CSE 331: Software Design & Implementation

Homework 2 (due Wednesday, April 10th 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 collection of all written answers should be your solution to "HW2 Written", and the following completed files should be directly submitted for the "HW2 Coding" assignment:

`funcs.ts` `funcs_test.ts`

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, yellow or blue.

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

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

What level of correctness is required for this function?

(b) Consider the following mathematical function:

$$\begin{aligned} \textbf{func } f(0) &:= 0 \\ f(n+1) &:= f(n) - 1 \quad \text{for any } n : \mathbb{N} \end{aligned}$$

It is implemented in TypeScript as follows:

```
const f = (n: bigint): bigint => return -n;
```

What level of correctness is required for f?

Write a short answer (maximum 2 sentences) to each of the following questions.

(c) Suppose you were telling your friend about how you're learning about code correctness in 331. They ask you to look at their code for a personal project they're working on so that you can recommend some strategies to verify that their code is correct. You take a look at one of their functions and notice that it is in TypeScript, it takes two bigints as input, it has no specification, and it mutates local variables (but no arrays or heap state). Which steps should your friend take to determine if their code is correct?

(d) 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).

(e) Suppose you are given a function implemented according to an imperative specification which has the parameter `(x: bigint)` and returns a `boolean`, and a function with a declarative specification which edits local variables and has the parameter `(c: Color)`. Does the function with the imperative or the declarative specification have a higher correctness level? Why?

2. Test Friends Forever (10 points)

The following parts describe written work. Write a short answer to each question (max 3 sentences).

(a) In part (d) 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 write 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.

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

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

(d) 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.

(e) 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 (14 points)

The following parts describe written work. Write a short answer for each question (max 3 sentences).

When we reason through code, in a basic code review or with the more formal tools we'll learn later on, it is valuable to be on the lookout for small details and common mistakes. The goal with these question is to help you get into a investigative, slightly paranoid mindset and practice finding sneaky common bugs in your code.

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

$$\textbf{func } f(a, b) := (3a, b)$$

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

(b) Suppose we are implementing the following function, where n and m are integers:

$$\textbf{func } g(n, m) := n^2 - m$$

If the TypeScript implementation for this function was a single return statement, what potential bug would not be caught with the test cases (n = 0, m = -1) and (n = -3, m = 2)? (Either write out the code with the bug or describe it in detail.)

(Side note: exponents are implemented in TypeScript with **)

(c) Suppose we implement the following function to correctly compute $|x|$:

```
export const abs_value = (x: bigint): bigint => {
    if (x < 0n) {
        return -x;
    } else {
        return x;
    }
}
```

How could the conditional of this code been written with a common typo such that it is incorrect for x = -1, but correct for all other inputs? If followed, which of our testing heuristics would have caught this bug?

(d) Suppose we implement the following function to attempt to compute $|m| + 2m$:

```
export const h = (m: bigint): bigint => {
    if (m <= -1n)
        return -m + 2n;
    if (m >= 1n)
        return m + 2n;
}
```

What is a bug in this code that the type checker could help us identify? What is a bug in this code that the type checker could not help us identify?

Considering the following function which recursively calculates the number of integer pairs contained in the input `n` (e.g. there are 4 pairs in 8 and 2 pairs in 5), and is defined only for non-negative integer inputs. The next two parts apply to this function.

```
export const count_pairs = (n: bigint): bigint => {
    if (n === 0n) {
      return 0n;
    } else if (n % 2n === 0n) {  // n > 0 is even
        return count_pairs(n - 2n) + 1n;
    } else {  // n > 0 is odd
        return count_pairs(n - 1n);
    }
}
```

(e) Describe a scenario while implementing `count_pairs` which would have caused all odd input cases to result in an output that is off-by-one?

(f) Suppose we have a single typo in the return value of the `n === 0n` branch that makes the output for n = 0 *incorrect*. Why is it impossible for other inputs to produce the *correct* result given that n = 0 produces an incorrect one? What does this tell us generally about testing base case for recursive functions? Why do we still care about testing base cases?

4. Keep Your Cards Close to the Test (18 points)

The following parts describe coding work.

