

Functional Programming

M269 Extension Tutorial

Contents

1	Agenda	3
2	Adobe Connect	4
2.1	Student View	4
2.2	Settings	5
2.3	Student & Tutor Views	7
2.4	Sharing Screen & Applications	9
2.5	Ending a Meeting	9
2.6	Invite Attendees	9
2.7	Layouts	10
3	Haskell & GHC	10
3.1	GHCi Commands	12
4	Types & Type Classes	12
4.1	Expressions & Types	13
4.2	Type Classes	14
4.2.1	Introduction to Haskell Builtin Type Classes	14
4.2.2	Equality Class	15
4.2.3	Ordered Class	15
4.2.4	Enumeration Class	16
4.2.5	Bounded Class	16
4.2.6	Read and Show Classes	16
5	Function Definitions — Styles	17
6	Higher-order Functions	18
6.1	Map, Filter	19
6.2	List Comprehensions	19
	Activity 1 List Comprehension Exercises	20
6.3	Fold Family	25
7	User Defined Data Types	27
7.1	Algebraic Datatypes	27
7.1.1	Standard Haskell Types	28
8	Algebraic Data Type Exercises	28
8.1	Alg Q 1	28
8.2	Alg A 1	29
8.3	Alg Q 2	29
8.4	Alg A 2	30
8.5	Alg Q 3	30
8.6	Alg A 3	30
8.7	Alg Q 4	31

8.8 Alg A 4	31
8.9 Alg Q 5	33
8.10 Alg A 5	33
8.11 Alg Q 6	34
8.12 Alg A 6	34
8.13 Laws for return and bind	34
9 Tree Data Types	35
9.1 Binary Trees and Recursion Schemes	35
9.2 Binary Trees: Data Types and Examples	36
9.2.1 egBSTree	36
9.2.2 egBSTree1	37
9.2.3 egBSTree2	37
9.2.4 egBSTree3	38
9.3 Alternative Tree Types	38
10 Tree Data Type Exercises	39
10.1 Tree Q 1	39
10.2 Tree A 1	40
10.3 Tree Q 2	41
10.4 Tree A 2	41
10.5 Tree Q 3	41
10.6 Tree A 3	42
10.7 Tree Q 4	42
10.8 Tree A 4	42
10.9 Tree Q 5	42
10.10 Tree A 5	43
10.11 Tree Q 6	43
10.12 Tree A 6	44
11 Recursion Schemes	44
12 Interactive Programming	45
12.1 I/O The Problem	45
12.2 I/O Solution (1)	45
12.3 Monadic I/O	46
12.4 do Notation	48
12.5 Control Structures	48
12.6 Interaction Exercises S0809 Q1	49
12.7 Interaction Exercises S0809 A1	49
12.8 Monadic I/O Review	50
12.8.1 Monad Data Structure	50
13 Future Work	51
14 References	52
References	52

1 Agenda

- Welcome & Introductions
- Functional programming introduction
- Programming environment and notation
- Program construction with functions and expressions rather than commands and statements
- Functions are first-class citizens
- Higher order functions
- Powerful combining forms
- Function composition
- Lazy evaluation or non-strict semantics
- Strong polymorphic type system
- Recursion and recursion patterns
- Efficiency and pragmatic issues

Introductions — Me

- *Name* Phil Molyneux
- *Background* Physics and Maths, Operational Research, Computer Science
- *First programming languages* [Fortran](#), [BASIC](#), [Pascal](#)
- *Favourite Software*
 - [Haskell](#) — pure functional programming language
 - Text editors [TextMate](#), [Sublime Text](#) — previously [Emacs](#)
 - Word processing in [L^AT_EX](#)
 - [Mac OS X](#)
- *Learning style* — I read the manual before using the software (really)

Introductions — You

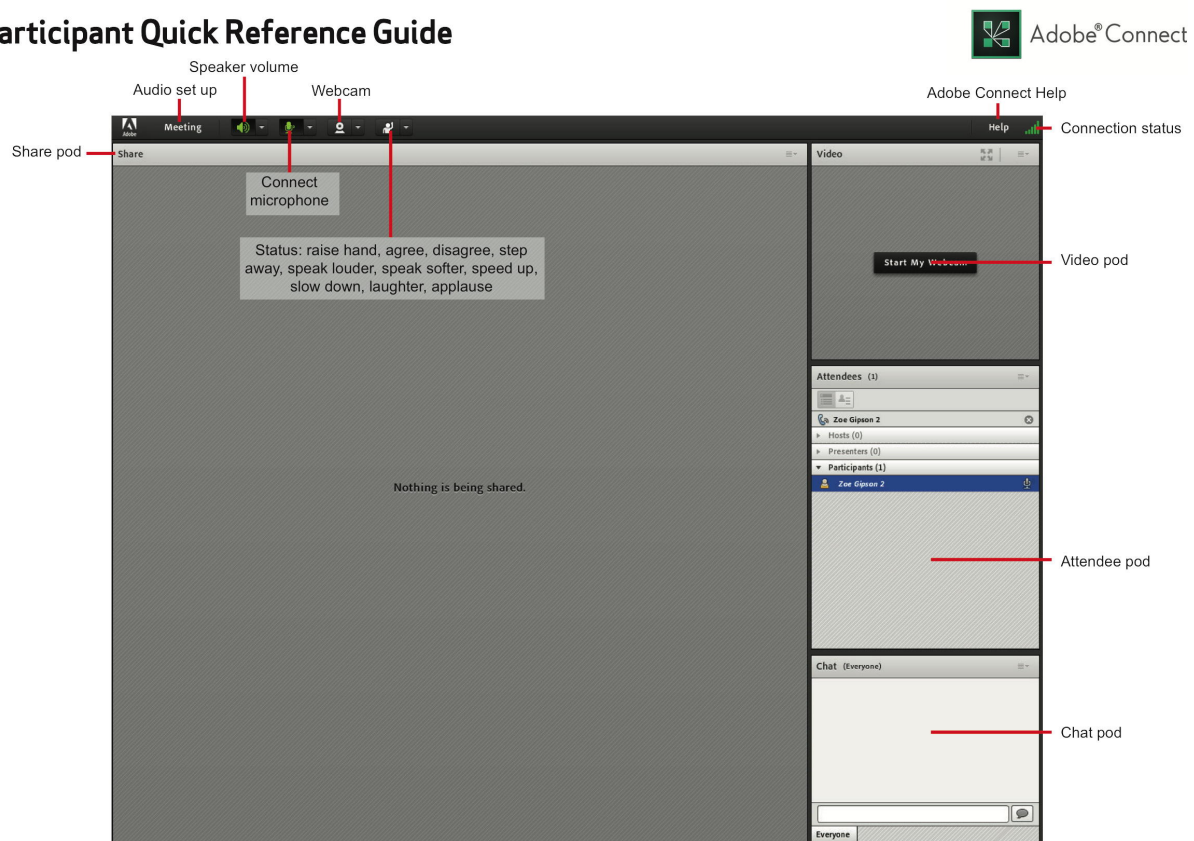
- *Name* ?
- *Position in M269* ? Which part of which Units and/or Reader have you read ?
- *Particular topics you want to look at* ?
- *Learning Syle* ?

2 Adobe Connect Interface and Settings

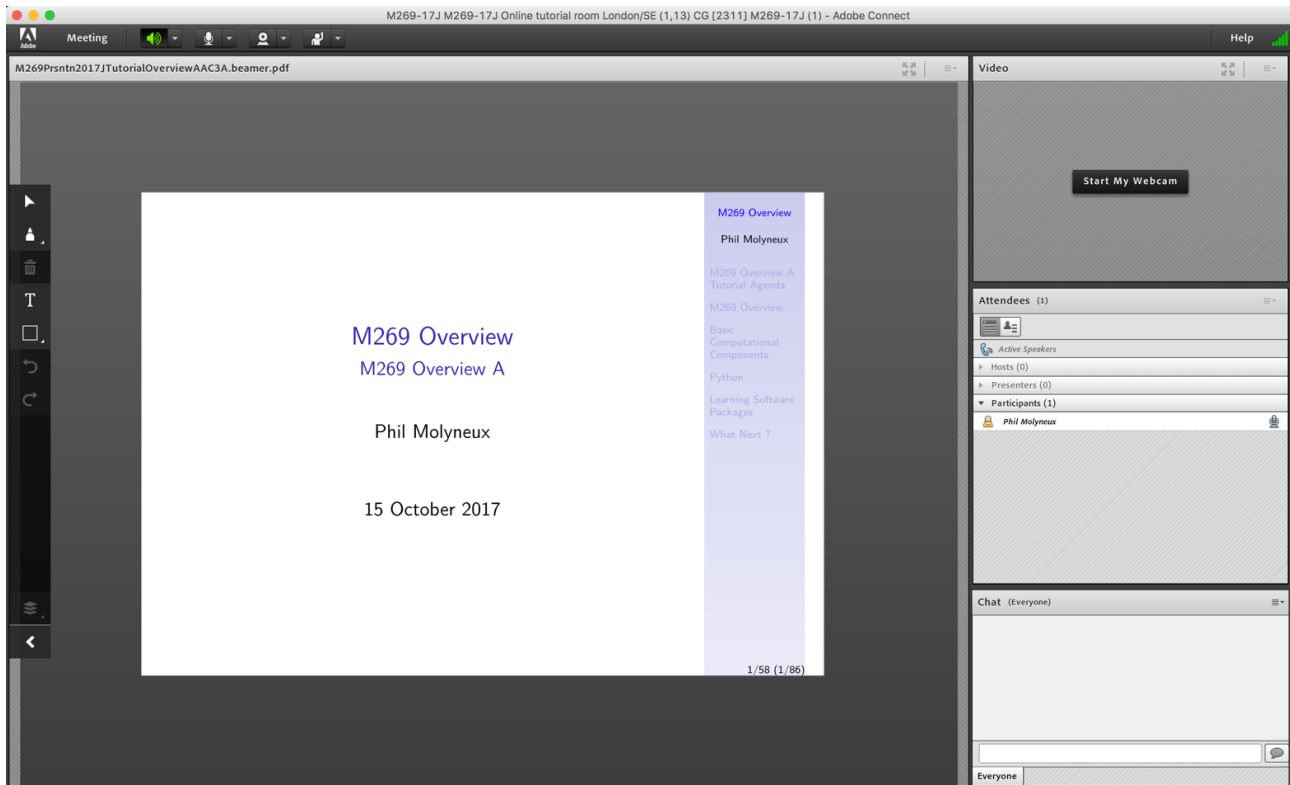
2.1 Adobe Connect Interface — Student View

Adobe Connect Interface — Student Quick Reference

Participant Quick Reference Guide



Adobe Connect Interface — Student View



2.2 Adobe Connect Settings

Adobe Connect Settings

- **Everybody: Audio Settings** Meeting > Audio Setup Wizard. . .
- **Audio** Menu bar > Audio > Microphone rights for Participants ✓
- Do not *Enable single speaker mode*
- **Drawing Tools** Share pod menu bar > Draw (1 slide/screen)
- Share pod menu bar > Menu icon > Enable Participants to draw ✓ gray
- Meeting > Preferences > Whiteboard > Enable Participants to draw ✓
- Cancel hand tool
- Do not enable green pointer. . .
- Meeting > Preferences > Attendees Pod > Disable *Raise Hand notification*
- **Cursor** Meeting > Preferences > General tab > Host Cursors > Show to all attendees ✓ (default *Off*)
- Meeting > Preferences > Screen Share > Cursor > Show Application Cursor
- **Webcam** Menu bar > Webcam > Enable Webcam for Participants ✓
- **Recording** Meeting > Record Meeting. . . ✓

Adobe Connect — Access

- **Tutor Access**

- TutorHome > M269 Website > Tutorials
- Cluster Tutorials > M269 Online tutorial room
- Tutor Groups > M269 Online tutor group room
- Module-wide Tutorials > M269 Online module-wide room

- **Attendance**

TutorHome > Students > View your tutorial timetables

- **Beamer Slide Scaling** 440% (422 x 563 mm)

- **Clear Everyone's Status**

Attendee Pod > Menu > Clear Everyone's Status

- **Grant Access**

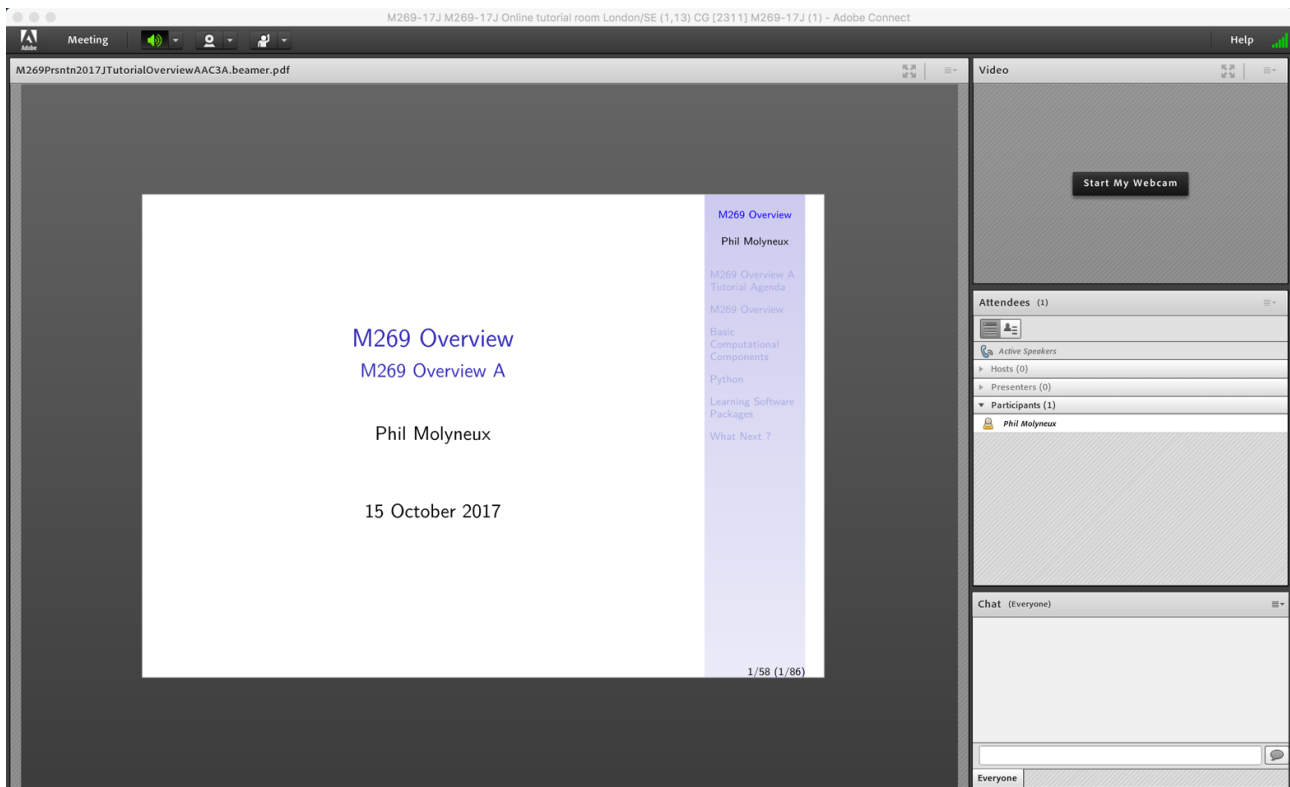
Meeting > Manage Access & Entry > Invite Participants... and send link via email

