Week 5

Ch 7: Making Our Own Types and Type Classes
Ch 8: Input and Output

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Overview

Chapter 7

- Data Types
- Nesting Types (7--12)
- Record Syntax (13--15)
- Type Parameters (16--28)
- Example 3D Vector (29--33)
- Derived Instances (34--49)
- Type Synonyms (50--61)
- Recursive Data Structures (62--69)
- Example Tree Type (70--101)
- The Functor Type Class (102--115)
- Kinds (116--119)

Chapter 8

- Separating Pure from Impure (121--125)
- Gluing I/O Actions Together (126--134)
- Reverse Strings in I/O (135--139)
- Demos of Some I/O Action Functions (140--154)

2

Creating a `Data` Type

Defining our own type follows the syntax for what could be the `Bool` type:

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

- `data` keyword, followed by the capitalized name of the type
- equal sign
- capitalized value constructors separated by "or" Sheffer stroke `|`

3

Multiple Values

We want a Shape type to have both circles or rectangles.

Then the type could be defined as:

```
{- multiline in ghci, otherwise scripts are forgiving -}
:{
data Shape =
    Circle Float Float Float
    | Rectangle Float Float Float Float
:}
```

- the uses of `Float` here act as constructor parameters
- `Circle` and `Rectangle` act like constructor functions
- try out the type description/signature for each `:t Circle`
- notice that they return type `Shape`

4

Constructor Pattern Matching

A function that takes a `Shape` and calculates its area:

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

- we cannot write the function as `Circle -> Float`
  - incorrect as `True -> Int`
- `Circle` is defined as a ***value*** and `Shape` is its type
  - `Circle` constructor function has the same name
- we can pattern match with a constructor and its parameters
- circle needs no position to calculate its area, so `_` is used

5

Deriving Type Class **Show**

To declare a type as an instance of the `Show` type class:

```
:{
data Shape =
        Circle Float Float Float
        | Rectangle Float Float Float Float deriving (Show)
:}
```

- then we can create values of Shape:

```
Circle 1 2 3
Rectangle 10 20 30 40
```

- and partially apply constructor functions:

```
map (Circle 10 20) [4,5,6,6]
```

6

— Nested Types —

7

Creating Nested Types

Make `Circle` type convey more with a nested `Point` type:

```
data Point = Point Float Float deriving (Show)
data Shape =
        Circle Point Float
        | Rectangle Point Point deriving (Show)
```

- the `Point` type is repeated as the one value name
- then we use `Point` in the constructor definition of `Circle` and `Rectangle`
- pattern matching needs to be updated

```
area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) *
(abs $ y2 - y1)
```

8

Using Nested Constructors

```
area (Circle (Point 0 0) 24)

area . Rectangle (Point 0 0) $ Point 100 100
```

- a few reminders:
  - dot product composes functions (that take one parameter each)
  - `$` applies the function immediately after it (avoid writing parentheses)

9

Export from Modules

At this point we have enough to justify making our own `Shape` module (see `shape.hs` script).

- specify constructors we want to export in parentheses:
- If we just want all of the parameters then use `(..)`,
- shorter than specifying `Shape (Rectangle, Circle)`

```
module Shapes
( Point(..)
, Shape(..)
, area
, nudge
, baseCircle
, baseRect
) where
```

10

Private Code

Without `(..)`:

- a user could not create new shapes except with functions `baseCircle` and `baseRect`

- hiding constructors makes the `Shape` type more abstract
- might be good if we want to stop users from pattern matching with value constructors
- edits to the value constructors would not cascade (like we saw earlier with `Shape` and `Point`)
- we get back to this discussion later with `Data.Map`

Previously, we defined functions with the notation `->` between input parameters, but not so for constructors.

11

Use Function Constructors

Try out the functions after loading `shape.hs`:

```
:l shape.hs
nudge (baseRect 40 100) 60 23
nudge (baseCircle 8) 3 5
```

12

— Record Syntax —

13

Record Syntax

We can create types with a record syntax:

- the value constructor parameters are given meaningful names
- avoids creating nested types
- also creates corresponding functions, one for every single parameter
- call the constructor and specify parameters in any order we choose

For example:

```
data Car = Car {
    company :: String,
    model :: String,
    year :: Int
} deriving (Show)
```

Call the constructor:

```
Car {company = "Ford", model = "Mustang", year = 1967}
```

14

Parameter Order

Choose to use record syntax when the order of the fields do not immediately make sense.

- a 3D vector would be obvious
  - the fields specify the coordinates `x` `y` `z` values
- but, for `Car` parameter order is arbitrary

15

— Type Parameters —

16

Type Parameters

Similar to functions taking parameters, we can generate new types by passing types as parameters.

Consider Maybe as a type constructor:

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

Pass in a type for the parameter `a`, we generate a new type, such as:

- `Maybe Int`, `Maybe Car`, `Maybe String`, etc.
- `Maybe` is a type constructor, not to be used to create values
- a type constructor must have *all* parameters passed in

17

Concrete Types

- the value `Just 'a'` has type `Maybe Char`
- sometimes we want a value to be of more specific type
  - `Just 3` is at its most generic of type `Num a => Maybe a`
  - we can specify like so: `Just 3 :: Maybe Int`

General vs specific type also apply to list types:

- `[Int]`
- `[[String]]` (***triple*** nested list)
- we cannot have a value that has a type of just `[]`
- any value always has a concrete type associated
- a concrete type has no unspecified type parameters

18

Polymorphic Types

A more generic type such as `Maybe a` is polymorphic:

- `Maybe a` can manage different kinds of subtypes with type parameters `a`

19

Practice with Maybe

```
Just "Haha"
Just 84
:t Just "Haha"
:t Just 84
:t Nothing
Just 10 :: Maybe Double
```

20

Practice Defining Concrete Types

Examples of defining concrete types similar to `Maybe a` where `a` is replaced by concrete type might be:

```
data IntMaybe = INothing | IJust Int

data StringMaybe = SNothing | SJust String

data ShapeMaybe = ShNothing | ShJust Shape
```


21

Working with Polymorphic Types

Or we might design something to be as entirely generic.

