Homework 4

Due: Saturday, October 26th, 11pm

While problem sets 1–3 focused on learning to debug, the next few will focus on how to write code in such a manner that debugging occurs much less frequently (ideally, not at all) because the code we write is known to be correct *before* we run it for the first time.

With that goal in mind, the next few assignments will be split into separate written and coding parts, where the written part is to be completed *before* starting on the coding part.

This worksheet contains only the *written* parts of HW4; the coding instructions will be released Thursday (10/24) and be due Monday (10/28).

Submission

After completing all parts below, submit your solutions on Gradescope. The collection of all written answers to problems described in this worksheet should be submitted as a PDF to **"HW4 Written"**.

Don't forget to check that the submitted file is up-to-date with all written work you completed!

You may handwrite your work (on a tablet or paper) or type it, provided it is legible and dark enough to read. When you turn in on Gradescope, please match each HW problem to the page with your work on it. If you fail to have readable work or assign pages, you will receive a point deduction.

Task 1 – Off the Beaten Math [16 pts]

For each of the following functions, translate the code into a function definition written in our math notation, using pattern matching. Unless the comments above the code say otherwise, you *can* assume that any value of the declared TypeScript type is allowed by the function.

Make sure your rules are *exclusive* and exhaustive!

(a)
```
const a = (o: boolean, t: [bigint, bigint]): [bigint, bigint] => {
  const [i, j]: [bigint, bigint] = t;
  if (o) {
    return [i + 1n, j];
  } else {
    return [j + 1n, i];
  }
};
```

(b)
```
const b = (s: bigint, o: boolean): List<bigint> => {
  if (o) {
    return cons(s, cons(-s, nil));
  } else {
    return cons(-s, cons(s, nil));
  }
};
```

(c)
```
// s and t allow only non-negative integers
const c = (r: {s: bigint, t: [bigint, bigint]}): bigint => {
  const [i, j]: [bigint, bigint] = r.t;
  if (r.s === 0n) {
      return i;
  } else if (r.s === 1n) {
      return i + 1n;
  } else {
      return j + c({s: r.s - 1n, t: r.t});
  }
};
```

Task 2 – Pell on Wheels [14 pts]

In this problem, we will practice a skill that you will use a lot in coming assignments. We'll start with an English description and formalize that description into a mathematical definition which could serve as a specification if we were to write this in TypeScript.

We'll be converting an English description of a function that calculates **Pell numbers** to math notation. We'll denote a Pell number as $p(n)$ for some non-negative integer n. We define $p(0) = 0$ and $p(1) = 1$. After that, we define $p(n)$ to be the sum of twice the previous Pell number and the Pell number before that. With that definition in hand, our goal is to write a function "$s(n)$" that gives the sum of the *first* n Pell numbers. For example, $s(2) = p(0) + p(1) = 1$.

For reference, here are the first five Pell numbers and their sums:

n	$p(n)$	$s(n)$
0	0	0
1	1	0
2	2	1
3	5	3
4	12	8
5	29	20

(a) The description above is in English, so our first step is to formalize it into a math notation.

Define two mathematical functions, "p" and "s", both taking non-negative integers as input. Define each function recursively with pattern matching.

(b) Show how your mathematical definition would execute $s(6) = 49$ by writing out the sequence of recursive calls. Include the arguments and what is returned for each recursive call.

Use any sensible notation to clearly show the sequence of calls. (If a call is made a second time, you can reuse the result of that call without showing all recursive calls that would be made.)

Task 3 – Good To the Last Swap [14 pts]

In this problem, we will create some functions that allow us to encode and decode secret string messages.

Our messages will be represented as lists of integers, where each integer is in the range 0–25 and the integer j represents the j-th Latin letter. For example, 'a' = 0, 'b' = 1, and 'z' = 25.

Let's start by discussing how we encode individual characters.

We will say that "negating" a range of characters means swapping the j-th character from the beginning with the j-th character from the end. For example, negating the range 0–25 turns 'a' (0) into 'z' (25), 'c' (2) into 'x' (23), etc.

Note that negating a range twice gets us back to where we started, so if we negate ranges as our cipher, we can write one function that both encodes and decodes!

Our encoding function will be a little more complicated than just that. It will negate the four ranges consisting of the first 6 characters (0–5), the next 7 characters (6–12), the next 6 characters (13–18), and last 7 characters (19–25), each individually. After negating these four ranges, it will also swap the first and third ranges and the second and fourth ranges. (Repeating a swap also gets us back to where we started, so again this one function will both encode and decode.)

To encode a message, which is a list of characters, we apply the character encoding individually to each character in the list.

(a) Above, we were given an English definition of the problem, so our first step is to formalize it.

The function "ns" will take an integer the range 0–25 as input and returns the value produced by *negating and swapping* ranges as described above. For integer values outside the range 0–25, we define ns to leave those unchanged.

