Assignment: 5

Due:	Tuesday, February 9th, 2016 at 9:00 pm
Language level:	Beginning Student with list abbreviations
Allowed recursion:	Pure structural recursion
Files to submit:	`notes.rkt`, `rle.rkt`, `plants.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
Practise exercises:	HtDP 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7

- You should note a heading at the top of the assignment: "Allowed recursion." In this case the heading restricts you to **pure structural recursion**. That is, the recursion that follows the data definition of the data it consumes, and parameters of the recursive calls are either unchanged or one step closer to the base case.
- Provide your own examples that are distinct from the examples given in the question description of each function.
- Of the built-in list functions, you may only use *cons*, *first*, *second*, *rest*, *empty?*, *cons?*, *(list ...)*, *member?*, and *append*.
- Do **not** use quoted lists for this assignment. Use *(list...)* instead.
- Your solutions must be entirely your own work.
- For this and all subsequent assignments, you should include the design recipe for all functions, including helper functions, as discussed in class.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- When naming the list-of-X parameters in your function contracts, follow the *(listof X)* notation that is given on Slide 34 of Module 05 (e.g., include a list of numbers as *(listof Num)*).
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.
- It is very important that the function names and number of parameters match ours. You must use the basic tests to be sure. The names of the functions must be written exactly. The names of the parameters are up to you, but should be meaningful. The order and meaning of the parameters are carefully specified in each problem.

Here are the assignment questions you need to submit.

1. In Western classical music, a *Note* is described by a letter (A through G inclusive) and a modifier that can be "sharp" (or ♯), "flat" (or ♭), or "natural" (or ♮).

   (a) Define a structure *Note* that complies with the following data definition:
   ;; A Note is a (make-note Sym Sym). Include the template for this structure.

   The first field is called *letter*, and it is one of 'A through 'G inclusive. The second field is called *modifier*, and it is one of 'sharp, 'flat, or 'natural.

   This representation is *overcomplete*, that is, there are more than one note that produce the same sound, or *pitch class*[1]. For example, C sharp and D flat have the same pitch class. In Western music, there are 12 distinct pitch classes in the chromatic scale. See Wikipedia[2] for a diagram showing which notes have the same pitch class. Furthermore, the notes C flat and F flat are very seldom written; B natural and E natural are written instead, respectively. Similarly, the notes B sharp and E sharp are very seldom written; C natural and F natural are written instead, respectively. Your code must correctly interpret these notes, however.

   (b) Write a function *normalize-note* that consumes a *Note* and produces an integer between 1 and 12 inclusive that gives the note's position on the chromatic scale. (C natural is at position 1, and B natural is at position 12.) The table here may be useful:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Note	C♮,B♯	C♯,D♭	D♮	D♯,E♭	E♮,F♭	F♮,E♯	F♯,G♭	G♮	G♯,A♭	A♮	A♯,B♭	B♮,C♭

   (c) Write a function *normalize-note-list* that consumes a list of *Note* structures, and produces the corresponding list of numbers that indicate the positions on the chromatic scale. The order of the values from the input list must be maintained in the produced list.

   (d) Write a function *interval*, which consumes two *Note* structures, and produces the number of increasing steps needed to get from the first note to the second note on the chromatic scale. For example from C natural to D natural takes 2 steps. From D natural to C natural takes 10 steps. The interval from a note to itself is zero.

   (e) Write a function *note-list-to-interval-list* that consumes a list of *Note* structures and produces a list of the intervals between every adjacent pair of notes, followed by the interval between the last note in the list and the first note in the list. If the consumed list has zero notes in it, produce the empty list. For example, when applied to the list (*cons* (*make-note* 'C 'natural) (*cons* (*make-note* 'D 'natural) (*cons* (*make-note* 'E 'natural) *empty*))), the function must produce (*cons* 2 (*cons* 2 (*cons* 8 *empty*))). Hint: Every non-empty list has a first and last element.

   Place your solutions in the file `notes.rkt`.

[1] http://en.wikipedia.org/wiki/Pitch_class
[2] http://en.wikipedia.org/wiki/Chromatic_scale

2. From Wikipedia[3]: Run-length encoding (RLE) is a very simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs.

   An *RlePair* is a *List* with two elements where the first element is *Any* and the second element is a non-zero *Nat*. The second element in the list tells how many times the first element should be repeated.

   An *RleList* is either *empty* or *(cons RlePair RleList)*.

   (a) Write data definitions for the *RlePair* and *RleList* data types.

   (b) Write a function *rle-decode* that consumes a run-length encoded list and produces the original list. For example, if the input was the list `(cons (cons 'red (cons 4 empty)) (cons (cons 'blue (cons 2 empty)) empty))`, then the output would be the list `(cons 'red (cons 'red (cons 'red (cons 'red (cons 'blue (cons 'blue empty))))))`.

   Place your solutions in the file `rle.rkt`.

