Introduction

1 What is Functional Programming

```
addone x = x + 1
map addOne [1,2,3]
> [2,3,4]
```

```
def addOne(x):
        return x+1

xs = []
for x in range(1,3):
        xs.append(addOne(x))
return xs
```

In functional programming languages, functions are first class. Imperative languages (e.g. java, python) describe a sequence of steps to compute a result. Functional languages describe how to reduce an expression to a value. Often functional programming code is very concise and much easier to reason about.

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort [y | y <- xs, y <= x] ++
        [x] ++
        quicksort [y | y <- xs y > x]
```

Haskell is a general purpose, statically typed, purely functional programming language with type inference and lazy evaluation

- **General purpose**: Usable for a wide variety of programming tasks
- **Statically typed**: Type system catches errors before a program is run
- **Purely functional**: Functional language with a separation between pure and side-effecting code
- **Type inference**: Types can be deduced from program code; do not need to be specified explicitly.
- **Lazy evaluation**: Computations performed on demand

Haskell is a mature, industry grade functional language, it has directly inspired other languages, like Idris, Agda, Elm, etc. It is **opinionated** (covered later on, refers to purity, laziness, static typing)

2 Expressions and Statements

Statement:

- An instruction/computation step, doesn't return anything

Expression:

- A term in the language that eventually reduces to a value, e.g. a string, integer, etc. Can be contained within a statement Conditional Statements as Expressions

```
if 2 < 3 then
        "two is less than three"
else
        "three is less than two"
```

Since a Haskell conditional is an expression, it must have an else clause. Statements as Expressions

```
let msg =
        if 2 < 3 then
                "two is less than three"
        else
                "three is less than two"
in
        "result: " ++ msg
```

3 Types

Each expression has a type, which classify values
5 :: int
True :: Bool
False :: Bool
(1, "Hello") :: (Int, String)
Types can rule out nonsensical expressions, like 5 + True. Haskell can infer a type for most expressions, but it is good practice to add in a type signature for top-level functions

4 Reduction

Evaluation in Imperative vs Functional languages **Imperative language**:

- Program Counter + Call Stack + State
- We record our current position in the program
- Statements can alter that position
- Variable assignments alter some store

```
def addOne(x):
        return x+1

xs = []
for x in range(1,3):
        xs.append(addOne(x))
return xs
```

Now compare to the Functional version

```
map addOne [1,2,3]
 >>[addOne 1, addOne 2, addOne 3]
 >>[2, addOne 2, addOne 3]
 >>[2, 3, addOne 3]
 >>[2, 3, 4]
```

Functional reduction : reduce complex expressions to a value, reductions can be performed in any order, this is the Church-Rosser property.

Functions

1 Tuples

A tuple is an ordered sequence of expressions, with a known length

```
(1, 2, "hello") :: (Int, Int, String)
(1.0 , 1) :: (Float, Int)
() :: ()
```

These are very useful for returning multiple values from functions.

1.1 Deconstructing Tuples

We can deconstruct a pair (a tuple of length 2) by using `fst` and `snd` functions

```
fst (1, "hello") -> 1
snd (1, "hello") -> "hello"
```

We can also deconstruct by pattern matching

```
let (x,y) = (1,"hello") in x -> 1
```

2 Functions & Equations

```
\name -> "Hello, " ++ name
\n -> n + 5
```

2.1 Functions

A function maps a value to another value. Function definitions have parameters, which act as placeholders for arguments when the function is applied. In Haskell, a function is **first-class**, we can let-bind it, return it from another function, pass it as an argument, etc... Functions without a name, like the ones above, are known as **anonymous**. The backslash should be read "lambda", after the Greek letter λ

A function maps a single argument to a result, but what about functions with multiple arguments? We could use a tuple `\(n1, n2) -> n1+n2`.

This way is a little annoying, we have to construct and deconstruct a tuple whenever calling the function. Can we do better?

2.2 Currying

In practice, functions with multiple arguments are defined by nesting multiple functions

$$\underset{\text{Int}\rightarrow(\ \text{Int}\ \rightarrow\ \text{Int}\ )}{\backslash n_1 \rightarrow (\backslash n_2 \rightarrow \underset{\text{Int}\ \rightarrow\ \text{Int}}{n_1 + n_2})}$$

Takes an int, and gives a function from int to int

```
((\n1 -> (\n2 -> n1 + n2)) 5) 10
> (\n2 -> 5 + n2) 10
>  5 + 10
> 15
```

2.3 Equations

An equations gives meaning to a name

```
favouriteNumber = 5
add = \x -> \y -> x+y
```

Equations can be used to define functions more concisely.

```
add x y = x + y
min x y = if x < y then x else y
```

The left hand side cannot contain any computations

2.4 Function Equations

A **function** takes some arguments, performs some computations, and returns a **result**. Haskell has many useful functions defined in the **Prelude**

```
min :: Int -> Int -> Int
min x y = if x < y then x else y
```

2.5 Function Application

```
addOne x = x + 1

addOne 5
addOne 5 * 2
addOne (5 * 2)
```

Expressions can contain function calls (like `addOne`). In Haskell and most other functional languages you do **not** need to parenthesise arguments to functions. Function application binds tightest :

$$fx * 2 \text{ means } (fx) * 2$$

2.6 Partial Application

An advantage of currying is partial application

```
add x y = x + y
addFive = add 5
```

This allows us to specialize a function without needing to rewrite or duplicate the logic

We can compose two functions using the function composition operator.

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

$$f(.)g = x \rightarrow f(g(x))$$

Function composition is used to write code in a **point-free** style, which tries to avoid introducing variable names where possible

```
show . add10
-- vs
\x -> show (add10 x)
```

Parentheses can keep things neat and avoid ambiguity. They are needed around things like negative numbers to disambiguate between subtraction. As a matter of style, use only where necessary.

$$1 + 2 * 3 = 1 + (2 * 3) = (1 + (2 * 3))$$

2.6.1 Equations not assignments

In an imperative language, we might write something like `x := x + 1` or `x++`. This modifies the value referred to by variable `x`.

Consider `x = x + 1` in Haskell - This is **not** the same as `x:x+1` or `x++` - Tries to define x as the successor of x! - Haskell will try to calculate it though..

Reassignment is not permitted. It doesn't make sense for x to be defined as **both** 5 and 6 .

Never destroy old values, just compute new ones!

2.7 Parametric Polymorphism

```
\x -> \y -> x :: Int -> Bool -> Int
\x -> \y -> x :: String -> Int -> String
\x -> \y -> x :: Float -> Bool -> Float
```

This is the same function with many different types.

2.8 Type variables.

Polymorphic functions: have **type variables** to stand for types

```
\x -> \y -> x :: a -> b -> a
\x -> x :: a -> a
```

Can also have type variables in types (e.g. tuples and lists)

```
reverse :: [a] -> [a]
\(x, y) -> (y, x) :: (a, b) -> (b, a)
```

3 Lists & Sequences

3.1 Lists

A list is an ordered sequence of values of the same type

$$[123] :: [Int]$$

$$[\text{"}Hello\text{"},\text{"}FP\text{"},\text{"}Students\text{"}] :: [String]$$

Haskell supports a concise notation for creating ordered lists - [1..10] - $['a'..'z']$ - [1..]

Note that these are constructed **lazily** - You can construct an infinite list and only compute what you need

3.2 List Comprehensions

Remember set builder notation from Algorithmic Foundations 2

$$\text{doubleEvens} = \{2x \mid x \in \mathbb{Z}^+, x \bmod 2 = 0\}$$

List comprehensions notation allows the same construction

```
doubleEvens = [2 * x | x <- [1..], x mod 2 == 0]
-- left-hand-side = [ body | generator(s), condition ]
```

3.3 Accessing Lists

```
head :: [a] -> a -- first elem of list
tail :: [a] -> [a] -- every elem after first
(!!) :: [a] -> Int -> a -- list indexing notation
```

Note : Only use these if you know what you are doing (e.g. definitely know the number of elements in the list, i.e. non empty otherwise head and tail will error.)

- List indexing, head, tail **all potentially undefined** - Often better to use **pattern matching**

3.4 Zipping Lists

$$\texttt{zip}\ \begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}\ \begin{bmatrix}\text{“One”}\\\text{“Two”}\\\text{“Three”}\\\text{“Four”}\\\text{“Five”}\end{bmatrix} = \begin{bmatrix}(1,\text{“One”})\\(2,\text{“Two”})\\(3,\text{“Three”})\\(4,\text{“Four”})\\(5,\text{“Five”})\end{bmatrix}$$

$$\text{zip} :: [a] \to [b] \to [(a,b)]$$

$$\texttt{zip}\ \begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}\ \begin{bmatrix}\text{“One”}\\\text{“Two”}\\\text{“Three”}\end{bmatrix} = \begin{bmatrix}(1,\text{“One”})\\(2,\text{“Two”})\\(3,\text{“Three”})\end{bmatrix}$$

Zipped list: as long as the **shortest** of the two lists, is this the only/best design? Other languages might just error

3.4.1 Dependent types

This is an occasion where our type signature is not specific enough to let us know the function's behaviour. Dependant type systems allow values within types, so we can be much more specific.

$$\text{zipDep} :: \text{Vect n a} \to \text{Vect n b} \to \text{Vect n (a,b)}$$

Vect n a is a List of length n containing values of type a.

Haskell has some support for dependent types, but those wont be relevant for now.

3.5 Zipping with a Custom Function

$$\texttt{zipWith (*)}\ \begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}\ \begin{bmatrix}2\\2\\2\\2\\2\end{bmatrix} = \begin{bmatrix}2\\4\\6\\8\\10\end{bmatrix}$$

$$\text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c]$$

When would you use a tuple, and when would you use a list?

Use a tuple when: - You know the number of values you are storing - The types of the values you are storing are different - data is heterogeneously typed - tuples can have diff types

Use a list when: - You don't necessarily know the number of elements in advance - You have an ordered sequence of values of the same type - You want to operate uniformly over your data.

Another question: Which is the better type signature?
- pythagTriples :: Int → [(Int,Int,Int)] «- more descriptive type
- pythagTriples :: Int → [[Int]]

4 Haskell Building Blocks

For a quadratic formula: $ax^2 + bx + c = 0$ Calculate x using:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
def roots (a,b,c):
    det2 = b * b - 4 * a * c
    det = sqrt(det2)
    rootp = (-b + det) / a / 2
    rootm = (-b - det) / a / 2
    return [rootm,rootp]
```

4.1 Let Bindings

```
roots a b c =
    let
        det2 = b * b - 4 * a * c
        det = sqrt det2  -- can use previously bound patterns
        rootp = (-b + det) / a / 2
        rootm = (-b - det) / a / 2
    in
        [rootm, rootp] -- body has access to all vars bound in let clause
```

Equivalently

```
roots a b c =
    let det2 = b * b - 4 * a * c in
    let det = sqrt det2 in
    let rootp = (-b + det)/ a / 2 in
    let rootm = (-b - det)/ a / 2 in
    [rootm, rootp]
```

Note: each let needs a continuation (the rest of the computation). This is because its **not a statement.**

4.2 Where Bindings

```
roots a b c = [rootm, rootp]
    where
        det2 = b * b - 4 * a * c
        det = sqrt det2
        rootp = (-b + det)/ a / 2
        rootm = (-b - det)/ a / 2
```

Equivalent to **let** bindings, but bindings go *after* the function body. Often a matter of taste whether to use **where** or **let**

5 Case Expressions

In languages like C/Java, we have a switch/case statement

```
switch (num) {
    case 1: return "one"
    case 2: return "two"
    case 3: return "three"
    default: return "something else"
}
```

In Haskell, we have a case expression, this looks like

```
caseExample :: Int -> String
caseExample x =
    case x of
        1 -> "one"
        2 -> "two"
        3 -> "three"
        _ -> "something else"
```

The idea is to allow pattern matching within a definition.

