CSE 331: Software Design & Implementation Spring 2025

Homework 4

Due: Wednesday, April 30th, 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 have written and coding components, where the written part is to be completed *before* starting on the coding part.

This worksheet contains the *written* and *coding* parts of HW4.

Written

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 [12 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 x = (a: bigint, b: boolean): List<bigint> => {
  if (!b) {
    return cons(-a, cons(a, nil));
  } else {
    return cons(a, cons(-a, nil));
  }
};
```

(b)
```
const y = (c: [bigint, bigint], b: boolean): [bigint, bigint] => {
  const [i, j]: [bigint, bigint] = c;
  if (b) {
    return [j + 1n, i];
  } else {
    return [i - 1n, j];
  }
};
```

(c)
```
// c and a allow only non-negative integers
const z = (d: {c: [bigint, bigint], a: bigint}): bigint => {
  const [i, j]: [bigint, bigint] = d.c;
  if (d.a === 0n) {
      return i * 2n;
  } else if (d.a === 1n) {
      return i;
  } else {
      return j + z({c: d.c, a: d.a - 1n});
  }
};
```

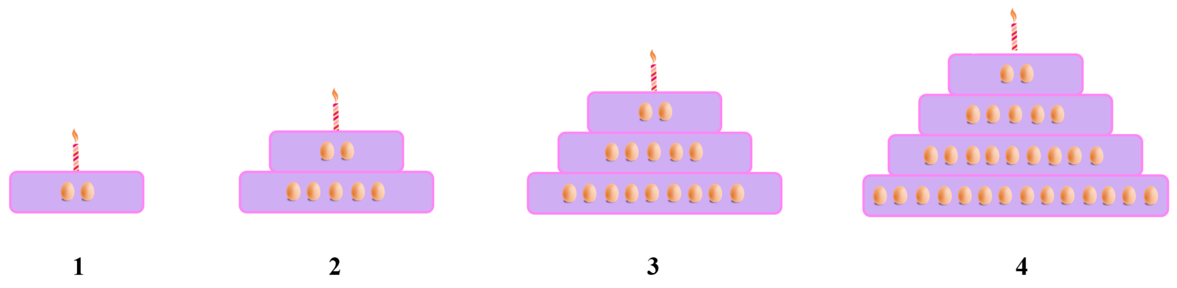
Task 2 – A Princess's Poor Egg-conomy [12 pts]

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.

Princess Peach is planning a party for her friends and wants to bake an amazing cake. Unfortunately, there's been a recent phenomenon where eggs hatch into Yoshis, regardless of the original egg type. Because the Nintendo world has been hit with this Yoshi-dino flu, an egg shortage epidemic has occurred. Princess Peach is worried that her cake cannot be made to correct size or will cost too much. Still, she wants to make a cake as her party's showstopper. Help Princess Peach figure out how many layers of cake she can bake!

To do this, we'll be converting an English description of a function that calculates **Cake-layer numbers** to math notation. We'll denote a Cake-layer number as $c(n)$ for some non-negative and non-zero integer n, where $c(n)$ is number of eggs it takes to make n layers of cake. According to Princess Peach's eggs-travagant cake recipe, each additional layer takes just as many eggs as the previous layer, but needs 1 more egg plus its layer number n on top of the previous layer's egg quantity. This means for the bottom layer of the 3 layer cake, we get its previous layer's egg quantity (5) plus it's layer number of (3) plus (1) which sums up to 9. This way, the cake will be nicely tiered like the image below shows, instead of stacking to a straight tower. Therefore, we define $c(1) = 2$ and $c(2) = 5$, where for 1 layer of cake, 2 eggs are needed and for 2 layers of cake, 5 eggs are needed. We define $g(n)$ to be the total number of eggs for n layers of cakes. With that definition in hand, our goal is to write a function "$g(n)$" that gives the sum of the *first* n Cake-layer numbers. For example, $g(2) = c(1) + c(2) = 7$.

For reference, here are the first four layers of the Cake-layer sequence and their eggs (candles are decorative and negligible):



n	$c(n)$	$g(n)$
1	2	2
2	5	7
3	9	16
4	14	30

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

Define two mathematical functions, "c" and "g", both taking non-negative and non-zero integers as input. This has the type $\mathbb{N}^+$. Define each function recursively with pattern matching.

(b) Princess Peach has decided that a 3-tiered cake, each tier having 3 layers, would be perfect. Show how your mathematical definition would execute $g(9) = 210$ 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 – I Toad Ya So [12 pts]

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

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

