CSE 331: Software Design & Implementation

Homework 2 (due Wednesday, October 11th 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. Your final version of `funcs.ts` should be your solution to the "HW2 Coding" assignment, and the collection of all written answers should be your solution to "HW2 Written". See the Homework Turn-in Guide for more details.

1. The Level Is in the Details (10 points)

The following parts describe written work.

Identify the Correctness Level that corresponds to the given code and write a one sentence explanation.

(a) Suppose that we have the following TypeScript function, defined on pairs of colors, where each color is either red, green or blue.

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

const f = (c1: Color, c2: Color): string => { ... }
```

What level of correctness is required for this function?

(b) Consider the following mathematical function defined on the non-negative integers:

$$\textbf{func } f(n) := 2\,g(n) + 3$$

where g is a *recursive* function. If f is implemented in TypeScript as a single statement returning the expression above, what level of correctness is required for f?

Write a short answer to each of the following questions.

(c) Your homework assignment includes a problem, described in English, that asks you to calculate some integer value from an integer input. You have no idea how to do it, so you ask ChatGPT for a program that solves it, and magically, it gives you back a program. Unfortunately, it's written in "Comfy", a language you have never heard of and cannot understand, but you want to make sure it's correct. What level of correctness does this fall into? If multiple are possible, choose the worst case (highest level).

(d) Unable to interpret the initial program it gives you, you decide to ask ChatGPT specifically for a TypeScript program and it magically gives you one. Looking at the code, you see that it involves loops and that local variables are mutated, but there isn't any mutation of heap-allocated data. What level of correctness does this fit into?

(e) Suppose we have two problems on a homework assignment that are described in English. The first of those problems has a short English explanation—just a single sentence—while the second problem's explanation is almost a full page. Is it fair to assume that the first problem has a lower correctness level? If so, why? If not, how could that not be the case?

2. Test Friends Forever (12 points)

The following parts describe written work. Write a short answer to each question.

(a) Suppose that we are implementing the following function, where a is an integer and b a boolean:

$$\textbf{func } f(a, b) := (2a, b)$$

What potential bug in the TypeScript implementation makes it necessary to test both $b = \mathsf{T}$ and $b = \mathsf{F}$? (Either write out the code with the bug or describe it in detail.)

(b) In part (c) of the previous problem, suppose that the Comfy program that ChatGPT wrote takes a list of integers as input and returns a list of integers. How many tests would we need to try to guarantee that it is correct?

For the remaining parts, suppose that, after showing our application to some users and getting feedback, we decide that we need to make some changes in one of our functions so that it calculates something different.

(c) Suppose that the only change made was replacing one arithmetic expression (appearing in some straight-line code, conditional, or function call) with another expression. Describe some circumstances where that change would require us to also change the *inputs* that we test.

(d) With the same setup as (c), describe some circumstances where we can be certain that the change would not require us to change the inputs that we test.

(e) Suppose that the only change we made was replacing a call to one function with a call to another within some straight-line calculation. Is it possible that we would need to change our test *inputs*? Explain.

(f) In which of the previous cases may we need to change the expected *outputs* in the test cases?

3. Many a Truth Is Spoken in Test (20 points)

The following parts describe coding work.

If you have not already performed the installation steps described at the beginning of the "Coding Setup" page in the section worksheet, you should do so now. Then, you can start this problem by checking out the starter code using the command

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

Install the modules using `npm install --no-audit`. You can run the tests using `npm run test`.

For each of the following functions in `funcs.ts`, fill in the body with code that passes the tests in `funcs_test.ts` but is wrong. Your implementation must follow any additional requirements given below and may not call any other functions.

Note that each part includes restrictions on the type of code that is allowed. No part allows anything beyond straight-line code, conditionals, and recursion. Straight-line code should not mutate anything!

Also note that your solutions to each part do not need to be very long. For example, my solutions to parts (a-c) have no more than 5 lines and to parts (d-e) have no more than 8 lines.

(a) Implement the function `quadratic3` using a straight-line calculation.

Your code must pass the first two test cases but still be wrong. This will demonstrate that, in general, two tests does not guarantee that even a straight-line calculation is correct. That is the *minimum* number of tests that we must write, but it does not guarantee correctness. (See the Extra Credit for more info.)

