Week 5

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

University of the Fraser Valley

Dr. Russell Campbell

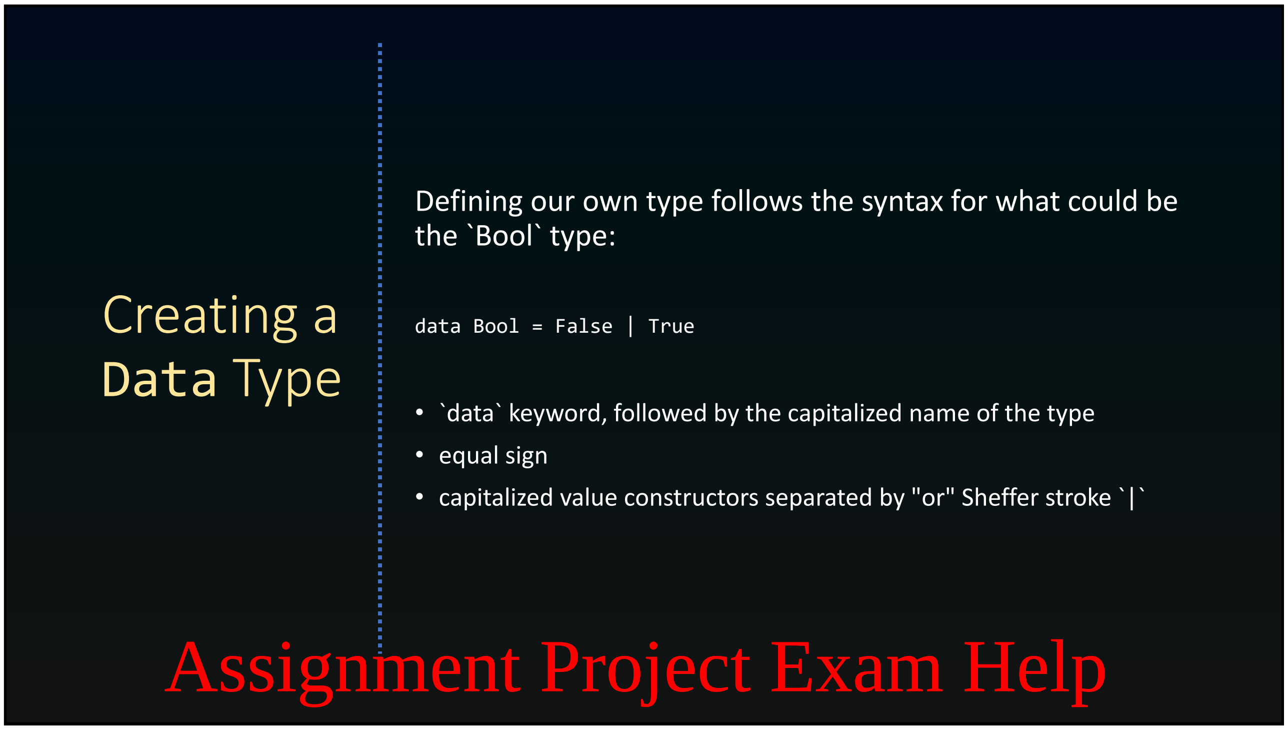
Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Overview

2



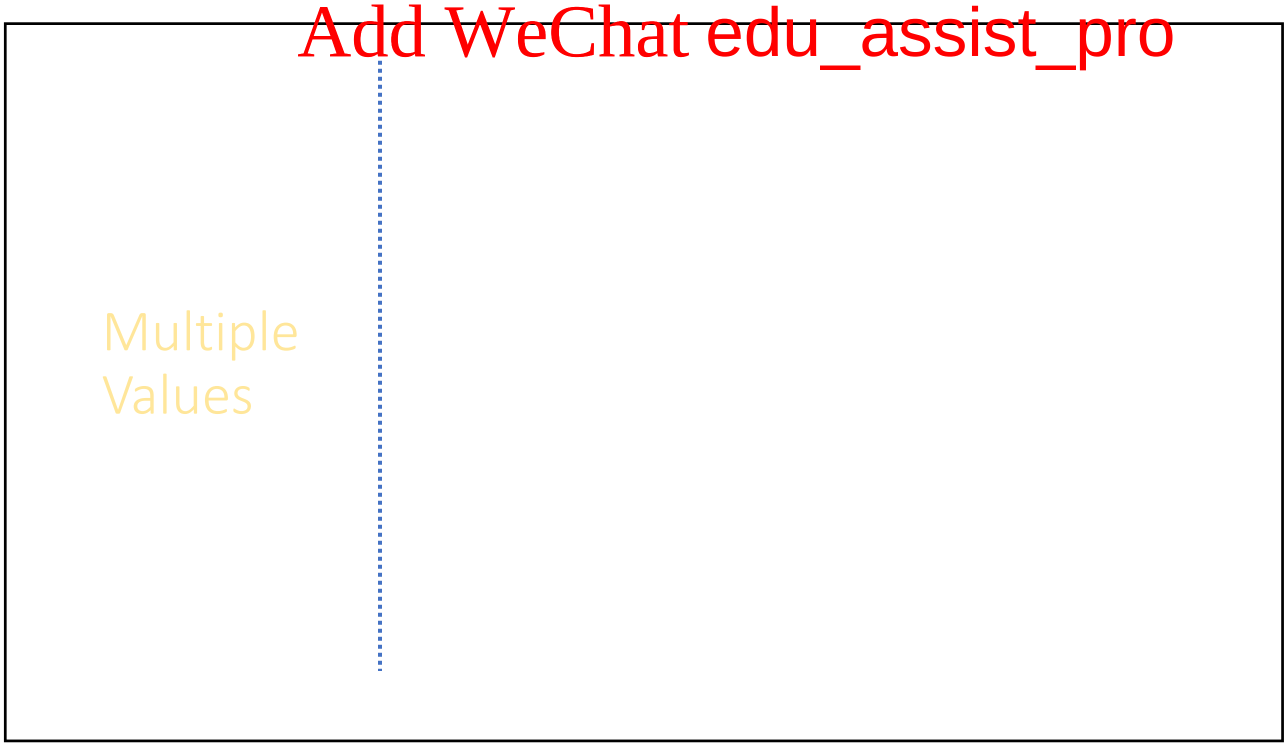
Creating a Data Type

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

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

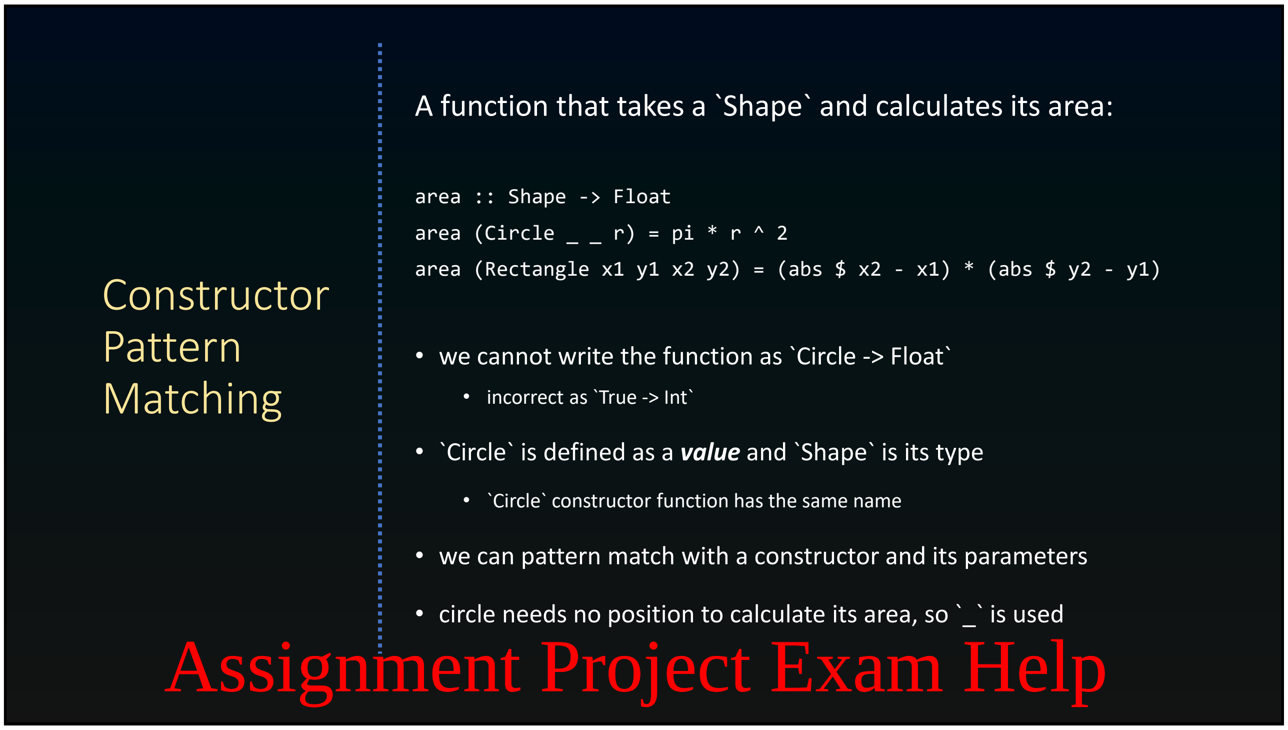
- `data` keyword, followed by the capitalized name of the type
- equal sign
- capitalized value constructors separated by "or" Sheffer stroke `|`