- `Maybe a` has type parameter `a` so that it could handle working with values of anything at all
- similarly, the type for empty list is `[a]`
- again, `a` is any type element so the empty list `[]` can operate with any other type of list:

```
[1,2,3] ++ []
["har","har","har"] ++ []
```

22

Generalizing Types

The `year` field of the `Car` type can be parameterized:

```
data Car a = Car {
    company :: String,
    model :: String,
    year :: a
} deriving (Show)
```

- have the above either in a script without `let`
- or in ghci give the definition all on one line without `let`
- or use multiline `:{ :}` and do not use `let`

23

Creating Functions that use Generic Types

A function to display the information of a `Car` would change from our previous definition of `Car`:

```
tellCar :: (Show a) => Car a -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

(compare signature without generic `tellCar :: Car -> String`)

Try out `tellCar` in ghci:

```
tellCar (Car {company = "Tesla", model = "Roadster", year = 2022 })
```

24

Polymorphism

This second version of `tellCar` allows us to work with various instance types of `Show`:

```
tellCar (Car "Ford" "Mustang" 1967)
tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
:t Car "Ford" "Mustang" 1967
:t Car "Ford" "Mustang" "nineteen sixty seven"
```

- we would likely only ever use the version of `tellCar` that has a year with `Int` type
- so, parameterizing is not worth the trouble in this case

25

Book Pattern Matching (Simranjit Singh)

```
data Book = Book {
      title :: String
    , author :: String
    , pageNum :: Int
    }
    deriving (Show)

Book { title = "Fairy Tale", author = "Stephen King", pageNum =  600 }

tellBook :: Book -> String
tellBook (Book {title = t, author = a, pageNum = p}) =
    t ++ " was written by " ++ a ++ ". It's " ++ show (p) ++ " pages long."
```

26

Conventions of Type Parameters

Notice the generic types that use type parameters:

- have little need in their implementation for anything with respect to the type parameters

- e.g.: we would only do things with a list itself that has nothing to do directly with the type of its elements
- anything we would do with elements, such as a `sum`, we can specify its implementation when we specify the concrete type
- the same goes for the `Maybe`
  - it allows us to specify an implementation when we need to deal with potentially not having a value of a concrete type we want
  - (`Nothing`)
  - or having it (`Just x`)

27

Class Constraints on Data Declarations

Type class constraints should not be in the data declarations of our parameterized types.

- functions which would use the parameterized types
  - then have to declare the type-class constraints regardless

28

— Example 3D Vector —

29

3D Vector

An example for implementing our own vector 3D data type with one parameterization:

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num a) => Vector a -> Vector a -> Vector a
(Vector i j k) `vplus` (Vector p q r) = Vector (i+p) (j+q) (k+r)

dotProd :: (Num a) => Vector a -> Vector a -> a
(Vector i j k) `dotProd` (Vector p q r) = (i*p) + (j*q) + (k*r)

vmult :: (Num a) => Vector a -> a -> Vector a
(Vector i j k) `vmult` p = Vector (i*p) (j*p) (k*p)
```

30

Matching Type Parameters

We restrict the vector functions for the parameter to be of type class `Num`, since we could not expect calculations where components are of type `Bool` nor `Char`.

- also notice that the definitions restrict only vectors of the same element concrete types to calculate together
  - cannot add vectors with one of type `Int` and the other `Double`
- notice no `Num` restriction in the type declaration of `Vector`
  - still need the same restrictions of type class in the functions anyway

31

Type vs Value Constructors

Note the difference between type constructors and value constructors:

- type constructors appear *on the left* of `=` in type definitions
- value constructors appear *on the right* of `=` in type definitions
- in function declarations
  - use type constructors where types would go
  - use value constructors in place of a pattern or value
- the following would not be possible with our `Vector` as defined:

```
Vector a a a -> Vector a a a -> a
```

- the type constructor is `Vector a` and not `Vector a a a`

32

Vector Functions

Give some vector functions a try:

```
Vector 3 5 8 `vplus` Vector 9 2 8
Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 3 9 7 `vmult` 10
Vector 4 9 5 `dotProd` Vector 9.0 2.0 4.0
Vector 2 9 3 `vmult` (Vector 4 9 5 `dotProd` Vector 9 2 4)
```


33

— Derived Instances —

34

Derived Instances

Type classes are like interfaces in Java:

- any type within the type class is considered an instance of it
- the type class specifies what kind of behaviour must be implemented in any type belonging to it
  - the type class has no implementation itself

We can take advantage of the type classes that already exist in Haskell:

* `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`

35

Deriving Keyword

The `deriving` keyword allows us to do so:

```
data Person = Person {
    firstName :: String,
    lastName :: String,
    age :: Int
} deriving (Eq)
```

- Haskell compares two Person values to see if their value constructors are the same
  - then checks if the corresponding fields also have equal values
- so, all fields must be types also each an instance of `Eq`

36

Using `==` on Eq Instances

We can test using `==` on values of our `Person` type:

```
mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
```

Give equality tests a try:

```
mca == adRock
mikeD == adRock
mikeD == mikeD
mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
```


37

Using Behaviour of Class Type Instances

Additionally, we can use `Person` values with anything that takes `Eq` as a type class constraint, e.g.: `elem` function:

```
let beastieBoys = [mca, adRock, mikeD]
mikeD `elem` beastieBoys
```

38

— `Show` and `Read` —

39

`Show` and `Read` Derived Instances

We can make `Person` type

- become an instance of `Show` and `Read` type classes in order to convert values between strings and back
- again, we need the fields to also be of a type that are instances of `Show` and `Read`

```
data Person = Person {
    firstName :: String,
    lastName :: String,
    age :: Int
} deriving (Eq, Show, Read)
```

- try printing some of the values:

```
mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
"mikeD is: " ++ show mikeD
```

40

Convert Between String and Back

If we tried to print without the `Show` derivation, then Haskell would give us an error message.

We can convert back the other direction and get a `Person` value from a `String`.

- put the following in a script, then load in ghci:

```
mysteryDude =
    "Person { firstName =\"Michael\"" ++
    ", lastName =\"Diamond\"" ++
    ", age = 43}"
```

Give a type annotation to tell Haskell what concrete type it should evaluate:

```
read mysteryDude :: Person
```


41

Type Annotations with Concrete Types

Type annotation is not needed if Haskell can infer the type:

```
read mysteryDude == mikeD
```

- if the type requires further specification with a parameter type, then we also need to annotate that
  - the following will give an error:

```
read "Just 3" :: Maybe a
```

  - so, then we should adjust the above to be concrete annotation:

```
read "Just 3" :: Maybe Int
```

42

— Ordering —

43

Deriving Instances of `Ord`

The `Ord` type class applied to a type definition

- means our defined order for values is considered their ordering from smallest to largest

See how it is implemented with `Bool`:

```
data Bool = False | True deriving (Ord)

True `compare` False
True > False
True < False
```

- try swapping `False` and `True` in declaration for `Bool`

44

Comparing `Maybe` Values

Two values created with the same constructor are equal, unless there are fields that must also be compared.

