CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment:	5
Due:	Tuesday, October 24th, 9:00 pm
Language level:	Beginning Student with list abbreviations
Allowed recursion:	Pure structural recursion
Files to submit:	`settheory.rkt`, `pronunciation.rkt`, `drawing.rkt`, `example-drawing.png` (bonus), `ulam.rkt` (bonus)
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.
- Of the built-in list functions, you may use only *cons*, *first*, *second*, *rest*, *empty?*, *cons?*, *list*, *member?*, and *append*. You may also use *equal?* to compare two lists.
- You should not need to define any of your own types in this assignment. If you do 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.

1. It's possible to represent a set of numbers using a list, as long as the list does not contain any duplicate entries. Standard set-theoretic operations like the union, intersection and difference of two sets can then be written by comparing every element of one list against every element of the other.

   If we instead demand that sets are represented by *sorted* lists of numbers, then all of these operations can be computed more elegantly and more efficiently, by using an approach modelled on the example in the notes of merging two lists. With that in mind, we make the following data definition:

   ```
   ;; A NumSet is a (listof Num)
   ;; requires: the numbers are strictly increasing
   ```

   For example, if a set A is represented by the list '(1 2 3 6) and a set B is represented by the list '(1 3 4), then we'd expect that the union of A and B would be '(1 2 3 4 6), the intersection would be '(1 3), the difference $A - B$ would be '(2 6), the difference $B - A$ would be '(4), and the symmetric difference of A and B (sort of like the exclusive-or: elements in one set or the other, but not both) would be '(2 4 6). In all cases, the results are sorted lists with no duplicates.

   To begin, copy the data definition above into a new file called `settheory.rkt`.

   (a) Write a function *union* that consumes two parameters of type *NumSet* and produces a single *NumSet* containing all the numbers that are *in either* of the consumed sets. You must base your solution on the form of the *merge* function in the course notes, making a single pass over the two lists. *You must write a single structurally recursive function*—do not use any helper functions as part of your solution. The only built-in list functions you may use are *first*, *rest*, *empty?*, *cons?*, and *cons*.

   (b) Write a function *intersection* that consumes two parameters of type *NumSet* and produces a single *NumSet* containing all the numbers that are *in both* of the consumed sets. The same constraints described in the previous sub-question apply here.

   (c) Write a function *difference* that consumes two parameters of type *NumSet* and produces a single *NumSet* containing all the numbers that are *in the first set but not the second one*. For example, we would expect (*difference* (*list* 1 3 4 5) (*list* 1 2 3)) to produce (*list* 4 5). The same constraints described in the previous sub-questions apply here.

   (d) Write a function *symmetric-difference* that consumes two parameters of type *NumSet* and produces a single *NumSet* containing all the numbers that are *in one set or the other, but not both*. Unlike the previous sub-questions, your solution should not call itself recursively; instead, use a combination of applications of some or all of *union*, *intersection* and *difference* to obtain the symmetric difference.

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

2. A spoken word can be broken down into a sequence of *phonemes*, which are the individual consonant and vowel sounds that make it up. The phonemes can be grouped into *syllables*, with each syllable containing some consonant sounds surrounding a single vowel sound. For example, we might describe the word "describe" using the following phonemes:

   ```
   D IH0 S K R AY1 B
   ```

   Names like D, IH, S, and K are standard labels for phonemes. The vowels also have digits attached to them, which tells us the stress pattern associated with the syllables surrounding those vowels. We use the digits 0, 1 and 2 to refer to no stress, primary stress, and secondary stress, respectively. Here, we know that the first syllable (D IH S) is unstressed, and the second syllable (K R AY B) is stressed: we say "desCRIBE", not "DEScribe".

   We use the following data definitions to describe a *Dictionary*, an association list mapping a word (represented as a *String*) to its pronunciation:

   ```
   ;; A Vowel is a (list Sym (anyof 0 1 2))
   ;; A Phoneme is an (anyof Sym Vowel)
   ;; A Pronunciation is a (listof Phoneme)
   ;; requires: the list contains exactly one vowel with a stress of 1
   ;; A Dictionary is a (listof (list Str Pronunciation))
   ;; requires: the strings in each sub-list appear in alphabetical
   ;;           order in the Dictionary.
   ```

   For example, the *Dictionary* entry associated with the word "describe" might look like this:

   ```
   (list "describe" '(D (IH 0) S K R (AY 1) B))
   ```

   A standard reference for English pronunciation is The CMU Pronouncing Dictionary, containing pronunciations for over 134,000 words and proper names, in a text format similar to the data definitions above.