3



Multiple Values

4



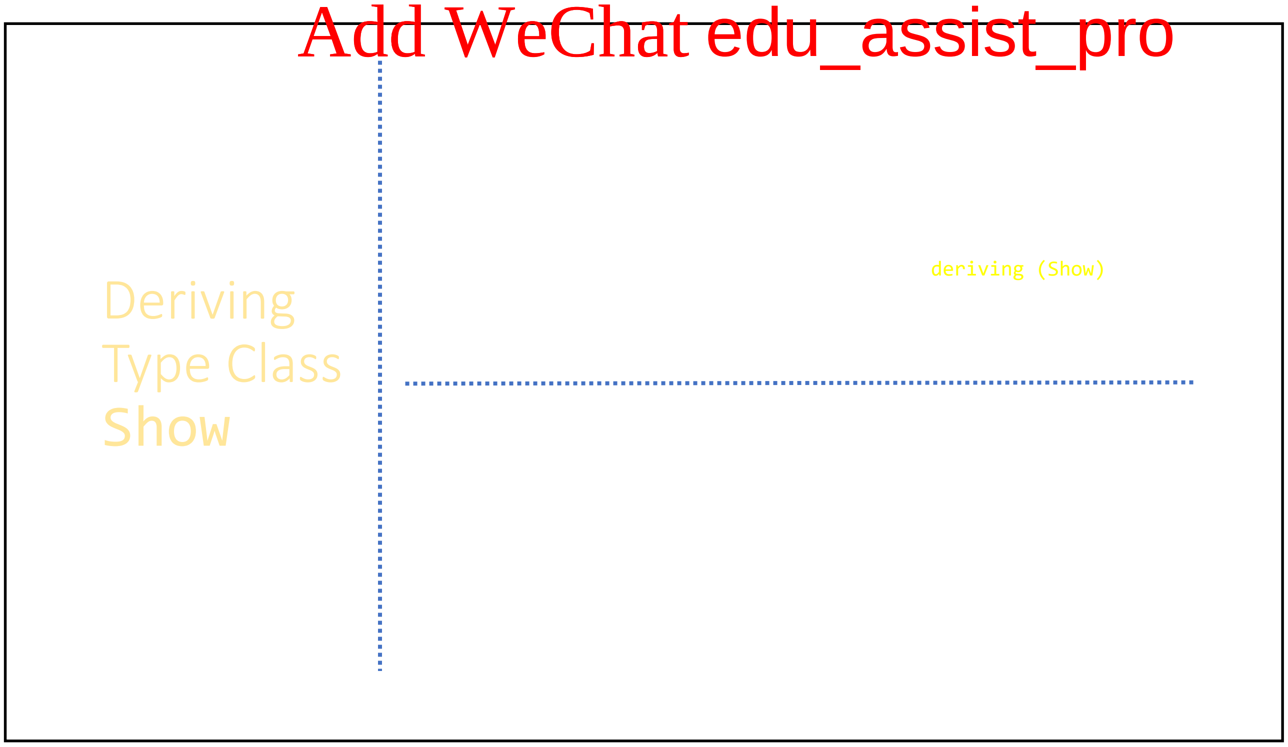
Constructor Pattern Matching

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

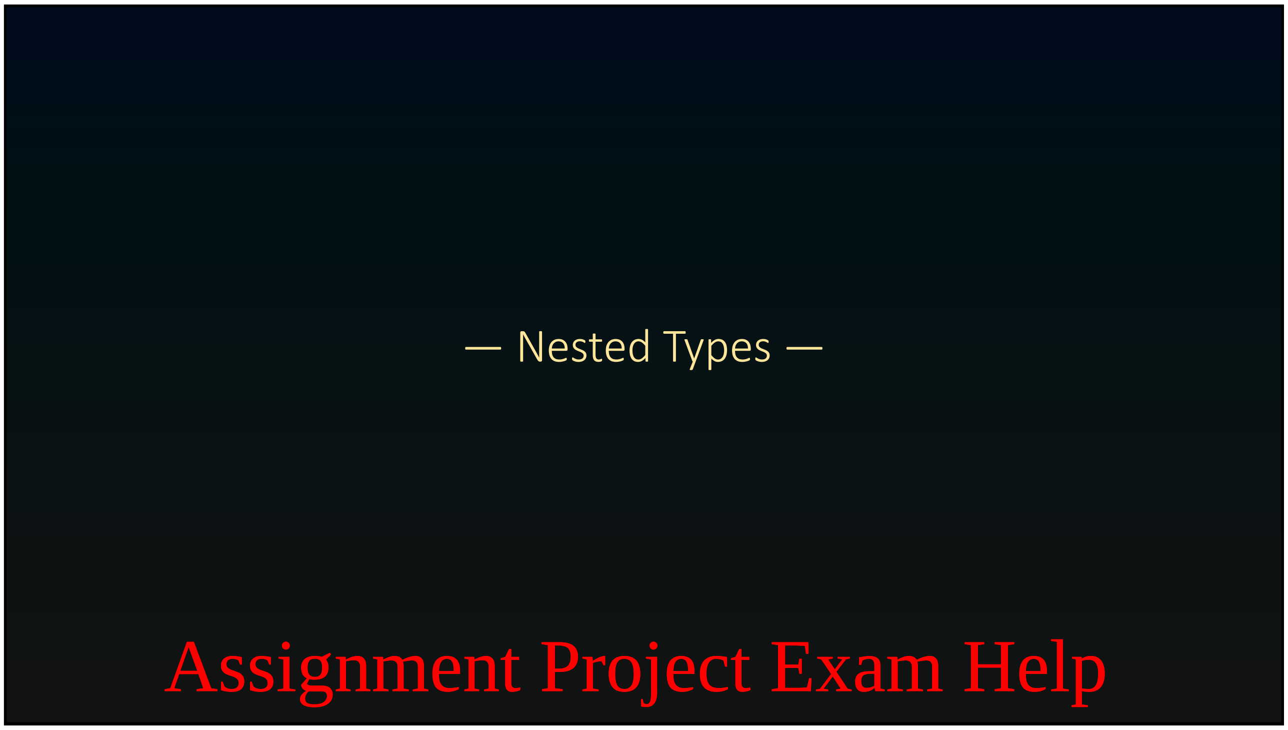
5



Deriving Type Class Show

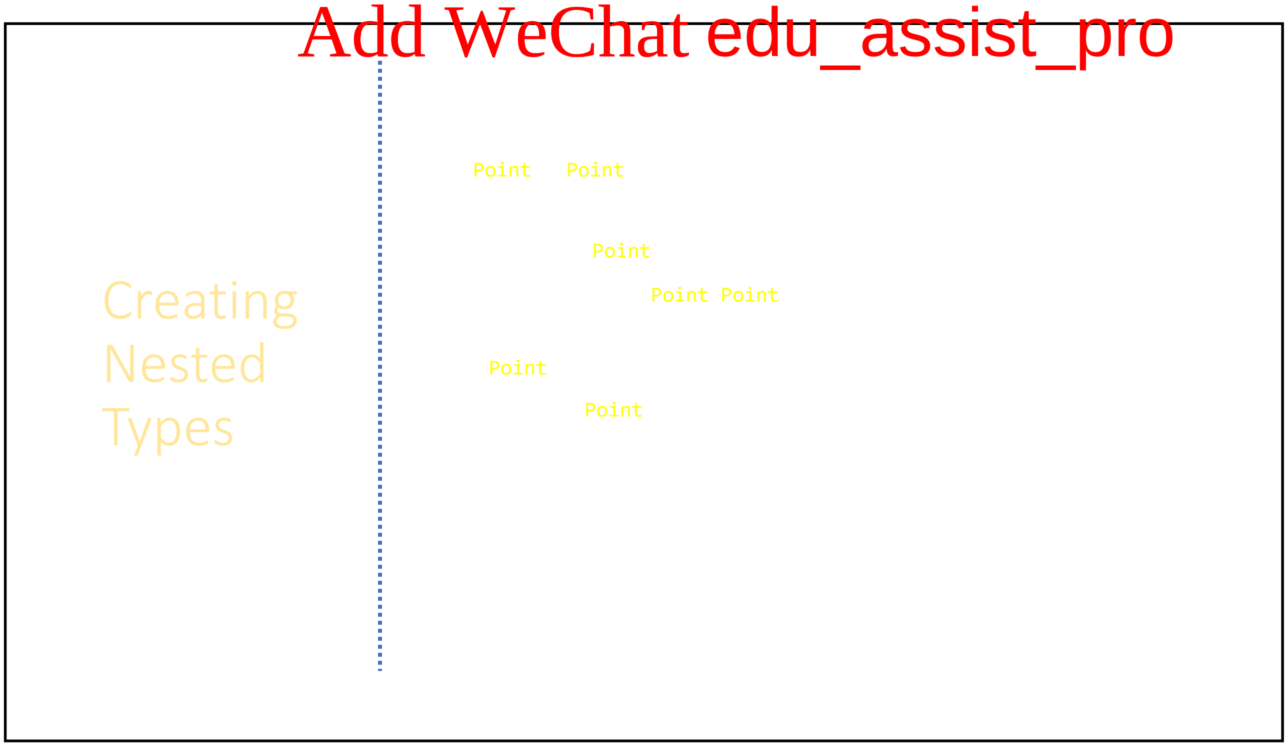
```
deriving (Show)
```

6



— Nested Types —

7



Creating Nested Types

Point
Point
Point
Point
Point
Point
Point

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

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

12

— Record Syntax —

13

Record Syntax

14

Parameter Order

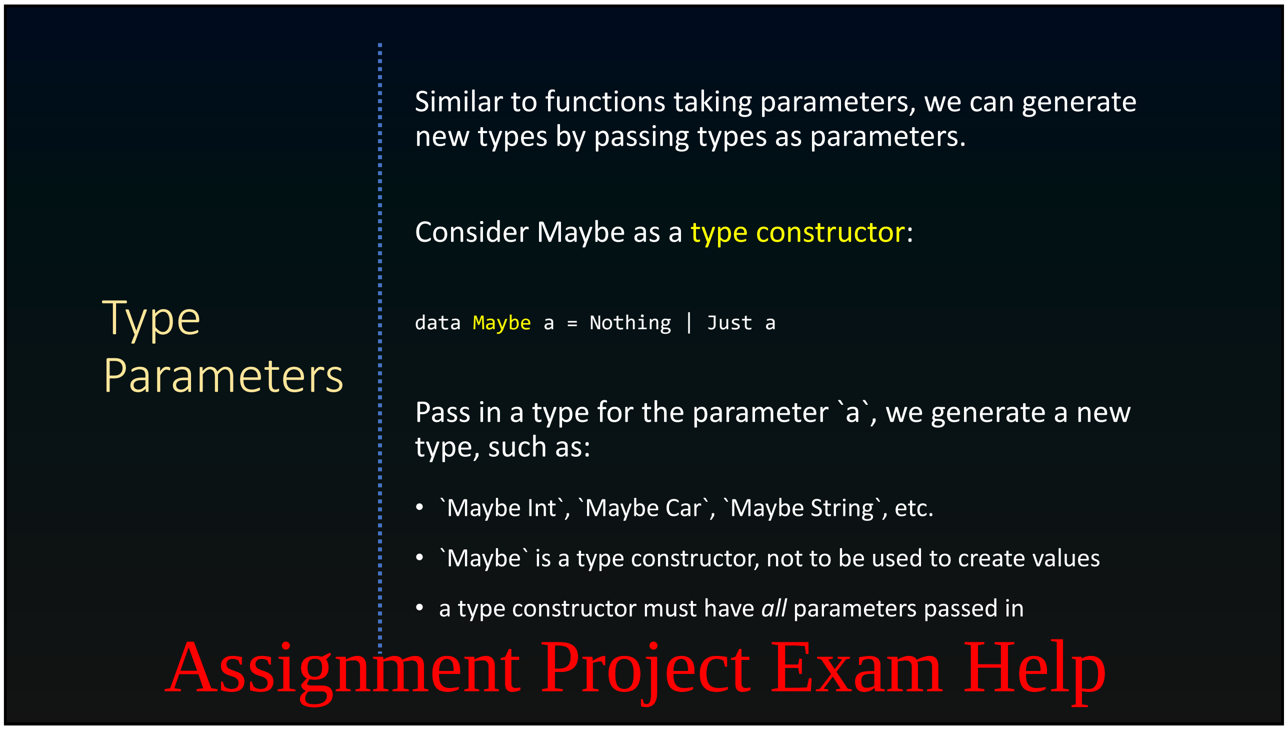
Choose to use record syntax when the order of the fields do not immediately make sense.

- 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.

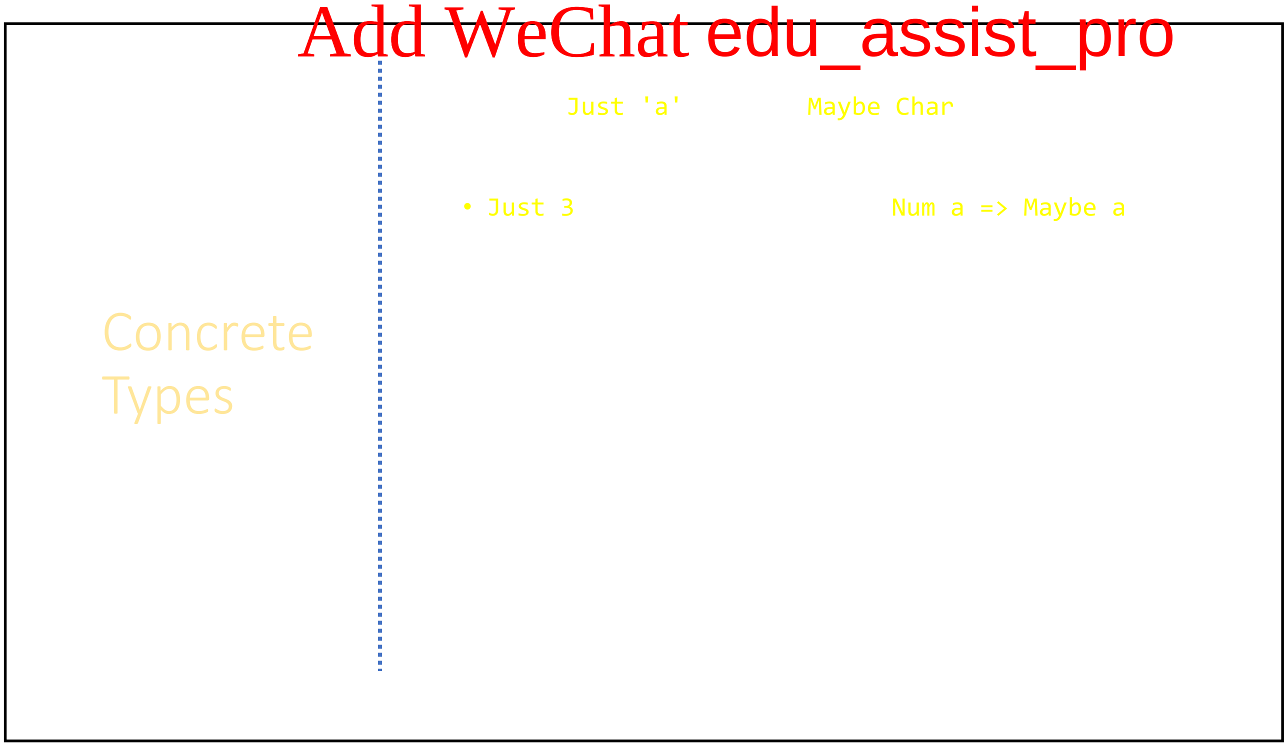
Consider Maybe as a type constructor:

```
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

Just 'a' Maybe Char

- Just 3 Num a => Maybe a

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

20

Practice Defining Concrete Types

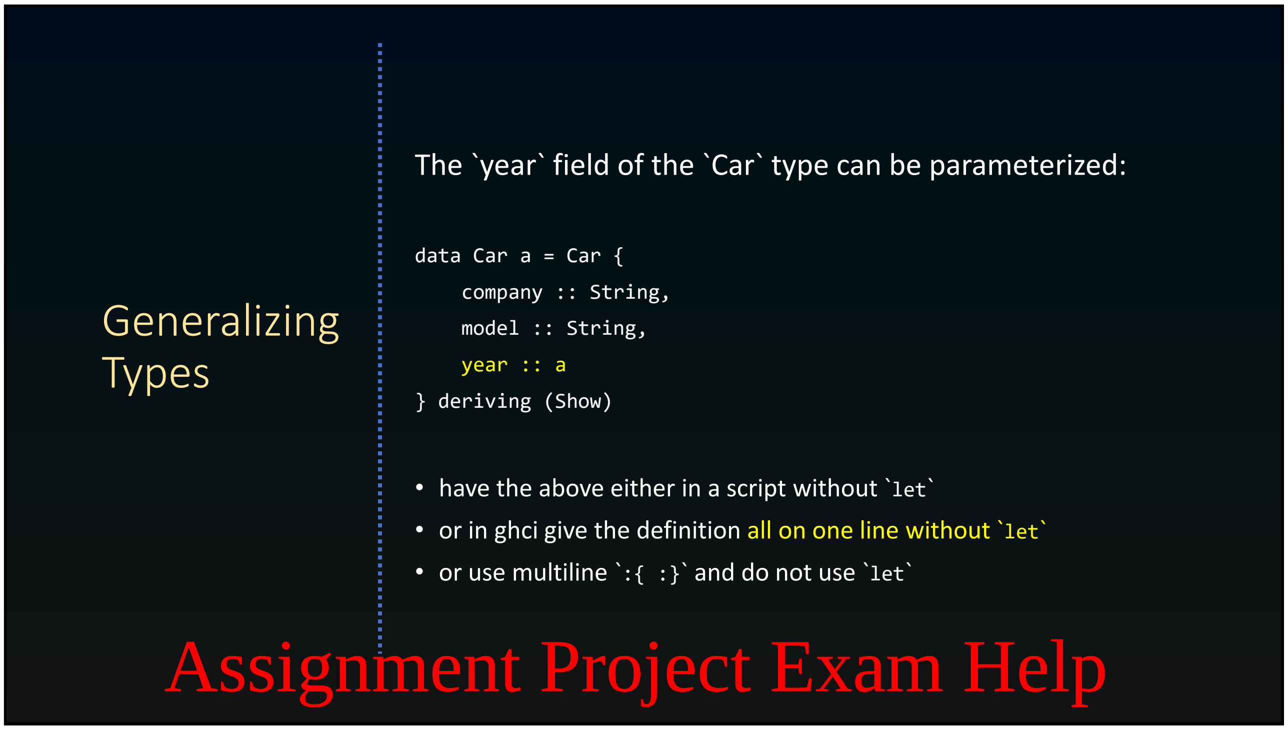
Examples of defining concrete types similar to `Maybe a` where `a` is replaced by concrete type might be:

```
data IntMaybe = INothing | IJust Int

data StringMaybe = SNothing | SJust String

data ShapeMaybe = ShNothing | ShJust Shape
```

21

Working with Polymorphic Types

22



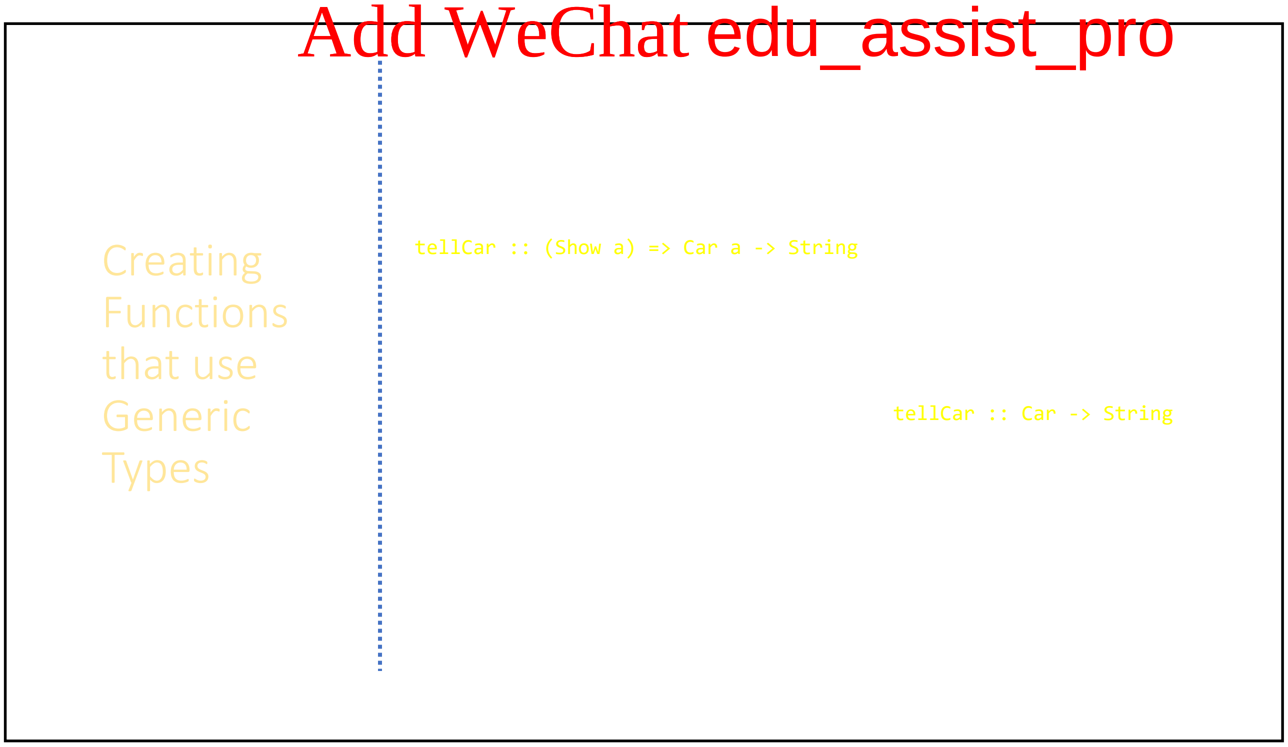
Generalizing Types

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

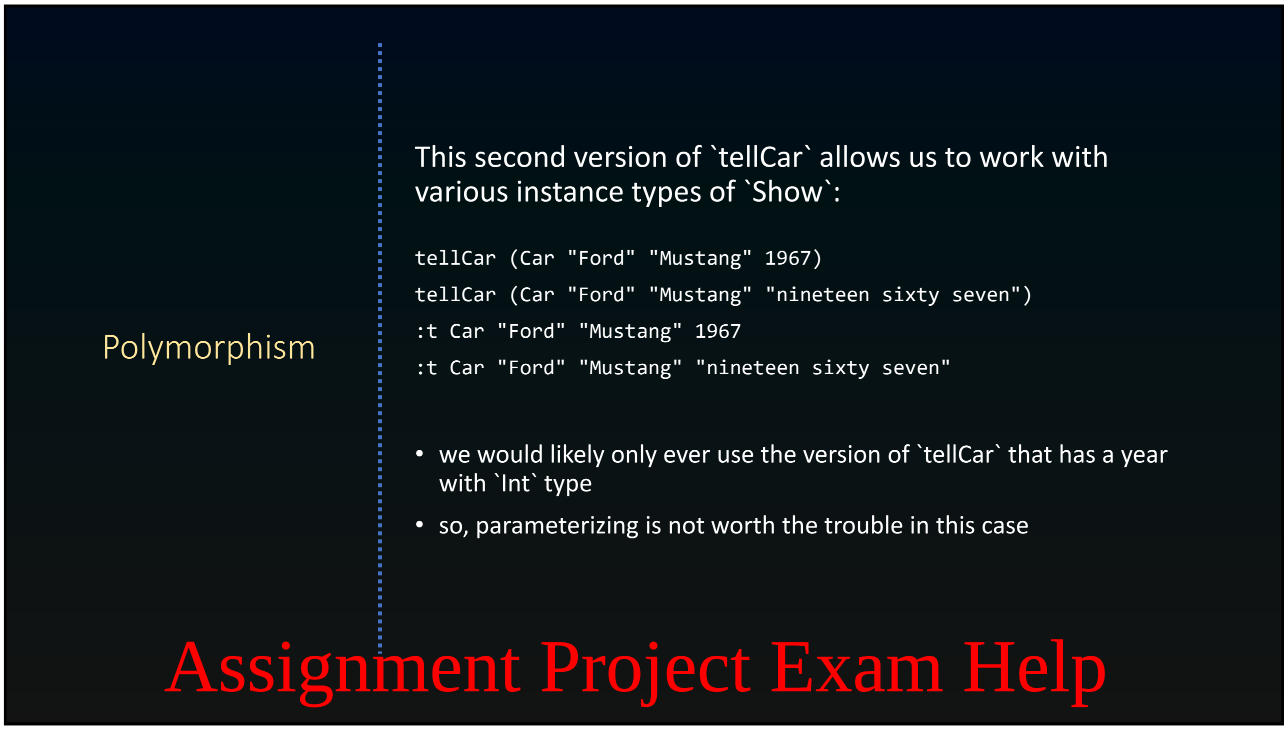


Creating Functions that use Generic Types

```
tellCar :: (Show a) => Car a -> String
```

```
tellCar :: Car -> String
```

24



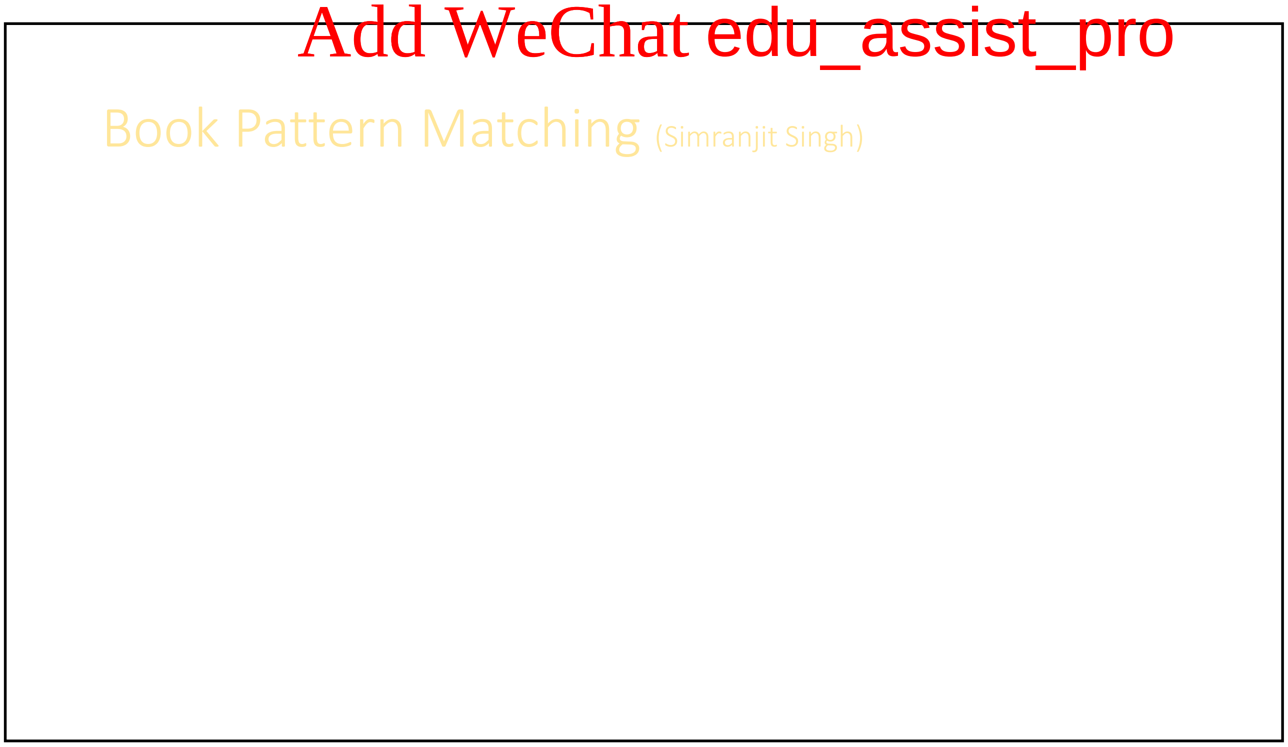
Polymorphism

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)

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

28

— Example 3D Vector —

29

3D Vector

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

type constructors

value constructors

`Vector a` `Vector a a a`

32

Vector Functions

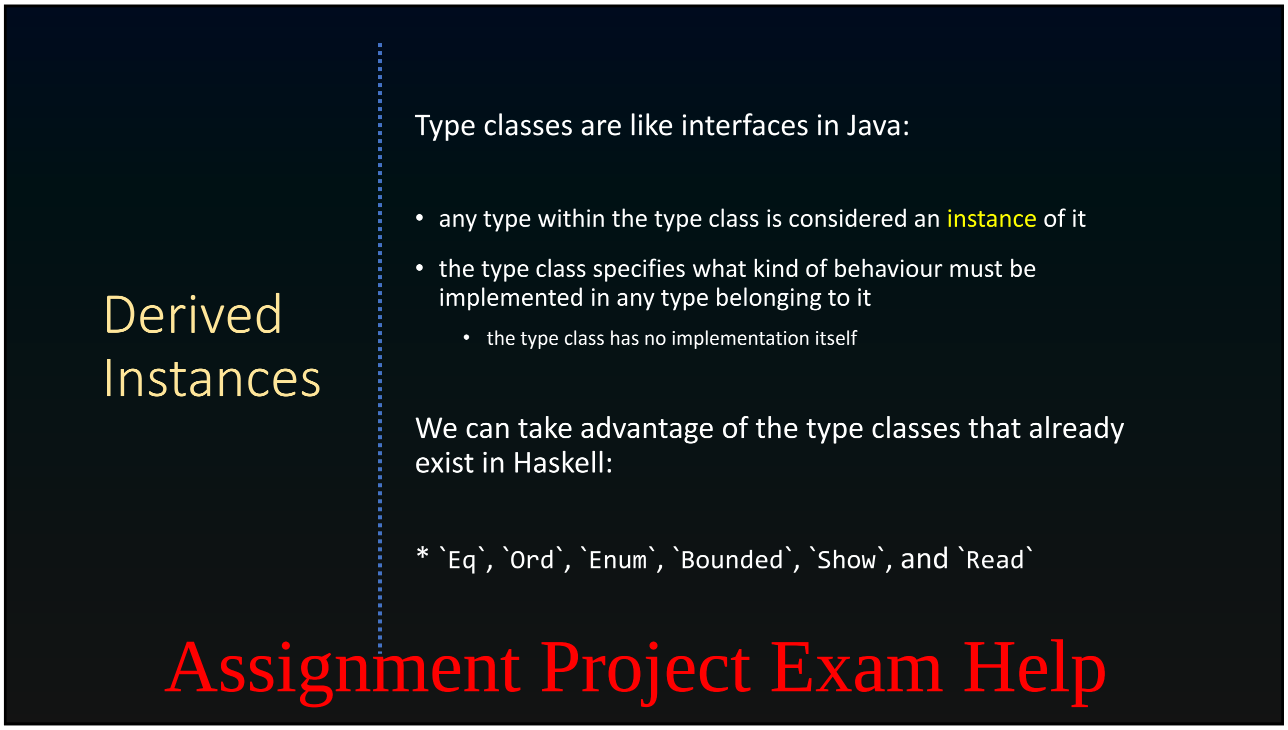
Give some vector functions a try:

```
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

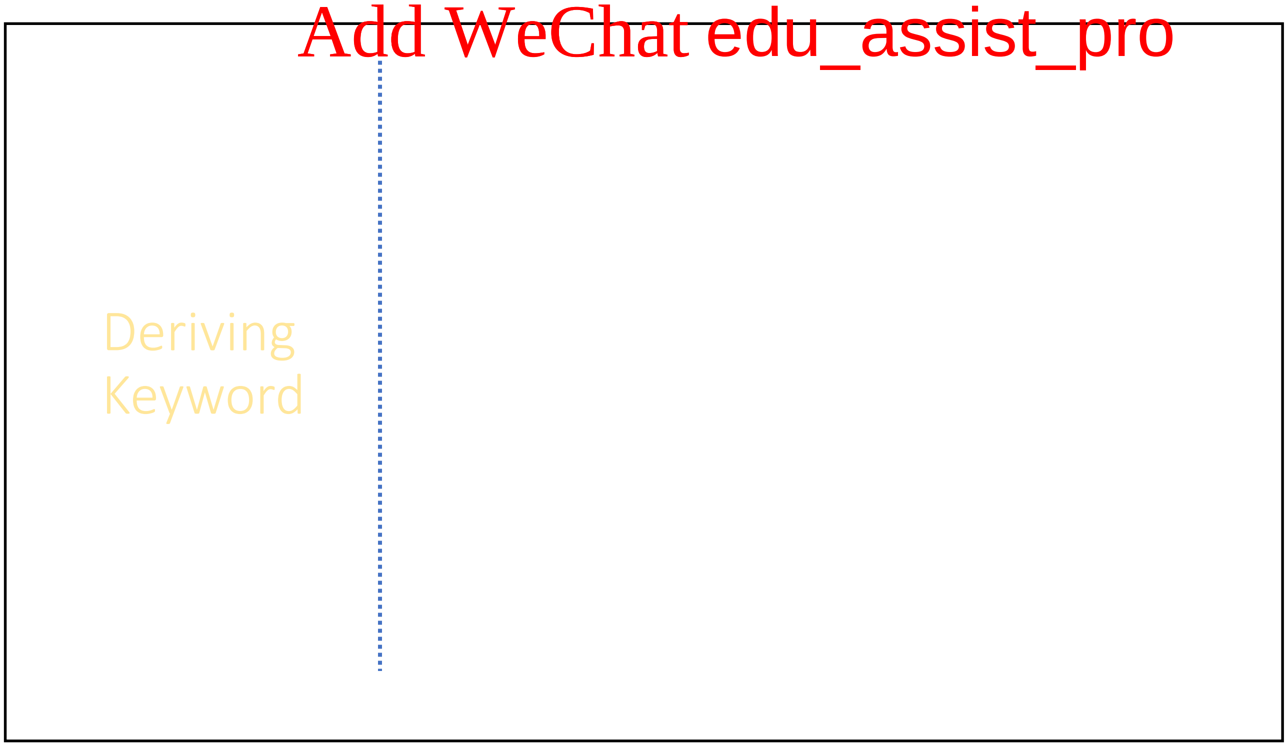
Type classes are like interfaces in Java:

- 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

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

38

— `Show` and `Read` —

39

`Show` and `Read` Derived Instances

Show Read

Show Read

Show Read

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

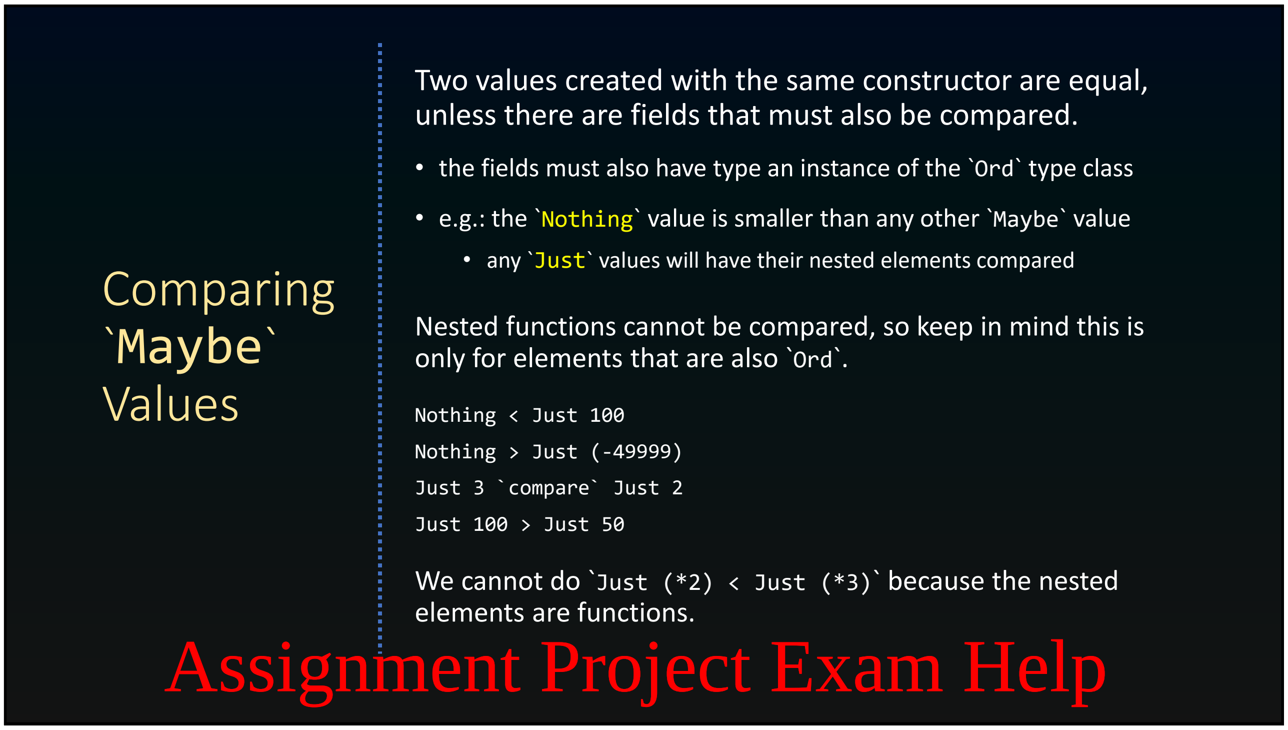
42

— Ordering —

43

Deriving Instances of `Ord`

44



Comparing `Maybe` Values

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

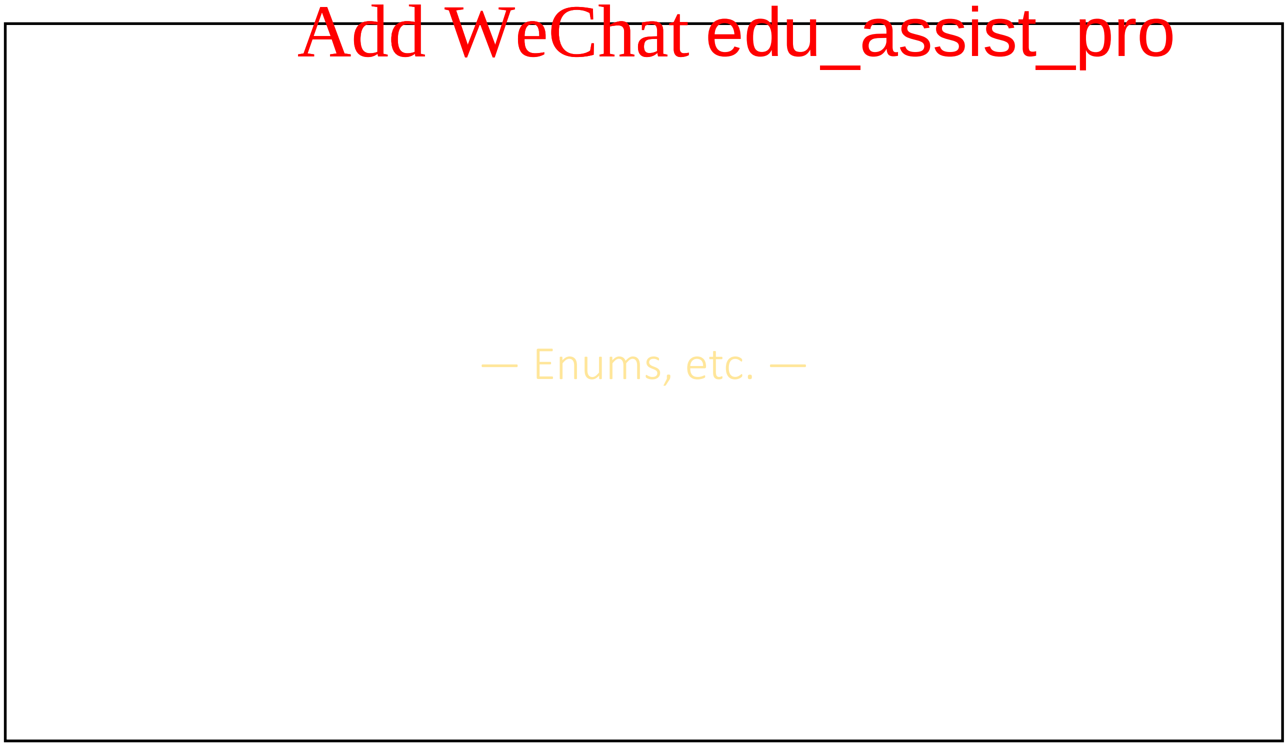
- the fields must also have type an instance of the `Ord` type class
- e.g.: the `Nothing` value is smaller than any other `Maybe` value
  - any `Just` values will have their nested elements compared