- the fields must also have type an instance of the `Ord` type class
- e.g.: the `Nothing` value is smaller than any other `Maybe` value
  - any `Just` values will have their nested elements compared

Nested functions cannot be compared, so keep in mind this is only for elements that are also `Ord`.

```
Nothing < Just 100
Nothing > Just (-49999)
Just 3 `compare` Just 2
Just 100 > Just 50
```

We cannot do `Just (*2) < Just (*3)` because the nested elements are functions.

45

— Enums, etc. —

46

Deriving `Enum` Instances

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

Some reminders:

- `Enum` places values in a sequential order, with each value having a predecesor and a successor
- `Bounded` expects a type to have a lowest value and a largest value

With the above, try out a few simple statements:

```
Wednesday
show Wednesday
read "Wednesday" :: Day
```


47

Instance Behaviour

By instance of the `Eq` and `Ord` type classes:

```
Saturday == Sunday
Saturday == Saturday
Saturday > Friday
Monday `compare` Wednesday
```

Also by `Bounded` instance:

```
minBound :: Day
maxBound :: Day
```

More possible as part of `Enum`:

```
pred Tuesday
succ Thursday
[Monday .. Friday]
[minBound .. maxBound] :: [Day]
```

48

— Type Synonyms —

49

Type Synonyms with `type`

We can define another name for a type like so:

```
type String = [Char]
```

This allows us to use either type name interchangeably:

- note that there is no use of the `` `data` `` keyword

The following are equivalent:

```
toUpperString :: [Char] -> [Char]
toUpperString :: String -> String
```

50

Using Type Synonyms

We first declared a `phoneBook` variable with type `[(String,String)]`.

Use type synonym to make things a bit more readable:

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]
```

See how much easier to read functions:

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnum pbook = (name, pnum) `elem` pbook
```


51

Use of `type`

Otherwise, without synonyms, the function would be described as:

```
inPhoneBook :: String -> String -> [(String,String)] -> Bool
```

Try to be judicious with synonyms; do not overuse them.

We can also use synonyms to create type constructors:

```
type AssocList k v = [(k, v)]
```

The signature of a function to get the value corresponding to a key in an association list might look like the following:

```
(Eq k) => k -> AssocList k v -> Maybe v
```

52

Reducing Type Constructors

Remember, a type constructor takes type parameters and returns a concrete type, for example `AssocList Int String`.

Perhaps a partially applied type constructor, for example:

```
type IntMap v = Map Int v
```

…can be expressed more simply:

```
type IntMap = Map Int
```

To implement the above, you will likely need to do a qualified import and precede the `Map` with module name:

```
type IntMap = Map.Map Int
```


53

Synonyms with Type Annotations

We cannot use synonyms as a constructor themselves:

```
AssocList [(1,2),(4,5),(7,9)]
```

Instead, they are used to specify with type annotations:

```
[(1,2),(4,5),(7,9)] :: AssocList Int Int
```

Note that type synonyms, and types in general:

- can only be used in Haskell where the syntax allows for it (as we have done in our definitions, descriptions, and declarations)

54

— Two Kinds of Values —

55

Either Type

A design for a kind of type we use later has the following description:

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

No need to define the above; it is already in `Prelude`.

See how values of the `Either` type are described:

```
Right 20
Left "woot"
:t Right 'a'
:t Left True
```

56

More Reasoning for Errors

The `Either` type has similar result as for `Nothing` as `Maybe a` where one of the parameters is polymorphic.

- `Maybe` type helped deal with computations that could have an error
  - the error is for exactly one reason
  - e.g.: `find` did not get a match for it to return

With an `Either` type description, we have flexibility to pass forward more reasoning for an error.

57

LockerMap Type

As an example, we will design a map that:

- manages locker numbers,
- tracks lockers' states so we know which ones are currently used,
- and store a secret combination string for each locker

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)
type Code = String
type LockerMap = Map.Map Int (LockerState, Code)
```

- the `LockerState` keeps track of whether the locker is taken or free
- the `Code` and `LockerMap` are only synonyms

58

Setup for Locker Lookup

Next, we will write a function that searches for the code in a locker map.

The return value of type `Either` helps deal with two ways the function could fail:

- the locker could be taken already, so no code should be given back
- the locker number might not exist

In both cases we use a different string to describe the error.

59

Lookup Function

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing ->
                Left $ "Locker " ++ show lockerNumber ++ " does not exist!"
        Just (state, code) ->
                if state /= Taken
                        then Right code
                        else Left $ "Locker " ++ show lockerNumber ++ " is already taken!"
```

60

Test Lookups

Add the lookup function to script file, then try some lookups:

```
lockerLookup 101 lockers
lockerLookup 100 lockers
lockerLookup 102 lockers
lockerLookup 110 lockers
lockerLookup 105 lockers
```


61

— Recursive Data Structures —

62

Creating with Recursion

The data definition that Haskell provides allows us to make a reference to itself.

- there is also another name for the `:` operator called "Cons" (short for constructor)
- recall we have used `:` with lists in a way that is recursive-like `3:4:5:6:[]` equal to `[3,4,5,6]`

We will design our own `List` type:

```
data List a = Empty | Cons a (List a)
        deriving (Show, Read, Eq, Ord)
```

The purpose is to see how to extend to other data types.

63

List Constructor with Record

- the value `a` directly after `Cons` would be the type of element to be attached to the head of the list
- the `a` after `List` is for the type of elements in the `List` itself

Alternatively, the same can be described in record syntax:

```
data List a = Empty | Cons { listHead :: a, listTail :: List a}
    deriving (Show, Read, Eq, Ord)
```

- so the `Cons` constructor takes two parameters with types: `listHead` and `listTail`
- the return type is a `List` (leftmost part of the type definition, before `=`)

64

Using List Constructor

Observe creating List values, and its use of recursion:

```
5 `Cons` Empty
4 `Cons` (5 `Cons` Empty)
3 `Cons` (4 `Cons` (5 `Cons` Empty))
```

The above would be equivalent to:

```
5:[]
4:5:[]
3:4:5:[]
```


65

Infix Declarations

We can define functions to be infix by default by naming them with only special characters.