Adobe Connect — Keystroke Shortcuts

- [Keyboard shortcuts in Adobe Connect](#)
- **Toggle Mic**  + **M** (Mac), **Ctrl** + **M** (Win) (On/Disconnect)
- **Toggle Raise-Hand status**  + **E**
- **Close dialog box**  (Mac), **Esc** (Win)
- **End meeting**  + ****

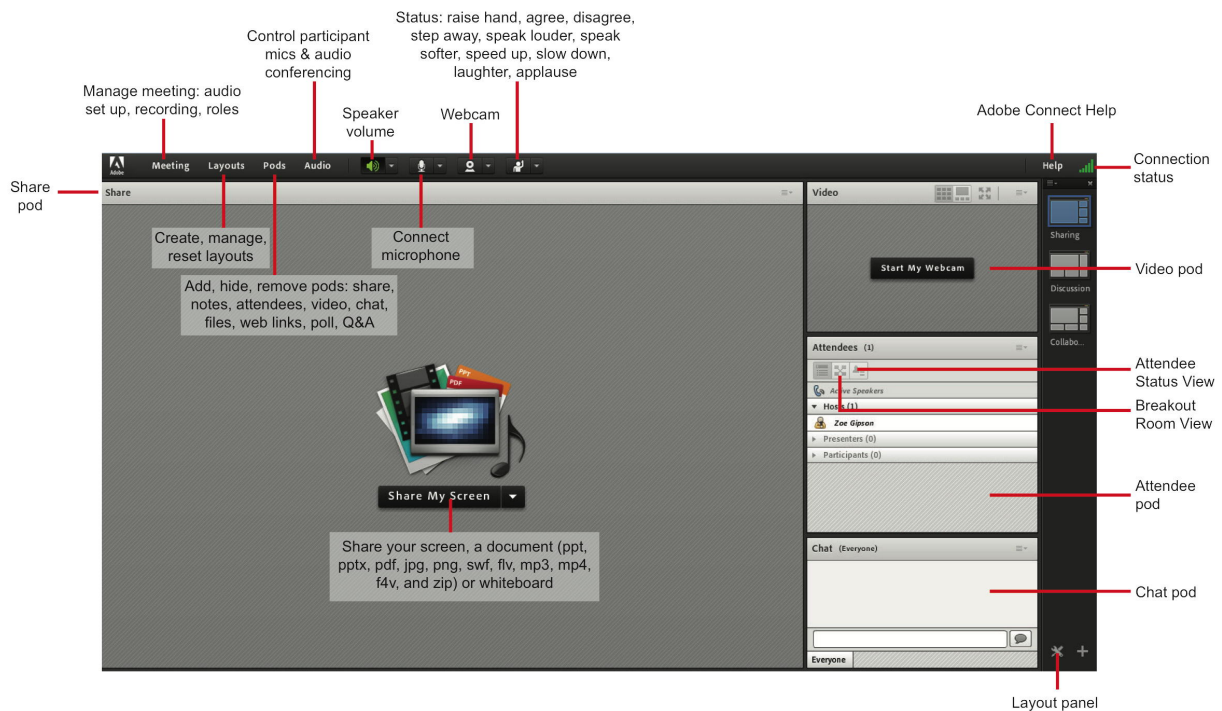
2.3 Adobe Connect Interface — Student & Tutor Views

Adobe Connect Interface — Student View (default)

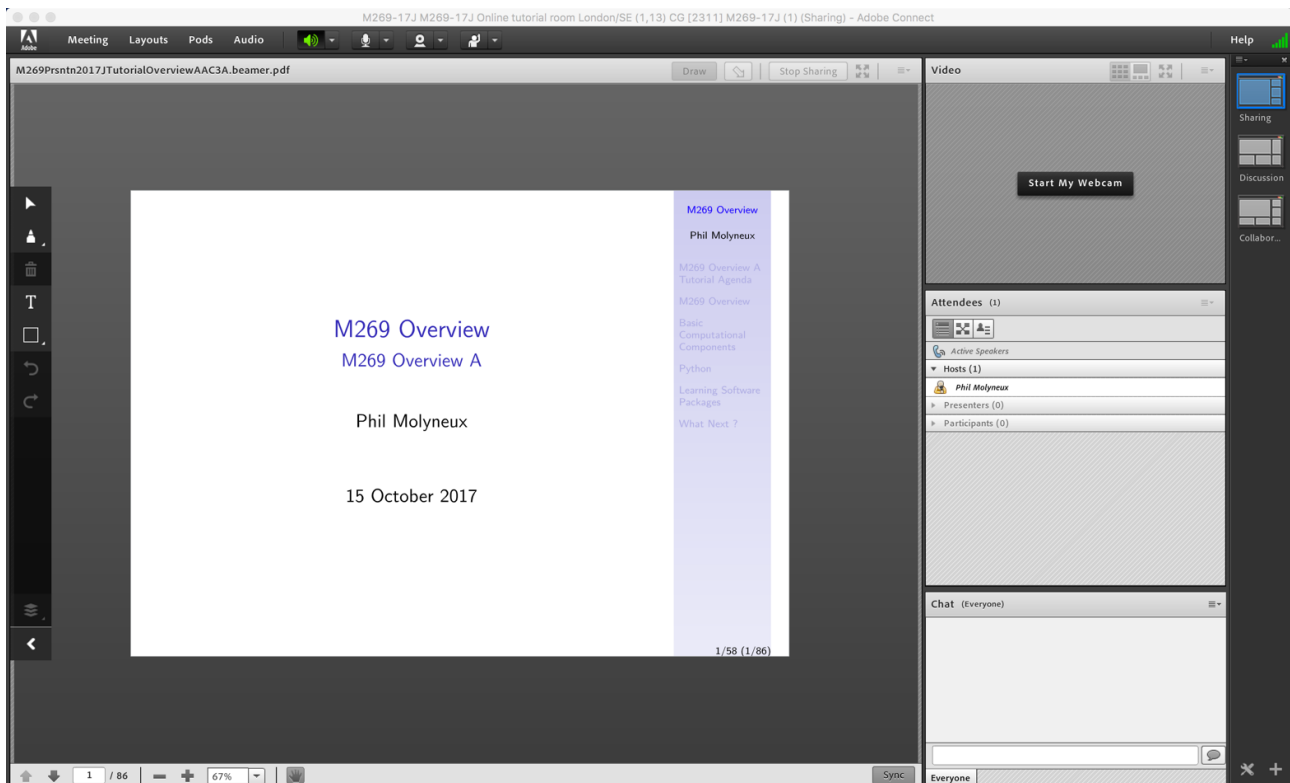


Adobe Connect Interface — Tutor Quick Reference

Host Quick Reference Guide



Adobe Connect Interface — Tutor View



2.4 Adobe Connect — Sharing Screen & Applications

- **Share My Screen** > **Application tab** > **Terminal** for **Terminal**
- **Share menu** > **Change View** > **Zoom in** for mismatch of screen size/resolution (Participants)
- (Presenter) Change to 75% and back to 100% (solves participants with smaller screen image overlap)
- Leave the application on the original display
- Beware blue hatched rectangles — from other (hidden) windows or contextual menus
- Presenter screen pointer affects viewer display — beware of moving the pointer away from the application
- First time: **System Preferences** > **Security & Privacy** > **Privacy** > **Accessibility**

2.5 Adobe Connect — Ending a Meeting

- *Notes for the tutor only*
- **Student:** **Meeting** > **Exit Adobe Connect**
- **Tutor:**
- **Recording** **Meeting** > **Stop Recording** ✓
- **Remove Participants** **Meeting** > **End Meeting. . .** ✓
 - Dialog box allows for message with default message:
 - *The host has ended this meeting. Thank you for attending.*
- **Recording availability** *In course Web site for joining the room, click on the eye icon in the list of recordings under your recording* — edit description and name
- **Meeting Information** **Meeting** > **Manage Meeting Information** — can access a range of information in Web page.
- **Attendance Report** see course Web site for joining room

2.6 Adobe Connect — Invite Attendees

- **Provide Meeting URL** **Menu** > **Meeting** > **Manage Access & Entry** > **Invite Participants. . .**
- **Allow Access without Dialog** **Menu** > **Meeting** > **Manage Meeting Information** provides new browser window with **Meeting Information** **Tab bar** > **Edit Information**
- Check *Anyone who has the URL for the meeting can enter the room*
- Default *Only registered users and accepted guests may enter the room*
- **Reverts to default next session but URL is fixed**
- Guests have blue icon top, registered participants have yellow icon top — same icon if URL is open

- See [Start, attend, and manage Adobe Connect meetings and sessions](#)

2.7 Layouts

- **Creating new layouts** example *Sharing* layout
- [Menu](#) > [Layouts](#) > [Create New Layout...](#) > [Create a New Layout dialog](#) > [Create a new blank layout](#) and name it *PMolyMain*
- New layout has no Pods but does have Layouts Bar open (see Layouts menu)
- **Pods**
- [Menu](#) > [Pods](#) > [Share](#) > [Add New Share](#) and resize/position — initial name is *Share n*
- **Rename Pod** [Menu](#) > [Pods](#) > [Manage Pods...](#) > [Manage Pods](#) > [Select](#) > [Rename](#) or [Double-click & rename](#)
- Add Video pod and resize/reposition
- Add Attendance pod and resize/reposition
- Add Chat pod — name it *PMolyChat* — and resize/reposition
- Dimensions of **Sharing** layout (on 27-inch iMac)
 - Width of Video, Attendees, Chat column 14 cm
 - Height of Video pod 9 cm
 - Height of Attendees pod 12 cm
 - Height of Chat pod 8 cm
- **Duplicating Layouts** does *not* give new instances of the Pods and is probably not a good idea (apart from local use to avoid delay in reloading Pods)

[Go to Table of Contents](#)

3 Haskell & GHC

- You can do *functional programming* in any language
- To really see some of the ideas it is best to use a language that directly implements these ideas
- These notes use [Haskell](#) and the implementation [GHC](#)
- We first set this file up as a [Literate Haskell Script](#) (this page explains roughly how I do my notes)

```
1 module M269TutorialExtension2019J where
2 import Data.List
```

- A Haskell script starts with a module header which starts with the reserved identifier, `module` followed by the module name, `M269TutorialExtension2019J`
- The module name must start with an upper case letter and is the same as the file name (without its extension of `.lhs`)

- Haskell uses *layout* (or the *off-side rule*) to determine scope of definitions, similar to Python
- The body of the module follows the reserved identifier where and starts with `import` declarations
- These import the built-in libraries
- We use the `sort` function from `Data.List`
- The Haskell standard library, `Prelude`, is always present
- We start the GHC REPL (Read-eval-print loop) from a command line with `ghci`

```
GHCi> :l M269TutorialExtension2019J
[1 of 1] Compiling -- stuff removed
Ok, one module loaded.
GHCi>
```

- At the `GHCi` prompt we can evaluate expressions with any builtin functions or in our script

```
GHCi> 6 * 7
42
GHCi> length [9,16,25]
3
GHCi>
```

- `length` is defined in the standard `Prelude` library
- It returns the *size* of its argument — in this case the length of the list `[9,16,25]`
- Notice the quiet notation for function application
- Function application is denoted by juxtaposition and is more binding than (almost) anything else (remember BODMAS ?)
- `f x` not `f(x)`
- We can define values at the `GHC` prompt

```
GHCi> let add x y = x + y
GHCi> add 2 3
5
```

- Function application is left associative
- So `add 2 3` means `(add 2) 3`
- What could `add x` mean ?
- And what is the *type* of `add` ? You said this language is strongly typed — where is the type specification (as in Java, but not Python, which is weakly typed)
- `GHC` can infer the most general type of a variable in the Haskell type system

```
GHCi> :type add
add :: Num a => a -> a -> a
```

- This means `add` takes two arguments of type `a` as long as that type is some sort of number, and it returns a number of the same type `a`
- `Num` is the *Type Class* of all the type that implement the behaviour of numbers
- This is similar to interfaces and generics in Java

- The type class `Num` is defined in the `Prelude` and includes the usual integers and floating point numbers and also arbitrary precision integers and rational numbers
- What is the meaning of `add x`?

```
GHCi> :type (add 2)
(add 2) :: Num a => a -> a
```

- This means `add 2` is a function which takes a number and adds 2 to the number
- `add x y` means `(add x) y` — function application is *left associative*
- The type `(a -> a -> a)` means `a -> (a -> a)`
- The function type arrow `(->)` associates to the right to be consistent with the left associativity of function application
- This means we get a notation for higher-order functions and partial application for free (no need for a special notation)

3.1 GHCi Commands

- `:?` display list of commands
- **GHC Manual** [GHC User Guide](#)
- `:load`, `:l` load module(s)
- `:reload`, `:r` reload current module set
- `:type`, `:t` show the type of an expression
- `:info`, `:i` display information about the given names
- `<statement>` evaluate/run `<statement>`
- `:set editor <cmd>` set the command used for `:edit`
- `:set +m` allow multilevel commands — see [Multiline input](#)
- `:set +s` print timing/memory stats after each evaluation
- See also the [.ghci](#) and [.haskeline](#) files

4 Types & Type Classes

- **Types** are collections of related values
- Common primitive and built-in data types include characters, numbers, Booleans, lists, strings, tuples and function types
- **Type systems** are syntactic methods for assigning a type to each expression in the programming language — the aim is to prove the absence of certain program behaviours by answering the following:
- **Type checking** — given a **type signature** for an expression `expr :: t`, is `expr` an instance of type `t`?