(b) Now, given "ns" defined in part (a), formalize "cipher", which encodes messages.

Write a formal definition using recursion. Assume that the mathematical function "ns" defined in part a) correctly implements the behavior described above.

Before we can get to the next two problems, we need the following mathematical definitions.

Blocks

In this assignment, we will write some math that displays mazes. Each maze is made up of Blocks. Mathematically, each block is a record of the following type:

$$\begin{aligned}\textbf{type}\ \text{Block}\ :=\ &\{\text{form} : \text{STRAIGHT},\ \text{color} : \text{Color},\ \text{direction} : \text{Line}\}\\ &\mid \{\text{form} : \text{ANGLED},\ \text{color} : \text{Color},\ \text{direction} : \text{Corner}\}\end{aligned}$$

Individual Blocks include a color property, which are elements of the following type:

$$\textbf{type}\ \text{Color}\ :=\ \text{BLUE} \mid \text{ORANGE}$$

Blocks also include a direction property that describes how the block is oriented (i.e. how it is rotated). direction is defined with *different types* depending on the form property of the block.

STRAIGHT Blocks contain a straight line that spans the block in 1 direction, either top to bottom (a vertical line), or right to left (a horizontal line). This is defined with the following type:

$$\textbf{type}\ \text{Line}\ :=\ \text{TB} \mid \text{RL}$$

ANGLED Blocks contain a line that starts at one side of the block, angles, and exits at another. ANGLED block directions are described as the corner of the square created by the angle within the block. This is defined with the following type:

$$\textbf{type}\ \text{Corner}\ :=\ \text{TR} \mid \text{TL} \mid \text{BR} \mid \text{BL}$$

Mazes

A maze is a 2D table of blocks. We will represent each maze as a list of lists of blocks. We will call a list of blocks a "row", and then a maze is a list of rows. As mentioned in the last problem, our current List type has integer elements, so to express rows and mazes we will define these two new types inductively as follows:

$$\begin{aligned}\textbf{type}\ \text{Row} :=&\quad \text{rnil} &\mid&\quad \text{rcons}(\text{hd} : \text{Block},\ \text{tl} : \text{Row})\\ \textbf{type}\ \text{Maze} :=&\quad \text{mnil} &\mid&\quad \text{mcons}(\text{hd} : \text{Row},\ \text{tl} : \text{Maze})\end{aligned}$$

All rows in a maze should have the same length. Mathematically, we define the function $\text{rlen} : \text{Row} \to \mathbb{N}$ defines the length of a row by:

$$\begin{aligned}\text{rlen}(\text{rnil}) &:= 0\\ \text{rlen}(\text{rcons}(a, L)) &:= 1 + \text{rlen}(L)\end{aligned}$$

Note, however, that our type definitions allow the maze to contain rows of different lengths! It is an additional *invariant* of the Maze type that all rows in each maze must have the same length.

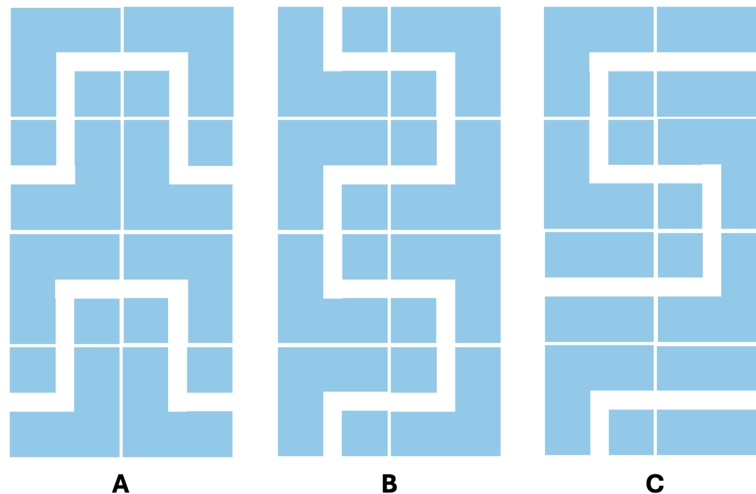
We can also define concatenation of rows. The function $\text{rconcat} : (\text{Row}, \text{Row}) \to \text{Row}$ is defined by:

$$\begin{aligned}\text{rconcat}(\text{rnil}, R) &:= R\\ \text{rconcat}(\text{rcons}(s, L), R) &:= \text{rcons}(s, \text{rconcat}(L, R))\end{aligned}$$

These two functions, whose names start with "r", are defined on lists of blocks (rows). There are analogous definitions of functions, mlen and mconcat, whose names start with "m", that operate on lists of rows (mazes).

