CSE 331: Software Design & Implementation

Homework 6 (due Wednesday, November 8th at 11:00 PM)

The problems in this assignment contain a mix of written and coding parts. When you have completed them, submit your solutions in Gradescope. The written problems should be clearly labeled and submitted as a pdf to the "HW6 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW6 Coding" assignment:

`fives.ts` `weave.ts` `ui.tsx` `index.tsx`

For complete instructions for how to submit assignments in this course see the Homework Turn-in Guide.

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:

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. The check the box to enable the **"Comfy TS Linter: Allow Mutation"** setting, and save your settings.

If you do not have the extension installed you will not need to update anything. You can continue to use the `npm run lint` command which we will update for these coming assignments to permit mutation.

For the reasoning problems in this assignment, there are a few rules that you can assume apply to all floyd logic **unless explicitly stated otherwise**:

- Assume that all initial variables represent integers ($\mathbb{Z}$).
- Assume that all code is JavaScript, meaning that division is **floating point division**.
- Unless explicitly allowed in the problem statement, do not use subscripts in assertions to refer to prior values, instead write the assertions in terms of current values of variables.
- Use forward reasoning as your default, not backward reasoning. (Note that at points we do ask you to use backward reasoning, and it will always be for straight-line code, never for conditionals.)
- You are always permitted and encouraged to show your work for any simplification or combination of facts (this can help us award partial credit in some cases), but please do so clearly to the side of your final answers.

1. Hit the Road, Back (10 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".

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

$\{\{\, y > 0 \,\}\}$
```
x = 3 * y + 3;
```
{{ _____ }}
```
z = x - y;
```
{{ _____ }}
```
z = 3 * z;
```
{{ _____ }}
$\{\{\, z \geq 15 \,\}\}$

(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 required for the code to be correct.

$\{\{\, 4x \geq u$ and $v \leq 1 \,\}\}$
{{ _____ }}
```
y = u + v;
```
{{ _____ }}
```
x = x * 4;
```
{{ _____ }}
```
w = x + 1;
```
$\{\{\, w \geq y \,\}\}$

2. Just a Working If (16 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".

(a) Use forward reasoning to fill in the assertions, with the "then" branch on the left and the "else" branch on the right. Then, complete two arguments showing that the two postconditions each imply $\{\{x > y\}\}$.

$\{\{\, y \geq 0 \,\}\}$
```
if (y == 0) {
```
$\{\{$ _____ $\}\}$
```
  x = 2;
```
$\{\{$ _____ $\}\}$
```
} else {
```
$\{\{$ _____ $\}\}$
```
  x = 3 * y;
```
$\{\{$ _____ $\}\}$
```
}
```
$\{\{$ _____ $\}\}$ $\{\{$ _____ $\}\}$

$\{\{\, x > y \,\}\}$

(b) Use forward reasoning to fill in the assertions, with the "then" branch on the left and the "else" branch on the right. Then, complete two arguments showing that the two postconditions each imply $\{\{x \leq 4\}\}$.

Note: While we said in class that we *prefer* not to use subscripts when reasoning forward, that is not always possible. This problem is an example of that, so for this problem, subscript are *permitted*.

$\{\{\, x \geq 1$ and $y > 0 \,\}\}$
```
if (y >= 5*x) {
```
$\{\{$ _____ $\}\}$
```
  x = 20 / y;
```
$\{\{$ _____ $\}\}$
```
} else {
```
$\{\{$ _____ $\}\}$
```
  x = (y - x) / x;
```
$\{\{$ _____ $\}\}$
```
}
```
$\{\{$ _____ $\}\}$ $\{\{$ _____ $\}\}$

$\{\{\, x \leq 4 \,\}\}$

3. Hula-Loop (10 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".

In this problem, we will prove the correctness of a loop that finds the *closest* multiple of 5 that is less than or equal to a given natural number n.[1] Specifically, it returns a number m such that $5m \leq n < 5(m+1)$. This condition says that $5m$ is a multiple of 5 that is less than or equal to n, while the next largest multiple of 5, namely $5(m+1)$, is larger than n, making $5m$ the closest multiple that satisfies the "$\leq$" constraint.

