WorkSheets

Writing, Running and Debugging JavaScript Code

JavaScript is the language of web development. It runs “client-side” on every modern browser and is increasingly used in server-side development using node.js.
For the first couple of tutes we will be running, testing and debugging our code in the Chrome browser (results may vary if you use a different browser).
It is also a multi-paradigm language. It has similar syntax to imperative languages like C and Java, but also supports anonymous functions that can be assigned to variables as values. Thus, we can begin to explore some functional-style programming.

We assume a Chrome browser and Windows desktop as this is the lab setup. If you have a different type of machine everything should still work but details like how to open files and keyboard shortcuts may vary - I assume you already know how to do this on your own machine.

Each week you will be given a code bundle on Moodle. Once you unzip the bundle onto your local drive there will be a file called **worksheetChecklist.html** that you will open in your browser (the easiest way is to double click the file icon in the windows explorer if Chrome is your default browser, or drag-drop it into a running Chrome session otherwise).
The first time you open it you will see lots of red error messages from automated tests. These will go away as you implement solutions to each of the exercises.

The worksheetChecklist.html file loads a file called **main.js.** This is the file you will edit to complete the exercises. I use and recommend the vscode editor. You may use a different one if you have a strong editor preference.
To start vscode on a lab machine, hit windows-key and type “code” and press enter.
Inside vscode, File -> Open Folder… and navigate to where you unzipped the code bundle.
Then click main.js in the Explorer to open it. Start writing code and ctrl-s to save.
After making changes and saving in the editor, reload the Chrome tab where you have worksheetChecklist.html open, and the tests will rerun, hopefully with more green and less red.

Week 1 - From Imperative to Functional Programming in JavaScript

To begin download the code bundle from the Week 1 section of the FIT2102 Moodle. Instructions above explain how to edit and run code.

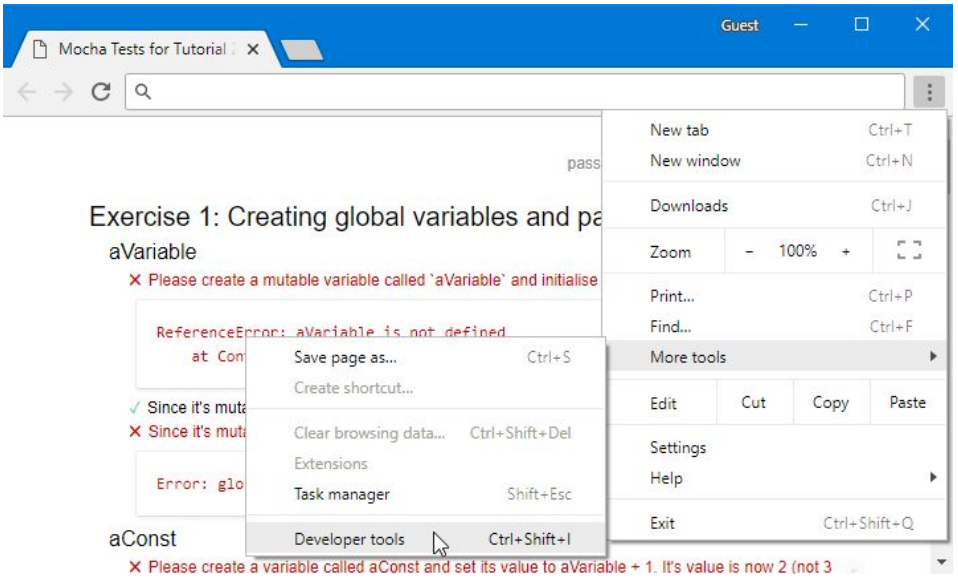
Exercise 1 - Creating global variables and passing the first test

The **let** and **const** keywords are for creating mutable and immutable variables respectively (we discuss the exact syntax for this in the lecture).

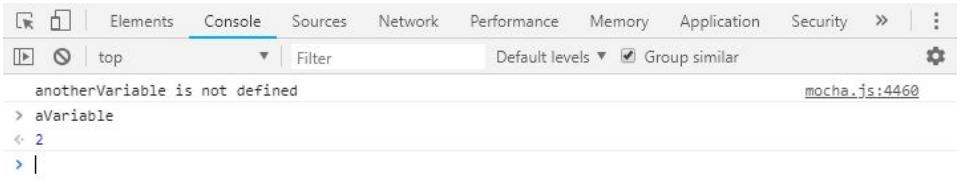
aVariable: Create a mutable variable called ‘aVariable’ and assign its value to 1.

aConst: Create an immutable variable called ‘aConst’ and assign its value to aVariable + 1.

Open the Chrome debugger: (... -> More tools -> Developer Tools).



In the pane that opens select the “Console” tab type
> aVariable <enter>



The debugger will tell you the value.

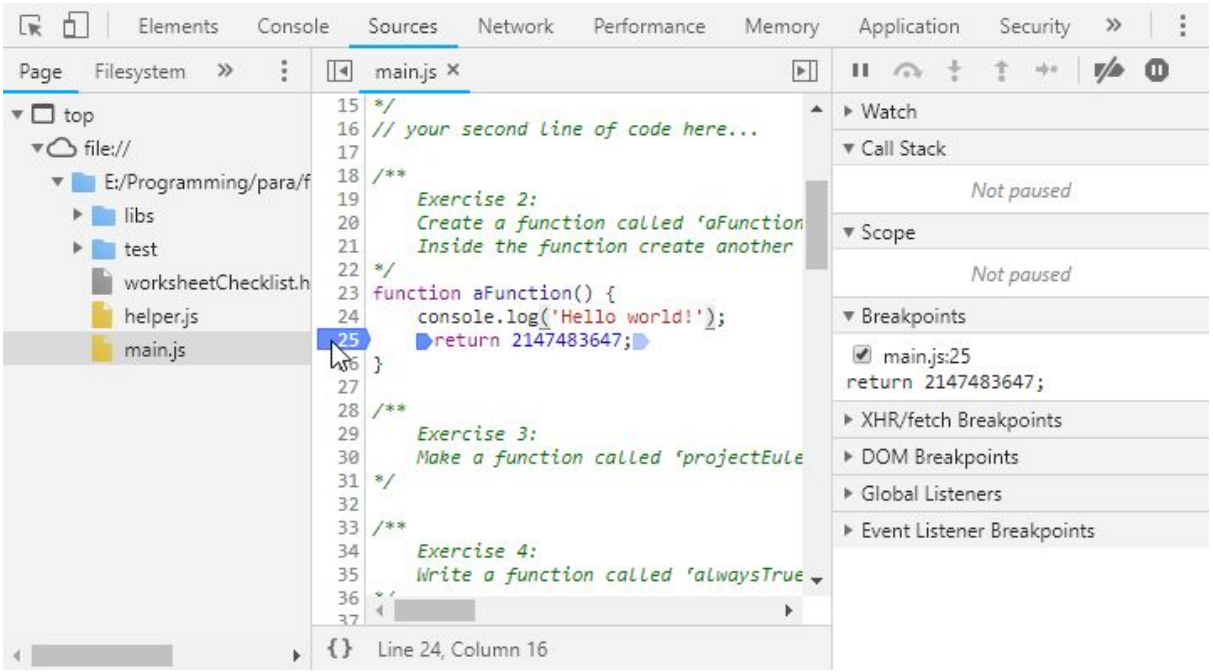
Note that since you created these outside of any code block or function, they exist at the “global” scope. This means once main.js is loaded by the browser (as its sourced by worksheetChecklist.html) these variables will be visible to any subsequently loaded code (and from the Console). This is a rather extreme “side effect”, that is generally to be avoided in good programming practice!

Exercise 2 - Create a function

aFunction: Create a function called ‘aFunction’ using the syntax discussed in the lecture. Inside the function create another variable called 'anotherVariable', set its value to 2 and return anotherVariable.

anotherVariable: In the debugger console, try to inspect the value of anotherVariable as you did for aVariable. You will receive an error message that the variable is undefined. Why?

Switch to the “Sources” tab in the Chrome Developer Tools and navigate to main.js. Click the line number of the return statement inside the function definition to place a breakpoint.



Press F5 to reload the page. The debugger will stop at the breakpoint. What do you see and why?

Play with the following buttons in the debugger to experiment with stepping through the code.

Exercise 3 - Create a function to solve Project Euler Problem 1

FYI: Working through the problems at Project Euler is a fantastic way to learn a new language. The problems start off easy and gradually get more difficult. If you create an account on the site, when you enter the correct solution to a problem it will give you access to forum where people discuss the solutions to that problem in lots of different programming languages.

Project Euler Problem 1 reads:

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Make a function called 'projectEulerProblem1' that calculates the answer using mutable variables, a while loop with an if statement, and returns the answer. You can also use the Modulo operator (%) which returns the remainder after an integer division, e.g.: 7 % 5 returns 2, and the boolean OR operator (||), e.g.: (0 == 1) || (2 == 2) returns true.