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

Introducing Toad Cipher. To encode a single character within the cipher we have a few cases. If we reach an i-th character representing a Latin letter in "toad" then we replace that ith character with the next latin letter in "toad". For example, 'd' becomes 't' after being encoded.

If we don't land on an i-th character representing a latin letter in "toad" then the i-th character will be negated over the range. We will say that "negating" the i-th character means swapping the i-th character with the i-th character from the end of our range (excluding the latin letters in "toad"). For example, negating turns 'b' (1) into 'z' (25), 'y' (24) into 'c' (2) etc.

As an example the list [0,1,2] is translated to [3, 25, 24] by Toad Cipher:

- 0 (representing a) becomes 3 (representing d) as a is in "toad" meaning we go to the next latin character
- 1 (representing b) becomes 25 (representing z), as b is not in "toad", z is the last letter in the latin alphabet, and b is the first letter in the latin alphabet that is not in "toad"
- 2 (representing c) becomes 24 (representing y), as c is not in "toad", y is the second to last letter in the latin alphabet, and c is the second letter in the latin alphabet that is not in "toad"

To encode a message, which is a list of character indices, 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 "td" will take an integer within the range 0–25 as input and returns the value produced by following the encoding rules as described above. For integer values outside the range 0–25, we define td to leave those unchanged.

(b) Now, given "td" defined in part (a), formalize the "Toad Cipher", which encodes messages.

Write a formal definition using recursion. Assume that the mathematical function "td" 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 pipes. Each pipe is made up of blocks. Mathematically, each block is a record of the following type:

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

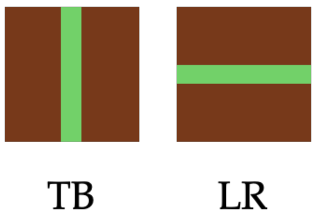
Individual Blocks include a color property, which are elements of the following type:

$$\textbf{type } \mathsf{Color} \ := \ \mathsf{DIRT} \mid \mathsf{WALL}$$

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.

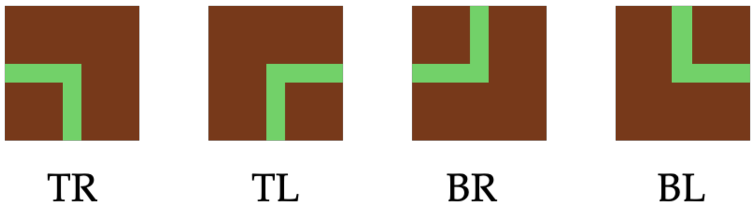
STRAIGHT Blocks contain a straight line that spans the block in 1 direction, either top to bottom (a vertical line), or right to left (a horizontal line). This is defined with the following type:

$$\textbf{type } \mathsf{Line} \ := \ \mathsf{TB} \mid \mathsf{LR}$$



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 (for intuition on which label corresponds to which block, think of where the "arrow" points to). This is defined with the following type:

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



Pipes

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

$$\begin{aligned}
&\textbf{type}\ \text{Row} := \quad \text{rnil} \quad | \quad \text{rcons}(\text{hd} : \text{Block},\ \text{tl} : \text{Row})\\
&\textbf{type}\ \text{Pipe} := \quad \text{pnil} \quad | \quad \text{pcons}(\text{hd} : \text{Row},\ \text{tl} : \text{Pipe})
\end{aligned}$$

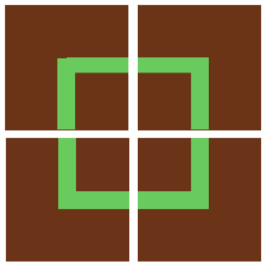
All rows in a pipe should have the same length. Mathematically, we define the function
rlen : Row → ℕ defines the length of a row by:

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

Note, however, that our type definitions allow the pipe to contain rows of different lengths! It is an additional *requirement* of the pipe type that all rows in each pipe must have the same length.
We can also define concatenation of rows. The function rconcat : (Row, Row) → Row is defined by:

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

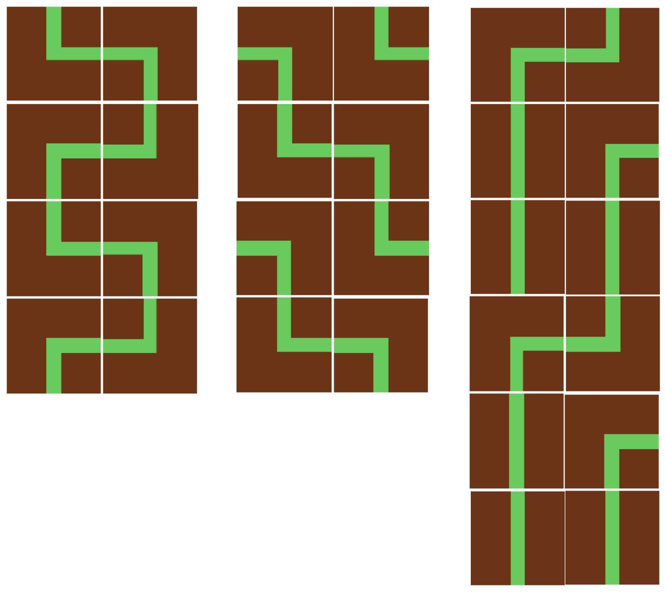
These two functions, whose names start with "r", are defined on lists of blocks (rows). There are analogous definitions of functions, plen and pconcat, whose names start with "p", that operate on lists of rows (pipes).



This pipe has the structure as follows:
pcons(rcons(tl, rcons(tr, rnil)), pcons(rcons(bl, rcons(br, rnil)), pnil)) where
tl = {form: ANGLED, color: DIRT, direction: TL}
tr = {form: ANGLED, color: DIRT, direction: TR}
bl = {form: ANGLED, color: DIRT, direction: BL}
br = {form: ANGLED, color: DIRT, direction: BR}

Task 4 – Piper, No Piping! [14 pts]

With these definitions, we can create pipes like those pictured below. If Mario and Luigi were traveling through these small, simple pipes they probably wouldn't get lost, but you can imagine that these simple designs could be composed to make more complex pipes.



(a) Our first exercise is formalizing the pipe designs above by using the definitions of the Block and Pipe types from the previous page. Write mathematical definitions for functions "pipeA", "pipeB" which creates 4 × 2 (rows × cols) pipes, and "pipeC" which creates a 6 × 2 pipe matching those shown above. Write them in the most straight-forward manner –no loops or recursion!

Additionally, have these pipes accept one parameter, an argument of type Color, which will determine whether the blocks of the pipe should be DIRT (as shown above) or WALL.

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

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.

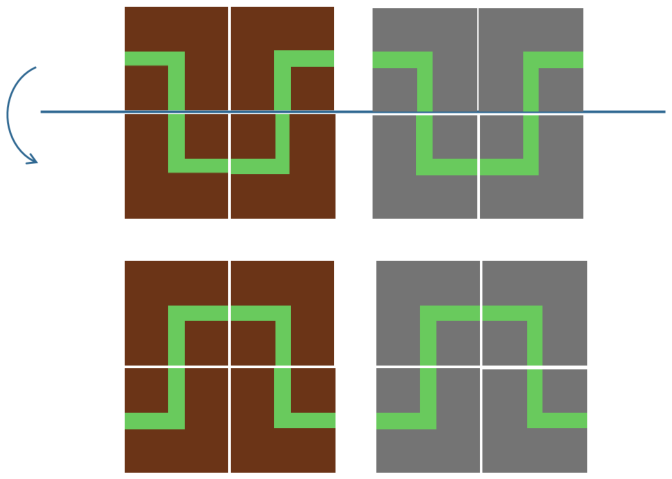
Pipe A and pipe B repeat every 2 rows. Pipe C repeats every 3 rows, where the first 3 rows of the design above are those to repeat (you can see the pattern start again in the fourth row, and you can assume it continues by repeating the second row next and then the third).

In other words, pipe A and pipe B are defined for any **even** number of rows while pipe 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 [16 pts]

In this problem, we will write a function that "flips a pipe vertically, as if mirrored across a horizontal line through the center".

Here is an example (the bottom pipe is the result of vertically flipping the top pipe over 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" that flips a **block** vertically.

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

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

Feel free to abbreviate bflip in your answer as "b".

rnil	____________________
rcons(d, rnil)	____________________
rcons(d, rcons(e, rnil))	____________________
rcons(d, rcons(e, rcons(f, rnil)))	____________________

...

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

(d) Now, we are ready to define a function "pflip" that flips a **pipe** vertically.

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 pflip to different pipes, which we will write as lists of rows. Note that this operation flips individual rows vertically, and also *switches* the order of the rows!

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

pnil	____________________
pcons(u, pnil)	____________________
pcons(u, pcons(v, pnil))	____________________
pcons(u, pcons(v, pcons(w, pnil)))	____________________
...	

(e) Write a mathematical definition of a function "pflip".

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 pconcat, which concatenates two pipes, is already provided for you.

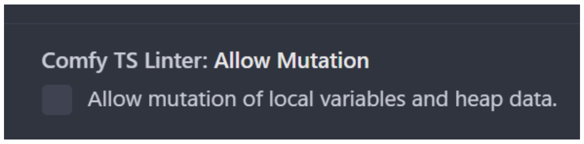
Coding

To get started, check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25sp/materials/hw4-pipes.git
```