3. *Perennial* plants and trees are those that can survive winter and continue to grow year after year; however, not all perennials can survive the winter everywhere in Canada. The geography of Canada is divided into *plant hardiness zones* that describe how severe the winter typically is.[4] A hardiness zone is described by a number from 0 through 9 inclusive indicating the Zone, and a letter indicating the SubZone, which can be `a` or `b`, with `b` being slightly milder than `a`. Zone `0a` has the most severe winter, and zone `9b` has the mildest. Waterloo is in zone `5b`.

   Each species of perennial plant is also associated with a hardiness zone, which gives the harshest zone in which it will survive. For example, blue eyed grass[5] is hardy in zone `5b`, which means it will survive the winter in Waterloo (Zone `5b`) and in Vancouver (Zone `8b`) but not in Edmonton (Zone `3a`).

   Hardiness zone information for a collection of plants or a collection of cities can be assembled into a list. For convenience, use the following data definitions for *PlantInfoList*, which is a list of *PlantInfo*s, and *CityInfoList*, which is a list of *CityInfo*s, in your code. No modification is required.

   ;; A PlantInfoList is a (listof PlantInfo)
   ;; A CityInfoList is a (listof CityInfo)

[3] http://en.wikipedia.org/wiki/Run-length_encoding
[4] http://www.planthardiness.gc.ca/
[5] http://en.wikipedia.org/wiki/Sisyrinchium_angustifolium

You can assume that *PlantInfoList* and *CityInfoList* will not include contradicting or duplicate entries, such as
(*cons* (*make-plantinfo* "chrysanthemum" 4 'b) (*cons* (*make-plantinfo* "chrysanthemum" 4 'a) *empty*))
for *PlantInfoList*, or
(*cons* (*make-cityinfo* "Calgary" 9 'b) (*cons* (*make-cityinfo* "Calgary" 4 'a) *empty*))
for *CityInfoList*.

Here are examples for each type:

```
(define sample-plant-data
  (cons (make-plantinfo "blue eyed grass" 5 'b) (cons (make-plantinfo "hosta" 3 'a)
    (cons (make-plantinfo "columbine" 4 'a) (cons (make-plantinfo "chrysanthemum" 3 'b)
      empty)))))
(define sample-city-data
  (cons (make-cityinfo "Vancouver" 8 'b) (cons (make-cityinfo "Edmonton" 3 'a)
      (cons (make-cityinfo "Waterloo" 5 'b) empty))))
```

Look for the file `plant_constants.rkt` on the assignment page for longer lists you can use for testing your code.

(a) Define a structure *CityInfo* that complies with the following data definition:
;; A CityInfo is a (make-cityinfo Str Nat Sym). The template is not needed for this structure. The first field is called *name*, and it gives the name of the city. The second field is called *zone*, and it ranges from 0 to 9 inclusive. The third field is called *subzone*, and is one of 'a or 'b.

(b) Define a structure *PlantInfo* that complies with the following data definition:
;; A PlantInfo is a (make-plantinfo Str Nat Sym). The template is not needed for this structure. The first field is called *name*, and it gives the name of the plant. The second field is called *zone*, and it ranges from 0 to 9 inclusive. The third field is called *subzone*, and is one of 'a or 'b.

(c) Write a function, *find-hardy-plants*, that consumes a *CityInfo* value and a *PlantInfoList* value, in that order. The function produces a *PlantInfoList* value containing only *PlantInfo*s for plants that will grow in the given zone. For example, based on the sample list above, (*find-hardy-plants* (*make-cityinfo* "Edmonton" 3 'a) *sample-plant-data*) produces a one-element list while (*find-hardy-plants* (*make-cityinfo* "Quebec" 4 'b) *sample-plant-data*) produces a three-element list. (You should be able to figure out what these lists would contain.) The order of the elements in the input list must be maintained in the output list.

(d) Write a function, *find-growing-cities*, that consumes a *PlantInfo* and a *CityInfoList* in that order, and produces a (*listof Str*) containing the names of all of the cities in which the plant will grow. The order of the names in the input list must be maintained in the output list.

(e) Suppose an online plant-selling business wants to ensure that its catalogue has plants suitable for all the cities in Canada. Write a function, *find-plantless-cities*, that consumes a *PlantInfoList* and a *CityInfoList* in that order, and produces a *CityInfoList* containing all of the *CityInfo*s in the given *CityInfoList* for which **no** plants in the *PlantInfoList* would survive the winter.

Place your solutions in the file `plants.rkt`.

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

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 *(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 *(sum-first 0)* as follows:

```
(sum-first 0)
yields (cond [(zero? 0) 0][else ...])
yields (cond [true 0][else ...])
yields 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, *(sum-first (sub1 n))* evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of *(sum-first n)* proceeds as follows:

```
(sum-first n)
yields (cond [(zero? n) 0][else ...]) ;(we know n > 0)
yields (cond [false 0][else ...])
yields (cond [else (+ n (sum-first (sub1 n)))])
yields (+ n (sum-first (sub1 n)))
```

Now we use the inductive hypothesis to assert that *(sum-first (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, *(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).