The loop operates by increasing `m` and decreasing n each time around:

$\{\{\ n = n_0$ and $n_0 \geq 0\ \}\}$
```
let m: number = 0;
```
$\{\{$ Inv: $5m = n_0 - n$ and $n \geq 0\ \}\}$
```
while (n >= 5) {
   m = m + 1;
   n = n - 5;
}
```
$\{\{\ 5m \leq n_0$ and $n_0 < 5(m+1)\ \}\}$

We denote the initial value of n at the top by n_0; hence, we include the fact "$n = n_0$" in the precondition. The invariant says that $5m$ is the amount that we have decreased n from its initial value n_0 so far.

(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 use **forward or backward reasoning** (but not a mix of both) to reduce the body to an implication and then prove it holds.

4. Too Many Chefs Spoil the Loop (12 points)

The following parts consist entirely of coding work. They should be submitted with "HW6 Coding".

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-23au-materials/hw-weave.git
```

Then, install the modules using `npm install --no-audit`. You can run the tests with `npm run test`

The file `fives.ts` includes the loop from the previous problem in the function `multOfFive`. This function is tested in `fives_test.ts`. Since loops are similar to recursive functions, our main approach to testing is the 0–1–many heuristic: test inputs that cause the code to go around the loop 0, 1, and many times. These tests also include the boundary cases heuristic, as it is *extremely* common to have an off-by-one error in a loop.

In this problem, we will write new versions of this function that pass many tests but are wrong. Your implementation must follow the rules described below for each version. The code you write should have bugs that a real person might write, demonstrating why it is important to include all the test cases mentioned above.

We provide you a set of tests for each problem that should pass once you are finished; for inputs your code should work on, it verifies correct output, and for inputs your code should fail on, it verifies not correct output.

(a) Implement the function `multOFive10` so that it works correctly on inputs 0, 4, 5, and 9, but fails on 10. Do this by changing just the one line of code "`m = m + 1`" in the body to `m` = something else.

[1] The right way to calculate this in TypeScript is `Math.floor(n/5)`. This is just an exercise.

(b) Implement the function `multOFive5` so that it works correctly on inputs 0, 4, 8, and 12, but fails on 5. Do this by changing just the loop exit condition.

(c) Implement the function `multOFive0` so that it works correctly on inputs 4, 8, and 12, but fails on 0. Do this by changing just the loop exit condition and the initial value of `m`.

(It's surprisingly hard to fail this case but pass the others. In general, the other test cases do a much better job of testing 0 than 0 does of testing the other cases. Hence, if could only do one test, you wouldn't want that test to be 0.)

5. Loop Dreams (16 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".

Consider the following functions, amount-greater and amount-less. amount-greater finds, for each element in a list greater than a given value x, the difference between the element and x, and returns the sum of their differences. amount-less does the same except for elements less than a given value x. These functions ignore values that are not strictly greater or strictly less than x, respectively. For example, for R = cons(10, cons(4, cons(6, cons(7, nil)))), amount-greater(R, 6) = 5 and amount-less(R, 7) = 4.

func	amount-greater(nil, x)	:=	0	
	amount-greater(cons(y, L), x)	:=	$(y - x)$ + amount-greater(L, x)	if $y > x$
	amount-greater(cons(y, L), x)	:=	amount-greater(L, x)	if $y \leq x$
func	amount-less(nil, x)	:=	0	
	amount-less(cons(y, L), x)	:=	$(x - y)$ + amount-less(L, x)	if $y < x$
	amount-less(cons(y, L), x)	:=	amount-less(L, x)	if $y \geq x$

In this problem, we will prove that the following code correctly calculates values amount-greater(L, x) and amount-less(L, x) in one pass over the list L. The invariant for the loop is already provided. It references L_0, which refers to the initial value of L when the function starts.

