CSE 331: Software Design & Engineering — Spring 2023

Final: Section A — 6/6/2023 4:30pm

Name: _____

UW Email: _____@uw.edu

This exam contains 16 pages (including this cover page) and 6 problems. Check to see if any pages are missing. Enter all requested information on the top of this page.

Instructions:

- Closed book, closed notes, no cell phones, no calculators.
- You have **1 hour and 50 minutes** to complete the exam.
- Answer all problems on the exam paper.
- If you need extra space use the back of a page.
- Problems are not of equal difficulty; if you get stuck on a problem, move on.
- It may be to your advantage to read all the problems before beginning the exam.

Problem	Points	Score
1	27	
2	15	
3	18	
4	12	
5	12	
6	16	
Total:	100	

The code in this test will work with horizontal lines drawn with a mix of different colors. The simplest way to represent such a line is as a `Color[]`, where a `Color` is defined as usual:

```
type Color = "red" | "green" | "blue" | ... ;
```

We can show this on the screen by creating one square, of the appropriate color, for each element in the array. (This is how we drew a line of the weave in HW6, for example.)

When the array has multiple elements in a row, all of the same color, however, it is possible to draw them all using a single HTML element that is changed to have the width of all the squares put together. For example, if there are three red elements in a row in the array, we could instead draw those with a single HTML element having three times the normal width, i.e., a *rectangle*.

We can represent the line this way, by making it a `Rect[]`, where a `Rect` represents multiple elements of the same color. It can be defined in TpeScript as follows:

```
type Rect = {clr: Color, amt: number};  // with amt >= 0 an integer
```

where `clr` is the color and `amt` is the amount of squares it represents.

For example, the following two arrays both represent the same line:

```
const A: Color[] = ["red", "red", "red", "white", "blue", "blue"];
const B: Rect[] = [{clr: "red", amt: 3}, {clr: "white", amt: 1},
                   {clr: "blue", amt: 2}];
```

Both would draw a line that is 6 squares wide, consisting of 3 red, followed by 1 white, followed by 2 blue. The first draws all 6 squares individually, while the second draws the same picture with 3 rectangles of widths 3, 1, and 2 (in that order).

The first functions we consider convert between the formats defined on the previous page. The function `Rectify` will convert a `Color[]` into a `Rect[]` that represents the same line, and the function `SquareUp` will convert from a `Rect[]` back to the original `Color[]`.

The conversion from rectangles back to squares can be defined mathematically as follows:

$$
\begin{array}{lll}
\textbf{func}\ \text{square-up}([]) & := & [] \\
\quad \text{square-up}(A \mathbin{+\!\!+} [\{\text{clr} : c,\ \text{amt} : 0\}]) & := & \text{square-up}(A) \\
\quad \text{square-up}(A \mathbin{+\!\!+} [\{\text{clr} : c,\ \text{amt} : n+1\}]) & := & \text{square-up}(A \mathbin{+\!\!+} [\{\text{clr} : c,\ \text{amt} : n\}]) \mathbin{+\!\!+} [c]
\end{array}
$$

where, A is any array of rectangles, c is any color, and n is any natural number.

With that definition in hand, we can specify the two conversion functions as follows:

```
/**
 * Returns an array of squares representing the same line as R.
 * @param R An array of rectangles
 * @returns square-up(R)
 */
function SquareUp(R: readonly Rect[]): Color[] { ... }

/**
 * Returns an array of rectangles representing the same line as S.
 * @param S An array of squares
 * @returns R such that S = square-up(R) and
 *     R[j].amt >= 1 for any 0 <= j <= R.length - 1 and
 *     R[j].clr != R[j+1].clr for any 0 <= j <= R.length - 2
 */
function Rectify(S: readonly Color[]): Rect[] { ... }
```

The postcondition of `Rectify` is more complex. The first line says that it is the inverse of `SquareUp`. The second line says that all the rectangles it returns cover at least one square (no empty rectangles). The third line says that that adjacent rectangles must have different colors. We disallow adjacent rectangles with the same color because those squares can instead be described with just one rectangle, whose width is the sum of the two rectangle's widths.