5.1 Guards

Sometimes we want to do different things based on our input value, we could do this with `if-else` chains...

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa =
    if gpa >= 18
    then "A" else
    if gpa >= 15
    then "B" else
    if gpa >= 12
    then "C" else
    "below C"
```

In practice, Haskell's *Guard* notation is often cleaner: each is guarded by a boolean predicate. 'otherwise' is just synonymous for True

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa
    | gpa >= 18 = "A"
    | gpa >= 15 = "B"
    | gpa >= 12 = "C"
    | otherwise = "below C"
```

Recursion and Algebraic Datatypes

1 Recursion

So far, we have considered lists to be primitive data types

$$[1, 2, 3] :: [Int]$$

In fact, lists are **inductively-defined data structures**

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

So, $[1, 2, 3]$ is actually syntactic sugar for `1 :(2 :(3: []))`. This allows us to write recursive functions over lists.

1.1 Recursion vs Iteration

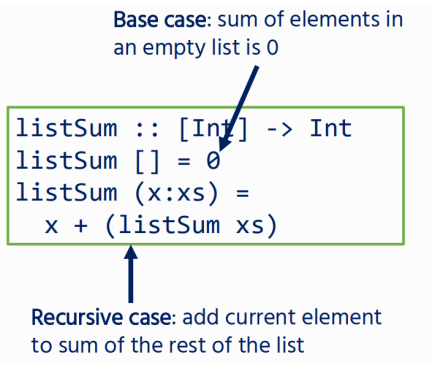
```
def listSum(xs):
    res = 0
    for x in xs:
        res = res + x
    return res
```

In **imperative** languages, we typically rely on **iteration** when working with collections of values. This is because we often perform statements using data in the collections.

```
listSum :: [Int] -> Int
listSum [] = 0    -- we pattern match on the different ways
listSum (x:xs) = x + (listSum xs)    -- of constructing a list
```

In **functional** languages, **recursion** is our basic building block. This is because we often want to use the data to construct a new result expression

1.2 Designing Recursive Function



1. Think about the structure you are defining recursion on (Integers? Lists?)
2. Write a **base case**, you will want (at least pure) functions to terminate
3. Write a **recursive** or **inductive** case which calls the same function with arguments converging to the base case.

If we have some side-effecting code it might make sense not to terminate, for example a server application,, we might keep accepting clients, sending/receiving messages forever, but when we dont have side effects, we want functions to terminate.

1.3 Example: Factorial

```
fac :: Int -> Int
fac 0 = 1
fac n =
    if n < 0 then
        error "bad argument"
    else
        n * (fac (n-1))
```

We can recurse on integers too. Just need to ensure we converge to a base case. We check negativity to avoid overshooting the base case.

1.4 Example: Fibonacci

```
fib :: Int -> Int
fib 0 = 0
fib n | n < 3 = 1
      | otherwise =
        (fib (n - 1)) + (fib (n - 2))
```

Fibonacci sequence: $fib(n) = fib(n-1) + fib(n-2)$

1.4.1 Better Fibonacci

```
betterFib :: Int -> Int
betterFib 0 = 0
betterFib goal =
    if goal < 2 then 1
    else betterFib' 0 1 2
  where
    betterFib' prev1 prev2 idx =
        if idx == goal then prev1 + prev2
        else
            betterFib' prev2 (prev1 + prev2) (idx + 1)
```

Idea: Calculate running sums as we go. Return sum of previous two running sums when we work up to goal

1.5 Tail Recursion

```haskell
listSum :: [Int] -> Int
listSum xs = listSum' xs 0
  where
    listSum' [] i = i
    listSum' (x:xs) i =
        listSum' xs (i+x)
```

To avoid stack overflow, Haskell uses **tail call optimisation**.

All tail calls (where a call is the last part of an expression) can be implemented using constant stack space.

1.6 Mutual Recursion

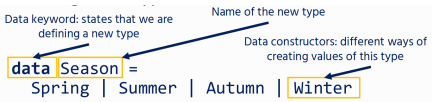
```haskell
tick :: Int -> String
tick 0 = ""
tick n =
    "tick " ++ (tock (n-1)) -- order that these are defined doesnt matter

tock :: Int -> String
tock 0 = ""
tock n =
    "tock " ++ (tick (n-1))
```

Sometimes, we write **mutually recursive** functions, which call each other. Haskell allows us to do that, since all other definitions are in scope. In other functional languages, you will sometimes need to mark these explicitly

2 Algebraic Datatypes

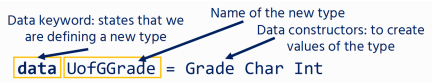
2.1 Defining Data Types



This is a *sum type*, with alternative values (like an enum in C or Java).

So far, we have mainly used built in types, `Int`, `String`, `Bool`, `[a]`. It is often useful to define our own data types. All of `Spring`, `Summer`, `Autumn`, `Winter` have type `Season`

Similarly, we can define our own product types, with a combination of values:



We could restrict to legal grades by enumerating allowed letters `A..H`, or adding validation functions

2.2 Richer Data Types with Content

```haskell
data Suit =
    Hearts | Diamonds | Clubs | Spades

data Card = King Suit | Queen Suit | Jack Suit | Number Suit Int
```

Each data constructor can have associated data. This is good for modelling, but need some behaviours (type classes)

2.3 Pattern Matching

```haskell
showCard :: Card -> String -- pattern match card values
showCard (King _)     = "K"   -- to convert to string
showCard (Queen _)    = "Q"
showCard (Number _ n) = (show n)
```

When working with a data type, we often need to pattern match to see which data constructor was used (needing brackets around compound values). If pattern matching is incomplete, GHC warns but doesnt give an error straight away, there might be an error at runtime if we encounter an unmatchable pattern.

The underscore means we *dont care*, match anything. We can bind a value to a name with the pattern matchm like n in the final line of the `showCard` function. Pattern matching can be done with equations as above or with a case expression.

3 Binary Trees

3.1 Recursive Data Types

```haskell
data Tree =
    Leaf | Node Int Tree Tree
```

Also known as inductively defined, this is a binary tree with an Int payload at each non-leaf node.

3.2 Parameterised Data Types

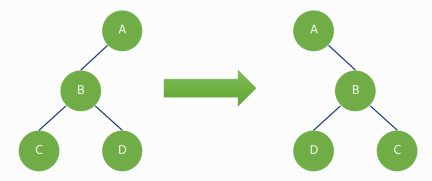
```haskell
data BinaryTree a =
    Leaf | Node a (BinaryTree a) (BinaryTree a)
```

We can parameterise a data type by a type variable on the left hand side of the data definition. An example of this would be the list type $[a]$, which can contain any type of values. In the above example, we have a tree which can contain any type of value

```haskell
data Tree =
    Leaf | Node Int Tree Tree

treeSum :: Tree -> Int
treeSum Leaf = 0
treeSum (Node x left right) =
    x + treeSum left + treeSum right
-- recursively traverse binary tree and compute sum
```

3.3 Example : Inverting a Binary Tree




```haskell
invert :: BinaryTree a -> BinaryTree a
invert Leaf = Leaf
invert (Node x t1 t2) =
    Node x (invert t2) (invert t1)
```

3.4 Challenge : TreeDepth

Write a `treeDepth` function treeDepth :: Tree -> Int which computes the max depth of th tree from the root to the furthest leaf. For example, the depth of the trees from above would be 3. can use the same pattern as `treeSum`:

```haskell
treeDepth :: Tree -> Int
treeDepth Leaf = 0
treeDepth (Node x left right) =
    1 + max (treeDepth left) (treeDepth right)
-- recursively traverse binary tree and compute *depth*
```

4 The Maybe Type

An example of parametric polymorphism, a is a type variable.

```haskell
data Maybe a = Just a | Nothing -- almost like a type safe Null

safeHead :: [a] -> Maybe a
safeHead []  = Nothing
safeHead (x : _) = Just x
```

If we have potentially failing computations, the `Maybe a` type says that either there is some data (`Just a`) or there isnt (`Nothing`), We pattern match to find out whether the computation failed so that we can handle all eventualities. This is like the `Option` type in Rust, Scala, or Java

```haskell
emote :: String -> Maybe String
emote "happy" = Just ":)"
emote "sad" = Just ":("
emote "realised FP quizzes were due a week early on Moodle" = Just ":|"
emote _ = Nothing
```

This requires pattern matching to handle both success (Just) and failure (Nothing).

Higher Order Functions & Property Based Testing

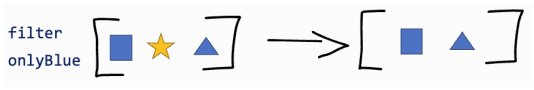
1 Higher-Order Function

A higher-order function is a function which takes another function as an argument. You may have seen several of these already:

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

It is good practice to use HOFs where applicable rather than hand-roll recursive functions yourself. Concentrate on application logic rather than recursive boilerplate.

1.1 Map

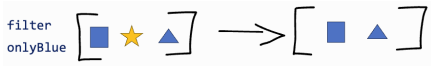



```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) =
    -- f :: (a -> b)
    -- x :: a
    -- xs :: [a]
    -- (:) :: c -> [c] -> [c]
    -- (f x) :: b
    -- map f xs :: [b]
    (f x) : (map f xs)
```

A map applies a function to every element in a list. It is defined recursively by **building up a new list** with a function applied to each element.

Map fusion: map f (map g xs) $\leftrightarrow$ map (f . g) xs

1.2 Filter




```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x : xs) =
    if (f x) then (x : filter f xs)
    else filter f xs
```

Filter retains every element which satisfies a predicate. Only prepend the element if the predicate evaluates to **true**.

In terms of list comprehensions,

`map f xs` is equivalent to `[f x | x <- xs]`

filter p xs is equivalent to `[x | x <- xs, p x]`

Can you define list comprehensions in terms of filter and map?

