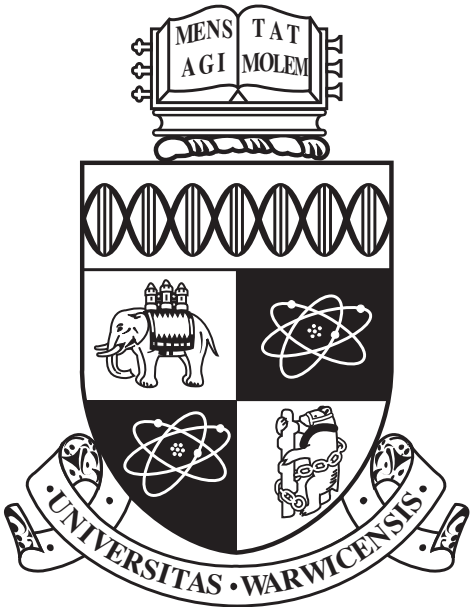
University of Warwick
Department of Computer Science

CS141

Functional Programming



Cem Yilmaz
June 6, 2022

Contents

1 What is Functional Programming

1.1 History

In 1928, David Hilbert posed the question "Given any true mathematical statement, is there an algorithm for verifying that it is true?". This is now known as the "Entscheidungsproblem", that is, the decision problem. E.g.

$$1 = 1$$
$$2 = 2$$
$$2 < 3$$
$$1 < 3$$

In 1931, Kurt Gödel came up with the paradox with the statement "This statement is not provable". If we assume not provable, there is a double negation which implies it is provable. This contradicts the assumption. If we assume true, we have a true statement that is not provable. This is called the incompleteness theorem. This answered the question that mathematics cannot answer every statement.

In 1936, Alonzo Church came up with $\lambda-$calculus as a system for describing algorithms. Kurt Gödel then believed that he can do better. He believed that some algorithms would not be able to be described with $\lambda-$calculus. Kurt then came up with a system of recursive functions. Alonzo Church then claimed that any algorithm that can be described using recursive functions can also be described using $\lambda-$calculus.

Alan Turing then came along and then came up with his own system of describing algorithms utilising Turing machines. However, he also showed that anything described using Turing machine could be described with $\lambda-$calculus. However, notice that despite these being different systems, they were also equivalent in describing algorithms.

1.2 Today

Today, programming as you know it, for example:

$$\prod_{i=1}^{4} = 1 \times 2 \times 3 \times 4$$

If we wanted to turn this to the roughly equivalent program in a language such as Java or C

```
1 int x = 1;
2 for (int = 1; i <= 4; i++) {
3   x *= i;
4 }
```

Listing 1: Product in Java

Table 1: Table of results

Variable	Value
x	1
x	2
x	6
x	24

However, in a function language, we can express it as

```
5 product[1..4]
```

Listing 2: Product in Haskell

That is, product is a function. You can also expand this to be

```
product[1,2,3,4]
1 * product[2,3,4]
1 * 2 * product[3,4]
1 * 2 * 3 * product[4]
1 * 2 * 3 * 4
24
```

Listing 3: Expanded Product

The definition of product function is rather simple:

```
product [n] = n -- Described in terms of 2 equations. First it checks if there a single item
    and returns it
product(n:ns) = n * product ns -- Takes the the first item in the list and keeps the rest
```

Listing 4: Product function

Let us now compare imperative programming with functional

Imperative	Functional
Mutation of state	Reduction of expression
Tell the computer how you want to do something	Tell the computer what you want to compute and let it work out how to do it
Statements executed in order specified	Sub-expressions can often be evaluated in an arbitrary order
Loops	Recursion

1.3 Programming Paradigms

In history, there used to be a clear distinction between programming paradigms. However, today, this has changed. That is,

- Java / C are now multi-paradigm: They're imperative, object-oriented, functional, etc.
- Python, JavaScript, C++ have similarly multi-paradigm.

As such, learning functional programming will be useful as paradigms have blended together.

1.4 What Haskell is good for

1.4.1 Web Services

Furthermore, a particularly nice application to functional programming is that they're good at web services.

