Programming in Haskell: Lecture 19

S P Suresh

October 21, 2019

User-defined data types

- The `data` keyword is used to define new types

User-defined data types

- The `data` keyword is used to define new types
- Enumerated data types:

```
data Bool = False | True
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

User-defined data types

- The `data` keyword is used to define new types
- Enumerated data types:

```
data Bool = False | True
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Data types with parameters:

```
data Shape = Circle Double | Square Double
                           | Rectangle Double Double
shapes :: [Shape]
shapes = [Circle 3.0, Square 4.0, Rectangle 3.0 4.0]
```

Functions on data types

- Functions can be defined using pattern matching

```
weekend :: Day -> Bool
weekend Sat = True
weekend Sun = True
weekend _ = False

area :: Shape -> Double
area (Circle r) = pi*r*r
area (Square x) = x*x
area (Rectangle l w) = l*w
    where pi = 3.1415927
```

Functions on data types

- What about the following function?

```
weekend2 :: Day -> Bool
weekend2 d
    | (d == Sun || d == Sat) = True
    | otherwise              = False
```

Functions on data types

- What about the following function?

```
weekend2 :: Day -> Bool
weekend2 d
    | (d == Sun || d == Sat) = True
    | otherwise              = False
```

- Error!

```
-- No instance for (Eq Day) arising from a use of '=='
```

Functions on data types

- What about this function?

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Sat = Sun
```

Functions on data types

- What about this function?

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Sat = Sun
```

- Invoke `nextday Fri` in `ghci`

Functions on data types

- What about this function?

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Sat = Sun
```

- Invoke `nextday Fri` in `ghci`
- Error again!

```
-- No instance for (Show Day)
   arising from a use of 'print'
```

Adding data type instances

- To check equality of two values of a data type `a` …

Adding data type instances

- To check equality of two values of a data type `a` …
- there must be an `Eq a` instance

Adding data type instances

- To check equality of two values of a data type `a` …
- there must be an `Eq a` instance
- We use `deriving` to create such instances:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving Eq
```

Adding data type instances

- To check equality of two values of a data type `a` …
- there must be an `Eq a` instance
- We use **`deriving`** to create such instances:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving Eq
```

- Default behaviour – `Sun == Sun`, `Tue /= Fri`, …

Adding data type instances

- To check equality of two values of a data type `a` ...
- there must be an `Eq a` instance
- We use **`deriving`** to create such instances:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving Eq
```

- Default behaviour – `Sun == Sun`, `Tue /= Fri`, ...
- Now `weekday2` compiles without error

Adding data type instances

- To make `nextday` work, we must make an instance for `Show Day`

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq, Show)
```

Adding data type instances

- To make `nextday` work, we must make an instance for `Show Day`

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq, Show)
```

- `show` provides a default text representation that can be printed on screen

Adding data type instances

- To make `nextday` work, we must make an instance for `Show Day`

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
     deriving (Eq, Show)
```

- `show` provides a default text representation that can be printed on screen
- `show Wed = "Wed"`

Adding data type instances

- Can also create an `Ord Day` instance:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq, Show, Ord)
```

Adding data type instances

- Can also create an `Ord Day` instance:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq, Show, Ord)
```

- Default behaviour: `Sun < Mon < Tue < Wed < Thu < Fri < Sat`

Adding data type instances

- Instances for Shape:

```
data Shape = Circle Double | Square Double
                         | Rectangle Double Double
     deriving (Eq, Ord, Show)
```

Adding data type instances

- Instances for Shape:

```
data Shape = Circle Double | Square Double
                           | Rectangle Double Double
    deriving (Eq, Ord, Show)
```

- Default behaviours:

```
show (Circle 5.0) == "Circle 5.0"
Square 4.0 == Square 4.0
Square 4.0 /= Square 3.0
Circle 5.0 /= Rectangle 3.0 4.0
[Square 2.0, Circle 3.0, Square 22.0]
    < [Square 2.0, Square 0.005]
```

Constructors

- `Square`, `Circle`, `Sun`, `Mon`, ...are **constructors**

Constructors

- `Square`, `Circle`, `Sun`, `Mon`, …are **constructors**
- They are just functions, but start with an uppercase letter