For simple list comprehensions, yes, for example, doubling eery even number.:

```haskell
[2 * x | x <- [0..], x `mod` 2 == 0]
= map ((*) 2)
    (filter (\x -> x `mod` 2 == 0) [0..])
```

However, this doesn't scale to multiple generators, e.g. Pythagorean triples:

$$[(x, y, z) \mid x \leftarrow [0..], y \leftarrow [0..], z \leftarrow [0..], x^2 + y^2 == z^2]$$

For this we need some way of flattening intermediate lists.

2 Folds

A fold is a way of reducing lists into a single value. We have a function which takes an element of the list, and an accumulator value, producing a new accumulator.

We have two types of fold, depending on the desired associativity:

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> [a] -> b
```

2.1 Left Fold

```haskell
foldl (+) 0 [1,2,3,4]
    (((0+1)+2)+3)+4
        10
```

Since `foldl` is left-associative: - Brackets are grouped from the left - The list is traversed from left-to-right

We can see how this works in GHCI:

```haskell
f = \x y -> "(f" ++ x ++ (show y) ++ ")"
foldl f "0" [1..5]
-- "(f(f(f(f(f 0 1) 2) 3) 4) 5)"
```

2.2 Right Fold

```haskell
foldr (+) 0 [1,2,3,4]
1+(2+(3+(4+0)))
10
```

Since `foldr` is right-associative: - Brackets are grouped from the right - The list is traversed from right-to-left

Again, in GHCI;

```haskell
f = \x y -> "(f" ++ x ++ (show y) ++ ")"
foldr f "0" [1..5]
-- "(f 1 (f 2 (f 3 (f 4 (f 5 0)))))"
```

2.3 Fold Definitions

```haskell
-- left fold
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc [] = acc
foldl f acc (x : xs) = foldl f (f acc x) xs

-- right fold
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc [] = acc
foldr f acc (x : xs) = f x (foldr f acc xs)
```

Main difference: - In `foldl`, the recursive call is outermost. - In `foldr`, the user-supplied function is outermost.

3 Property-Based Testing with QuickCheck

3.1 Property Based Testing

You have a program, you specify some properties that should be true for that program, coded up as boolean predicates. The QuickCheck tool for Haskell can be rule with your boolean predicates to generate lots of random, type correct input data and check that the properties hold true.

Note that program testing can be used to show the presence of bugs, not their absence.

3.2 Example - Length of a List

The property that we're testing is: " a list containing n elements has length n"

```
let prop_len = \n -> (length [1..n] == n)
```

Properties must return a boolean value, we can test the property above with `quickCheck prop_len`

There is a problem, our property is incorrect. A better property to test would be "a list containing n elements has length n, for non-negative integer values n"

```
let prop_len = \n -> (if n>= 0 then length [1..n] == n else True)
```

This should now work, and we can use `verboseCheck` to see all the test cases with their randomised input values

3.2.1 QuickCheck Behaviour

It looks for the simplest case it can find that violates a property, then reports this. In general, a lot of research in property-based testing has concentrated on shrinking counterexamples into a minimal case.

3.3 Another Example

Lets test a new property, that a list is sorted in strictly ascending order

```
isAscending :: [Int] -> Bool
isAscending [] = True
isAscending [x] = True
isAscending [x:y:xs] = (x < y) && isAscending xs
```

Can we think of some properties to test this function?

- Any sublist of the natural numbers beginning at one should be ascending
  `prop_asc1 = \n isAscending (take n [1..])`
- List containing repeats of a single value are not ascending
  `prop_asc2 = \n -> if n < 2 then True else not (isAscending (take n (repeat 1)) )`
- Adding the length as an element to the end of an ascending list makes it not ascending
  `prop_asc3 = \n  > if n<=0 then True else not (isAscending ([1..n]++[n]))`

property 3 fails, because our `isAscending` is wrong, what we *actually* need to do, is this;

```
isAscending :: [Int] -> Bool
isAscending [] = True
isAscending [x] = True
isAscending [x:y:xs] = (x < y) && isAscending (y:xs)
                                        -- ^^ note this change
```

Evaluation Strategies and Polymorphism

1 Evaluation Strategies

Expressions are evaluated as a program runs. The order of expression evaluation depends on both the language semantics and the implementation pragmatics.

- **Eager evaluation** - also known as strict evaluation or call by value - **Lazy evaluation** - also known as non-strict evaluation or call by need

```
fib :: Int -> Int
-- Naïve Fibonacci: expensive to evaluate

useFirst :: (a -> c) -> a -> b -> c
-- Function that only uses first argument

useFirst id 42 (fib 1000)
```

In python, this would take a very long time to evaluate. In Haskell, it would terminate immediately.

1.1 Benefits of lazy evaluation

We can compute with large data structure, including potentially infinite lists, and only evaluate what we need.. We can compute with expensive functions, which are only evaluated when we need. We can compute with dangerous values, such as undefined or bottom - this will only cause problems if they are evaluated.

- undefined raises a runtime exception when evaluated, but has a generic type so can be used in any expression context. - let bottom = bottom is an infinite defined type whose evaluation will loop forever (again, it has a generic type).

1.2 Drawback of lazy evaluation

Difficult to combine with exception handling since the order of evaluation is unclear. Sometimes more difficult to predict, e.g. lazy IO. Runtime memory pressure as an evaluation graph builds up, full of unevaluated expressions (thunks) - evaluation is only triggered when a value is 'needed' - e.g. when output via IO.

1.2.1 Examples of Lazy Evaluation

C operators `&&` and `||`, python 3 `range()` function.

In general, most languages (imperative, OO, or functional) operate with eager evaluation, haskell is unusual in this sense.

1.3 Some Interesting Infinite Lists

```
[1..]
-- the list of all non-negative integers

let ones = 1:ones
-- an infinite list of 1 values

primes = sieve [2..]
  where sieve (x:xs) = x:(sieve [x' | x' <- xs, x' `mod` x /= 0])
    -- infinite list of primes via sieve of eratosthenes

fibs = 1:1:( zipWith (+) fibs (tail fibs))
    -- infiniite list of function applications to initial value
```

Some useful functions for infinite lists are:

```
take :: Int -> [a] -> [a]
-- selects the first n elements from a list.

repeat :: a -> [a]
--  produces an infinite list of repeated values.

cycle :: [a] -> [a]
--  produces an infinite list by appending the list to itself infinitely.

iterate :: (a -> a) -> a -> [a]
--  produces an infinite list of function applications to an initial value
--  [x,f x, f (f x) ...].
```

2 Polymorphism

The idea that code can operate over values from a variety of different types, enabling code reuse, and sometimes helping us to reason about program behaviour

Parametric polymorphism: Functions operate on the *shape* of the arguments rather than with the data, so can operate on many types. Uses type variables.

Ad-hoc polymorphism: Like method overriding in Java. Same function names but different implementations for different types. Achieved in Haskell using typeclasses.

2.1 Parametric Polymorphism

Some classic examples:

```
id :: a -> a
const :: a -> b -> a
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

a and **b** are type variables. These will be specialised to concrete types when the map is called on concrete values. The behaviour of map does not depend in any way on the types of the list elements - It just applies the function to each element.

There is precisely one function with type `a -> b -> a`

```
const :: a -> b -> a
const x _ = x
```

This is one consequence of parametricity

We can define our own polymorphic functions, for example;

Duplicate and put in a list

```
dapial :: a -> [a]
dapial x = [x,x]
```

Extract element from triple

```
takeLast :: (a,b,c) -> c
takeLast (x,y,z) = z
```

Both of these only manipulate rather than use the variables.

2.2 Polymorphic Data Types

We can also use type variables within data type declarations:

```
data Tree a =
    Leaf a
  | Node (Tree a) (Tree a)
```

`Tree` can contain values of any type within it (although once it is specialised, we have fixed the concrete type).

```
Node (Leaf 5) (Leaf 10) :: Tree Int
Node (Node (Leaf "Hello") (Leaf "World"))
     (Leaf "!") :: Tree String
```

Functions for polymorphic container types

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Node t1 t2) = Node q1 q2
  where
    q1 = treeMap f t1
    q2 = treeMap f t2
```

2.3 Ad-hoc Polymorphism via Typeclasses

Method or operator overloading in Java is a form of *ad-hoc polymorphism*. Ad-hoc is not meant to be a bad thing - just that, unlike parametric polymorphism, it is core part of the type system. It is useful in allowing us to perform similar operations (e.g. addition, converting to a string...) on different types where each operation acts potentially differently on each type.

In Haskell, ad-hoc polymorphism is achieved using typeclasses. A typeclass specifies a list of operations to be defined for a type. In a sense, typeclasses bring parametric and ad-hoc polymorphism closer together. Since we can say "we don't care what a particular type *a* is, only that it supports certain operations".

2.4 Typeclasses

```
class Show a where
  show :: a -> String
```

For type a to be part of the Show typeclass, we must implement a function show (which take a and returns a string)

Only values with a show function can be represented as strings, but we can use the **Show** typeclass to indicate that a value has a show function defined.

```
show :: (Show a) => a -> String
-- typeclass constraings go on the left hand side of the big arrow
```

How do we show that a type belongs to a typeclass? We can specify that an algebraic data type belongs to a typeclass in two different ways;

- `instance` declaration - Here we specify the implementations of each function explicitly
- `deriving` clause - This will implement the typeclass using default behaviour

```
data Insect = Spider | Centipede | Ant
```

2.5 Instances

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

class Show a where
  show :: a -> String
```

We can specify that a type belongs to a typeclass by writing instance declarations that specify the implementation for each function specification in the class.

```
instance Eq Insect where
  Spider == Spider = True
  Centipede == Centipede = True
  Ant == Ant = True
  _ == _ = False

instance Show Insect where
  show Spider = "Spider"
  show Centipede = "Centipede"
  show Ant = "Ant"
```

2.6 Deriving

```
data Insect = Spider | Centipede | Ant
   deriving (Show, Eq)
```

The deriving clause provides us with default implementations for the appropriate functions, here show and (==) and (/=).

```
Ant == Ant
Spider /= Ant
show Centipede -> "Centipede"
```

This is possible because we can derive a string by using the data constructor, and declare two insects equal if they use the same data constructor. If the data constructors had associated data, in order to support deriving, their types would need to support Show/Eq too.

2.7 When to use Deriving vs. Writing an Instance

Deriving - When you want "default" behaviour / behaviour is as you would expect. - When the behaviour is already specified for any associated data. - When the typeclass supports deriving.

Instance - When the typeclass does not support deriving (e.g. a custom typeclass). - When you want behaviour that's different to the default (e.g. you want show to pretty-print an abstract syntax tree).

2.8 The Read Typeclass

read allows us to convert styring values into other values, like input in Python or scanf in C.

```
data Insect = Spider | Centipede | Ant
    deriving (Read, Show, Eq)

(read "Centipede") :: Insect
```

note that we must give an explicit type annotation!