Exercise 4 - First Higher-order Functions

alwaysTrue
Write a function called ‘alwaysTrue’ which always returns true, no matter what argument it is given.

imperativeSummer
Write a function called imperativeSummer that takes two parameters: a number n, and a function f. It should use an imperative loop to sum over the numbers from 1 up to (but not including) n, including the number x in the sum only if f(x) is true.
Hint: you may like to use the javascript ‘ternary conditional’ operator to test with f in your sum expression:
E.g. sum += f(n) ? n : 0

sumTo
Use imperativeSummer and alwaysTrue to create a one-line function ‘sumTo’ that takes an number parameter n, and simply sums all numbers upto but not including n.

isDivisibleByThreeOrFive
Write a function called ‘isDivisibleByThreeOrFive’ which takes a number as parameter, tests if it is divisible by 3 or 5, returning true if it is.

projectEulerProblem1UsingImperativeSummer
Write a function called projectEulerProblem1UsingImperativeSummer that uses your imperativeSummer and isDivisibleByThreeOrFive to again solve Project Euler Problem 1. It should be one line of code!

Exercise 5 - Immutable loop

Write a function called 'immutableSummer' which computes the sum of numbers from 1 up to (but not including) n that satisfy f, but does *not* use while, for, or any mutable variables (defined with let or var).
Hint: use recursion!

Week 2 - Simulating a digital circuit

- Build familiarity with JavaScript for the assignment
- Work with JavaScript object's (prototype)
- Work with building blocks (wires) (logic gates) to build abstractions
- Have enough confidence in JS before diving into TypeScript (a superset of JS)

This week's tutorial will walk you through building a digital circuit simulator. This program will build your confidence in the JavaScript environment and hopefully clarify a lot of the syntax that will be necessary to read TypeScript fluently. Because an intimate knowledge of JavaScript is not the focus of this worksheet, a few helper functions have been provided in the **helper.js** file.

This worksheet is accompanied by a file called **worksheetChecklist.html**. With this file open in the browser, your code will be automatically checked. This contains tests for all the exercises contained in the worksheet.

You'll start by implementing code shown in the lecture and getting your first green ticks in the worksheetChecklist. Then you'll start building up the digital simulator.

We'll start by creating a "scheduler" that can keep a list of functions and the time in the simulation when they'll be run. We'll code it simply as a naive Priority Queue. A priority queue is a queue which returns the element with the highest priority. In our simulation, the highest priority will be the element with lowest simulation time. Once this is implemented you'll have invented simulated time!

Next you'll create wires. They'll be very simple digital wires that contain either a 1 or 0. Using functions they'll be able to pass on their signal or modify their signal. Modifying these functions is how you'll create logic gates and have invented a simple language of circuits in JavaScript.

The techniques taught in this worksheet can be built on in other contexts:

- Chat rooms (passing around chat messages)
- Turn based multiplayer games (passing around turns)
- Constraint propagation (passing around constraints)
- Passing around something!

Exercise 1 - Passing your first tests!

After you've downloaded the Tutorials code, open the file **worksheetChecklist.html** in the *Chrome* browser. This file tests your code and shows if the code works as expected. It only tests some cases.
There's a lot of red, so let's pass that first test!

Open **main.js**. This is where your code goes for each exercise. Notice you've been provided with `const myObj = undefined;`.

Please replace `undefined` with an object. This *myObj* object needs to be similar to the *myObj* shown in the notes.
It needs a key *aProperty* with any string as a value and a key *anotherProperty* with any number as a value. (Opposite to what's in the notes)

Once you've written an answer into the **main.js** file, save your file and *refresh* the browser. If the browser doesn't change, try hard refreshing (which makes sure your file is reloaded into the browser and not from cache).

Exercise 2 - C => Closures

Functions can be written in many different ways. In the lecture you've seen that the following is possible:

```
function add(x) {
    return y => y+x;
}
let addNine = add(9)
addNine(10)
> 19
addNine(1)
> 10
```

Here is another identical way of writing the **add** function from above.

```
const add = x => y => x + y
```

Here we're passing numbers into **add**, but it's also possible to pass functions as arguments. Please create a function **operationOnTwoNumbers** that takes a function as its argument and returns a function that must be called twice (like the **add** function above) before returning an answer.

A concrete example will make it easier to understand. We want to be able to do the following:

```
const add = operationOnTwoNumbers((x,y) => x + y)
const addNine = add(9)
addNine(3)
> 12
const multiply = operationOnTwoNumbers((x,y) => x * y)
const double = multiply(2)
double(4)
> 8
```

It works when all the green ticks appear!
Note: If you want to write a block of code in an arrow function you can use curly braces, i.e.

```
let sayName = name => {
    let message = "Hello ";
    console.log(message + name);
}
```

Exercise 3 - A Brief Look at Arrays

In the lecture you've seen the following:

```
['tim', 'sally', 'anne'].forEach(person=> console.log('hello ' + person))
```

You'll need a function that calls all the functions stored in an array. This should be quite a small function. Please write a function **callEach(array)** which takes in an array of functions, and iterates along the array calling each function. Don't use the JavaScript loop construct, and instead use the **.forEach** method on the array.

Example:

```
const fn1 = _ => console.log("Hey!");
const fn2 = _ => console.log("Cool!");
callEach([fn1, fn2]);
> "Hey!"
> "Cool!"
```

Exercise 4 - Inventing Time

It's time to start building the simulation. We'll start with the most important part, the queue/scheduler/action dispatcher or **UniversalClock**. This epic sounding clock will keep time in our simulation allowing for delays to happen in function calls. It'll work like this:

```
let clock = new UniversalClock();
clock.addToSchedule(2, () => console.log("World!") );
clock.addToSchedule(1, () => console.log("Hello") );
runSimulation(clock);
> "Hello"
> "World!"
```

The UniversalClock constructor function has been given to you.

- *this.schedule* is the array where the [time, function] pairs will be stored.
- *this.simulationTime* will store the current time of the simulation.

this.schedule when containing entries should look something like this:

```
[ [2, fn1], [2, fn3], [10, fn2], [12, fn4] ]
```

It's important that the first field is the simulationTime when the function will trigger and not just the delay. Therefore, when implementing **addToScheduler** make sure to add the current simulationTime to the delay before working out where in the array to insert the [time, function].

Please implement the following:

a) Complete the getter method **isEmpty** in **UniversalClock** so that it returns true if the array of actions is empty and false otherwise. Note that because this method is declared with the get keyword, it can be accessed like a property
b) Add the method **addToSchedule** to the prototype **UniversalClock** that takes two arguments. A delay (number) and a payload (function). **addToSchedule** adds the function to the correct place in the array using the time when the function will be executed (time when scheduled + delay). The UniversalClock's array of actions should always be in ascending time order. It's also important that functions scheduled to happen at the same time are ordered in the order they were scheduled.
    - Use the helper function **insertInArray(array, index, item)** which returns a new array with the inserted item.
    - Argument 1 is the **delay** before the action function will be called.
    - Argument 2 is the **action** function to be called.
    - Please use the tests to find errors and see expected behaviour.

Exercise 5 - Experiencing Time

Now that we can add actions to the UniversalClock we need a method **popFirstItem** that removes and returns the next function from the schedule, and updates the UniversalClock simulation time to the time on that item.
If the scheduler is empty, throw an error:

```
throw new Error("UniversalClock is empty -- UniversalClock.popFirstItem");
```

Because the array is in order, use JavaScript's array method *shift* to remove the first item.

Please implement **popFirstItem**.

With these methods implemented we have invented time! Well not quite. We need to be able to actually run time. Here's the **runSimulation** function. Feel free to implement it yourself if you want a challenge. It just needs to grab the first item from the UniversalClock, call it, and repeat until the list is empty.

```
function runSimulation(clock) {
   if (clock.isEmpty) {
       return;
   }
   clock.popFirstItem()();
   return runSimulation(clock);
}
```

Now we can test the UniversalClock concretely:

```
let clock = new UniversalClock();
clock.addToSchedule(2, _=> console.log("World!") );
clock.addToSchedule(1, _=> console.log("Hello") );
runSimulation(clock);
> "Hello"
> "World!"
```

Optional: Exercise 6 - Wires

Wires are just an object that contain a signal value and list of actions. Your Wire will only need two methods, **setSignal(signalValue)** and **addAction(actionFunction)**. **addAction** adds a function to an array of functions that will simulate signal passing. If we want a wire to send its signal to another wire, you can write the following function:

```
// This example doesn't use your scheduler yet.
function wirePassesSignal(inputWire, outputWire) {
    inputWire.addAction(() => outputWire.setSignal(inputWire.signalValue));
}
```

These two methods provide an incredibly powerful interface.

Whenever a wire's signal changes, all of its actions need to be called (Use **callEach** from Ex.3.). This propagates the change to the next wires. If the signalValue doesn't change, don't call the actions.

Whenever an action is added, it should be added to the *actions* array and called (to propagate the wires signal down this new connection). Use JavaScript's array method *push* to add an item to the end of an array.

Please satisfy the Wire's tests.