- **Type inference** — given an expression `expr` what is its most general type ?
- Given a type `t`, is there any expression for it or does the type have no values ? This is related to the [Curry-Howard isomorphism](#)

4.1 Expressions & Types

- A **type** is a collection of related values and operations

```
GHCi> :t (2 == 3)
(2 == 3) :: Bool
GHCi> (2 == 3)
False
```

- **Basic types**
- **Booleans** type name `Bool` values `False`, `True`
- **Characters** type name `Char` values `'a'`, Unicode plus ways of escaping special characters such as new line
- **Strings** type name `String` values `"Hello"` strings are actually syntactic sugar for `[Char]`
- **Numbers** the usual `Int`, `Float`, `Double` but also arbitrary precision `Integer`, `Ratio` and also `Complex`
- Lists are sequences of elements of the same type

```
GHCi> :t [True,False, not (1 == 3)]
[True,False, not (1 == 3)] :: [Bool] -- no evaluation is done
GHCi> length []
-- [] is an empty list
0
GHCi> 5 : [3,4]
-- (:) list constructor
[5,3,4]
GHCi> :t (:)
(:) :: a -> [a] -> [a]
GHCi> head [5,3,4]
5
GHCi> tail [5,3,4]
[3,4]
GHCi> ["Athos","Porthos"] ++ ["Aramis","d'Artigan"]
["Athos","Porthos","Aramis","d'Artigan"] -- (++) appends two lists
GHCi> [5,3,4] !! 2
-- (!!) indexes from 0
4
GHCi> take 2 [5,3,4]
[5,3]
GHCi> drop 2 [5,3,4]
[4]
```

- **List Comprehensions** provide a concise way of performing calculations over lists
- Example: Square the even numbers between 0 and 9

```
GHCi> [x^2 | x <- [0..9], x `mod` 2 == 0]
[0,4,16,36,64]
GHCi>
```

- In general

```
[expr | qual1, qual2, ..., qualN]
```

- The qualifiers `qual` can be
 - Generators `pattern <- list`

- Boolean guards — acting as filters
- Local declarations with `let decls` for use in `expr` and later generators and boolean guards
- Note `'mod'` is a function made into an infix operator
- **Arithmetic sequences** provide a concise way of generating a list of values from an enumerable type

```
GHCi> [1..10]
[1,2,3,4,5,6,7,8,9,10]
GHCi> [1,3..10]
[1,3,5,7,9]
```

- We can also denote an infinite list (as long as we only consume a finite part) — lazy evaluation gives us this but it is special in Python

```
GHCi> take 10 (drop 10 [100..])
[110,111,112,113,114,115,116,117,118,119]
```

- And it works with any enumerable type

```
GHCi> ['A', 'D' .. 'Z']
"ADGJMPSVY"
```

- Strings are just syntactic sugar for list of characters `[Char]`

4.2 Type Classes

- **Types** specify sets of elements or **data constructors**
- Primitive: Numbers, characters
- Builtin: Booleans, Lists, Tuples, **Maybe** for failure or success, **Either** for error or correct value, the Unit type, a type with only one (non-bottom) value, when you have to have a type but don't want to do anything with it — see [What is \(\) in Haskell](#)
- User defined types: algebraic data type (naming the type constructor and data constructors — see LYAH chp 7), type synonyms, Datatype renamings
- Bottom, $\perp$ or **undefined** is the value of a program that crashes or loops forever
- **Type Classes** provide a structured way of *overloading*
- For example, `(+)` works with **Int**, **Integer**, **Float** and other types of numbers
- **Type Classes** are specified by *behaviour*
- For a type to be a member of a type class, we have to provide an implementation of some functions

4.2.1 Introduction to Haskell Builtin Type Classes

- **Eq** for equality — all basic data types are instances except functions and IO
- **Ord** for ordering — for types that have a total ordering
- **Enum** for enumeration — defining operations on sequentially ordered types

- [Bounded](#) to name the upper and lower limits of a type
- Numbers have a family of classes
- [Show](#) and [Read](#) for printable and readable types
- Further type classes express types which capture common patterns of computation — see LYAH chp 7 ([Functor](#)), chp 11 ([Applicative](#)), chp 12 ([Monoid](#), [Foldable](#)), chp 13 ([Monad](#)), and [Traversable](#)
- See [Functors, Applicatives, And Monads In Pictures](#) and [Typeclassopedia](#) for good introductions to these

4.2.2 Equality Class

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

  -- Minimal complete definition
  -- (==) or (/=)

  x /= y  = not (x == y)
  x == y  = not (x /= y)
```

4.2.3 Ordered Class

```
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<),(<=),(>=),(>) :: a -> a -> Bool
  max, min :: a -> a -> a

  -- Minimal complete definition
  -- (<=) or compare

  compare x y
  | x == y    = EQ
  | x <= y    = LT
  | otherwise = GT

  x <= y  = compare x y /= GT
  x < y   = compare x y == LT
  x >= y  = compare x y /= LT
  x > y   = compare x y == GT

  -- data Ordering = LT | EQ | GT
  -- deriving (Eq,Ord,Enum,Read,Show,Bounded)
```

```
max x y
| x <= y    = y
| otherwise = x

min x y
| x <= y    = x
| otherwise = y

-- note (min x y, max x y) = (x,y) or (y,x)

-- data Ordering = LT | EQ | GT
-- deriving (Eq,Ord,Enum,Read,Show,Bounded)
```

- Note that the [Ordering](#) algebraic data type is defined elsewhere in the Haskell Prelude and is not part of the [Ord](#) type class declaration

4.2.4 Enumeration Class

```
class Enum a where
  succ, pred      :: a -> a
  toEnum          :: Int -> a
  fromEnum        :: a -> Int
  enumFrom        :: a -> [a]           -- [n..]
  enumFromThen    :: a -> a -> [a]     -- [n,p..]
  enumFromTo      :: a -> a -> [a]     -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a] -- [n,p..m]

-- Minimal complete definition
-- toEnum, fromEnum
```

- Class `Enum` defines operations on sequentially ordered types

```
succ      = toEnum . (+1) . fromEnum
pred      = toEnum . (subtract 1) . fromEnum
enumFrom n = map toEnum [fromEnum n ..]
enumFromThen n p
  = map toEnum [fromEnum n, fromEnum p ..]
enumFromTo n m
  = map toEnum [fromEnum n .. fromEnum m]
enumFromThenTo n p m
  = map toEnum [fromEnum n, fromEnum p .. fromEnum m]
```

```
GHCi> enumFromThenTo 'a' 'c' 'z'
"acegikmoqsuwy"
GHCi> ['a','c' .. 'z']
"acegikmoqsuwy"
```

- Note that the spaces either side of `..` are sometimes required (to avoid misidentifying a qualified name)

4.2.5 Bounded Class

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

```
GHCi> minBound :: Bool
False
GHCi> maxBound :: Bool
True
GHCi> minBound :: Int
-9223372036854775808
GHCi> 2^63
9223372036854775808
GHCi> maxBound :: Int
9223372036854775807
GHCi> minBound :: Word
0
GHCi> maxBound :: Word
18446744073709551615
GHCi> 2^64 - 1
18446744073709551615
```

4.2.6 Read and Show Classes

```
class Read a where

class Show a where
```



```

GHCi> :t read
read :: Read a => String -> a
GHCi> :t show
show :: Show a => a -> String
GHCi> read "True" :: Bool
True
GHCi> read "321" :: Int
321
GHCi> read "Just_True" :: Maybe Bool
Just True
GHCi> read "(Nothing,321)" :: (Maybe Bool, Int)
(Nothing,321)
GHCi> show (Just True)
"Just_True"
GHCi> show "True"
 "\"True\""

```

5 Function Definitions — Styles

- [Declaration vs. expression style](#)
- **Declaration style:** you formulate an algorithm in terms of several equations that shall be satisfied
- **Expression style:** you compose big expressions from small expressions.
- **Declaration style:**
- Function arguments on left hand side

```

4   treble01 x = 3 * x
6   square01 x = x * x

```

- Pattern matching in function definitions

```

7   length01 [] = 0
8   length01 (x : xs) = 1 + length01 xs

```

- Guards on function definitions

```

9   length02 xs
10  | null xs = 0
11  | otherwise = 1 + length02 (tail xs)

```

- [where](#) clause
- **Expression style:**
- [Function composition \(.\)](#)

```

12  trebleThenSquare x = (square01 . treble01) x
14  squareThenTreble   = treble01 . square01

```

- Where did the argument go ? [Pointfree style](#) — can confuse beginners
- Do evaluations of:

```

16  test01 = trebleThenSquare 2
18  test02 = squareThenTreble 2

```

- **if** expression

```

20 length03 xs
21   = if null xs
22     then 0
23     else 1 + length03 (tail xs)

```

- **Expression style:**

- **Lambda abstraction**

```

25 square02 = \x -> x * x

```

- **case** expression

```

27 length04 xs = case xs of
28   [] -> 0
29   (y : ys) -> 1 + length04 ys

```

- **let** expression

- **Let expression**

```

let
  dec11
  dec12
  ...
  dec1N
in
  expr

```

- **Where clause** — declarations local to the right hand side of a function definition (also used in top level **class** and **instance** declarations)
- See example usage (and misuse) in [M269 Graph Algorithms tutorial notes](#)
- See [Let vs Where](#)
- To evaluate a function applied to actual arguments, substitute the actual arguments into the body of the definition of the function where the corresponding formal arguments occur

```

length01 []      = 0          -- (A)
length01 (x : xs) = 1 + length01 xs -- (B)

```

- Evaluate `length01 [6,8,3]`

```

length01 [6,8,3]
-> 1 + length01 [8,3]      -- by (B)
-> 1 + (1 + length [3])    -- by (B)
-> 1 + (1 + (1 + length [])) -- by (B)
-> 1 + (1 + (1 + 0))        -- by (A)
-> 3                        -- by arithmetic

```

6 Higher-order Functions

- Instead of special syntactic constructs such as **for**, **while** we capture common patterns with higher-order functions
- *Higher order functions* are functions that can take functions as arguments and/or return functions as results

- In functional programming, functions are first class citizens — they can be treated as data
- You just can't print a function or compare functions for equality
- This section looks at the most commonly used higher order functions
- `map`, `filter`, function composition (`.`), function application (`$`) and the `fold` family

6.1 Map, Filter

- `map` takes a function and a list and applies the function to every element of the list
- `map` can be defined with recursion: (name change to avoid Prelude clash)

```
31 map01 :: (a -> b) -> [a] -> [b]
32 map01 f [] = []
33 map01 f (x:xs) = f x : map01 f xs
```

- `map` can also be defined with a list comprehension:

```
35 map02 :: (a -> b) -> [a] -> [b]
36 map02 f xs = [f x | x <- xs]
```

- `filter` takes a predicate (a function that returns a Boolean) and a list and returns all the elements that satisfy the predicate
- `filter` can be defined with recursion: (name change to avoid Prelude clash)

```
38 filter01 :: (a -> Bool) -> [a] -> [a]
39 filter01 p [] = []
40 filter01 p (x:xs)
41   = if p x
42     then x : filter01 p xs
43     else filter01 p xs
```

- `filter` can also be defined with a list comprehension:

```
44 filter02 :: (a -> Bool) -> [a] -> [a]
45 filter02 p xs = [x | x <- xs, p x]
```

6.2 List Comprehensions

List Comprehensions — Python

- **List Comprehensions** provide a concise way of performing calculations over lists (or other iterables)
- Example: Square the even numbers between 0 and 9

```
Python3>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
Python3>>> [(x,y) for x in range(4)
...           for y in range(4)
...           if x % 2 == 0
...           and y % 3 == 0]
[(0, 0), (0, 3), (2, 0), (2, 3)]
Python3>>>
```

- In general

```
[expr for target1 in iterable1 if cond1
  for target2 in iterable2 if cond2 ...
  for targetN in iterableN if condN ]
```

- Lots example usage in the algorithms below

List Comprehensions — Python

- **List Comprehensions** provide a concise way of performing calculations over lists
- Example: Square the even numbers between 0 and 9

```
GHCi> [x^2 | x <- [0..9], x `mod` 2 == 0]
[0,4,16,36,64]
GHCi>
```

- In general

```
[expr | qual1, qual2, ..., qualN]
```

- The qualifiers `qual` can be
 - Generators `pattern <- list`
 - Boolean guards — acting as filters
 - Local declarations with `let decls` for use in `expr` and later generators and boolean guards

Activity 1 (a) Stop Words Filter

- **Stop words** are the most common words that most search engines avoid: 'a', 'an', 'the', 'th
- Using list comprehensions, write a function `filterStopWords` that takes a list of words and filters out the stop words
- Here is the initial code