2.9 Example : The Leggy Class

```
class Leggy a where
  numLegs :: a -> Int
```

`a` is the type variable that is a placeholder for the concrete type that will be an instance of the typeclass. We will use `a` in the type specifications of the functions

Now, we can add instance to the typeclass

```
instance Leggy Insect where
  numLegs Spider = 8
  numLegs Ant = 6
  numLegs Centipede = 100
```

Note:

- Put the typeclass name before the instance name read "Insect is an instance of the Leggy typeclass"
- WE have to provide definitions for all functions specified in the typeclass

```
describe :: (Show a, Leggy a) => a -> String
describe x = (show x) ++ "s have " ++
             (show $ numLegs x) ++ " legs"
```

This could just have the type `describe :: Insect -> String`. But the above code is much more extensible - we can describe any instance of Leggy.

```
data Mammal = Human | Dog | Cat | Pig
    deriving (Show, Eq)

instance Leggy Mammal where
  numLegs Human = 2
  numLegs _ = 4
```

and now we can describe Mammal values too, so *ad-hoc polymorphism* enables type-safe code reuse.

Intro to IO

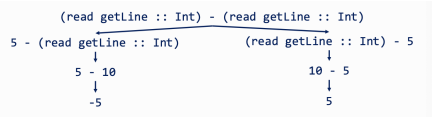
1 Purity

So far, all the haskell we have seen has been *pure* code, but what does functional purity *mean*:
Stateless - The function always evaluates to the same result given the same arguments
Side-effect free - The function does nothing apart form the simple expression evaluation, no interaction from the 'outside world'
Total : Defined for all inputs (ideally). Not everything is total, even in prelude, head or tail of empty lists are good examples

However, the problem with purity is that code *needs* to do I/O - to get input from peripheral devices, send output to the user, and to read/write files.

These are externally visible operations - side effecting code. How do we manage this in a pure functional language? - how do we integrate pure and impure code?

1.1 The naive approach



We lose the Church-Rosser property, and referential transparency - the property that means we can replace a piece of code with the value is produces. It also wreaks havoc with lazy evaluation. Generally speaking, we lose all our reasoning power.

1.2 IO Types

Key idea: Each side-effecting operation is marked with a type constructor, IO

```
putStrLn :: String -> IO()
```

- `()` denotes unit type, contains no information
- `IO()` : computation that will produce the unit value
- `String -> IO()` Function that takes a string, and returns a computation that will produce the unit value

1.2.1 Mixing Pure Functions and IO Computations

```
reverse getline -- This does not work

reverse :: String -> String
getLine :: IO String
```

A string is not the same as an IO String. A string is **data**, an IO string is a **computation** which produces a string. Side effecting computations are **always** marked in their types.

Instead of the above, we should do:

```
getAndPrintReverse :: IO()
getAndPrintReverse = do -- marks that we're putting together a computation
    str <- getLine  -- gives a name (str) to the result of an io operation
    let revStr = reverse str  -- bind result of rev functio to revStr
    putStrLn sevStr -- print reversed string to console
```

Similarly, IO functions can return values, with the following changes to the above function:

```
getAndPrintReverse :: IO String -- define IO computation returns a String
getAndPrintReverse = do
    str <- getLine
    let revStr = reverse str
    return sevStr
    -- ^^ use the return function, revstr is string and we need IO string
```

Note that the return function has the type `return :: a -> IO a`

1.3 The bigger Picture

As with other languages, every haskell program has an entry point, main

```
main :: IO()
main = do
    line <- getLine
    putStrLn (makeUpper line)
```

main function is evaluated when a program is run. It can make use of other functions with IO type. GHCI runs everything as an IO computation - which is why we can print things out. It is good practice to keep as much of the program as pure as possible.

1.3.1 Escaping IO

There is (almost) no way to "project" a value out of an IO computation. While you might want a function with type `IO String -> String` This doesn't make sense, we would run into the same issues with church-rosser as before.

Instead, think of IO as though you are using do-notation to build a bigger computation by stringing together smaller IO computations, with `main` as your entry point.

One way you could project out a value would be using `System.IO.Unsafe`, and the `unsafePerformIO :: IO a -> a` function. This is not idiomatic haskell and should only be used when you really know what you're doing.

It is sometimes useful to do 'print debugging', where we wish to print some program state to the console. We can do this using the trace function.

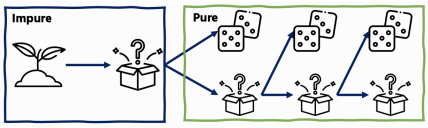
```
import Debug.Trace
trace :: String -> a -> a    -- trace "returning 42" 42
```

This uses unsafePerformIO under the hood, and should only be used for debugging.

1.4 Beyond Console IO

IO encompasses a large number of impure operations beyond just reading/writing to a console. - Getting current time - `getCurrentTime :: IO UTCTime`
- File operations - `readFile :: FilePath -> IO String` or `writeFile :: FilePath -> String -> IO()`
- Also sockets, graphics, printing, spawning processes

1.5 Pseudo-Random Number Generation



Random number generation might seem to be an impure operation, however a psuedo-random number generator generates a random value and a new generator. Only seeding the PRNG is impure, generation is pure.

With IO, we can make use of mutable reference cells that store some data, and its contents can be changed:

```
-- Creates a new IO reference with an initial value of type a
newIORef :: a -> IO (IORef a)
-- Reads the contents of an IORef
readIORef :: IORef a -> IO a
-- Writes to an IORef
writeToIORef :: IORef a -> a -> IO ()
```

Intro to Monads

1 Potentially-Failing Computations

Remember the Maybe type? `data Maybe a = Just a | Nothing`. The `Maybe a` data type is a 'type-safe' null and is used to define potentially failing computations.

For example, both `Just 5` and `Nothing` can have type `Maybe Int` - Although only `Just 5` actually contains an Int. Whenever we want to use a value of type `Maybe a`, we'll need to case split to see whether it contains a value or not.

1.1 Managing Multiple Maybes

The Maybe type allows us to return Nothing if a computation fails

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

Suppose we want to use the `safeDiv` function twice and add the results, how do we deal with multiple values with a Maybe type?

```
-- divAndAdd : divide numbers x by d1 and y by d2,
-- and add the results together --
divAndAdd :: Int -> Int -> Int -> Int -> Maybe Int
divAndAdd x y d1 d2 =
    case (safeDiv x d1, safeDiv y d2) of
        (Just x', Just y') -> Just(x' + y')
        (_,_) -> Nothing
```

This is horrible to write, we have to write a lot of boilerplate to check whether a value is present or not. Worse yet, this style may lead to deeply nested case statements, making code difficult to read, write, and maintain.

We want to find a way that we can write `divAndAdd` as if each `safeDiv` computation succeeds, and have it automatically evaluate to `Nothing` if any subcomputation fails.

Recall the following: - We have higher order functions - With backticks we can use any functions as an infix operator.

We can write a function that takes a `Maybe a`, and a function that we can pass the unwrapped value to if it exists, and if it doesn't, we return `Nothing` overall

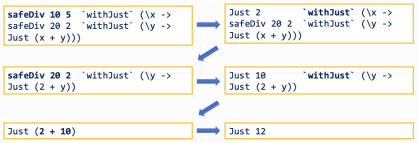
```
      -- the potentially
      -- failing computation          The final result
      --    V                            V
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b
                  --      ^
                  -- the function to
                  -- run with the just
                  -- value if it exists
withJust Nothing _ = Nothing
withJust (Just x) f = f x
```

Using the function above ...

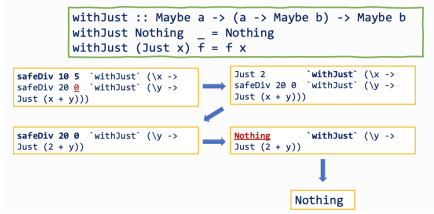
```
safeDiv 10 5 `withJust` (\x ->
safeDiv 20 10 `withJust` (\y ->
Just (x + y)))
```

`safeDiv 10 5` is our potentially failing computation. We use `withJust`, which tries to unwrap and returns `Nothing` otherwise. In the continuation function, x has type Int, as we've already unwrapped it in `withJust`

given valid inputs, this functions computes the following with these steps:



However, if we try to divide by 0,



2 Nondeterministic Computations

Sometimes we might want to have a computation that returns multiple possible results: we call this a nondeterministic computation. For example, a coin toss can either return heads or tails. How about if we wanted to build a list of the possible outcomes of two tosses?

```haskell
coinToss :: [CoinToss]
coinToss = [Heads,Tails]
```

We want `[(Heads, Heads), (Heads, Tails), (Tails, Heads), (Tails, Tails)]`

We want to implement this by drawing the first element of the first list, and use that result for the remaining computation. Then draw the first element of the second list, and return the pair. Then draw the second element from the second list, and return the pair. Repeat this for the second element of the first list.

Can we use a similar approach as with Maybe?

```haskell
withEach :: [a] -> (a -> [b]) -> [b]
withEach [] _ = []
withEach (x:xs) f = f x ++ (withEach xs f)

-- equivalently --

withEach :: [a] -> (a -> [b]) -> [b]
withEach xs f = concat (map f xs)
```

2.1 Coin tosses using `withEach`

```haskell
twoCoinTosses :: [(CoinToss, CoinToss)]
twoCoinTosses =
    coinToss `withEach` (\x ->
    coinToss `withEach` (\y ->
    [(x,y)]))
```

`[[(Heads, Heads)] ++ [(Heads, Tails)] ++ [(Tails, Heads)] ++ [(Tails, Tails)] == [(Heads,Heads), (Heads,Tails), (Tails,Heads), (Tails,Tails)]` Giving us our desired result.

We *could* use a list comprehension of this, but note that this requires multiple generators, which we cant express with just map and filter. In fact, list comprehensions are just map and filter. In fact, list comprehensions are syntactic sugar, implemented using an analogue of `withEach`.

3 IO Recap

So far we've used do-notation to build up IO computations. If you think about it though, we are describing a computation that runs as an IO computation and uses its result in the rest of the computation.

```haskell
getandPrintReverse :: IO()
getAndPrintRevese = do
    str <- getLIne
    let revStr = reverse str
    putStrLn revStr

-- What about writing IO computations without do?

withIOResult :: IO a -> (a -> IO b) -> IO b
getAndPrintReverse' :: IO()
getAndPrintReverse' =
    getLine `withIOResult` (\str ->
    let revStr = reverse str in
    putStrLn revStr)
```

The implementation of `withIOResult` maps to a primitive and is handled by the runtime system. It runs the IO computation, and applies the given function to the result. Nevertheless it allows us to write our `getAndPrintReverse` function without do-notation.

4 Monads

```haskell
-- consider the following, can we see a pattern?
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b
withEach :: [a] -> (a -> [b]) -> [b]
withIOResult :: IO a -> (a -> IO b) -> IO b
```

in each case, we have

- A computation producing a result, be it potentially-failing, nondeterministic, or side-effecting
- A function that builds a bigger computation from the result
- The overall result being the bigger computation

We can generalise the pattern we have seen so far - Maybe, List, and IO are all instances of a more general pattern known as a monad.