To help inspect your Wire, feel free to use the **probe** function contained in the helper.js file.

Optional: Exercise 7 - Completing the Circuit language

With the Wire's methods **setSignal** and **addAction** we can now simulate our logic gates. Logic gates are functions that add the appropriate actions to the wires. For example, a **not** logic gate will be a function **not(wireIn, wireOut)** that adds an action to the **wireIn** to change the signal of **wireOut** to be the opposite of whatever its signal is! And we can include a delay if we add the setSignal method on the scheduler. To include our UniversalClock in the logic gate functions, we'll need to use the closure concept from Exercise 2.

The code below creates 3 functions.

- **addAfterDelay** - a function that takes in an instance of the UniversalClock and returns a new function with the clock hidden inside. This is the function we'll be passing into our logic gate's factory[1] functions.
- **logicalNot -** switches a 1 into a 0 or a 0 into 1.
- **notFactory** - takes in a delay and an afterDelayFn which will be the function returned by addAfterDelay.

Passing in functions makes the code more testable and less reliant on the global scope.

```
function addAfterDelay(clock) {
   return (delay, action) => clock.addToSchedule(delay, action);
}
const logicalNot = signal => signal === 0 ? 1 : 0;

const notFactory = (delay, afterDelayFn) => (input, output) => {
   const notAction = () => {
       afterDelayFn(delay, () => output.setSignal(logicalNot(getSignal(input))))
   }
   input.addAction(notAction);
}
```

The notFactory generates a function that takes an input wire and output wire. Passing in these wires causes the input wire to add an action which adds a function on the scheduler. Finally, when the scheduler triggers the function, the output wire has its signal changed to be the negation of the input wire. The notFactory also allows us to specify the delay in the not logic gate.

Here's the code for an **andFactory** and please implement an **orFactory**.

```
const andFactory = (delay, afterDelay) => (in1, in2, output) => {
```

[1] The Factory name references that the function creates other functions. notFactory is a function that *constructs* and returns a **not** logic gate function.

```
    const andAction = () => {
        afterDelay(delay, () => {
            output.setSignal(getSignal(in1) & getSignal(in2));
        });
    }
    in1.addAction(andAction);
    in2.addAction(andAction);
}
```

Challenge exercise - Half Adder

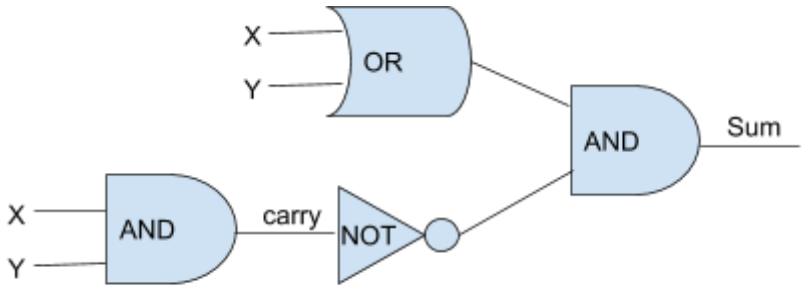
The final exercise requires you to implement a half adder. A half adder adds two single digit binary numbers together, and returns a sum and a carry. A half adder takes in two wires (**x** and **y**) and outputs on two wires (**sum** and **carry**). The truth table for a half adder is as follows:

x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Using only the logic gates written, create a half adder function. It'll need to be a factory function that takes in an **or**, **not**, and **and** logic function.

```
const halfAdderFactory = (or, and, not) => (x, y, s, carry) => {/*YOUR CODE*/}
```

The [circuit diagram for a half adder] is also helpful:



Once you've implemented the half-adder, you can use it to construct (details here):

- A full adder
- A ripple carry adder

Setting up your TypeScript project and running it

You've been introduced to TypeScript, and transpiling your TypeScript to JavaScript. This document will get you set up with a simple workspace that can be used to complete the workshops, do exercises, and work on the first assignment.

To transpile from TypeScript to JavaScript we'll need to install Node.js which comes with a command-line tool called `npm` (Node Package Manager). Using this tool we'll be able to install TypeScript and get set up quickly.

Step 1 - Installing Node.js and npm

installer and accept the defaults. By default Node will be added to your computer's path. This Install Node by visiting nodejs.org and download either version. Once downloaded run the will allow you to access node in the command line.

To check that node installed properly open your command line (Windows) or Terminal (Mac and Linux) and type `node --version`. You should get the node version you installed returned to you. Also check `npm --version` to make sure your Node Package Manager installed correctly.

```
$ node --version
V8.1.0    (your installed version will be newer than this)
```

Checking npm version on Windows:

```
C:\Users\userName>npm --version
3.10.10    (your installed version will be newer than this)
```

If the installation didn't work, try these more comprehensive instructions for your platform:

- Node and npm installation on Windows
- Node and npm installation on Mac
- Node and npm installation on Debian Linux

Step 2 - Installing VS Code and making your project folder

Now that you have npm installed we can create the project. Create a new folder to contain your project. This is where we will be placing our files, configuration and transpiling our code. Any packages we install using npm will also be stored locally in this project folder. This stops us cluttering up our computer with many JavaScript modules.

If you are using Visual Studio Code (VS code)[2], open your project folder in Visual Studio Code. If you want to use VS Code, install it from code.visualstudio.com. VS code has a nice feature that allows you to access the command line from within the editor. If you're not using

[2] Note: Visual Studio Code is not the monolithic Visual Studio IDE. It runs on all of Windows/Mac/Linux. In fact it is built in JavaScript/TypeScript!

VS code, simply follow along using the command line or terminal with your working directory set as your project folder.

Access the command line in VS Code by selecting View -> Integrated Terminal.

Step 3 - Initialise your TypeScript project

Your project will need a `tsconfig.json` file to be able to transpile your code. First install TypeScript globally using npm:

```
$ npm install -g typescript
```

The -g command tells npm to install typescript globally on your computer. Now you can verify if TypeScript installed correctly with `tsc -v`.

```
$ tsc -v
Version 2.3.4 // yours may be newer than this
```

To initiate the `tsconfig.json` file run `tsc` with the `--init` flag. This will create a default `tsconfig.json` file allowing you to use transpilation on your folder.

Note: It's important that the `tsconfig.json` is placed in the root of your project.

Step 4 - Transpiling your files

Currently you've got a project folder containing a single `tsconfig.json` file. Let's create our first TypeScript file. Create a file called `main.ts` next to the `tsconfig.json` file.
Add some code to `main.ts`.

```
// main.ts
console.log("Hello World!");
```

Now it's transpiling time! In your command line type `tsc`. This will automatically transpile everything in your project folder with the file extension `.ts`. It will follow the configuration in `tsconfig.json`.

```
$ tsc
```

A `main.js` file has appeared. Does the `main.js` contain what you expected?
Modify the `main.ts` and add a type declaration. For example:

```
// main.ts
console.log(<string>"Hello World!");
```

Before transpiling, do you think the contents of `main.js` will change? Transpile this file with `tsc` and check your answer.

Step 5 - Running your transpiled code

This is your current folder structure:

```
yourProjectFolder/
├── main.js
├── main.ts
├── tsconfig.json
```

We can run our transpiled main.js file in the command line using Node. Use the command `node <fileName>` to run your transpiled main file.

```
$ node main.js
Hello World!
```

Congratulations for setting up and running your project! You're now ready to write type safe JavaScript.

For more information:

- Editing TypeScript in VS Code - code.visualstudio.com/docs/languages/typescript

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 fluent-style programming chaining

Worksheet constraints

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

Marking

All seven exercises will be worth 2 marks, with a maximum mark of 10 for the entire workshop. We recommend you attempt as many exercises as possible!

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!", undefined))));
> 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. The class' constructor takes either an array or an existing ConsList<T>. This class will "wrap" around the existing ConsList<T>: 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)
    .forEach(x=>console.log(x))
    .reduce((x,y)=>x+y,0)
```

Complete the constructor of this class and add these methods such that all the tests pass:

- forEach - see the forEach note below
  SPOILER:
- filter
  SPOILER:
- map
  SPOILER:
- reduce
  SPOILER:

Your forEach method should return a copy of the current list. This allows for "tapping into" a fluent programming style chain, for when you want to `console.log` every item in the chain at a certain point like the example above.

You do not need to add the "T" type parameter to the methods above - the "T" type parameter is already given as part of the List class.

If you are stuck on the type signatures for a specific method, we have provided spoilers for the types if you are very stuck. Highlight the spoiler to reveal it!
You will have already written similar functions operating on ConsLists in exercise 2 and 3 - they may come in handy for this task too.

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 [number, 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.
<see next page>

```
type BinaryTree<T> = BinaryTreeNode<T> | undefined

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

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

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]>
```

Nest should take a layout and increase the indent of all lines by the given indent. For example:

```
let data: [number, string][] = [
   [0, '{'],
   [2, 'name: Mary'],
   [0, '}'],
];

console.log(nest(1, new List(data)).toArray());
// [
//     [1, '{'],
//     [3, 'name: Mary'],
//     [1, '}'],
// ]
```

You may encounter some type errors when writing nest, such as “Type '(string | number)[]' is not assignable to type '[number, string]'”. TypeScript may get confused about the type of an object in the form `[number, string]` - it can also be interpreted as an array where each element is `string | number`! As a result, TypeScript may infer the type of a variable or a type variable differently from what you’d expect.

If this occurs, you can “cast” the ambiguous object into another type, or override the type variable to a function / method call that TypeScript inferred for you.
To cast an object into another type, put the type before the object wrapped in <>:

```
// infers type to be (string | number)[]
const exampleLine1 = [0, "nice!"];
// explicitly cast type of right hand side to be [number, string]
// so TypeScript infers the type to be [number, string]
const exampleLine2 = <[number, string]>[0, "nice!"];
```

To override the type variable inference of a function / method call, put the type variables wrapped in <> before the parentheses of a function / method call - or cast the type of the arguments in a way which is unambiguous:

```
// infers type to be string[]
const stringArray = ["array", "of", "strings"];
// infers type to be (string | number)[]
const lineArray1 = stringArray.map(str => [0, str]);
// explicitly set the type variable U to be [number, string] in
// the type definition of Array.map:
// Array<T>.map<U>(callbackfn: (value: T) => U): U[],
// thus inferring the type of lineArray2 to be [number, string][]
const lineArray2 = stringArray.map<[number, string]>(str => [0, str]);
// explicitly cast the return value of the mapped function to be
```

```
// [number, string], thus inferring U to be [number, string] like lineArray2
// and inferring the type of lineArray3 to be [number, string][]
const lineArray3 = stringArray.map(str => <[number, string]>[0, str]);
```

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: List<NaryTree<T>> = new List(undefined),
  ){}
}

// Example tree for you to print:
let naryTree = new NaryTree(1,
  new List([
      new NaryTree(2),
      new NaryTree(3,
      new List([
          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 (named "any" in TypeScript").

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

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:

```
jsonPrettyToDoc: (json: any | 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: any | string | boolean | number | null) => List<[number,
string]> = json => {
   if (Array.isArray(json)) {
       // Handle the Array case.
   } else if (typeof json === 'object' && json !== null) {
       // Handle the object case.
       // Hint: use Object.keys(json) to get a list of
       // keys that the object has.
```

```
    } else if (typeof json === 'string') {
        // Handle string case.
    } else if (typeof json === 'number') {
        // Handle number
    } else if (typeof json === 'boolean') {
        // Handle the boolean case
    } else {
        // Handle the null case
    }

    // Default case to fall back on.
    return new List<[number, string]>([]);
};
```

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.
There is no automated test for this optional exercise. If you want to check that it is correct, copy-paste the output into the console and it should evaluate to an object identical to the one you started with (there's your test).
It should also be regularly indented and be wrapped with one property declaration per line.

Week 4 - Model infinite sequences

Outcomes:

- Practice coding lazily and solving problems using laziness.
- Build more familiarity with class syntax.
- Method chaining as a way to make code clearer.
- Revise referential transparency and function purity.
- Functional Reactive Programming with Observables and Observers.

This week we're going to look at laziness. The best thing about lazy iterators, is that you can define infinite sequences with them, and only use what's needed. These techniques are especially useful when dealing with huge data sets in the real world (or other infinite sequences like user interaction with your program).

First you'll start by implementing an infinite sequence initialisation function (7.4 in the notes). This function will become very useful as you use it to approximate Pi (π=3.14159...), using techniques covered in the second week (closures) and third week (map, filter, reduce).

You'll then implement Observables which you can listen to with an Observer! This opens up a paradigm of programming called Functional Reactive Programming. An Observable allows a way to react to asynchronous data streams (keyboard presses, web-sockets, mouse moves, server requests).

You'll then use your observable to add visualisations to the approximation of π via Monte Carlo[3] method!

Exercise 1

Please implement a general purpose infinite sequence initialisation function! *Question 7.4 in the notes*. Make sure it passes the tests. It must also match the following type signature:

Note: `LazySequence<T>` is defined in your code and in the notes.

Exercise 2

Still following the notes, please implement map, filter, take and reduce, making sure your functions match the given type signatures:

```
function map<T>(func: (v: T)=>T, seq: LazySequence<T>): LazySequence<T>
function filter<T>(func: (v: T)=>boolean, seq: LazySequence<T>): LazySequence<T>
```

[3] A simple statistical method used to help invent the Atomic Bomb in Los Alamos, 1945.

```
function take<T>(amount: number, seq: LazySequence<T>): LazySequence<T>
function reduce<T,V>(func: (v:V, t: T)=>V, seq: LazySequence<T>, start:V): V
```

Exercise 3 - Reduce everything!

Reduce is an extremely powerful tool for any algorithm that requires a single traversal of a structure. Therefore please implement the two unimplemented functions `maxNumber` and `lengthOfSequence`.

Use only reduce.

`maxNumber` should return the largest number in a lazy sequence.
`lengthOfSequence` should return the length of the lazy sequence.

If testing these yourself, make sure you wrap your lazy sequence in a `take` so that you don't get trapped in infinite recursion.

Exercise 4 - Lazy Pi approximations

Using the lazy list, let's approximate $\frac{\pi}{4}$. This can be done by defining an infinite series that looks like so:
$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$
The series alternates between a plus and minus sign while the denominators are an ascending sequence of odd numbers starting from 1.

Write your solution into the function:

```
function exercise3Solution (seriesLength: number): number {
     // Your solution using lazy lists.
     // Use `take` to only take the right amount of the infinite list.
};
// Expect return of approximation of pi/4 based on the length of the series passed in.
```

This function should return whatever the approximation of $\frac{\pi}{4}$ is for `*seriesLength*` elements of the series.

Some ideas for solving this:

Technique 1 - Use reduce to accumulate the series.

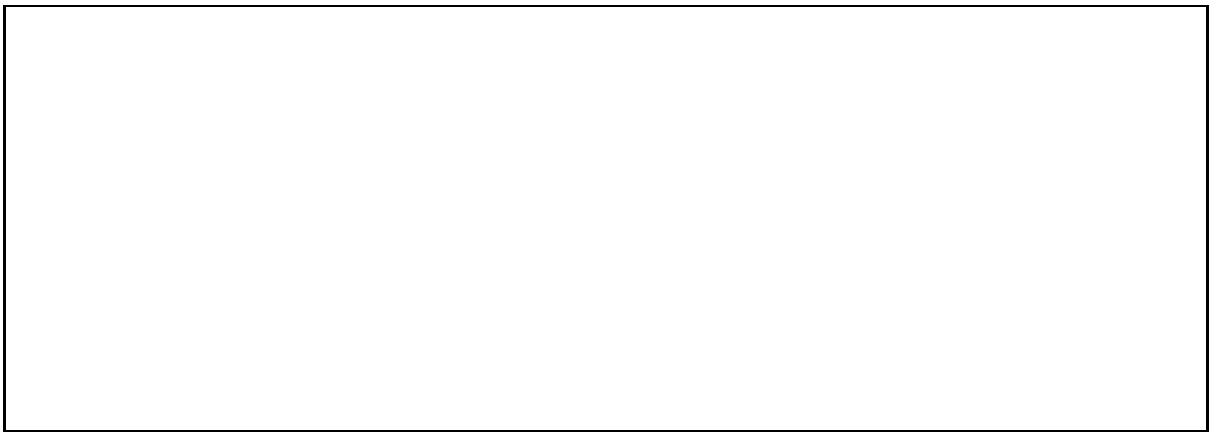
Generate a sequence of odd numbers with alternating signs. (+1, -3, +5, -7, ...)
Use `take`, `reduce` and `map` to generate the approximation.

Technique 2 - Accumulating iterator

Another way is to create a generator that generates the approximation of $\frac{\pi}{4}$:

- Hint 1: Use closures like in Week 2's exercise 1.
- Hint 2: Does the order of arguments matter?

BIGGER HINT: A fibonacci sequence that only returns a single number and stores state (It's spoilers.) Highlight over it to see it:



Technique 3 - Whatever functional method you choose using an iterator.

Whatever you choose, you'll need to explain it to your demonstrator.

Exercise 5 - Listening to an Observable with an Observer, a naive and simple implementation

Provided is a lovely block of code demonstrating the simplest Observer interacting with the simplest Observable. The Observable in this case is a function that subscribes the Observer to the "data source". The "data source" is a timer that calls the Observer's `next` method with a random integer. After 1000 milliseconds the Observable calls the `complete` method on the Observer closing the Observable.

There is also an Observer called `loggingObserver`. Please answer the following questions in comments in-line in the code in the marked places. Each answer should only be a sentence or two:

a) What happens when you execute this code?
b) What is the side effect that occurs when `loggingObserver.next(10)` is executed?
c) Is the `next` method in the `loggingObserver` a pure function?
d) Does the Observer stop executing `next` calls after the `complete` method is called?
e) Comment out the line `clearInterval(timer);`. What happens when you execute the code? Why?

Exercise 6 - SafeObserver

Above you saw an Observable that…. never actually completes. Please fill in the skeleton code for the SafeObserver so that it has similar behaviour to the simple observer, except it

completes when `complete` is called. Any further calls to `next` should be ignored. It should also complete when `unsubscribe` is called.

Please complete the SafeObserver class and pass the tests.

Exercise 7 - Your good friends map, filter and forEach

A skeleton Observable has been provided with a constructor, fromArray and subscribe methods. These three provided methods allow us to use this Observable to listen to a simple data stream.

