Functional Programming

M269 Extension Tutorial

Phil Molyneux

17 May 2020

Extension Tutorial

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

M269 Tutorial

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 LaTeX
  - Mac OS X
- *Learning style* — I read the manual before using the software (really)

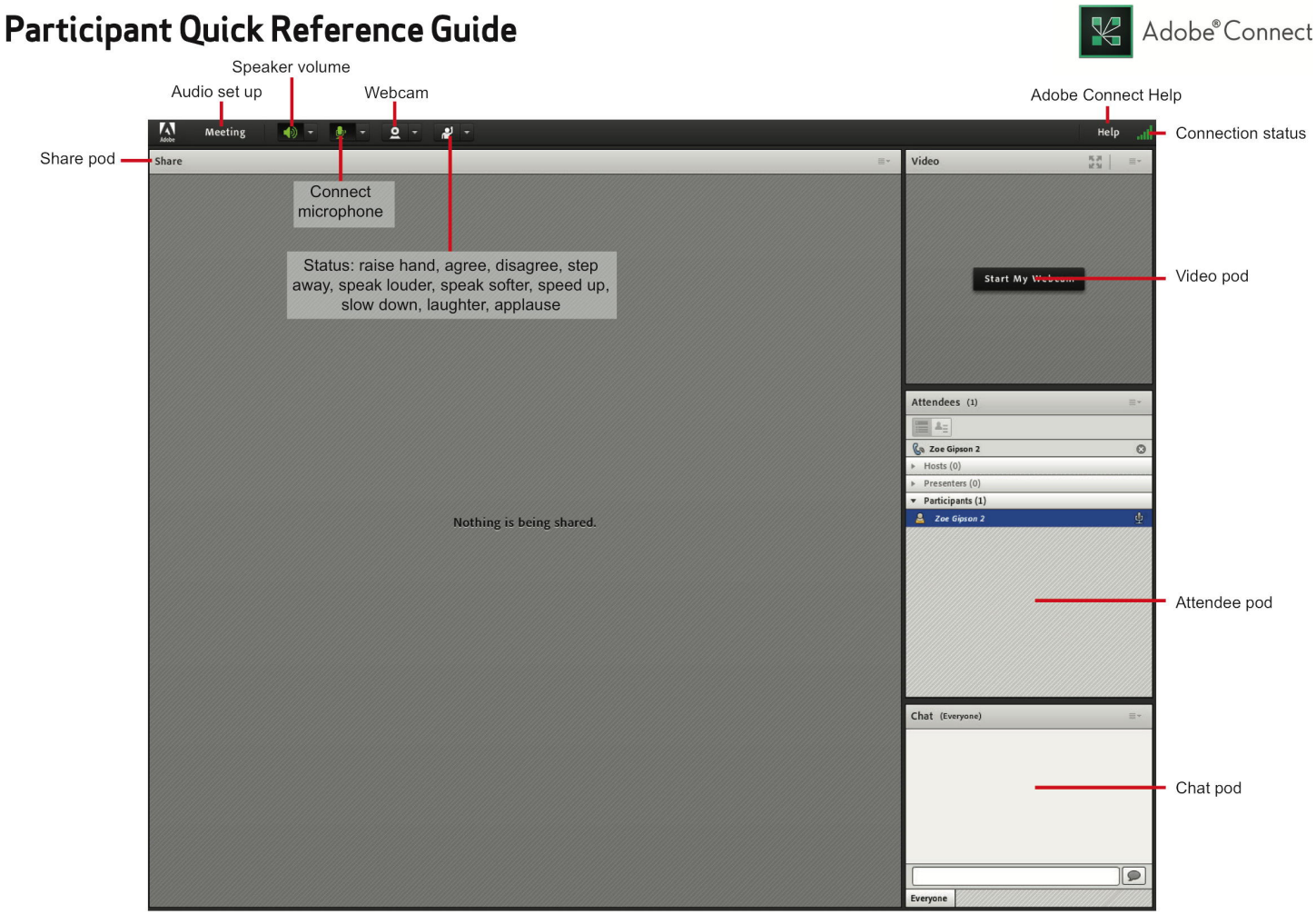
M269 Tutorial

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* ?

