Review Session #2

Functions, Lists, Recursion

Objectives

- Be able to define and call functions
- Be able to work with the standard List functions
- Be able to understand the concept of recursion
- Be able to use recursion to solve problems

Function Review

- Definition / Implementation
- Calling

Function Definitions

- Parts of function definitions:
  - **Name:** the name of our function
  - **Parameters:** place holder variables for expected inputs
  - **Return type:** the type of value we want to return from the function
  - **Body:** the statements or lines of code that make up the function
    - This will often include return statements

- We define functions using the following syntax:

```
let <name> = (<parameters>): <return type> => {
    //function body
};
```

```
let applePie = (one:string, two:string) : string => {
    return “We combined “ + one + “ with ” + two + “ and now we have apple pie yummm”;
};
```

Calling functions

- To run the code in our functions, we call them
- If the function definition has parameters, we must pass values (arguments) into the function call
- Arguments must match parameters in type and order
- To call functions we use the following syntax:

```
<name>(<arguments>);
applePie(“apples”, “sugar”);
```

```
let applePie = (one:string, two:string) : string => {
    return “We combined “ + one + “ with ” + two + “ and now we have apple pie yummm”;
};
```

Lists

- What is a List?
- Ordered collection of values

- Ex: What do I need at the store?
- “Oranges” -> “Bananas” -> “Milk” -> `null`
- `null` means we have reached the end of our list. Done shopping!

Cons

- How do we make a list?
- cons (short for construct) is a **function** that we use to create a List.

- `// cons function definition`
- `let cons = (p1: <type>, p2: List<type>): List<type> => {`
  `// create a list and return it`
- `};`

Cons

- Usage:
- Let’s make a list of restaurants we need to go to on Franklin
- We want: “Bonchon” -> “Time Out” -> “TOPO” -> null

```
let restaurants: List<string>;
restaurants = cons(“Bonchon”, cons(“Time Out”, cons(“TOPO”, null)));
print(restaurants);
```

First

- How do we get individual items out of the list?
- Let’s take the first one item.

```
let studyOrder: List<string> = cons(“BIOL”, cons(“COMP”, null));
let someNumbers: List<number> = cons(12, cons(5, cons(42, null)));
print(first(studyOrder));      ← Prints BIOL
print(first(someNumbers));     ← Prints 12
```

Rest

- What if we don’t want only the first item?
- “Spiderman” “Superman” -> “Wonderwoman” -> null

Rest

- Usage:

```
let roster: List<string>;
roster = cons(“Paige”, cons(“Maye”, cons(“Berry”, null)));

print(first(roster) + “ just went pro!”);
print(rest(roster));
```

How could we get certain list values?

- What if we wanted, say, the third value in a list?
- “Spiderman” “Superman” -> “Wonderwoman” -> null

First Rest

“Superman” “Wonderwoman” -> null

First Rest

```
first(rest(rest(list)));
```

Recursion

- Base case
  - Need to change *something* about an argument to reach the base case
- Recursive case
  - Whenever we call a function inside of itself

Recursion motivation

- You've all experienced something where recursion would have been useful if you have ever refreshed the page to re-run code in 110.
- Let's take a look at lec05/03-includes-app.ts (Hands-on)

Recursion Example

- Let's construct a list using recursion

```
let clone = (input: List<number>): List<number> => {
      if (input === null) {
            return null;
      } else {
            return cons(first(input), clone(rest(input)));
      }
};
```

Recursion Example

- # of occurrences of an item in a list
- Declare a function named “count” that:
  - Takes in 2 parameters. One of type List<number> and one of type number
  - Returns a number, which is the # of times that number occurs in the list
  - Recursively goes through the list and counts how many times a number appears in the list.

Find Count

```
let count = (input: List<number>, find: number): number => {
      if (input === null) {
            return 0;
      } else {
            if (first(input) === number) {
                  return 1 + count(rest(input), number);
            } else {
                  return count(rest(input), number);
            }
}
```