CSE 331: Software Design & Engineering Fall 2024

Homework 4 Code – Solutions

If you have not completed HW4 written. Close this file and finish the written component before starting the coding parts in this worksheet.

To get started, check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-24au-materials/hw4-maze.git
```

Navigate to the `hw4-maze` directory and run `npm install --no-audit`. For this assignment, there will be no client side app or server to run. Instead, we have provided (and will ask you to write) unit tests. To run the tests, use the command `npm run test`.

Linter

Starting with this assignment, we will also be using a custom linter, comfy-tslint, which checks that your code follows our course-specific coding conventions. The linter is not able to catch every coding convention mistake, so you should write code that looks similar to the examples we go over in class, and refer to the coding conventions document released last week for a written explanation of our expectations.

We will run the linter on your code through the autograder when you submit on Gradescope, so you should make sure it also passes locally. You can always run the linter in the command line with `npm run lint`, or you can download the recommended VS Code extension for the linter. The linter warnings will appear as helpful popups while you're coding (similar to the type checker). If you run the command, the output will always have the first two lines listed in the image below, and if your code has any errors, those will be listed below.

```
> cse331-hw-fib@0.0.1 lint
> comfy-tslint --no-mutation src/*.{ts,tsx}

src/fib.ts:7:32: any type is not allowed
src/index.tsx:6:10: top-level variable declarations must have a type
```

One major check the linter will perform is that you **do not mutate any variables for this assignment**. The following problems will ask you to write tests and translate your math definitions directly to code, neither of which should require any mutation. Our next few assignments will have the same requirement.

Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW4 Code"**:

`pell.ts` `designs.ts` `funcs_test.ts`

After you submit your work, an autograder will run which verifies you have submitted the correct files, runs the linter, and runs tests (including those you submit, the tests we provide in the starter code, and some additional staff tests). As usual, the autograder is worth points, so you should wait until the autograder completes to make sure it passes, and otherwise resubmit. Meaning you should **leave enough time** to fix possible issues before the deadline. As usual, we will also manually grade your code (including test cases).

Task 6 – Get the Pell out of Dodge [12 pts]

In this problem, we will translate mathematical definitions for functions into TypeScript code.

Specifically, you will translate math definitions that you wrote in the written Tasks of this assignment. We will treat the math definitions as the imperative specifications for the TypeScript functions, so the translations should be "straight from the spec" –a direct translation.

We have provided tests for these functions based on their correct behavior as described in the English/picture descriptions from the written tasks. You should run these tests to get a good idea of if your implementations are correct using the command `npm run test`.

If the tests fail, indicating a bug in your TypeScript functions, you should fix these bugs to try to get the tests to pass. However, if the bug was due to a mistake in your mathematical definition (rather than a typo or mistake in translation), we will allow you to explain what went wrong to earn some points back on those mistakes and avoid being penalized for not writing a direct translation

To do this, write a brief paragraph (3–4 sentences max is sufficient) within an inline (`//` style) comment at the top of the function explaining how your mathematical definition was incorrect and where the TypeScript function deviates to fix this mistake.

(a) Translate your mathematical definitions from HW4 Written Task 2, functions p and s, into TypeScript code in `pell.ts`.

The provided tests for this function are in `pell_test.ts`. These tests and the tests for part (b) are examples of the minimum required tests (according to our course testing guidelines) for these functions.

(b) Translate your mathematical definitions from HW4 Written Task 4(b), the recursive functions for each maze design, into TypeScript code in `designs.ts`.

You should only translate the *recursive* definitions that you wrote in part **(b)**, you *do not* need to translate the 4×2 versions from part (a). Please maintain the order of the parameters as given in those declarations, even if it deviates from the order in your mathematical definition, as it is required for testing.

The types and helper functions related to mazes (as described in the HW4 Written spec) are translated to TypeScript for you in `maze.ts`; import those to `designs.ts` and use as needed. The provided tests for this function are in `designs_test.ts`.

Task 7 – Test Friends Forever [12 pts]

Now that you've written some TypeScript code and tested it, it is your turn to write some tests!

In `funcs.ts`, there are 9 functions (fun fact: the first 3 are the same as the functions from HW4 Written Task 1!) that you must write tests for. Your tests should follow the testing requirements we have described in lecture and in our testing notes summary (also linked on the website "Topics" page).

Write your tests for each function in `funcs_test.ts`.

Additionally, write short labels describing which coverage requirement is met by each test. See the below example function and tests (from this week's section, Task 3):

```
const twice = (L: List): List => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return cons(2 * L.hd, twice(L.tl));
  }
}
```

In test file:

```
it("twice", function() {
  // Statement coverage: [] executes 1st return, [3] executes 2nd
  assert.deepStrictEqual(twice(nil), nil);
  assert.deepStrictEqual(twice(cons(3, nil)), cons(6, nil));

  // Branch coverage: covered above, [] executes 1st branch, [3] executes 2nd

  // Loop/recursion coverage, 0 case: covered above by []
  // Loop/recursion coverage, 1 case: covered above by [3]

  // Loop/recursion coverage: many case
  assert.deepStrictEqual(twice(cons(1, cons(2, cons(3, nil)))),
    cons(2, cons(4, cons(6, nil))));
});
```

Notice how you don't need to add additional tests if previous tests cover multiple requirements, just make sure there are clear comments for the required coverage areas. You are welcome to organize your comments differently than the example or use different wordings, just make sure all necessary details are conveyed.