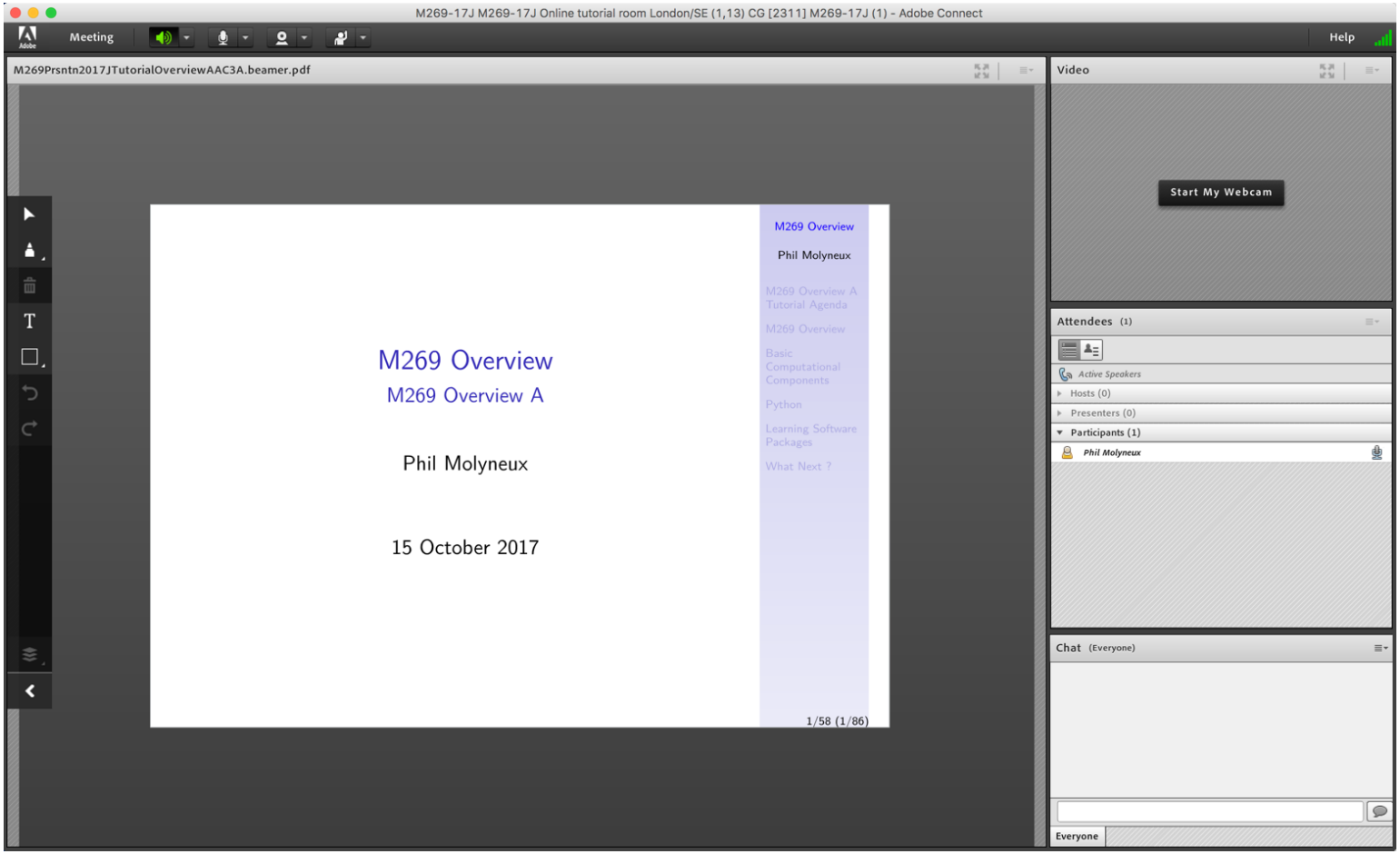
Adobe Connect

Interface — Student Quick Reference



Adobe Connect

Interface — Student View



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

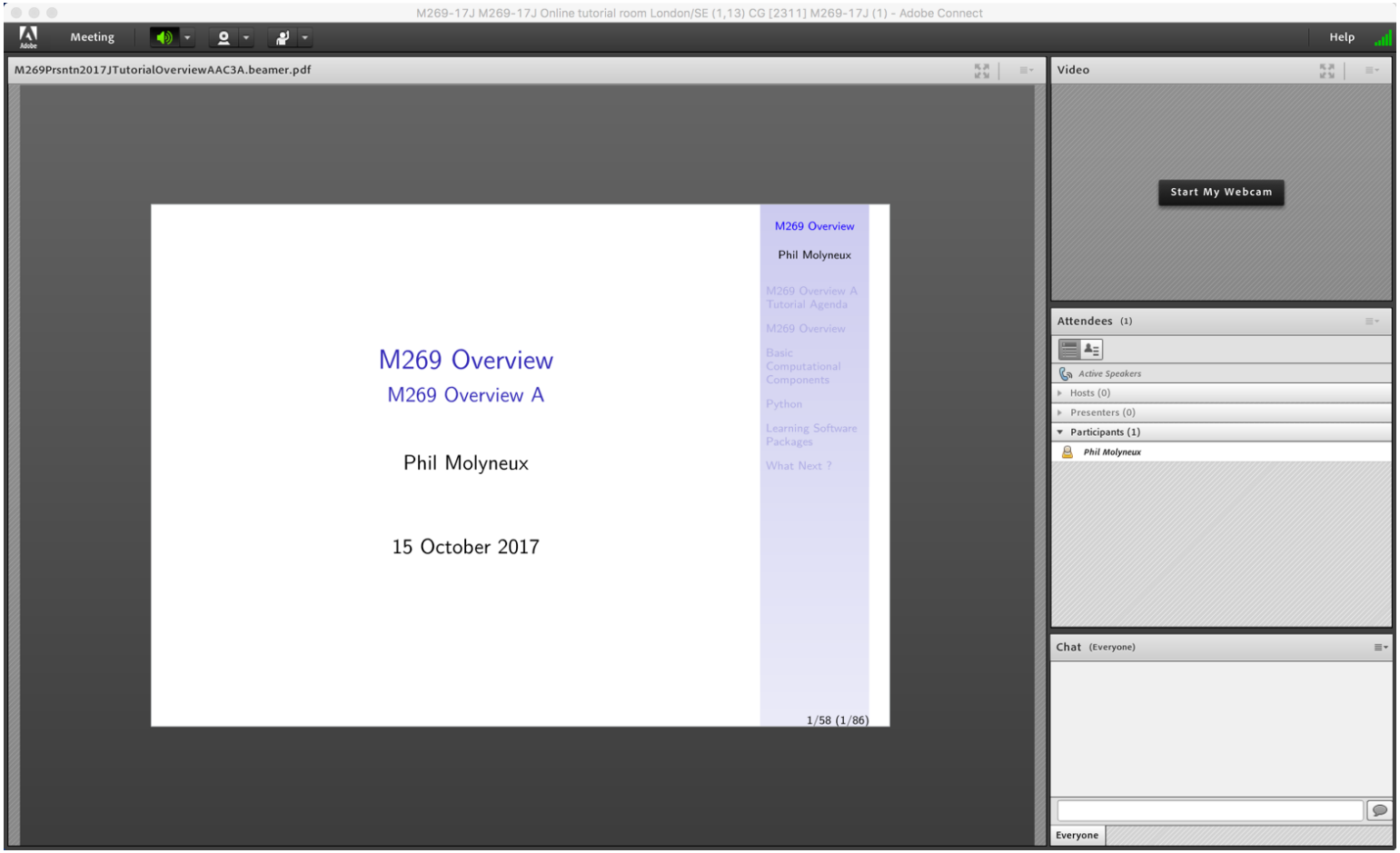
Adobe Connect

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** ⌘ + \

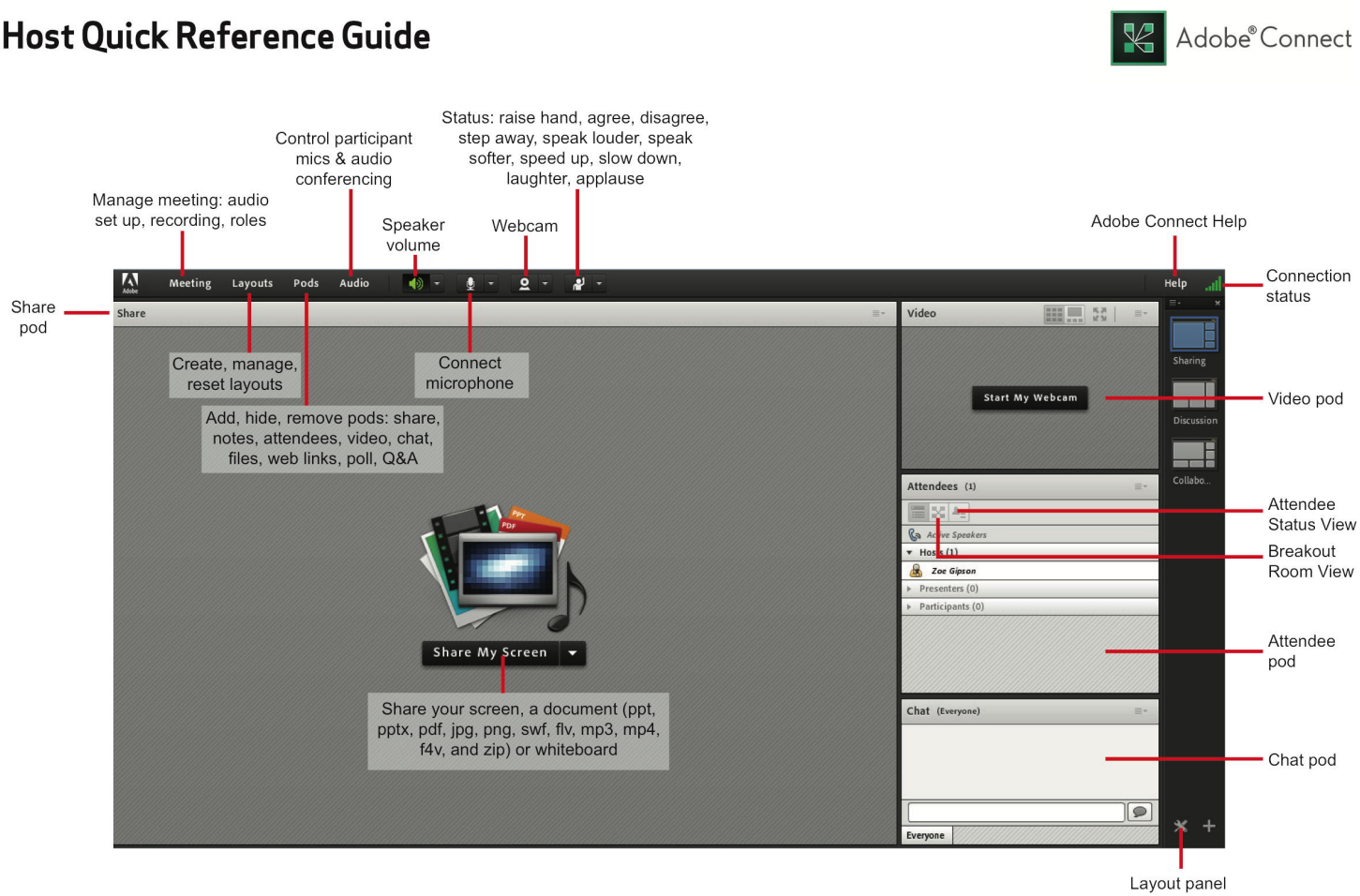
Adobe Connect Interface

Student View (default)



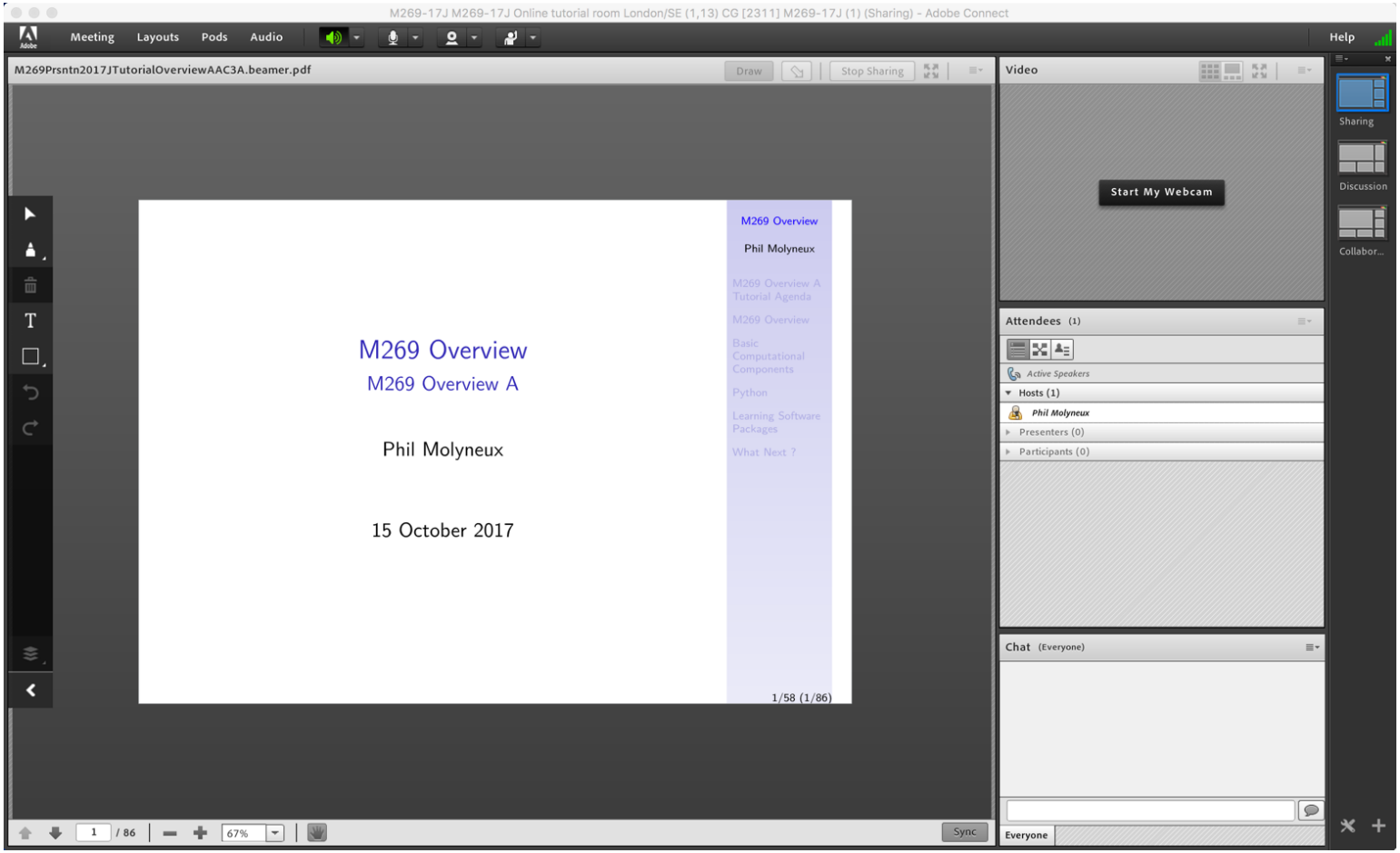
Adobe Connect Interface

Tutor View



Adobe Connect Interface

Tutor View



Adobe Connect Interface

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 blued 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

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

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

Adobe Connect

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

Adobe Connect

Layouts

- 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)

Haskell & GHC

Programming Environment

- 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)

Haskell & GHC

Script Setup

```haskell
module M269TutorialExtension2019J where
  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

Haskell & GHC

GHC Session

- 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

Haskell

Notation

- 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)

Haskell Notation

Types

- 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

Haskell Notation

Types

- 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)

Haskell

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

Types & Type Classes

Overview

- 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

Types & Type Classes

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`

Expressions & Types

Lists

- 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]
```

Expressions & Types

List Comprehensions

- **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

Expressions & Types

Arithmetic Sequences

- **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]

Type & Type Classes

Type Classes

- `Types` specify sets of elements or `data constructors`
- Primitive: Numbers, characters
- Builtin: Booleans, Lists, Tuples, and others
- 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

Type Classes

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

Haskell Standard Classes

Equality Class

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

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

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

Haskell Standard Classes

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)
```

Haskell Standard Classes

Ordered Class (contd)

```
  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

Haskell Standard Classes

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

Haskell Standard Classes

Enumeration Class (contd)

```
succ              = toEnum . (+1) . fromEnum
pred              = toEnum . (subtract 1) . fromEum
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)

Haskell Standard Classes

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
```

Haskell Standard Classes

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\""
```

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.

Function Definitions

Declaration Style

- **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

Function Definitions

Expression Style (1)

- **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)
```

Function Definitions

Expression Style (2)

- **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

Function Definitions

Let vs. Where

- **Let expression**

```
let
  decl1
  decl2
  ...
  declN
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

Evaluating Expressions

Applying a Function t Argumnents

- 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
```

Higher-order Functions

Map, Filter

- 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

Higher Order Functions

Map

- `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]
```

Higher Order Functions

Filter

- `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]
```

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

Haskell

- **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

List Comprehension Exercises

Activity 1 (a) Stop Words Filter

- Stop words are the most common words that most search engines avoid: `'a','an','the','that',...`
- 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

List Comprehension Exercises

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

Go to Answer

List Comprehension Exercises

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]]

