CSE 331: Software Design & Implementation Spring 2025

Homework 6

Due: Wednesday, May 14th, 11pm

Written

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW6 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. Make sure you match each HW problem to the page with your work. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

Reasoning rules

Apply the following rules to all reasoning problems unless stated otherwise.

- Assume that all code is TypeScript, and numerical variables are `bigints`.
- **All assertions should use math notation.**
- **You should only use subscripts** for variable mutation that is **not** invertible. Otherwise, you should leave the expressions in terms of the current value of variables.
- Arithmetic simplification is not required. If you choose to do so though, you are always permitted and encouraged to show your work for any simplification or combination of facts. However, please do so clearly to the side of your final assertions.
- If you find yourself applying math definitions during forward or backward reasoning, that is *not permitted* simplification; math definitions should only be applied within a proof.
- If you choose to abbreviate any function names within assertions, you *must* clearly define that abbreviation at the top of the problem.
- There's no need to include the value of *constant* variables in assertions.

Task 1 – The House That Back Built [12 pts]

Use Floyd logic to fill in the assertions for each part, then prove the required implication to show the post condition holds. In proving implications, you may use algebra to rewrite your assertions and support your math with English, if needed, (see the section solutions as examples of valid proofs for these problems).

(a) Use **forward reasoning** to fill in the missing assertions (strongest postconditions) in the following code. Then, prove that the stated postcondition holds.

$\{\{ z > 0 \}\}$
```
x = 1n + 3n * z;
```
$\{\{$ _____ $\}\}$
```
y = z * z;
```
$\{\{$ _____ $\}\}$
```
x = x + y;
```
$\{\{$ _____ $\}\}$
```
z = z + 1n;
```
$\{\{$ _____ $\}\}$
$\{\{ x > 4 \}\}$

(b) Use **backward reasoning** to fill in the missing assertions (weakest preconditions) in the following code. Then, prove that the stated precondition implies what is needed for the postcondition to hold.

$\{\{ y < 0$ and $x > 0 \}\}$
$\{\{$ _____ $\}\}$
```
y = y + 1n;
```
$\{\{$ _____ $\}\}$
```
x = -y;
```
$\{\{$ _____ $\}\}$
```
z = x + 2n;
```
$\{\{ z \geqslant 0 \}\}$

(c) Use forward reasoning to fill in the assertions. Then, prove, by cases, that what we know at the end of the conditional implies the post condition.

Consider x and y to be constant integer variables. This means you do not need to carry the precondition facts about them through your assertions. However, you may still use those facts in your proofs. You should, however, include any *new* facts related to x or y that you learn in your assertions.

$\{\!\{ y > 0 \text{ and } x = y^2 \}\!\}$
```
if (y < 5n) {
```
$\{\!\{$ _____ $\}\!\}$
```
  z = x + y;
```
$\{\!\{$ _____ $\}\!\}$
```
  z = z - 2n;
```
$\{\!\{$ _____ $\}\!\}$
```
} else {
```
$\{\!\{$ _____ $\}\!\}$
```
  z = x - y;
```
$\{\!\{$ _____ $\}\!\}$
```
  z = z + 2n;
```
$\{\!\{$ _____ $\}\!\}$
```
}
```
$\{\!\{$ _____ or _____ $\}\!\}$
$\{\!\{ z + y \geqslant x \}\!\}$

Task 2 – Hula-Loop [10 pts]

In this problem, we will prove the correctness of a loop that finds the quotient of q divided by 3, i.e., the *largest* value r such that $3r \leqslant q$. To say that r is the largest such value means that any larger value would not work, i.e., that $3(r+1) > q$.

We denote the initial value of q at the top by q_0. This is explicitly stated in the precondition as the fact "$q = q_0$". The first two facts of the postcondition say that r is the quotient of q_0 divided by 3. The third fact says that q is the remainder, i.e., the remaining amount not divisible by 3.

This loop calculates the quotient without division. Instead, it just uses subtraction. It operates by increasing `r` and decreasing `q` each time around. The first part of the invariant says that the distance from q_0 down to $3r$ (i.e., $q_0 - 3r$) is the same as the distance from q down to 0 (i.e., $q - 0 = q$). The second part of the invariant says that q has not moved below 0 (i.e., $q \geqslant 0$).

$\{\!\{\ q = q_0$ and $q_0 \geqslant 0\ \}\!\}$
```
let r: bigint = 0n;
```
$\{\!\{$ Inv: $q_0 - 3r = q$ and $q \geqslant 0\ \}\!\}$
```
while (q >= 3n) {
    r = r + 1n;
    q = q - 3n;
}
```
$\{\!\{\ 3r \leqslant q_0$ and $q_0 < 3(r+1)$ and $q = q_0 - 3r\ \}\!\}$

(a) Prove that the invariant is true when we get to the top of the loop the first time.