Navigate to the `hw4-pipes` directory and run `npm install --no-audit`. For this assignment, we have provided (and will ask you to write) unit tests. To run the tests, use the command `npm run test`. To run the linter, use `npm run lint`.

This assignment (and the next few) **does not allow mutation**. To disallow mutation with the VS-Code extension, follow similar steps as HW3. Open the extension, select the gear icon, open "Settings", and uncheck the checkbox to no longer allow mutation. The linter settings should look like this:



Comfy TS Linter: **Allow Mutation**

☐ Allow mutation of local variables and heap data.

Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW4 Code"**:

`designs.ts`   `pipe_ops.ts`   `App.tsx`   `Viewer.tsx`   `funcs_test.ts`

After you submit your work, an autograder will run which verifies you have submitted the correct files, runs the linter, and runs tests (including those you submit, the tests we provide in the starter code, and some additional staff tests). As usual, the autograder is worth points, so you should wait until the autograder completes to make sure it passes, and otherwise resubmit. Meaning you should **leave enough time** to fix possible issues before the deadline. As usual, we will also manually grade your code (including test cases).

Task 6 – It's Simply Design [22 pts]

In the first part of this problem, we will translate mathematical definitions for functions into TypeScript code. Then, we will complete a client side app that utilizes those functions.

When translating, we will treat the math definitions as imperative specifications for the TypeScript functions, so the translations should be "straight from the spec" –a direct translation.

We have provided tests for these functions based on their correct behavior as described in the English/picture descriptions from the written tasks. You should run these tests to get a good idea of if your implementations are correct using the command `npm run test`.

If the tests fail, indicating a bug in your TypeScript functions, you should fix these bugs to try to get the tests to pass.

(a) Translate your mathematical definitions from HW4 Task 4(b), the recursive functions for each pipe design, into TypeScript code in `designs.ts`.