(b) Implement the function `abs_value3` using a conditional and straight-line calculations in each branch.

(c) Implement the function `abs_value4` using two conditionals, where each branch condition is a simple numeric comparison and the body of each branch, if present, is a straight-line calculation. You can leave out an `else` branch entirely if you want. You cannot explicitly return `undefined`. Instead, the code should return `undefined` implicitly by reaching the end of the body without executing a `return` statement.

These restrictions are intended to make your solution look natural, like a bug that might really happen. If that is not the case, think some more about how to solve it.

Note that you may see a warning in VSCode with a correct solution. That is good! It demonstrates that tools are useful to help you spot the bug you wrote.

(d) Implement the function `count_pairs`, which is defined only for non-negative integer inputs, using separate branches for 0, positive even numbers, and positive odd numbers. The base case should perform a straight-line calculation and the positive even and odd branches should make a single recursive call. Both the odd and even branches should make recursive calls on even numbers. Do not use division (mod is okay).

(e) Implement the function `count_pairs2` using the same rules as `count_pairs`.

`count_pairs2` needs to work correctly on 3 but fail on 5, whereas `count_pairs` failed already at 3. You will need to get more creative to do this!

Hint: we saw in part (a) that it is possible for a straight-line calculation to work properly on two cases but still fail for others in a problem involving quadratics.

4. Keep Your Cards Close to the Test (15 points)

The following parts describe coding work.

Translate each of the following functions into TypeScript and add them to `funcs.ts`. Once each function is implemented, you can uncomment the tests for that function in `funcs_test.ts` and confirm that your translation works properly by running `npm run test`.

(a) The following rules define $u : \{b : \mathbb{B}, n : \mathbb{N}\} \to \mathbb{Z}$, which has a record as input and an integer as output:

$$\begin{array}{lll} \textbf{func } u(\{b\text{: } b,\ n\text{: } 0\}) & := 0 & \text{for any } b : \mathbb{B} \\ \quad u(\{b\text{: } \mathsf{T},\ n\text{: } n+1\}) & := n+1 & \text{for any } n : \mathbb{N} \\ \quad u(\{b\text{: } \mathsf{F},\ n\text{: } n+1\}) & := -(n+1) & \text{for any } n : \mathbb{N} \end{array}$$

Note that these rules are exclusive (and exhaustive)! The second and third rules only apply if the "n" field of the record is not 0.

(b) The following rules define $v : \mathbb{N} \mid (\mathbb{B} \times \mathbb{N}) \to \mathbb{N}$, whose input is either a non-negative integer or a tuple containing a boolean and a non-negative integer and whose output is a non-negative integer:

$$\begin{array}{lll} \textbf{func } v(n) & := n & \text{for any } n : \mathbb{N} \\ \quad v((b, 0)) & := 1 & \text{for any } b : \mathbb{B} \\ \quad v((b, n+1)) & := n+1 & \text{for any } b : \mathbb{B} \text{ and } n : \mathbb{N} \end{array}$$

(c) The following rules define $w : \mathbb{N} \times \{b : \mathbb{B}, n : \mathbb{N}\} \to \mathbb{Z}$, which has a tuple as input (with its second part being a record) and an integer as output:

$$\begin{array}{lll} \textbf{func } w((m, \{b\text{: } \mathsf{T},\ n\text{: } n\})) & := m+n & \text{for any } m, n : \mathbb{N} \\ \quad w((m, \{b\text{: } \mathsf{F},\ n\text{: } n\})) & := m-n & \text{for any } m, n : \mathbb{N} \end{array}$$

Note that you should make sure your tests pass without modification to the names or parameters of the function calls (meaning your implementation of the function declarations should match). We will be running additional tests on your code that rely on the declarations to be the same also.

5. What in Notation Is Going On Here? (15 points)

The following parts describe written work.

For each of the following functions, translate the code into our math notation using pattern matching. Unless specification is given, you *can* assume typescript types behave in these functions with no additional restriction. (Note that this code uses array indexing on tuples, which is against our coding conventions. This problem is a good example of why we don't allow it!)

(a)
```
/** @param t consisting of a boolean and a non-negative integer */
const u = (t: [boolean, number]): number => {
  if (t[1] === 0) {
    return 1;
  } else if (t[1] === 1) {
    return 2;
  } else {
    return 3 + u(t[0], t[1]-1);
  }
};
```

(b)
```
const v = (t: [number, [boolean, number]]): [number, number] => {
  if (t[1][0]) {
    return [t[1][1], t[0]];
  } else {
    return [t[0], t[1][1]];
  }
};
```

(The math version should be much easier to read!)

(c)
```
const w = (t: {s: [boolean, number], b: boolean}): number => {
  if (t.s[0] === t.b) {
    return t.s[1];
  } else {
    return -t.s[1];
  }
};
```

Make sure your rules are *exclusive* and exhaustive!

6. Good, Better, Test (18 points)

The following parts describe written work.

For each of these functions, state the number of tests required and explain why that is the required number. Then, describe a specific set of tests to use (with the same number of tests you as said before). Describe each test by giving the input (identify a specific input rather than saying, e.g., "some positive number"), saying what portion of the function it tests, and explaining why our rules require that test. Please note that you are describing a set of tests, which by nature, should include only tests that are distinct (no repeats).

(a)
```
type Color = "red" | "green" | "blue";
```