```
Observable.fromArray([1,2,3,4,5,6])
      .subscribe(e => console.log(e));
```

Note that above we don't write `new Observable`. This is because we're calling a static method on the class and don't need to create a new instance of the class.

Please implement map, filter and forEach on your Observable.
Below are the type signatures:

```
map<R>(f: (_:T)=>R): Observable<R>
filter(f: (_:T)=>boolean): Observable<T>
forEach(f: (_:T)=>void): Observable<T>
```

Hint: Look at the `scan` method.

Exercise 8 - Observable.interval method

Instead of observing an array, it would be more interesting to create a static method that returns an Observable that emits the elapsed time in milliseconds since subscribing at increments specified by the user.

Implement a method `interval` on the Observable with the following type signature:

```
interval(milliseconds: number): Observable<number>
```

This will be quite similar to the very simple observable shown in exercise 5. Hint: the first emitted value should not be 0!

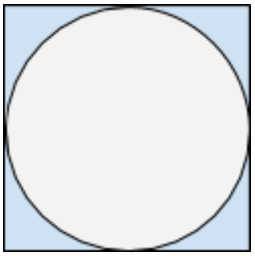
Optional Challenge - Improving a Visualisation of a Monte Carlo approximation of Pi

You're the newest hire on a data visualisation team that's working on visualising the process of approximating Pi using the Monte Carlo method. Congratulations!

Unfortunately some things haven't been finished and you've volunteered to improve the demo! But first, what the heck is a Monte Carlo method!?

The Monte Carlo method is used when a phenomena is easier to simulate than measure. Below we are going to randomly place dots on a unit circle and tally up which dots hit the circle and which miss. This will approximate π (because we know the ratio between the circle and square).

Mathematically the area for a circle is $\pi \times radius^2$. Thus the area for a unit circle, or circle with a radius of 1, is π. Let's draw a square around the circle.



The side of the square is $2 \times radius$ and therefore the area of the square is equal to $(2 \times radius) \times (2 \times radius) = 4 \times radius^2$.

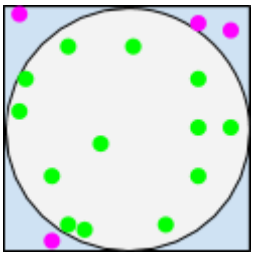
Therefore the ratio between the circle area and square area =

$$circle\ area : square\ area$$
$$\pi \times radius^2 : 4 \times radius^2$$
$$\pi : 4$$

Thus the probability of a random point falling within the circle is $\frac{\pi}{4}$.
Using this probability we can calculate pi!

$$\frac{points\ that\ fell\ within\ circle}{total\ data\ point} = \frac{\pi}{4}$$
$$\frac{points\ that\ fell\ within\ circle}{total\ data\ point} \times 4 = \pi$$

Below is a concrete example with 17 points.



13 green points inside circle.
4 purple points outside circle.
17 total points.

$$\frac{\textit{points that fell within circle}}{\textit{total data point}} \times 4 = \pi$$

$$\frac{13}{17} \times 4 = \pi$$

$$3.05882 = \pi$$

Notice when you open `challenge.html` in the chrome browser that a pi approximation is drawn to the screen, but lacking dots. Please complete the visualisation by drawing dots on the canvas with the dots coloured according to where they land on the canvas. Points within the circle should be a different colour to those outside the circle.

You even found a gif of what the visualisation should look like (*spoilers*):
http://www.giphy.com/gifs/26n6LHbwmraTyBmmY

Week 5

Part 1: Observable

Download the Assignment 1 code and look at the `Observable` implementation in observable.ts, reconsidering questions 5-8 from last week now that you have the answer. Discuss anything you still struggle to understand with your tutor. Open the basicexamples.html file in Chrome and play with the interactive widgets.

1. Watch the videos on Moodle this week discussing the basicexamples.ts code.
2. Switch the function calls at the end of the basicexamples.ts file to use the Observables versions of each function.
3. Complete the function `drawRectsObservable` using the given `Observable` class such that it has the same functionality as `drawRectsEvents`
4. Complete the function `drawAndDragRectsObservable.` See if you can refactor the code from `dragRectsObservable` and `drawRectsObservable` into separate draw and drag functions that are composable and reusable in this function. For example, the entire body of my `drawAndDragRectsObservable` looks like this:

   ```
   const svg = document.getElementById("drawAndDragRects");
   draw(svg).map(drag(svg)).subscribe(()=>console.log("done"));
   ```

   Yours doesn't have to look exactly the same - there are many different ways to make clean, reusable, understandable code.

There are no automated tests for this, your code is done when the interactions work correctly in the basicexamples.html page.

Complete the basicexamples.ts code and submit in a zip bundle with the answers to Part 2, ready to be checked by your tutor next week, as it will ensure that you are ready to create your pong implementation for Assignment 1.
Also, take the opportunity to discuss ideas for your assignment with your tutor.

Part 2: Lambda Calculus

This rest of this week's worksheet will be more theoretical and shorter to allow time for working on your assignment. Please submit answers to moodle in a document or submit photos of your work if you decide to hand write the solutions. If you submit a text file, you can write "λ" as "\" (this is the haskell way). Bundle all this up in a zip file along with your solutions to basicexamples.ts from Part 1.
These lambda calculus questions supplement the notes and it's recommended you work through the questions supplied in the course notes as well.

Outcomes:

- Build familiarity with lambda calculus and perform:
  - Alpha equivalence

    - Beta reduction
    - Eta conversion
- Identify divergent lambda expressions and combinators
- Y-Combinator

The following exercises are inspired by Chapter 1 of the Haskell Book. If you have not already picked up a copy, check out the unit announcements forum for the discount code.

Exercise 1: I-Combinator

What is the I-Combinator and why might you use it? Please translate its JavaScript code to a lambda calculus expression:

```
// I-Combinator in JavaScript
x => x
```

Exercise 2: Alpha equivalence

For the lambda expressions below, choose which of the options is alpha equivalent.

1. Which lambda expression is alpha equivalent to $\lambda x.x$
    a. $\lambda x.y$
    b. $\lambda a.a$
    c. $\lambda z.x$
2. Which lambda expression is alpha equivalent to $\lambda xy.yx$
    a. $\lambda az.az$
    b. $\lambda a.(\lambda b.ba)$
    c. $\lambda az.ba$
3. Which lambda expression is alpha equivalent to $\lambda xy.xz$:
    a. $\lambda xz.xz$
    b. $\lambda mn.mz$
    c. $\lambda z.(\lambda x.xz)$

Exercise 3: Beta normal form or divergence?

With the following lambda expressions, can you simplify them using beta reduction or do they diverge? Please show the normal form, or divergence by applying the lambda functions. Write out your steps.

Reminder: normal form is when an expression has been evaluated and is an answer. (You cannot further reduce the expression)

1. $(\lambda x.x)\ y$
2. $\lambda x.xx$
3. $(\lambda z\ .\ zz)(\lambda y\ .\ yy)$
4. $(\lambda x.xx)\ y$

Exercise 4: Beta reduction

Beta reduce the following expressions showing your working:

1. $(\lambda y.zy)a$
2. $(\lambda x.x)(\lambda x.x)$
3. $(\lambda x.xy)(\lambda x.xx)$
4. $(\lambda z.z)(\lambda a.aa)(\lambda z.zb)$

Exercise 5: Eta conversion

Use Eta reduction/conversion and the above methods to simplify the following expressions:

1. $\lambda x.zx$
2. $\lambda x.xz$
3. $(\lambda x.bx)(\lambda y.ay)$

Exercise 6: Which of the following are combinators?

1. $\lambda x.xxx$
2. $\lambda xy.zx$
3. $\lambda xyz.xy(zx)$
4. $\lambda xyz.xy(zxy)$

Optional Exercise: Y-Combinator application

Apply the function `g` to the Y-Combinator.
The definition of the Y-Combinator is: $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ where Y is the Y combinator.

Use the rules of reduction and equivalence to apply the function g to the Y combinator, and thus reduce the equation: Yg

Challenge exercise: Y-Combinator in JavaScript

The magic of the Y-combinator is that it allows anonymous functions to recursively call themselves! Complete the code below, implementing the function `Y` so that the function `fac` is able to recursively call itself anonymously.