You should only translate the *recursive* definitions that you wrote in part **(b)**, you *do not* need to translate the 2×4 / 2×6 versions from part (a). Please maintain the order of the parameters as given in those declarations, even if it deviates from the order in your mathematical definition, as it is required for testing.

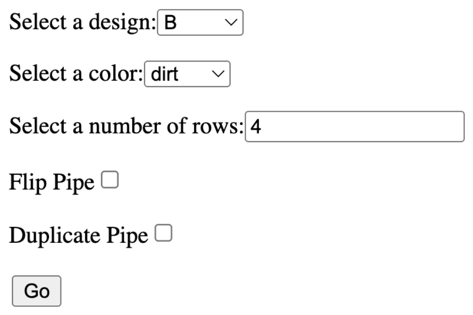
The types and helper functions related to pipes (as described in the HW4 Written spec) are translated to TypeScript for you in `pipe.ts`; import those to `designs.ts` and use as needed. The provided tests for these functions are in `designs_test.ts`.

Make sure to complete the TODO in the comment of each recursive pipe function to copy over your math definition from Task 4.

(b) Translate your mathematical definitions from HW4 Task 5 **(a)**, **(c)**, and **(e)**, the functions for each type of flip, into TypeScript code in `pipe_ops.ts`. The provided tests for these functions are in `pipe_ops_test.ts`.

Again, make sure to complete the TODO in the comment of each flip function to copy over your math definition from Task 5.

With these interesting pipe functions in hand, we can use them in a super cool app. You can run the app with the command `npm run start`. Currently, you will get lots of errors about unused variables and other problems (since the app is incomplete), but after this next part, you'll have a pipe designer app where you can design A, B and C pipes with different colors, numbers of rows, and flips.


Select a design: B

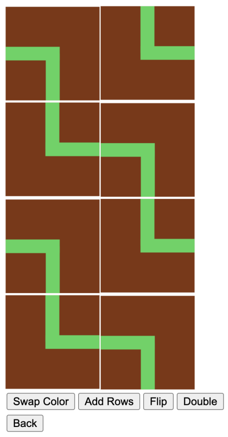
Select a color: dirt

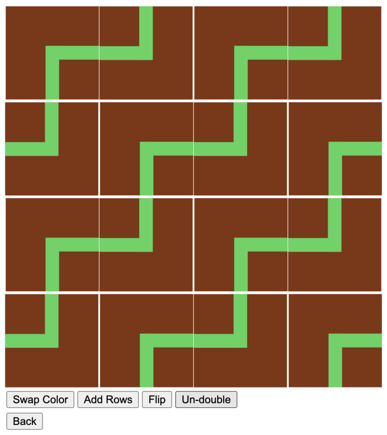
Select a number of rows: 4

Flip Pipe ☐

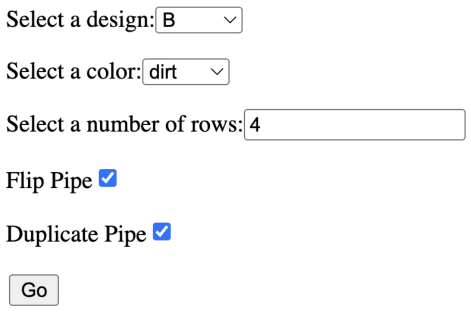
Duplicate Pipe ☐

It will display the designed pipe in an "experimentation" view, where you can continue to adjust the pipe design. For example, the image on the left is the result of clicking "Go" with the selections shown in the image on the previous page, and the image on the right is the result of clicking "Flip" and "Double" on that initial design.





Importantly, we want users to be able to maintain their experiment settings and see them reflected in the initial input areas when they go "Back", as so:


Select a design: B

Select a color: dirt

Select a number of rows: 4

Flip Pipe ✓

Duplicate Pipe ✓

Go

(c) We have provided starter code that lays out the components we will use, constructs all the html for the buttons and input areas, and handles rendering all the pipe blocks. It is your job to fill in some gaps that will allow the input page, `App`, and the pipe viewing page, `Viewer`, to communicate with each other.