```
11 sentence \
12   = "the_quick_brown_fox_jumps_over_the_lazy_dog"
14 words = sentence.split()
16 wordsTest \
17   = (words == ['the', 'quick', 'brown'
18               , 'fox', 'jumps', 'over'
19               , 'the', 'lazy', 'dog'])
21 stopWords \
22   = ['a', 'an', 'the', 'that']
```

[Go to Answer](#)

Activity 1 (a) Stop Words Filter

```
11 sentence \
12   = "the_quick_brown_fox_jumps_over_the_lazy_dog"
14 words = sentence.split()
16 wordsTest \
17   = (words == ['the', 'quick', 'brown'
18               , 'fox', 'jumps', 'over'
19               , 'the', 'lazy', 'dog'])
```

```

21 stopWords \
22 = ['a', 'an', 'the', 'that']

```

- Notice the [Python Explicit line joining](#) with (`\<n1>`) and [Python Implicit line joining](#) with (`(...)`)
- The [backslash](#) (`\`) must be followed by an [end of line character](#) (`<n1>`)
- The (`' '`) symbol represents a space (see Unicode U+2423 Open Box)

[Go to Answer](#)

Activity 1 (b) Transpose Matrix

- A matrix can be represented as a list of rows of numbers
- We *transpose* a matrix by swapping columns and rows
- Here is an example

```

38 matrixA \
39 = [[1, 2, 3, 4]
40    , [5, 6, 7, 8]
41    , [9, 10, 11, 12]]
42
43 matATr \
44 = [[1, 5, 9]
45    , [2, 6, 10]
46    , [3, 7, 11]
47    , [4, 8, 12]]

```

- Using list comprehensions, write a function `transMat`, to transpose a matrix

[Go to Answer](#)

Activity 1 (c) List Pairs in Fair Order

- Write a function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- If we do this in the simplest way we get a bias to one argument
- Here is an example of a bias to the second argument

```

68 yBiasLstTest \
69 = (yBiasListing(5,5)
70    == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
71        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
72        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
73        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
74        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Answer](#)

Activity 1 (c) List Pairs in Fair Order

- Rewrite the function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument
- Here is an example output

```

81 fairLstTest \
82   = (fairListing(5,5)
83     == [(0, 0)
84         , (0, 1), (1, 0)
85         , (0, 2), (1, 1), (2, 0)
86         , (0, 3), (1, 2), (2, 1), (3, 0)
87         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```

[Go to Answer](#)

Activity 1 (c) List Pairs in Fair Order

- Rewrite the function which takes a pair of positive integers and outputs a list of lists of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument — further, the output should be segment by each initial prefix (see example below)
- Here is an example output

```

94 fairLstATest \
95   = (fairListingA(5,5)
96     == [[(0, 0)]
97         , [(0, 1), (1, 0)]
98         , [(0, 2), (1, 1), (2, 0)]
99         , [(0, 3), (1, 2), (2, 1), (3, 0)]
100        , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])

```

[Go to Answer](#)

Answer 1 (a) Stop Words Filter

- Answer 1 (a) Stop Words Filter
- *Write here:*

Answer 1 (a) Stop Words Filter

- Answer 1 (a) Stop Words Filter

```

24 def filterStopWords(words) :
25   nonStopWords \
26     = [word for word in words
27         if word not in stopWords]
28   return nonStopWords

31 filterStopWordsTest \
32   = filterStopWords(words) \
33     == ['quick', 'brown', 'fox'
34         , 'jumps', 'over', 'lazy', 'dog']

```

[Go to Activity](#)

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix
- *Write here:*

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix

```

49 def transMat(mat) :
50     rowLen = len(mat[0])
51     matTr \
52     = [[row[i] for row in mat] for i in range(rowLen)]
53     return matTr
54
55 transMatTestA \
56 = (transMat(matrixA)
57    == matATr)

```

- Note that a list comprehension is a valid expression as a target expression in a list comprehension
- The code assumes every row is of the same length

[Go to Activity](#)

Answer 1 (b) Transpose Matrix

- Note the differences in the list comprehensions below

```

38 matrixA \
39 = [[1, 2, 3, 4]
40    , [5, 6, 7, 8]
41    , [9, 10, 11, 12]]

```

```

Python3>>> [[row[i] for row in matrixA]
...           for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
Python3>>> [row[i] for row in matrixA
...          for i in range(4)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Python3>>> [row[i] for i in range(4)
...          for row in matrixA]
[1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12]
Python3>>> [[row[i] for i in range(4)]
...           for row in matrixA]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

```

[Go to Activity](#)

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix
- The Python [NumPy](#) package provides functions for N-dimensional array objects
- For transpose see [numpy.ndarray.transpose](#)

```

Python3>>> import numpy as np
Python3>>> ar = np.array([[1,2],[3,4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> arT = ar.transpose()
Python3>>> arT
array([[1, 3],
       [2, 4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> ar.shape
(2, 2)

```

[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order — first version

- Write here

```

69 yBiasLstTest \
70   = (yBiasListing(5,5)
71     == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
72        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
73        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
74        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
75        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order
- This is the *obvious* but biased version

```

63 def yBiasListing(xRng,yRng) :
64   yBiasLst \
65     = [(x,y) for x in range(xRng)
66          for y in range(yRng)]
67   return yBiasLst
68
69 yBiasLstTest \
70   = (yBiasListing(5,5)
71     == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
72        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
73        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
74        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
75        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])

```

[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order — second version
- Write here

```

83 fairLstTest \
84   = (fairListing(5,5)
85     == [(0, 0)
86        , (0, 1), (1, 0)
87        , (0, 2), (1, 1), (2, 0)
88        , (0, 3), (1, 2), (2, 1), (3, 0)
89        , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```

[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order — second version
- This works by making the sum of the coordinates the same for each prefix

```

77 def fairListing(xRng,yRng) :
78   fairLst \
79     = [(x,d-x) for d in range(yRng)
80          for x in range(d+1)]
81   return fairLst
82
83 fairLstTest \
84   = (fairListing(5,5)
85     == [(0, 0)
86        , (0, 1), (1, 0)
87        , (0, 2), (1, 1), (2, 0)
88        , (0, 3), (1, 2), (2, 1), (3, 0)
89        , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])

```


[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order — third version
- Write here

```

97 fairLstATest \
98   = (fairListingA(5,5)
99     == [[(0, 0)]
100        , [(0, 1), (1, 0)]
101        , [(0, 2), (1, 1), (2, 0)]
102        , [(0, 3), (1, 2), (2, 1), (3, 0)]
103        , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])

```

[Go to Activity](#)

Answer 1 (c) List Pairs in Fair Order

- Answer 1 (c) List Pairs in Fair Order — third version
- The *inner loop* is placed into its own list comprehension

```

91 def fairListingA(xRng,yRng) :
92   fairLstA \
93     = [[(x,d-x) for x in range(d+1)]
94        for d in range(yRng)]
95   return fairLstA
97
98 fairLstATest \
99   = (fairListingA(5,5)
100     == [[(0, 0)]
101        , [(0, 1), (1, 0)]
102        , [(0, 2), (1, 1), (2, 0)]
103        , [(0, 3), (1, 2), (2, 1), (3, 0)]
104        , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])

```

[Go to Activity](#)

6.3 Fold Family

- `foldr` captures a common pattern of combining elements of a list
- Consider `sum` and `product`

```

46 sum01 :: Num a => [a] -> a
47 sum01 []      = 0
48 sum01 (x:xs)  = x + sum01 xs
50
51 product01 :: Num a => [a] -> a
52 product01 []   = 1
53 product01 (x:xs) = x * product01 xs

```

- We abstract out the common pattern:

```

53 foldr01 f v [] = v
54 foldr01 f v (x:xs) = f x (foldr01 f v xs)

```

- We now can define:

```

55 sum02 xs      = foldr01 (+) 0 xs
56 product02 xs = foldr01 (*) 1 xs

```

- `foldr` takes an operator ($\otimes$), a final value $\star$ and a list `xs`

```
foldr (⊗) ★ [x1, x2, ..., xn]
= x1 ⊗ (x2 ⊗ (... (xn ⊗ ★) ...))
```

- The operator ($\otimes$) is substituted for each list constructor (:)
- The final value ★ is substituted for the empty list []
- The function is called *fold right* because of the direction of the bracketing
- Beware operator associativity

Further Foldr Examples

- `or` takes a list of Booleans and finds the disjunction of all the values
- Recursive version followed by `foldr` version

```
57 or01 :: [Bool] -> Bool
58 or01 [] = False
59 or01 (x:xs) = x || or01 xs
```

```
61 or02 xs = foldr (||) False xs
```

- `and` takes a list of Booleans and finds the conjunction of all the values
- Recursive version followed by `foldr` version

```
63 and01 :: [Bool] -> Bool
64 and01 [] = True
65 and01 (x:xs) = x && and01 xs
```

```
67 and02 xs = foldr (&&) True xs
```

- `foldr` is more general than you might expect
- `length` takes a list and returns its length

Health warning: `length` is more general than shown here

Recursive version followed by `foldr` version

```
69 length05 :: [a] -> Int
70 length05 [] = 0
71 length05 (x:xs) = 1 + length05 xs
```

```
73 length06 xs = foldr01 (\x n -> 1 + n) 0 xs
```

- `reverse` takes a list and returns the reverse

Recursive version followed by `foldr` version

```
75 reverse01 :: [a] -> [a]
76 reverse01 [] = []
77 reverse01 (x:xs) = reverse01 xs ++ [x]
```

```
79 reverse02 xs = foldr01 snoc [] xs
80 where snoc x xs = xs ++ [x]
```

Type of foldr01

- As we have defined `foldr01` it has the type

```
82 foldr01 :: (a -> b -> b) -> b -> [a] -> b
```

- Without the later examples you may have thought it was

```
foldr01 :: (a -> a -> a) -> a -> [a] -> a
```

- The GHC Prelude has a more general version since this pattern of computation can be performed over more data types than just lists — see later

7 User Defined Data Types

7.1 Algebraic Datatypes

- Haskell provides a way of providing new concrete data types by declaring the names of a type and names of the elements of the type
- The names of a type is called a *type constructor*
- The names of elements of a type is called a *data constructor*
- Example: `Day` for days of the week

```
83 data Day
84   = Monday | Tuesday | Wednesday | Thursday
85   | Friday | Saturday | Sunday
86   deriving (Show, Read, Eq, Ord, Enum, Bounded)
```

- Names of type constructors start with upper case letters
- Names of data constructors start with upper case letters but symbolic infix constructors can be formed
- The `deriving` clause creates automatic instances of the type classes `Show`, `Read`, `Eq`, `Ord`, `Enum`, `Bounded`
- `tomorrow` takes a `Day` and returns the next

```
88 tomorrow dy
89   = if dy == Sunday then Monday else succ dy
91 tomorrow01 :: Day -> Day
92 tomorrow01 dy
93   = toEnum ((fromEnum dy + 1) `mod` 7)
```

- Note that `tomorrow01` requires the type signature (or type annotation) otherwise `toEnum` and `fromEnum` would not know which type
- The brackets are required since `'mod'` has precedence 7, the same as `(*)`, `(/)`
- Several provided types are defined this way

```
data Bool = False | True
  deriving (Show, Read, Eq, Ord, Enum, Bounded)
```

- Note that `Day` has 8 elements, `Bool` has 3 elements since `undefined` (bottom, $\perp$) is a member of every type

7.1.1 Standard Haskell Types

- We have already met characters, strings, numbers and `Bool`
- Lists are an algebraic data type with a special syntax — it is as if it had the following declaration

```
data [a] = [] | a : [a]
    deriving (Eq, Ord)
```

- Tuples are an algebraic data type with special syntax — for pairs the single constructor is `(,)`

```
GHCi> (3,5) == (,) 3 5
True
GHCi> :t (,)
(,) :: a -> b -> (a, b)
```

- The Unit datatype `()` has only one non- $\perp$ member, the nullary constructor `()`

```
data () = ()
    deriving (Eq, Ord, Bounded, Enum, Read, Show)
```

- Function types — functions are an abstract type — no constructors directly create functional values.
- The `Maybe` datatype provides a simple optional value — useful for error handling — here is the declaration and the `maybe` function as an example usage

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord)

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

- The `Either` datatype provides for richer error handling

```
data Either a b = Left a | Right b
    deriving (Eq, Ord, Read, Show)

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

8 Algebraic Data Type Exercises

8.1 Algebraic Data Type Exercises S0607 Q1

- Here is an algebraic data type representing temperature

```
94 data Temperature
95   = Celsius Float | Fahrenheit Float | Kelvin Float
96   deriving (Eq, Show, Read)
```

- Write the following functions
- `tempToCelsius` takes a temperature and converts it to Celsius
- `tempToFahrenheit` takes a temperature and converts it to Fahrenheit

- `tempToKelvin` takes a temperature and converts it to Kelvin
- The formulas are at [Conversion of units of temperature](#)

[Go to Algebraic Data Type Exercises S0607 A 1](#)

8.2 Algebraic Data Type Exercises S0607 A1

```

97 tempToCelsius (Celsius x) = Celsius x
98 tempToCelsius (Fahrenheit x) = Celsius ((x - 32)*5/9)
99 tempToCelsius (Kelvin x) = Celsius (x - 273.15)

101 tempToFahrenheit (Celsius x) = Fahrenheit (x*9/5 + 32)
102 tempToFahrenheit (Fahrenheit x)
103   = Fahrenheit x
104 tempToFahrenheit (Kelvin x) = Fahrenheit (x*9/5 - 459.67)
105 -- 459.67 = -273.15*9/5 + 32

107 tempToKelvin (Celsius x) = Kelvin (x + 273.15)
108 tempToKelvin (Fahrenheit x)
109   = Kelvin ((x + 459.672)*5/9)
110 tempToKelvin (Kelvin x) = Kelvin x

112 temp01 = Celsius 0
113 temp02 = Kelvin 0
114 temp03 = Fahrenheit 0
115 temp04 = Celsius 100

117 temps = [temp01,temp02,temp03,temp04]
118 tempConvs = [tempToCelsius,tempToFahrenheit,tempToKelvin]

120 test03 = [f x | f <- tempConvs, x <- temps]
121 test03out
122   = [Celsius 0.0,Celsius (-273.15),Celsius (-17.777779),Celsius 100.0
123     ,Fahrenheit 32.0,Fahrenheit (-459.67),Fahrenheit 0.0,Fahrenheit 212.0
124     ,Kelvin 273.15,Kelvin 0.0,Kelvin 255.37332,Kelvin 373.15]

127 test04 = [[f x | f <- tempConvs] | x <- temps]
128 test04out
129   = [[Celsius 0.0,Fahrenheit 32.0,Kelvin 273.15]
130     ,[Celsius (-273.15),Fahrenheit (-459.67),Kelvin 0.0]
131     ,[Celsius (-17.777779),Fahrenheit 0.0,Kelvin 255.37332]
132     ,[Celsius 100.0,Fahrenheit 212.0,Kelvin 373.15]]

