# CSE 331: Software Design & Implementation

## Homework 6 (due Wednesday, May 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 to the "HW6 Written" assignment. `weave.ts` should be submitted for the coding portion to the "HW6 Coding" assignment.

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:

- Assume that all code is TypeScript, and number variables are `bigints`.
- **You should not use subscripts** (unless we explicitly tell you otherwise or introduce subscriptted variables in the problem to refer to the original version of variable when the function was called). Instead, write all assertions in terms of the current value of variables.
- Arithmetic simplicication is not required, but if you choose to do so, 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.

The next problem involves the following ADT. It refers to two functions explained below.

```
/** A list of string values containing _no duplicates_. */
export interface StringSet {
  /** @returns contains(s, obj) */
  contains: (s: string) => boolean;

  /** @returns remove(s, obj) */
  delete: (s: string) => StringSet;

  /** @returns cons(s, remove(s, obj)) */
  insert: (s: string) => StringSet;
}
```

Note: This interface's `delete` and `insert` are producer functions, meaning they create a new set with an element deleted/inserted instead of modifying the existing set.

Find the definition for the function contains in HW5 and the function remove is defined as follows:

| | | | | | |
|---|---|---|---|---|---|
| **func** | remove$(s, \text{nil})$ | $:=$ | nil | | for any $s : \mathbb{Z}$ |
| | remove$(s, \text{cons}(t, L))$ | $:=$ | remove$(s, L)$ | if $s = t$ | for any $t : \mathbb{S}^*$ and $L$ : List |
| | remove$(s, \text{cons}(t, L))$ | $:=$ | cons$(t, \text{remove}(s, L))$ | if $s \neq t$ | for any $t : \mathbb{S}^*$ and $L$ : List |

The most direct way to implement this ADT would be as a list containing no duplicates:

```
/** Represents a StringSet as a list with _no duplicates_. */
class StringSetWithNoDups implements StringSet {

  // RI: this.list has no duplicates
  // AF: obj = this.list
  readonly list: List<string>;

  // @requires list has no duplicates
  // @effects makes obj = list
  constructor(list: List<string>) {
    this.list = list;
  }

  insert = (s: string): StringSet => {
    // note, the ordering of elements in a set does not matter
    return new StringSetWithNoDups(cons(s, remove(s, this.list)));
  };

  delete = (s: string): StringSet => {
    return new StringSetWithNoDups(remove(s, this.list));
  };
}
```

This implementation of `insert` is as direct as possible because our abstract state is simply the concrete state. In the next problem, we will consider a different implementation....

## 1. Cool your Sets (18 points)

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

We will consider implementing the `StringSet` interface with the following class:

```
/** Represents a StringSet as a list that _may have duplicates_. */
class StringSetWithDups implements StringSet {

  // AF: obj = uniquify(this.list)
  readonly list: List<string>;

  // @effects makes obj = uniquify(list)
  constructor(list: List<string>) {
    this.list = list;
  }

  insert = (s: string): StringSet => {
    return new StringSetWithDups(cons(s, this.list));  // no remove!
  };

  delete = (s: string): StringSet => {
    return new StringSetWithDups(remove(s, this.list));
  };
}
```

This representation allows duplicates, but since the abstract state (as defined in the specification) does not have duplicates, we must remove them to produce the abstract state. We do so with the following function:

$$\begin{aligned} \textbf{func}\ \text{uniquify}(\text{nil}) &:= \text{nil} \\ \text{uniquify}(\text{cons}(t, L)) &:= \text{cons}(t, \text{remove}(t, \text{uniquify}(L))) \quad \text{for any } t : \mathbb{S}^* \text{ and } L : \text{List} \end{aligned}$$

(a) We want to prove that the methods of our ADT are correct. First, we will need to prove the following claim by induction on $L$ to use in part (d):

$$\text{uniquify}(\text{remove}(s, L)) = \text{remove}(s, \text{uniquify}(L)) \qquad \text{for any } s \in \mathbb{S}^* \text{ and } L \in \text{List}$$

*Hint*: In your inductive step, you may need cases to know which line of remove's definition to apply!

