Week 3 - Pretty Printing

Worksheet tips

You'll be coding in TypeScript. Unfortunately the browser can only understand JavaScript :(. This means you must type the command `tsc` in the command prompt or terminal every time you want to see your changes appear on the checklist. However you can use `tsc` with the `--watch` property to have your code transpiled whenever you save your file.

With the command line working directory at your project:

```
tsc --watch
```

Then in another command line or terminal, you can also run your JavaScript file with node like so:

```
node main.js
```

Similarly to last week, run the tests by opening the file worksheetChecklist.html in the Chrome browser.

VS Code can also be setup to compile typescript on save.

Outcomes

- Practice using and implementing types in TypeScript
- Implement and use: cons, head, rest, map, filter, reduce
- Create a program in a functional style
    - Using recursion, map, filter, reduce instead of loops.
    - Traversing simples trees.
- Get introduced to Classes

Worksheet constraints

- **Loops** are **not** allowed. Instead use your `map`, `filter`, `forEach` or `reduce` functions, as well as recursion.

Introduction

In this problem set we will explore the benefits of functional programming by building a simple pretty printer using the concepts shown in the lecture. We'll also spend time adding types to our code using TypeScript.

Pretty printers are often behind the scenes when you're coding. They help keep your code indented, keywords coloured and syntactic errors pointed out. Often they're used to enforce

a “style” keeping the code looking consistent. The programming language Go has a tool called ‘gofmt’ that formats your code. When writing Go it’s customary to format your code before sharing it (and often after every save). There’s a fantastic 8 minute talk about pretty printers in JavaScript here.

By the end of this workshop you’ll be able to pretty print N-ary trees, and apply this technique to pretty printing JSON (JavaScript Object Notation). Programming languages are often represented by an abstract syntax tree. Therefore the skills to pretty print trees could be used to pretty print a file of code.

First, we’ll expand on the functions seen in this week’s lecture and add type signatures.

Then we’ll use a TypeScript class to create a linked list that’s closer to what you’d see in Python but with generic types.

Next, you’ll be introduced to the pretty printer data model which you’ll be manipulating to create a pretty output.

Finally, using your functions you’ll visualise some trees and JSON (if you want)!

Exercise 1 - Typing numbers, strings and functions

This week you’ve seen some TypeScript. It’s like an expansion pack to JavaScript, introducing types and other cool things.

In main.ts there is some untyped JavaScript (in the tsconfig.json file we are allowing this to compile without complaint by setting noImplicitAny to false). Please add type annotations for each of these. You will know you have added complete type definitions when your code compiles with noImplicitAny set to true.

To help with typing your functions here are some examples:

```
// Typing the default syntax function
function add(a: number, b: number): number {
    return a + b;
}
```

The type highlighted in light blue is the type returned by the function.
If a function is assigned to a variable, it’s possible to show the type of the function using arrow functions.
I.e.

```
// The green part is the type. Then the implementation.
const add: (a: number, b: number) => number = (a, b) => a + b;
```

This arrow function type declaration can be used anywhere you want to define some function. Notice you need to add argument names. This helps create nice documentation as TypeScript can tell you the argument names expected as well as the types.

Example of adding types to an argument that's a function:
Using the function `operationOnTwoNumbers` defined in last week's worksheet:

```
const operationOnTwoNumbers = (f: (a: number, b: number) => number) =>
     (x: number) => (y: number) => f(x,y);
```

Soon you'll be seeing <T> and <V> everywhere. You can use any letter or any word you like. These are generic type signatures. If you define the following functions:

```
function identity (x) { return x };
```

You might want to allow different types of x to pass through. Therefore in typescript you can define x as a generic type.

```
function identity<T> (x:T):T { return x };
let aNumberVariable = identity(10);          // Here T becomes number
let aStringVariable = identity("Hello");     // Here T becomes string
```

The net result is that the generic function is just as easy to use (no more syntax to call) than the untyped or <any> typed version - but the compiler enforces type correctness, e.g. that the return type of identity matches the parameter type.

Exercise 2 - Map!

This week you've covered the following functions:

- cons
- head
- rest

You've been provided with these functions with their types added! Implement the `map` function (from the notes) and see how the types help catch errors!

```
console.log(head(map(v => v.length, cons("Hey!", null))));
> 4
```

Exercise 3 - More Manipulations

Again using the notes, please add types for and implement the following functions:

- fromArray (5.1)
- filter (5.2)
- reduce (5.3)

If you click on the test titles, you can see your functions in action!

Exercise 4 - Creating a List with types!

The `main.ts` file contains the skeleton of the List<T> class. This will allow composition of `map`, `filter` and `reduce` through fluent programming style chaining, e.g.:

```
new List([1,2,3,4,5])
    .filter(x=>x%2>0)
    .reduce((x,y)=>x+y,0)
```

Complete the constructor of this class and add methods for map, forEach, filter and reduce such that all the tests pass.

Exercise 5 - The Beginning of Pretty Text

Your pretty printer will follow a very simple data model. We'll store each line as a tuple [integer, string] representing the indentation of the line and the content respectively.
Thus we aim to make a program that converts text to a model and finally to a pretty string.