   To begin, download the files `pronunciation.rkt`, `pronunciationlib.rkt`, and `cmudict.txt` to the same directory. (Remember: download the .rkt files, don't cut and paste their contents into Racket.) The first file, `pronunciation.rkt`, is the one in which you'll place your solution. The file `pronunciationlib.rkt` contains extra code to provide support. In particular, it automatically reads `cmudict.txt` and builds an association list called *cmudict* with most of the words in the file. (The constant *cmudict* is defined via a complicated parsing process that you can ignore.) It also offers a constant called *toy-dictionary* with twenty-one words, so that you can see what pronunciations look like in practice. You can also use the toy dictionary for testing purposes, or define your own. Do not modify `pronunciationlib.rkt` or `cmudict.txt` in any way!

   (a) Write a function *num-syllables* that consumes a string representing an English word and a *Dictionary*. The function produces a natural number saying how many syllables are in the given word, according to the consumed dictionary. If the word is not found in the dictionary, produce the number 0. Recall that there is a one-to-one correspondence between a word's syllables and its vowel phonemes. For example, "describe" has two syllables, one using '(IH 0) and one using '(AY 1).

(b) Write a function *find-stress-pattern* that consumes a list of natural numbers that are each one of 0, 1 or 2, and a *Dictionary*. It produces a list of all those words in the dictionary whose stress pattern (the numbers associated with its vowels, in order) is equal to the given pattern. For example, (*find-stress-pattern* '(0 1) *toy-dictionary*) would produce a list of all the "natural iambs" in the small test dictionary, namely (*list* "adopt" "concrete" "deprive" "describe" "petite"). The produced list should be in alphabetical order, as they appear in the consumed dictionary.

(c) When writing metered poetry, a word's *rhyme* can be taken to be the suffix of the word's pronunciation from the vowel with primary stress until the end of the word. For example, the rhymes for "describe" and "violation" are '((AY 1) B) and '((EY 1) SH (AH 0) N), respectively. Write a function *find-rhymes* that consumes a string containing an English word and a *Dictionary*. It produces a list of strings, in alphabetical order, consisting of all words in the dictionary that rhyme with the given word (i.e., that have the same rhyme, as described above), *not including the word itself*. For example, (*find-rhymes* "pigeon" *cmudict*) produces (*list* "bijan" "pridgen" "religion" "smidgen"). You can assume that the consumed word appears in the dictionary.

In this question, you may occasionally find it helpful to use *equal?* to compare two lists.

Place your solutions in the file `pronunciation.rkt` and submit it. You should not submit `pronunciationlib.rkt` or `cmudict.txt`.

3. Let's define some types that we can use to describe simple drawings composed of coloured circles and squares. We'll need data definitions for a circle, for a square, and for a complete drawing. It will also be convenient to define a type for colours.

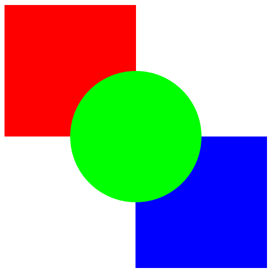
```
;; A Colour is a (list Num Num Num)
;; requires: the three numbers are between 0 and 255 inclusive.
(define-struct circle135 (centre radius fill))
;; A Circle135 is a (make-circle135 Posn Num Colour)
;; requires: radius ≥ 0
(define-struct square135 (corner side fill))
;; A Square135 is a (make-square135 Posn Num Colour)
;; requires: side ≥ 0
;; A Drawing is a (listof (anyof Circle135 Square135))
```