Continuing the example from the previous page, a call to `Rectify(A)` would return an array containing the same values as `B`, and a call to `SquareUp(B)` would return an array containing the same values as `A`.

1. (27 points) **Everybody Loops**

Consider the following code, which claims to implement `SquareUp` from the prior page.

```
const S: Color[] = [];
let k: number = 0;
{{ P1: S = [] and k = 0 }}
{{ Inv1: S = square-up(R[0 .. k − 1]) }}
while (k !== R.length) {
    let j = 0;
    {{ P2: Inv1 and j = 0 }}
    {{ Inv2: S = square-up(R[0 .. k − 1] ++ [{clr : R[k].clr, amt : j}]) }}
    while (j !== R[k].amt) {
        {{ P3: Inv2 and j ≠ R[k].amt }}
        {{ Q3: S ++ [R[k].clr] = square-up(R[0 .. k − 1] ++ [{clr : R[k].clr, amt : j + 1}]) }}
        S.push(R[k].clr);
        j = j + 1;
    }
    {{ P4: Inv2 and j = R[k].amt }}
    {{ Q4: S = square-up(R[0 .. k]) }}
    k = k + 1;
}
{{ P5: S = square-up(R[0 .. k − 1]) and k = R.length }}
{{ Post: S = square-up(R) }}
return S;
```

(a) Use reasoning to fill in all blank assertions above. The 'P_i's should be filled in with forward reasoning and the 'Q_i's should be filled in with backward reasoning.

(b) Prove that P_1 implies Inv_1.

Solution:

$$
\begin{aligned}
S &= [] \\
&= \text{square-up}([]) && \text{def of square-up} \\
&= \text{square-up}(R[0 \,..\, -1]) \\
&= \text{square-up}(R[0 \,..\, k-1]) && \text{since } k = 0
\end{aligned}
$$

(Continued on next page...)

Note: You do not need to prove that the array accesses are valid in the following parts. (They are, but you do not need to prove that.)

(c) Prove that P_2 implies Inv_2.

Solution:

$$\begin{aligned} S &= \text{square-up}(R[0 \,..\, k-1]) && \text{by } \text{Inv}_1 \\ &= \text{square-up}(R[0 \,..\, k-1] +\!\!+ [\{\text{clr} : R[k].\text{clr},\ \text{amt} : 0\}]) && \text{def of square-up} \\ &= \text{square-up}(R[0 \,..\, k-1] +\!\!+ [\{\text{clr} : R[k].\text{clr},\ \text{amt} : j\}]) && \text{since } j = 0 \end{aligned}$$

(d) Prove that P_3 implies Q_3.

(Hint: my proof of this is very short. If you are finding this difficult to prove, make sure you are proving the right claim. A false claim will be impossible to prove!)

Solution:

$$\begin{aligned} S +\!\!+ [R[k].\text{clr}] &= \text{square-up}(R[0 \,..\, k-1] +\!\!+ [\{\text{clr} : R[k].\text{clr},\ \text{amt} : j\}]) +\!\!+ [R[k].\text{clr}] && \text{by } \text{Inv}_2 \\ &= \text{square-up}(R[0 \,..\, k-1] +\!\!+ [\{\text{clr} : R[k].\text{clr},\ \text{amt} : j+1\}]) && \text{def of square-u} \end{aligned}$$

(Continued on next page...)

(e) Prove that P_4 implies Q_4.

Note that $R[k] = \{\mathsf{clr} : R[k].\mathsf{clr},\ \mathsf{amt} : R[k].\mathsf{amt}\}$ is true by definition (of R). You are free to use this fact without proof or explanation.

Solution:

$$\begin{aligned} S &= \text{square-up}(R[0\ ..\ k-1] +\!\!+ [\{\mathsf{clr} : R[k].\mathsf{clr},\ \mathsf{amt} : j\}]) \\ &= \text{square-up}(R[0\ ..\ k-1] +\!\!+ [\{\mathsf{clr} : R[k].\mathsf{clr},\ \mathsf{amt} : R[k].\mathsf{amt}\}]) \quad \text{since j = R[k].amt} \\ &= \text{square-up}(R[0\ ..\ k-1] +\!\!+ [R[k]]) \\ &= \text{square-up}(R[0\ ..\ k]) \end{aligned}$$

(f) Prove that P_5 implies Post.