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

48

— Type Synonyms —

49

Type Synonyms with `type`

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

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

54

— Two Kinds of Values —

55

Either
Type

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

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

60

Test Lookups

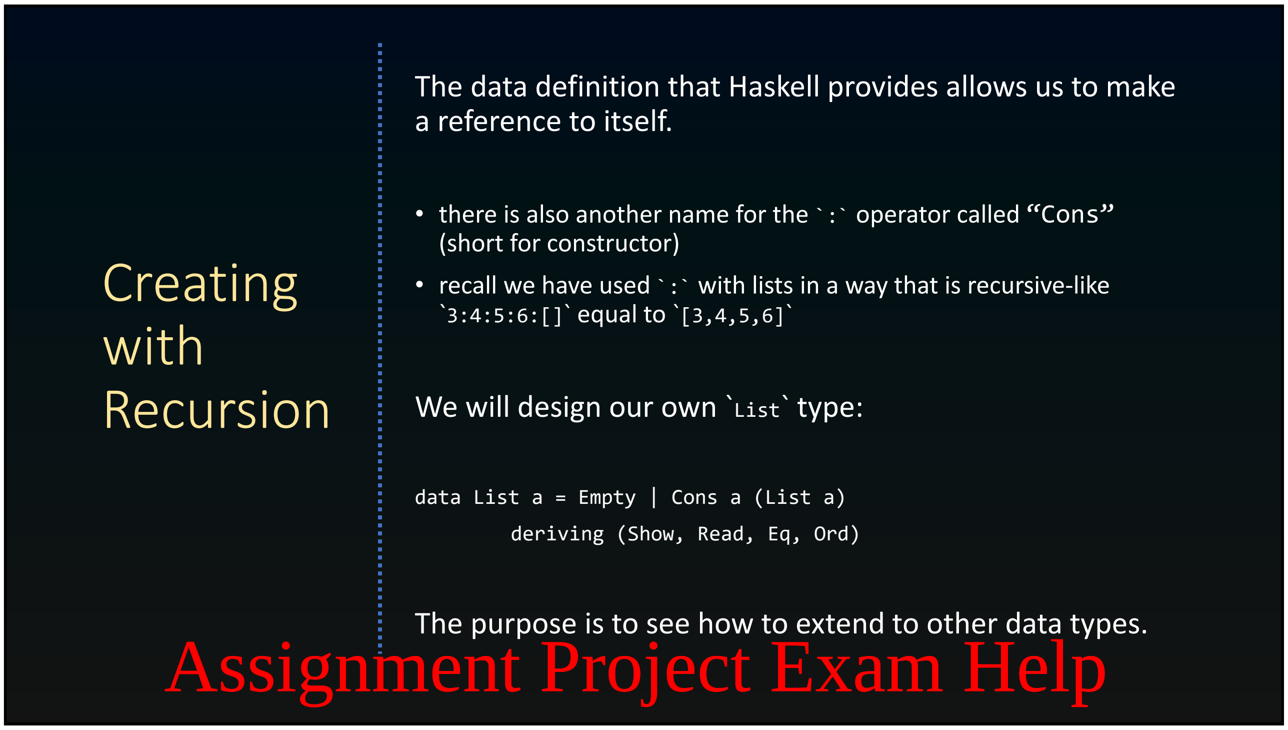
Add the lookup function to script file, then try some lookups:

```
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.

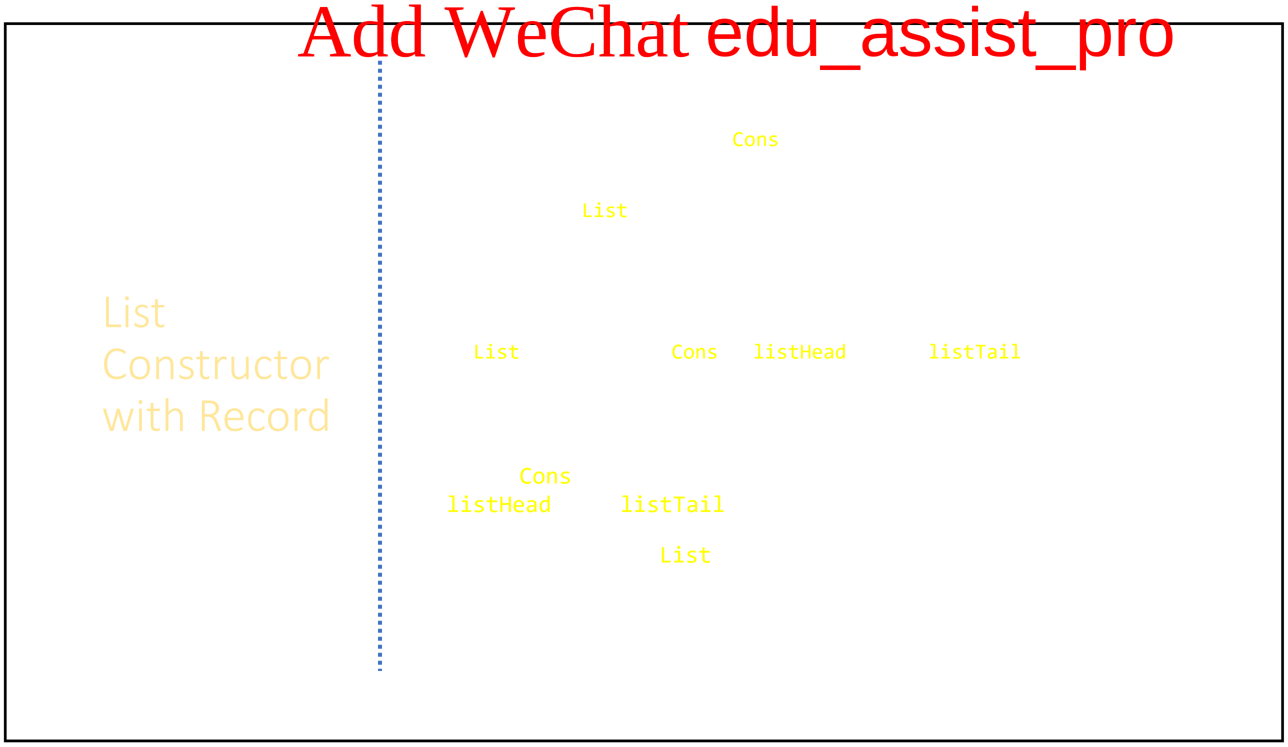
- there is also another name for the `:` operator called "Cons" (short for constructor)
- recall we have used `:` with lists in a way that is recursive-like `3:4:5:6:[]` equal to `[3,4,5,6]`

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

Cons

List

List Cons listHead listTail

Cons
listHead listTail
List

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:

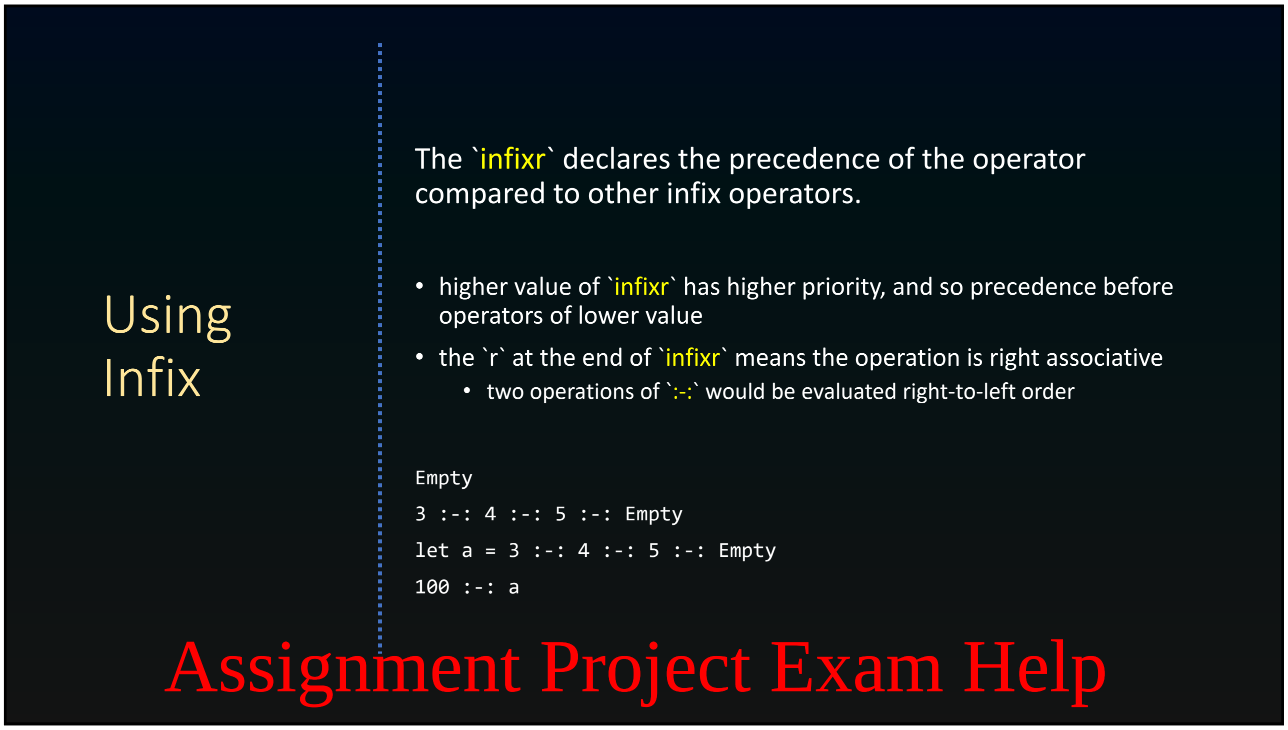
```
5:[]
4:5:[]
3:4:5:[]
```

65

Infix Declarations

data types

66



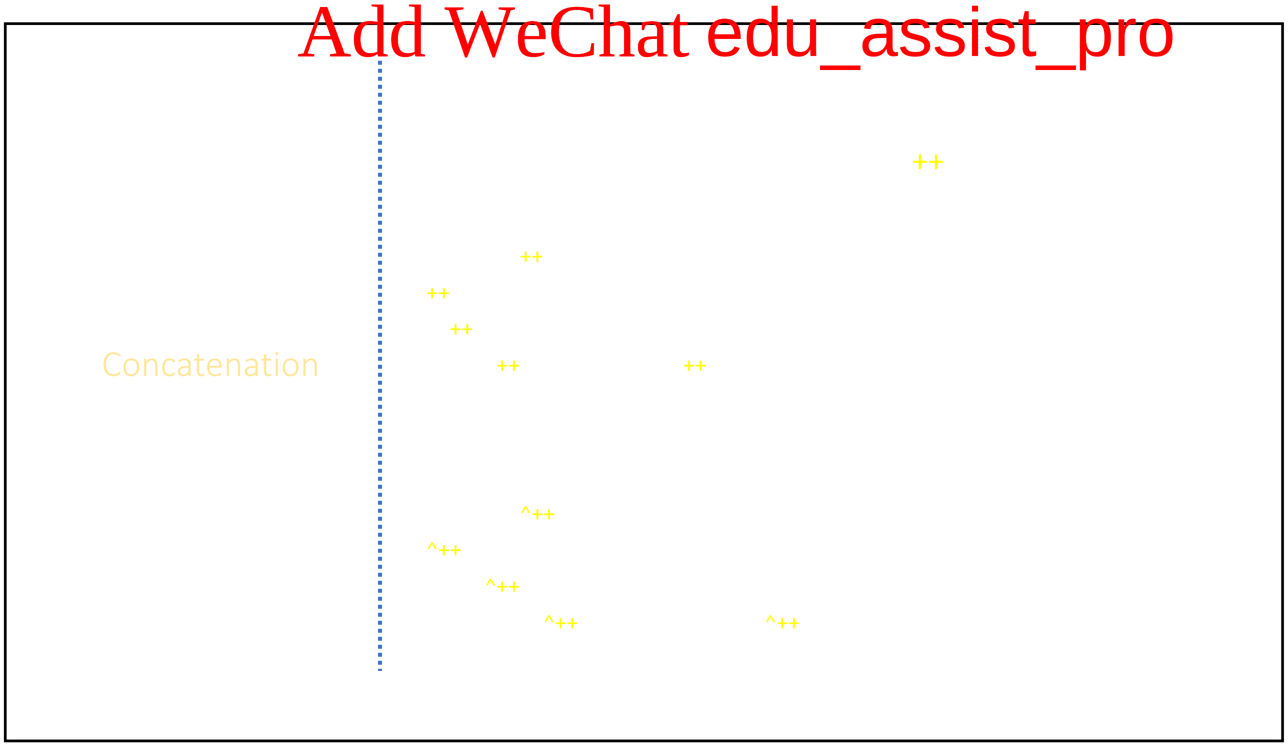
Using Infix

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

++

++
++
++
++ ++

^++
^++
^++
^++ ^++

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

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)

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

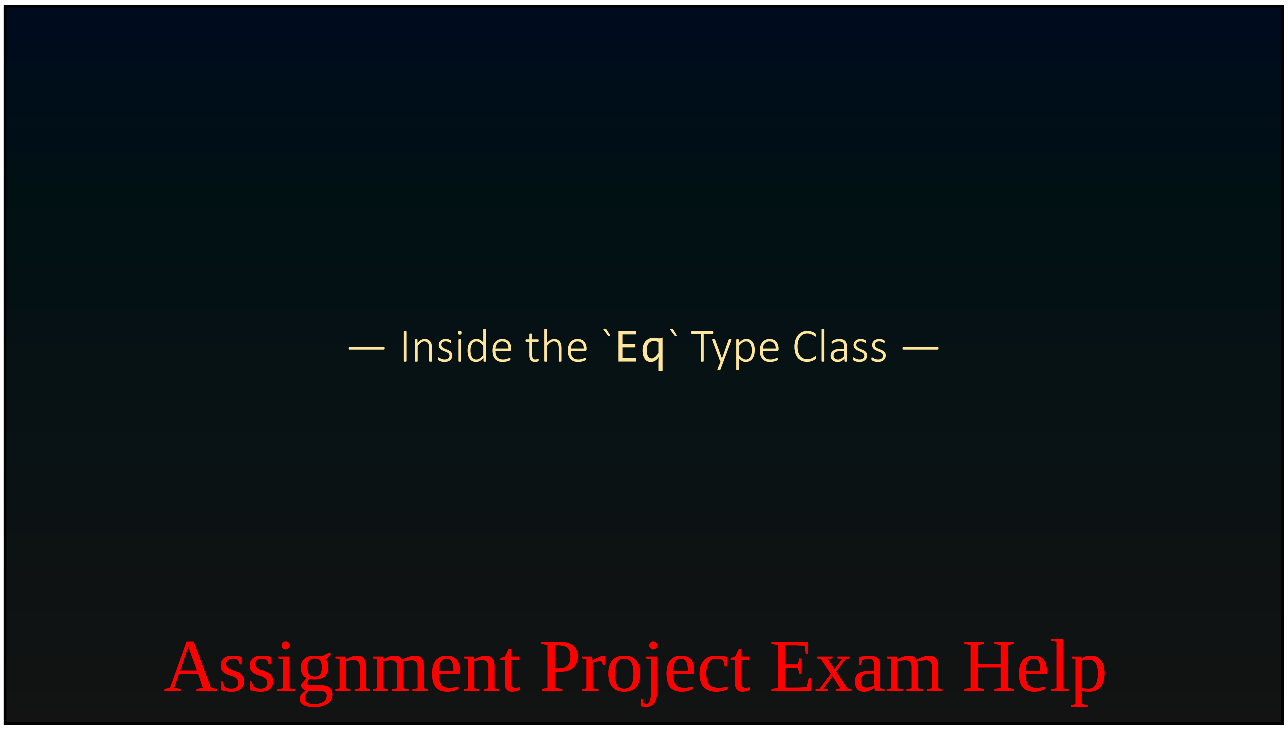
- one element at a time (from the right of the list)

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

75

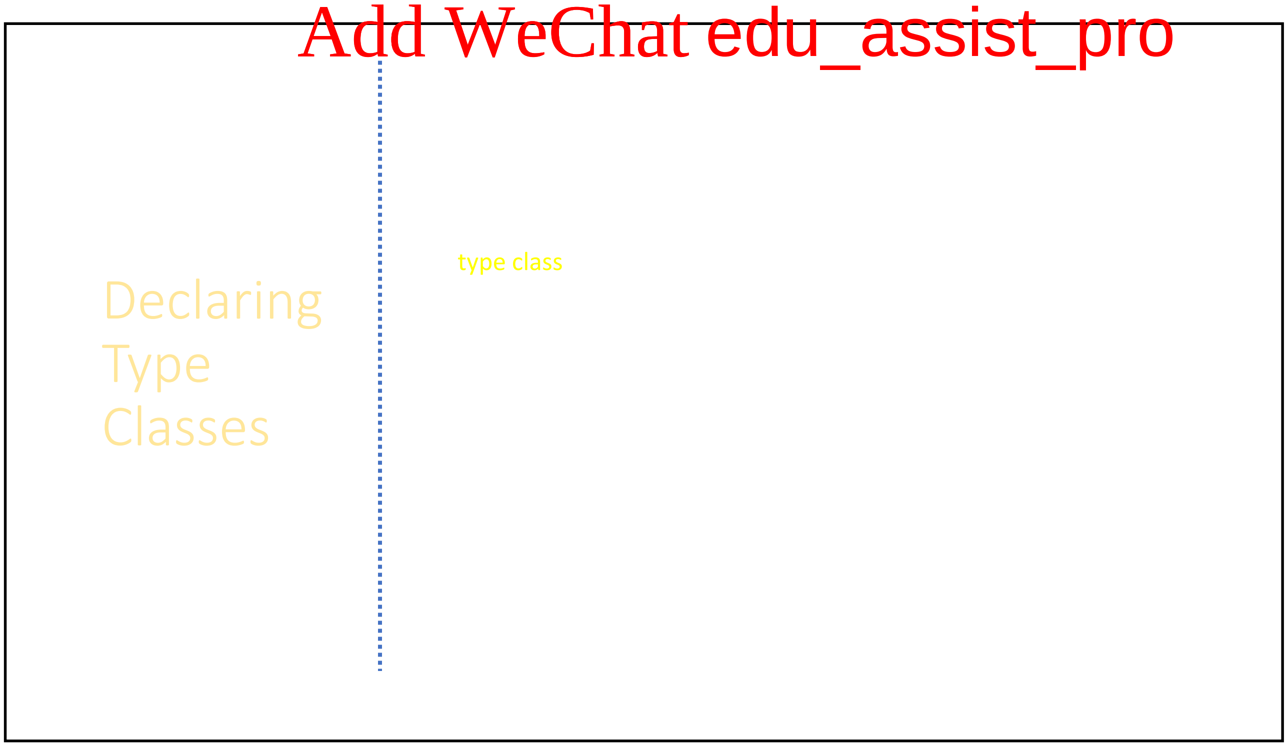
Check for Elements in a Tree

76



— Inside the `Eq` Type Class —

77



Declaring Type Classes

type class

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

minimal complete definition

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:

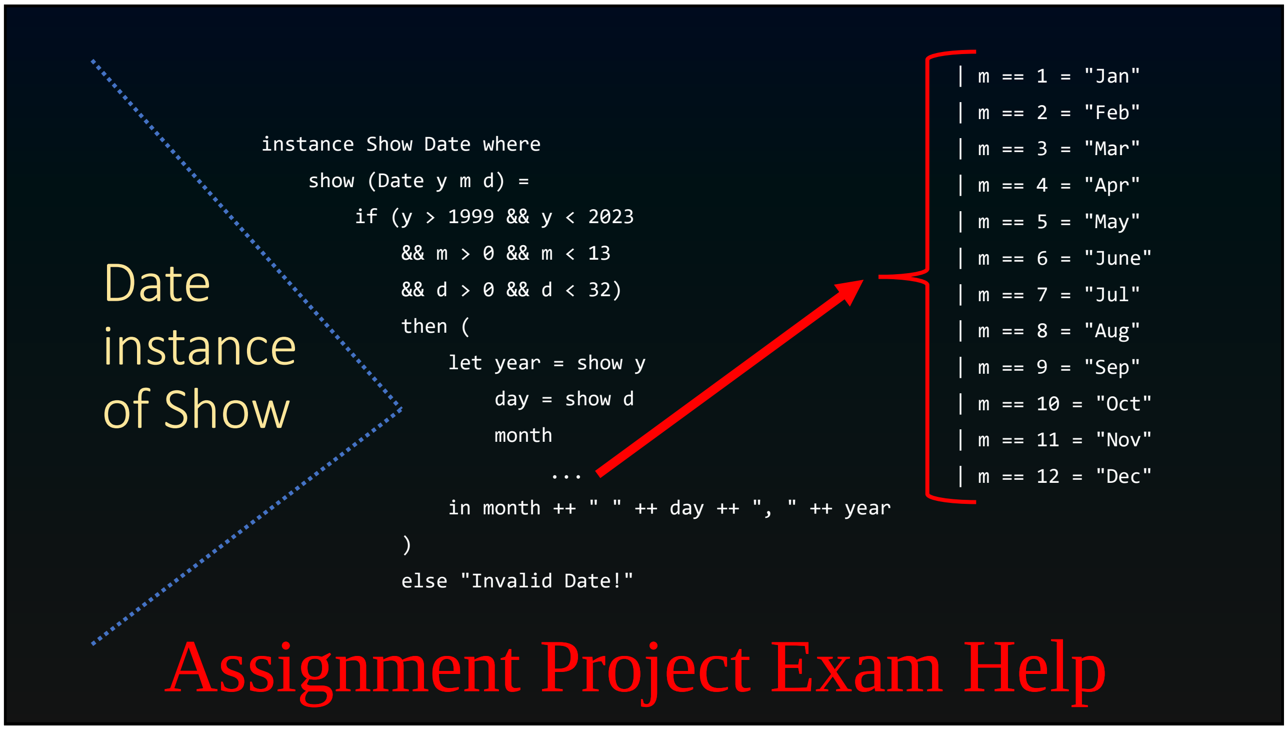
```
Red == Red
Red == Yellow
Red `elem` [Red, Yellow, Green]
[Red, Yellow, Green]
```

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

83

Date Instance of Show
(David Semke)

84

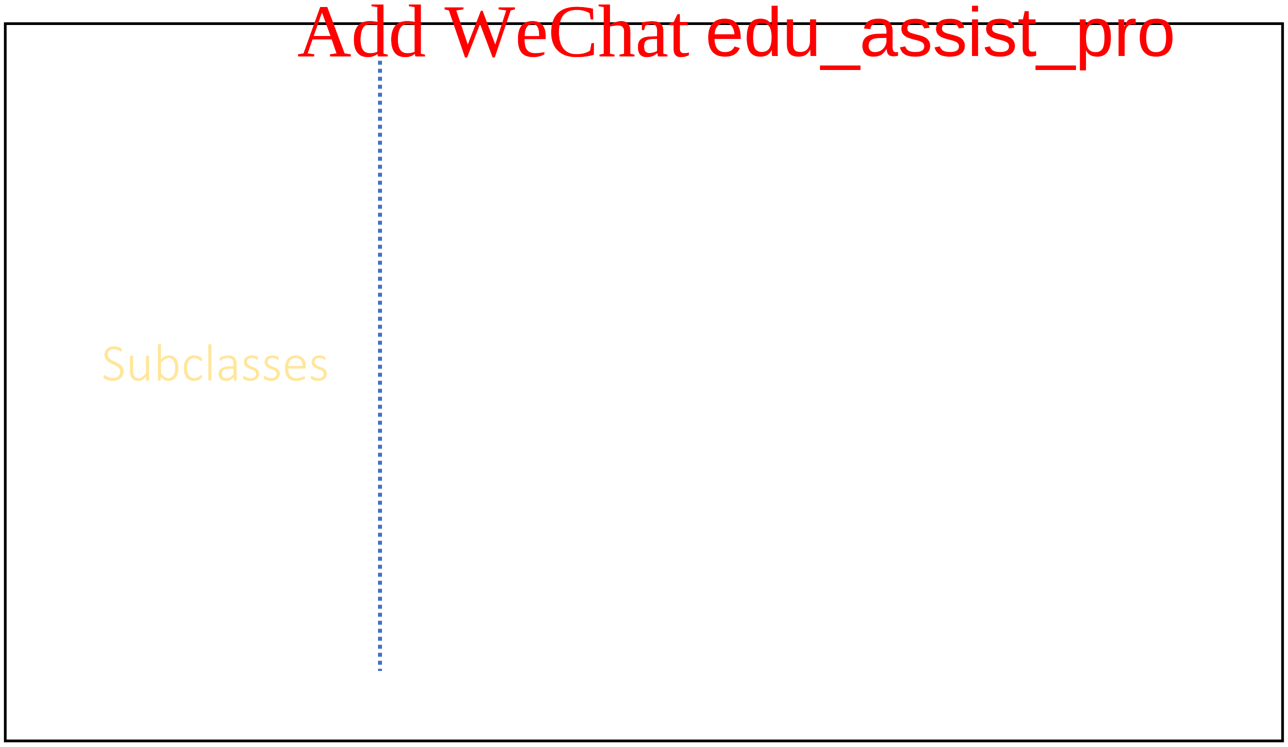


Date instance of Show

```
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
                        ...
                in month ++ " " ++ day ++ ", " ++ year
            )
            else "Invalid Date!"
```

```
| 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"
```

85



Subclasses

86

— Parameterized Types as Instances of Type Classes —

87

Toward Parameterization

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

Maybe a   a

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

`:info`

`:info Maybe`

92

— `YesNo` Type Class —

93

Boolean-like Type Class

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

```
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)

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

`Maybe a`

`Tree a`

`TrafficLight`

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

100

— The Functor Type Class —

101

Functors

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`

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

m a m b

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

112

— Either as a Functor —

113

Two Type Parameters

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

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

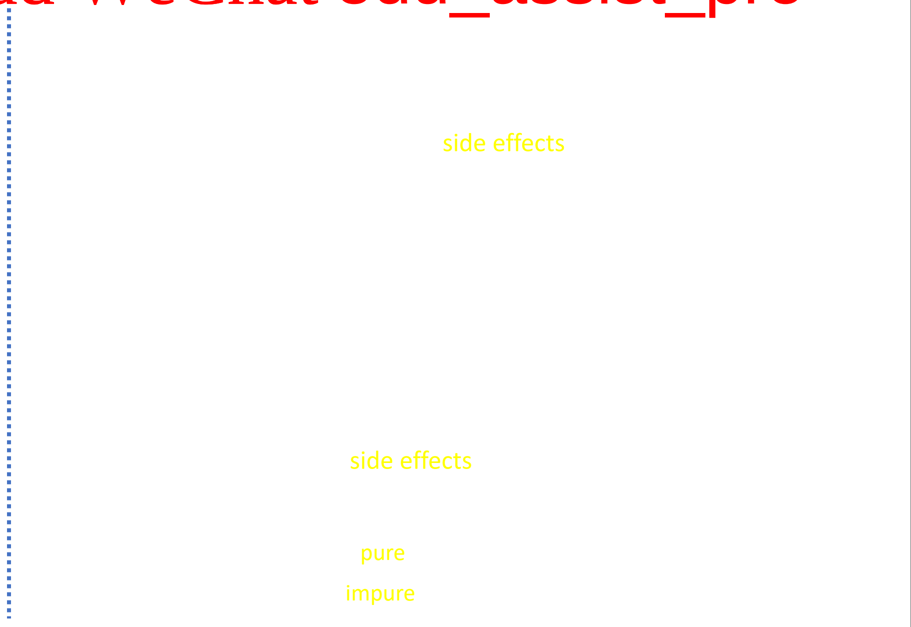
— Chapter 8: Input and Output —

120

— Separating Pure from Impure —

121

I/O has Side Effects



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

`ghc --make hello`

`hello.exe`

`./hello`

`main`

`main`

`:script hello.hs`

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.