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

List Comprehension Exercises

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

List Comprehension Exercises

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

List Comprehension Exercises

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

List Comprehension Exercises

Answer 1 (a) Stop Words Filter

- Answer 1 (a) Stop Words Filter
- *Write here:*
- Answer 1 continued on next slide

Go to Activity

List Comprehension Exercises

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

List Comprehension Exercises

Answer 1 (b) Transpose Matrix

- Answer 1 (b) Transpose Matrix
- *Write here:*
- Answer 1 continued on next slide

Go to Activity

List Comprehension Exercises

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

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
- Answer 1 continued on next slide

Go to Activity

List Comprehension Exercises

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

List Comprehension Exercises

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

List Comprehension Exercises

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

List Comprehension Exercises

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

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

List Comprehension Exercises

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

List Comprehension Exercises

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

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

List Comprehension Exercises

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

List Comprehension Exercises

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  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

Higher Order Functions

Foldr

- `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  product01 :: Num a => [a] -> a
51  product01 []     = 1
52  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
```

Higher Order Functions

Foldr (contd)

- `foldr` takes an operator `(⊗)`, a final value `⋆` and a list `xs`

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

- The operator `(⊗)` 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

Foldr

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

Further Foldr Examples (2)

- `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]
```

Foldr

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

User Defined Types and Classes

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

Algebraic Datatypes

Example: Days

- `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 `(*)`,`(/)`

Algebraic Datatypes

Example: Bool

- 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

Algebraic Data Types

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)
```

Algebraic Data Types

Standard Haskell Types (contd)

- 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
```

Algebraic Data Type Exercises S0607

Q 1

- 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 temerature

Go to Algebraic Data Type Exercises S0607 A 1

Algebraic Data Type Exercises S0607 Answers

A 1

```
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
```

- Soln 1 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 1

Algebraic Data Type Exercises S0607 Answers

A 1 (contd)

```
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

Algebraic Data Type Exercises S0607

Q 2

- 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]
```

- Q 2 continued on next slide

▸ Go to Algebraic Data Type Exercises S0607 A 2

Algebraic Data Type Exercises S0607

Q 2 (contd)

- 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

Algebraic Data Type Exercises S0607 Answers

A 2

- ### 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

Algebraic Data Type Exercises S0607

Q 3

- 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 maternal grandfather (if known)

```
160  paternalGrandfather01 :: Person -> Maybe Person
```

Go to Algebraic Data Type Exercises S0607 A 3

Algebraic Data Type Exercises S0607 Answers

A 3

▶ 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

Algebraic Data Type Exercises S0607

Q 4

- 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

Algebraic Data Type Exercises S0607 Answers

A 4

- 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)
```

- Soln 4 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 4

Algebraic Data Type Exercises S0607 Answers

A 4 (contd)

- 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
```

- Soln 4 continued on next slide

▸ Go to Algebraic Data Type Exercises S0607 Q 4

Algebraic Data Type Exercises S0607 Answers

A 4 (contd)

- We now rewrite the previous three functions:

```
194  maternalGrandfather02 p
195   = mother p >>= father

197  paternalGrandfather02 p
198   = father p >>= father

200  bothGrandfathers02 p
201   = father p >>=
202       (\dad -> father dad >>=
203         (\gf1 -> mother p >>=
204           (\mum -> father mum >>=
205             (\gf2 -> return (gf1,gf2)
206       ))))
```

- Soln 4 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 4

Algebraic Data Type Exercises S0607 Answers

A 4 (contd)

- 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
- Soln 4 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 4

Algebraic Data Type Exercises S0607 Answers

A 4 (contd)

- 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  }
```

- Soln 4 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 4

Algebraic Data Type Exercises S0607 Answers

A 4 (contd)

- 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 an implementation of the semicolon
- The `bind`, `(>>=)` operator is an 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

Algebraic Data Type Exercises S0607

Q 5

- 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

Algebraic Data Type Exercises S0607 Answers

A 5

- 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)
```

- Soln 5 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 5

Algebraic Data Type Exercises S0607 Answers

A 5

- 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       ))))
```

- Soln 5 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 5

Algebraic Data Type Exercises S0607 Answers

A 5

- 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)
```

- Soln 5 continued on next slide

Go to Algebraic Data Type Exercises S0607 Q 5

Algebraic Data Type Exercises S0607 Answers

A 5

- 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

Algebraic Data Type Exercises S0607

Q 6

- 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

Algebraic Data Type Exercises S0607 Answers

A 6

- Posible answer

```
272  maternalGrandfather :: Person -> Maybe Person
273  maternalGrandfather p = do
274    mum <- mother p
275    gfm <- father mum
276    return gfm

278  paternalGrandfather :: Person -> Maybe Person
279  paternalGrandfather p = do
280    dad <- father p
281    gfp <- father dad
282    return gfp

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

Return and Bind

Laws

- 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

Return and Bind

Laws (contd)

- 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
```

Return and Bind

Laws (contd)

- 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

Tree Data Types

Binary Trees and Recursion Schemes

- Binary trees appear in lots of applications and have common patterns of recursive definitions fr many functions

Tree Data Types

Binary Trees and Recursion Schemes

- In imperative, procedural programming, common patterns of control flow with GOTOs were astracted 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

Binary Trees

Data Types and Examples

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

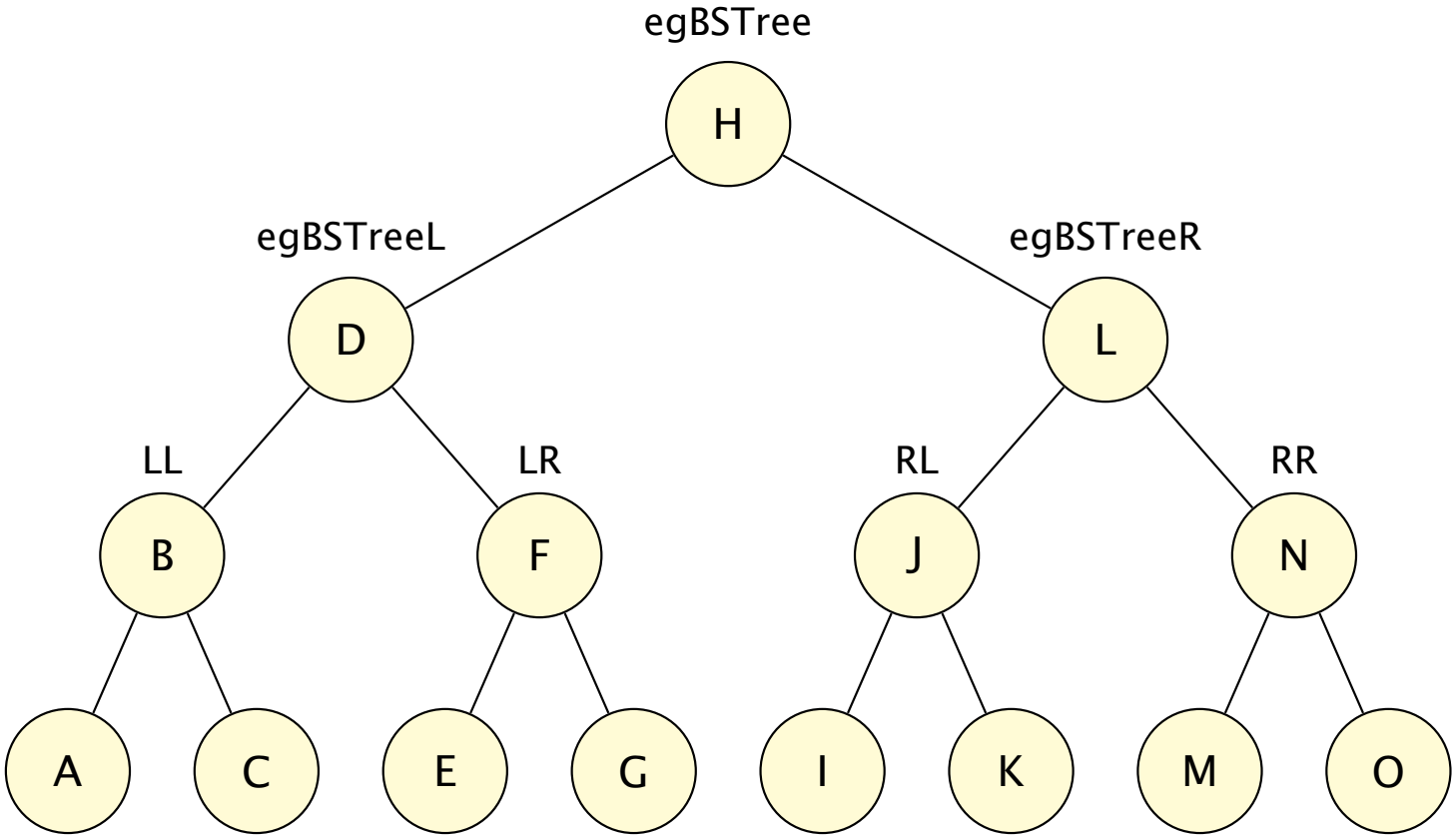
```
291 data BinTree a
292   = EmptyBT | NodeBT a (BinTree a) (BinTree a)
293     deriving (Eq, Show, Read)
```