- same for constructors, but keep in mind that Haskell constructors return data types

Then a redesign for infix by default of our `List` type:

```
:{
infixr 5 :-:
data List a = Empty | a :-: (List a)
        deriving (Show, Read, Eq, Ord)
:}
```

66

Using Infix

The `infixr` declares the precedence of the operator compared to other infix operators.

- higher value of `infixr` has higher priority, and so precedence before operators of lower value
- the `r` at the end of `infixr` means the operation is right associative
  - two operations of `:-:` would be evaluated right-to-left order

```
Empty
3 :-: 4 :-: 5 :-: Empty
let a = 3 :-: 4 :-: 5 :-: Empty
100 :-: a
```


67

Concatenation

We will implement concatenation for our `List` types.

It helps to see the implementation of `++` for `[]`:

```
{- ### default implementation ### -}
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

So, we will want:

```
infixr 5 ^++
(^++) :: List a -> List a -> List a
Empty ^++ ys = ys
(x :-: xs) ^++ ys = x :-: (xs ^++ ys)
```

68

Constructors and Pattern Matching

Then give it a try:

```
let a = 3 :-: 4 :-: 5 :-: Empty
let b = 6 :-: 7 :-: Empty
a ^++ b
```

Notice how we pattern matched in the definition of `^++` with `x :-: xs` which is a constructor.

- pattern matching (only) works on constructors
- this follows our previous use of pattern matching `:` and `[]` and constant values like 8 and 'a'
  - which are actually constructors for the numeric and character types, respectively

69

— Tree Type —

70

Defining Trees

We now have enough to start implementing binary search trees. Recall…

- a tree node stores a value, and references to
  - a left subtree
  - a right subtree
- all values in left subtree are smaller than the current node
- all values in the right subtree are larger than the current node

We will not worry about keeping our trees balanced in this implementation.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
    deriving (Show)
```


71

Node Values

- `Node` is a recursive constructor, and is not explicitly describing some kind of `Node` type on its own
  - the closest thing to an individual node with stored value, say 3:
    - `Node 3 EmptyTree EmptyTree`
- in a language such as C, we could modify the pointers and the tree directly for an insertion
- in Haskell, we need to replace the entire tree when we do an insert
  - the new value should insert at some leaf
  - Haskell cannot change a value, but can only replace it with a new value
  - Haskell will need to replace for each subtree to complete the insertion
  - at least Haskell can share most of the subtrees from the old tree to keep things somewhat efficient

72

Insert with Trees (1)

We will start by making a function to create a one-node root-level tree:

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree
```

Then we can use it to help us write an insertion function:

```
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node y left right)
 | x == y  = Node x left right
 | x < y   = Node y (treeInsert x left) right
 | x > y   = Node y left (treeInsert x right)
```


73

Insertion with Trees (2)

Notice that the guards list three cases for dealing with insertion:

- if the current node already contains the value we want to insert
- if the left subtree should take care of the insertion
- if the right subtree should take care of the insertion

The first pattern takes care of assigning a singleton in the location where we find an empty subtree:

- this would be the same as placing a new leaf to the tree

74

Creating Trees Concisely

Now we can write code to create a tree very quickly:

```
let nums = [8,6,4,1,7,3,5]
let numsTree = foldr treeInsert EmptyTree nums
```

We insert numbers from a list into our tree

- one element at a time (from the right of the list)

The `numsTree` value is awkward to read all on one line.

75

Check for Elements in a Tree

We can also test for whether elements are inside the tree:

```
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node y left right)
 | x == y  = Node x left right
 | x < y   = Node y (treeInsert x left) right
 | x > y   = Node y left (treeInsert x right)
```

```
8 `treeElem` numsTree
100 `treeElem` numsTree
1 `treeElem` numsTree
10 `treeElem` numsTree
```

76

— Inside the `Eq` Type Class —

77

Declaring Type Classes

There is a bit of confusion to clear up when we define our own type classes.

- a type class is NOT similar to a class in other programming languages
- remember, it is more like an interface (declaring what ***must*** be implemented later)

Observe the standard library `Eq` definition:

```
class Eq a where
 (==) :: a -> a -> Bool
 (/=) :: a -> a -> Bool
 x == y = not (x /= y)
 x /= y = not (x == y)
```

78

`Eq` Type Class Behaviour

- keyword `` `class` `` (not `` `type class` ``!)
- the `` `a` `` only need be lowercase, and represents whatever the type is that will become part of the `` `Eq` `` type class
- then the type descriptions for the functions (==) and (/=)
- these function type descriptions would elsewhere be observed to have restrictions of `` `(Eq a)` ``
- lastly, the two definitions of the functions (==) and (/=) are only implemented here involving the types
  - they are called mutually recursive (they depend on each other)

79

— Traffic-Light Data Type —

80

Creating Instances of Type Classes

We can also create our own instances of type classes.

But first, we will create a new type:

```
data TrafficLight = Red | Yellow | Green
```

Above lists the possible states of a `TrafficLight`.

Now let us make it an instance of `Eq`:

```
instance Eq TrafficLight where
    Red == Red       = True
    Green == Green   = True
    Yellow == Yellow = True
    _ == _  = False
```


81

Implementing Mutually Recursive Functions

- the use of `instance` keyword will mean we specify a *concrete* type `TrafficLight` instead of a *generic* `a` type parameter
- we only needed to give an implementation for **one** of the two mutually recursive functions from the `class` description of `Eq`
  - this is called minimal complete definition

If we instead described the two functions of `Eq` as:

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

…then Haskell would expect us to impelement *both* functions.

82

Further Instancing with Show

Now we also make `TrafficLight` an instance of the `Show` type class:

```
instance Show TrafficLight where
    show Red = "Red Light"
    show Green = "Green Light"
    show Yellow = "Yellow Light"
```

Then make sure to not gloss over the following interactive testing:

```
Red == Red
Red == Yellow
Red `elem` [Red, Yellow, Green]
[Red, Yellow, Green]
```

- `Eq` could have just been derived, but not `Show` as you observe the last expression printed

83

Date Instance of Show (David Semke)