To be a monad, a data type needs two things:

- A way of constructing a trivial computation
  - `a -> Maybe a` - we can use Just
  - `a -> [a]` - we can use the singleton list constructor
  - `a -> IO a` - it's a library function, but it creates a pure computation without side effects
- A way of building a larger computation from the result of a previous one

4.1 The Monad Typeclass

Every monad must also be an applicative functor.

```haskell
class (Applicative m) => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
```

- Every monad must also be an applicative functor.
- Return : Injects a pure value into the monad, for example, `return 5 = Just 5` in the Maybe monad
- `>>` - pronounced sequence : runs but ignores first computation, returns the result of the second, derivable from the definition of `>>=`. - `m1 (>>) m2 = m1 >>= \_ -> m2`
- `>>=` - pronounced bind : allows us to build up a computation, takes a computation of type `m a`, and a function `a -> m b` to build a new computation `m b` and returns `m b`

A monad is just a data type that implements the monad typeclass, and satisfies the monad laws.

4.1.1 Example: The Maybe Monad

The Maybe Monad will stop at Nothing. We can sequence many computations, but if any return Nothing, then the whole computation returns Nothing. We are implementing `>>=` the same as our `withJust` function

```haskell
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing >>= f = Nothing
```

we can also manage multiple maybes monadically

```haskell
divAndAdd' :: Int -> Int -> Int -> Int -> Maybe Int
divAndAdd' x y d1 d2 = safeDiv x d1 >>= (\res1 ->
                       safeDiv y d2 >>= (\res2 ->
                       return (res1 + res2)))
```

Here we use the bind operator `>>=` and provide a function where we assume each safe division has succeeded. We can then use return to create a `Maybe Int` from `res1 + res2`. If any sub-computation fails, then the whole result will be `Nothing`.

4.2 do-notation

Since IO is a monad, we can also write IO computations using explicit `>>=` notation without needing `do`.

```haskell
greetReverse :: IO()
greetReveres = do
  name <- getLine
  let reverseName =
    reverse name
  putStrLn "Hello"
  putStrLn reverseName
```

Becomes

```
greetReverse :: IO()
greetReverse =
  getLine >>= \name ->
  let reverseName =
    reverse name in
  putStrLn "hello" >>
  putStrLn reverseName
```

do notation is syntactic sugar – it is widely used and very useful, but its sometimes quicker and more concise to write the monadic expression directly

do-Notation	Monadic notation
`do x <- M` `  N`	`M >>= \x -> N`
`do M` `  N`	`M >> N`
`do let x = M` `  N`	`let x = M in N`

Do notation works for any data type that is a member of the monad typeclass, not just for IO. We can rewrite our Maybe example from earlier using do notation:

```
divAndAdd' :: Int -> Int -> Int -> Maybe Int
divAndAdd' x y d1 d2 = do res1 <- safeDiv x d1
                          res2 <- safeDiv y d2
                          return (res1 + res2)
```

4.3 Monad typeclass hierarchy

Recently the monad typeclass changed so that each monad must also be an instance of two other typeclasses called Functor and Applicative. There is another proposal to remove return from the monad typeclass and rely on `pure :: (Applicative f) => a -> f a` from applicative. This makes sense mathematically but is terrible pedagogically, explicit definitions of return now trigger a warning in recent versions of GHC.

More Monads

1 Revisiting The Monad Typeclass

Recall it has two characteristic methods, `return` and bind (`>>=`). The `return` function puts something into the monad, i.e. boxed it up into the structure. The bind function injects a monad value into an arbitrary function `f` as a parameter, with the added constraint that `f` returns a value in the same monad — i.e. wrangles the types so functions can take monadic values as parameters.

2 Maybe Monad

We will motivate these somewhat abstract concepts with the now familiar Maybe datatype, which is monadic. Consider a function `allButLast :: [Char] -> Maybe [Char]`. This function will return the first (n-1) characters of a string with n characters, wrapped up as a Just value. If the input value is empty, then the function evaluates to nothing.

```
allButLast [] = Nothing
allButLast [x] = Just []
allButLast (x:xs) =
  let rest = allButLast xs
  in case rest of
    Nothing -> Just [x]
    (Just xs') -> Just (x:xs')
```

Now let's think about what happens when we repeatedly apply this function to a string input value, eventually we will run out of characters so we will bottom out at the Nothing value.

```
take 10 \$ iterate (x → x >>= allButLast) (Just "abcde
```

This should return `Nothing`

3 Identify Monad

Next, the `Identity` monad: We need to import `Control.Monad.Identity`, which might require some library configuration in ghci... perhaps: `set -package mtl` on the interactive prompt.

We can put something into the Identity monad with the `return` function: `(return 42) :: Identity Int`

and we can apply a function to Identity monadic values with `(>>=)`: `((return 42) :: Identity Int) >>= (x -> return (x+1))`

We extract a value out of an Identity computation by running the monad, using the `runIdentity :: Identity a -> a` function.

```
runIdentity $ ((return 42) :: Identity Int) >>=
(x → return (x+1))
```

suppose the equivalent case of running the IO monad is the invocation of the main function at top level in Haskell.

4 List Monad

We can think of using a monad as being like putting a value into a structure, elevating or lifting the value into the monad. What structures do we already know in Haskell? The list is probably simplest, and it is a monad.

`Listreturn` simply puts a value into a singleton list (i.e. syntactically, just put square brackets around it)

```
listreturn :: a -> [a]
listreturn x = [x]
```

List bind takes each element out of the list, applies a function to that element, giving a list result, then concatenates all the results into a single, new results list. The key point (which confuses some people) is that all the results are glued together into a single list, rather than being separate sublists ... like they might be with a `map` function call.

```
listbind :: [a] -> (a->[b]) -> [b]
listbind xs f = concat $ map f xs
```

Here is a simple example of list bind:

```
ghci> let f x = [x,x]
ghci> f 2
[2,2]
ghci> [1,2,3] >>= f
[1,1,2,2,3,3]
```

Here is another example:

```
ghci> let f = x -> (show x) ++ " mississippi... "
ghci> [1,2,3] >>= f
"1 mississippi... 2 mississippi... 3 mississippi... "
```

So we see that list bind is reminiscent of a `join` in Python, or a `flatMap` in Java/Scala.

5 Reader, Writer and State Monads

So far we have looked at fairly straightforward monads, many of which you have seen before. We are going to look at three useful library Monads. For each one, we will look at their API and a typical use case. These examples *might* be helpful for your coursework, coming up in a few weeks.

These three monads involve an *environment*, which we pass around, threading it from function context to function context. Typical use cases for each monad are as follows:

- `Reader` - shared environmental configuration
- `Writer` - logging operations
- `State` - computing a large value recursively

5.1 Reader

The `Reader` monad is also known as the **environment** monad. It's useful for reading fixed values from a shared state environment, and for passing this shared state between multiple function calls in a sequence.

Here is a trivial example:

```
import Control.Monad.Reader
--  ^^^ may need some GHCi hackery ...

hi = do
  name <- ask
  return ("hello " ++ name)

bye = do
  name <- ask
  return ("goodbye " ++ name)

conversation = do
  start <- hi
  end <- bye
  return (start ++ " ... " ++ end)

main = do
  putStrLn $ runReader conversation "jeremy"
```

The Reader structure has two parameters, one of which is the environment (here a string) and the other is the result of the computation. So the Reader datatype is parameterised on two type variables: `Reader environment result`

The `runReader :: Reader r a -> r -> a` function takes a Reader expression to execute, along with initial environment, and extracts the final value from it.

5.2 Writer

The `Writer` monad builds up a growing sequence of state (think about logging a sequence of operations). Officially, this state is a *monoid*, but we will only consider `String` values for today.

The `tell` function appends something to the log.

```
import Control.Monad.Writer

addOne :: Int -> Writer String Int
addOne x = do
  tell ("incremented " ++ (show x) ++ ",")
  return (x+1)

double :: Int -> Writer String Int
double x = do
  tell ("doubled " ++ (show x) ++ ",")
  return (x*2)

compute =
  tell ("starting with 0,") >> addOne 0 >>= double >>= double >>= addOne

main = do
  print $ runWriter (compute)
```

5.3 State

Finally, the `State` monad. This is very similar to the `Reader` and `Writer` monads, in that it allows us to pass around a shared state, but also to update it.

We can `get` the current state, or `put` a new value into the state.

Let's do this with the Fibonacci function ...

```
import Control.Monad.State

fibState :: State (Integer, Integer, Integer) Integer
fibState = do
  (x1,x2,n) <- get
  if n == 0
  then return x1
  else do
    put (x2, x1 + x2, n - 1)
    fibState

main = do
  print $ runState fibState (0,1,100)
```

More More Monads

1 Semigroups

Think about a set of elements `S` with an associative binary operator $\oplus$. In Haskell terms, $\oplus$ has type `S->S->S`, i.e. it takes only two parameter values from S and returns a single result value in S. The only constraint imposed on $\oplus$ is associativity, i.e. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

This abstract mathematical structure $(S, \oplus)$ is known as a semigroup. There is a corresponding Haskell typeclass, called `Data.Semigroup` with the characteristic function being the infix operator `(<>) :: a -> a -> a`

The haskell typeclass definition is as follows:

```
class Semigroup m where
  (<>) :: m -> m -> m
```

Note that haskell is not expressive enough to specify the associativity constraint, we have to be aware of this as programmers.

Examples of semigroups include:

- the set of positive integers with addition
- the set of all finite strings over a fixed alphabet with string concatenation
- 2x2 square non-negative matrices with matrix multiplication

2 Monoids

A *monoid* is a set S with an associative binary operation $\oplus$ and an identity element e. The identity element must satisfy the constraint that $\forall a \in S.\ e \oplus a = a$ and $a \oplus e = a$. In effect, a monoid is a semigroup with an identity element. The identity element is unique within the monoid.