- 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)
```

Binary Trees

Example egBSTree diagram



- 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

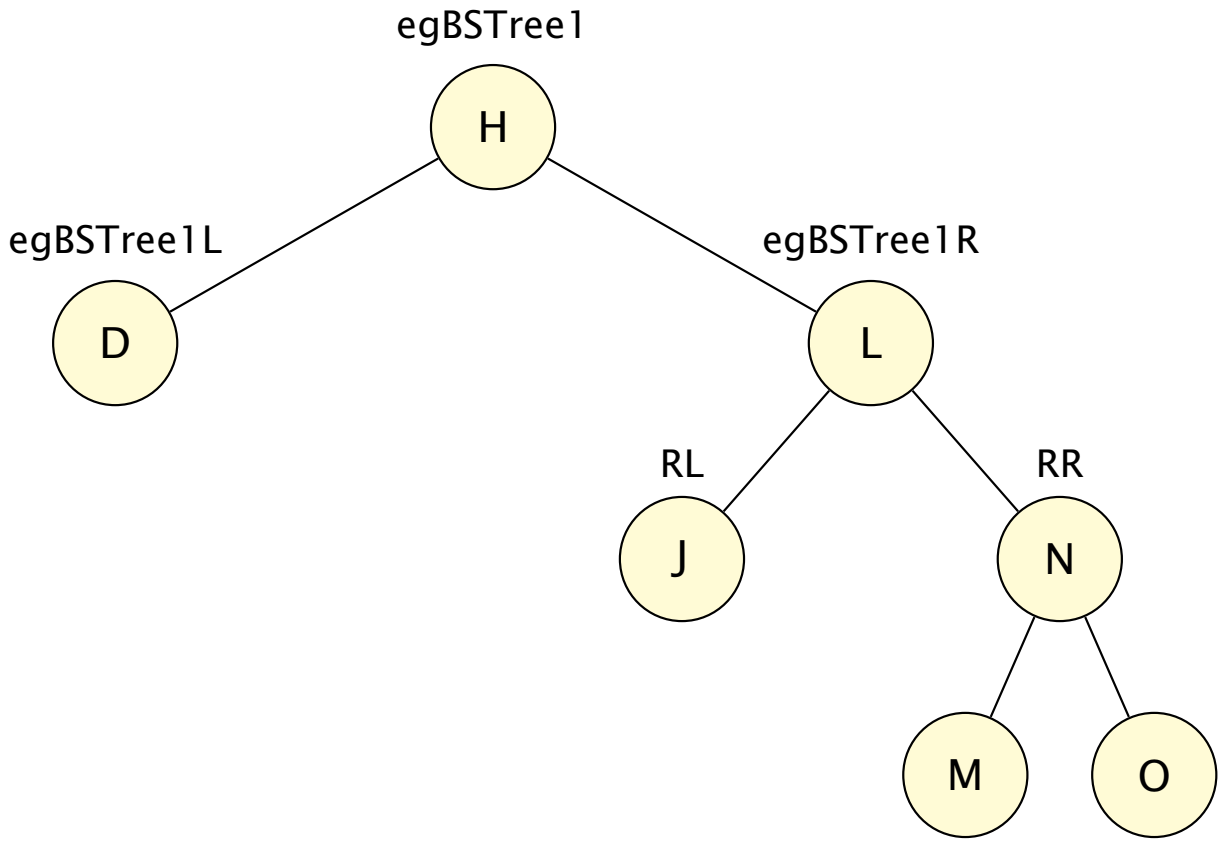
Binary Trees

Example egBSTree code