- Lots of cool frameworks for developing web applications
- Easy to embed domain-specific languages for routing, templates, etc.
- Servant: describe web service as a type, automatically generate client programs
- One of the coursework assignments uses a web service written in Haskell to provide a browser-based interface

1.4.2 Domain-specific language

Domain-specific languages are also a thing. For example, you can write music in a Haskell library to describe music.

```
import Mezoo

v1 = d qn :|: g qn :|: fs qn :|: g en :|: a en :|: bf qn :|: a qn :|: g hn
v2 = d qn :|: ef qn :|: d qn :|: bf_ en :|: a_ en :|: b_ qn :|: a_ qn :|: g_ hn

main = playLive (v1 :-: v2)
```

Listing 5: Music in Haskell

What's cooler is that if we try to compile, we would get an error that the composition is not harmonic, that is, if it does not sound good, it does not compile. In particular,

- Major sevenths are not permitted in harmony: Bb and B_
- Direction motion in a perfect octave is forbidden: Bb and B_, then A and A_
- Parallel octaves are forbidden: A and A_, then G and G_

1.4.3 Games

For example, the game magic cookies utilises functional reactive programming to describe its game logic. It is the same code across different platforms. It is also good for time-travel debugging.

1.4.4 System Software

- XMonad: Window manager
- OS: Mirage (OCaml), House (Haskell) and more

2 Haskell Files

2.1 Defining functions

A haskell function looks like

```
20 triangle n = sum [1..n]
```

Listing 6: Function Definition

Function calls do not take brackets. We only use parentheses to set the order of operations.

2.1.1 Lambda Functions

Recall Alonzho Church's λ-calculus terms are

$$\lambda a.\lambda b.ab$$

In haskell, we can write functions using with lambda notation:

```
21 double = \x -> x*2
```

Listing 7: Lambda

equivalent to

$$\lambda x.x \times 2$$

Notice this can also be defined as

```
22 double x = x * 2
23 double = \x -> x * 2
```

Listing 8: syntactic sugar

and so

```
24 foo x y = x * y
25 foo = \x -> \y -> x * y
```

Listing 9: Syntactic sugaring

2.1.2 Arguments

Let us see how this will be evaluate by computer step by step

```
foo 5 6
-- (\x -> \y -> x * y) 5 6
-- (\y -> 5 * y) 6
-- 5 * 6
-- 30
```

Listing 10: Step by step evaluation

If we put in a single argument instead,

```
foo 5
-- (\x -> \y -> x * y ) 5
-- (\y -> 5 * y)
```

Listing 11: Partial function application

Notice that we get back a function. This is called partial function application, and it is used everywhere in functional programming.

2.2 Operators

We are used to the standard mathematical operators such as $+$, $*$. If we want our own custom operators, we could implement them. Operator definitions look very similar to function definitions:

```
x ^^^ y = max x y
```

Listing 12: defining operators

We can also partially apply them and make them functions:

```
plusFive = (+ 5)
plus = (+)
(+) 5 6
-- this would give 11 as it is equal to 5 + 6
```

Listing 13: Operator

The difference between operators and functions is that functions go before their arguments (prefix) and operators go between their arguments (infix). We are able to treat a function like an operator by enclosing it in backticks. We are able to treat an operator like a function by enclosing it in parentheses

```
5 `max` 6
-- gives an answer of 6
(*) 5 6
-- gives an answer of 30
```

Listing 14: Operator Function relation

Note that the number of argument stays the same, i.e., you cannot put 3 arguments into $(*)$ function.