The Haskell typeclass definition is as follows:

```
class Semigroup m => Monoid m where
  mempty :: m

  -- defining mappend is unnecessary, it copies from Semigroup
  mappend :: m -> m -> m
  mappend = (<>)

  -- defining mconcat is optional, since it has the following default:
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

All instances of Monoid must satisfy the following laws:

```
-- left and right identity
x <> mempty == x
mempty <> x == x

-- associativity
(x <> y) <> z == x <> (y <> z)
```

Again, we can't express these constraints in the typeclass definition — its up to the haskell developer to ensure the laws are respected by Monoid instances.

2.1 List as a Monoid

The most straightforward example of a monoid instance is the list datatype:

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```

The list concatenation operation is `mappend`, and the empty list is the identity element. So we can construct lists using a new syntax now ... `[1] <> [2] <> [3,4,5] <> mempty` etc.

2.2 Sum as a Monoid

It's easy to see why lists are monoids - the `mempty` element and the `mappend` operation are obvious. How about integers? If the `mappend` operation is integer addition, then the `mempty` element is 0. Lets express this in haskell:

```
newtype Sum n = Sum n

instance Num n => Semigroup (Sum n) where
  Sum x <> Sum y = Sum (x + y)

instance Num n => Monoid (Sum n) where
  mempty = Sum 0
```

there is a library `Sum` datatype defined in Data.Semigroup already.

Now we can construct integer values using monoid syntax: `Sum 0 <> Sum 1 <> Sum 100`

2.3 Writers use Monoids

Lets revisit `Writer` and build up a sum accumulator value in the log, counting how many operations we perform:

```
-- our writer state accumulates the number of
-- arithmetic operations performed
-- (i.e. subtractions and multiplications)

fact :: Integer -> Writer (Sum Integer) Integer
fact 0 = return 1
fact n = do
  let n' = n-1
  tell $ Sum 1
  m <- fact n'
  let r = n*m
  tell $ Sum 1
  return r
```

2.4 Foldables *also* use Monoids

Now we know about monoids, we recognise that if we have an identity element `mempty`, and an associative combining function `mappend`, then we can easily fold over an arbitrary collection of monoid values. If we import `Data.Foldable`, then we can use `fold :: (Foldable t, Monoid m) => t m -> m` and `foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m`, which, when given a Monoid m, these functions know how to aggregate values using `mappend`:

```
fold [[1,2], [3,4], [5]]
-- evaluates to [1,2,3,4,5]

fold (Sum (Sum 3))
-- evaluates to (Sum 3)

foldMap Sum [1..5]
```

3 Foldable Trees

Recall the Binary Tree container data structure. We can make this type an instance of the `Foldable` typeclass, simply by providing a definition of the `foldMap` function (or, alternatively and equivalently, a definition of the `foldr` function).

Notice that a `Leaf` value will map onto a `mempty` value, and a non-empty `Node` value is recursively combined with `mappend` operations:

```
import Data.Semigroup
import Data.Foldable

data Tree a = Leaf
            | Node a (Tree a) (Tree a)
  deriving (Show,Eq)

instance Foldable Tree where
  foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap _ Leaf = mempty
  foldMap f (Node x left right) =
    f x <> foldMap f left <> foldMap f right
```

Parser Combinators

1 What is Parsing?

A *parser* is a program that recognizes sentences from a grammar. Sometimes a parser may be hand-written but often it is synthesized from a higher-level grammar description by a parser generator tool (antlr, yacc, etc). The parser is generally *table*-driven or *rules*-driven.

In Haskell, there is an alternative approach involving the use of *parser combinators*, whereby a parser can be constructed incrementally based on combining functions.

Informally, a *combinator* is a higher-order function that combines 'things' to create more complex 'things' of the same type. For example, we have already looked at list combinators like `map` and `filter`. A *parser combinator* is a higher-order function that combines two simpler parsers (sub-parsers) to create a compound parser.

2 The Parsec Library

`parsec` is a Haskell parser combinator library. You can install it on your system with cabal:

```
cabal install --lib parsec
```

or provision an interactive interpreter with stack:

```
stack ghci --package parsec
```

Note that Haskell library management is highly complex. You will need parsec installed for the lab on Friday, so it's worth trying to complete this step ahead of time, if you can. Alternatively come to the lab on Friday to get technical support from our team of tutors.

You can verify that your parsec installation is working properly in ghci:

```
ghci> import Text.Parsec
ghci> parse anyChar "input" "a"
Right 'a'
```

3 My First Parser

This simplest of parsers recognizes (and accepts) `String` values containing the single character `'c'`.

```
firstParser = (char 'c') :: Parsec String st Char
```

The three type parameters for the `Parsec` type are, respectively:

- `String` - type of the input data stream to be parsed
- `st` - type for user state (unused in this trivial example)
- `Char` - type of the value recognized by this parser

Let's see how this parser behaves:

```
parseTest firstParser "c"
parseTest firstParser "cat"
parseTest firstParser "dog"
```

We notice that our `firstParser` will accept any `String` that begins with the `'c'` character.

There are several different ways to run a parser. Above we have shown the `parseTest` function, which is fine for interactive input. In larger programs, we might use the `parse` or `runParser` functions.

```
parse firstParser "foo" "hello"
runParser firstParser () "foo" "cat"
```

4 Detour: the `Either` datatype

Do you remember `Maybe`? Of course you do! It's our favourite example type so far. `Maybe` has a data constructor, i.e. `Just` and a default 'empty' value, i.e. `Nothing` which represents an error value. Sometimes, we want the error value to contain richer information, perhaps specifying some detail about the problem. This is where the `Either` datatype is useful. It has *two* data constructors: `Right` (for correct output) and `Left` (for error output). The definition of `Either` is:

```
data Either a b = Left a | Right b
```

We use `Either` in parsing, where `Left` values include information about the parse error and its context.

As with `Maybe`, you can work with `Either` values in a monadic style.

5 More Complex Parsing

We can recognize an entire `String`, as a sequence of characters.

```
dogParser = string "dog" :: Parsec String st String
```

This parser recognizes a string of characters (and returns the entire `String` value).

Now we can start combining simpler sub-parsers into more complex parsers using the *alternative* operator `<|>`.

```
catParser = string "cat"
petParser = dogParser <|> catParser
parseTest petParser "dog"
parseTest petParser "cat"
parseTest petParser "cog" --This fails, parse error
-- unexpected "o"
-- expecting "cat"
```

6 Parsers as Monads

Since Parser values are monads, we can sequence their operation using monadic `do` notation in Haskell. For example:

```
animalParser :: Parsec String st String
animalParser = do
  animal <- dogParser
  char '/'
  country <- count 2 anyChar
  let bark =
        case country of
          "UK" -> "woof"
          "FR" -> "waouh"
          "GR" -> "gav"
  return bark
```

7 Building a Parse Tree for Arithmetic Expressions

The main use for parsers is to process a sequence of tokens (lexemes) and build a parse tree data structure, representing abstract syntactic structure and relationships.

Let's consider a toy example. Suppose we want to process simple integer arithmetic expressions, along with bracketed sub-expressions. Sentences in our language would look like:

```
(1+2) * 3
```

We could define simple Haskell algebraic datatypes to encapsulate such expressions:

```
data BinOp = Add | Sub | Mul | Div deriving (Show,Eq)

data ArithExpr =
      Compound ArithExpr BinOp ArithExpr
    | Value Int
    deriving Show
```

and now we can construct a series of miniature parsers to process different input character sequences and construct the appropriate Haskell in-memory data structures to represent the abstract syntax.

```
numberParser:: Parsec String st ArithExpr
numberParser = do
  digs <- many1 digit
  let num = read digs
  return $ Value num
```

Note that `digit` is a built-in primitive matching any single digit character; `many1` matches one or more occurrences. The standard Prelude `read` function converts the `String` value to an `Integer`. We `return` an `ArithExpr` which is constructed as a `Value` value, storing the number we have parsed from the input.

```
operatorParser:: Parsec String st BinOp
operatorParser = do
  op <- (oneOf "+-*/")
  return (selectOp op)
    where selectOp '+' = Add
          selectOp '-' = Sub
          selectOp '*' = Mul
          selectOp '/' = Div
```

Here we want to parse a single operator character and construct the appropriate `BinOp` value. The `oneOf` parser will match one character from a list of characters.

```
expressionParser:: Parsec String st ArithExpr
expressionParser = (between (char '(') (char ')') binaryExpressionParser) <|>
                   numberParser

binaryExpressionParser:: Parsec String st ArithExpr
binaryExpressionParser = do
  e1 <- expressionParser
  op <- operatorParser
  e2 <- expressionParser
  return (Compound e1 op e2)
```

We can run this parser with code like this:

```
parseTest expressionParser "(1+1)"
-- or
parse expressionParser "error" "(1+(2*3))"
```

Once we have a parse tree style data structure, we can process this to evaluate the expression or perhaps to rewrite it in some way.

8 Alternative typeclass

Sometimes we want to try a computation, then if it fails, try something else. The `Alternative` typeclass helps us here ... in particular, the `<|>` operator, which intuitively means: 'try the left hand action or, if it fails, try the right hand action'.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

(Minor reassurance: don't worry about `Applicative` for now ... pretend it says `Monad` instead ... we will fill in the details later in the course.)

Monads and Monad Laws

1 Defining Custom Data Types

We can model a *student* with a name, address, and level in a few ways in Haskell.

- As a *tuple*

```
type Student = (String, String, Int)
```

- As a *data type*

```
data Student = Student String String Int
```

However neither of the above approaches is ideal. It is difficult to know which 'String' is a name and which is an address. We need to *pattern match* to deconstruct and get a relevant field. All fields must be specified when updating the data structure instance.

In Haskell, *records* allow us to have *named fields*.

```
data Student = MkStudent { name :: String, address :: String,
                           level :: Int }
```

We can construct a record as normal, or by explicitly specifying fields:

```
s =  MkStudent "jeremy" "glasgow" 1
s' = MkStudent { name = "simon", address = "glasgow", level = 6 }
```

We can project a record by using its field name, effectively the field name becomes a function:

```
-- implicitly ... name :: Student -> String
putStrLn(name s)
```

We can also *update* a record without needing to specify all fields.

```
let s2 = s { level = 2 }
```

A 'newtype' declaration is a special form of a data declaration where there is only one data constructor. It can be used like a *type alias*, while ensuring that types are treated separately in the type system.