```
let a: number = 0;
let b: number = 0;
{{ Inv: amount-greater(L_0, x) = a + amount-greater(L, x) and
        amount-less(L_0, x) = b + amount-less(L, x)}}
while (L !== nil) {
   if (L.hd > x) {
      a = a + (L.hd - x);
   } else if (L.hd < x) {
      b = b + (x - L.hd);
   } else {
      // Do nothing
   }
   L = L.tl;
}
{{ a = amount-greater(L_0, x) and b = amount-less(L_0, x) }}
```

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

Note that our precondition is $L = L_0$, per the comments above.

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

(c) Prove that the invariant is preserved by the body of the loop. To do this, use **backward reasoning** to reason through the last assignment statement "`L = L.tl;`". Then, use **forward reasoning** for each branch of the "`if`" statement. Finally, complete the correctness check by verifying that each of the assertions you produced for each branch with forward reasoning implies the assertion produced by backward reasoning immediately above the last assignment statement.

Recall, as we noted in quiz section, that $L \neq \text{nil}$ means that $L = \text{cons}(L.\text{hd}, L.\text{tl})$.

6. Take It Or Weave It (20 points)

The following parts consist of entirely coding work. They should be submitted with "HW6 Coding".

In this problem, we will implement some helper routines needed for a weaving application that we will finish in the next problem. In the instructions and function names we refer to the term "warp" which are the vertical threads of a weave ("weft" are the horizontal threads).

We will start by looking at the functions `weaveWarpFacedOdds` and `weaveWarpFacedEvens` in `weave.ts`. These function take a list of colors as input and returns take(colors) and skip(colors) respectively, where "take" and "skip" have the math definitions as in HW4.

Both take and skip are defined recursively on lists, so they fit the "bottom-up" template from lecture. However, like "swap" (which we translated to a loop in section), these functions make a recursive call on a list that is two elements shorter. (Technically, take is defined to make a recursive call to skip, but skip then makes a recursive call to take. All together, the next recursive call to take is on a list that is two elements shorter.) As a result of this, your loops will need to process two elements at a time, rather than one, as we did with swap.

(a) Using the template described in lecture, write the invariants for the loop implementations of both functions. Include the invariants in `weave.ts` above the provided loop outlines in these functions.

(b) The initialization and exit conditions of both functions are already provided. Both loops will exit when there are 0 or 1 elements left in the list. To start, we will assume the list has even length, which means we will exit the loop with 0 elements remaining.

Fill in the body of the loop in `weaveWarpFacedOdds` so that it correctly calculates take(colors) and the body of the loop in `weaveWarpFacedEvens` so that it correctly calculates skip(colors). Your code must be correct with the invariants you specified in part (a).

Think carefully about what this code should do. Try to get it right on your first attempt.[2]

Run the tests and confirm that the "`weaveWarpFacedOdds` - even length" and "`weaveWarpFacedEvens` - even length" tests now pass.

(c) Add code at the beginning of `weaveWarpFacedOdds` and `weaveWarpFacedEvens` to detect lists of odd length. (It is fine to call len in order to do so.) When the list has odd length, each function should return the correct answer by calling the other function (e.g., `weaveWarpFacedOdds` calls `weaveWarpFacedEvens`) on the *tail* of the passed-in list, which will have even length, and using the value it returns to calculate the correct answer for the original list with odd length.

Think carefully about what this code should do to return the correct answer.

Once you are confident it is correct, run the tests and confirm that the "`weaveWarpFacedOdds` - odd length" and "`weaveWarpFacedEvens` - odd length" tests now pass.

[2] Good practice for interviews.

The other two weaving helper functions, `weaveBalancedOdds` and `weaveBalancedEvens`, will use loops to implement two new functions defined as follows:

$$
\begin{array}{lll}
\textbf{func}\ \text{leave}(\text{nil}, c) & := \text{nil} & \text{for any } c : \mathbb{Z} \\
\quad \text{leave}(\text{cons}(a, L), c) & := \text{cons}(a, \text{replace}(L, c)) & \text{for any } a, c : \mathbb{Z} \text{ and } L : \text{List} \\
\textbf{func}\ \text{replace}(\text{nil}, c) & := \text{nil} & \text{for any } c : \mathbb{Z} \\
\quad \text{replace}(\text{cons}(a, L), c) & := \text{cons}(c, \text{leave}(L, c)) & \text{for any } a, c : \mathbb{Z} \text{ and } L : \text{List}
\end{array}
$$

These functions are like "take" and "skip" except that, rather than skipping an element, they replace it with the value "c", passed in as the second argument.

(d) Using the template described in lecture, write the invariants for the loop implementations of both functions. Include the invariants in `weave.ts` above the provided loop outlines in these functions.

(e) Fill in the missing loop bodies in `weaveBalancedOdds` and `weaveBalancedEvens` so that they correctly calculate leave(colors, c) and replace(colors, c), respectively. Your code must be correct with the invariants you specified in part (d).

Think carefully about what this code should do. Try to get it right on your first attempt.