```

[Go to Algebraic Data Type Exercises S0607 Q 1](#)

8.3 Algebraic Data Type Exercises S0607 Q2

- Here is a (very) simple family database

```

134 data Person = Person {name :: String
135                       ,father :: Maybe Person
136                       ,mother :: Maybe Person}
137   deriving (Eq,Show,Read)

139 phil   = Person "Phil" (Just ron) (Just hilda)
140 beryl  = Person "Beryl" Nothing (Just dora)
141 ron    = Person "Ron" (Just joe) (Just jane)
142 hilda  = Person "Hilda" (Just sam) (Just florrie)
143 dora   = Person "Dora" (Just arthur) (Just hannah)
144 joe    = Person "Joseph" Nothing Nothing
145 jane   = Person "Jane" Nothing Nothing
146 sam    = Person "Sam" Nothing Nothing
147 florrie = Person "Florence" Nothing Nothing
148 arthur = Person "Arthur" Nothing Nothing
149 hannah = Person "Hannah" Nothing Nothing

```

```

150 people = [phil, beryl, ron, hilda, dora
151           , joe, jane, sam, florrie, arthur, hannah]

```

- In the data, `Nothing` represents a missing value
- Write a function `nameStr` which takes a `Maybe Person` and returns the name if present otherwise the string “Unknown”
- Use the standard Prelude function `maybe` — see [GHC Prelude](#) — note you can search quickly by typing `s` — try it, it’s neat (it is part of Hackage)

```

153 nameStr :: Maybe Person -> String

```

- Write a function `nameMbe` which takes a `Maybe Person` and returns the name (if known) as a `Maybe String`

```

154 nameMbe :: Maybe Person -> Maybe String

```

[Go to Algebraic Data Type Exercises S0607 A 2](#)

8.4 Algebraic Data Type Exercises S0607 A2

- `nameStr`

```

155 nameStr mPers = maybe "Unknown" name mPers

```

- `nameMbe`

```

156 nameMbe (Just pers) = Just (name pers)
157 nameMbe Nothing     = Nothing

```

[Go to Algebraic Data Type Exercises S0607 Q 2](#)

8.5 Algebraic Data Type Exercises S0607 Q3

- Write a function `maternalGrandfather01` that takes a `Person` and returns their maternal grandfather (if known)

```

158 maternalGrandfather01 :: Person -> Maybe Person

```

- Write a function `paternalGrandfather01` that takes a `Person` and returns their paternal grandfather (if known)

```

160 paternalGrandfather01 :: Person -> Maybe Person

```

[Go to Algebraic Data Type Exercises S0607 A 3](#)

8.6 Algebraic Data Type Exercises S0607 A3

- `maternalGrandfather01`

```

161 maternalGrandfather01 p
162   = case mother p of
163     Nothing -> Nothing
164     Just mum ->
165       case father mum of
166         Nothing -> Nothing
167         Just mgf ->
168           Just mgf

```

- `paternalGrandfather01`

```

169 paternalGrandfather01 p
170   = case father p of
171     Nothing -> Nothing
172     Just dad ->
173       case father dad of
174         Nothing -> Nothing
175         Just pgf ->
176           Just pgf

```

[Go to Algebraic Data Type Exercises S0607 Q 3](#)

8.7 Algebraic Data Type Exercises S0607 Q4

- Write a function `bothGrandfathers01` that takes a `Person` and returns a pair of grandfathers, if they both exist

```

177 bothGrandfathers01 :: Person
178                   -> Maybe (Person, Person)

```

[Go to Algebraic Data Type Exercises S0607 A 4](#)

8.8 Algebraic Data Type Exercises S0607 A4

- `bothGrandfathers01`

```

179 bothGrandfathers01 p
180   = case father p of
181     Nothing -> Nothing
182     Just dad ->
183       case father dad of
184         Nothing -> Nothing
185         Just gf1 ->
186           case mother p of
187             Nothing -> Nothing
188             Just mum ->
189               case father mum of
190                 Nothing -> Nothing
191                 Just gf2 ->
192                   Just (gf1, gf2)

```

- In each of the last three examples we had a common pattern:
- If a computation fails at any point we return `Nothing`
- If it succeeds we pass the value on to the next stage
- Finally we return a value wrapped in a `Maybe` value
- Haskell captures this pattern with two functions

```

(>=>) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >=> g = Nothing
Just x >=> g = g x
-- (>=>) is spoken as \emph{bind}
return :: a -> Maybe a
return x = Just x

```

- We now rewrite the previous three functions:

```

194 maternalGrandfather02 p
195   = mother p >=> father
196
197 paternalGrandfather02 p
198   = father p >=> father
199
200 bothGrandfathers02 p
201   = father p >=>
202     (\dad -> father dad >=>
203       (\gf1 -> mother p >=>
204         (\mum -> father mum >=>
205           (\gf2 -> return (gf1,gf2)
206             ))))

```

- Haskell further provides the `do` notation to reduce syntactic clutter

```

do {p} = p
do {p; stmnts} = p >> do {stmnts}
do {x <- p; stmnts} = p >=> \x -> do {stmnts}

```

```

(>>) :: Maybe a -> Maybe b -> Maybe b
m >> n = m >=> \x -> n
-- (>>) is spoken then

```

- `(>>)` is a convenience function that sequences two computational contexts where the second does not involve the value carried in the first
- We can now give the brief form of `bothGrandfathers`
- Note that the offside rule means we can dispense with `(;)` or choose not to
- Without `(;)`

```

208 bothGrandfathers03 p = do
209   dad <- father p
210   gf1 <- father dad
211   mum <- mother p
212   gf2 <- father mum
213   return (gf1,gf2)

```

- With `(;)` — what does it look like ?

```

215 bothGrandfathers04 p = do {
216   dad <- father p ;
217   gf1 <- father dad ;
218   mum <- mother p ;
219   gf2 <- father mum ;
220   return (gf1,gf2) ;
221 }

```

- The last two examples look like code snippets from an imperative language
- The expression `father p` which has type `Maybe Person` is interpreted as a statement in an imperative language that returns a `Person` as a result or fails
- Under this interpretation, the `then`, `(>>)` operator is an implementation of the semicolon

- The `bind`, (`>=>`) operator is an implementation of the semicolon and assignment (binding) of the result of a previous computational step

[Go to Algebraic Data Type Exercises S0607 Q 4](#)

8.9 Algebraic Data Type Exercises S0607 Q5

- Write a function `bothGFNames` that takes a `Person` and returns the names of both grandfathers, if they both are known

```
222 bothGFNames :: Person
223             -> Maybe (String, String)
```

[Go to Algebraic Data Type Exercises S0607 A 5](#)

8.10 Algebraic Data Type Exercises S0607 A5

- `bothGFNames` long version

```
225 bothGFNames p
226   = case father p of
227     Nothing -> Nothing
228     Just dad ->
229       case father dad of
230         Nothing -> Nothing
231         Just gf1 ->
232           case mother p of
233             Nothing -> Nothing
234             Just mum ->
235               case father mum of
236                 Nothing -> Nothing
237                 Just gf2 ->
238                   Just (name gf1, name gf2)
```

- `bothGFNames` with `return` and `bind`, (`>=>`)

```
240 bothGFNames01 :: Person
241                -> Maybe (String,String)
242 bothGFNames01 p
243   = father p >=>
244     (\dad -> father dad >=>
245       (\gf1 -> mother p >=>
246         (\mum -> father mum >=>
247           (\gf2 -> return (name gf1,name gf2)
248             )))
```

- `bothGFNames` with `do` notation

```
250 bothGFNames02 :: Person
251                -> Maybe (String,String)
252 bothGFNames02 p = do
253   dad <- father p
254   gf1 <- father dad
255   mum <- mother p
256   gf2 <- father mum
257   return (name gf1,name gf2)
```

- `bothGFNames` with `do` notation and explicit `(;)`, `{}`, `{}`

```
259 bothGFNames03 :: Person
260                -> Maybe (String,String)
261 bothGFNames03 p = do {
262   dad <- father p ;
263   gf1 <- father dad ;
```

```

264   mum <- mother p ;
265   gf2 <- father mum ;
266   return (name gf1,name gf2) ;
267 }

```

[Go to Algebraic Data Type Exercises S0607 Q 5](#)

8.11 Algebraic Data Type Exercises S0607 Q6

- Write `eitherGFNames` which takes a `Person` and returns a pair of names if either or both or none are known

```

269 eitherGrandfather
270   :: Person -> (Maybe String, Maybe String)

```

[Go to Algebraic Data Type Exercises S0607 A 6](#)

8.12 Algebraic Data Type Exercises S0607 A6

- Possible answer

```

272 maternalGrandfather :: Person -> Maybe Person
273 maternalGrandfather p = do
274   mum <- mother p
275   gfm <- father mum
276   return gfm
277
278 paternalGrandfather :: Person -> Maybe Person
279 paternalGrandfather p = do
280   dad <- father p
281   gfp <- father dad
282   return gfp
283
284 -- eitherGrandfather
285 --   :: Person -> (Maybe Person,Maybe Person)
286 eitherGrandfather p
287   = (nameMbe (maternalGrandfather p)
288     ,nameMbe (paternalGrandfather p))

```

[Go to Algebraic Data Type Exercises S0607 Q 6](#)

8.13 Laws for return and bind

- The `return` and `bind`, (`>>=`) functions are provided by Haskell since they are much more general than just being used for the `Maybe a` datatype
- They are provided by a *type class*
- Any *instance* must obey the following laws

```

1  return x >>= f = f x  -- left unit
2  m >>= return  = m     -- right unit
3  (m >>= f) >>= g = m >>= (\x -> f x >>= g)
4  -- associativity

```

- These laws ensure that the *instance* of this *type class* works as expected and fits with other instances (and other type classes)

- Exercise: verify the laws for the definitions of `return` and `bind`, (`>>=`) for the `Maybe a` type
- The `return` and `bind`, (`>>=`) — verification of laws

```

1  return x >>= f
2  → Just x >>= f
3  → f x

5  m >>= return
6  Nothing >>= return → Nothing (= m)
7  Just x >>= return → return x → Just x (= m)

9  (m >>= f) >>= g
10 (Nothing >>= f) >>= g → Nothing >>= g → Nothing
11 (Just x >>= f) >>= g → f x >>= g

13 m >>= (\x -> f x >>= g)
14 Nothing >>= (\x -> f x >>= g) → Nothing
15 Just x >>= (\x -> f x >>= g)
16 → (\x -> f x >>= g) x
17 → f x >>= g

```

- The examples above come from [Haskell Wikibook: Understanding Monads](#)
- We are being a bit premature and introducing the `Maybe a` instance of the `Monad` type class as a motivating example (it is meant to look useful)
- This pattern of computation is very common (it encapsulates just about all imperative programming)
- **Return as a neutral element** — the behaviour of `return` is specified by the left and right unit laws — `return` does not perform computation, it just collects values
- **Associativity of bind** — this makes sure that the bind operator (like the semicolon) only cares about the order of computations not about their nesting

9 Tree Data Types

9.1 Binary Trees and Recursion Schemes

- Binary trees appear in lots of applications and have common patterns of recursive definitions for many functions
- In imperative, procedural programming, common patterns of control flow with GO-TOs were abstracted out with structured programming in the 1970s — sequence, selection and iteration — which required new language constructs
- In functional programming, we can often express new constructions and abstractions as higher-order functions
- This decouples *how* a function recurses over data from *what* the function actually does
- Whilst it takes some effort to learn about the common patterns and their higher-order functions, there are several advantages (as there are for any abstraction)
- We can discover general properties of the abstraction and hence infer properties of specific instances for free.