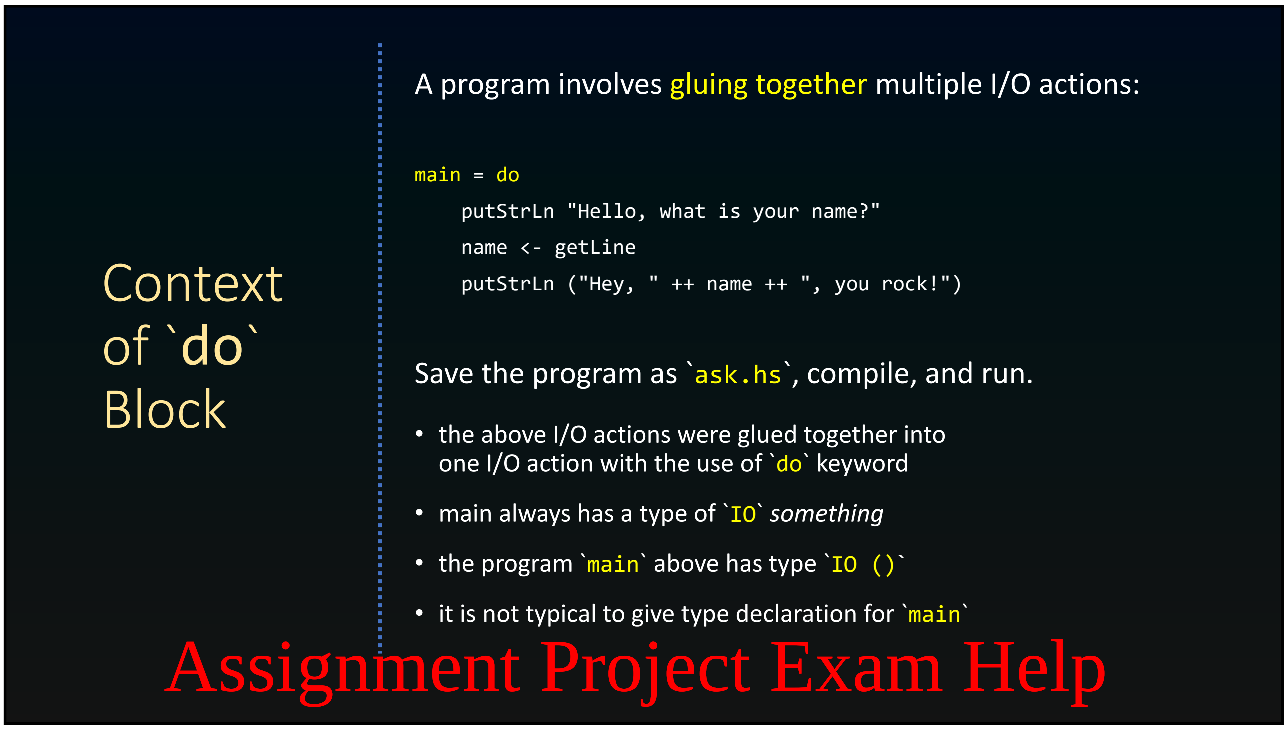
- printing to the terminal does not have any meaningful side effect, so the empty tuple represents a dummy value (`()` is also the description of its type)

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

125

— Gluing I/O Actions Together —

126



Context of `do` Block

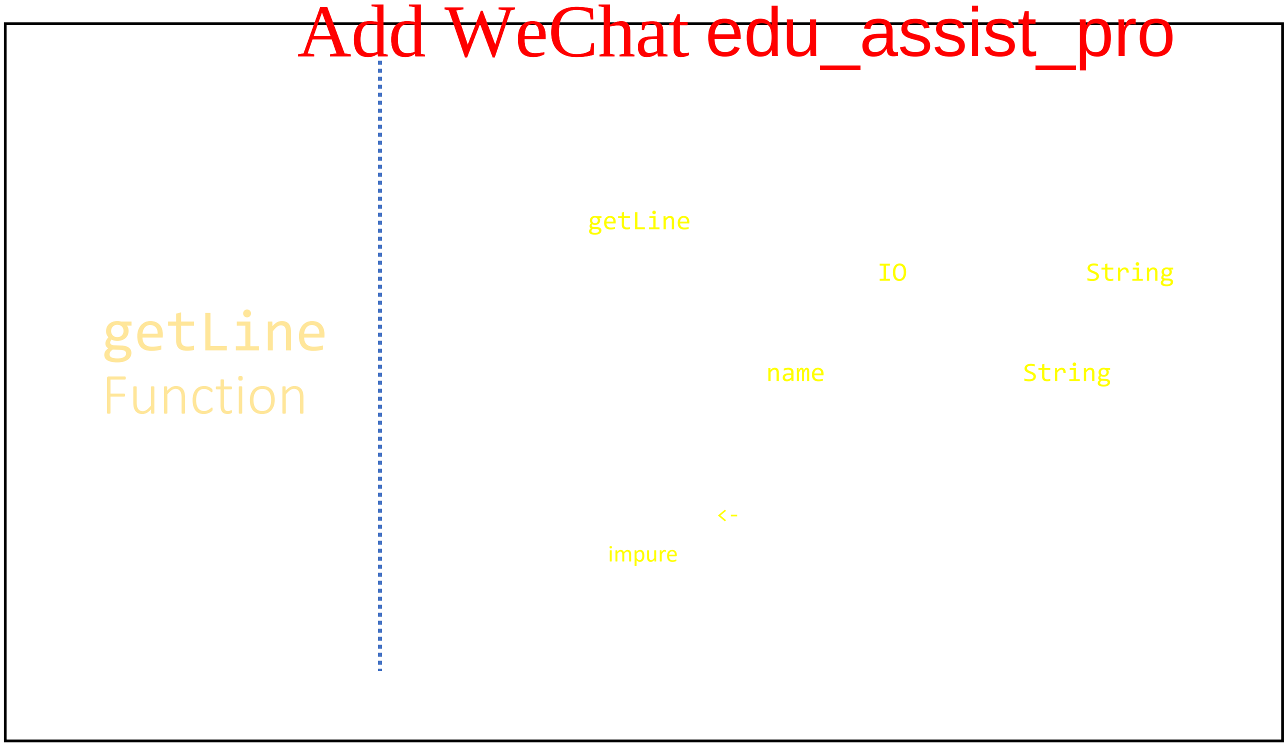
A program involves gluing together multiple I/O actions:

```
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

getLine

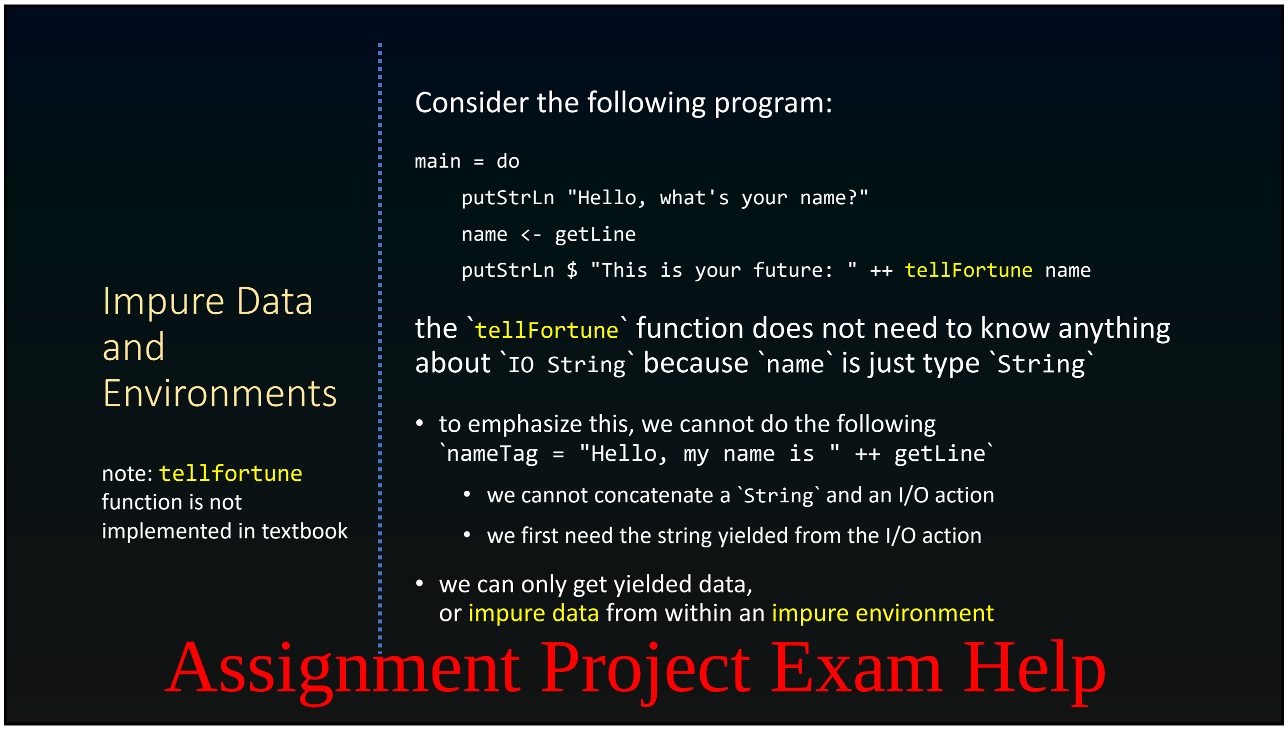
IO String

name String

<-

impure

128



Impure Data and Environments

note: tellfortune function is not implemented in textbook

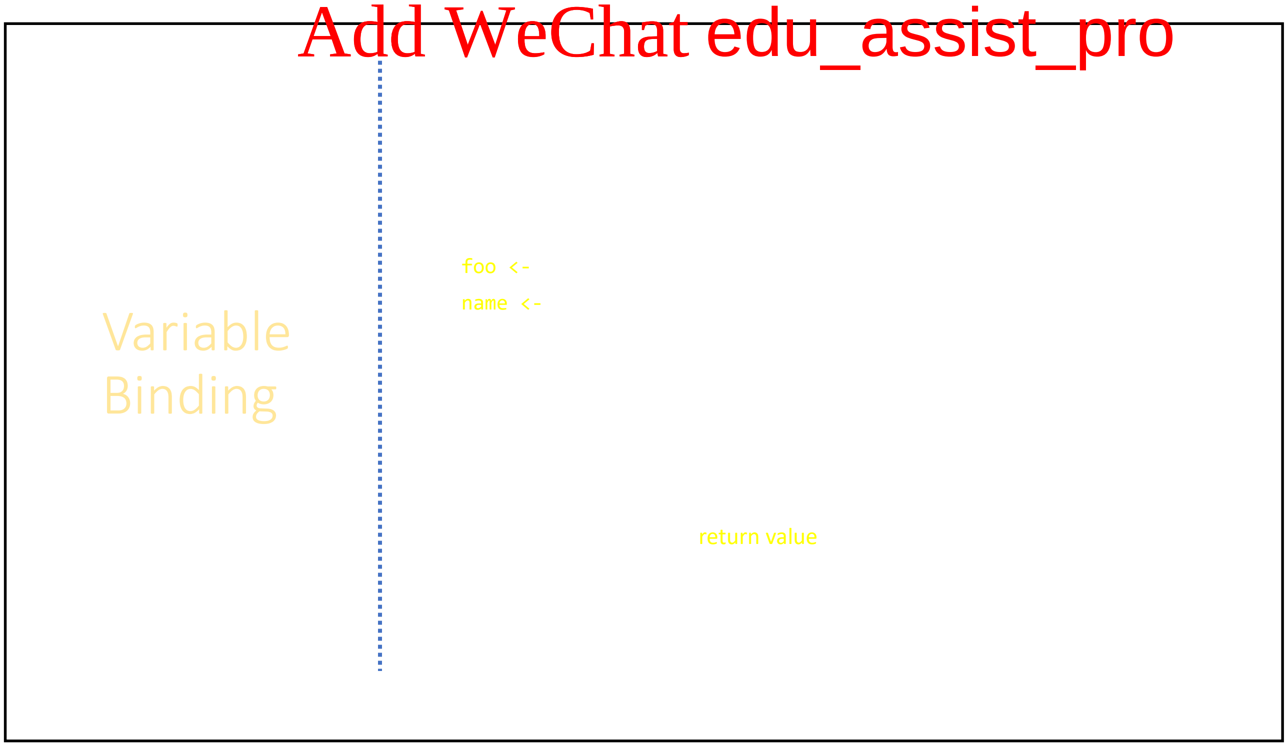
Consider the following program:

```
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

```
    foo <-
    name <-
```

return value

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.

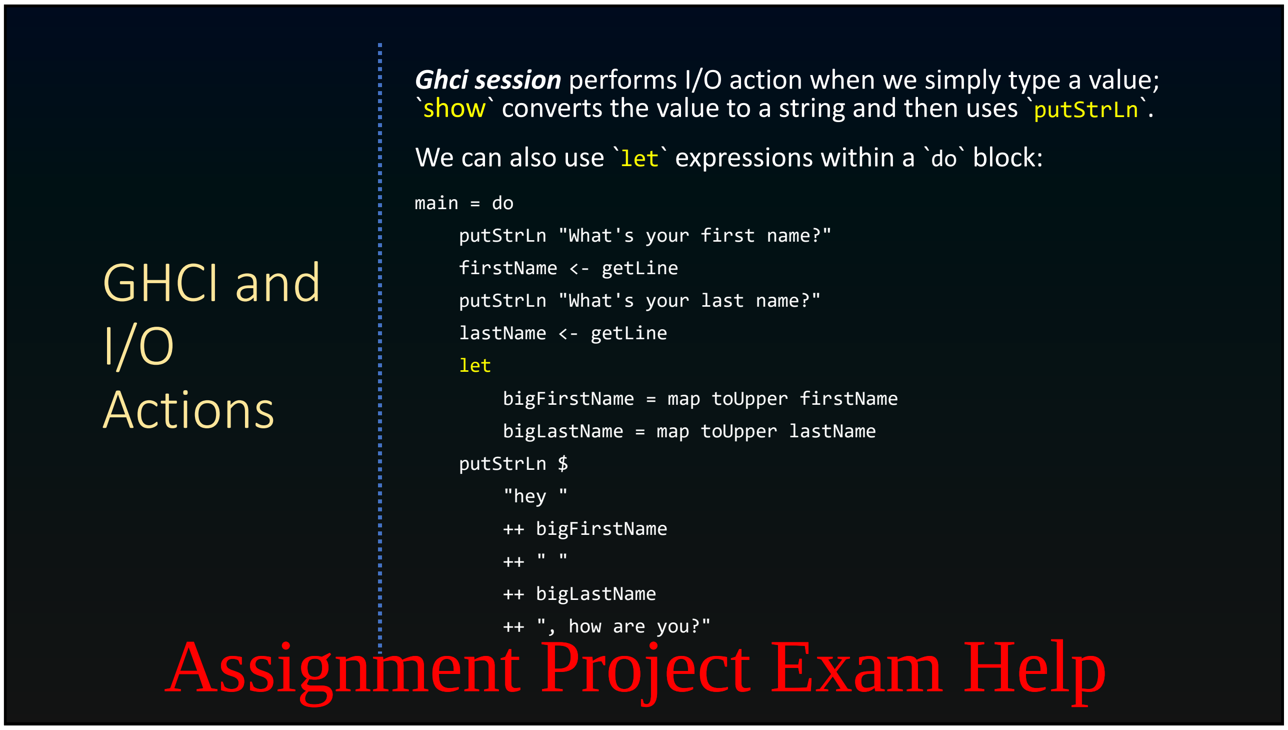
I/O actions will be performed:

- 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)

132



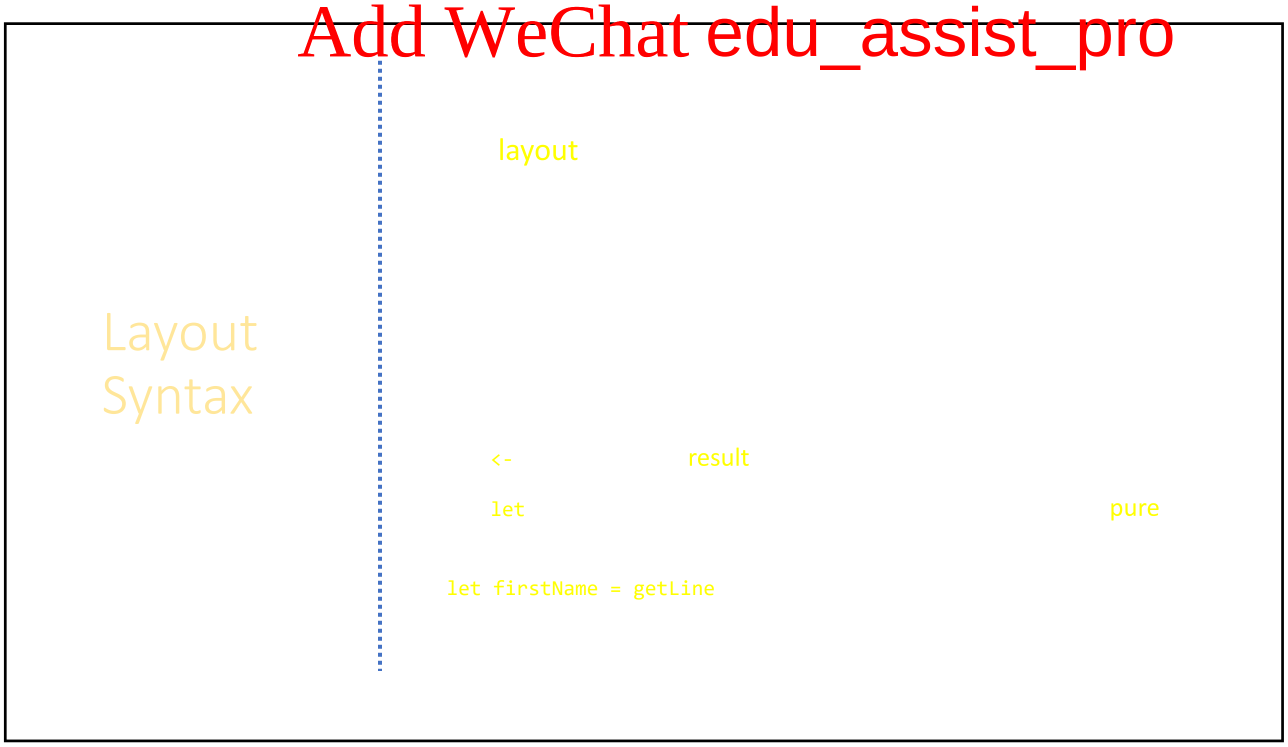
GHCI and I/O Actions

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

layout

<- result

let pure

`let firstName = getLine`

134

— Reverse Strings in I/O —

135

Recursive IO Actions

```
main





                   reverseWords
            main
    putStrLn $ "exiting main"

reverseWords
reverseWords
```

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

return

return

return

return

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

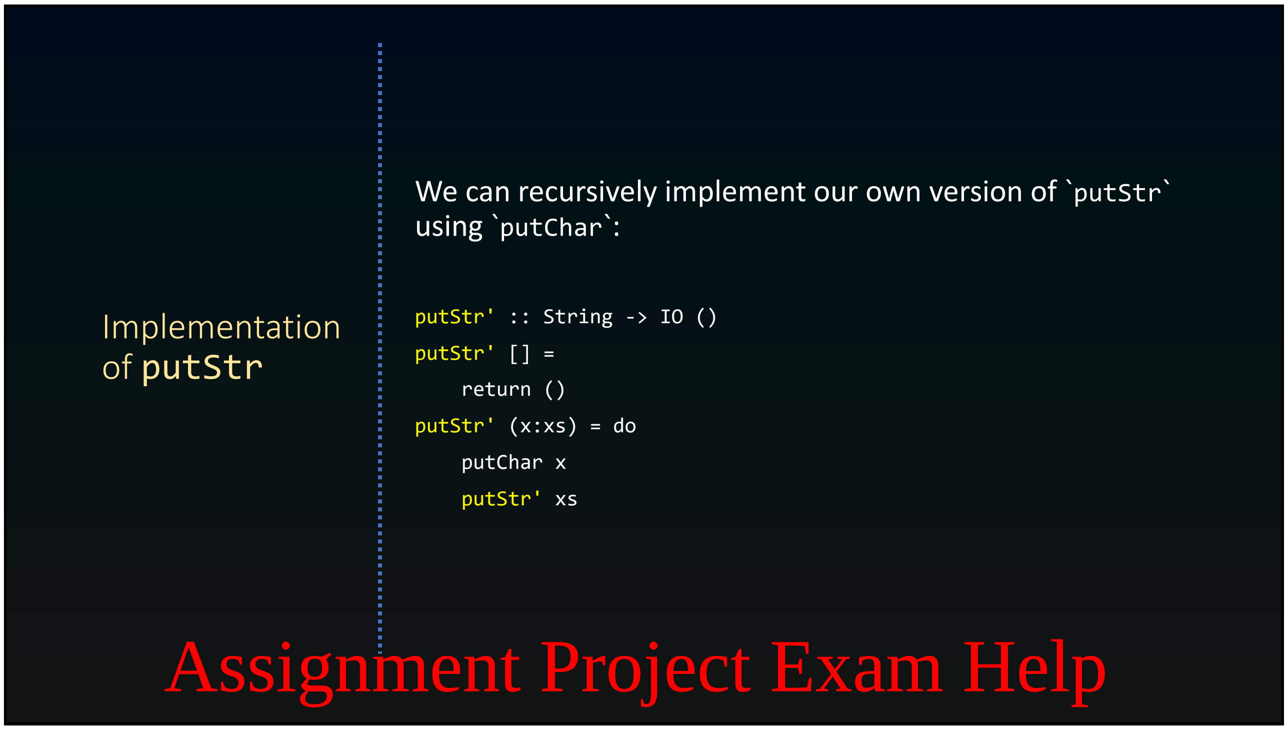
141

One Output Character

putChar

```
    putChar
    putChar
    putChar
```

142

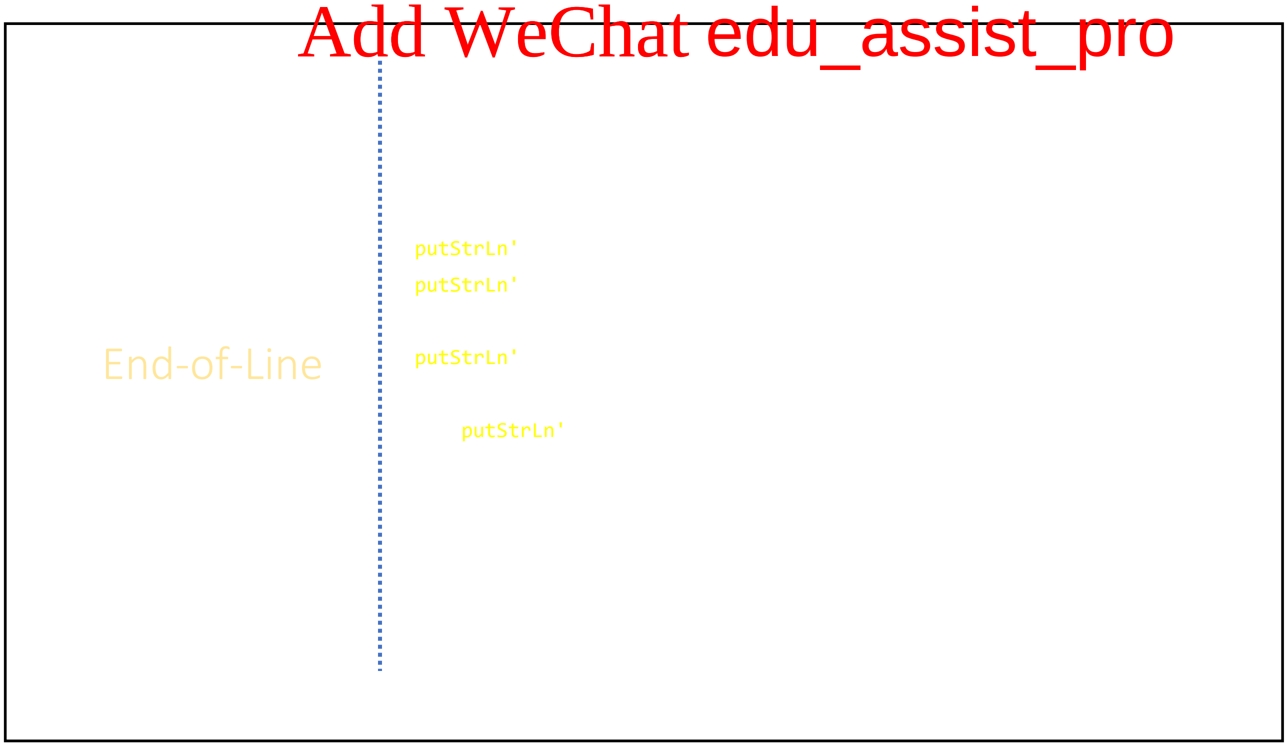


Implementation of `putStr`

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

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

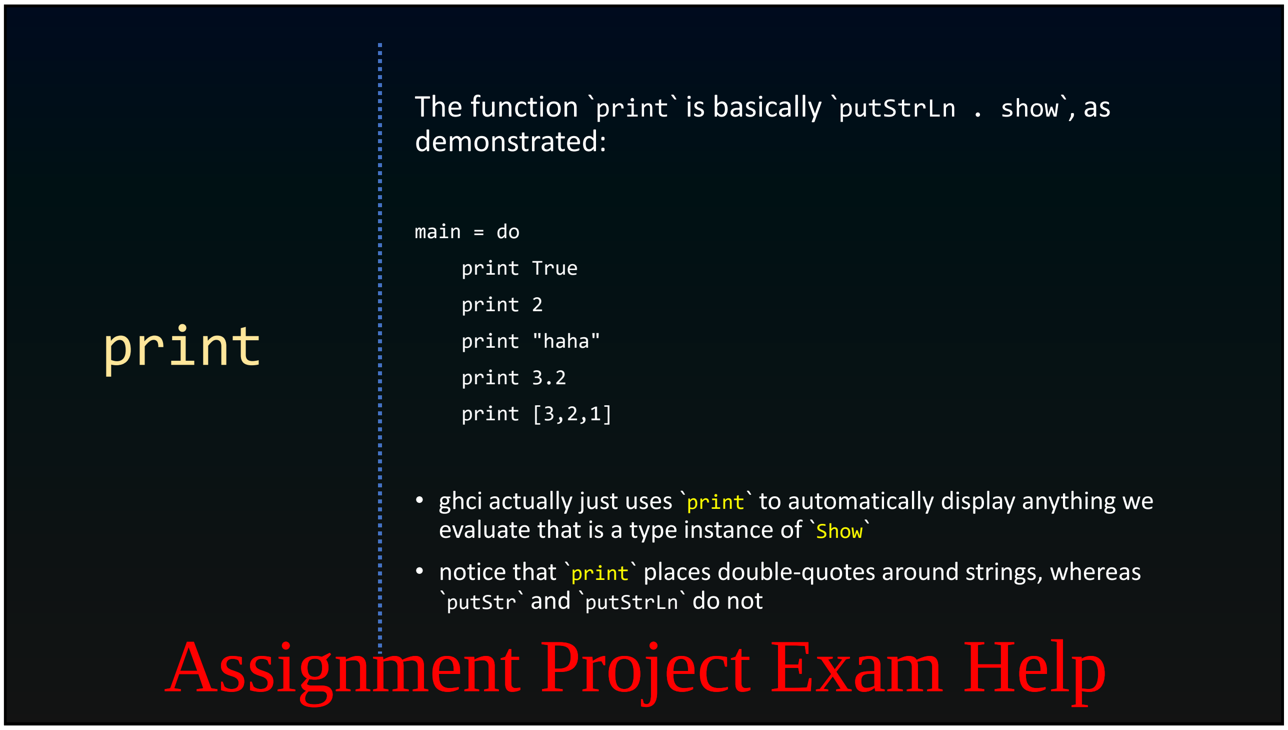
143



End-of-Line

```
putStrLn'
putStrLn'

putStrLn'

    putStrLn'
```

144



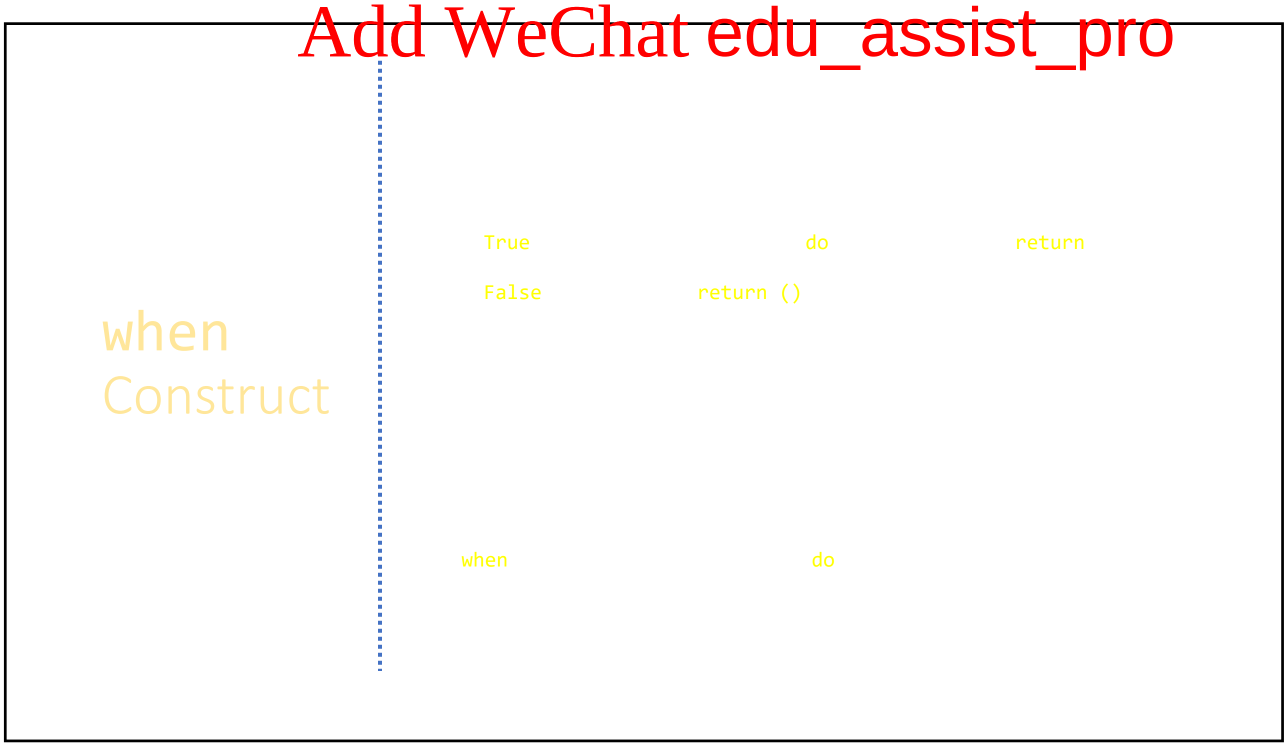
print

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

True do return

False return ()