(b) Prove that, when we exit the loop, the postcondition holds.

(c) Prove that the invariant is preserved by the body of the loop. You may choose to use either forward or backward reasoning (but *not* a mix of the two) to reduce the body to an implication and then check that it holds. Please indicate which direction you reasoned through the code.

Include all intermediate assertions within the loop (after/before *every* line of code).

Task 3 – Loop Dreams [15 pts]

In this problem, you will write a loop that finds the integer square root of a, i.e., the *smallest* integer b such that $a \leqslant b^2$. That b is the smallest such integer means that no smaller integer would work, i.e., that $(b-1)^2 < a$. These two facts are the postcondition of the loop below.

It is only possible to have a number smaller than a if $a > 0$, so that is required by the precondition.

Your loop should calculate the square root using *only addition*. It will operate by increasing b until it is the integer square root of a. In order to do this without multiplication or subtraction, we will need to keep track of two other values. The variable "c" stores $2b-1$, and the variable d stores b^2. The invariant states these two facts and the first part of the postcondition, namely, that $(b-1)^2 < a$.

The basic structure of the loop is as follows. You will fill in the missing pieces below.

$\{\{\, a > 0 \,\}\}$

```
let b: bigint = _____________;
let c: bigint = _____________;
let d: bigint = _____________;
```

$\{\{$ Inv: $(b-1)^2 < a$ and $c = 2b-1$ and $d = b^2$ $\}\}$

```
while (_____________________________________) {
    b = b + 1n;
    c = _____________________________________;
    d = _____________________________________;
}
```

$\{\{\, (b-1)^2 < a$ and $a \leqslant b^2 \,\}\}$

```
return b;
```

(a) Fill in the initialization code above the loop. Then, prove that the invariant holds with your code.

(b) Fill in the loop condition. Then, prove that the post condition holds when the loop exits.

(c) The first line of the body of the loop increases `b` by 1. Fill in the updates to `c` and `d` so that the invariant remains true with a b that is one larger.

Give the two lines of code. Then using either forward or backward reasoning (your choice, but do *not* use a mix of the two) to reduce the body to an implication and then check that it holds. Please indicate which direction you reasoned through the code.

Include all intermediate assertions within the loop (after/before *every* line of code).

Task 4 – If It Ain't Broke, Don't Mix It [18 pts]

We can define the set of secondary colors as an enum-like inductive data type as follows:

$$\textbf{type}\ \text{Secondary} := \text{PURPLE} \mid \text{GREEN} \mid \text{ORANGE}$$

These colors can be expressed as 50/50 mixtures of pairs of red, yellow, and blue. Specifically, PURPLE is 50% red and 50% blue, GREEN is 50% blue and 50% yellow, and ORANGE is 50% red and 50% yellow.

The two functions, $\text{amt-red}, \text{amt-blue} : \text{List}\langle\text{Secondary}\rangle \to \mathbb{R}$, take lists of secondary colors and return the amounts of red and blue, respectively, present in the list:

$$\begin{aligned}
\text{amt-red}(\text{nil}) &:= 0 \\
\text{amt-red}(\text{PURPLE} :: \text{cs}) &:= 0.5 + \text{amt-red}(\text{cs}) \\
\text{amt-red}(\text{GREEN} :: \text{cs}) &:= \text{amt-red}(\text{cs}) \\
\text{amt-red}(\text{ORANGE} :: \text{cs}) &:= 0.5 + \text{amt-red}(\text{cs})
\end{aligned}$$

$$\begin{aligned}
\text{amt-blue}(\text{nil}) &:= 0 \\
\text{amt-blue}(\text{PURPLE} :: \text{cs}) &:= 0.5 + \text{amt-blue}(\text{cs}) \\
\text{amt-blue}(\text{GREEN} :: \text{cs}) &:= 0.5 + \text{amt-blue}(\text{cs}) \\
\text{amt-blue}(\text{ORANGE} :: \text{cs}) &:= \text{amt-blue}(\text{cs})
\end{aligned}$$

In this problem, we will prove the correctness of a loop that finds the amount of red and blue present in a list L of secondary colors. The loop operates by moving forward through the list, updating L at each point, to keep track of where we are, until the list is empty. As usual, L_0 refers to the initial value of the variable L, which is the full list.