```
newtype Variable = MkVariable String
```

We will encounter plenty of 'newtype's as we look at the various monad instances.

2 What is a Monad?

As we have already seen, 'Monad' is a type class in Haskell, which provides a systematic way of sequencing operations. Monads enable values to be put into 'contexts', enforced by the type system. We have considered contexts such as lists, 'Maybe's, 'IO', and 'Reader'/'Writer'. Effectively, 'Monad' is a design pattern — a recurring solution to a common problem in functional programming.

3 Laws

What is a law? It's a principle that governs or constrains things. In theoretical computing terms, laws constrain behaviours and relationships between entities.

The `map` function has the following laws

- identity law:

$$\texttt{map id} == \texttt{id}$$

- composition:

$$(\texttt{map f}).(\texttt{map g}) == \texttt{map (f.g)}$$

And we also looked at three laws for the 'Monoid' typeclass:

- left identity law:

$$\texttt{mempty <> x} == x$$

- right identity law:

$$\texttt{x <> mempty} == x$$

- associativity law:

$$(\texttt{x <> y}) <> z == x <> (y <> z)$$

Remember that we stressed the point that Haskell cannot enforce these laws in the language or type system directly. Instead, developers have to manually ensure the laws are respected by their code ... or we might use a theorem prover.

3.1 First Monad Law (Left Identity)

$$\texttt{(return x) >>= f} == \texttt{f x}$$

If we think about the types of 'return :: Monad m => a -> m a' and '(>>=) :: Monad m => m a -> (a -> m b) -> m b' and then fit in the types of 'x :: a' and 'f :: Monad m => a -> m b' then everything makes sense.

In natural language terms, we put 'x' into the monad, then bind strips it out of the monad, feeds it to function 'f' and the result 'f x' is evaluated.

3.2 Second Monad Law (Right Identity)

$$\texttt{(y >>= return)} == \texttt{y}$$

Here, 'y' is a monadic context containing a value 'x', and the bind operation feeds value 'x' to 'return', which *re-wraps* 'x' into the monadic context, to recover 'y'.

3.3 Third Monad Law (Associativity)

$$\texttt{((y >>= f) >>= g)} == \texttt{(y >>= (\x -> (f x >>= g)))}$$

On the left hand side, the first bind feeds 'y' to 'f' to get a monadic result, then the second bind feeds this interim result to 'g' to get the final result.

On the right hand side, we do the innermost bind in the brackets first ... so we bind the result of 'f x' to g, where 'x' is the parameter from the intermediate lambda abstraction we have to construct to get the types correct.

In natural language, the third monad law says that when we have a chain of monadic function applications with '>>=', it doesn't matter how they're nested.

4 Monad Laws for 'Maybe'

Let's consider the monad laws and the 'Maybe' monad. Suppose we defined 'return' for 'Maybe' as follows:

```
return :: a -> Maybe a
return _ = Nothing
```

then would the two identity laws hold? No, of course they wouldn't!

Instead, we define it as:

```
return :: a -> Maybe a
return x = Just x
```

Now both identity laws do hold.

Error Handling

1 What is an Error?

A *error* is a fault in the specification, implementation or operation of software, which causes an incorrect or unexpected result, leading to unanticipated behaviour.

Many of the errors we have encountered so far in our Haskell coding have been *static* errors, such as type errors. These are highlighted by the GHC compiler or interpreter when the code is parsed.

However sometimes errors occur when the program executes; we refer to these errors as *runtime exceptions*.

Examples might include:

- arithmetic exceptions, such as integer division by zero
- taking `head` or `tail` of an empty list
- I/O exceptions when dealing with files or network access
- incomplete pattern matching in a function evaluation

2 Immediate Termination

The simplest way to deal with a runtime error is to terminate the program immediately. This is achieved in Haskell by calling the function:

```
error :: String -> a
```

which has a polymorphic return type that matches any expression context. The `error` function takes a `String` parameter that specifies the human-readable error message which gets printed to the standard error stream.

When the program is interpreted in GHCi, the error is reported and a stack trace is provided.

This behaviour is equivalent to the `exit()` function in C or Python.

3 Indicate Error Conditions with Wrapper Types

Software engineering principles like *defensive programming* and *graceful failure* can be followed in Haskell programming. This involves actively anticipating problems and handling errors within the program logic.

We have already encountered how to indicate error values using wrapper types like `Maybe` or `Either`. The `Maybe` type uses `Nothing` to indicate the absence of a value, i.e. a safe null value, and `Just x` for a normal value `x`.

One drawback of using the `Maybe` data type to deal with errors is that there is a lack of error information in `Nothing` – we have no specific details about what caused the error.

`Either` is a richer type which supports error information:

```
data Either a b = Left a | Right b
```

Here, `Left a` indicates an error value, with the `a` value (often `String`) holding error information. `Right b` indicates a proper value, equivalent to `Just b` in the `Maybe` type. In the same way, `Left` is equivalent to `Nothing`

In the same way that `Maybe` is monadic, so it can take part in a bind call sequence or a `do` block – so is `Either`.

When we looked at parsing, we noted that parser error values are wrapped as `Left` values, when we invoke the parser with the `runParser` function.

```
runParser :: Parsec String st a -- our parser
             -> String      -- state (ignore)
             -> String      -- filename
             -> String      -- string to parse
             -> Either ParseError a  -- result

catParser = string "cat"
runParser animalParser "" "" "cat" --  result: Right "cat"
runParser animalParser "" "" "dog" -- result: Left ParseError
```

4 Runtime Exception Handling

Like in Java, there are different types of exceptions that occur for different sorts of problems. There are specific circumstances in which each type of exception can be raised.

There are plenty of exception handling facilities (actually functions) in the Haskell `Control.Exception` module. We look at two design patterns in particular:

1. the `catch` function, and
2. the `try` function.

4.1 Catching Exceptions

The `catch` function specifies what to do if an expression evaluation raises an error. We provide a handler with a type-compatible alternative expression to evaluate. Below are three examples:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a

-- example 1
(randomIO :: IO Bool) >>= \x -> (if x then
  putStrLn "hello" else error "oops!") `catch`
  (\(e ::ErrorCall) -> putStrLn "exception")

-- example 2
((readFile "foo.txt") >>= putStrLn) `catch` (\e
  -> (putStrLn $ "unable to open file: " ++ (show (e::IOException))))

-- example 3
getContentFrom :: URL -> IO (Maybe String)
getContentFrom url =
   catch (do { str <- scrapeURL url scrapeBody; return $ str })
         (\err -> do {putStrLn $ show (err::HttpException); return Nothing})
```

Note it is necessary to annotate the exception names with their types explicitly, to make type inference work properly.

4.2 Try Clauses for Exceptions

The `try` function embeds the result in an `Either e b` — where `Left e` is an Exception and `Right b` is a normal result. Subsequently, we can pattern-match on this Either value. Effectively this allows us to wrap runtime errors neatly as an `Either` result type.

```
try :: Exception e => IO a -> IO (Either e a)

do
   x <- try ((readFile "foo.txt") >>= putStrLn)
   case x of
     Left e -> putStrLn $ "you've got problems: " ++ (show (e::IOException))
     Right r -> return r
```

Note that unlike in Java, Python and other mainstream languages, exception handling is based purely on function calls rather than requiring built-in language keywords.

Monad Transformers

1 What is a Monad?

Recall from earlier lectures that a *monad* is a typeclass in Haskell, representing a computational structure or context. A monad combines computations into an ordered sequence with the *bind* operator (`>>=`). Monads enable containment of side-effects in the type system, i.e. the `IO` monad.

2 What is a Monad Transformer?

This is a technique for *composing* two monads into a single combined monad that has the joint behaviour of both individual monads. Monad transformers are a *monad composition technique*.

There is a Haskell typeclass `MonadTrans` with a characteristic function `lift`:

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

The `lift` function promotes a base monad function into the combined monad.

3 Motivating Example

Do you remember we had a pattern matching triangle, before we knew how to use the monadic bind operator (`>>=`)? We are going to see something similar here, when we try to use `Maybe` values in the context of the `IO` monad.

The library function we will use for this example is

```
findExecutable :: String -> IO (Maybe FilePath)
```

The `findExecutable` function searches through the system `PATH` to find an executable file matching the string parameter value.

```
import System.Directory

findAllExes = do
  a <- findExecutable "git"
  case a of
    Nothing -> return Nothing
    Just pathA -> do
      b <- findExecutable "ghc"
      case b of
        Nothing -> return Nothing
        Just pathB -> do
          c <- findExecutable "clang"
          case c of
            Nothing -> return Nothing
            Just pathC -> return $ Just (pathA, pathB, pathC)
```

The key problem is that we are trying to deal with `Maybe` values while we are with the `IO` monad. The return type of `findExecutable` is `IO (Maybe FilePath)`, and strictly speaking this isn't a monad in itself - it's a monad-within-a-monad.

4 Example with Transformers

Let's think about combining the `IO` monad and the `Maybe` monad. We will let `IO` be our *base* monad — this is generally the case in monad transformer stacks. So we use the `MaybeT` monad transformer. In this way, we provide the `IO` monad with the characteristics of the Maybe monad.

Here is the same example program as before, which looks for three executable files and returns a triple of their paths: (just much more concise)

```
import Control.Monad.Trans.Maybe
import System.Directory

findAllExes :: MaybeT IO (FilePath, FilePath, FilePath)
findAllExes = do
  a <- MaybeT $ findExecutable "git"
  b <- MaybeT $ findExecutable "ghc"
  c <- MaybeT $ findExecutable "clang"
  return (a, b, c)

runMaybeT findAllExes
```

`findExecutable` must be in the `IO` monad, because it is touching file storage, looking for exe files. The `Maybe` is necessary in case a particular exe file is not found in the `PATH`. If any exe is not found, we return `Nothing`, otherwise we return a `Just` triple value, which is simultaneously in the `Maybe` monad *and* in the `IO` monad.

The `MaybeT` constructor puts an `IO Maybe` value into the monad transformer, and the `runMaybeT` function extracts the `IO` value from the monad transformer.

Similarly, the `lift` function will `run` an IO action in the `MaybeT` context. See the example below for more details, where we use `getLine` and `putStrLn` inside the `MaybeT IO` monad.

Note the use of the `guard` monadic function here. If the condition fails, `Nothing` is returned. The `guard` call succeeds or fails, in the monadic context, based on the value of the input boolean.

```
import Control.Monad -- for guard
import Control.Monad.Trans -- lift
import Control.Monad.Trans.Maybe -- MaybeT

-- is there an ! in the string?
isValid :: String -> Bool
isValid v = '!' `elem` v

-- gets input string from user
maybeExcite :: MaybeT IO String
maybeExcite = do
  v <- lift getLine
  guard $ isValid v
  return v

doExcite :: MaybeT IO ()
doExcite = do
  lift $  putStrLn "say something exciting!"
  exciting <- maybeExcite
  lift $ putStrLn ("this is very exciting! " ++ exciting)
```

Note: Just like returns takes something to monadic context, `lift` takes something to monad transform context.

5 Revisiting a Lie

Do you remember when we looked at the `Reader`, `Writer` and `State` monads in an earlier lecture? I told you I was lying about the types ... well, now we understand the reason for this. Actually, `Reader` is a `ReaderT` monad transformer, with the `Identity` monad at the base of the transformer stack. Similarly, `Writer` is a `WriterT` with an `Identity`.

Functors & Applicatives

1 Back to map

Recall the `map` function. It was the first *higher-order* function we encountered. The `map` function takes a function `f` and maps it over a list:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

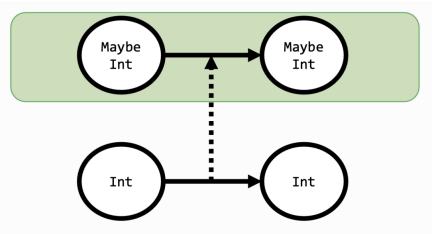
The `map` function preserves the *structure* of the list, but transforms each individual element in the list according to the `f` function.

So, we realise that the length of the mapped list will be the same as the length of the original input list, and each transformed element in the mapped list occupies the corresponding position as its untransformed element in the original list.

2 Mapping `Maybe` values

The `map` function allows us to operate on elements in a list context; it would be nice to do the same to values in other contexts. For instance, if we have an increment function `x->x+1`, how could we apply this function to a value `Just 41`?