You may use without proof the following facts about remove:

$$\begin{aligned} &\text{remove}(s, \text{remove}(s, L)) = \text{remove}(s, L) && \text{for any } s \in \mathbb{S}^* \text{ and } L \in \text{List} \\ &\text{remove}(s, \text{remove}(t, L)) = \text{remove}(t, \text{remove}(s, L)) && \text{for any } s, t \in \mathbb{S}^* \text{ and } L \in \text{List} \end{aligned}$$

We will call these "Lemma A" and "Lemma B", respctively.

(b) Explain, in a couple English sentences, why we need to prove the following equations in order to know that `insert` and `delete` are correct, respectively:

$$\text{uniquify}(\text{cons}(s, \text{this.list})) = \text{cons}(s, \text{remove}(s, \text{obj}))$$

$$\text{uniquify}(\text{remove}(s, \text{this.list})) = \text{remove}(s, \text{obj})$$

(c) Prove by calculation that `insert` is correct by showing that the following equation holds:

$$\text{uniquify}(\text{cons}(s, \text{this.list})) = \text{cons}(s, \text{remove}(s, \text{obj}))$$

(d) Prove by calculation that `delete` is correct by showing that the following equation holds (remember to use part (a)!):

$$\text{uniquify}(\text{remove}(s, \text{this.list})) = \text{remove}(s, \text{obj})$$

## 2. Hit the Road, Back (22 points)

The following parts consist entirely of written work. They should be submitted with "HW6 Written".
In your assertion implication proofs, you do not need to write formal proofs by calculation, english sentences explaining the facts can be sufficient (see the section 6 worksheet solutions for an example).

(a) Use **forward reasoning** to fill in the missing assertions (strongest postconditions) in the following code. Then, prove that the stated postcondition holds. **Do not use subscripts in this part.**

$\{\{\ b > 5$ and $c > 1\ \}\}$
`a = 2n * b + 3n;`
$\{\{$ _____ $\}\}$
`b = b - 5n;`
$\{\{$ _____ $\}\}$
`c = c * b;`
$\{\{$ _____ $\}\}$
$\{\{\ a < 2c + 15\ \}\}$

(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.

$\{\{\ c < 0\ \}\}$
$\{\{$ _____ $\}\}$
`b = -c;`
$\{\{$ _____ $\}\}$
`c = c + 1n;`
$\{\{$ _____ $\}\}$
`a = b + 1n;`
$\{\{\ a \geq c\ \}\}$

(c) Use forward reasoning to fill in the assertions. Then, combine these conditions to assert the invariant we know at the end of the conditional and complete an argument by cases that this invariant implies $\{\{b > 5\}\}$. **Do not use subscripts in this part.**

Note that because we have 3 branches in our conditional, the fact we know just after the conditional will have 3 cases "or"ed together.

$\{\{ a \geq 1$ and $b = a^2 \}\}$

```
if (b < 20n) {
```

$\{\{$ _____ $\}\}$

```
  b = b + 5n;
```

$\{\{$ _____ $\}\}$

```
} else if (b < 30n) {
```

$\{\{$ _____ $\}\}$

```
  b = (b / a) + 1n;
```

$\{\{$ _____ $\}\}$

```
} else {
```

$\{\{$ _____ $\}\}$

```
  b = b / a;
```

$\{\{$ _____ $\}\}$

```
}
```

$\{\{$ _____ or _____ or _____ $\}\}$

$\{\{ b > 5 \}\}$

## 3. Hula-Loop (12 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 *smallest* value when multiplied by 5 that is greater than a given natural number $x$.[1] Specifically, it returns a number $y$ such that $x < 5y \leq x + 5$. This condition says that $5y$ is the *smallest* multiple of 5 greater than x because $x < 5y$, but $x + 5 < 5(y + 1)$, which doesn't satisfy $5(y + 1) \leq x + 5$.

