Homework 8

Due: Wednesday, March 5th, 11pm

Written

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW8 Written**". Don't forget to check that the submitted file is up-to-date with all written work you completed!

Make sure your work is legible and scanned clearly if you handwrite it, or compiled correctly if you choose to use LaTeX. Match each HW problem to the page with your work when you turn in. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

Squares

In this written portion, we will define a tree-like type "squares" mathematically and define operations we will use to edit them.

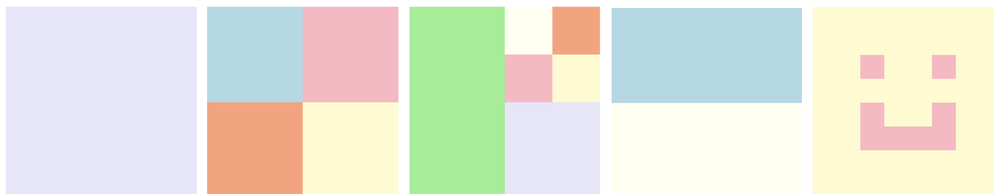
First we define some inductive types. First, a Color is one of the following:

$$\textbf{type}\ \mathsf{Color} \quad := \quad \mathsf{pink} \mid \mathsf{orange} \mid \mathsf{yellow} \mid \mathsf{green} \mid \mathsf{blue} \mid \mathsf{purple} \mid \mathsf{white}$$

Then, a Square is a tree, where each node is either a "solid" (leaf) node of a single color or a "split" (internal) node that breaks the square into four quadrants, each of which can be any square:

$$\begin{aligned} \textbf{type}\ \mathsf{Square} \quad := & \quad \mathsf{solid}(\mathsf{color} : \mathsf{Color}) \\ \mid & \quad \mathsf{split}(\mathsf{nw} : \mathsf{Square},\ \mathsf{ne} : \mathsf{Square},\ \mathsf{sw} : \mathsf{Square},\ \mathsf{se} : \mathsf{Square}) \end{aligned}$$

Using squares, we can use the recursive constructor to create patterns and pictures. For example, the following images all represent squares:



We will also need a way to refer to one of the children of a split square. We can do so as follows:

$$\textbf{type}\ \mathsf{Dir} \quad := \quad \mathsf{NW} \mid \mathsf{NE} \mid \mathsf{SW} \mid \mathsf{SE}$$

With that, we define a path to be a list of directions, describing how to get to the node, starting from the root:

$$\textbf{type}\ \mathsf{Path} \quad := \quad \mathsf{List}\langle \mathsf{Dir} \rangle$$

The following function, find retrieves the subtree at the given path:

$$\text{find} : (\text{Path}, \text{Square}) \to \text{Square}$$

$$\begin{aligned}
&\text{find}(\text{nil},\ S) && := S \\
&\text{find}(\text{NW} :: L,\ \text{split}(\text{nw}, \text{ne}, \text{sw}, \text{se})) && := \text{find}(L,\ \text{nw}) \\
&\text{find}(\text{NE} :: L,\ \text{split}(\text{nw}, \text{ne}, \text{sw}, \text{se})) && := \text{find}(L,\ \text{ne}) \\
&\text{find}(\text{SW} :: L,\ \text{split}(\text{nw}, \text{ne}, \text{sw}, \text{se})) && := \text{find}(L,\ \text{sw}) \\
&\text{find}(\text{SE} :: L,\ \text{split}(\text{nw}, \text{ne}, \text{sw}, \text{se})) && := \text{find}(L,\ \text{se}) \\
&\text{find}(x :: L,\ \text{solid}(c)) && := \text{undefined}
\end{aligned}$$

The final case in this definition describes when the given square does not have a subtree as described by the given path; in other words, there is a dead-end where the square is solid, meaning there are no subtrees to make recursive calls on, despite the path including more steps.

The undefined output here indicates that this case is invalid. We assume, here, that if an expression includes a call to this function which results in undefined, then the *entire expression is undefined*. You can think of an undefined result like throwing an Error in TypeScript; the undefined value is not really returned, but rather the original call to find results in undefined.

Task 1 – Couldn't Square Less [20 pts]

(a) Define a mathematical function

$$\text{replace} : (\text{Path}, \text{Square}, \text{Square}) \to \text{Square}$$

that returns the second Square (the last argument to the function) except with the subtree at the given path replaced by the first Square (the middle argument).

