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 at [https://www.student.cs.uwaterloo.ca/~cs115/built_in](https://www.student.cs.uwaterloo.ca/~cs115/built_in)

• 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 `a6qY.rkt`, where Y is a value from 1 to 3.

**Plagiarism: The Following applies to all assignments in CS115:**

• Be sure to read the Plagiarism section at: [https://www.student.cs.uwaterloo.ca/~cs115/assignments](https://www.student.cs.uwaterloo.ca/~cs115/assignments)
Assignment 6
Due: November 7th, 2018
Language Level: Beginning Student with List Abbreviations
Coverage: Modules 5, 6

Question 1

As a Racket programmer, you know about the importance of balancing your open and close parentheses. But in how many ways can this be done? For example, how many ways can you arrange three open and three close parentheses so that they nest in a valid way? As it happens, there are five ways to do this:

`[((())) ()(()) ()()() (())() (()())`  

A beautiful result in combinatorics shows that for any `n`, the number of valid ways to arrange `n` pairs of parentheses is given by `C_n`, the `n`th Catalan number. One way to compute it, for all `n ≥ 2`, is as follows:

\[ C_n = \frac{n + 2}{2} \cdot \frac{n + 3}{3} \cdot \frac{n + 4}{4} \cdot \ldots \cdot \frac{n + n}{n} \]

The first few Catalan numbers, starting from `n = 2`, are 2, 5, 14, 42, 132, 429, 1430, ...

Write a Racket function `catalan` that consumes a natural number `n ≥ 2` and produces the `n`th Catalan number. For example, `(catalan 2) ⇒ 2`. Hint: use one of the templates developed in class for counting over ranges of natural numbers. (Note that there are other formulas for computing Catalan numbers; please use the one given here.)
**Question 2**

You are probably familiar with simple mathematical formulas to calculate the area of a triangle or a rectangle. There is also an elegant formula, sometimes called the “shoelace formula”, to find the area of a general polygon. Let a polygon be described by a sequence of points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\). Then the area of the polygon is given by

\[
\frac{1}{2} \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i)
\]

In this formula, the subscripts “wrap around”: even though the point \((x_{n+1}, y_{n+1})\) is not defined above, we treat it as a copy of \((x_1, y_1)\). The formula even works for a “polygon” with only one or two points, in which case it produces an area of 0.

For example, consider the pentagon shown on the right. If we write down its vertices as the sequence \((0,0), (1,0), (2,1), (1,1), (0,2)\), then the area is given by

\[
\frac{1}{2} \left[ (0 \cdot 0 - 1 \cdot 0) + (1 \cdot 1 - 2 \cdot 0) + (2 \cdot 1 - 1 \cdot 1) + (1 \cdot 2 - 0 \cdot 1) + (0 \cdot 0 - 0 \cdot 2) \right],
\]

which is equal to 2.

If we travel around the polygon in the other direction, giving the list of points in the order \((0,0), (0,2), (1,1), (2,1), (1,0)\) the same formula will produce an area of \(-2\). Because the area can be positive or negative, we refer to it as the *signed area*.

Write a Racket function `signed-area` that consumes a polygon given by a non-empty `(listof Posn)` and uses the shoelace formula to produce the polygon’s area. Note that you will need to figure out how to carry a copy of the first point in the list “along for the ride” to the end so that it can play the role of \((x_{n+1}, y_{n+1})\).
Question 3

“Curling” is a fake sport that Canadians and Norwegians pretend to play in order to confuse the rest of the world. In each round or “end”, two teams take turns propelling stones down the ice towards a bullseye-shaped “house”. The team that has a stone closest to the “button” (the centre of the house) then receives one point for each stone of theirs closer to the button than the other team’s closest stone. The other team receives zero points for that end.

For example, in the diagram on the left, yellow has the closest stone, which overlaps the white centre of the house. The closest red stone is touching the lower-left part of the red ring. There are two yellow stones closer than that red stone, so Yellow receives two points for this end. The yellow stone in the blue ring doesn’t count for any points, because it’s farther away than the closest red stone.

When an end concludes, we can build a list of all the stones in the house, where each stone has a colour (red or yellow) and a distance to the button:

\[
\text{(define-struct stone (colour distance))}
\]

\[
\text{;; A Stone is a (make-stone (anyof “red” “yellow”) Num)}
\]

\[
\text{;; Requires:}
\]

\[
\text{;; * distance \(\geq 0\)}
\]

Given a list of stones, we can then tabulate the score for the end, noting which team received the points and how many points they received:

\[
\text{(define-struct score (colour points))}
\]

\[
\text{;; A Score is a (make-score (anyof “red” “yellow”) Nat)}
\]

(a) Write a Racket function \text{sort-stones} that consumes a \text{(listof Stone)} and produces a new list containing the same stones, sorted from smallest distance to largest, regardless of colour. For example:

\[
\text{(sort-stones}
\text{ (cons (make-stone “red” 2) (cons (make-stone “yellow” 1) empty)))}
\]

\[
\Rightarrow (\text{(cons (make-stone “yellow” 1) (cons (make-stone “red” 2) empty))})
\]

You can assume that the list contains no ties, i.e., all the distances are distinct.
(b) Write a Racket function curling-score that consumes a (listof Stone) of any positive length and produces a Score that records which team won the end and how many points they received. For example:

```
(curling-score
  (cons (make-stone "yellow" 7)
       (cons (make-stone "red" 6)
            (cons (make-stone "yellow" 5)
                 (cons (make-stone "red" 8) empty))))
⇒ (make-score "yellow" 1)
```

You can assume that the list contains no ties. Your curling-score function is likely to be a small wrapper around a recursive function that operates on a sorted version of the original list.