- We can also use the general properties to *calculate* functions

9.2 Binary Trees: Data Types and Examples

- We shall (mainly) use the following algebraic data type for binary trees

```

291 data BinTree a
292   = EmptyBT | NodeBT a (BinTree a) (BinTree a)
293   deriving (Eq, Show, Read)

```

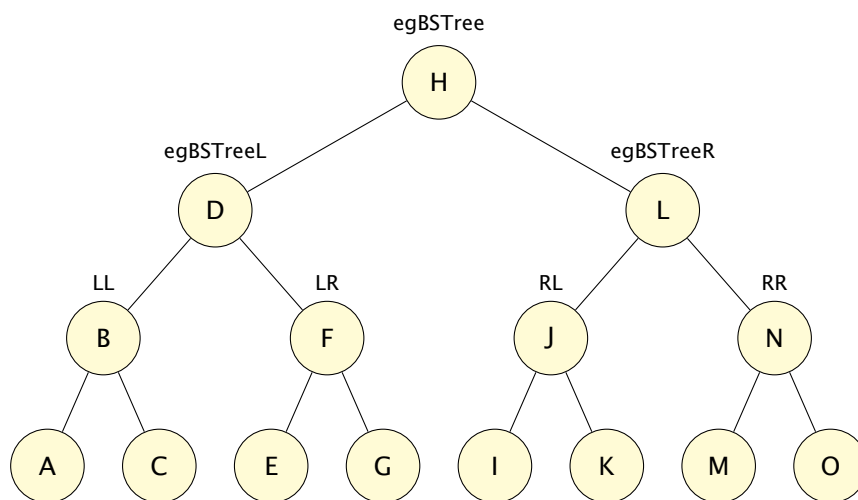
- We also declare a **Letter** algebraic data type for convenience

```

295 data Letter
296   = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O
297   deriving (Eq, Ord, Enum, Bounded, Show, Read)

```

9.2.1 Example Binary Tree: egBSTree



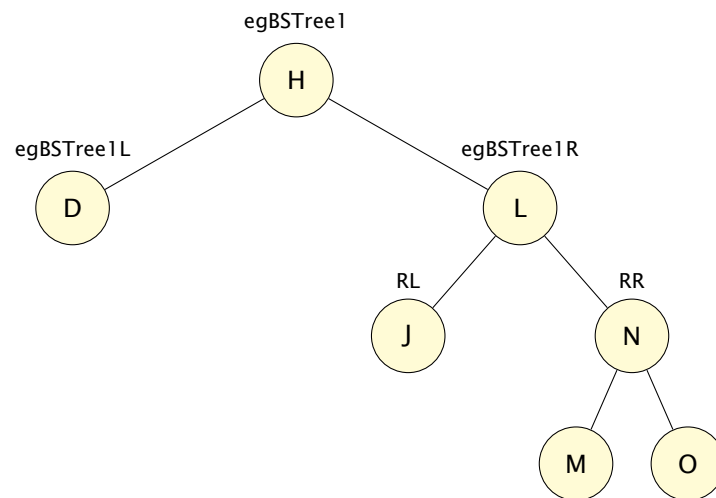
- Name convention: variables must start with lower case so we have **eg** (for *example, exempli gratia*), **BSTree** indicates this is not just a Binary Tree but also a Binary Search Tree

```

299 egBSTree :: BinTree Letter
300 egBSTree
301   = NodeBT H
302     (NodeBT D
303       (NodeBT B
304         (NodeBT A EmptyBT EmptyBT)
305         (NodeBT C EmptyBT EmptyBT))
306       (NodeBT F
307         (NodeBT E EmptyBT EmptyBT)
308         (NodeBT G EmptyBT EmptyBT))
309     )
310     (NodeBT L
311       (NodeBT J
312         (NodeBT I EmptyBT EmptyBT)
313         (NodeBT K EmptyBT EmptyBT))
314       (NodeBT N
315         (NodeBT M EmptyBT EmptyBT)
316         (NodeBT O EmptyBT EmptyBT))
317     )

```

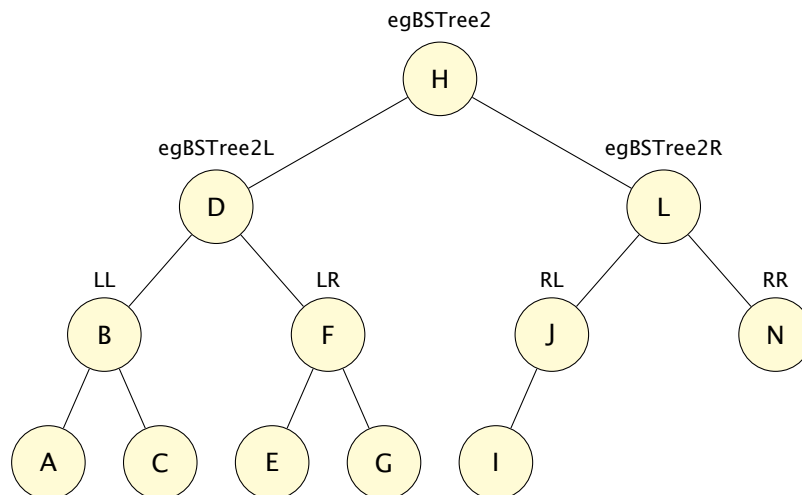
9.2.2 Example Binary Tree: egBSTree1



```

319 egBSTree1 :: BinTree Letter
320 egBSTree1
321   = NodeBT H
322     (NodeBT D EmptyBT EmptyBT)
323     (NodeBT L
324       (NodeBT J EmptyBT EmptyBT)
325       (NodeBT N
326         (NodeBT M EmptyBT EmptyBT)
327         (NodeBT O EmptyBT EmptyBT)))
  
```

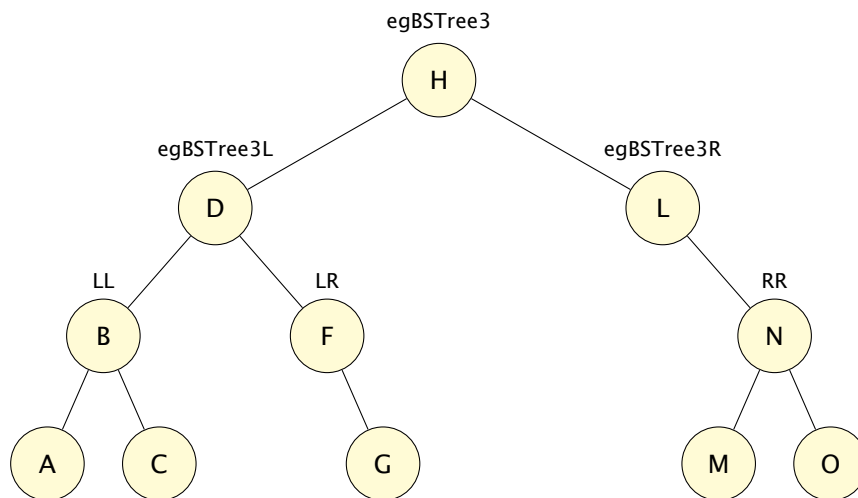
9.2.3 Example Binary Tree: egBSTree2



```

329 egBSTree2 :: BinTree Letter
330 egBSTree2
331   = NodeBT H
332     (NodeBT D
333       (NodeBT B
334         (NodeBT A EmptyBT EmptyBT)
335         (NodeBT C EmptyBT EmptyBT))
336       (NodeBT F
337         (NodeBT E EmptyBT EmptyBT)
338         (NodeBT G EmptyBT EmptyBT)))
339     (NodeBT L
340       (NodeBT J
341         (NodeBT I EmptyBT EmptyBT)
342         EmptyBT)
343       (NodeBT N EmptyBT EmptyBT))
  
```

9.2.4 Example Binary Tree: egBSTree3



```

345 egBSTree3 :: BinTree Letter
346 egBSTree3
347   = NodeBT H
348     (NodeBT D
349       (NodeBT B
350         (NodeBT A EmptyBT EmptyBT)
351         (NodeBT C EmptyBT EmptyBT))
352       (NodeBT F
353         EmptyBT
354         (NodeBT G EmptyBT EmptyBT)))
355     (NodeBT L
356       EmptyBT
357       (NodeBT N
358         (NodeBT M EmptyBT EmptyBT)
359         (NodeBT O EmptyBT EmptyBT))))

```

```

361 data BinTreeL a = EmptyBTL | NodeBTL {dataItemBTL :: a
362                                     ,leftBTL :: BinTreeL a
363                                     ,rightBTL :: BinTreeL a}
364   deriving (Eq,Show,Read)

366 egBSTreeL
367   = NodeBTL H
368     (NodeBTL D
369       (NodeBTL B
370         (NodeBTL A EmptyBTL EmptyBTL)
371         (NodeBTL C EmptyBTL EmptyBTL))
372       (NodeBTL F
373         (NodeBTL E EmptyBTL EmptyBTL)
374         (NodeBTL G EmptyBTL EmptyBTL))
375     )
376     (NodeBTL L
377       (NodeBTL J
378         (NodeBTL I EmptyBTL EmptyBTL)
379         (NodeBTL K EmptyBTL EmptyBTL))
380       (NodeBTL N
381         (NodeBTL M EmptyBTL EmptyBTL)
382         (NodeBTL O EmptyBTL EmptyBTL))
383     )

```

9.3 Alternative Tree Types

- In computing, trees can come in many forms. There can be trees with data only at the leaves, data only in the internal nodes, data of different types in alternate levels (useful for game trees), or multiway trees

```

385 data LeafTree a = LeafLT a
386 | NodeLT a (LeafTree a) (LeafTree a)
387   deriving (Eq, Show, Read)
388 data IntTree a = LeafIT a
389 | NodeIT a (IntTree a) (IntTree a)
390   deriving (Eq, Show, Read)
391 data DualTree a b = LeafDT a
392 | NodeDT a (DualTree b a) (DualTree b a)
393   deriving (Eq, Show, Read)
394 data RoseTree a = LeafRT a [RoseTree a]
395   deriving (Eq, Show, Read)

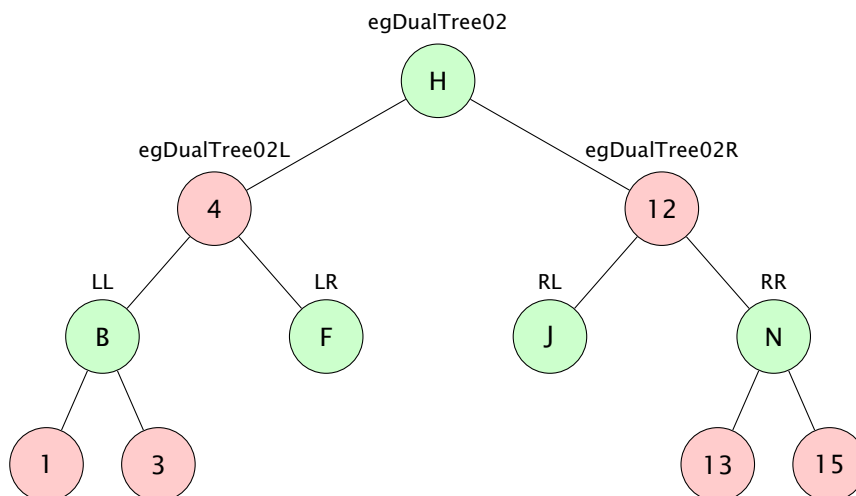
```

- Exercise: give an example of **DualTree Letter Integer**
- Examples of **DualTree Letter Integer**

```

397 egDualTree01
398   = LeafDT H
400 egDualTree02
401   = NodeDT H
402     (NodeDT 4 (NodeDT B (LeafDT 1) (LeafDT 3))
403              (LeafDT F))
404     (NodeDT 12 (LeafDT J)
405                (NodeDT N (LeafDT 13) (LeafDT 15)))
407 egDualTree03
408   = NodeDT 8
409     (NodeDT D (NodeDT 2 (LeafDT A) (LeafDT C))
410              (LeafDT 6))
411     (NodeDT L (LeafDT 10)
412                (NodeDT 14 (LeafDT M) (LeafDT 0)))

```



10 Tree Data Type Exercises

- These exercises or short topics are aimed at illustrating common patterns of recursion in tree structures and showing how the **fold** family of functions naturally extends to tree structures (or any algebraic data type)

10.1 Tree Data Type Exercises S0607 Q1

- Write functions **inOrderBT01**, **preOrderBT01**, **postOrderBT01** which take a **BinTree a** and returns in order, pre order, post order traversals of the tree

```

414 inOrderBT01  :: BinTree a -> [a]
416 preOrderBT01 :: BinTree a -> [a]
418 postOrderBT01 :: BinTree a -> [a]

```

[Go to Tree Exs S0607 A 1](#)

10.2 Tree Data Type Exercises S0607 A1

- Here are the usual recursive definitions

```

419 inOrderBT01 EmptyBT = []
420 inOrderBT01 (NodeBT x leftBT rightBT)
421   = (inOrderBT01 leftBT)
422     ++ [x] ++ (inOrderBT01 rightBT)

424 preOrderBT01 EmptyBT = []
425 preOrderBT01 (NodeBT x leftBT rightBT)
426   = [x]
427     ++ (preOrderBT01 leftBT) ++ (preOrderBT01 rightBT)

429 postOrderBT01 EmptyBT = []
430 postOrderBT01 (NodeBT x leftBT rightBT)
431   = (postOrderBT01 leftBT)
432     ++ (postOrderBT01 rightBT) ++ [x]

```

- Each of the functions has a common pattern
- The constructors of the algebraic data type are replaced by functions (or a value) that *consume* or *transform* the data structure
- This is a generalisation of the [fold](#) function given in S0405 for lists

```

434 foldBinTree
435   :: (a -> b -> b -> b) -> b -> BinTree a -> b

437 foldBinTree fNodeBT fEmptyBT EmptyBT = fEmptyBT
438 foldBinTree fNodeBT fEmptyBT (NodeBT x leftT rightT)
439   = fNodeBT x (foldBinTree fNodeBT fEmptyBT leftT)
440               (foldBinTree fNodeBT fEmptyBT rightT)

```

- We now define the traversal functions in terms of the fold function

```

442 inOrderFoldBT :: BinTree a -> [a]
443 inOrderFoldBT t
444   = foldBinTree
445     fNodeBTToInOrderList fEmptyBTToInOrderList t

447 fEmptyBTToInOrderList = []
448 fNodeBTToInOrderList x leftTList rightTList
449   = leftTList ++ [x] ++ rightTList

```

```

451 preOrderFoldBT :: BinTree a -> [a]
452 preOrderFoldBT t
453   = foldBinTree
454     fNodeBTToPreOrderList fEmptyBTToPreOrderList t

456 fEmptyBTToPreOrderList = []
457 fNodeBTToPreOrderList x leftTList rightTList
458   = [x] ++ leftTList ++ rightTList

460 postOrderFoldBT :: BinTree a -> [a]
461 postOrderFoldBT t
462   = foldBinTree
463     fNodeBTToPostOrderList fEmptyBTToPostOrderList t

```



```

465 fEmptyBTToPostOrderList = []
466 fNodeBTToPostOrderList x leftTList rightTList
467   = leftTList ++ rightTList ++ [x]

```

[Go to Tree Exs S0607 Q 1](#)