```
type Year = Int
type Month = Int
type Day = Int

data Date = Date {
    year :: Year,
    month :: Month,
    day :: Day
} deriving (Eq)
```

84

Date instance of Show

```
instance Show Date where
    show (Date y m d) =
        if (y > 1999 && y < 2023
            && m > 0 && m < 13
            && d > 0 && d < 32)
            then (
                let year = show y
                    day = show d
                    month
                        | m == 1 = "Jan"
                        | m == 2 = "Feb"
                        | m == 3 = "Mar"
                        | m == 4 = "Apr"
                        | m == 5 = "May"
                        | m == 6 = "June"
                        | m == 7 = "Jul"
                        | m == 8 = "Aug"
                        | m == 9 = "Sep"
                        | m == 10 = "Oct"
                        | m == 11 = "Nov"
                        | m == 12 = "Dec"
                in month ++ " " ++ day ++ ", " ++ year
            )
            else "Invalid Date!"
```


85

Subclasses

Next, we can make type classes subclasses of other type classes:

```
class (Eq a) => Num a where
    ...
```

(the rest of the code … is omitted)
The constraint forces:

- any type to be made an instance of `Num`
- must first be made an instance of `Eq`

- so, here `Num` is a subclass of the `Eq` type class
- it makes sense for a value to be considered a number that it first be possible to check if one value could be equal to another

86

— Parameterized Types as Instances of Type Classes —

87

Toward Parameterization

The goal here is to make a parameterized type.

- e.g.: `Maybe` is not a concrete type
- its constructor will require a type parameter, e.g.: `Char`
- providing the type parameters will create a concrete type, e.g.: `Maybe Char`

Now we want to make such a parameterized type an instance of a type class. Consider again the `Eq` type class:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

88

Parameterization

Remember, you must have a concrete type for all parameters listed for describing a function:

- e.g.: you cannot have a function of type `a -> Maybe`
- e.g.: you **can** have a function of type `a -> Maybe a`

This is also why we ***cannot*** have the following:

```
instance Eq Maybe where
```

- similar to why `Maybe` is not a concrete type
- we want to avoid repeating implementation for all the different concrete types
  - **not**: `instance Eq (Maybe Int)`, `instance Eq (Maybe Char)`, etc
  - so, use a type variable, i.e.:

```
instance Eq (Maybe m) where
    Just x == Just y    = x == y
    Nothing == Nothing  = True
    _ == _  = False
```


89

Constraints on Parameters

The previous demonstration is **PSEUDOCODE.**

We would like to make all types of `Maybe something` an instance of `Eq`:

- we still need more syntax for one more restriction in proper Haskell
- the role of `Maybe a` is the same as `a` in `class Eq a where`

The previous pseudocode has the issue of applying `==` operations on values of type `m`.

So, we need a class restriction to complete the code:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y    = x == y
    Nothing == Nothing  = True
    _ == _  = False
```

90

Class Constraints in Instance Declarations

So the class constraint makes sure that any `m` we pass in is of type `Eq`.

Note the two uses of class constraints:

- in class declarations, to make one type class a subclass of another
- in instance declarations,
  to require some possibly nested contents to be of some type
  - e.g.: we required contents of `Maybe` to be instance of type class `Eq`

We use similar syntax for describing functions,
e.g.: `(==) :: (Eq m) => Maybe m -> Maybe m -> Bool`:

- mentally replacing `a` type variable with your concrete types is what you need to do in your own implementations, since `==` has type `(==) :: (Eq a) => a -> a -> Bool`

91

:info

The `:info` command in ghci is helpful for learning about types, type classes, and type constructors.

Try the command `:info Maybe` to see the type classes for which `Maybe` is an instance.

92

— `YesNo` Type Class —

93

Boolean-like Type Class

We can practice some implementation to match behaviour in **JavaScript** for the following `if`-statements:

```
if (0)  alert("YEAH!") else alert("NO!")
if ("") alert("YEAH!") else alert("NO!")
if (false) alert("YEAH!") else alert("NO!")
```

All of the above would throw an alert of "NO!", however, the next statement would throw an alert of "YEAH!":

```
if ("WHAT") alert("YEAH!") else alert("NO!")
```

94

YesNo Type Class (1)

We implement this for practice, and we typically would be better to rely on the default `Bool` type for test conditions.

```
class YesNo a where
    yesno :: a -> Bool
```

The above class type will mean any instance types will need to implement the `yesno` function.

- the intention of the `yesno` function:
  - should check value of type `a`
  - return some Boolean-like value of `True` or `False` of our custom design

95

YesNo Type Class (2)

Let us implement the concept of numbers being

- true (yes-ish value)
- or false (no-ish value)

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

96

YesNo Instance for Lists

Similarly, we can instance YesNo for lists:

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

- we put a type variable `a` inside the list square brackets to
  - make the type concrete
  - without making any assumptions about the concrete type passed in

An interesting concise way to implement the `Bool` type:

```
instance YesNo Bool where
    yesno = id
```

The `id` (short for "identity") is a function from the standard library that just returns the parameter passed in.

97

Many Instances of YesNo

Instance for Maybe:

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```

We can implement for the type we created, `Tree`:

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

And for `TrafficLight`:

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _ = True
```

98

YesNo Testing

Test out the behaviour of `YesNo` instances:

```
yesno $ length []
yesno "haha"
yesno $ Just 0
yesno True
yesno EmptyTree
yesno []
yesno [0,0,0]
:t yesno
```


99

JavaScript-like Behaviour

We can implement a function similar to an `if`-statement, but meant to test values of `YesNo` instance type:

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal
    then yesResult
    else noResult
```

Then test out the behaviour similar to JavaScript:

```
yesnoIf [] "YEAH!" "NO!"
yesnoIf [2,3,4] "YEAH!" "NO!"
yesnoIf True "YEAH!" "NO!"
yesnoIf (Just 500) "YEAH!" "NO!"
yesnoIf Nothing "YEAH!" "NO!"
```

100

— The Functor Type Class —

101

Functors

The type classes we have learned help deal with:

- converting from Strings and back,
- equating,
- and ordering

After all that practice, we can likely handle dealing with a type class that describes mapping things.

To this end, the `Functor` type class will help manage lists:

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

102

Functor Behaviour

This may look strange at first, but it will remind you of how we worked with the `map` function before on lists.

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

The `Functor` type class is quite a bit different from the previous ones we have seen so far.

