryptor is a (list Nat Nat Nat)
\end{verbatim}

Write a function \texttt{encode-msg} that consumes two strings and produces the Decryptor needed to hide the second string in the first one. (i.e. If \texttt{coded-3-char} were given the first string and the produced Decryptor, it would produce the second string.) For example:

\begin{verbatim}
(encode-msg "abcdefg" "abc") produces (list 0 0 0)
(encode-msg "abcdefg" "beg") produces (list 1 2 1)
(encode-msg "abcdefg" "ace") produces (list 0 1 1)
\end{verbatim}

You may assume that a valid Decryptor exists and that the second string has length 3 and does not contain \texttt{*}. If more than 1 valid Decryptor exists, you should produce the one containing the lowest values, prioritizing the first number over the second, and the second over the third. As in Question 2, you may use \texttt{string->list} and \texttt{list->string}, but other string functions are banned. You may use \texttt{length}. You must ensure that your solution uses only \textbf{pure structural recursion}.

Place your solutions in the file \texttt{bonus.rkt}

---

**Enhancements:** Reminder—enhancements are for your interest and are not to be handed in.

There is a strong connection between recursion and induction. Mathematical induction is the proof technique often used to prove the correctness of programs that use recursion; the structure of the induction parallels the structure of the function. As an example, consider the following function, which computes the sum of the first \(n\) natural numbers.
(define (sum-first n)
  (cond
   [(zero? n) 0]
   [else (+ n (sum-first (sub1 n)))]))

To prove this program correct, we need to show that, for all natural numbers \( n \), the result of evaluating \((\text{sum-first } n)\) is \( \sum_{i=0}^{n} i \). We prove this by induction on \( n \).

**Base case:** \( n = 0 \). When \( n = 0 \), we can use the semantics of Racket to evaluate \((\text{sum-first } 0)\) as follows:

\[
(\text{sum-first } 0) \\
\Rightarrow (\text{cond} [(\text{zero? } 0) 0][\text{else ...}]) \\
\Rightarrow (\text{cond} [\text{true } 0][\text{else ...}]) \\
\Rightarrow 0
\]

Since \( 0 = \sum_{i=0}^{0} i \), we have proved the base case.

**Inductive step:** Given \( n > 0 \), we assume that the program is correct for the input \( n - 1 \), that is, \((\text{sum-first } (\text{sub1 } n))\) evaluates to \( \sum_{i=0}^{n-1} i \). The evaluation of \((\text{sum-first } n)\) proceeds as follows:

\[
(\text{sum-first } n) \\
\Rightarrow (\text{cond} [(\text{zero? } n) 0][\text{else ...}]) \text{ (we know } n > 0) \\
\Rightarrow (\text{cond} [\text{false } 0][\text{else ...}]) \\
\Rightarrow (\text{cond} [\text{else } (+ n (\text{sum-first } (\text{sub1 } n)))] \\
\Rightarrow (+ n (\text{sum-first } (\text{sub1 } n)))
\]

Now we use the inductive hypothesis to assert that \((\text{sum-first } (\text{sub1 } n))\) evaluates to \( s = \sum_{i=0}^{n-1} i \). Then \((+ n s)\) evaluates to \( n + \sum_{i=0}^{n-1} i \), or \( \sum_{i=0}^{n} i \), as required. This completes the proof by induction.

Use a similar proof to show that, for all natural numbers \( n \), \((\text{sum-first } n)\) evaluates to \( (n^2 + n)/2 \).

**Note:** Summing the first \( n \) natural numbers in imperative languages such as C++ or Java would be done using a for or while loop. But proving such a loop correct, even such a simple loop, is considerably more complicated, because typically some variable is accumulating the sum, and its value keeps changing. Thus the induction needs to be done over time, or number of statements executed, or number of iterations of the loop, and it is messier because the semantic model in these languages is so far-removed from the language itself. Special temporal logics have been developed to deal with the problem of proving larger imperative programs correct.

The general problem of being confident, whether through a mathematical proof or some other formal process, that the specification of a program matches its implementation is of great importance in safety-critical software, where the consequences of a mismatch might be quite severe (for instance, when it occurs with software to control an airplane, or a nuclear power plant).