```
Sun :: Day
Rectangle :: Double -> Double -> Shape
Circle :: Double -> Shape
```

Constructors

- `Square`, `Circle`, `Sun`, `Mon`, ...are **constructors**
- They are just functions, but start with an uppercase letter

```
Sun :: Day
Rectangle :: Double -> Double -> Shape
Circle :: Double -> Shape
```

- They can be used just like any other function

```
Circle 5.0 :: Shape
map Circle :: [Double] -> [Shape]
map Circle [2.0, 3.0] = [Circle 2.0, Circle 3.0]
```

Records

- Data types with a single constructor

Records

- Data types with a single constructor
- Convention: The single constructor has the same name as the type

```
data Person = Person String Int Double String
    deriving (Eq, Show)
gal = Person "Ashvini" 21 5.9 "ashvini@me.com"
```

Records

- Data types with a single constructor
- **Convention**: The single constructor has the same name as the type

```
data Person = Person String Int Double String
    deriving (Eq, Show)
gal = Person "Ashvini" 21 5.9 "ashvini@me.com"
```

- The four parameters are supposed to stand for **name**, **age**, **height** and **email id**

Records

- Data types with a single constructor
- Convention: The single constructor has the same name as the type

```
data Person = Person String Int Double String
    deriving (Eq, Show)
gal = Person "Ashvini" 21 5.9 "ashvini@me.com"
```

- The four parameters are supposed to stand for name, age, height and email id
- How do we extract the height of `gal`?

Records

- Data types with a single constructor
- **Convention**: The single constructor has the same name as the type

```
data Person = Person String Int Double String
    deriving (Eq, Show)
gal = Person "Ashvini" 21 5.9 "ashvini@me.com"
```

- The four parameters are supposed to stand for **name**, **age**, **height** and **email id**
- How do we extract the height of `gal`?
- We need **destructors**

Destructors

```
name :: Person -> String
name (Person n _ _ _) = n

age :: Person -> Int
age (Person _ a _ _) = a

height :: Person -> Double
height (Person _ _ h _) = h

email :: Person -> String
email (Person _ _ _ e) = e
```

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double, email :: String
                     } deriving Show
```

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double, email :: String
                     } deriving Show
```

- We can name fields while creating values of type `Person`

```
gal = Person {name = "Ashvini", email = "ashvini@me.com"
             age = 21, height = 5.9}
```

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double, email :: String
                     } deriving Show
```

- We can name fields while creating values of type `Person`

```
gal = Person {name = "Ashvini", email = "ashvini@me.com"
              age = 21, height = 5.9}
```

- Order of fields not important

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double, email :: String
                     } deriving Show
```

- We can name fields while creating values of type `Person`

```
gal = Person {name = "Ashvini", email = "ashvini@me.com"
              age = 21, height = 5.9}
```

- Order of fields not important
- The following also works, but fields have to be in order!

```
gal = Person "Ashvini" 21 5.9 "ashvini@me.com"
```

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double , email :: String
                     } deriving Show
```

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double , email :: String
                     } deriving Show
```

- The field names are actually functions

Record syntax

- We can name the fields:

```
data Person = Person { name :: String, age :: Int
                     , height :: Double , email :: String
                     } deriving Show
```

- The field names are actually functions
- Automatically defined for us when we use record syntax

```
name :: Person -> String
age :: Person -> Int
height :: Person -> Double
email :: Person -> String
```

Stack

- Consider a `Stack` data type

Stack

- Consider a `Stack` data type
- A collection of **`Int`**s stacked one on top of the other

Stack

- Consider a `Stack` data type
- A collection of **`Int`**s stacked one on top of the other
- `push`: place an element on top of the stack

Stack

- Consider a `Stack` data type
- A collection of **`Int`**s stacked one on top of the other
- `push`: place an element on top of the stack
- `pop`: remove the topmost element of the stack

Stack

- Consider a `Stack` data type
- A collection of `Int`s stacked one on top of the other
- `push`: place an element on top of the stack
- `pop`: remove the topmost element of the stack
- Behaviour similar to lists: top of stack is head of list

Stack

- We could declare `Stack` to be a type synonym

```
type Stack = [Int]
push :: Int -> Stack -> Stack
push n st = n:st
pop :: Stack -> (Int, Stack)
pop (n:st) = (n, st)
```

Stack

- We could declare Stack to be a type synonym