```
const f1 = (color?: Color): number => {
  switch (color) {
    case "red":   return 1;
    case "green": return 2;
    case "blue":  return 3;
    default:      return 0;
  }
}
```

(b)
```
const f2 = (color: Color, x: number): number => {
  switch (color) {
    case "red":   return x;
    case "green": return -x;
    case "blue":  return 0;
  }
}
```

(c)
```
const f3 = (color1: Color, color2: Color): number => {
  return f2(color1, 5) + f2(color2, 7);
}
```

(d) The following function takes an array as input. (As in Java, "`number[]`" means an array of numbers.)

```
const f4(A: number[]): number => {
  if (A.length === 0) {
    return 0;
  } else {
    return A[0] + f4(A.slice(1));
  }
}
```

The `slice` method returns a subarray. You can read more about it here.

(e) Assume the following function is defined only for *integer* values of `n`:

```
const f5 = (n: number): number => {
  if (n <= 0) {
    return 0;
  } else {
    return f5(n-1) + 2*n - 1;
  }
}
```

(f) Assume the following function is defined only for *non-negative integer* values of `n`:

```
const f6 = (n: number): number => {
  if (n === 0) {
    return 0;
  } else if (n % 3 === 0) {   // n > 0 is a multiple of 3
    return f6(n-3) + 1;
  } else if (n % 3 === 1) {   // n - 1 is a multiple of 3
    return f6(n-1);
  } else {                    // n - 2 is a multiple of 3
```

```
    return f6(n-2);
  }
}
```

7. Extra Credit: Galaxy Test (0 points)

We saw in Problem 2 that two examples are not always enough to ensure that straight-line code is correct. In this problem, we will examine this situation in more detail under the assumption that our function takes a single, numeric argument `n` and that the expressions use only the operations `+`, `-`, and `*`.

```
function f(n: number): number {
  const x_1 = ...;
  const x_2 = ...;  // only use +, -,  * applied to constants, n, and these variables
  ...
  const x_N = ...;
  return x_N;
}
```

(a) [0 Points] Explain why the value returned can instead be written as a polynomial in the variable `n`, with no references to any other variables.

(b) [0 Points] How many tests are necessary to ensure that the return value is correct if it is described by a degree-k polynomial in `n`. (Assume that all calculations performed are exactly.)

(c) [0 Points] What needs to be true about k in order for two tests to detect any bug present?