I.e. Ugly string -> data model -> Beautiful string

```
let uglyString = "{ 'name': 'Mary' }"
let data = [
    [0, '{'],
    [2, 'name: Mary'],
    [0, '}'],
]

/** Output as:
{
  name: Mary
}
*/
```

You'll need some functions that can be composed together to create your pretty printer. Please implement the following functions:

a) Implement function `line` that takes a string and returns this string wrapped as a tuple representing indentation. Examples:

```
line("nice!")
// returns [0, "nice!"]

line("");
// returns [0, ""]
```

b) You can't combine your lines unless they're wrapped in a List<T>. Therefore implement function `lineToList` which returns a List<[number, string]> with only 1 element and the type signature:

```
function lineToList(line: [number, string]): List<[number, string]>
```

c) Implement the method `concat` on the List<T> class. This method allows another List<T> to be joined onto the end of the list. It must pass these tests:

```
new List([]).concat(new List([1]);
// equal to `new List([1])`

new List([1]).concat(new List([]));
// equal to `new List([1])`

new List([0]).concat(new List([1]));
// equal to new `List([0, 1])`

let a = new List([0]);
let b = new List([1, 2]);

let c = a.concat(b);
// c will now be equal to new List([0,1,2]);
```

All that is missing is the ability to indent your line model: [number, string].

Exercise 6 - Pretty Binary Trees

You want to use the pretty printer you've been constructing to draw a binary tree. You've found some code that draws a binary tree, however there is a problem. You haven't implemented the function `nest` yet. After reading the code below, please implement the function `nest` making sure your output matches the comments at the end of the code sample, and passes the tests.

```
class BinaryTree<T> {
   constructor(
       public data: T,
       public leftChild?: BinaryTree<T>,
       public rightChild?: BinaryTree<T>,
   ){}
}

function prettyPrintBinaryTree<T>(node: BinaryTree<T>): List<[number, string]> {
   return !node
       ? new List([])
       : lineToList(line(node.data.toString()))
           .concat(nest(1, prettyPrintBinaryTree(node.leftChild)
                           .concat(prettyPrintBinaryTree(node.rightChild))))
}
```

```
const myTree = new BinaryTree(
   1,
   new BinaryTree(
       2,
       new BinaryTree(3)
   ),
   new BinaryTree(4)
);

const output = prettyPrintBinaryTree(myTree)
                   .map(aLine => new Array(aLine[0] + 1).join('-') + aLine[1])
                   .reduce((a,b) => a + '\n' + b,
                       '').trim();
console.log(output);

// Prints:
//1
//-2
//--3
//-4
```

Please implement nest:

```
function nest (indent: number, layout: List<[number, string]>): List<[number, string]>
```

Exercise 7 - Pretty N-ary Trees

Often syntax trees aren't binary trees, but N-ary trees. Currently the `prettyPrintTree` function only supports a left and right child. Write a new function called `prettyPrintNaryTree` that takes in the following tree type:

```
class NaryTree<T> {
   constructor(
       public data: T,
       public children: NaryTree<T>[] = [],
   ){}
}

// Example tree for you to print:
let naryTree = new NaryTree(1,
   [
```

```
        new NaryTree(2),
        new NaryTree(3,
        [
            new NaryTree(4),
        ]),
        new NaryTree(5)
    ]
)

// Sample output:
//1
//-2
//-3
//--4
//-5
```

Again make sure that the indentation only indents by 1. This is what the test cases assume. Also make sure that `prettyPrintNaryTree` returns a List<[number, string]> type.

Hint:

- `filter`?
- `reduce`?
- `map`?

Optional Challenge Exercise 8: JSON pretty printer or N-ary tree with children styled based on their type

Lots of communication between servers and webpages happens using JSON. Often this JSON isn't formatted and looks like a wall of text! Often we want to check the contents of the JSON wall of text and need a pretty printer. Quite a few websites exist to help beautify JSON text dumps:

- JSON Formatter and Validator
- JSON Pretty Print
- JSON Viewer
- *The one you're about to implement!*

JSON (JavaScript Object Notation) is just a JavaScript object. A JavaScript object is just like a hashMap or dictionary in other languages.
In previous exercises we've converted trees into a format that we can output! JSON is also just a tree, and there is an inbuilt function `JSON.parse` that helps us convert a JSON string into a JavaScript object.

JSON.parse takes a string as an argument, and returns the JavaScript object.

```
JSON.parse(text: string) => object
```

The structure of the JSON object can be found at this site: http://www.json.org/
JSON can be an array, object, number, string, boolean or null. Arrays can contain the listed types, and objects will always contain a key, value pair where the key is a string. The value can once again be any of the 6 types listed.
Therefore we can write a function `jsonPrettyPrint` with the following type signature:

```
jsonPrettyPrint: (json: object | string | boolean | number | null) => List<[number, string]>
```

Keep in mind that JSON is very similar to the N-ary tree implemented above with the addition of type checking the node of the tree. Here's some skeleton code to get you started:

```
const jsonPrettyToDoc: (json: object | string | boolean | number | null) =>
List<[number, string]> =
  json => {
    if (Array.isArray(json)) {
      // Handle the Array case.
    } else if (typeof json === 'object') {
      // Handle object case.
    } else if (typeof json === 'string') {
      // handle string
    } else if (typeof json === 'number') {
      // Handle number
    } else if (typeof json === 'boolean') {
      // Handle the boolean case
    } else {
      // Handle the null case
    }
}
```

There is currently no way to merge two lines so your JSON output can have key value pairs on separate lines. Feel free to implement this function/method if you wish.