10.3 Tree Data Type Exercises S0607 Q2

- A *level order* traversal takes a tree and returns the list of lists of items at each level
- In the Binary Trees notes, the final functional definition:

```

469 levelOrderBT :: BinTree a -> [[a]]
470 levelOrderBT EmptyBT = []
471 levelOrderBT (NodeBT x leftT rightT)
472   = [x] : longZipWith (++)
473       (levelOrderBT leftT)
474       (levelOrderBT rightT)

```

```

476 longZipWith :: (a -> a -> a) -> [a] -> [a] -> [a]
477 longZipWith f [] ys      = ys
478 longZipWith f (a:xs) []   = (a:xs)
479 longZipWith f (a:xs) (b:ys)
480   = (f a b) : (longZipWith f xs ys)

```

- Define *level order* as a fold

[Go to Tree Exs S0607 A 2](#)

10.4 Tree Data Type Exercises S0607 A2

```

482 levelOrderFoldBT :: BinTree a -> [[a]]
483 levelOrderFoldBT t
484   = foldBinTree
485       fNodeBTToLevelOrder fEmptyBTToLevelOrder t

```

```

487 fEmptyBTToLevelOrder = []

```

```

489 fNodeBTToLevelOrder :: a -> [[a]] -> [[a]] -> [[a]]
490 fNodeBTToLevelOrder x leftTOrder rightTOrder
491   = [x] : longZipWith (++)
492       leftTOrder rightTOrder

```

```

GHCi> levelOrderFoldBT egBSTree
[[H],[D,L],[B,F,J,N],[A,C,E,G,I,K,M,O]]
GHCi> levelOrderFoldBT egBSTree1
[[H],[D,L],[J,N],[M,O]]

```

[Go to Tree Exs S0607 Q 2](#)

10.5 Tree Data Type Exercises S0607 Q3

- Using a fold, define `heightBT` which returns the height of a tree
- here is the usual recursive definition

```

493 heightBT :: BinTree a -> Int

```

```

495 heightBT EmptyBT = 0
496 heightBT (NodeBT x leftT rightT)
497   = 1 + max (heightBT leftT) (heightBT rightT)

```

[Go to Tree Exs S0607 A 3](#)

10.6 Tree Data Type Exercises S0607 A3

```

498 heightFoldBT t
499   = foldBinTree
500     fNodeBTToHeight fEmptyBTToHeight t
502 fEmptyBTToHeight = 0
503 fNodeBTToHeight x leftTHeight rightTHeight
504   = 1 + max leftTHeight rightTHeight

```

```

GHCi> heightBT egBSTree
4
GHCi> heightFoldBT egBSTree
4

```

[Go to Tree Exs S0607 Q 3](#)

10.7 Tree Data Type Exercises S0607 Q4

- Using a fold, define `sizeBT` which returns the size of a tree
- Here is the usual recursive definition

```

505 sizeBT :: BinTree a -> Int
507 sizeBT EmptyBT = 0
508 sizeBT (NodeBT x leftT rightT)
509   = 1 + (sizeBT leftT) + (sizeBT rightT)

```

[Go to Tree Exs S0607 A 4](#)

10.8 Tree Data Type Exercises S0607 A4

```

510 sizeFoldBT t
511   = foldBinTree
512     fNodeBTToSize fEmptyBTToSize t
514 fEmptyBTToSize = 0
515 fNodeBTToSize x leftTSize rightTSize
516   = 1 + leftTSize + rightTSize

```

```

GHCi> sizeBT egBSTree
15
GHCi> sizeFoldBT egBSTree
15

```

[Go to Tree Exs S0607 Q 4](#)

10.9 Tree Data Type Exercises S0607 Q5

- Write a function `numLeavesBT` which takes a tree and returns the number of leaves
a leaf is a node with two empty subtrees

```

517 isEmptyBT EmptyBT = True
518 isEmptyBT (NodeBT x leftT rightT) = False

520 isBothEmptyBT t1 t2
521   = isEmptyBT t1 && isEmptyBT t2

523 numLeavesBT EmptyBT = 0
524 numLeavesBT (NodeBT x leftT rightT)
525   = if isBothEmptyBT leftT rightT
526       then 1
527       else numLeavesBT leftT
528           + numLeavesBT rightT

```

- Write `numLeavesFoldBT` which uses `foldBinTree`

[Go to Tree Exs S0607 A 5](#)

10.10 Tree Data Type Exercises S0607 A5

- We calculate the function using the *Universal Property*

```

numLeaves t = fold f v t
numLeaves EmptyBT = v
numLeaves (NodeBT x leftT rightT)
  = f x leftTNL rightTNL
-- defn of numLeaves
  = if isBothEmptyBT leftT rightT
      then 1
      else (numLeavesBT leftT)
          + (numLeaves rightT)
-- Eureka step to get rid of isolated leftT, rightT
  = if isBothZero (numLeavesBT leftT) (numLeaves rightT)
      then 1
      else (numLeavesBT leftT)
          + (numLeaves rightT)
-- this gives us the required definition

```

```

529 isBothZero x y
530   = x == 0 && y == 0

532 numLeavesFoldBT t
533   = foldBinTree
534     fNodeBTToNumL fEmptyBTToNumL t

536 fEmptyBTToNumL = 0
537 fNodeBTToNumL x leftTNL rightTNL
538   = if isBothZero leftTNL rightTNL
539       then 1
540       else leftTNL + rightTNL

```

[Go to Tree Exs S0607 Q 5](#)

10.11 Tree Data Type Exercises S0607 Q6

- The function `minDepthBT` can be defined recursively as

```

541 minDepthBT EmptyBT = 0
542 minDepthBT (NodeBT x leftT rightT)
543   = 1 + min (minDepthBT leftT) (minDepthBT rightT)

```

- This will visit every node in the tree but the computation can stop earlier
- See [egBSTree1](#) — we can stop when we meet node `D`

- Suggest ways of making this more efficient — this may or may not use fold

[Go to Tree Exs S0607 A 6](#)

10.12 Tree Data Type Exercises S0607 A6

- We can do this by keeping track of the depth in the tree and the minimum depth so far

```

544 minDepthBT01 :: BinTree a -> Int
545 minDepthBT01 t
546   = minD t 0 maxBound
547   -- here maxBound is regarded as infinity
548 minD :: BinTree a -> Int -> Int -> Int
549 minD EmptyBT d m = min d m
550 minD (NodeBT x leftT rightT) d m
551   = if d + 1 >= m
552     then m
553     else minD leftT (d + 1) (minD rightT (d + 1) m)

```

- We can do better than this if we consider the tree level by level
- TODO: complete A 6

[Go to Tree Exs S0607 Q 6](#)

11 Recursion Schemes

- [Get Height of Tree](#) (20 August 2018) StackExchange Code Review — uses *catamorphism*
- [Practical Recursion Schemes](#) (20 August 2018) — Jared Tobin 5 September 2015
- [Haskell WikiBook: Category theory](#) (20 August 2018)
- [Recursion schemes for dummies?](#) (21 August 2018)
- [Wikipedia: Recursion schemes](#) (21 August 2018)
- [An Introduction to Recursion Scheme](#) — Patrick Thomson 15 February 2014
- [Recursion Schemes, Part II: A Mob of Morphisms](#) — 21 August 2015
- [Recursion Schemes, Part III: Folds in Context](#) — 20 July 2016
- [Recursion Schemes, Part IV: Time is of the Essence](#) — 11 October 2017
- [Recursion Schemes, Part 4 1/2: Better Living Through Base Functors](#) — 24 January 2018
- [Recursion Schemes, Part V: Hello, Hylomorphisms](#) — 17 April 2018 Patrick Thomson

12 Interactive Programming

12.1 I/O The Problem

- Calculating values in the language and performing actions outside the language are different
- Actions have to be performed in the correct order
- Call-by-value (or strict) functional languages take the approach of imperative languages
- I/O is *treated* as a function (even though it is a side effect)
- The language design has to specify the order of evaluation of expressions
- Suppose we have a function `printChar` that takes a character, prints it to standard output and returns nothing
- In imperative languages and strict functional languages, the programmer has to ensure that calls to `printChar` happen in the correct order

- Consider

```
xs = [printChar 'a', printChar 'b']
```

- Call-by-need (or lazy) languages (such as Haskell) do not specify order of evaluation
- The `printChar` calls are only performed if the elements of the list are evaluated
- `length xs` would return `2` but does not need to evaluate the elements of `xs`
- *Laziness* and *side effects* appear incompatible

12.2 I/O Solution (1)

- First version of Haskell:
- **View of Program** — Top level program is a function from a (lazy) list (stream) of system responses returning a (lazy) list of system requests.

```
main :: [Response] -> [Request]
```

- `Request` and `Response` are both ordinary algebraic data types

```
type FilePath = String

data Request = ReadFile FilePath
             | WriteFile FilePath String
             | ...

data Response = RequestFailed
              | ReadSucceeded String
              | WriteSucceeded
              | ...
```

- This was used in the first version of Haskell
- but it has problems:

- Hard to extend since it can only be extended by changing the `Request` and `Response` types
- There is no close connection between a request and its corresponding response — hence easy to write a program that gets out of step
- Even if not out of step, it is too easy to evaluate the response stream too eagerly and hence block emitting a request

12.3 Monadic I/O

- Need an abstract data type that allows us to calculate programs that have side effects in a purely functional way
- A value of type `IO a` is an *action* that, when *performed*, may do some input/output, before delivering a value of type `a`
- We distinguish between *evaluating an expression* and *performing an action*
- Sometimes *actions* are referred to as *computations*
- It is as if we have:

```
type IO a = World -> (a,World)
```

- A value of `IO a` is a function that takes an argument of type `World` and delivers a new `World` together with a result of type `a`
- We then provide some primitive operations and a small number of ways of combining the primitive operations
- The top level program is of type `IO ()`

```
getChar :: IO Char
putChar :: Char -> IO ()
```

- `getChar`, when performed, reads a character from the standard input and returns it
- `putChar` takes a character and returns an action which, when performed, prints the character on the standard output
- An *action* is a first class value
- *Evaluating* an action has no effect; *performing* an action has an effect
- To combine actions, `(>>=)` (spoken *bind*) is provided

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

```
echo :: IO ()
echo = getChar >>= putChar
```

- `echo`, when performed, reads a character from the standard input and prints it to the standard output.
- When `a >>= f` is performed, it performs action `a`, takes the result, applies `f` to it to get a new action and then performs the new action
- In the `echo` example, we first perform the action `getChar`, yielding a character `c` and then we perform `putChar c`

- To combine two actions without using the result of the first, we construct (`>>`) (spoken `then`)

```
(>>) :: IO a -> IO b -> IO b
(>>) a1 a2 = a1 >>= (\x -> a2)

echoTwice :: IO ()
echo = echo >> echo
```

- (`>>`) is analogous to the semicolon (`;`) in (some) imperative programming languages
- It is common for the second argument of (`>>=`) to be an explicit lambda abstraction
- Example: `echoDup` reads a character and prints it twice

```
echoDup :: IO ()
echoDup = getChar >>= \c ->
          (putChar c >> putChar c)
```

- All the parentheses above are optional, since a lambda abstraction extends as far to the right as possible — so

```
echoDup :: IO ()
echoDup = getChar >>= \c ->
          putChar c >>
          putChar c
```

- This looks like a sequence of imperative actions and that is no coincidence — the `do` notation (see later) mirrors an imperative program more closely
- We need one more primitive to allow us to combine several values

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
              return (c1, c2)
```

- The action (`return v`) is an action that does no I/O and immediately returns `v` without any side effects

```
return :: a -> IO a
```

- (`return v`) lifts a value of type `a` into the `IO a` data type and does nothing else
- `getLine01` reads a whole line of input

```
getLine01 :: IO [Char]
getLine01 = getChar >>= \c ->
            if c == '\n' then
              return []
            else
              getLine01 >>= \cs ->
              return (c : cs)
```

- We use the name `getLine01` to not conflict with the builtin `getLine` which is defined as

```
getLine :: IO String
getLine = hGetLine stdin

hGetLine :: Handle -> IO String
```

- `hGetLine` is more general and efficient — it also does some error checking – see `System.IO`

- A complete Haskell program defines a single I/O action of type `IO ()`
- The program is executed by *performing* the action
- The following example reads a line, reverses it and prints the result

```
main :: IO ()
main = getLine    >>= \cs ->
      putLine (reverse cs)
```

- *Monadic I/O* can be thought of as composable action descriptions
- The essence of this style is the separation of the *composition calculations* from the composed action's *execution timeline*
- Note that `(>>=)` is the only (primitive) operation that combines or composes I/O actions
- There is no operator of the type `IO a -> a` — all we can do is feed the result of an action into another action
- This prevents the programmer bypassing the sequencing of actions

12.4 do Notation

- Haskell provides the `do` notation to re-write long chains of `(>>)` and `(>>=)`

```
do {e; stmts} = e >> do {stmts}
do {x <- e; stmts} = e >>= \x -> do {stmts}
do {e} = e
do {let decls; stmts}
  = let decls in do {stmts}
```

- Layout can be used to get rid of the braces `{ }`, `( )` and semicolons `;`
- This gives monadic computations an imperative feel
- Note that `x <- e` binds the variable `x` — it is not an assignment as in an imperative language

12.5 Control Structures

- Control structures such as `for` and `while` loops were invented in the 1960s as part of structured programming for imperative languages
- These required modifying the language
- However in functional programming we can build control structures out of functions in the language
- See [Control.Monad](#)
- See examples in [monad-loops: avoiding writing recursive functions by refactoring](#)
- See [Control.Monad.Loops](#)
- An infinite loop

```
forever :: IO () -> IO ()
forever a = a >> forever a
```


- Repeat an action a number of times

```
repeatN :: Int -> IO a -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

- A `for` loop

```
for :: [a] -> (a -> IO ()) -> IO ()
for [] f = return ()
for (n:ns) f = f n >> for ns f
```