2.3 Pattern Matching

We can change behaviour depending value of arguments. One way to do this is using if statements:

```
fac x = if x == 0
  then 1
  else x * f (x - 1)
```

Listing 15: Factorial

This is ugly, and we can find other ways of writing it.

```
fac x = case x of
  0 -> 1
  x -> x * f (x - 1)
```

Listing 16: Cases

In fact, there is another way of writing the same thing

```
fac 0 = 1
fac x = x * f (x - 1)
```

Listing 17: Top-level pattern

When evaluating code, the computer will pick the first definition whose arguments line up with patterns. Another way to write it is also

```
f x
  | x == 0 = 1
  | otherwise = x * f (x - 1)
```

Listing 18: Guards

This is the same as

$$f(x) = \begin{cases} 1 & \text{if} x = 0 \\ x \times f(x-1) & \text{otherwise} \end{cases}$$

2.4 Module

We separate Haskell code into files called modules. A module file always starts like

```
module Whatever where
```

Listing 19: Modules

The filename with $.hs$ and is the same as the module name. We can import one module into another using

```
module Foo where
double x = x * 2
-- In the file Foo.hs

module Bar where
import Foo
quadruple x = doubke (double x)
-- In the file Bar.hs
```

Listing 20: Import

The same syntax is also used to import libraries. A default library is always loaded, defined as prelude.

3 Compiling

Compilation is the process of turning raw source code into a format the computer can run. The Haskell compiler GHC has three main compilation stages:

$$\text{Raw source} \implies \text{Parser} \implies \text{Type checker} \implies \text{Code generator} \implies \text{Binary code}$$

3.1 Types

In Java, variables have types, e.g.,

```
63 int x = 5;
```

Listing 21: integer

we have assigned memory for int type to the variable x. In Haskell, every expression has a type. For example,

```
64 someBoolean = True && False || True
```

Listing 22: Expression

someBoolean has type Bool. The earlier example of min 5 6 gave us the type Integer. A string has type String. Concrete types, like String, Bool and Integer always start with an uppercase letter. We need to write type declarations when defining functions, that is

```
65 five :: Integer
66 five = min 5 6
```

Listing 23: Defining a function

The :: is a special piece of notation, that tells the compiler we're declaring a type. In Haskell, the compiler is able to find what types we are using and therefore we do not necessarily have to declare them. This is called type inference.

3.2 Polymorphism

There are 4 types of polymorphism

1. Parametric
2. Ad-hoc

Consider the lambda function

```
67 \x -> x
```

Listing 24: polymorphism

In maths, we call this particular function the identity function. But what is the type of this function?

```
68 id :: Integer -> Integer
69 id :: Bool -> Bool
70 id :: String -> String
```

Listing 25: Type of id

We could use any of these, but it is not general enough. I.e., we can only define it once. We instead could write

```
71 id x = x
```

Listing 26: id

If we want to give it a general type here, we need to use a type called *variable*. We write these with lowercase letters. E.g.

```
72 id :: a -> a
73 id x = x
```

Listing 27: variable type

The compiler will specialise the type for us when we use the function. This is the same as overloading a function. This works for arguments with more than one argument too.

3.3 Tuples

A tuple is a sequence of known, finite length. For example,

$$(1, 3) \in \mathbb{Z}^2$$

Haskell's tuples are very similar to mathematical tuples. We would denote them by, for example,

```
74 (5,"Hello") :: (Int, String)
```

Listing 28: Tuple

We can write functions that take tuples as an argument, for example

```
75 combine :: (Int, Int) -> Int
76 combine (x, y) = x + y
```

Listing 29: combine

3.4 Currying

We know that functions over multiple arguments have a type signature of this shape:

```
77 f :: Int -> Int -> Int
```

Listing 30: currying

When we apply one argument to f, we get back a function that takes the second argument and gives back the final return value. So f 's can be read like this:

```
78 f :: Int -> (Int -> Int)
```

Listing 31: f

The process of taking a function that takes several arguments at once and abstracting them one-at-a-time is called currying. All Haskell functions are curried by default, which enables partial function application everywhere. Consider the idea if we want to write functions over pairs in terms of functions over two arguments e.g.

```
79 orPair :: (Bool, Bool) -> Bool
80 orPair (x,y) = ...
```

Listing 32: orPair

But we only have access to

```
81 (||) :: Bool -> Bool -> Bool
```

Listing 33: or function

How can we convert from one form to the other? We have two functions:

```
82 curry :: ((a,b) -> c) -> ( a -> b -> c)
83 curry f = \a -> \b -> f (a,b)
84 uncurry :: (a -> b -> c) -> ((a,b) -> c)
85 uncurry f = \(a,b) -> f a b
```

Listing 34: curry and uncurry

Therefore the answer would be

```
86 orPair :: (Bool,Bool) -> Bool
87 orPair = uncurry (||)
```

Listing 35: or pair solved

3.5 Lists

Lists are critical tool for building programs with interesting behaviour. Note that lists are homogeneous, meaning that all types in the list must be the same.