Like find, you can describe the case where the path is invalid for the given subtree with an undefined result.

(b) As a simple check, prove by calculation that $\text{find}(\text{nil}, \text{replace}(\text{nil}, T, S)) = T$.

(c) As a better check, prove that $\text{find}(L, \text{replace}(L, T, S)) = T$ (or $=$ undefined) for all paths L and square S and T. Your proof should be by structural induction **on the Square S**.

Feel free to cite the fact that you proved in part(b).

Recall the assumption about undefined that we described above this task. You can cite this in your proof, but make sure that you also state the function call that will result in undefined (making the whole expression undefined).

For this one proof only, if you do a proof by cases with 4 or more cases, all of which are substantially similar, we will allow you to do only two of the cases and say that the others "are analogous" without writing them out. Assuming that is true (they really are analogous), then you will get full credit for that proof by cases.

Coding

In this assignment, you will implement the client and server portions of an application that allows users to edit images made up of squares and to save them to and load them from a server.

Submission

Submit your final version to "HW8 Coding". Turning in your work for this assignment is a little trickier than usual, so follow these steps carefully!:

- `cd` into the directory that contains the `/client` and `/server` directories (likely called `hw8-squares`).
- Delete the `node_modules` directories from each directory (you can do this in VSCode or with the command: `rm -r client/node_modules && rm -r server/node_modules`).
- Generate a zipped file containing both of these folders.
  On Mac you can run: `zip -r submission.zip client/ server/`.
  On Windows you can select both folders, right click, and select 'Send to' → 'Compressed (zipped) folder', then rename the zip that is created to 'submission.zip'.
- Go to Gradescope and select the `submission.zip` file that was created by running the last command.
- Make sure you get all the autograder points! If you decide to work on your app some more after turning it in,you'll need to run `npm install --no-audit` in both of the directories again to get your `node_modules` back.