Start this problem by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-24sp/cse331-24sp-materials/hw-levels.git
```

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

Translate each of the following functions into TypeScript and add them to `funcs.ts`. Your translations should be **'straight from the spec'**, meaning it should be a direct translation of the cases of the math definitions to code. 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 $r : \{n : \mathbb{N}, m : \mathbb{N}\} \to \mathbb{Z}$, which has a record as input and an integer as output:

$$\begin{array}{lll} \textbf{func } r(\{n\colon 0,\ m\colon m\}) & := 1 & \text{for any } m : \mathbb{N} \\ \quad r(\{n\colon n+1,\ m\colon 0\}) & := -1 & \text{for any } n : \mathbb{N} \\ \quad r(\{n\colon n+1,\ m\colon m+1\}) & := 0 & \text{for any } n, m : \mathbb{N} \end{array}$$

Note that these rules are exclusive (and exhaustive)! The first rule applies in any case when the "n" field is 0, regardless of the "m" field, the second rule applies only if the "n" field of the record is not 0 and the "m" field is 0, and the last case applies to cases when both are not 0.

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

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

(c) The following rules define $t : \mathbb{B} \times \{n : \mathbb{R}, m : \mathbb{R}\} \to \mathbb{R}$, which has a tuple as input (with its second part being a record) and a real number as output:

$$\begin{array}{lll} \textbf{func } t((T, \{n\colon n,\ m\colon m\})) & := n * m & \text{for any } m, n : \mathbb{R} \\ \quad t((F, \{n\colon n,\ m\colon m\})) & := n - 2m & \text{for any } m, n : \mathbb{R} \end{array}$$

Note that you should implement your TypeScript functions with the same function and parameter names as specified in the math definitions and make sure that the given tests pass without modification. We will be running additional tests on your code that also rely on the Typescript function declarations being the same.

5. What in Notation Is Going On Here? (18 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.

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

(b)
```
const v = (t: [bigint, [boolean, bigint]]): [bigint, bigint] => {
  const [i, [b, j]]: [bigint, [boolean, bigint]] = t
  if (b) {
    return [j, i];
  } else {
    return [i, j];
  }
};
```

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

Make sure your rules are *exclusive* and exhaustive!

6. Good, Better, Test (15 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).

(Hint: switch statement cases are conditionals branches just like if/else conditions.)

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

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

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

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

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

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

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

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

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

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

7. Tough Fact to Follow (15 points)

The following parts describe a mix of written and coding work. Parts (a-b) are written and parts (c-d) are coding.

In this problem we will practice a skill that you will use a lot in coming homeworks. We'll start with an english description, formalize that description into a mathematical definition, translate it into TypeScript 'straight from the spec,' similar to what you did in problem 4, then test it.

We'll be implementing a function that calculates a factorial. A factorial is denoted as $n!$ for some non-negative n, and is the product of all positive integers less than or equal to n. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$. 0 is a special case; there aren't any positive integers less than 0 so it doesn't follow the standard pattern, specifically $0! = 1$.

(a) The description above is an english description, so our first step is to formalize it into a mathematical definition.

Your mathematical function should be named "fact" and take a non-negative integer as input. It should be defined recursively with pattern matching. You will need 1 base case, and in the recursive branch, you should call fact on the (given input - 1) and build on that result.

(b) Show how your mathematical definition would execute fact$(5) = 120$ by writing out the sequence of recursive calls. Include the arguments and what is returned for each recursive call.

Use any sensible notation to clearly show the sequence of calls.

Do not move on to the next part until you are confident that your mathematical definition in part (a) correctly implements the function. Completing part (b), and running through other examples, can help convince you.

The goal of this course is to help you think and understand code, so these steps on paper to understand the logic around a problem before getting the help of a computer is valuable practice. Building these skills is necessary to be successful while interviewing for jobs and proving its worth paying you to do them. :)

(a) Now, translate your mathematical definition into TypeScript code in `funcs.ts`. We have provided a function stub for you to fill in with the following shape:

```
export const fact = (n: bigint): bigint => {...}
```

This translation must be 'straight from the spec.' There may be multiple ways to implement this function, but your code should match your mathematical definition which should follow the guidelines we gave above.

The motivation for implementing this code 'straight from the spec', is that it falls in a lower correctness level than if we deviated from it. To verify it is correct, it is sufficient to do simple code review and write test cases, which we will do next.

(b) Write tests for `fact` in `funcs_test.ts` according to the heuristics we've discussed in class.

In `funcs_test.ts` there are some helpful comments for the syntax for writing test cases and some example tests to get you started.

Include a brief comment above each test case in your code, justifying why you chose it. See page 10 for more instructions and an example for test justification comments.

Confirm that your tests all pass by running npm run test.

8. 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: bigint): bigint {
  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?

Writing Test Cases

In this homework, we ask you to write test cases based on the heuristics we have learned in class. For more details about how to use assert statements and the format of test files, see the inline comments in `funcs_test.ts`.

For each test case, we ask that you include a comment above in your code justifying why you chose to include that case. This justification should cite a testing heuristic or explain what plausible bug could have been missed without the case. This comment can be brief; certainly not more than a sentence.

Below is an example function with tests written according to our heuristics and comment justifications that are the length and level of detail that we expect from you. You don't have to follow this format exactly, but this is the right idea.

`example.ts`:

```
const f = (x: bigint): bigint => {
    if (x === 0n) {
        return 0n;
    } else {
        return 2n * f(x - 1n) + 1n;
    }
};
```

`example_test.ts`:

```
import * as assert from 'assert';
import { f } from './example'

describe('example', function() {
    it('testf', function() {
        // 0-1-many heuristic, base case
        assert.deepStrictEqual(f(0n), 0n);

        // 0-1-many heuristic, 1 recursive call (only 1 possible)
        assert.deepStrictEqual(f(1n), 1n);

        // 0-1-many heuristic, more than 1 recursive call
        assert.deepStrictEqual(f(2n), 3n);
        assert.deepStrictEqual(f(3n), 7n);
    });
});
```