Solution:

$$\begin{aligned} S &= \text{square-up}(R[0\ ..\ k-1]) \\ &= \text{square-up}(R[0\ ..\ R.\text{length} - 1]) \quad \text{since } k = R.\text{length} \\ &= \text{square-up}(R) \end{aligned}$$

2. (15 points) **In One Fell Loop**

Fill in the missing parts of the implementation of `Rectify`. Your code must be correct with the **provided invariant**. (You do not need to turn in a proof, but it must be correct.)

Items (3-4) in the invariant are exactly the same as in the postcondition. Item (2) is a weakening of the first line of the postcondition. Only item (1) in the invariant is new.

```
/**
 * Returns an array of rectangles representing the same line as S.
 * @param S An array of squares
 * @returns R such that S = square-up(R) and
 *     R[j].amt >= 1 for any 0 <= j <= R.length - 1 and
 *     R[j].clr != R[j+1].clr for any 0 <= j <= R.length - 2
 */
function Rectify(S: readonly Color[]): Rect[] {
  if (S.length === 0)
    return [];

  const R: Rect[] = [(S[0], 1)];
  let k: number = 1;

  // Inv: (1) 0 < k <= S.length and
  //      (2) S[0 .. k-1] = square-up(R) and
  //      (3) R[j].amt >= 1 for any 0 <= j <= R.length - 1 and
  //      (4) R[j].clr != R[j+1].clr for any 0 <= j <= R.length - 2
  while (k !== S.length) {
    if (S[k].clr == S[k-1].clr) {
      R[R.length-1].amt += 1;
    } else {
      R.push({clr: S[k], amt: 1});
    }
    k = k + 1;
  }

  return R;
}
```

3. (18 points) **Wicked Witch of the Test**

Fill in the body of the following unit test for `Rectify`. Include comments explaining the test cases, as we did in the coding homework problems.

There is only space for 6 test cases below. If our heuristics give more than 3 subdomains to your code, choose your test cases to cover **as many distinct subdomains as possible**, rather than having two test cases for just 3 subdomains.

```
// first branch
assert.deepStrictEqual(
    Rectify([]),
    []);

// 0 times through the loop
assert.deepStrictEqual(
    Rectify(["red"]),
    [{clr: "red", amt: 1}]);

// 1 time through the loop, top branch
assert.deepStrictEqual(
    Rectify(["red", "red"]),
    [{clr: "red", amt:2}]);

// 1 time through the loop, bottom branch
 assert.deepStrictEqual(
    Rectify(["red", "white"]"),
    [{clr: "red", amt: 1}, {clr: "white", amt: 1}]);

// many times through the loop. top branch
assert.deepStrictEqual(
    Rectify(["red", "white", "white"]),
    [{clr: "red", amt: 1}, {clr: "white", amt: 2}]);

// many times through the loop, bottom branch
assert.deepStrictEqual(
    Rectify(["red", "white", "blue"]),
    [{clr: "red", amt: 1}, {clr: "white", amt: 1}, {clr: "blue", amt: 1}]);
```

The next two problems involve the implementation of the following ADT. It represents a line, which clients think of as an array of squares.

```
/** An array of colors (i.e., squares). */
interface Line {
  /**
   * Adds the given color to the end of the array.
   * @modifies obj
   * @effects obj = obj_0 ++ [c]
   */
  add: (c: Color) => void;

  /**
   * Removes and returns the last color in the array.
   * @requires obj.length > 0
   * @modifies obj
   * @effects obj ++ [c] = obj_0
   * @returns a color c
   */
  remove: () => Color;
}
```

We will implement `Line` with the following class, whose concrete representation is an array of rectangles rather than an array of squares.