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

Listing 36: lists

The second constructor is an operator which we pronounce cons. It takes two elements - the head and the tail - and joins the head to the tail. We can write any lists we like in this way

```
True : (False : (True : []))
==> [True, False, True]
```

Listing 37: : operator

```
startsWithFive :: [Int] -> Bool
startsWithFive []  = False
startsWithFive (x:xs) = x == 5
```

Listing 38: startsWithFive

We can nest pattern matches to simplify the code

```
startsWithFive :: [Int] -> Bool
startsWithFive (5:xs) = True
startsWithFive _ = False
```

Listing 39: Starts with five

We also can obtain the head and the tail of a list using the

```
head :: [a] -> a
tail :: [a] -> [a]
```

Listing 40: head and tail

Unfortunately, head and tail is not well defined i.e., non-total functions and do not work for the empty list. Some other useful functions include:

```
-- take the first n elements of a list
take :: Int -> [a] -> [a]
-- drop the first n emenets of a list
drop :: Int -> [a] -> [a]
-- get the nth element of a list
!! :: [a] -> Int -> a
-- join lists
++ :: [a] -> [a] -> [a]
```

Listing 41: Useful list functions

Lists are not the same thing as arrays and getting things like length would take more load and furthermore does not have things like re-defining a specific index element.

4 List Comprehensions

In maths, we know how to construct sets with interest features

$$A = \{x^2 | x \in \{1, 2, 3\}\}$$

Informally: "A is the set of elements x squared, for each x in the set $\{1, 2, 3\}$ ". This is a set comprehension. Haskell borrows comprehension syntax for its lists.

```
a = [ x^2 | x <- [1,2,3] ]
```

Listing 42: List comprehension

The $x < -[1, 2, 3]$ is a generator. It binds each of the values in turn so they can be used. x^2 is an expression that makes use of the bindings from the generators. The vertical pipe separates the expression from any generators. This allows us to use multiple generators.

```
[ (x,y) | x <- [0..3], y <- [0..x] ]
[ (0,0)
, (1,0), (1,1)
, (2,0), (2,1), (2,2)
, (3,0), (3,1), (3,2), (3,3)
[
```

Listing 43: Multiple generators

Sometimes we want to specify a property that must hold for us to include our element in a set

$$\text{Evens} = \{x | x \in \mathbb{N}, x \text{ is even}\}$$

In haskell:

```
evens = [ x | x <- [0..], even x ]
```

Listing 44: even set

We call this boolean predicate a guard. Like our guards on functions, it only lets us proceed if it evaluates to true

5 Map and filter

5.1 Filter

Suppose we already have the function

```
squares = [ x^2 | x <- [1..] ]
```

Listing 45: squares

How can we get even square numbers from this? We could wrap it with another list comprehension:

```
evenSquares :: [ s | s <- squares, even s ]
```

Listing 46: evenSquares

But this is ugly and could be written simpler, using function filter

```
filter :: (a -> Bool) -> [a] -> [a]
-- and for our example
evenSquares = filter (\s -> even s) squares
-- which is the same as
evenSquares = filter even squares
```

Listing 47: filter

5.2 Map

We know how to apply a function to a single argument. How can we apply a function to a whole list? This is called *mapping*.

```
123 map :: (a -> b) -> [a] -> [b]
```

Listing 48: Mapping

For example,

```
124 map succ [1,2,3]
125 ==> [4,5,6]
```

Listing 49: mapping example

5.3 Ranges

We can specify that a list should include a range of values. The syntax is as follows: $[x..z]$ x is the start of the range and z is the end of the range. Note that upper end of the range is optional. Ranges are arithmetic progressions.