Run the tests and confirm that the "`weaveBalancedOdds` - even length" and "`weaveBalancedEvens` - even length" tests now pass.

(f) Add code at the beginning of `weaveBalancedOdds` and `weaveBalancedEvens` to detect lists of odd length. When the list has odd length, each function should return the correct answer by calling the other function on the *tail* of the passed-in list, which will have even length, and using the value it returns to calculate the correct answer for the original list with odd length.

Think carefully about what this code should do to return the correct answer.

Once you are confident it is correct, run the tests and confirm that the "`weaveWarpFacedOdds` - odd length" and "`weaveWarpFacedEvens` - odd length" tests now pass.

7. Weave Got it Made (16 points)

The following parts consist entirely of coding work. They should be submitted with "HW6 Coding".

In this problem, we will finish the weaving application. To do that, we will need to write a few more functions. The first two of those will be `weaveWarpFaced` and `weaveBalanced` in `weave.ts`.

Both functions take as arguments a number of `rows` and a list of `colors`. `weaveWarpFaced` should return a list containing that number of items (each called a "row"), where the items at even indexes (0, 2, 4, etc.) in the list are the result of calling `weaveWarpFacedEvens(colors)` and the items with odd indexes (1, 3, 5, etc.) in the list are the result of calling `weaveWarpFacedOdds(colors)`.

We can define this formally as follows (abbreviating weaveWarpFaced to "weave"):

$$
\begin{array}{lll}
\textbf{func}\ \text{weave}(0, \text{colors}) & := \text{nil} & \\
\quad \text{weave}(1, \text{colors}) & := \text{evens} & \\
\quad \text{weave}(n+2, \text{colors}) & := \text{cons}(\text{evens}, \text{cons}(\text{odds}, \text{weave}(n, \text{colors}))) & \text{for any } n : \mathbb{N} \\
\qquad \text{where evens} := \text{weaveWarpFacedEvens}(\text{colors}) & & \\
\qquad\quad \text{and odds} := \text{weaveWarpFacedOdds}(\text{colors}) & &
\end{array}
$$

weaveBalanced (abbreviated to "weaveBal") is defined similarly, but with weaveWarpFacedEvens and weaveWarpFacedOdds replaced by weaveBalancedEvens and weaveBalancedOdds, respectively. The latter two functions also require an additional argument ("c", the replacement color), which is provided as an argument to

`weaveBalanced` as well.

$$
\begin{aligned}
\textbf{func}\ \text{weaveBal}(0, \text{colors}, c) &:= \text{nil} \\
\text{weaveBal}(1, \text{colors}, c) &:= \text{evens} \\
\text{weaveBal}(n+2, \text{colors}, c) &:= \text{cons}(\text{evens}, \text{cons}(\text{odds}, \text{weaveBal}(n, \text{colors}))) \quad \text{for any } n : \mathbb{N} \\
&\quad \text{where evens} := \text{weaveBalancedEvens}(\text{colors}, c) \\
&\quad\ \ \text{and odds} := \text{weaveBalancedOdds}(\text{colors}, c)
\end{aligned}
$$

Note that these functions are defined by recursion on a natural number, so these fit the natural number template from lecture. However, as in the previous problem, the functions make recursive calls on a number that is two smaller, so we will need the body of the loop to jump by two on each iteration, rather than one.

Also note that whether the *last* element in the list produced is "`evens`" or "`odds`" depends on whether the **total number of rows** requested is **even or odd**. For example, $\text{weave}(2, \text{colors}) = \text{cons}(\text{evens}, \text{cons}(\text{odds}, \text{nil}))$, which ends with "`odds`", but $\text{weave}(3, \text{colors}) = \text{cons}(\text{evens}, \text{cons}(\text{odds}, \text{cons}(\text{evens}, \text{nil})))$, which ends with "`evens`". Since we are building the return value bottom-up, this means that the *first* element we need to add to the list is different in those two cases!

Handle this in your code by initializing the variable "i" from the template to 0 when the total number of rows requested is even and to 1 when the total number of rows requested is odd. You will need to initialize the variable "S" differently in the two cases so that the invariant is true initially. The loop body can then add two elements to the list each time through, and when the loop exits, we should have the correct answer stored in S.

(a) Implement the function `weaveWarpFaced` in `weave.ts` using a loop as described above.

Include a loop invariant in the comments. Make sure your code is correct with *that* invariant. (Note that your code would still pass the tests if it is correct with some *other* invariant, but it would not be clear to the reader why it is correct.)

Verify that all the tests for `weaveWarpFaced` now pass by running `npm run test`.

(b) Use the same approach to implement `weaveBalanced`. It should similarly call `weaveBalancedEvens` to calculate the colors for the even rows and `weaveBalancedOdds` to calculate the colors for the odd rows.

This should largely be a copy-and-paste from the previous problem. Make sure you update the invariant as well as the code. Then, verify that all the tests for `weaveBalanced` now pass.

Confirm that the application now works by running `npm run start`. You should be able to enter a list of colors (e.g., "ROYGBPPBGYOR") and click draw to see a picture with two rows of a weave with those colors! (Warp-Faced or Balanced respectively)

The only problem is that the application only shows 2 rows of the weave, which isn't enough to get a good sense of whether it will look nice. To fix this, we will need to write one more function in `ui.tsx`:

```
export const DrawWeave =
    (weave: List<List<Color>>, index: number): List<JSX.Element> => {..};
```

`DrawWeave` takes a list of rows as an argument and returns a list of HTML elements, one for each row, that displays it. The code to display a given row as HTML is already provided in `DrawWeaveRow`. That function takes three arguments: the list of colors in the row, a boolean indicating whether to offset the colors on the left, and a key parameter to include on the HTML element returned (which must be different for each row). The offset parameter should be true for rows at even indexes (starting from 0) and false for rows at odd indexes.

The `index` parameter to `DrawWeave` is discussed below. Clients will always pass in a value of 0.

(c) `DrawWeave` can be formally defined recursively on the list of rows. That means it fits the "bottom up" with lists template, so we could implement it as a loop using the template, but instead, let's use recursion this time to make sure we haven't forgotten. Some *hints*:

- To keep track of even and odd rows, it may be useful to split into cases of 0, 1, 2+ remaining rows in your function.
- To correctly give each row a unique key, utilize the `index` parameter to `DrawWeave` to keep track of which row you're on. The index of the first row is 0, which is indeed what clients pass in as `index`.

(d) Finally, change the code in `index.tsx` that uses `Weave` to pass in 20 for the "rows" property instead of just 2. When you run the application, you should now see a picture with 20 rows of alternating patterns.

The picture shows what a weave using those warp colors would look like. The application will let users find a set of colors that make a pattern they like before they get to work making it on a loom.

Congratulations! You have finished another app.

8. Extra Credit: Do Bears Loop in the Woods? (0 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".

In this problem, we will prove that the following loop correctly calculates concat(L_0, R_0), where, as usual, L_0 and R_0 refer to the initial values of the lists L and R:

```
let S: List = rev(L);
while (S !== nil) {
   R = cons(S.hd, R);
   S = S.tl;
}
```

(a) Unfortunately, the author of this code didn't actually document the loop invariant. Ugh.

What is the invariant of the loop?

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

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

(d) Prove that the invariant is preserved by the body of the loop. You use any combination of forward and backward reasoning to reduce this to an implication and then prove it holds.