- note that `f` is the major difference, being a parameterized type
  - so `f` is not a concrete type
  - `f` can be thought of as a context we want containing nested elements
- this allows us to program code to avoid having many nested calls
  - e.g.: `map` applies some function to elements of a list

103

Implementation of `fmap`

Now see how the list type is an instance of the `Functor` type class:

```
instance Functor [] where
    fmap = map
```

- in the above, `[]` is not a concrete type, but will require a type parameter later
  - i.e.: `[a]` in describing functions, or more specifically `[Int]`, `[Char]`, `[String]`, etc.

This implementation of lists and `fmap` works exactly the same as `map` (since `fmap = map`):

```
fmap (*2) [1..3]
map (*2) [1..3]
```

104

Empty Lists

Note that `fmap`:

- of an empty list of concrete type `[a]`
- just results in an empty list of concrete type `[b]`

(whatever `a` and `b` happen to be implemented as)

105

— Maybe as a Functor —

106

`Maybe` as a Functor

Then any parameterized type is ripe for implementation with `Functor`, such as `Maybe`:

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

Again, notice we are filling in a type constructor `Maybe` and not a concrete type `(Maybe a)`.

- for `fmap` we have a description `(a -> b) -> Maybe a -> Maybe b`
- nested value of `Just` has the `a -> b` function applied to it
  - then we do not have to explicitly write that nesting ourselves later

107

Check with Signatures

In code later, it is possible to become confused because the syntax we are using depends heavily on the context.

To help yourself remember, mentally fill in what you try to do with either the type constructor or the concrete type.

See if the resulting syntax makes sense…

```
(a -> b) -> Maybe m a -> Maybe m b
```

…which the above ***does not make sense.***
(since `Maybe` only takes one type parameter)

108

`fmap` Practice

Some expressions to try:

```
fmap (++ " BECOMES PART OF THE NESTED MAYBE VALUE!")
        (Just "Something serious.")
fmap (++ " BECOMES PART OF THE NESTED MAYBE VALUE!") (Nothing)
fmap (*2) (Just 200)
fmap (*2) Nothing
```


109

— `Trees` as Functors —

110

Tree Instance of Functor

Anything we make an instance of `Functor` type class is some kind of container, such as `Tree` we implemented:

- the `Tree` type constructor takes only one parameter
- to implement `fmap` it looks like `(a -> b) -> Tree a -> Tree b`

This time, we will have to implement things recursively:

- the base case of an empty tree is another empty tree
- anything else has the function applied to the root node, and `fmap` applied to left and right subtrees separately

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x left right) =
        (Node f x) (fmap f left) (fmap f right)
```


111

Testing Tree Functor Behaviour

Try it out:

```
fmap (*4) (foldr treeInsert EmptyTree [5,7,3])
```

…but remember, that values such as

`(Node 20 EmptyTree EmptyTree)`

are just a single node with the value `20` stored.

- the demonstration here exposes the possibly that a property of binary search trees is broken
  - values mapped on the left subtree may no longer be smaller than their root
  - same for values mapped on the right being larger than the root
  - any monotonic function would be fine, otherwise, the result is likely a binary search tree no longer

112

— `Either` as a Functor —

113

Two Type Parameters

We may have a type constructor with two parameters.

We only want to apply a function to one of its possible values, and not the other.

Implementing `Either` to be an instance of `Functor`:

```
instance Functor (Either a) where
    fmap f (Right x) = Right x
    fmap f (Left x) = Left f x
```

114

Map Instance of Functor

Similar types that have multiple type parameters can be made instance of Functor, say `Data.Map` with its type description `Map k v`.

- then `fmap` would take
  - first parameter some function `v -> w`
  - second parameter a map of type `Map k v`
- `fmap` should then return a map of type `Map k w`
- see if you can implement how to make `Map k` an instance of `Functor`

Important: a `Functor` instance is the "context", e.g.: `Map`, and not the same thing as the function passed in to `fmap`.

115

— Kinds —

116

The `kind` Command

Every value in Haskell has a concrete type, but even each type has a type—known as a kind.

To check the kind of a type, use `:k` command:

```
:k Int
```

The result gives `Int :: *` where the `*` just means `Int` is a concrete type.

- read out loud as "star" or "type"

117

Now for a type that is not concrete:

```
:k Maybe
```

- the result is `Maybe :: * -> *`
- so, `Maybe` is a function that takes a concrete type and returns a concrete type

118

One Type Parameter for Functors

Then let us take a look at the kind of `Either`:

```
:k Either
:k Either String
```

- the result for `:k Either` is `* -> * -> *`
- the next result is `* -> *`
- the `Functor` type class expects a type constructor of kind `* -> *`
- a `Functor` instance must be a type constructor function that takes one type parameter and returns a concrete type

119

— Chapter 8: Input and Output —

120

— Separating Pure from Impure —

121

I/O has Side Effects

Remember that functions in Haskell are expected to always return the ***same result*** when given the ***same input values***.

- inconsistent functions would have side effects
- functions in Haskell are not allowed to have side effects

How does the result of a function even get displayed?
It must *change* something to do so.

- for a result to get printed to the screen, some ***state*** of the system ***has to change***
- the part that changes has side effects
- Haskell separates the parts of the program that
  - do not have side effects (pure)
  - from the parts that do (impure)

122

Scripts

We will move on to writing scripts that perform more than simply defining functions (and then testing them in ghci).

Let us start with the familiar "`Hello, World!`" program:

```
main = putStrLn "Hello, World!"
```

Save the above in a file called `` `hello.hs` ``.

- there could be different commands for compiling depending on your development environment

For Windows and the Haskell stack command line:

```
stack ghc hello
```


123

Compile and Run

you may have installed the Glasgow Haskell Compiler (GHC):

```
ghc --make hello
```

Run the generated program with `` `hello.exe` `` in Windows or `` `./hello` `` in most other platforms.

It is possible to run a script within ghci:

- add a blank line at the end of your script, and then `` `main` `` as the final statement
  - this will execute `` `main` `` in ghci
- then in ghci type `` `:script hello.hs` `` to run the script
- you can load modules, etc., separately, as opposed to defining everything inside the script

124

I/O Actions

Let us take a look at the type of the function `putStrLn`:

```
:t putStrLn
:t putStrLn "Hello, World!"
```

- First result: `putStrLn :: String -> IO ()`
- Second result: `putStrLn "Hello, World!" :: IO ()`

The first result has `putStrLn` function that takes a string and returns an I/O action that yields an empty tuple.

- printing to the terminal does not have any meaningful side effect, so the empty tuple represents a dummy value (`()` is also the description of its type)

An I/O action will be executed when we execute `main`.

125

— Gluing I/O Actions Together —

126

Context of `do` Block

A program involves gluing together multiple I/O actions:

```
main = do
    putStrLn "Hello, what is your name?"
    name <- getLine
    putStrLn ("Hey, " ++ name ++ ", you rock!")
```

Save the program as `ask.hs`, compile, and run.

- the above I/O actions were glued together into one I/O action with the use of `do` keyword
- main always has a type of `IO` *something*
- the program `main` above has type `IO ()`
- it is not typical to give type declaration for `main`

127

`getLine` Function

Let us take a closer look at the newest statement:

```
:t getLine
```

The result is `getLine :: IO String`

- `getLine` is a function that returns an `IO` which yields a `String`
- the user will type in the terminal
- `name <- getLine` means `name` assigned a yield `String` value
- an I/O action is separate from the rest of our program
  - we can ask it to give us some data from its travels
  - get the data back with `<-` construct
  - `getLine` is impure, because it does not guarantee the same value when executed twice
  - we can only ask for data when inside another I/O action

128

Impure Data and Environments

note: `tellfortune` function is not implemented in textbook

Consider the following program:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "This is your future: " ++ tellFortune name
```

the `tellFortune` function does not need to know anything about `IO String` because `name` is just type `String`

- to emphasize this, we cannot do the following `nameTag = "Hello, my name is " ++ getLine`
  - we cannot concatenate a `String` and an I/O action
  - we first need the string yielded from the I/O action
- we can only get yielded data, or impure data from within an impure environment

129

Variable Binding

All I/O actions in a `do` block can have yield stored in a variable, except for the last action:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey, " ++ name ++ ", you rock!")
```

- parentheses after `putStrLn` for one string parameter
- `foo` would just have type `()`
- again, we **cannot** assign the last I/O action to anything
  - its yield becomes the return value of the do block
- any I/O action result could be ignored, `_ <- putStrLn "something"`, but we can leave off the assignment without worry of any error

130

When I/O Actions Happen

The following is possible:

```
myLine = putStrLn
```

but it just gives another name to the `putStrLn` I/O action, so there is not much need.

I/O actions will be performed:

- when `main` is executed
- a `do` block is executed in main with an I/O action nested inside
  - `do` blocks glue together I/O actions
  - these blocks can be nested inside another `do` block
  - all will be performed if within any level nested inside `main`
- when we type an I/O action statement in ghci and press `ENTER`

131

Example for Combining I/O Actions
(Hunter Klassen)

```
multiply :: Int -> Int -> Int
multiply x y = x*y

main :: IO()
main = do
    putStrLn "---Multiply---"
    putStrLn "Enter first integer:"
    x <- getLine
    putStrLn "Enter second integer:"
    y <- getLine
    let a = read x :: Int
    let b = read y :: Int
    print(multiply a b)
```

132

GHCI and I/O Actions

Ghci session performs I/O action when we simply type a value; `show` converts the value to a string and then uses `putStrLn`.

We can also use `let` expressions within a `do` block:

```
main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let
        bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $
        "hey "
        ++ bigFirstName
        ++ " "
        ++ bigLastName
        ++ ", how are you?"
```


133

Layout Syntax

Take a moment to observe the indentation in examples, called layout in Haskell:

- needs to follow proper block indentation alignment
- you are familiar with the errors
- good practice to work out your own style

Further comments:

- use `<-` for storing the result of an I/O action
- use `let` for assignment for the result of an expression that is pure (no IO action)
- `let firstName = getLine` just reassigns the name of `getLine` without executing it nor storing any yield

134

— Reverse Strings in I/O —

135

Recursive IO Actions

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main
    putStrLn $ "exiting main"

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

The above program will reverse each line we type into its user input until we type a blank line (nothing).

136

Working with I/O Actions

- `putStrLn` statement when `main` exits helps you see recursion
- the recursive call to `main` is itself an I/O action
  - a nested `do` block glues `putStrLn` and `main` calls into ***one*** I/O action expected by `else`
- the `unwords` function concatenates all the words together from its input list
- the `return ()` statement is special when used inside an I/O action because it ***wraps*** a pure value into the type `IO a`
  - similarly, `return "ha"` will wrap a yield into the type `IO String`
- condition `if null line` at some point expects an empty string returned into it yielded from `getLine`

137

`return` in Haskell

Unlike typical `return` statements in other programming languages is that `return` in Haskell does not end the execution of a block.

To see this, try:

```
main = do
    a <- return "oh, "
    b <- return "yeah!"
    return $ a ++ " " ++ b
```

You can just use `:set +m` for multiline in ghci:

- paste the above into the session for fast small programs
  - do not include the `main =` in the paste to just execute it after you press `ENTER` on a blank line

138

I/O Action Wrapping and Unwrapping

Realize that `return` is the inverse of `<-`, wrapping and unwrapping, respectively, I/O action yield values.

- keep in mind the examples are just demonstration—use `let` to simply assign variables
- `return` is used to wrap as the result given back at the end of a `do` block when an expression itself is not an I/O action

139

— Demonstrations of Some I/O Action Functions —

140

End of Line

The following demonstrates the difference between `putStrLn` and `putStr`:

```
main = do
    putStr "Hey, "
    putStr "I'm "
    putStrLn "Andy!"
```

- observe that `putStr` does not move output to the next line

141

One Output Character

The `putChar` function simply prints one character as an I/O action:

```
main = do
    putChar 't'
    putChar 'h'
    putChar 'e'
    putStrLn ""
```

142

Implementation of `putStr`

We can recursively implement our own version of `putStr` using `putChar`:

```
putStr' :: String -> IO ()
putStr' [] =
    return ()
putStr' (x:xs) = do
    putChar x
    putStr' xs
```


143

End-of-Line

Suppose we wanted an end-of-line:

```
putStrLn' :: String -> IO ()
putStrLn' [] = do
    putChar ('\n')
putStrLn' (x:xs) = do
    putChar x
    putStrLn' xs
```

- you get the idea for how to start using `do` blocks
- both functions print the first character, and then recurse for printing the rest of the string

144

print

The function `print` is basically `putStrLn . show`, as demonstrated:

```
main = do
    print True
    print 2
    print "haha"
    print 3.2
    print [3,2,1]
```

- ghci actually just uses `print` to automatically display anything we evaluate that is a type instance of `Show`
- notice that `print` places double-quotes around strings, whereas `putStr` and `putStrLn` do not

145

when Construct

The `when` function helps execute I/O actions, but only if the Boolean condition is satisfied. Note that:

- for `True` condition, the result of `do` block will have `return` applied
- for `False` condition, a `return ()` applies instead

```
import Control.Monad

main = do
    putStr "What fish is the sharpest? "
    input <- getLine
    when (input == "SWORDFISH") $ do
        putStrLn input
```

146

return ()

Without the `when` function, then we are forced to write the `else` statement corresponding to `if`:

```
main = do
    input <- getLine
    if (input == "SWORDFISH")
        then putStrLn input
        else return ()
```


147

sequence

The `sequence` function allows us:

- to work with an input list
- where the elements are I/O actions
- which will execute from first element to last element

When used with user input `getLine` the result will be stored in a corresponding list element when assigned.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

148

Lists of I/O Actions

`map print [1,2,3,4]` results in a list of I/O actions,

- but not itself an I/O action!
- therefore, it will not be executed when we press `ENTER` or run it in a program

This is the next use of `sequence`,

to execute such a list:

```
sequence $ map print [1,2,3,4]
```


149

Ignoring Output

```
sequence $ map print [1,2,3,4]
```

if executed in ghci, it will print out the expected values…

…but there will also be the result of the sequence itself with `[(),(),(),()]`.

- this is undesired; suppress it by prefixing the expression with `_ <-`
- turns out `_ <-` is actually useful for something!
- also, for expressions that are ***not*** I/O actions, we can do `_ =`, for example `_ = 3` to ignore printing any evaluation

150

mapM and mapM_

The uses of `sequence` together with `map` for I/O actions was so common that a combined function is provided now:

```
mapM print [1,2,3]
```

And if you do not want the evaluated list of empty actions, there is also the `mapM_` version:

```
mapM_ print [1,2,3]
```


151

forever

The `forever` function will just repeat an input I/O action over and over:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input:"
    x <- getLine
    putStrLn $ map toUpper x
```

- each line is echoed back with all capital letters

Do not press `CTRL+C` to stop ghci interaction.
https://www.haskellforall.com/2012/07/breaking-from-loop.html

152

forM

With the function `forM`, we finally get at code that looks something like other programming languages:

```
import Control.Monad

main = do
    colours <- forM [1,2,3,4] (\a -> do
        putStrLn $
            "Which colour do you associate with the number "
            ++ show a ++ "?"
        colour <- getLine
        return colour)
    putStrLn "The colours you associated with 1, 2, 3, and 4 are: "
    mapM_ putStrLn colours
```

- `forM` function evaluates to an I/O action

153

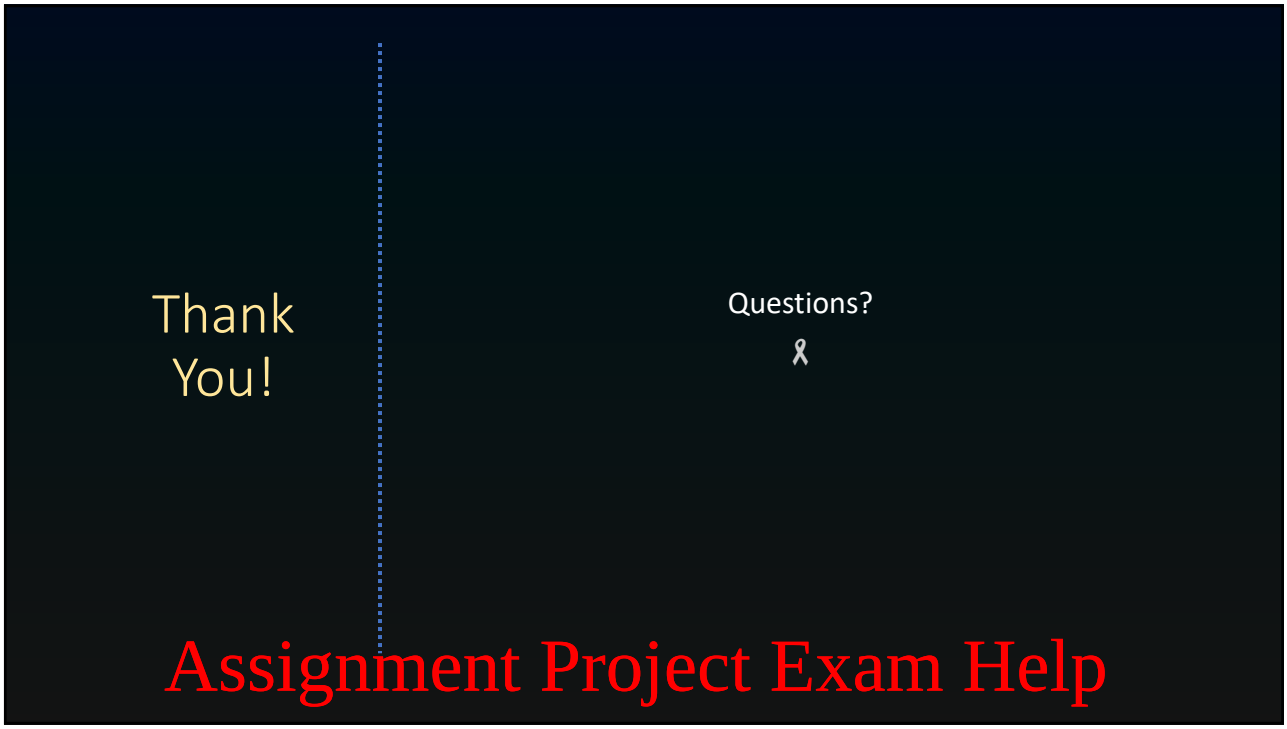
Simplifying do Blocks

The last program shows how `forM` is the same as `mapM`,

- but the input parameters are swapped so that the function can go last

- this does make the code more readable
- we could simplify a bit for the last two lines of the lambda function
  - remove `return colour` and replace `colour <- getLine` with `getLine`
  - these last two lines were unpacking and repacking the yield value

154



155