Familiarize yourself with the existing code in both components, and make note of the TODO comments which outline where you will need to add some code. If you have any questions about the existing code, it's a good idea to ask on Ed or in OH before writing any code yourself.

`App` contains states that reflect the values in each of the input areas on the screen. It also has a state `renderPipe` which it uses to determine whether the initial input page or the pipe display page should be rendered. It also has some error handling to make sure users can only create well-formed pipes (e.g. prevents pipes with no color, or a design C pipe with an even number of rows).

`Viewer` is responsible for rendering the pipe pattern specified which is done with a custom `pipeElem` component which takes a `pipe` as a prop specifying the design to create. We provided this custom component for you, and you are not required to understand how it works, but feel free to check it out. In order to create a `pipeElem`, the `Viewer` needs to have access to all the necessary attributes of the pipe to create.

Currently, the `Viewer` has states for some pipe attributes, associated with the experimentation buttons, which it updates when they are clicked, but they are initialized with hardcoded values instead of the real values that the user inputted on the initial page.

Once a user is done experimenting, they need to be able to click "Back" to return to the main page and see all the up-to-date inputs for the pipe they were just experimenting with. Currently, the `Viewer` just prints out a TODO message when the "Back" button is clicked, and there is no way for the `App` component to see the updates to the pipe design that have been made while experimenting.

To fix these problems, we will need to define some **props** passed from the parent component, `App`, to the child component, `Viewer`. Recall that props both allow parent components to send data to their child components, and allow child components to send data back to their parent through **callbacks**.

Specifically, complete the following steps:

- Initialize `Type ViewerProps` in `Viewer.tsx` with the necessary props that `App` needs to pass to `Viewer`, so the pipe attributes can be initialized properly.
- Pass the necessary props to `Viewer` from `App`.
- Complete `renderPipe` in `Viewer.tsx` to generate the `Pipe` using the appropriate pipe attributes.
- Update `doAddRowsClick` so the "Add Rows" button works for each design.
- Write a handler method for the `onClick` of the "Back" button in `Viewer`. Guarantee that clicking "Back" returns to the input page and that every input area reflects the state of the pipe when "Back" is clicked.

It may make sense to do these in a different order, and you may need to make changes in areas not explicitly mentioned here!

Once you think your app is complete, make sure you test it thoroughly! Unlike parts a-b which we can test with unit tests, it is easiest and most effective to test an app by using it and visually inspecting the results.

Congratulations! At this point you should have a super cool, functioning, pipe design app!

Task 7 – Test Friends Forever [12 pts]

Now that you've written some TypeScript code and tested it, it is your turn to write some tests!

In funcs.ts, there are 9 functions that you must write tests for. Your tests should follow the testing requirements we have described in lecture and in our testing notes summary(also linked on the website "Topics" page).

Write your tests for each function in funcs_test.ts.

Additionally, write short labels describing which coverage requirement is met by each test. We've written up some tips on using Mocha on the course website. For a direct example see below:

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
}
```

In test file:

```
it("twice", function() {
  // Statement coverage: [] executes 1st return, [3] executes 2nd
  assert.deepStrictEqual(twice(nil), nil);
  assert.deepStrictEqual(twice(cons(3, nil)), cons(6, nil));

  // Branch coverage: covered above, [] executes 1st branch, [3] executes 2nd

  // Loop/recursion coverage, 0 case: covered above by []
  // Loop/recursion coverage, 1 case: covered above by [3]

  // Loop/recursion coverage: many case
  assert.deepStrictEqual(twice(cons(1, cons(2, cons(3, nil)))),
    cons(2, cons(4, cons(6, nil))));
});
```

Notice how you don't need to add additional tests if previous tests cover multiple requirements, just make sure there are clear comments for the required coverage areas. You are welcome to organize your comments differently than the example or use different wordings, just make sure all necessary details are conveyed.