The variables "r" and "b" keep track of the amount of red and blue, respectively, in the part of the list processed so far. The first part of the invariant says that the amount of red in the full list is equal to r plus the amount remaining in the list L. The second part states a similar fact for blue.

The postconditions states that r and b contain the full amount of red and blue, respectively, in the full list.

$\{\{\, L = L_0 \,\}\}$
```
let r: number = 0;
let b: number = 0;
```
$\{\{$ Inv: amt-red$(L_0) = r +$ amt-red(L) and amt-blue$(L_0) = b +$ amt-blue$(L) \,\}\}$
```
while (L.kind !== "nil") {
   if (L.hd === "PURPLE") {
```
$\{\{\, P_1 :$ Inv and _____ $\}\}$

$\{\{\, Q_1 :$ _____ $\}\}$
```
      r = r + 0.5;
      b = b + 0.5;
   } else if (L.hd === "GREEN") {
```
$\{\{\, P_2 :$ Inv and _____ $\}\}$

$\{\{\, Q_2 :$ _____ $\}\}$
```
      b = b + 0.5;
   } else { // "ORANGE"
```
$\{\{\, P_3 :$ Inv and _____ $\}\}$

$\{\{\, Q_3 :$ _____ $\}\}$
```
      r = r + 0.5;
   }
```
$\{\{\, Q_0 :$ _____ $\}\}$
```
   L = L.tl;
}
```
$\{\{\, r =$ amt-red(L_0) and $b =$ amt-blue$(L_0) \,\}\}$

(a) Prove that the invariant is true when we get to the top of the loop the first time.

(b) Prove that, when we exit the loop, the postcondition holds.

(c) Use forward reasoning to fill in each of the P_i's above and backward reasoning to fill in each of the Q_i's above. You only need to fill out the assertions which are left blank, you do not need to include any other intermediate assertions.

Note that $L \neq$ nil means that we can write $L = L.\text{hd} :: L.\text{tl}$ since "::" is the only non-nil constructor.

(d) Prove that the invariant is preserved by the loop body by showing that each P_i implies each Q_i.

Note that Q_0 is an exception, Q_0 is just an intermediate backward reasoning step that does not have an associated P_0 or proof.

Task 5 – Barking Up the Wrong Three [18 pts]

Suppose we define the set of base-3 digits as

$$\textbf{type}\ \text{Digit} := 0 \mid 1 \mid 2$$

Then, we can represent number written in base 3 as a List$\langle$Digit$\rangle$.

The following function, zeros : List$\langle$Digit$\rangle \to \mathbb{N}$, counts the number of zero digits in a given base-3 number:

$$\begin{aligned}
\text{zeros}(\text{nil}) &:= 0 \\
\text{zeros}(0 :: \text{ds}) &:= 1 + \text{zeros}(\text{ds}) \\
\text{zeros}(1 :: \text{ds}) &:= \text{zeros}(\text{ds}) \\
\text{zeros}(2 :: \text{ds}) &:= \text{zeros}(\text{ds})
\end{aligned}$$

The next function, even : List$\langle$Digit$\rangle \to \mathbb{B}$, determines whether the given base-3 number is even:

$$\begin{aligned}
\text{even}(\text{nil}) &:= \text{true} \\
\text{even}(0 :: \text{ds}) &:= \text{even}(\text{ds}) \\
\text{even}(1 :: \text{ds}) &:= \text{not even}(\text{ds}) \\
\text{even}(2 :: \text{ds}) &:= \text{even}(\text{ds})
\end{aligned}$$

A base-3 number is even if the sum of its digits is an even number.

In this problem, you will write a loop that, at the same time, calculates the number of zero digits, stored in a variable a, and whether the digits are even, stored in a variable b. The two facts of the postcondition state that these variables contain the values of these two functions on the full list.

Your loop should calculate these values by making a single pass through the list from front to back, exiting when you reach the end of the list. The first fact of the invariant states that the number of zero digits in the whole list is a plus the number of zeros remaining in L. The second fact of the invariant states that the number is even exactly when the evenness of the remaining digits matches the value of b (i.e., they are both true or both false).

The basic structure of the loop is as follows. You will fill in the missing pieces below.

$\{\{\, L = L_0 \,\}\}$
```
let a: bigint = _____________;
let b: boolean = _____________;
```
{{ Inv: zeros(L_0) = a + zeros(L) and even(L_0) = (b = even(L)) }}
```
while (L.kind !== "nil") {
    ...
    // fill in the code here
    ...
    L = L.tl;
}
```
$\{\{\, a = \text{zeros}(L_0) \text{ and } b = \text{even}(L_0) \,\}\}$

(a) Fill in the initialization code above the loop. Then, prove that the invariant holds with your code.