when do

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

sequence

getLine

sequence

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

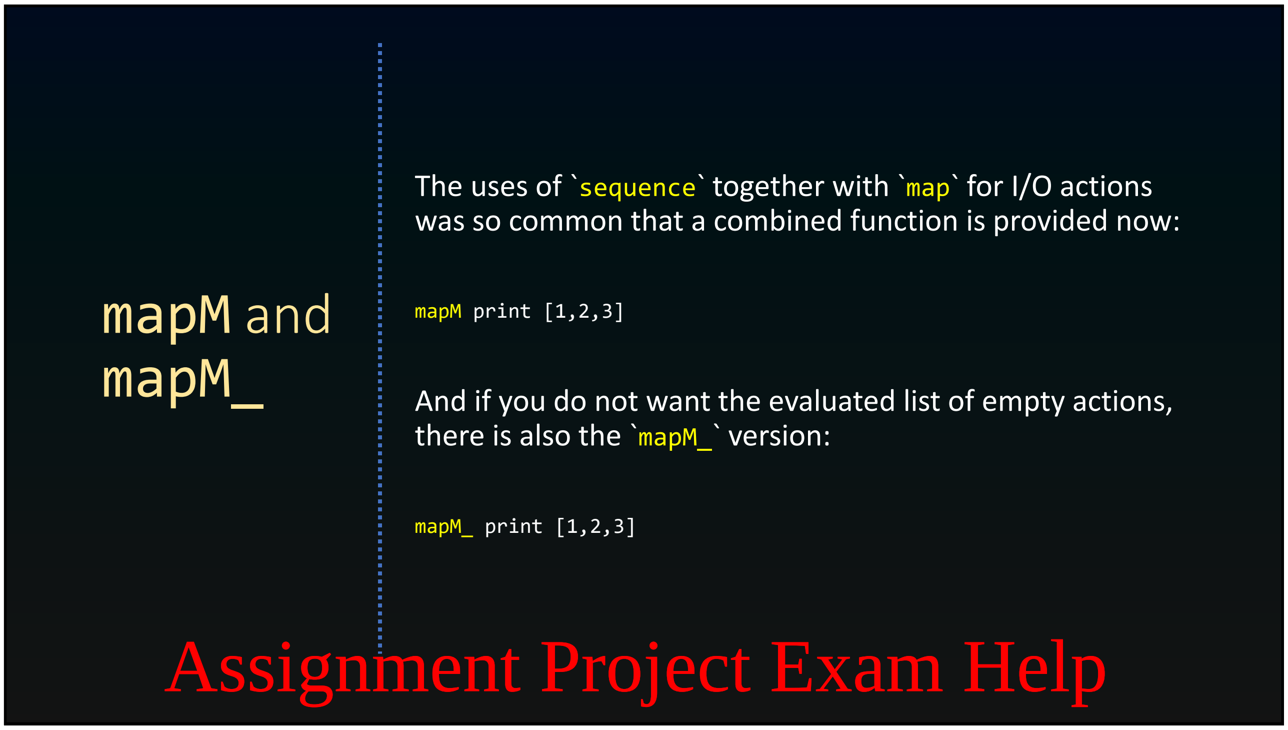
_ <-

_ <-

_ =

_ = 3

150



mapM and mapM_

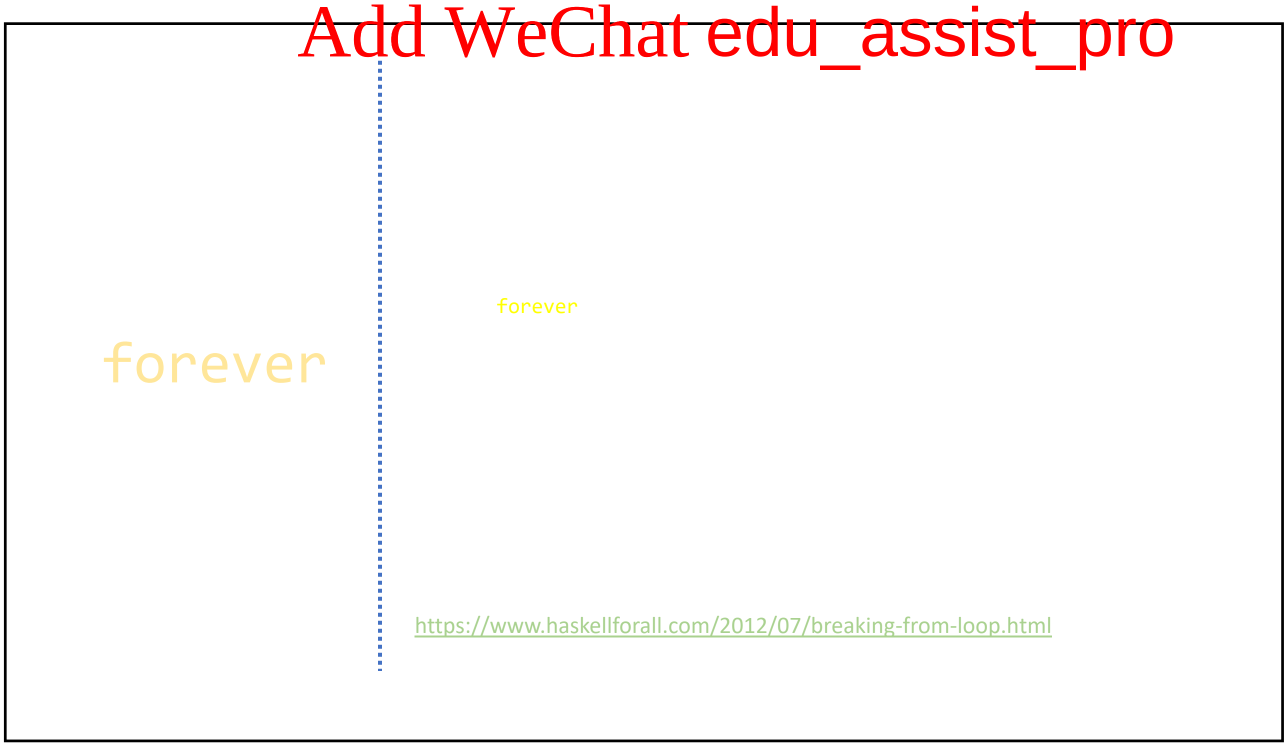
The uses of `sequence` together with `map` for I/O actions was so common that a combined function is provided now:

```
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

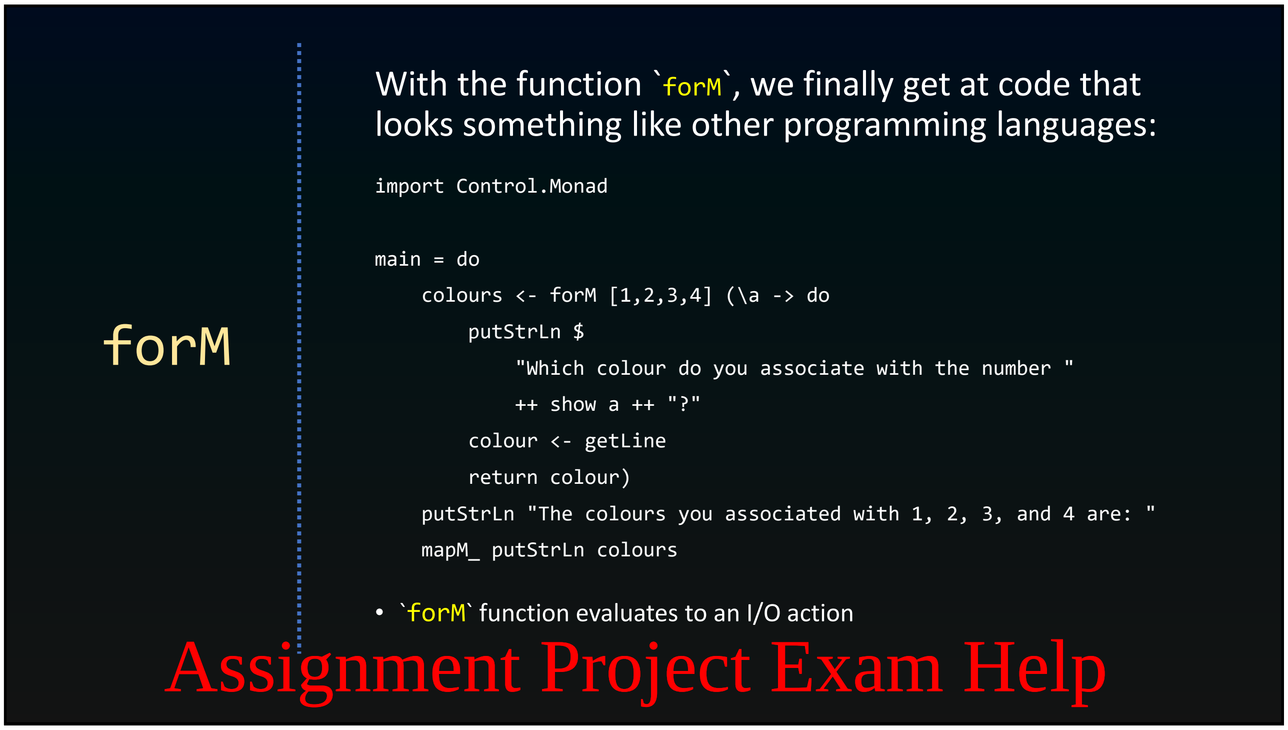


forever

```
forever
```

https://www.haskellforall.com/2012/07/breaking-from-loop.html

152



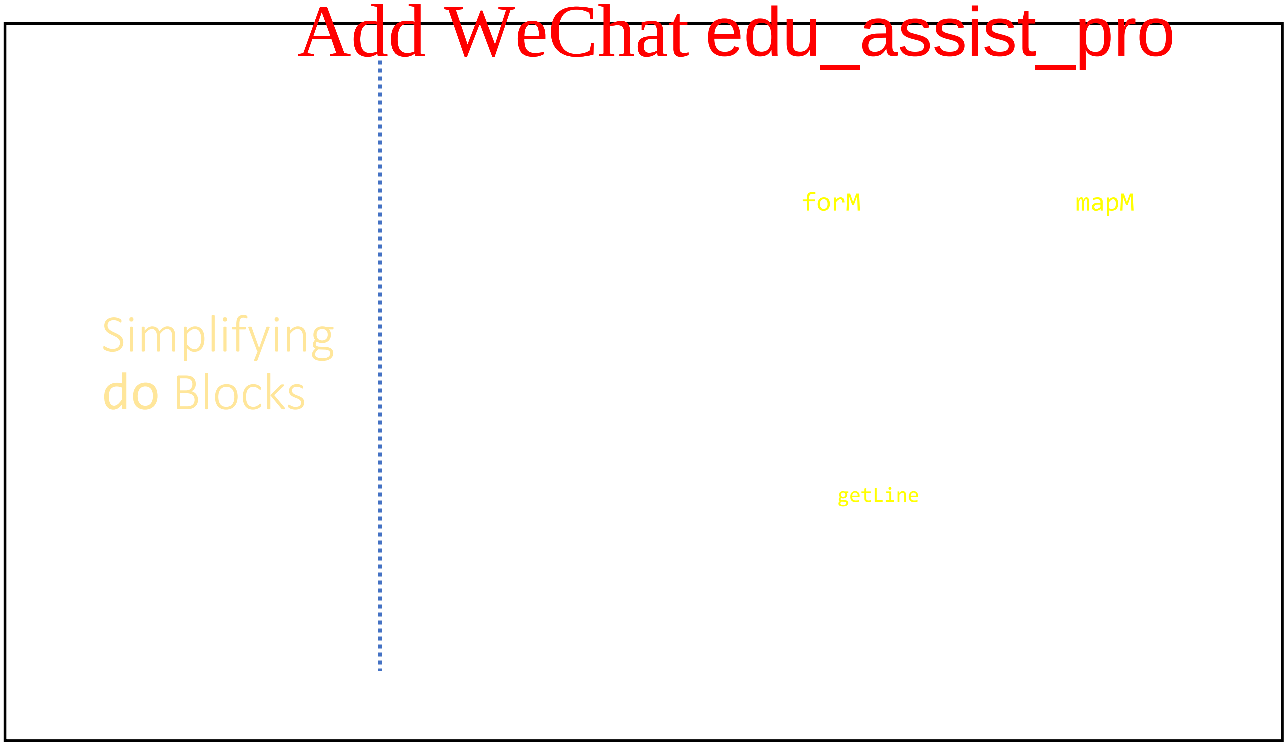
forM

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

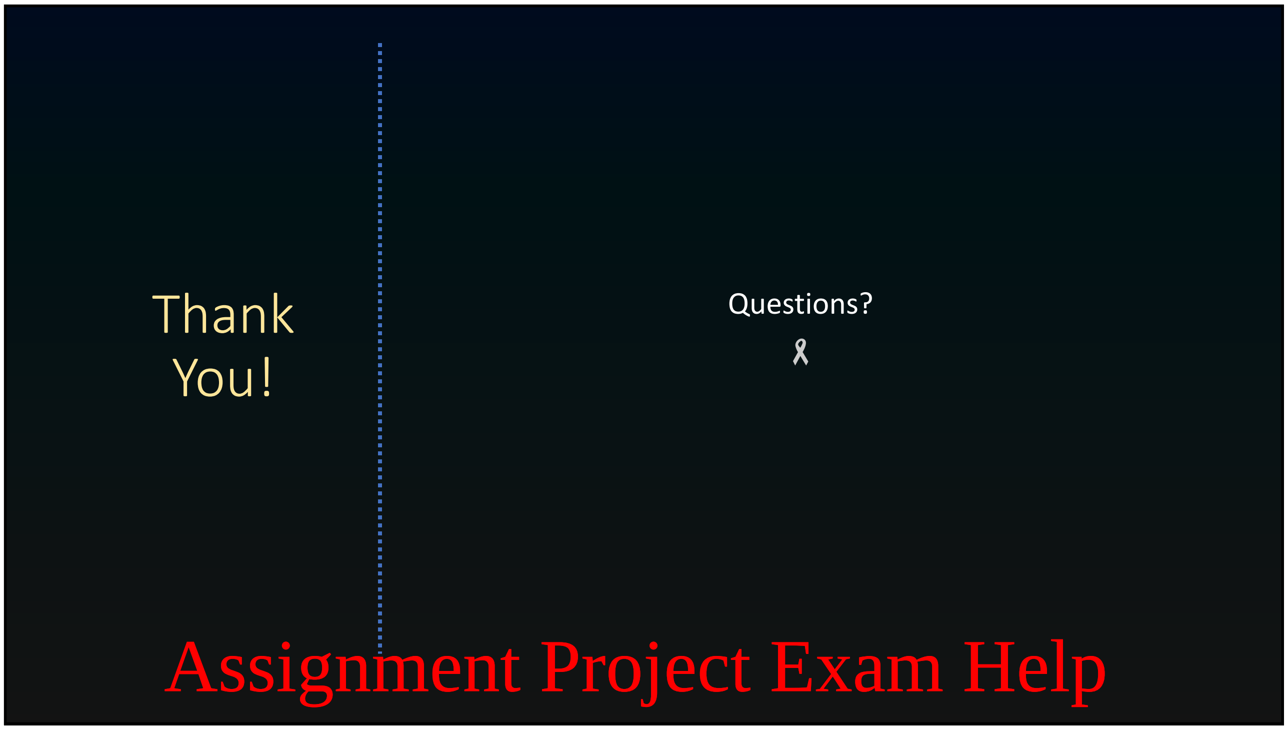


Simplifying do Blocks

forM mapM

getLine

154



Thank You!

Questions?