Set up

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-25wi/materials/hw8-squares.git
```

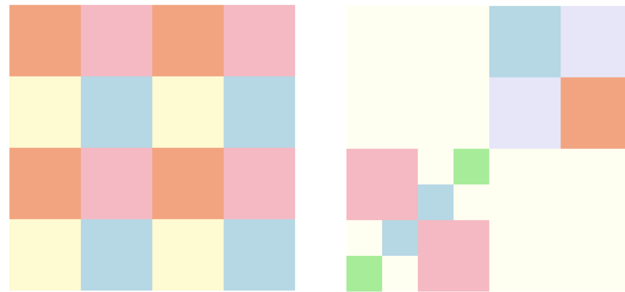
Install the node modules: Create two terminals. Navigate to the `client` directory in one and the `server` directory in the other, and run `npm install --no-audit` in both.

To run the app, you'll need to run the command `npm run start` in both the `client` and `server` directories. Then navigate to where the client is running `http://localhost:8080/`. Currently, you'll get a few "Unused variable" warnings, but if you dismiss them with the 'X' you should see a square on the screen, then try clicking around.

Right now there's not much functionality, but later, the app will allow users to create and save designs of colored squares. **This video includes a walk-through of what the final product looks like as well as an introduction to some important parts of the starter code**. You should *absolutely* watch this video and read through the starter code before starting any parts.

(Note: Your app needs to work like the app in the video but it doesn't need to look exactly the same. Specifically, and contrary to what is shown in the video, the buttons like merge and split don't need to disappear if a square isn't selected, and you do not need to show a javascript alert when saving).

Here are some more pictures of square designs for you:



It is difficult to try to get all the pieces working at once. Instead, you should write one piece at a time, testing it individually. Once the main pieces work individually, you can put them together.

It will be useful to review the content from earlier in the quarter on UI code, making server routes, server requests, and updating client-side state after a request is complete. As before, we expect you to use lecture and section examples them as a template, but it is essential that you understand what those examples are doing!

Note that mutation is not permitted in any of the code on the `/client` side. Mutation on the `/server` side is allowed.

The square types described on the first page are already defined for you in TypeScript in `square.ts`. That file also includes functions `toJson` and `fromJson` that convert between `Square` and JSON, which we represent as "`unknown`" in TypeScript.

Task 2 – Not a Square in the World [16 pts]

Implement the two operations (find and replace) from the written portion. These will be helper functions for parts of the UI we create in the next question.

Write your implementation in the provided function declarations: `findSquare` and `replaceSquare` in `square.ts`. Your implementation should be straight from the spec, though recall the note from the written part which explains that the case the results in "undefined" is an invalid case. In the specs for these functions, we explain that this case is invalid, meaning that users *should* only pass in paths that are valid for the given square.

This means you should not actually return `undefined` in those cases, but you can use defensive programming and throw an `Error` in these cases.

Complete the `TODO` to copy your math definition of replace into the comment above the function.

Make sure you test these operations before moving to the next problem. Tests you write should go in `square_test.ts`. Again, debugging other parts of the client will be easier if you can be confident that any errors you see are due to bugs in that code and not these operations.

Drawing Squares

The file `square_draw.tsx` includes a `SquareElem` tag that can display a square. (The code is just a recursive translation from one tree, `Square`, to another, `JSX.Element`. However, arranging these so they look right on the page is a little tricky, so we provided this for you.)

You can also tell `SquareElem` to "select" a specific solid square, by giving the path to it. That causes the square to display in a slightly different color. Solid squares also change color when the mouse hovers over them. Lastly, `SquareElem` allows you to provide a callback function in the `onClick` property, which will be called when the user clicks on any solid square, providing you a path to the one that was clicked on as a parameter.

The provided code always displays a single split square, with four solid square children. When the user clicks on any of the squares, it tells them to stop that. You can get rid of this code, we'll be enhancing the app to take advantage of these properties of `SquareElem`.

Task 3 – Rage Split [16 pts]

In `FileEditor.tsx` there is a mostly empty React component `FileEditor`. Instead of rendering a `SquareElem` in `App`, as it currently does, change `App` to render an `FileEditor` instead, and the `FileEditor` will render the `SquareElem`. This will allow us to add some design editing functionalities in `FileEditor` and other functionalities in the `App`

When a square element is selected, the `FileEditor` will allow the user to perform the following operations to edit the square and create a design:

1. Change to a different solid color.
2. Split that square into four parts (Initially, 4 solid squares of the same color).
3. Merge the square with its siblings, i.e., replace its parent with a single solid square of that color.

The starter code for `FileEditor` includes some TODOs as guidance, but you should read through the following notes carefully before starting and revisit them later also:

- The first thing to think about is what state you need to keep track of in order to implement these operations. We include state in the starter that we think will be sufficient, so check that out and make sure you understand their purpose.
- You can use any HTML you want to let the user invoke these operations, but one simple choice would be BUTTON (for split & merge) and SELECT (for change color). **Note that the `SquareElem` callback `onClick` already returns the path to the selected square.**
- You may find some functions from `list.ts` useful in this part! Take note of the new function `prefix()` which returns the first `n` elements of a given list.
- When any operation wants to change the square that is displayed, calculate the new `Square` with the changes, and then call `setState` with the new `Square`. That will cause React to invoke `render` again and display the new UI. Remember the `Square` related functions in `square.ts`.
- While it is never a bad thing to write unit tests, for UI like this, you really need to see it in the browser to know that it is really working. For that reason, perform manual testing; however, you should follow the usual rules for deciding which cases to try manually. (To be clear, we will not expect you to turn in any unit tests for your UI code.)

Task 4 – One Foot in the Save [16 pts]

Implement the server portion of the application by adding routes to perform the following operations:

1. Save the contents of a file with a given name.
2. Load the last-saved contents of a file with a given name.
3. List the names of all files currently saved.

The server should allow the file contents to be any valid JSON, which we represent in TypeScript as the "`unknown`" type. The lack of type information should not be a problem as it should not be necessary to examine the contents of the file. We simply store it in save, and return it in load.

Properly test all of these operations before moving to the next problem. Debugging the client-server interaction will be easier if you can be confident that any errors you see are due to bugs in the interaction itself. We have never had you write tests for server code before, so we have provided an example test in `routes_test.ts`, and we will be lenient in grading your tests. You should attempt to write a couple tests per route you build, but if you do not meet all our testing expectations, that is okay.

Additionally, remember to incorporate appropriate error checking.

Some additional notes:

- Don't worry too much about the idea of a "file" if that's confusing. You can think of it as some data stored with a label name. So this is essentially a way to store and lookup values associated with names.
- In `assoc.ts`, we have provided an association list type, as discussed in lecture, called `AssocList`. You should use this to store file names and content. You should carefully read through the helper functions and their documentation in `assoc.ts` to avoid re-implementing already provided functionality.
- The provided code just has a dummy route to remind you what the code looks like for creating and testing routes. Feel free to delete it.
- For debugging/testing purposes, we have provided a "resetSavesForTesting" function that you can call at the end of each test to remove any files you saved during the test. (Not required to use, but encouraged!)
- We've have also provided a "addSaveForTesting" function that can be called to set your saves for testing before you test your list and load endpoints (Not required to use, but encouraged!)
- It is sufficient for the Load and List operations to be GET operations since they are simple requests that just return data, but the Save operation requires passing in the file contents, so we will need to use a POST request rather than GET so we can accept this contents through the body of the request.

Paging

To control which page is displayed in our App, we will create a page enum, like you saw in lecture and on recent homeworks, to define the options for pages to render. Though, this time it will be a little more complicated.

Recall that these enums are a union of constructors, and, like other constructors, they take arguments that define how the page is constructed. You will need to define some constructors to take additional arguments beyond the `kind` field. Good questions to ask to determine if and which arguments should be included are: is there data that is unique to this page? and is there data that defines or identifies what is rendered on this page?

When the App renders a child component, it can pass the fields of the current page in the state as props. As an example of this, check out this lecture slide.

Also, because this app utilizes a server, any time we are in an intermediate state, while waiting for the server to respond, your app should be in a loading state, with a corresponding constructor in the Page enum. For example, to represent the state while the client is waiting for the server to respond with all the files that have been saved, we could include the following constructor:

```
type Page = { kind: "loading-files"} | ...
```

When in a loading state, you can render a simple text loading message on the screen.

Task 5 – Pick-or-Treat [16 pts]

Change the `App` component to have a starting screen that asks the user to type in a name for their square design before opening the page to edit the squares. This starting screen with the text box should be in the `FileList` component. As this is the starting screen, this would be a good time to adjust the initial page view in `App` and start creating your `Page` enum.

Then, add a "Save" button, in the `FileEditor`, that causes the square design they have created to be saved under the name they typed in.

It is a cleaner design to have all the file management code like **fetch requests** in `App`, so we recommend having the `App` pass an `onSave` callback to the `FileEditor` component. When this is invoked, the `App` should send the server a "file" (the design's name and the current state of the edited design) to be saved.

Add a "Back" button as well, in the `FileEditor` component, that goes back to the starting screen. That should also be a callback. The "Back" button should not trigger a file to be saved, and a "Save" should not cause the page to change to the starting screen.

Run the app a few times and save designs under different file names to test this functionality. It may be useful to use `console.log` to print out the state of the app at the top of `render()` to see if the saved files are properly added to your association list.

Task 6 – Service With a File [16 pts]

Change the `FileList` component to show the user the names of all the existing design files. Clicking on any of them should open that file in its last-saved state in the `FileEditor` and allow them to continue editing it. The functionality to pick a name and create a new design from scratch should still be there as well.

You will need to utilize a couple of the routes that you implemented in Task 4 to implement this functionality.

You can use any HTML you want to display the existing file names to the user, but one simple choice would be `<ul>` / `<li>` (unordered list / list items within it), with each containing an `<a>` (link) with the name of that file. The `onClick` event of a link will be called when they click on it. (You can set the `href` tag as `href=''#''` so that the link itself does nothing.)

Congratulations!! Now you have a complex, fully functional, client-server squares application! Give yourself a pat on the back. And, please celebrate by creating a fun squares design and posting it in the "Square Designs" thread on Ed :)

Task 7 – Extra Credit: Picked the Wrong Week to Stop Sniffing New [0 pts]

Add any new features that seem useful! You will get points for any feature that works correctly and seems like it would be valuable to the user.

Though make sure you're fully done with the required app before starting this since extra credit doesn't provide an explicit boost to your grade. If you are close to a grade cutoff, extra credit completion can be reason to bump you grade, but this is entirely at the discretion of the professor (no official formula).

If you complete any extra credit, please create a file in `client/src` called "`extra_credit.text`" and in it list all new features that you added and how to invoke them (if it's not obvious in the UI). This will help us while grading to make sure we don't miss you EC attempts!