```
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        )
```

Binary Trees

Example egBSTree1 diagram



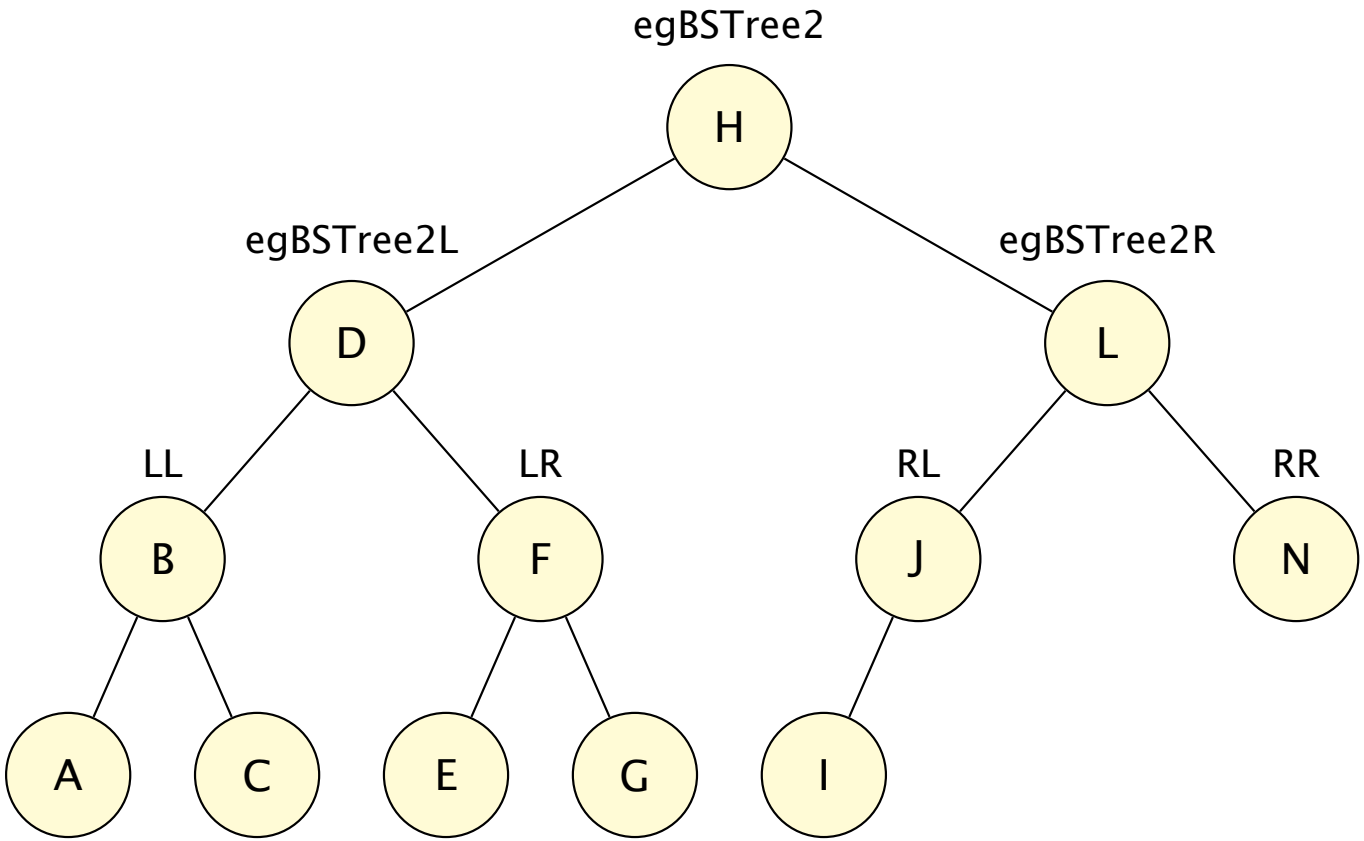
Binary Trees

Example egBSTree1 code

```
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)))
```

Binary Trees

Example egBSTree2



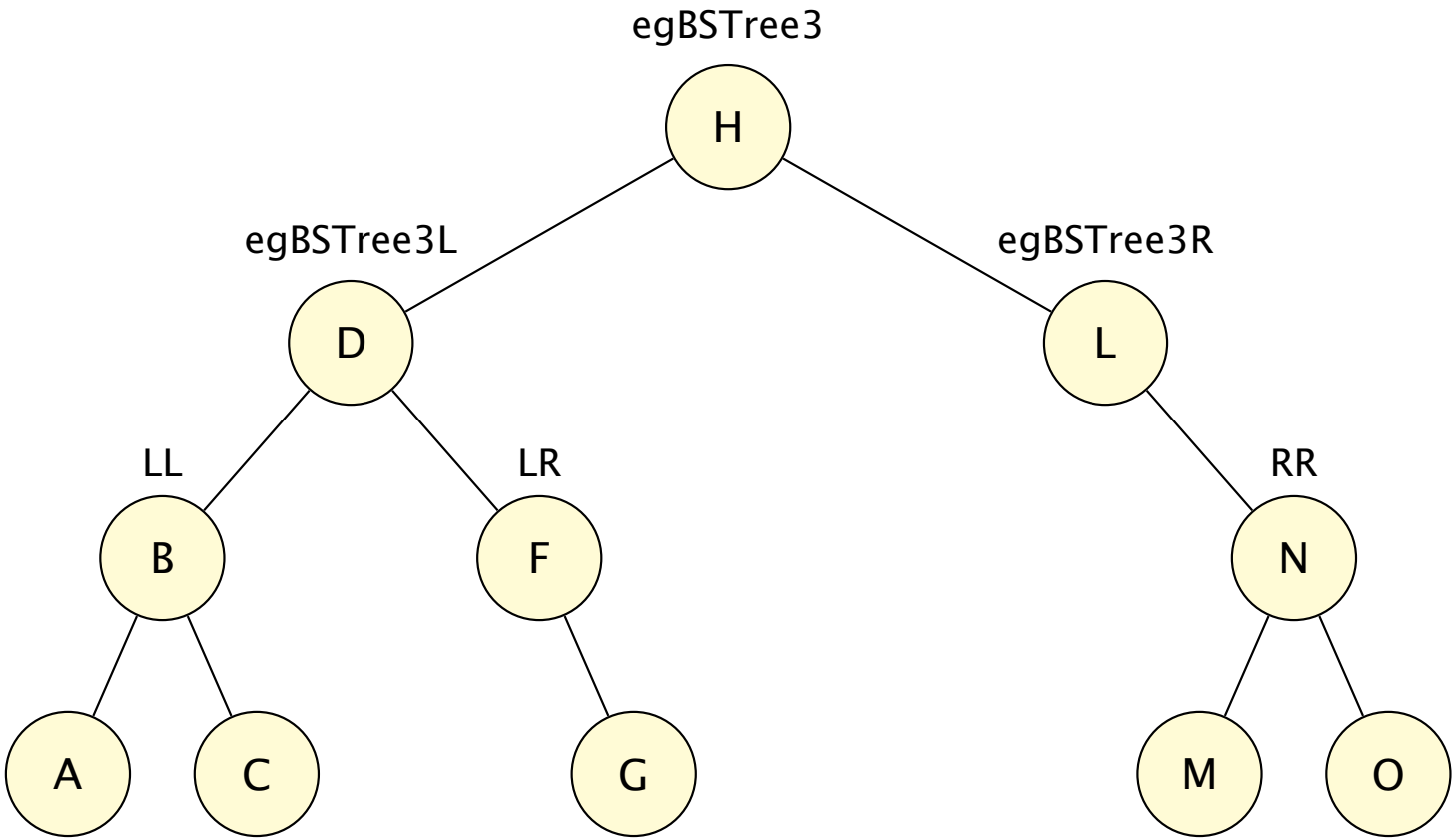
Binary Trees

Example egBSTree2 code

```
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))
```

Binary Trees

Example egBSTree3



Binary Trees

Example egBSTree3 code

```
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)))
```

Tree Types

Alternative Trees

- 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

