CSE 331: Software Design & Implementation

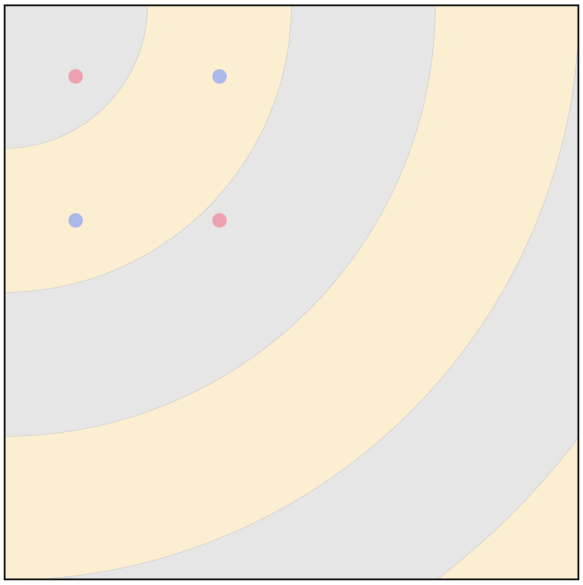
Section 5

This section starts with coding problems. To get started, check out the starter code using the command

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

Then, install the modules using `npm install --no-audit`.

The application allows the user to type in the coordinates for a list of points and then draws them on a canvas as shown in this picture.



The background of the canvas is striped to show distance from the origin (the upper-left corner). Colors are drawn differently depending on whether they are in a blue or beige stripe.

You can start the app with `npm run start` and open it at `http://localhost:8080`. To draw the picture above, input the points:

100 100
100 300
300 100
300 300

1. Let's Blow this Point

The application currently stores each `Point` as a (readonly) pair `[number, number]`. In this problem, we will replace it with an ADT so that we have the freedom to change the representation later.

(a) Replace `Point`, in `points.ts`, with an `interface` that has the operations `getX` and `getY`, each returning a `number`. Be sure to properly document both functions (with `@return` etc.).

(b) Change the following functions that use `Point`, which currently assume it is a pair, to instead use the new operations: `distToOrigin` in `points.ts` and `makeCircles` in `ui.tsx`.

(c) Create a class called `SimplePoint` that stores the coordinates in two fields, `x` and `y`, and implements the operations by returning the values in those fields. You will also need to include a `constructor`. Be sure to properly document the abstraction function with a comment of the form

```
// AF: obj = ...
```

(d) Change the function that creates points, `parsePointLines` in `points.ts`, to create an instance of the class instead of creating a pair.

(e) Change the tests to use the new class and interface. *Conveniently*, the tests in `points_test.ts` are already written to create points only using `parsePoints` and check their values only by converting them into strings in a function called `makeLines`. As a result, the only change left is to fix `makeLines`.

Make sure that all the tests still pass by running `npm run test`.

(f) Make sure that the app still works by running `npm run start`.

2. Making a Dist, Checking It Twice

Suppose that the application ends up being too slow because it makes a large number of calls to `distToOrigin`, each of which performs a complex square root calculation. In this problem, we will change the ADT to store this distance in the object rather than recalculating it each time.

(a) Add a new operation, `distToOrigin`, on the `Point` interface. Be sure to document it properly.

(b) Change `SimplePoint` to have a field that holds this distance. Fill in its value in the constructor and return it in the `distToOrigin` operation.

Update the class documentation. In particular, it should now include a representation invariant ("RI").

(c) Remove the old `distToOrigin` function and change the following uses to instead use the new operation: function `getPointsByDistToOrigin` in `points.ts` and test "distToOrigin" in `points_test.ts`.

(d) Make sure that the tests still pass and the app still works.

Lists

Recall the (non-generic) List type, which we defined as follows

$$\textbf{type}\ \text{List} \ := \ \text{nil} \ \mid \ \text{cons}(\text{hd} : \mathbb{Z},\ \text{tl} : \text{List})$$

Below, we will also need the length function defined recursively by

$$\begin{aligned} \textbf{func}\ \text{len}(\text{nil}) \quad &:= \ 0 \\ \text{len}(\text{cons}(a, L)) \quad &:= \ 1 + \text{len}(L) \quad \text{for any } a : A \text{ and } L : \text{List} \end{aligned}$$

Trees

The last problem makes use of the following inductive type, representing a *left-leaning* binary tree

$$\begin{aligned} \textbf{type}\ \text{Tree} := \ & \text{empty} \\ \mid \ & \text{node}(\text{val} : \mathbb{Z},\ \text{left} : \text{Tree},\ \text{right} : \text{Tree}) \quad \text{with height(left)} \geq \text{height(right)} \end{aligned}$$

The "with" condition is an *invariant* of the node. Every node that is created must have this property, and we are allowed to use the fact that it holds in our reasoning.

The height of a tree is defined recursively by

$$\begin{aligned} \textbf{func}\ \text{height}(\text{empty}) \quad &:= \ -1 \\ \text{height}(\text{node}(x, S, T)) \quad &:= \ 1 + \text{height}(S) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \text{Tree} \end{aligned}$$

In a general binary tree, the height of a non-empty tree is the length of the *longest* path to a leaf. With a left-leaning tree, we know the longest path is the one that always travels toward the left child.

3. Count From One to Len

Consider the function sep defined as follows:

$$
\begin{array}{lrll}
\textbf{func}\ \mathsf{sep}(\mathsf{nil}, x) & := & (\mathsf{nil}, \mathsf{nil}) & \\
\quad \mathsf{sep}(\mathsf{cons}(y, L), x) & := & (\mathsf{cons}(y, A), B) & \text{if } y \leq x \\
\quad \mathsf{sep}(\mathsf{cons}(y, L), x) & := & (A, \mathsf{cons}(y, B)) & \text{if } x < y \\
 & & \text{where } (A, B) := \mathsf{sep}(L, x) &
\end{array}
$$

A call to $\mathsf{sep}(L, x)$ returns a pair of lists (A, B), where A contains all the elements of L that are less than or equal to x and B contains all the elements that are greater than x.

Prove by induction on the list L that $\mathsf{len}(A) + \mathsf{len}(B) = \mathsf{len}(L)$, where $(A, B) = \mathsf{sep}(L, x)$. Note that, because the recursive case of sep is split into cases, you will need to handle the inductive step by cases as well.

4. One, Two, Tree...

We can define the size of a tree, the number of values stored in it, as follows:

$$\begin{aligned} \textbf{func}\ \mathsf{size}(\mathsf{empty}) &:= 0 \\ \mathsf{size}(\mathsf{node}(x, S, T)) &:= 1 + \mathsf{size}(S) + \mathsf{size}(T) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \mathsf{Tree} \end{aligned}$$

Prove by structural induction that, for any left-leaning tree T, we have

$$\mathsf{size}(T) \leq 2^{\mathsf{height}(T)+1} - 1$$