6 Type Classes

Reminder: parametric polymorphism

```
126 id :: a -> a
127 id x = x
128 ($) :: (a -> b) -> a -> b
129 ($) = \a -> \b -> a b
```

Listing 50: parametric polymorphism

If we know nothing about the type of our arguments, we are unable to use any interesting features of the type! This function works on any type that meets the requirement X, Y and Z. The solution to this is Type classes. Consider

```
130 :t (==)
131  (==) :: (Eq a) => a -> a -> Bool
```

Listing 51: Eq

In other words, this can be read as that the $==$ works as long as a is of the type class Eq.

```
132 class Eq a where
133 (==) :: a -> a -> Bool
```

Listing 52: Eq

The first line defines the name of the type class eq, and a type variable a to use in the definitions. After that comes a series of function type definitions, without implementations. The type class declaration is a specification that must be implemented for each type that wants to be part of the Eq type class. This declaration implicitly creates:

- A set of types called Eq, initially empty;
- An operator called (==) which can be used on any type which is a member of Eq.

For example

```haskell
module LectureSix where

class MyEq a where
  (===) :: a -> a -> Bool

-- and if we tried to run this
True === False
-- we would get an error saying that Bool is not a part of type class MyEq, since we have not
    implemented it yet.
```

Listing 53: MyEq

There are also other type classes such as the Num type class used in regular operators such as $+, -, \times$ etc.

6.1 Implementation

We specify that a type is a member of a type class by writing an instance declaration.

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  True == True = True
  False == False = True
  _ == _   = False
```

Listing 54: Instance declaration

Type classes seem similar to interfaces from Java.

6.2 Using type classes

For the code

```haskell
doubleInt :: Int -> Int
doubleInt x = x + x
doubleFloat :: Float -> Float
doubleFloat x = x + x
```

Listing 55: double

It would be easier if we instead used Num type class:

```haskell
double :: (Num n) => n -> n
double x = x + x
```

Listing 56: Num in double

6.3 Different Type classes

6.3.1 Eq

The Eq type class lets us test two values for equality if they are of the same type. In other words, it gives us access to the function $==$ for polymorphic type a.

6.3.2 Num

Num is the numeric type class and it gives us the access to standard mathematical operations such as $+, -, \times, abs$. Integer literals are of type Num.

6.3.3 Ord

The ord type class that imposes a total ordering on elements of the type. I.e., it gives us access to $<, >, <=$ functions which gives us boolean result. Every member of the Ord type class is also a member of Eq.

6.3.4 Read and Show

Read and Show let us convert value sto and from strings

```
read :: (Read a) => String -> a
show :: (Show a) => a -> String
-- there is a law about these two type classes
read (show x) == x
-- E.g. this would read as Int 5
read "5" :: Int
```

Listing 57: Read and showw

6.3.5 Integral and Floating

Integral and floating represent integer-like and floating-types respectively.

```
div :: (Integral a) => a -> a -> a
(/) :: (Fractional a) => a -> a -> a
```

Listing 58: division

Integral rounds off a fractional type to an integer type. The fractional type does not round and outputs floating.

6.3.6 Enum

The Enum type class plays an important role when working with lists. Recall our original question:

- How does the list range syntax know how to generate 'a', 'b', .. etc.
- for Char and $1, 2, 3 \ldots$ for Int?

Haskell uses Enum type class to achieve this.

6.3.7 Foldable

The foldable type class abstracts those types which can be folded.

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

Listing 59: foldable

The foldable type class only works for type which have a single parameter. Previously, we defined foldr for lists only. However, it does not have to be only lists and we could extend its type classes by

```
sum :: (Foldable t, Num a) => t a -> a
```

Listing 60: Sum for more data

so we could use the foldable class for MyList as

```
foldr f z Nil = z
folr f z (Cons x xs) = f x (foldr f z xs)
```

Listing 61: Foldable for MyList

6.4 Referential Transparency

We can always substitute the LHS of an expression for the RHS and vice versa. The ability to perform this substitution is called referential transparency. The LHS and the RHS of a definition mean exactly the same thing, always.

7 Recursion

7.1 Recursion on lists

Consider

```
168 sum :: Num a => [a] -> a
169 sum [1..4]
170 -- the answer would be 10
```

Listing 62: sum

Recall that lists have two constructors:

```
171 [] :: [a]
172 (:) :: a -> [a] -> [a]
```

Listing 63: constructors of lists

We can pattern match these.