```
// The Y-Combinator, gives anonymous functions recursive powers.
const Y = f => {
    // Your code here...
}

// A simple function that recursively calculates 'n!'.
const fac = Y(f => n => n>1 ? n * f(n-1) : 1);
```

```
console.log(fac(3)); // Prints 6
console.log(fac(5)); // Prints 120
```

Week 6

Discovering Haskell

Preamble

Welcome to your week 6 tutorial, first in Haskell! The goal of this session is to learn about the basic syntax of the language. First of all you will need to install Haskell. We will use the "stack" tools to install and build haskell. There are installers for all major platforms.

Stack creates a contained environment based on the information provide in the file stack.yaml, this allows you to work on multiple projects with different, and possibly conflicting, requirements at the same time.

To test stack, from a command prompt in the home directory (cmd.exe on windows, terminal on mac and linux) run:

```
$ stack new testproj
$ cd testproj
$ stack setup
$ stack build
```

The first time you run stack setup it will take a minute or so (it is installing a local version of GHC under C:\SR or ~/.stack) – later invocations will use the cached version and will therefore be very quick.

The default stack installation (stack new) will create a number of files in a project directory. The file stack.yaml contains build requirements, while `testproj`.cabal contains the code dependencies – external libraries and the like. The file you will have to worry about is src/Lib.hs where the logic resides, app/Main.hs is used to build an executable to run, and the test will not be used as of now.

In addition, you can install Haskero extension in VS code. It is a full feature IDE for Haskell programing.

For the Week 6 lab class, we have provided you with a stack project ready to go. The starter code is in src/week6 . There are three files in this directory: Pair.hs , List.hs and BinTree.hs . Currently the functions in these files are left undefined. Your task is to complete the functions. So that the tests pass.

After unzipping the folder, cd into it from the command line and run stack setup . If all goes well, after this step, your environment will be set up and you can start hacking. To run code within the environment, use:

```
$ stack ghci
```

You should see something similar to:

```
$ stack ghci
...
[1 of 3] Compiling BinTree
[2 of 3] Compiling List
[3 of 3] Compiling Pair
Ok, modules loaded: Pair, List, BinTree.
[4 of 4] Compiling Main
Ok, modules loaded: Pair, List, BinTree, Main.
Loaded GHCi configuration from ...
* Main BinTree List Pair>
```

This is the Glasgow Haskell Compiler interactive interpreter, you will be able to load and run your code directly in there. This interactive console has tab completion, information on the code loaded, etc.

For example, you should be able to create a Pair from REPL:

```
* Main BinTree List Pair> Pair 2 1
Pair 2 1
```

To test functions defined with the same name in more than one module (e.g. there is a “size” function defined in both List and BinTree), you can either load only one module at a time, e.g.:

```
$ stack ghci src/week6/List.hs
*List> size (List 3 [1,2,3] 1 3)
3
```

Or, dissambiguate by specifying the module name for the function:

```
*Main BinTree List Pair> List.size (List 3 [1,2,3] 1 3)
3
```

If you still have problems, the tutors are there to help.
In every file of the tutorial, you will see functions with a type signature and an `undefined` in the body, you need to replace this with your code.

Comments starting with >>> are called doctests, to verify they work run:

```
$ stack test
```

If your code is complete, they should all pass:

```
Examples: 11  Tried: 11  Errors: 0  Failures: 0
Examples: 10  Tried: 10  Errors: 0  Failures: 0
```

```
Examples: 17  Tried: 17  Errors: 0  Failures: 0
```

Otherwise you see an error and you still have work to do.
You can also run the tests for just one file at a time:

```
$ stack exec doctest src/week6/List.hs
Examples: 10  Tried: 10  Errors: 0  Failures: 0
```

Exercise 1: Pairs

A Pair is an element comprising of two Int , that is two integer numbers.
Your goal is to use pattern matching to access the elements in a pair, or multiple pairs, in order to apply functions to them.

Exercise 2: Binary Tree

A BinTree is a recursive data structure, which means that each element in a binary tree is a binary tree. Each node in a binary tree is therefore either a Nil, or empty tree; or a Node . A Node has three elements:
1. an integer value;
2. a left sub-tree; and
3. a right sub-tree.

Your task is to write basic functions on a binary tree.

Exercise 3: List

A List is an extended container, implemented as a record, which holds information about its inner list:
• size : the size of the list;
• elems : the actual list;
• low : the lowest element in the list;
• high : the highest element in the list.

Your goal is to implement functions on this data structure while keeping the members up to date; e.g., if you add an element to the list, you have to update the size, and potentially the low or high members.

Week 7

Preamble

In this week's tutorial we will see better ways to handle undefined behaviours. Last week, we realised that in some cases you cannot reliably define the behaviour of a function. For example, to define the minimum of an empty binary tree: would you use the maximum integer? an error? 0? None of these really makes sense.

One of your tasks, now, will be to write the type of your functions, where last week they were given to you. We will start using polymorphism in our code instead of fixed types. Polymorphism is one of the strong suits of Haskell, allowing users to write a single function that will work on multiple types instead of having to define one per possible type. By convention, a polymorphic type is called ‘a’ (compared to <T> in TypeScript), but any (lower case) letter would work.

Finally, you will learn to use *typeclasses*. Typeclasses are a bit like interfaces in TypeScript, a typeclass in Haskell is a guarantee that a type has certain functions. Some commonly encountered examples of typeclasses are:

* `Show`: enable the display of elements of the type. It must implement a function called ‘show’ that converts an element of the type to a String.
* `Read`: allow an object to be created from its String representation - must implement the ‘read’ function.
* `Eq`: enable equality testing between objects of the type with (==) and (!=) operators.
* And many more that you can explore online, or see the end.

Reminder

Once you have downloaded, extracted the code, run the tests by:

<from the terminal open in the top level of the unzipped code directory>

```
$ stack build
```

You can also try the functions in your code interactively from Haskell's interactive environment GHCi:

```
$ stack ghci
```

It should load all the source files in the current project. If you modify your code and want to test it, you need to:

```
GHCi> :reload
```

Or just:

```
GHCi> :r
```

GHCi is a REPL (Read-Eval-Print-Loop), which means that whatever code you type in it is interpreted as if it were source code; loading a file is equivalent to copy/pasting it. You can also load a particular file with the :load/:l command:

```
*Main Lib> :l src/Maybes.hs
[1 of 1] Compiling Maybes           ( src/Maybes.hs, interpreted )
Ok, modules loaded: Maybes
*Maybes> :r
[1 of 1] Compiling Maybes           ( src/Maybes.hs, interpreted )
Ok, modules loaded: Maybes.
```

The interactive environment gives you a number of tools to explore the code, typing `:type/:t` will give you the type of a function; to have more information, you can use `:info/:i`.

```
*Maybes> :t isJust
isJust :: Maybe a -> Bool
*Maybes> :i isJust
isJust :: Maybe a -> Bool       -- Defined at src/Maybes.hs:16:1
```

Using GHCi makes debugging faster as each time you reload your file(s) it will inform you of any syntax or type error.

You can also type code directly within GHCi and use its results either by copy/pasting into your code, or using `it` to recall the last results.

```
Prelude> map (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude> map (*2) it
[4,6,8,10,12,14,16,18,20,22]
```

Once you have finished writing the code for a file, you can test it using:

```
$ stack exec doctest src/<file>.hs
```

If you want to test the whole project, you can use:

```
$ stack test
```

Exercise 1: Safe Lists

Haskell offers an alternative to the issue of undefined behaviours by providing an *optional value* type: `Maybe`. A `Maybe` is either *just* a value or an empty result.

```
data Maybe a = Nothing | Just a
```

Your task is to implement **safe** functions on lists and this time you are also tasked with writing the **type signature** of the functions.

1. A safe version of `head` and `tail` which return an empty result instead of an error if the list is empty.
2. A *better* sum which differentiates empty lists and nil sums. Hint: you cannot just sum any types together, use the `Num` typeclass.

Optional

Rewrite the `minTree` function from last week's binary tree using a `Maybe`.

Exercise 2: Maybes

This exercise is about type manipulation rather than actual computation, your first foray into functional exercises. You need to implement a number of helper functions around `Maybe` constructs.

Hint: A general piece of advice in FP: "follow the types."

Question

Can you write a function with the following signature?

```
mystery :: Maybe a -> a
```

Exercise 3: Rock Paper Scissors

The goal of this exercise is to create your own typeclasses. A typeclass is a *property* of a type. By default, types in Haskell do not do anything, you cannot even compare two instances of the same type together! However, you can derive a number of typeclasses by default. E.g., you can derive `Eq` on all sum types without any custom code.

To create an instance of `Ord` it is necessary to define a *complete definition* using either: `<=` or `>=` but it requires `Eq`; or use `compare`. The latter is the most compact way to define the ordering of a type, it has the following definition:

```
data Ordering =  EQ | LT | GT
compare :: Ord a => a -> a -> Ordering
```

You will implement a (simple) game of Rock-Paper-Scissors, first define the necessary type classes then two functions:

1. `whoWon` takes two hands and return a results.
2. `competition` takes two series of hands and a number, and return whether a player has won `n` or more times.

An important feature of this exercise is to try to write **elegant** code. Elegant code leverages functional constructs as much as possible, think:
`map`, `filter`, etc.

Optional

Generalise `competition` to return which player won.