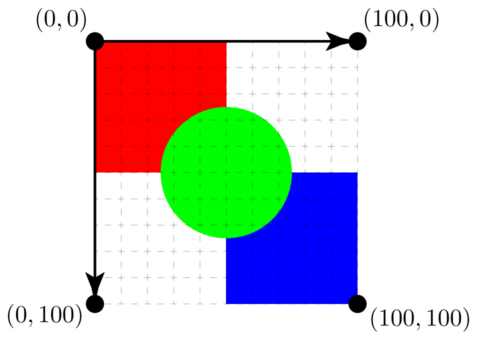
We think of the drawing as being located in an image whose origin is at the top-left, with the x axis pointing to the right and the y axis pointing down. A circle is described via the (x,y) location of its *centre*, the radius of the circle, and its fill colour. A square is described via the (x,y) location of its *top-left corner*, its side length, and its fill colour. In both cases, the fill colour is a list of three numbers corresponding to a colour's red, green and blue values. (If you're not familiar with RGB colour, do a google search on "colour picker"— Google provides one right in the search results.) A drawing is just a list, where each element is either a circle or a square.

For example, consider the following *Drawing*:

```
(list
  (make-square135 (make-posn 0 0) 50 '(255 0 0))    ;; Red
  (make-square135 (make-posn 50 50) 50 '(0 0 255))  ;; Blue
  (make-circle135 (make-posn 50 50) 25 '(0 255 0))) ;; Green
```

That drawing would look like the image on the left below. The version on the right is also annotated with coordinate axes and an evenly spaced grid, for visualization purposes.





To begin, download the files `drawing.rkt` and `drawinglib.rkt` into the same directory. The first file, `drawing.rkt`, is the one in which you'll place your solution; the other file contains extra code to provide support, including the structure and data definitions above.

In the file `drawing.rkt`, do the following:

(a) Define a constant *example-drawing* that contains a *Drawing* of your own design. You can make any drawing you want, as long as it satisfies the following conditions:

i. The drawing contains at least three circles.

ii. The drawing contains at least three squares.

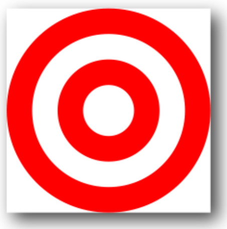
iii. The drawing should be designed for a canvas of size 300×300. That is, the line of code (*draw-picture example-drawing* 300 300) (explained below) would produce a nicely composed image.

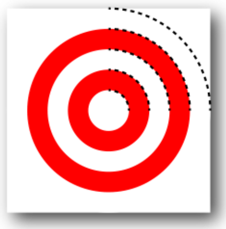
Your drawing can be an explicit list of values, or it can be the result of applying a function or functions you write to define a drawing programmatically.

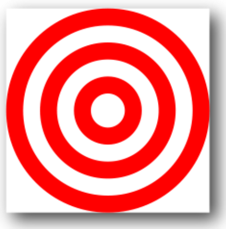
Optional: Your drawing doesn't have to be fancy or interesting, but we'll award a bonus of up to 5% for the best drawings, as judged by the ISAs and their subjective, personal tastes. If you would like to show your drawing to the ISAs (and be considered for this bonus), please upload an image of it in the file `example-drawing.png`, by following the instructions below on producing and saving an image inside of Racket.

(b) Write a function *cull* that simplifies a drawing. Specifically, it consumes a drawing and natural numbers m and n (in that order). It produces a new drawing containing the first m squares and the first n circles in the original drawing. If the original drawing contains fewer than m squares, keep all of them (and similarly for circles). The shapes in the resulting drawing should be in the same order that they appeared in the original, with some shapes potentially removed. You may want to use the drawing you created in (a) for testing here.

(c) Write a function *bullseye* that consumes a single natural number n and produces a *Drawing* containing n alternating red and white discs centred on the point $(100, 100)$. The innermost disc should always be white; this means that if n is odd, the outermost disc will be white as well, and therefore invisible against a white background. The rings should be of equal width, and you should choose their radii so that the whole drawing fits precisely in a 200×200 box. (It will help to think of *bullseye* as a wrapper function that computes the ring width and passes this number to a recursive helper function.) Here are a few examples:


`(bullseye 4)`


`(bullseye 5)`

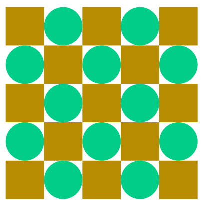

`(bullseye 6)`

The drop shadows in the images above are not part of the produced drawings; they are there to make sure you can see their bounds. The centre drawing is annotated with dotted lines so that you can see the outline of the invisible outer white ring.