```
361 data LeafTree a = LeafLT a
362                 | NodeLT a (LeafTree a) (LeafTree a)
363                 deriving (Eq, Show, Read)
364 data IntlTree a = LeafIT a
365                 | NodeIT a (IntlTree a) (IntlTree a)
366                 deriving (Eq, Show, Read)
367 data DualTree a b = LeafDT a
368                 | NodeDT a (DualTree b a) (DualTree b a)
369                 deriving (Eq, Show, Read)
370 data RoseTree a = LeafRT a [RoseTree a]
371                 deriving (Eq, Show, Read)
```

- Exercise: give an example of `DualTree Letter Integer`

Alternative Trees

Dual Trees Examples

- Examples of `DualTree Letter Integer`

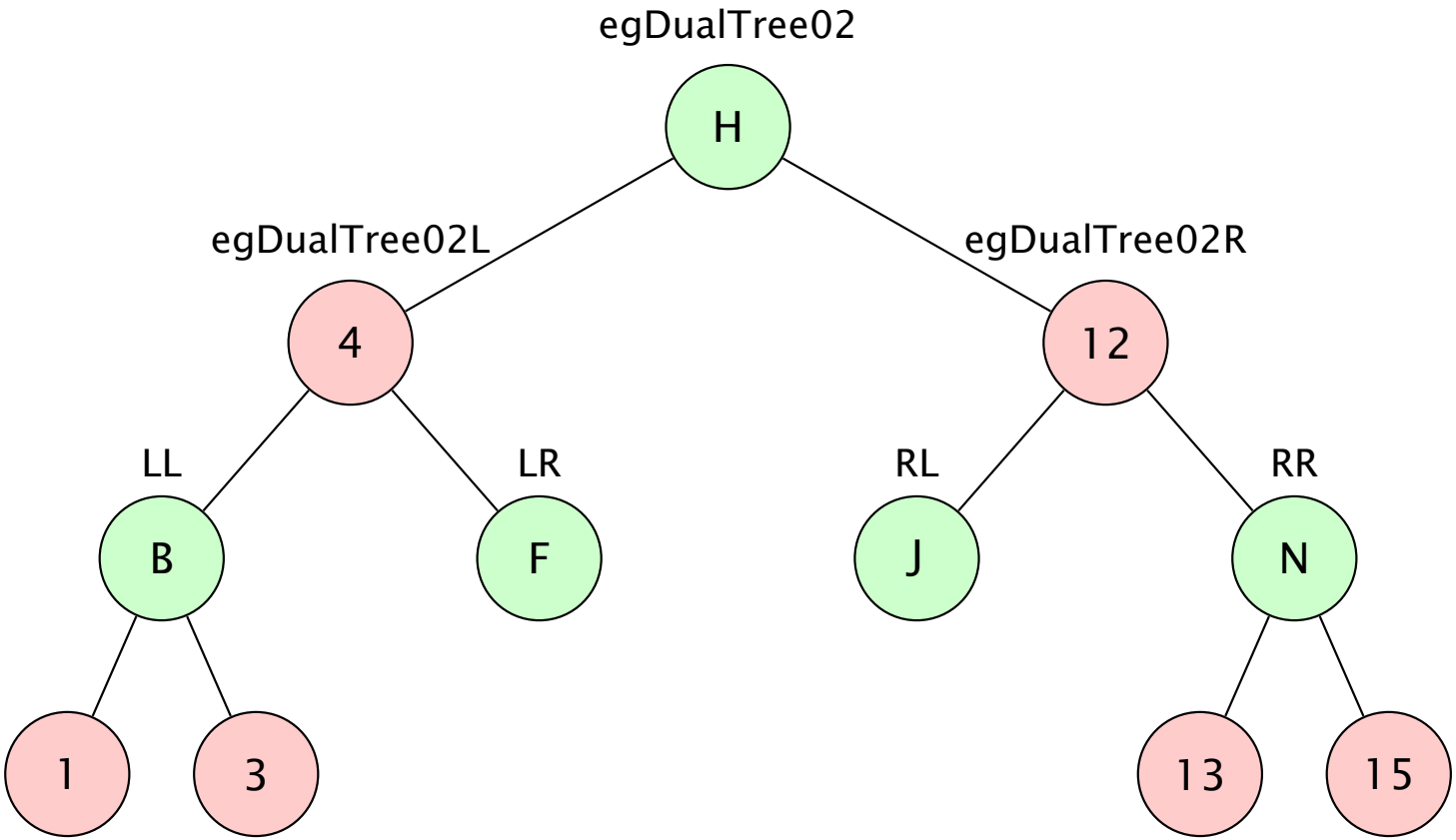
```
373 egDualTree01
374   = LeafDT H

376 egDualTree02
377   = NodeDT H
378       (NodeDT 4 (NodeDT B (LeafDT 1) (LeafDT 3))
379                 (LeafDT F))
380       (NodeDT 12 (LeafDT J)
381                  (NodeDT N (LeafDT 13) (LeafDT 15)))

383 egDualTree03
384   = NodeDT 8
385       (NodeDT D (NodeDT 2 (LeafDT A) (LeafDT C))
386                 (LeafDT 6))
387       (NodeDT L (LeafDT 10)
388                 (NodeDT 14 (LeafDT M) (LeafDT O)))
```

Dual Trees

Example egDualTree02



Tree Data Type Exercises

Introduction

- 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)

Tree Data Type Exercises S0607

Q 1

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

```
390 inOrderBT01   :: BinTree a -> [a]

392 preOrderBT01  :: BinTree a -> [a]

394 postOrderBT01 :: BinTree a -> [a]
```

Go to Tree Exs S0607 A 1

Tree Data Type Exercises S0607 Answers

A 1

- Here are the usual recursive definitions

```
395 inOrderBT01 EmptyBT = []
396 inOrderBT01 (NodeBT x leftBT rightBT)
397  = (inOrderBT01 leftBT)
398    ++ [x] ++ (inOrderBT01 rightBT)

400 preOrderBT01 EmptyBT = []
401 preOrderBT01 (NodeBT x leftBT rightBT)
402  = [x]
403    ++ (preOrderBT01 leftBT) ++ (preOrderBT01 rightBT)

405 postOrderBT01 EmptyBT = []
406 postOrderBT01 (NodeBT x leftBT rightBT)
407  = (postOrderBT01 leftBT)
408    ++ (postOrderBT01 rightBT) ++ [x]
```

- Soln 1 continued on next slide

Go to Tree Exs S0607 Q 1

Tree Data Type Exercises S0607 Answers

A 1 (contd)

- 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

```
410  foldBinTree
411   :: (a -> b -> b -> b) -> b -> BinTree a -> b

413  foldBinTree fNodeBT fEmptyBT EmptyBT = fEmptyBT
414  foldBinTree fNodeBT fEmptyBT (NodeBT x leftT rightT)
415    = fNodeBT x (foldBinTree fNodeBT fEmptyBT leftT)
416                (foldBinTree fNodeBT fEmptyBT rightT)
```

- Soln 1 continued on next slide

Go to Tree Exs S0607 Q 1

Tree Data Type Exercises S0607 Answers

A 1 (contd)

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

```
418 inOrderFoldBT :: BinTree a -> [a]
419 inOrderFoldBT t
420   = foldBinTree
421       fNodeBTToInOrderList fEmptyBTToInOrderList t

423 fEmptyBTToInOrderList = []
424 fNodeBTToInOrderList x leftTList rightTList
425   = leftTList ++ [x] ++ rightTList
```

- Soln 1 continued on next slide

Go to Tree Exs S0607 Q 1

Tree Data Type Exercises S0607 Answers

A 1 (contd)

```
427  preOrderFoldBT :: BinTree a -> [a]
428  preOrderFoldBT t
429    = foldBinTree
430        fNodeBTToPreOrderList fEmptyBTToPreOrderList t

432  fEmptyBTToPreOrderList = []
433  fNodeBTToPreOrderList x leftTList rightTList
434    = [x] ++ leftTList ++ rightTList

436  postOrderFoldBT :: BinTree a -> [a]
437  postOrderFoldBT t
438    = foldBinTree
439        fNodeBTToPostOrderList fEmptyBTToPostOrderList t

441  fEmptyBTToPostOrderList = []
442  fNodeBTToPostOrderList x leftTList rightTList
443    = leftTList ++ rightTList ++ [x]
```

▸ Go to Tree Exs S0607 Q 1

Tree Data Type Exercises S0607

Q 2

- 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:

```
445  levelOrderBT :: BinTree a -> [[a]]
446  levelOrderBT EmptyBT = []
447  levelOrderBT (NodeBT x leftT rightT)
448    =  [x] : longZipWith (++)
449             (levelOrderBT leftT)
450             (levelOrderBT rightT)

452  longZipWith :: (a -> a -> a) -> [a] -> [a] -> [a]
453  longZipWith f [] ys      = ys
454  longZipWith f (a:xs) []  = (a:xs)
455  longZipWith f (a:xs) (b:ys)
456   = (f a b) : (longZipWith f xs ys)
```

- Define *level order* as a fold

Go to Tree Exs S0607 A 2

Tree Data Type Exercises S0607 Answers

A 2

```
458  levelOrderFoldBT :: BinTree a -> [[a]]
459  levelOrderFoldBT t
460   = foldBinTree
461      fNodeBTToLevelOrder fEmptyBTToLevelOrder t

463  fEmptyBTToLevelOrder = []

465  fNodeBTToLevelOrder :: a -> [[a]] -> [[a]] -> [[a]]
466  fNodeBTToLevelOrder x leftTOrder rightTOrder
467   = [x] : longZipWith (++)
468      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

Tree Data Type Exercises S0607

Q 3

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

```
469  heightBT :: BinTree a -> Int

471  heightBT EmptyBT = 0
472  heightBT (NodeBT x leftT rightT)
473   = 1 + max (heightBT leftT) (heightBT rightT)
```

Go to Tree Exs S0607 A 3

Tree Data Type Exercises S0607 Answers

A 3

```
474 heightFoldBT t
475  = foldBinTree
476      fNodeBTToHeight fEmptyBTToHeight t

478 fEmptyBTToHeight = 0
479 fNodeBTToHeight x leftTHeight rightTHeight
480  = 1 + max leftTHeight rightTHeight
```

```
GHCi> heightBT egBSTree
4
GHCi> heightFoldBT egBSTree
4
```

▸ Go to Tree Exs S0607 Q 3

Tree Data Type Exercises S0607

Q 4

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

```
481  sizeBT :: BinTree a -> Int

483  sizeBT EmptyBT = 0
484  sizeBT (NodeBT x leftT rightT)
485   = 1 + (sizeBT leftT) + (sizeBT rightT)
```

Go to Tree Exs S0607 A 4

Tree Data Type Exercises S0607 Answers

A 4

```
486  sizeFoldBT t
487   = foldBinTree
488      fNodeBTToSize fEmptyBTToSize t

490  fEmptyBTToSize = 0
491  fNodeBTToSize x leftTSize rightTSize
492   = 1 + leftTSize + rightTSize
```

```
GHCi> sizeBT egBSTree
15
GHCi> sizeFoldBT egBSTree
15
```

Go to Tree Exs S0607 Q 4

Tree Data Type Exercises S0607

Q 5

- Write a function numLeavesBT which takes a tree and returns the number of leaves

  a leaf is a node with two empty subtrees

```
493  isEmptyBT EmptyBT = True
494  isEmptyBT (NodeBT x leftT rightT) = False

496  isBothEmptyBT t1 t2
497   = isEmptyBT t1 && isEmptyBT t2

499  numLeavesBT EmptyBT = 0
500  numLeavesBT (NodeBT x leftT rightT)
501   = if isBothEmptyBT leftT rightT
502     then 1
503     else numLeavesBT leftT
504          + numLeavesBT rightT
```

- Write numLeavesFoldBT which uses foldBinTree

Go to Tree Exs S0607 A 5

Tree Data Type Exercises S0607 Answers

A 5

- 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 righT)
 -- Eureka step to get rid of isolated leftT, righT
 = if isBothZero (numLeavesBT leftT) (numLeaves righT)
   then 1
   else (numLeavesBT leftT)
        + (numLeaves righT)
 -- this gives us the required definition
```

- Soln 5 continued on next slide

Go to Tree Exs S0607 Q 5

Tree Data Type Exercises S0607 Answers

A 5 (contd)

```
505  isBothZero x y
506   = x == 0 && y == 0

508  numLeavesFoldBT t
509   = foldBinTree
510      fNodeBTToNumL fEmptyBTToNumL t

512  fEmptyBTToNumL = 0
513  fNodeBTToNumL x leftTNL rightTNL
514   = if isBothZero leftTNL rightTNL
515     then 1
516     else leftTNL + rightTNL
```

▸ Go to Tree Exs S0607 Q 5

Tree Data Type Exercises S0607

Q 6

- The function `minDepthBT` can be defined recursively as

```
517 minDepthBT EmptyBT = 0
518 minDepthBT (NodeBT x leftT rightT)
519  = 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

Tree Data Type Exercises S0607 Answers

A 6

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

```
520 minDepthBT01 :: BinTree a -> Int
521 minDepthBT01 t
522  = minD t 0 maxBound
523 -- here maxBound is regarded as infinity
524 minD :: BinTree a -> Int -> Int -> Int
525 minD EmptyBT d m = min d m
526 minD (NodeBT x leftT rightT) d m
527  = if d + 1 >= m
528    then m
529    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

Recursion Schemes

References (1)

- 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)

Recursion Schemes

References (2)

- 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

Input and Output

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

Input and Output

The Problem (2)

- 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

Input and Output

Initial 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
              | ...
```

Input and Output

Initial Solution (2)

- 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

Input and Output

Monadic I/O (1)

- 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`

Input and Output

Monadic I/O (2)

- 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

Input and Output

Monadic I/O (3)

- 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`

Input and Output

Monadic I/O (4)

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

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

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

- (>>) is analogous to the semicolon (;) in (some) imperative programming languages

Input and Output

Monadic I/O (5)

- 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

Input and Output

Monadic I/O (6)

- 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

Input and Output

Monadic I/O (7)

- `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

Input and Output

Monadic I/O (8)

- 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)
```

Input and Output

Monadic I/O (9)

- *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

Monadic I/O

do Notation

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

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

- 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

Monadic I/O

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

Monadic I/O

Control Structures (2)

- 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
```

Monadic I/O

Control Structures (3)

- 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 contol structures provided by the language designer, we are free to invent new ones
- This is a very powerful technique

Monadic I/O

Control Structures (4)

- 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)
```

Interaction Exercises S0809

Q 1

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

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

Go to Interaction Exercises S0809 A 1

Interaction Exercises S0809 Answers

A 1

- We first define `putStr01`

```
533  putStr01 :: String -> IO ()
534  putStr01 [] = return ()
535  putStr01 (x : xs)
536   = putChar x >>
537     putStr01 xs
```

- . . . and just add a newline

```
539  putLine01 xs
540   = putStr01 xs >>
541     putChar '\n'
```

Go to Interaction Exercises S0809 Q 1

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

Monads

Monadic 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

Monads

Monadic Data Structure (2)

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

Monads

Monadic Data Structure (3)

- 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

Monads

Monadic Data Structure (4)

- 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
```

Future Work

Topics

- 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 technoloies

Haskell References

Textbooks

- Miran Lipovača: Learn You a Haskell (LYAH) (Lipovaca, 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, 2020) — sequel to the previous book

Haskell References

Textbooks (2)

- 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, 2008) — learning Haskell through multimedia and music (Hudak was a jazz musician)

Haskell References

Functional Programming Papers & Reference

- 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, 1996)