- Instead of having a fixed collection of control structures provided by the language designer, we are free to invent new ones
- This is a very powerful technique
- Another definition of `for`

```
for ns f = sequence_ (map f ns)

sequence_ :: [IO a] -> IO ()
sequence_ as = foldr (>>) (return ()) as
```

- The `(_)` in the name `sequence_` reminds us that it throws away the results of the sub-actions

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:as)
  = do r <- a
      rs <- sequence as
      return (r:rs)
```

12.6 Interaction Exercises S0809 Q1

- Define `putLine01` which takes a `String` and prints the string with a new line at the end

```
555 putLine01 :: String -> IO ()
```

[Go to Interaction Exercises S0809 A 1](#)

12.7 Interaction Exercises S0809 A1

- We first define `putStr01`

```
557 putStr01 :: String -> IO ()
558 putStr01 [] = return ()
559 putStr01 (x : xs)
560   = putChar x >>
561     putStr01 xs
```

- ... and just add a newline

```
563 putLine01 xs
564   = putStr01 xs >>
565     putChar '\n'
```

[Go to Interaction Exercises S0809 Q 1](#)

12.8 Monadic I/O Review

- A complete Haskell program is a single `IO ()` action called `main`
- Note that `GHCi` allows various expressions at the prompt (see the [GHC User Guide](#))
- Larger I/O actions are constructed by gluing together smaller actions with `(>>=)` and `return`
- An I/O action is a *first-class value*: it can be passed to a function as an argument or returned as the result of a function call; it can be stored in a data structure
- Because I/O actions are first-class values, it is easy to define new combinators in terms of existing ones.
- The Monadic data structure for I/O allows us to separate calculating values in the language from calculating effects to be performed outside the language

12.8.1 Monad Data Structure

- The Monad data structure is more generally useful and we will return to discuss its other uses in a later section
- A `monad` is a triple of a type constructor, `m` and two function `return` and `(>>=)` with types

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

- These must also satisfy the following laws

```
return x >>= f == f x    -- left unit
m >>= return == m        -- right unit
m1 >>= (\x -> m2 >>= (\y -> m3)) -- assoc.
== (m1 >>= (\x -> m2)) >>= (\y -> m3)
```

- The above laws can also be expressed in `do` notation which may their meaning more obvious
- The monad laws in `do` notation

```
do x0 <- return x
  f x0
==
do f x    -- left unit

do x <- m
  return x
==
do m      -- right unit

do x <- m1
  do y <- m2 x
    m3 y
==
do y <- do x <- m1
        m2 x
    m3 y
==
do x <- m1
  y <- m2 x
  m3 y    -- associativity
```

- The monad laws just describe how we expect imperative code to behave

```
skipAndGetA
= do unused <- getLine
  line <- getLine
  return line

skipAndGetB
= do unused <- getLine
  getLine
```

- We expect the above two to have the same behaviour
- Now use `skipAndGet`

```
main
= do answer <- skipAndGet
  putStrLn answer
```

- We expect this to be the same as

```
main
= do answer <- do unused <- getLine
  getLine
  putStrLn answer
```

- and applying associativity

```
main
= do unused <- getLine
  answer <- getLine
  putStrLn answer
```

13 Future Work

- Functional programming is having a significant impact on the mainstream
- Program construction with functions and expressions rather than commands and statements
- Functions are first-class citizens
- Higher order functions
- Powerful combining forms
- Function composition
- Lazy evaluation or non-strict semantics
- Strong polymorphic type system
- Recursion and recursion patterns
- Efficiency and pragmatic issues
- Languages such as Scala, Kotlin, Rust, Julia and others have many of these features
- Notice the interplay between ideas and particular languages and technologies

14 Web Sites & References

- Miran Lipovača: Learn You a Haskell (LYAH) ([Lipovača, 2011](#)) written when he was a student in Ljubljana, Slovenia, well written but has no exercises. [Online version](#)
- Graham Hutton: Programming in Haskell ([Hutton, 2016](#)) — aimed at beginners — does have sections on [Monoid](#), [Foldable](#), [Traversable](#), [Functor](#), [Applicative](#), [Monad](#) without being mathematical in the formal sense.
See also [Erik Meijer: C9 Lectures — Functional Programming Fundamentals](#)
- Richard Bird: Thinking Functionally with Haskell ([Bird, 2014](#)) — third edition of a classic text — concentrates on derivation and transformation of functions
- Richard Bird & Jeremy Gibbons: Algorithm Design with Haskell ([Bird and Gibbons, 2020](#)) — sequel to the previous book
- Simon Thompson: Haskell The Craft of Functional Programming ([Thompson, 2011](#)) — a lot more examples and sections on coping with error messages from GHC
- Christopher Allen & Julie Moronuki: Haskell Programming ([Allen and Moronuki, 2016](#)) [Web site](#) — more formal than LYAH and does have exercises
- O’Sullivan et al: Real World Haskell ([O’Sullivan et al., 2008](#)) [Web site](#) — practitioners book
- Hudak: The Haskell School of Expression ([Hudak, 2000](#)) — learning Haskell through multimedia and music (Hudak was a jazz musician)

Functional Programming Papers

- [Haskell](#)
- [Haskell Documentation](#)
- [Haskell 2010 Language Report](#)
- [Glasgow Haskell Compiler](#)
- [GHC User Guide](#)
- [GHC Prelude](#)
- A History of Haskell: Being Lazy with Class ([Hudak et al., 2007](#))
- Conception, Evolution, and Application of Functional Programming Languages ([Hudak, 1989](#))
- Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity ([Hudak and Jones, 1994](#))
- Why Functional Programming Matters ([Hughes, 1989](#))
- Haskore music notation –an algebra of music– ([Hudak et al., 1996](#))

References

- Adelson-Velskii, G M and E M Landis (1962). An algorithm for the organization of information. In *Doklady Akademii Nauk SSSR*, volume 146, pages 263–266. Translated from *Soviet Mathematics — Doklady*; 3(5), 1259–1263.
- Allen, Christopher and Julie Moronuki (2016). *Haskell Programming*. Gumroad. URL <http://haske11book.com>.
- Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460.
- Bird, Richard (2014). *Thinking Functionally with Haskell*. Cambridge University Press. ISBN 1107452643. URL <http://www.cs.ox.ac.uk/publications/books/functional/>.
- Bird, Richard and Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL <https://www.cambridge.org/core/books/algorithm-design-with-haskell/824BE0319E3762CE8BA5B1D91EEA3F52>.
- Bird, Richard and Phil Wadler (1988). *Introduction to Functional Programming*. Prentice Hall, first edition. ISBN 0134841972.
- Chiswell, Ian and Wilfrid Hodges (2007). *Mathematical Logic*. Oxford University Press. ISBN 0199215626.
- Copeland, B Jack, editor (2004). *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma*. Oxford University Press. ISBN 0198250800.
- Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2009). *Introduction to Algorithms*. MIT Press, third edition. ISBN 0262533057. URL <http://mitpress.mit.edu/books/introduction-algorithms>.
- Davis, Martin (1995). Influences of mathematical logic on computer science. In *The Universal Turing Machine A Half-Century Survey*, pages 289–299. Springer.
- Davis, Martin (2012). *The Universal Computer: The Road from Leibniz to Turing*. A K Peters/CRC Press. ISBN 1466505192.
- Dowsing, R.D.; V.J Rayward-Smith; and C.D Walter (1986). *First Course in Formal Logic and Its Applications in Computer Science*. Blackwells Scientific. ISBN 0632013087.
- Fulop, Sean A. (2006). *On the Logic and Learning of Language*. Trafford Publishing. ISBN 1412023815.
- Gibbons, Jeremy; Graham Hutton; and Thorsten Altenkirch (2001). When is a function a fold or an unfold? *Electronic notes in theoretical computer science*, 44(1):146–160.
- Halbach, Volker (2010). *The Logic Manual*. OUP Oxford. ISBN 0199587841. URL <http://logicmanual.philosophy.ox.ac.uk/index.html>.
- Halpern, Joseph Y; Robert Harper; Neil Immerman; Phokion G Kolaitis; Moshe Y Vardi; and Victor Vianu (2001). On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, pages 213–236.
- Hindley, J. Roger and Jonathan P. Seldin (1986). *Introduction to Combinators and λ -Calculus*. Cambridge University Press. ISBN 0521318394. URL <http://www-maths.swan.ac.uk/staff/jrh/>.

- Hindley, J. Roger and Jonathan P. Seldin (2008). *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press. ISBN 0521898854. URL <http://www-maths.swan.ac.uk/staff/jrh/>.
- Hodges, Wilfred (1977). *Logic*. Penguin. ISBN 0140219854.
- Hodges, Wilfred (2001). *Logic*. Penguin, second edition. ISBN 0141003146.
- Hopcroft, John E.; Rajeev Motwani; and Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition. ISBN 0-201-44124-1.
- Hopcroft, John E.; Rajeev Motwani; and Jeffrey D. Ullman (2007). *Introduction to Automata Theory, Languages, and Computation*. Pearson, third edition. ISBN 0321514483. URL <http://infolab.stanford.edu/~ullman/ialc.html>.
- Hopcroft, John E. and Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, first edition. ISBN 020102988X.
- Hudak, Paul (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3).
- Hudak, Paul (2000). *The Haskell School of Expression*. Cambridge University Press. ISBN 0-421-64408-9.
- Hudak, Paul; John Hughes; Simon Peyton Jones; and Phil Wadler (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55. ACM New York, NY, USA.
- Hudak, Paul and Mark P Jones (1994). Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. *Contract*, 14(92-C):0153.
- Hudak, Paul; Tom Makucevich; Syam Gadde; and Bo Whong (1996). Haskore music notation –an algebra of music–. *Journal of Functional Programming*, 6(3):465–484.
- Hughes, John (1989). Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107.
- Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372.
- Hutton, Graham (2016). *Programming in Haskell*. Cambridge University Press, second edition. ISBN 9781316626221. URL <http://www.cs.nott.ac.uk/~pszgmh/pih.html>.
- Lee, Gias Kay (2013). Functional Programming in 5 Minutes. Web. <http://gsklee.im>, URL <http://slid.es/gsklee/functional-programming-in-5-minutes>.
- Lemmon, Edward John (1965). *Beginning Logic*. Van Nostrand Reinhold. ISBN 0442306768.
- Lipovača, Miran (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press. ISBN 1593272839. URL <http://learnyouahaskell.com>.
- Manna, Zohar (1974). *Mathematical Theory of Computation*. McGraw-Hill. ISBN 0-07-039910-7.
- Marlow, Simon and Simon Peyton Jones (2010). Haskell Language and Library Specification. Web. URL http://www.haskell.org/haskellwiki/Language_and_library_specification.

- Miller, Bradley W. and David L. Ranum (2011). *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle Associates Inc, second edition. ISBN 1590282574. URL <http://interactivepython.org/courselib/static/pythonds/index.html>.
- O'Donnell, John; Cordelia Hall; and Rex Page (2006). *Discrete Mathematics Using a Computer*. Springer, second edition. ISBN 1846282411. URL <http://www.dcs.gla.ac.uk/~jtod/discrete-mathematics/>.
- Okasaki, Chris (1998). *Purely Functional Data Structures*. Cambridge University Press. ISBN 0-521-63124-6.
- O'Sullivan, Bryan; John Goerzen; and Donald Stewart (2008). *Real World Haskell*. O'Reilly, first edition. ISBN 0596514980. URL <http://book.realworldhaskell.org/>.
- Pelletier, Francis Jeffrey and Allen P Hazen (2012). A history of natural deduction. In Gabbay, Dov M; Francis Jeffrey Pelletier; and John Woods, editors, *Logic: A History of Its Central Concepts*, volume 11 of *Handbook of the History of Logic*, pages 341–414. North Holland. ISBN 0444529373. URL <http://www.ualberta.ca/~francisp/papers/PellHazenSubmittedv2.pdf>.
- Pelletier, Francis Jeffry (2000). A history of natural deduction and elementary logic textbooks. *Logical consequence: Rival approaches*, 1:105–138. URL <http://www.sfu.ca/~jeffpell/papers/pelletierNDtexts.pdf>.
- Rayward-Smith, V J (1983). *A First Course in Formal Language Theory*. Blackwells Scientific. ISBN 0632011769.
- Smith, Peter (2003). *An Introduction to Formal Logic*. Cambridge University Press. ISBN 0521008042. URL <http://www.logicmatters.net/ifl/>.
- Smullyan, Raymond M. (1995). *First-Order Logic*. Dover Publications Inc. ISBN 0486683702.
- Teller, Paul (1989a). *A Modern Formal Logic Primer: Predicate and Metatheory: 2*. Prentice-Hall. ISBN 0139031960. URL <http://tellerprimer.ucdavis.edu>.
- Teller, Paul (1989b). *A Modern Formal Logic Primer: Sentence Logic: 1*. Prentice-Hall. ISBN 0139031707. URL <http://tellerprimer.ucdavis.edu>.
- Thompson, Simon (1991). *Type Theory and Functional Programming*. Addison Wesley. ISBN 0201416670. URL <http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/>.
- Thompson, Simon (2011). *Haskell the Craft of Functional Programming*. Addison Wesley, third edition. ISBN 0201882957. URL <http://www.haskellcraft.com/craft3e/Home.html>.
- Tomassi, Paul (1999). *Logic*. Routledge. ISBN 0415166969. URL <http://emilkirkegaard.dk/en/wp-content/uploads/Paul-Tomassi-Logic.pdf>.
- van Dalen, Dirk (1994). *Logic and Structure*. Springer-Verlag, third edition. ISBN 0387578390.
- van Dalen, Dirk (2012). *Logic and Structure*. Springer-Verlag, fifth edition. ISBN 1447145577.

Author Phil Molyneux Written 17 May 2020 *Printed 16th May 2020*

Subject dir: `<baseURL>/OU/M269/M269TutorialNotes`

Topic path: `/M269TutorialExtension/M269TutorialExtension2019J/M269TutorialExtension2019J.pdf`