```
competition' :: [RockPaperScissors] -> [RockPaperScissors] -> Result
```

Week 8

Functor and Applicative

Preamble

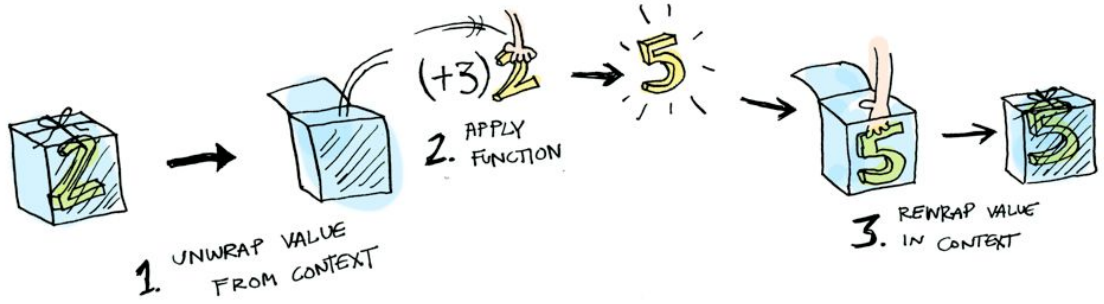
This week, we attack *real* functional programming: Functors and Applicatives, two of the main building blocks of functional programming theory.

Functor and Applicative are *typeclasses* like we saw last week. That is they are properties you apply on types. Types, by themselves, cannot enforce certain properties it is therefore the programmer's task to implement them.

When we implemented other typeclasses we implicitly used a concept called "minimal complete definition." That is, the minimum definition(s) – or function(s) – that are needed to express a property.[4] Functor and Applicative are no different. Read the about Functor and Applicative below and then complete the definitions in Functor.hs and Applicative.hs and then the exercises in Exercises.hs.

NOTE: The last exercise in each file is marked "tricky". The solutions are each only a couple of lines of code, but really require you to think about the definitions of Functor and Applicative. You should attempt them all, but if you are unable to solve them in a reasonable time, at least show your attempt and type some comments and questions about what you tried.

Functor



Functor application

Minimal complete definition

The minimal definition of a functor is `<$>` (fmap[5]).

---

[4] E.g., `Eq` needs only `==`, `Ord` can be expressed with `compare` or `Eq` and `<`, etc.

[5] Note that `fmap` stands for "functor map" because `map` was already reserved, it is not `flatMap`.

```
(<$>) :: (a -> b) -> f a -> f b
```

Laws

All instances of the `Functor` type-class must satisfy two laws. These laws are not checked by the compiler. These laws are given as:

1. The law of identity [6]

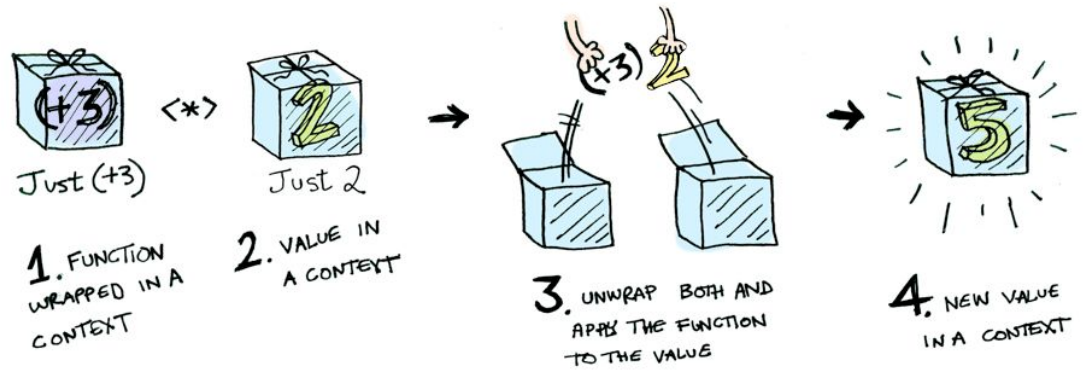
$$\forall x : (id \ <\$> \ x) \equiv x$$

2. The law of composition

$$\forall f, g, x : (f \circ g \ <\$> \ x) \equiv (f \ <\$> \ (g \ <\$> \ x))$$

So a functor takes a function, an element in a context, applies the function to the element, and returns the result in the context.[7]

Applicative



Using apply

Minimal complete definition

The minimal definition of an applicative functor is `pure` and `<*>` (apply):

```
pure  :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Laws

All instances of the `Applicative` type-class must satisfy four laws. These laws are not checked by the compiler. These laws are given as:

1. The law of left identity

---

[6] Where `id` is the identity function.

[7] If you replace `f` with `[]` you will notice that `fmap` is a general version of `map`; in a way, a type implementing functor is a type over which you can map.

$$\forall x : pure\ id\ <*>\ x \equiv x$$

2. The law of composition (where $(\circ)$ is composition i.e. (.))
$$\forall a, b, c : ((\circ)\ <\$>\ a\ <*>\ b\ <*>\ c) \equiv (a\ <*>\ (b\ <*>\ c))$$

3. The law of homomorphism
$$\forall f, x : pure\ f\ <*>\ pure\ x \equiv\ pure\ (f\ x)$$

4. The law of interchange
$$\forall u, y : u\ <*>\ pure\ y \equiv\ pure\ (\$\ y)\ <*>\ u$$

It's not necessary to memorise these - just know that they exist.
More details: [WikiBook on Applicative Functors]

So, an applicative takes a function within a context, an element within the same context, applies the function, and returns the results in the context.

Exercises

Now that the fundamentals are out of the way, you will implement some functions that leverage the functor and applicative typclasses:

1. `lift`, aka the *cheat function*. `lift` takes a binary function and applies it to two elements wrapped in a context.
2. `sequence` turns a list of structured items into a structured list of item, bailing on nils.
3. `replicate` takes an element in a context and replicates the effect a given number of times.
4. `filtering` is a compound filter function which takes a predicate that also produces an effect.

Optional

What does the following code produce?

```
filtering (const $ [True, False])
```

Credits

Images from adit.io in Functors, Applicatives, And Monads In Pictures.

Week 9

Foldable and Traversable

Preamble

In this week's tutorial, we will explore the power of foldable and traversable structures. Both are typeclasses such as Functor or Applicative. Actually, we only need three things for this week:

1. **Functor**. A functor is a container, we can apply a function over (or inside) of it.
2. **Foldable**. A foldable is a container that we can reduce to a single value.
3. **Traversable**. A traversable is both a foldable and a functor.

Prior to the Week 8 tutorial deadline (Monday morning), you will have to copy your `Functor.hs` and `Applicative.hs` file from last week into this week's `src/` folder to have access to Functor and Applicative instances.[8] After the Week 8 deadline, we will provide the `Functor.hs` and `Applicative.hs` source as part of the Week 9 code bundle.

Follow the types: Many of the exercises this week require a solid understanding of the types expected of each expression you need to write. A useful GHC feature that will help you with this is *type holes* which allow GHCi to tell you the expected type at a location.

For example, if you are defining an addThreeIntegers function:

```
addThreeIntegers :: Integer -> Integer -> Integer -> Integer
addThreeIntegers x y z = (+) (??? x y) z
```

and you're not sure what you should put at "???", put a hole "_" there:

```
addThreeIntegers x y z = (+) (_ x y) z
```

When you run GHCi, it will tell you the type of the hole:

```
error:
 Found hole: _ :: Integer -> Integer -> Integer
```

[8] To keep the project working as a single entity, please keep the header (`import` declarations) in the files while copying your code over.

Functions

*You have to write the body **and** the type signature for all exercises.*

First, we will explore the power of `fold`. Folding is a very basic operation in functional programming. A lot of functions are actually implemented as folds.

In our case, we will focus on a right-fold. A right-fold takes a binary function which merges elements, a base case, and a list of elements to return the *fold* of these elements through the function.

```
foldr ::          -- right-fold
  (a -> b -> b) -- binary function
  -> b            -- base case
  -> [a]          -- list of elements
  -> b            -- result
```

Another way to look at a right-fold is: replace all the *cons* `(:)` in a list with a function and the *nil* `[]` with the base case.

```
-- Sum the elements of a list with fold
[1, 2, 3, 4, 5]        -- Rewrite using cons
1 : 2 : 3 : 4 : 5 : [] -- Fold (+) on the list
1 + 2 + 3 + 4 + 5 + 0  -- 0 is the base case
15                     -- Apply
```

Interlude: Monoid

A Monoid is a type with an associative binary operation that follows the law of identity. Monoids allow folding operations using the binary function `(<>)`.

1. Law of identity (left and right).

   `mempty <> x ≡ x` `x <> mempty ≡ x`

2. Law of associativity.

   `x <> (y <> z) ≡ (x <> y) <> z`

Monoids, conveniently, provide a default value for their operation.[9]

```
mempty :: (Monoid m) => m
```

Monoids also provide a generalised `fold`, also called `mconcat`.[10]

```
mconcat :: (Monoid m) => [m] -> m
```

[9] In mathematics we call that an "identity element."