The loop operates by increasing `y` and decreasing `x` each time around:

$\{\{ x = x_0 \text{ and } x_0 \geq 0 \}\}$

```
let y: bigint = 0n;
```

$\{\{ \text{Inv: } 5y = x_0 - x \text{ and } x \geq -5 \}\}$

```
while (x >= 0n) {
    y = y + 1n;
    x = x - 5n;
}
```

$\{\{ x_0 < 5y \text{ and } 5y \leq x_0 + 5 \}\}$

We denote the initial value of $x$ at the top by $x_0$; hence, we include the fact "$x = x_0$" in the precondition. The invariant says that $5y$ is the amount that we have decreased $x$ from its initial value $x_0$ so far.

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

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

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

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

## 4. Loop Dreams (20 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.

$$
\begin{array}{lll}
\textbf{func}\ \text{amount-greater}(\text{nil}, x) & := & 0 \\
\quad \text{amount-greater}(\text{cons}(y, L), x) & := & (y - x) + \text{amount-greater}(L, x) \quad \text{if } y > x \\
\quad \text{amount-greater}(\text{cons}(y, L), x) & := & \text{amount-greater}(L, x) \quad \text{if } y \leq x \\
\textbf{func}\ \text{amount-less}(\text{nil}, x) & := & 0 \\
\quad \text{amount-less}(\text{cons}(y, L), x) & := & (x - y) + \text{amount-less}(L, x) \quad \text{if } y < x \\
\quad \text{amount-less}(\text{cons}(y, L), x) & := & \text{amount-less}(L, x) \quad \text{if } y \geq x
\end{array}
$$

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: bigint = 0;
let b: bigint = 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 the invariant is preserved by the body of the loop (meaning the invariant assertion is true at the bottom of the loop body). Use reasoning to fill in assertions throughout the loop body. For the conditionals, you must use forward reasoning; to reason over the last line, you may choose which direction to reason. Finally, complete the correctness check for all cases where the assertions reduce to an implication.

(We'll see an example similar to this in section Tuesday).

Recall, that $L \neq$ nil means that $L =$ cons$(L.\text{hd}, L.\text{tl})$.

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

## 5. Take It Or Weave It (28 points)

The following parts consist of entirely 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/cse331-24sp/cse331-24sp-materials/hw-weave.git
```

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

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

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

Both keep and drop 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, keep is defined to make a recursive call to drop, but drop then makes a recursive call to keep. All together, the next recursive call to keep 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 `warpFacedOdds` so that it correctly calculates keep(colors) and the body of the loop in `warpFacedEvens` so that it correctly calculates drop(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 "`warpFacedOdds` - even length" and "`warpFacedEvens` - even length" tests now pass.

(c) Add code at the beginning of `warpFacedOdds` and `warpFacedEvens` 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., `warpFacedOdds` calls `warpFacedEvens`) 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 "`warpFacedOdds` - odd length" and "`warpFacedEvens` - odd length" tests now pass.

Now, we will finish the weaving application by writing one more function: `weave` in `weave.ts`, described on the next page.

`weave` takes as arguments a number of `rows`, a list of `colors`, and a replacement color ("c"). `weave` should return a list containing that number of items (each called a "row"), where the items at even indexes (0, 2, 4,

[2] Good practice for interviews.

etc.) in the list are the result of calling `warpFacedEvens(colors)` and the items with odd indexes (1, 3, 5, etc.) in the list are the result of calling `warpFacedOdds(colors)`.

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

Note that this function is defined by recursion on a natural number, so these fit the "natural number" template from lecture. However, as in the previous problem, it makes 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 $S$.

(d) Implement the function `weave` in `weave.ts`.

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 `weave` now pass by running `npm run test`.

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



Congratulations! You have finished another app.

## 6. 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.kind !== "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.