The representation invariant contains the same facts as the postcondition of `Rectify`: no empty rectangles and no adjacent rectangles of the same color.

```
class RectLine implements Line {

  // RI: this.rects[j].amt >= 1
  //         for any for any 0 <= j <= this.rects.length - 1
  // and this.rects[j].clr != this.rects[j+1].clr
  //         for any 0 <= j <= this.rects.length - 2
  // AF: obj = square-up(this.rects)
  rects: Rect[];

  // Creates obj = squares
  constructor(squares: readonly Color[]) {
    // The postcondition of Rectify ensures obj = squares and the RI
    this.rects = Rectify(squares);
  }

  ...
```

4. (12 points) **Line-Craft**

Consider the following code, which claims to implement add from the prior page.

{{ **Pre**: this.rects $=$ this.rects$_0$ }}
```
const last = this.rects.length - 1;
if (last >= 0 && this.rects[last].clr == c) {
  this.rects[last] = {clr: c, amt: this.rects[last].amt + 1};
```
{{ P_1: last ≥ 0 and this.rects$_0$[last].clr $= c$ and
  this.rects $=$ this.rects$_0$[0 .. last $- 1$] $+\!\!+$ [{clr : c, amt : this.rects$_0$[last].amt $+ 1$}] }}
```
} else {
  this.rects.push({clr: c, amt: 1})
```
{{ P_2: (last < 0 or this.rects$_0$[last].clr $\neq c$) and
  this.rects $=$ this.rects$_0$ $+\!\!+$ [{clr : c, amt : 1}] }}
```
}
```

(a) P_1 and P_2 have been filled in above using forward reasoning. Explain briefly, in English, why P_2 is the correct assertion according to forward reasoning.

Solution: The first line holds since we entered the "else" branch. The second line follows from the precondition: pushing a new element changes the array to be its original value concatenated with the newly pushed element.

(Continued on next page...)

(b) Prove that the postcondition holds, i.e., that obj $=$ obj$_0 +\!\!+ [c]$, **given** that P_2 holds.

Solution:

obj $=$ square-up(this.rects)	by AF
$=$ square-up(this.rects$_0 +\!\!+ [\{$clr : $c,$ amt : $1\}])$	by P_1
$=$ square-up(this.rects$_0 +\!\!+ [\{$clr : $c,$ amt : $0\}]) +\!\!+ [c]$	def of square-up
$=$ square-up(this.rects$_0$) $+\!\!+ [c]$	def of square-up
$=$ obj$_0 +\!\!+ [c]$	by AF

(c) Suppose we repeated parts (a-b) for P_1 as well. Explain why that would tell us that the postcondition always holds at the end of the code above.

Solution: Parts (a-b) showed that the postcondition holds at the end of the "else" branch. Doing the same with P_1 would tell us that it holds at the end of the "then" branch. Together, that would show that it holds after the "if" statement, no matter which branch we take.

(Continued on next page...)

(d) To prove that this method is correct, we also need to prove that the representation invariant still holds at the end. Explain *briefly*, in English, why the first part of the invariant — that all rectangles have `amt` at least 1 — is still true at the end.

Solution: Since the rep invariant was true initially, all rectangles start with an amount of at least 1. In the "then" branch, we increase the amount of the last rectangle, so it now must be at least 2, and all others are unchanged, so they are still at least 1. In the "else" branch, we don't change any of the existing rectangles, so their amounts are still at least 1, and the newly added rectangle has an amount of 1. Since the rep invariant is true at the end of both branches, it is true at the end of the method.

5. (12 points) **Your Guess Is As Good As Line**

Fill in the implementation of `remove` in `RectLine`. The specification of the method (from `Line`) is repeated here for your convenience.

Be sure to follow our usual rules for defensive programming.

```
/**
 * Removes and returns the last color in the array.
 * @requires obj.length > 0
 * @modifies obj
 * @effects obj ++ [c] = obj_0
 * @returns a color c
 */
remove = (): Color => {
  if (this.rects.length === 0)
    throw new Error("cannot remove from empty array");

  const last = this.rects[this.rects.length - 1];
  if (last.amt > 1) {
    last.amt -= 1;
  } else {
    this.rects.pop();
  }
  return last.clr;
};
```

6. (16 points) **Life In the Last Lane**

Write a short answer to each of the following questions.