```
173 sum :: Num a => [a] -> a
174 sum []   = 0
175 sum (x:xs) = x + sum xs
```

Listing 64: pattern matching list constructors

Where x is the head and xs is the tail.

7.2 Explicit Recursion and Implicit Recursion

When writing recursive functions over lists, we have been pattern matching on the two constructor of lists: [] and (:). For example,

```
176 succAll :: Enum a => [a] -> [a]
177 sucAll [] = []
178 sucAll (x:xs) = succ x : succAll xs
```

Listing 65: Succ all

This would explicit recursion. Implicitly, we could use map function. More specifically,

```
179 succAll xs = map succ xs
```

Listing 66: succ implicit

Implementing some recursive behaviour via one of the helper functions is called implicit recursion. Note that maps are higher-order functions. That is, it takes a function as an argument as an argument or one that returns a function. So far,

```
180 map :: (a -> b) -> [a] -> [b]
181 filter :: (a -> Bool) -> [a] -> [a]
```

Listing 67: Higher-order functions

7.3 Folding

Recall our explicit definition of sum:

```
182 sum :: (Num a) => [a] -> a
183 sum []   = 0
184 sum (x:xs) = x + sum xs
```

Listing 68: sum

Can we implement this using our map or filter higher order functions? We need a new kind of recursive pattern here. We have:

- A function we use to combine in the recursive case
- A value for the base case
- The list we are folding over

```
185 foldr :: (a -> b -> b) -> b -> [a] -> b {- the first argument is a function, the second
        argument is base case, the third argument is the list we apply it to -}
186 foldr f z [] = z {- in the case it is empty, we return the base case -}
187 foldr f z (x:xs) = f x (foldr f z xs) {- we apply x and the parantheses to the function and
        continue to fold over the tail -}
```

Listing 69: folding

For example, for sum we could

```
188 sum :: (Num a) => [a] -> a
189 sum xs = foldr (+) 0 xs
```

Listing 70: Sum using fold

Notice that if we expand this definition over the list $[1, 2, 3]$ we would get a gathering of parentheses on the right, meaning it foldr is right associative. Note that the cons operator $(:)$ is right associative also. E.g., $[1, 2, 3] = (1 : (2 : (3 : [])))$ The left associative fold is called

```
190 foldl :: (b -> a -> b) -> b -> [a] -> b
191 foldl f z [] = z
192 foldl f z (x:xs) = foldl f (f z x) xs
```

Listing 71: Left associative fold

This implementation is different to foldr, but because addition is associative, it does the same job for addition.

```
193 sum :: Num a => [a] -> a
194 sum = foldl (+) 0
195 sum [1,2,3]
196 sum (1 : 2 : 3 : [])
197 foldl (+) 0 (1 : 2 : 3 : [])
198 foldl (+) (0 + 1) (2 : 3 : [])
199 foldl (+) ((0 + 1) + 2) (3 : [])
200 foldl (+) (((0 + 1) + 2) + 3) []
```

Listing 72: Sum using foldl

There is also a function called foldl' which does the same thing as foldl, except it calculates everything as it goes instead of leaving it to the end.

```
sum [1,2,3]
foldl' (+) 0 (1 : 2 : 3 : [])
foldl' (+) 1 (2 : 3 : [])
foldl' (+) 3 (3 : [])
foldl' (+) 6 []
6
```

Listing 73: foldl'

7.4 Functor

We have seen other recursive patterns than foldr. Map applies a function to every element of a list. In theory, we could map over elements of any data structure. The map function is specifically defined for lists. The functor type class is an abstraction of those types which can be mapped over. Like Foldable, this only works for parameterised types.