Task 4 – Mazed and Confused [14 pts]

With these definitions, we can create mazes like those pictured below. Someone trying to make their way through these small, simple mazes perhaps wouldn't get lost, but you can imagine that these simple designs could be composed to make more complex mazes.



(a) Our first exercise is formalizing the maze designs above by using the definitions of the Block and Maze types from the previous page. Write mathematical definitions for functions "mazeA", "mazeB", and "mazeC" which create a 2×4 maze matching those shown above. Write them in the most straight-forward manner –no loops or recursion!

Additionally, have these mazes accept one parameter, an argument of type Color, which will determine whether the blocks of the maze should be BLUE (as shown above) or ORANGE.

(b) Next, we'll define recursive functions that repeat these designs to allow for creating larger mazes.

Your mathematical definitions should accept an argument, n, which is a natural number defining the number of rows the design should have, as well as the same color parameter as before. Your functions should have 1 recursive case, in which rows of blocks should be added to the result, and 1 base case.

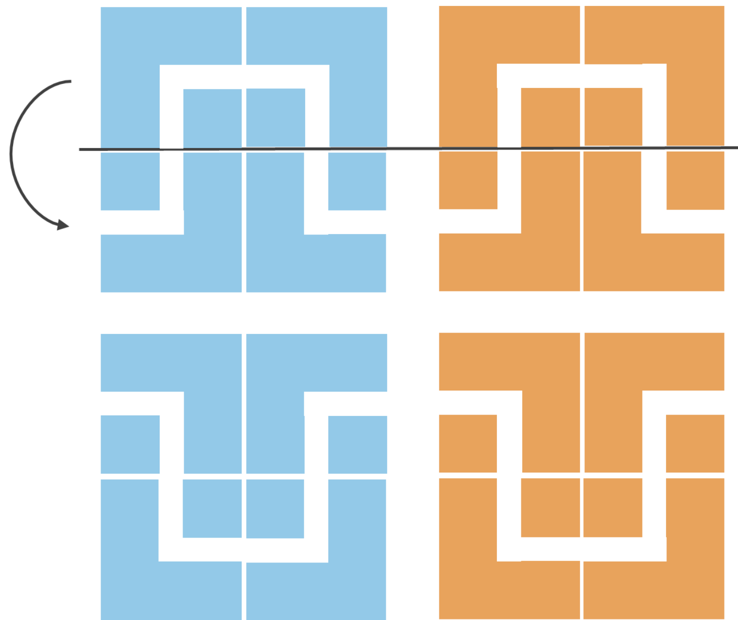
Maze A and maze B repeat every 2 rows. Maze C repeats every 3 rows, where the first 3 rows of the design above are those to repeat (you can see the pattern start again in the forth row, and you can assume it continues by repeating the second row next and then the third).

In other words, maze A and maze B are defined for any **even** number of rows while Maze C is defined on numbers of row that are **multiples of 3**. This should be useful insight for how to define your recursive cases, and you can assume this restriction on allowed inputs is part of the specification for these functions visible to users.

Task 5 – My Flips are Sealed [18 pts]

In this problem, we will write a function that "flips a maze vertically, as if spun around a horizontal line through the center".

Here is an example (the bottom maze is the result of vertically flipping the top maze around the center line):



(a) The problem definition was in English, so our first step is to formalize it.

Start by writing a mathematical definition of a function "bflip-vert" that flips a **block** vertically.

(b) Next, we will define a mathematical function "rflip-vert" that flips a **row** vertically.

Let's start by writing this out in more detail. Let a, b, and c be blocks. Fill in the blanks showing the result of applying rflip-vert to different rows, which we will write as lists of blocks.

Feel free to abbreviate bflip-vert in your answer as "bv".

rnil ____________________

rcons(a, rnil) ____________________

rcons(a, rcons(b, rnil)) ____________________

rcons(a, rcons(b, rcons(c, rnil))) ____________________

...

(c) Write a mathematical definition of a function rflip-vert using recursion.

(d) Now, we are ready to define a function "mflip-vert" that flips a **maze** vertically. Note that this operation flips individual rows vertically, but also *reverses* their order!

Again, let's start by writing this out in more detail. Let u, v, and w be rows. Fill in the blanks showing the result of applying mflip-vert to different mazes, which we will write as lists of rows.

Your answers should use rflip-vert (not bflip-vert), which you can abbreviate as just "r".

mnil ____________________

mcons(u, mnil) ____________________

mcons$(u, \text{mcons}(v, \text{mnil}))$ ____________________

mcons$(u, \text{mcons}(v, \text{mcons}(w, \text{mnil})))$ ____________________

...

(e) Write a mathematical definition of a function "mflip-vert".

Hint: it may be useful to review definition of the function rev, for reversing a list, which is defined in the notes on lists posted on the website. Also, remember that the function mconcat, which concatenates two mazes, is already provided for you.