Note that, if x is a boolean, then x = true is true exactly when x is true, and x = false is true when (not x) is true.

(b) Prove that the post condition holds when the loop exits.

(c) Fill in the missing code in the body of the loop so that the invariant is preserved when L moves forward to the next element of the list.

Then, use forward or backward reasoning (or both) to reduce correctness of the loop body to implication(s) and prove that they hold. Please indicate which direction you reasoned in at each step using an arrow or P_i and Q_i labels (similar to the labels provided for you in Task 4).

When doing forward or backward reasoning, you don't need to show intermediate assertions after every step, but you must include all assertions used in the implications for proving that the body of the loop preserves the invariant.

Hint: note that, if b and c are booleans, then "not $b = c$" is the same as "b = not c". Both expressions are true exactly when the values of b and c are different (one is true and one is false).

Coding

After finishing the written part, to get started on the coding part, check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25sp/materials/hw6-paint.git
```

Navigate to the `hw6-paint` directory and run `npm install --no-audit`. Run tests with the command `npm run test`, run the app with the command `npm run start`, and run the linter with the command `npm run lint`.

Starting with this homework, mutation is no longer prohibited (provided you use it correctly)! If you have the comfy-tslint extension in VS Code, you will need to update a setting to allow mutation and prevent it from giving you errors when you do things like use `let` and reassign variables. You will also need to start writing invariants so make sure to renable the checkbox.

Open the **comfy-tslint extension** and press the **gear icon** to the right of the "Disable" and "Uninstall" buttons. Open **"Extension Settings"** from the drop down options that appear. Then check the box to enable the **"Comfy TS Linter: Allow Mutation"** setting and **"Comfy TS Linter: Req Invariants"** setting, and save your settings.

The `npm run lint` command has been updated to allow mutation and require invariants.

Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW6 Code"**:

`even_zeros.ts` `even_zeros_test.ts` `color_ops_test.ts` `App.tsx`
`Canvas.tsx` `Palette.tsx` `Store.tsx`

Wait after submitting to make sure the autograder passes, and leave yourself time to resubmit if there's an issue. The autograder will run your tests, additional staff tests, and the linter. Like homework 5, there will be additional hidden staff tests, the results of which will be available after the deadline.

Task 6 – Wicked Witch of The Test [12 pts]

In this problem, we will copy some of the code we reasoned about on paper into TypeScript functions and test them.
The tests you write should follow the testing requirements for this course (see the notes on testing for a reminder). Additionally, write short labels describing which coverage requirement is met by each test. See the example from HW4 for reference.

Just like last week, we have not provided any tests for these functions. However, you proved that these functions fulfill their post conditions in the written parts, so if you copy it directly, you should already have confidence that it is correct.

(a) Copy the code you wrote on paper in Task 5, into TypeScript code in `even_zeros.ts`.

We have already provided the same scaffolding code as we did in the paper part, so you should keep these parts and fill in the blanks. You do not need to copy over your Floyd logic assertions, but you can if you want.

Write tests for even_zeros in `even_zeros_test.ts`.

(b) Task a look at the function `amtPrimaries` in `color_ops.ts`. This is a copy from the function you reasoned about in Task 4, with the additional behavior of counting amt-yellow which has a parallel definition to amt-red and amt-blue.

This function is already complete - you don't need to do anything to it! But, we will use this function to build our app (in the next task). To help us feel *more* confident in our code,

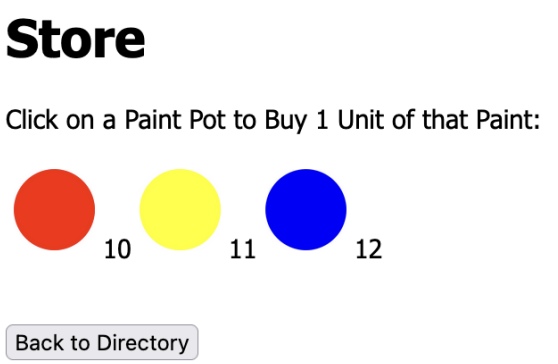
Write tests for amtPrimaries in `color_ops_test.ts`.

Make note of the specs provided for these functions. They describe the behavior in math definitions and English descriptions (matching the written instructions). These functions are implemented with a loop instead of "straight from the spec", so it is important that we know the code implements the behavior the users expect, given the spec, which is why we practice formally proving that correctness in the written part.

Task 7 – Mind Like a Steel App [15 pts]

This week we have designed a (genuinely) super cool app that we hope you enjoy! The app has pages that allow the user to buy, mix, and draw with paint!

The **Paint Store** page allows the user to click the "paint pots" to buy a unit of paint.

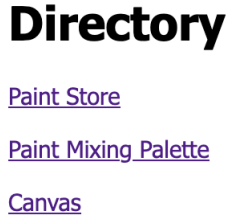


The paint store only sells primary paint colors (red, yellow, blue), so the **Paint Palette** allows the user to "Mix!" together the primary paint colors to create secondary paint colors (orange, green, purple).

Recall from Task 4 that 0.5 or any two distinct primary colors creates 1 new secondary color. This page operates the same way, by decreasing a half unit of paint from the two mixed primary colors and adding a unit of paint to the produced secondary color. If the user no longer wants their mixed secondary colors, they can "Unmix All" which utilizes the `amtPrimaries` function that you wrote tests for in the last task, to unmix all secondary colors back into the primary colors they were created from.

With all of these colors in hand the **Canvas** component allows the user to draw beautiful dot masterpieces. Clicking on the paint pots selects that color to draw with, and clicking on the canvas will place a dot. Each dot drawn spends 1 unit of paint. So if you run out, and you haven't finished your artwork, you'll need to go back to the store!

To keep track of all of these pages, there is a main page directory with links to enter each page.



We have provided the code for the `Store`, `Palette`, and `Canvas` components. In this task, you will put the pieces together. To take a sneak-peak at these pages before they're all put together, you can change the `render` in `App.tsx` to return one of the commented out components instead.

The App component will render the directory on start up, then, when a link to another page is clicked, it will adjust which page is shown.

We have already worked with apps in this class that conditionally show one thing on the screen or another, usually by checking some boolean, string, or if a variable is defined. As our apps become more complex, we will want a more sophisticated way to identify which page of an app should be shown.

We will define a "Page" type which is a union of records, each representing a page the app can take on. The records will each have a "kind" field with some identifying string for that page. As an example, this is how we could have defined Page if we had used this technique for our HW5 cipher app:

```
type Page = {kind:  "input"} | {kind:  "encoding"}
```

As you may observe, this is an inductive data type (specifically an "enum" inductive type). It defines 2 distinct ways to construct a "Page".

As with other constructors, we may want to construct a "Page" with some defining fields in addition to "kind". In general, these additional fields of a page don't include data that needs to be shared across or maintained by the app, but rather identifying features specific to how that page is constructed for a single viewing (e.g. a user profile component which is different each time it's opened depending on which user it's for).

For this app, our pages won't need to be constructed with any additional fields beyond it's "kind," but we will see apps later in the quarter that have more interesting "Page" constructors.

(a) Create a "Page" type, mimicking the example above, at the top of `App.tsx` that represents all the possible pages for this app.

Update `App.tsx` to utilize this type to control the page rendered by the app. There are TODOs in this component to guide where you will want to make edits.

(b) Update the props for each component, so they can share commonly used data across the app.

Notice that each of the pages utilizes the same paint pots. The store can increase the amount or red, yellow, and blue, the palette updates all the colors as they are mixed and unmixed, and the canvas uses the paint as it draws. Currently, each of the individual components has a `paintInventory` state that is initialized as an array of 6 numbers, each representing the available "units" of paint for a color (ROYGBP in order).

Instead of each initializing this state, the App component should have an inventory state that is initialized as empty for each color (`[0, 0, 0, 0, 0, 0]`) and passed as a prop to each child component. Then, that component can initialize its inventory state with the prop it receives instead.

Create `onBackClick` callbacks for each of the `Store`, `Palette`, and `Canvas` components that pass any local data back to the App to share with the other components.

Create handler methods in App that update the state of App to correctly change the page view, and store any data passed from those components.

(c) Enable the app to preserve drawn dots on the canvas. Specifically, update `Canvas.tsx` such that drawing dots, exiting and then reentering the canvas will show the same drawn dots as before exiting.

You are allowed to change any part of the `Canvas` component to do this. Though you do not have to understand how the majority of the component is implemented (e.g. how the dots are drawn on the canvas), so skim the component for the important parts, and use your knowledge of how data is shared between components (hint: what did you do in previous parts?).

Update the rest of the app as needed to support this behavior (hint: you should only need to edit 1 other component).

(d) Have fun with your working app! We had a lot of fun designing this app, so we hope you have fun using it. If you create any cool designs, please share your magnum opus with us on the megathread! (Feel free to *temporarily* hardcode extra paint for your self if you need to.)