[10] Note how `mconcat` does not require a base case as it is defined within the monoid.

Foldable

A foldable is a structure which can be reduced to a single value. Think of it as a structure on which we can use `foldr`. To define an instance of foldable, we need to define the following function:

```
foldMap :: (Monoid m) => (a -> m) -> t a -> m
```

That is, given a monoid and a function to fold an item into it, and a sequence of non-monoidal items, we can reduce the sequence to a single element.

Defining an instance of foldable allows us to derive a number of very useful functions *for free.* For example, a generalised `fold`, both left- and right-fold, a list converter, a length function, element existence, etc.[11]:

```
class Foldable t where
  fold   :: (Monoid m) => t m -> m
  length :: t a -> Int
  toList :: t a -> [a]
  elem   :: (Eq a) => a -> t a -> Bool
  -- ...
```

Traversable

Traversable are structures which can be traversed while performing an action. A traversable has to be a foldable and a functor. They are defined using `traverse`.

```
traverse :: (Traversable t, Applicative f) => (a-> f b) -> t a -> f (t b)
```

All instances of traversable have to respect the following laws (you don't have to memorise these - but it's useful to know that they exist):

1. The law of identity.[12]

$$traverse\ Id = Id$$

2. The law of naturality:[13]

$$t \circ traverse\ f = traverse\ (t \circ f)$$

where: $(f \circ g)\ x = f\ (g\ x)$

3. The law of composition.[14]

$$traverse(Compose \circ fmap\ g \circ f) = Compose \circ fmap\ (traverse\ g) \circ (traverse\ f)$$

---

[11] See the list on hackage.

[12] Where *Id* is the identity functor as per previous tutes.

[13] Where *t* is an Applicative transformation: `t :: (Applicative f, Applicative g) => f a -> g a`.

[14] Where *Compose* is the composition of functors - more info.

Week 10

Monads

Preamble

This week, we tackle Monads, a less restrictive approach to effectful computation in Haskell. Monads are infamous as they allow "side-effects in Haskell." Which is not true. Monads allow us to encapsulate effects within side-effect free computation by wrapping the results with a known effect. We retain referential transparency, laziness, and all perks of functional programming.

As we have seen over the past few weeks, once you are *in* a context, you cannot get *out of it.* This is what monadic computation brings to the table: a way to apply functions that start with a *pure* value and return a wrapped result. As such, they sit in-between functors and applicatives.

Like "Monoid", the term "Monad" comes from category theory. But, according to the father of Haskell,[15] his biggest mistake was not calling them: "warm fuzzy things," because the term monad scares people into thinking they are hard.

Monad

In this module you will implement a few examples of monad typeclasses. A monad instance has to respect one law:

1. The law of associativity:

```
(m >>= f) >>= g  ≡  m >>= (\x -> f x >>= g)
```

Monad's minimal definition is:[16]

```
(>>=) :: m a -> (a -> m b) -> m b
```

That is: given a wrapped element and a function that produces an effect, apply the function.

Task 1: work through the exercises in `src/Monad.hs`

Example

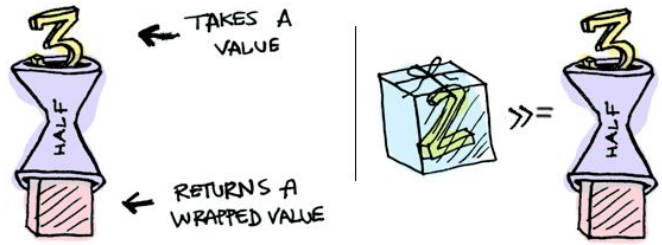
Consider the following function which takes an `Int` and returns its half if it is possible:

---

[15] Simon Peyton Jones

[16] Do note we will implement "bind" which is the flipped version of this function.

```haskell
half :: Int -> Maybe Int
half x | even x    = Just (x `div` 2)
       | otherwise = Nothing
```



Monad

Monad Example

Now what happens if you call the following:[17]

```
*Main> Just 20 >>= half >>= half >>= half
```

Do Notation

The law of associativity allows us to chain operation, to make this process easier on the eye Haskell has the `do` notation. It is only syntactic sugar to allow chaining monad operation.

```haskell
greet = do
  putStrLn "Enter your first then last name"
  first <- getline -- Get user input
  last  <- getline -- Another
  putStrLn ("Your name is " ++ first ++ " " ++ last)
```

Is exactly equivalent to:

```haskell
greet =
  putStrLn "Enter two numbers" >> -- Ignore the value of this line
  getline >>= \first ->              -- Bind user input
    getline >>= \last ->             -- Again
      putStrLn ("Your name is " ++ first ++ " " ++ last)
```

File I/O

Task 2: implement a program that takes a file with a list of files and prints the content of each of the listed file on the screen.

To do so you will have to implement a number of support functions in `src/FileIO.hs` and then write the user code in `app/Main.hs`.

Given a file containing a list of files separated by new lines:

---

[17] Answer at the end of the sheet.

```
$ cat share/test.txt
a.txt
b.txt
c.txt
```

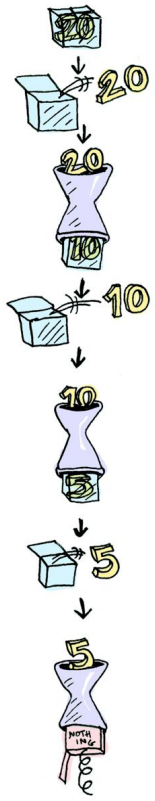
Your results should like the following:

```
*Main> :main "share/test.txt"
============ share/a.txt
Content of a.

============ share/b.txt
Content of b.

============ share/c.txt
Content of c.
```

Credit



Half 20

Images from adit.io.

Week 11

Parser Combinators

Preamble

For our last tutorial we will implement a basic Parser Combinator library and (optionally) use it to build a JSON parser. Parser combinators are a classic example of applying functional programming approaches to standard practices. The definition is:

"A parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as its output." – Wikipedia

We have been working for quite a while with functions and how to combine them, parser combinators are one more example of how you can combine operations.

The core idea is instead of having a black-box parser, you actually write small parsers for different entities and combine them together to create a parser for a specific format. This pattern is gathering quite a bit of attention as it is way easier to maintain than traditional parsers.

Parser

In this module you will implement the core logic for a parser.
A `Parser` is a new type defined as follows:

```
data Parser a = P { parse :: Input -> ParseResult a }
```

Quite a surprising type at first glance, but worry not. When creating a `Parser` we are actually creating a function that takes an input and returns the result of parsing the input.

Hence, a parser works as follows: consume a character of the input, apply its behaviour to it, return a new parser with the rest of the input or fail. Returning a new parser is what makes it a "parser-combinator" as we can chain the behaviours of multiple parsers together.

The file `Instances.hs` contains the definition of `Parser` and `ParseResult` and instances of `Show` for `ParseResult`, and `Functor`, `Applicative` and `Monad` for `Parser`. Because our `Parser` is an instance of `Monad`, we can use *bind* `(>>=)` to chain operations.

Parser Grammar

When using parsers, a natural way to think about their behaviour is to translate it to English.

English **Parser library**

and then	`>>=`
always	`pure`, `return`
or	`|||`
0 or many	`list`
1 or many	`list1`
is	`is`
exactly n	`thisMany n`
fail	`failed`
call it x	`\x ->`, `x <-`

Example

```
-- | Parse a phone number given as: 0-000-000-000.
parsePhone = do
  zero <- digit
  is '-'
  first <- thisMany 3 digit
  is '-'
  second <- thisMany 3 digit
  is '-'
  third <- thisMany 3 digit
  return $ zero : first ++ second ++ third
```

The lines with `<-` save the consumed input, the last line returns a new parser containing the saved result and any further input. The best part about this block of code is that it will fail gracefully if the string is badly formatted.

```
> parse parsePhone "0-491-570-156"
Result >< "0491570156"
> parse parsePhone "0-491-570-156aeu"
Result >aeu< "0491570156"
> parse parsePhone "0-491570-156aeu"
Unexpected character: "5"
> parse parsePhone "0-491-570-15aeu"
Unexpected character: "a"
```

Start building your parser by filling in the `undefineds` in `Parser.hs`.

Extras

If you want to implement extra features, that may also come in handy when writing `JSON.hs`, you can do the exercises in `Extra.hs`.

For a further challenge you can remove the instances for `Parser` by copying `Blank.hs` instead of `Instances.hs` and writing them yourself.

```
$ cp src/Blank.hs src/Instances.hs
```

JSON

The JSON parser is a specialisation (or combination) of the parser library you wrote in `Parser.hs`. You can try it out by writing the function `main` in `app/Main.hs` and using the examples in `share/`.

Note, completing the JSON parser takes a little while, so we are leaving it optional for the tute. But it is highly recommended if you plan to write a parser for your Assignment 2.