```
class Functor f where
fmap :: (a -> b) -> f a -> f b
```

Listing 74: functor

7.5 Laws

A type class law is a rule which must be met in order for a type class to be considered "legal". They are not enforced by the compiler - we must check them by hand. The laws for any type class are defined in the documentation that accompanies the type class.

7.5.1 Functor laws

1. The mapping must be structure preserving - the output shape must be the as the input
2. The identity law:

```
    fmap id === id
```

Listing 75: identity law

3. Distributivity of fmap over (.):

```
    fmap (f . g) === fmap f . fmap g
```

Listing 76: distributivty of fmap

7.5.2 Associativity

In mathematics we say an operation is associate if we get the same answer no matter where we would put the brackets we would obtain the same result. E.g., addition.

$$a + (b + c) \iff (a + b) + c$$

Therefore addition is associative. However, not everything is associative. For example. 2^{2^3} is not associative, i.e., whether it is 2^8 or 4^3. How does haskell know how to compute

```
2^2^3
```

Listing 77: exponentiation

For this we introduce the concept of left and right associativity. Every operator in a programming language is either left or right associative. That determines which way bracketing should occur. E.g,

$$a \sim b \sim c$$
$$(a \sim b) \sim c \text{ if the operator is left associative}$$
$$a \sim (b \sim c) \text{ if the operator is right associative}$$

Normally, functions are left associative.

7.5.3 Operator Precedence

Operators have a notion of precedence. In mathematics, this is called the order of operations. Haskell's precedence goes from 0 to 9. Each operator has its own precedence.

Table 2: Precedence

Operator	Meaning	Associativity	Precedence
(^)	Exponentiation	Right	8
(*)	Multiplication	Left	7
(+), (-)	Addition, Subtraction	Left	6

Note that function application has precedence 10. Sometimes we want to implement our own operators and we want to declare its precedence and whether it is left or right associative. This is called fixity declaration. For addition, this is

```
infixl 6 +
-- The infix suggest that it is a fixity declaration. The l suggests it is left associative.
    The 6 gives its precedence. The plus shows its name of the operator.
```

Listing 78: Fixity declaration for addition

8 Let and Where

Suppose we want to define the volume of a sphere defined as

$$\frac{4}{3}\pi r^3$$

Suppose we want to define

```
diameterToVol :: Floating a => a -> a
diameterToVol d = 4/3 * pi * (d/2) * (d/2) * (d/2)
```

Listing 79: diameterToVol

This will compute $\frac{d}{3}$ 3 times. Therefore, in order to make it more efficient, we can use the syntax to define intermediate values let in.

```
diameterToVol d =
  let
    r = d / 2
  in
    4/3 * pi * r * r * r
```

Listing 80: Let in

Expressions defined inside of a let can access the arguments of their parent function. Let expressions have a friend called where. Where works the same, except it comes after the main expression.

```haskell
diameterToVol d = 4/3 * pi * r * r * r
  where
    r = d / 2
```

Listing 81: where

The choice of let and where is personal preference, but let is a little more flexible. Where is syntactic sugar for let.

9 Data

In oop, data is represented by classes. In Haskell, the definition of our data representation is completely separate from the function that use it. This is a much more flexible way of working with data that is readable and reusable. We introduce new data types with the data keyword. Here is one data declaration

```haskell
data Bool = False | True
```

Listing 82: Bool

This definition brings to constructors into scope as values:

```haskell
False :: Bool
True :: Bool
```

Listing 83: scope as values

Note that to print out the Bool for the REPL, we would require to create an instance of Show for Bool so we can convert it to a string and print. I.e.

```haskell
instance Show Bool where
  show True = "True"
  show False = "False"
```

Listing 84: REPL for Bool

9.1 Constructors with parameters

So far we've been able to create custom values, but we haven't got a way of containing any data in our data types. To do this, we put parameters to our constructors. For example

```haskell
data Date =
  BC Int Int Int
  | AD Int Int Int
-- and if we were to write :t BC we would get BC :: Int -> Int -> Int -> Date
So we could write
epoch :: Date
epoch = AD 1 1 1970

instance Show Date where
show (AD dd mm yyyy)
  = show yyyy
  ++ "-"
  ++ show mm
  ++ "-"
  ++ show dd

show (BC dd mm yyyy)
  = show yyyy
  ++ "-"
  ++ show mm
```

```
  ++ "-"
  ++ show dd
  ++ "BC"
```

Listing 85: Constructors with parameters

9.2 Polymorphic Data types

Everything we've seen so far is very concerete: we can't use our types to represent wrappers around arbitrary values. We can introduce polymorphism by adding type variables to our data types::

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

Listing 86: Maybe

This brings two constructors into scope:

```
Nothing :: Maybe a
Just :: a -> Maybe a
```

Listing 87: scope

We can use Nothing and Just to construct values of the Maybe type:

```
noBoolSadTimes :: Maybe Bool
noBoolSadTimes = Nothing
yayFive :: Maybe Int
yayFive = Just 5
```

Listing 88: Example

And we can pattern match on them as well:

```
valueOrZero :: Maybe Int -> Int
valueOrZero (Just x)  = x
valueOrZero Nothing = 0
```

Listing 89: Pattern matched

In other words, if we get int we get just, otherwise get nothing. The Maybe type is very important. If we have something of type Maybe a, then we know that either there is a value (the Just constructor), or there is not (the Nothing constructor). The maybe is efficient to use for impartial functions: i.e., we can use it to output errors also which are not necessarily the same type. E.g.,

```
  safeDiv :: Integral a => a -> a -> Maybe a
  safeDiv x 0 = Nothing
  safeDiv x y = Just (div x y)
```

Listing 90: safeDiv

Which would do error handling for division by 0.

9.3 Deriving type classes

Sometimes implementations of type classes is very easy to write but requires time. Computers are very good at doing this exact thing.

```haskell
data Shape = Circle Int | Rectangle Int Int
-- if we ran this
Rectangle 5 10
-- We would get an error saying no instance for show shape, where we could create an instance,
    or we could write
data Shape = Circle Int | Rectangle Int Int
  deriving Show
```

Listing 91: data Shape

10 Combinators

A combinator is an operator which combines. We have already seen one:

```haskell
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Listing 92: dollar

This is right associative and can help us flip the associativity of function application. It has precedence 0, meaning it is applied last. We can use it to avoid putting parentheses around things.

```haskell
double $ double 5
double (double 5)
-- These are the same thing, as $ has precedence 0, we would start with right hand side first.
```

Listing 93: dollar sign

10.1 Function Composition

In mathematics, we can compose functions together

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

In haskell, we define this with . and is as follwing

```haskell
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

Listing 94: function composition

11 Algebraic Data Types

An algebra is a system by which we assign meaning to symbols; define relationships between those symbols; and express operations between them. Haskell data types are constructed in the following ways:

- Built into the language (e.g., Char, Int, Integer)
- Built from constructors with no arguments (like True, False and Nothing)
- A function from one type to another
- The product of one or more types
- The sum of one or more types

11.1 Either

Either is defined as

```haskell
data Either a b = Left a | Right b
-- for example
data Mark = Mark Int
intTomark :: Int -> Either String Mark
  | x < 0   = Left "Too low!"
  | x > 100 = Left "Too high!"
  | otherwise = Right (Mark x)
```

Listing 95: Either

11.2 Either

Sometimes we may want to give a custom name to an already existing type, so that we ccan treat it in a special way. When we create a data type with a single unary constructor, we tend to call it a wrapper type.

```haskell
newtype Mark = Mark Int
```

Listing 96: newtype

Newtypes can also be used to manage type class instances. For example,

```haskell
newtype Backwards = Backwards String

instance Show Backwards where
  show (Backwards s) = reverse (show s)
```

Listing 97: Backwards

11.3 Recursive data types

The (:) constructor for lists takes a list as an argument - recursive. In fact, we could implement our own list also using data:

```haskell
data MyList a
  = Nil
  | Cons a (MyList a)
-- this gives us access to following constructors
Nill :: MyList a
Cons :: a -> MyList a -> MyList a
-- which are similar
OneTwoThree :: [Int]
oneTwoThree = 1 : (2 : (3 : []))
oneTwoThree' :: MyList Int
oneTwoThree' = 1 Cons (2 Cons (3 Cons Nil))
```

Listing 98: Data