(a) Our `RectLine` class is mutable. If a method of some class takes a `RectLine` as an argument and wants to store it in a field of the class, what must the code do to ensure that any invariants involving that field are not violated by the code of other classes?

> **Solution:** They must copy it and store the copy instead.

(b) Suppose that, in order to avoid the problems just mentioned, we created an immutable version of `RectLine`, called `ImmutableRectLine`, and added a method to `RectLine` that would return an `ImmutableRectLine` with the same abstract state. What design pattern would `RectLine` then be an example of?

> **Solution:** Builder

(c) In lecture, we discussed the Visitor pattern, which is potentially useful for any "tree-like" inductive data-type. What pattern provides the most analogous functionality for "list-like" inductive data types?

> **Solution:** Iterator

(d) For two concrete data types A and B, we said that "B is a *subtype* of A" when the set of values allowed by type B is a *subset* of the set of values allowed by type A. As briefly as possible, how did we define "B is a subtype of A" when both A and B are ADTs?

> **Solution:** We used substitutability for ADTs. (Or equivalently, B must have all the methods of A and each corresponding method of B must have a stronger spec than the method of A.)

Consider the following specification for `Rectify2`:

```
// @returns R such that S = square-up(R)
function Rectify2(S: readonly Color[]): Rect[] { ... }
```

It differs from `Rectify` by removing the parts of the postcondition that require R to have no empty rectangles and no adjacent rectangles of the same color.

(e) Is the original `Rectify` specification weaker, stronger, or incomparable to `Rectify2`? Explain your answer.

Solution: `Rectify` is stronger because the postcondition is stronger.

Consider the following specification for `SquareUp2`:

```
// @requires R[j].amt >= 1 for any 0 <= j <= R.length - 1 and
//       R[j].clr != R[j+1].clr for any 0 <= j <= R.length - 2
// @returns square-up(R)
function SquareUp2(R: readonly Rect[]): Color[] { ... }
```

It differs from `SquareUp` by including a precondition that R has no empty rectangles and no adjacent rectangles of the same color.

(f) Is the original `SquareUp` specification weaker, stronger, or incomparable to `SquareUp2`? Explain your answer.

Solution: `SquareUp` is stronger because its precondition is weaker.

Consider the following specification for `SquareUp3`:

```
// @requires R.length > 0
// @returns S = square-up(R) with S.length > 0
function SquareUp3(R: readonly Rect[]): Color[] { ... }
```

It differs from `SquareUp` by including a precondition that R is non-empty and a postcondition that S (the array returned) is also non-empty.

(g) Is the original `SquareUp` specification weaker, stronger, or incomparable to `SquareUp3`? Explain your answer.

Solution: The two are incomparable because `SquareUp3` weakens in one way (precondition) and strengthens in another (postcondition).

Suppose that a colleague writes code to display a line on the screen as HTML, but he declares his function as follows:

```
type Stroke = {clr: Color, amt: number};

function DrawLine(strokes: Stroke[]): JSX.Element { ... }
```

His `Stroke` is the same as your `Rect` except for the name.

(h) In TypeScript, can you legally pass a `Rect[]` to your colleague's function? Explain why or why not.

> **Solution:** Yes, you can. The names are just aliases for the underlying record type, which is the same.

Now, suppose that you were working in Java and declared these types as classes. For example, the type above would become

```
class Stroke { public Color clr; public int amt; }
```

`Rect` would also be declared as a class and would have the same fields.

(i) In Java, can you legally pass a `Rect[]` to your colleague's function? Explain why or why not.

> **Solution:** No, you cannot. Java uses nominal typing, so it does not consider these the same even though they have the same structure.

(j) What is one of the ways in which being an early engineer at a startup company can be better than the other career paths discussed by our guest lecturer, Ken Horenstein?

> **Solution:** It can have higher financial upside, you are given a lot more responsibility, and you usually learn a lot more.