Hint: you can decide whether to draw a red or white ring in a structurally recursive way by checking whether n is even or odd.

See below for information about examples and tests for this question.

(d) Write a function *checkerboard*. It consumes three values: a natural number n and two colours c_1 and c_2. It produces a checkerboard that alternates between two tile shapes: squares filled with colour c_1 and circles filled with colour c_2. Each of the tiles should have a width and height of 10, and the checkerboard should be a grid of n rows and n columns of tiles. The top-left element should be a square, with its corner at (0,0). For example, (*checkerboard* 5 '(184 140 0) '(0 206 136)) should look like this:



You may find the built-in function *append* useful for this question, but there are many possible solutions that do not use it. Hint: if we use x and y to denote the column number and row number of each grid cell respectively, then the choice of shape and colour to use at grid position (x, y) can be determined from whether $x + y$ is even or odd.

See below for information about examples and tests for this question.

In addition to the data definitions for *circle135* and *square135*, the file `drawinglib.rkt` contains two utility functions that will make this question more fun. *draw-picture* consumes a *Drawing* and two *Nat*s, a width and a height. It produces a real image of the drawing, which will be shown in the Interactions window. You can also use the function *save-image*, which consumes one of these images together with a file name (a string), to write a copy of the image (in PNG format) to the directory containing your Racket code. Do not use *draw-picture* or *save-image* in your submitted code in the Definitions window; they're only useful for testing and playing around in the Interactions window. We also provide constant definitions for a few standard colours.

Examples and tests for (c) and (d): The goal for (c) and (d) is really to produce images, not lists of data. The best way to test these functions is to use *draw-picture*, for example by typing this into the Interactions window:

```
(draw-picture (bullseye 5) 200 200)
```

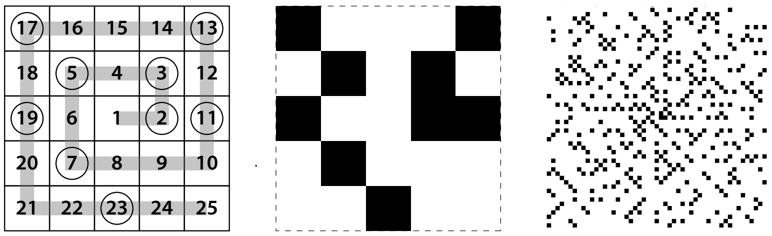
With that in mind, *you do not need to provide examples or tests for parts (c) and (d) of this question*. You must still provide contracts and purposes for *bullseye* and *checkerboard* and a full design recipe for *cull* (*example-drawing* is a constant and therefore does not require a design recipe). Of course, you are still welcome to include tests if they help you design correct code.

Place your solutions in the file `drawing.rkt` and submit that file. You should not submit `drawinglib.rkt`.

This concludes the list of questions for which you need to submit solutions. Always remember to check your basic test results by email or on MarkUs after making a submission.

---

4. **5% Bonus**: The Ulam Spiral is a classic visualization of the distribution of prime numbers. Place the number 1 in the centre of a grid. Move one step to the right and then wind around the centre in a counter-clockwise spiral, marking cells with successive integers. Finally, colour all the prime-numbered cells black and leave the rest white. The result will look like the visualization below, with a larger grid shown on the right.



Using the data definitions in Question 3, write a function called *ulam-spiral* that consumes two natural numbers: the first is an odd number that determines the number of rows and columns in the grid, and the second is the side length of the squares to draw for the black grid cells. The function produces a *Drawing* containing a black square of the given side length for each prime-numbered location in the spiral. Do not include explicit white squares for other grid cells—we'll simply let the image's white background show through.

Remember that you must use pure structural recursion in your solution.

Place your solution in the file `ulam.rkt`. Add the line (*require* "drawinglib.rkt") to the top of your file to gain access to the *Drawing* types and support code.

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)
⇒ (cond [(zero? 0) 0][else ...])
⇒ (cond [true 0][else ...])
⇒ 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*)
⇒ (**cond** [(*zero? n*) 0][**else** . . .]) ;(we know $n > 0$)
⇒ (**cond** [*false* 0][**else** . . .])
⇒ (**cond** [**else** (+ *n* (*sum-first* (*sub1 n*)))])
⇒ (+ *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).