Ideally, we would like to *lift* our increment function into the `Maybe` context. See the diagram below:



In fact, there is a function to do this, it's called `fmap`.

```
fmap (\x->x+1) (Just 41)
-- > evaluates to (Just 42)
```

Note the `fmap` can transform the type of the contained element, as well ...

```
fmap (length) (Just "Glasgow University")
-- > evaluates to (Just 18)
```

The reason why we can map over `Maybe` values is because `Maybe` is an instance of the `Functor` typeclass in Haskell.

3 What is a Functor?

The Functor typeclass is used for types that can be mapped over. A functor is a container or context (e.g., `Maybe`) that allows us to apply a function to its contents.

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

Conventionally, Haskell syntax also allows us to use an infix operator for `fmap`:

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

Here are some examples of `fmap` in action:

```
fmap (+1) (Just 3)
fmap (*2) [1,2,3]
fmap (fmap Data.Char.toUpper) getLine
```

all of which could be rewritten using the equivalent infix syntax:

```
(+1) <$> (Just 3)
(*2) <$> [1,2,3]
(Data.Char.toUpper <$>) <$> getLine
```

To summarise, an instance of a `Functor` is a `map`-able type.

4 Functor instances

We can make any container type into an instance of the `Functor` typeclass by providing a `fmap` definition. For example, the Haskell Prelude defines:

```
instance Functor Maybe where
  fmap f (Nothing) = Nothing
  fmap f (Just x) = Just (f x)
```

Consider our binary tree custom datatype used in the labs:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
 deriving (Show,Eq)
```

We can provide an `fmap` definition for `Tree` as follows:

```
instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node x t1 t2) =
    Node (f x) (fmap f t1) (fmap f t2)
```

5 Functor Laws

The Functor typeclass is used for types that can be mapped over. The `fmap` function modifies the *elements* in the structure, but preserves the structure itself.

To guarantee this property of functors, instances of the `Functor` typeclass are expected to follow these rules:

1. *identity*: `fmap id == id` 2. *composition*: `fmap (f.g) == (fmap f).(fmap g)`

Like the monad laws we saw previously, Haskell is unable to encode or check these laws directly. It is the responsibility of the developer (or a theorem prover) to check the laws hold for a `Functor` instance.

So, for example, here is an example of a *bad* functor that doesn't obey the laws:

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = (f x) : (f x) : (fmap f xs)
```

You can see that the structure of the list is altered - each element is duplicated. Hence both the identity law and the composition law will be broken by this definition of `fmap`.

6 Motivation for Applicative Functors

Suppose we have a function (like `(+1)`) wrapped up inside a context (like a `Maybe`). How do we apply this function to values inside other `Maybe` contexts? The answer is the `Applicative` typeclass, particularly the `<*>` or *apply over* function.

```
(<*>) :: Applicative f => f (a->b) -> f a -> f b
```

Look at the following code fragments:

```
(Just (+1)) <*> (Just 2)
-- > evaluates to (Just 3)

[(*3)] <*> [1..4]
-- > evaluates to [3,6,9,12]
```

7 Applicative Functors

The `Applicative` typeclass is defined as follows:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Here `pure` puts a value (often a function) into the applicative context and `(<*>)` applies a wrapped function to a wrapped value. Consider this example:

```
pure (+) <*> (Just 41) <*> (Just 1)
-- > evaluates to (Just 42)
```

Note that `pure` exactly corresponds with monadic `return` function.

All `Applicative` instances are also `Functor` instances. This is because `fmap` may be defined directly in terms of `pure` and `(<*>)`, i.e.

```
fmap f x = (pure f) <*> x
```

7.1 `Maybe` is an Applicative

The built-in definition of `Applicative` for `Maybe` is as follows:

```
instance Applicative Maybe where
  pure = Just
  (<*>) Nothing _ = Nothing
  (<*>) _ Nothing = Nothing
  (<*>) (Just f) (Just x) = Just (f x)
```

It's possible to 'lift' two-operand functions into Applicative structures, e.g. using the utility `liftA2` function. try:

```
import Control.Applicative
liftA2 (+) (Just 1) (Just 2)
liftA2 (+) [1,2,3] [4,5,6]
```

Note that `liftA2` can be defined in terms of pure and `<*>` —can you see how?

In summary, `Applicative` functors take the concept of regular functors one step further, since `Applicatives` enable function application to occur *within* the container context.

7.2 Applicative Laws

These are similar to the laws for functors and monads. The definitions of the applicative laws are beyond the scope of this course, and therefore non-examinable.

8 Relationship between Functors, Applicatives and Monads

Look at the characteristic functions for these three typeclasses:

```
(<$>) :: Functor x => (a->b) -> x a -> x b
(<*>) :: Applicative x => x (a->b) -> x a -> x b
(>>=) :: Monad x -> x a -> (a -> x b) -> x b
```

Can you observe the similarities between these three functions, in terms of how they apply functions on values in contexts? Admittedly, to make it neater, we should probably consider the *reverse bind* function `(=<)` which flips the order of the parameters:

```
(=<<) :: Monad x => (a -> x b) -> x a -> x b
```

The `fmap` function lifts a function into a context and applies it to elements in that context. The `<$>` function applies a function in a context to a value in the same context. The *bind* function applies a function that expects a value outside the context but returns a result in the context, to a value *already* in the context.

If you understand the above sentence, you are well on the way to Haskell enlightenment. If you don't understand it, think hard about why `(Just (Just (Just (Just "foo"))))` is not the same as `(Just "foo")`.

Lambda Calculus and Equational Reasoning

1 Referential Transparency

An expression is said to be referentially transparent if that expression can be replaced with its corresponding value without modifying the program behaviour. Expression value is always independent of expression context, i.e. think pure functions in python, java, haskell, etc. Expressions always evaluate the same way.

Take a look at this side effecting code in C

```
int f (int x){
    static int secret = 0; // hidden preserved state across calls
    secret++;
    return x+secret;
}

printf("%d\n", f(1) + f(1)); // prints 5
// compare with ...
int tmp = f(1);
printf("%d\n",tmp+tmp); // prints 4
```

This shows non-referential transparency. Instead, below is an example of referentially transparent code in haskell

```
let f x = x+1
let tmp = (f 1) in print $ tmp + tmp
-- cf.
print $ (f 1) + (f 1)
```

These evaluate in the same way. If we wanted to preserve state across calls to f, we would need to use the State monad, like such

```
import Control.Monad.State
import Control.Monad.IO.Class (liftIO)
f :: Int -> StateT Int IO () -- Hidden preserved state across calls
f x = do
    secret <- get
    let result = x+secret+1
    liftIO $ putStrLn (show result) -- liftIO to do IO action within state
    put (secret+1)
main :: IO()
main = do
    ((), state') <- runStateT (f 1) 0
    ((), state'') <- runStateT (f 1) state'
    return ()
```

Referential transparency is good for reasoning around programs, whether manually, automatically, informally, or formally. Its good for parallelism, there is no side effects or inter thread dependencies. Evaluation is simply term rewriting, evaluation of lambda terms.

2 Lambda Calculus

basic components of **variables** like x or y, **function abstraction** like $\lambda x.M$, and **function application** : M N

2.1 Rewriting Rules for Lambda Calculus

α conversion is bound-variable remaining

$$\lambda x.M \xrightarrow{\alpha} \lambda y.M[y/x]$$

y is a fresh variable name. Identity function is the same, no matter what name we give its parameters.

$$\lambda x.x \xrightarrow{\alpha} \lambda z.z$$

β reduction is function call evaluation

$$(\backslash x.M)A \xrightarrow{\beta} M[A/x]$$

parameter x is replaced by lambda term A. In this example, x is replaced by $\backslash z.z$ in $(\backslash y.yx)$:

$$(\lambda x.\lambda y.y\ x)(\lambda z.z) \xrightarrow{\beta} \lambda y.y(\lambda z.z)$$

2.2 Computability

Any computable function can be represented in lambda calculus. A church numeral is a function that takes 2 parameters: $\backslash f- > \backslash x- > ....$

$$0 : \backslash f \rightarrow \backslash x \rightarrow x$$
$$1 : \backslash f \rightarrow \backslash x \rightarrow fx$$
$$2 : \backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$$
$$n : \backslash f \rightarrow \backslash x \rightarrow f^n\ \ x \qquad \text{– n applications of f}$$

(iterate f x) !! n

To add church numerals, take two church numerals M and N and use M as the zero parameter for N, add = $\backslash\ M\ N\ f\ x\ \rightarrow N\ f\ (M\ f\ x)$ N applications off, applied to M applications of f, applied to x. This evaluates to: $f\ (\ f\ (\ f\ ...(\ f\ ...(f\ x)\ )\ )\ )$

```
unchurch:: (a->a) -> a -> a -> Int
unchurch n = n (+1) 0
```

2.3 Notes on Lambda Calculus

Its a theoretical representation, not actually used for computation in the real world. It shows the smallest programming language that we can build to represent any computable function. Such a programming language only needs function abstraction and function application.

Of course, although the programming language is trivially simple, the corresponding programs will be very verbose.

3 Equational Reasoning

Previously, we used QuickCheck to do property based testing, this was dynamic testing, using random input values to test specified invariant holds, for a fixed number of tests. Now, we want to perform static verification, and prove properties of functions generally, that hold for all inputs.

To give a simple example, take a function definition:

```
example :: Int -> Int -> Int -> Int
example x y z = x*(y+z)
```

We want to prove that example a b c == example a c b.

3.1 Proofs by induction

3.1.1 Structural Induction

Given a property p(x) where x is a list, prove p([]), i.e. property holds for empty list *(base case)*. Prove that p(xs) -> p(x:xs), i.e. if a property holds for list xs of length n, then we can show it holds for list (x:xs) of length n+1 *(inductive step)*. By structural induction, p holds for arbitrary lists.

Assume for list xs of length that length (xs ++ ys) = length xs + length ys. Now, given list (x:xs) of length n+1,

$$\begin{array}{cc} \text{length((x:xs) ++ ys)} = & \text{length(x:xs) + (length ys)} \\ \text{definition of length} & \text{definition of length} \\ 1 + \text{length (xs ++ ys)} = & \text{(1+length xs) + (length ys)} \\ \text{Inductive hypothesis} & \text{associativity of +} \\ \text{1+((length xs) + (length ys))} = & \text{1+((length xs) + (length ys)} \end{array}$$

Hence, we have proven this is true. Now to look at another example, we want to prove that for all list xs, length xs >= 0, that is, the length of arbitrary list xs is non negative.

Base case, xs is [], length is 0, 0 >=0 is true, so base case is proven.

Inductive step - Assume for all list xs, length xs >= 0, we need to prove this for lists x:xs (length(x:xs) >=0). Definition of length says that length(x:xs) == 1 + length(xs). Since we are assuming length xs >=0, 1+length xs>= 0 (laws of arithmetic), proving the inductive step.