```
type Stack = [Int]
push :: Int -> Stack -> Stack
push n st = n:st
pop :: Stack -> (Int, Stack)
pop (n:st) = (n, st)
```

- But this allows operations other than push and pop

```
take n st
st1 ++ st2
take (n-1) st ++ [x] ++ drop (n-1) st
```

Stack

- We want to allow only functions defined for stack

Stack

- We want to allow only functions defined for stack
- First step: make it a data type

```
data Stack = Stack [Int]
push :: Int -> Stack -> Stack
push x (Stack xs) = Stack (x:xs)
pop :: Stack -> (Int, Stack)
pop (Stack (x:xs)) = (x, Stack xs)
```

Stack

- We want to allow only functions defined for stack
- First step: make it a data type

```
data Stack = Stack [Int]
push :: Int -> Stack -> Stack
push x (Stack xs) = Stack (x:xs)
pop :: Stack -> (Int, Stack)
pop (Stack (x:xs)) = (x, Stack xs)
```

- If `st`, `st1`, `st2` are of type `Stack`, the following will not typecheck!

```
take n st
st1 ++ st2
take (n-1) st ++ [x] ++ drop (n-1) st
```

Type parameters

- Clearly, the operations of a stack do not depend on the type of elements stored

Type parameters

- Clearly, the operations of a stack do not depend on the type of elements stored
- Polymorphic stack

```
data Stack a = Stack [a]
push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (x:xs)
pop :: Stack a -> (a, Stack a)
pop (Stack (x:xs)) = (x, Stack xs)
empty :: Stack a
empty = Stack []
isEmpty :: Stack a -> Bool
isEmpty (Stack xs) = null xs
```

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type
- It is a **type constructor**

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type
- It is a **type constructor**
- For any type `a`, `Stack a` is a type

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type
- It is a **type constructor**
- For any type `a`, `Stack a` is a type
- The `Stack` on the right is a **value constructor** or a **data constructor**

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type
- It is a **type constructor**
- For any type `a`, `Stack a` is a type
- The `Stack` on the right is a **value constructor** or a **data constructor**
- Given `xs :: [a]`, it constructs a value of type `Stack a`

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- `Stack` (occurring on the left) is not a type
- It is a **type constructor**
- For any type `a`, `Stack a` is a type
- The `Stack` on the right is a **value constructor** or a **data constructor**
- Given `xs :: [a]`, it constructs a value of type `Stack a`
  - `Stack xs :: Stack a`

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- Suppose we want to define `sumStack`:

```
sumStack (Stack xs) = sum xs
```

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- Suppose we want to define `sumStack`:

```
sumStack (Stack xs) = sum xs
```

- What is the type of `sumStack`?

Type parameters

- Polymorphic stack

```
data Stack a = Stack [a]
```

- Suppose we want to define `sumStack`:

```
sumStack (Stack xs) = sum xs
```

- What is the type of `sumStack`?
- Makes sense only when `xs` consists of numeric elements

```
sumStack :: Num a => Stack a -> a
```

Maybe

- Recall `Maybe`

Maybe

- Recall `Maybe`
- It is a type constructor!

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

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

- Suppose you want to find the score corresponding to a name

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

- Suppose you want to find the score corresponding to a name
- If name is not in the list

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

- Suppose you want to find the score corresponding to a name
- If name is not in the list
  - Return a default value

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

- Suppose you want to find the score corresponding to a name
- If name is not in the list
  - Return a default value
  - Not always easy to find a default value that is not also a possible score

Maybe

- Consider a table representing a list of scores

```
type Name = String
type Score = Int
type Scorelist = [(Name, Score)]
```

- Suppose you want to find the score corresponding to a name
- If name is not in the list
  - Return a default value
  - Not always easy to find a default value that is not also a possible score
- Use `Maybe` instead

Maybe

- Built-in function `lookup`

```
lookup :: Name -> Scorelist -> Maybe Score
lookup n []           = Nothing
lookup n ((n0,s0):sl)
    | n == n0         = Just s0
    | otherwise       = lookup n sl
```

Maybe

- Handle **Maybe** objects using **case**

```
f :: Name -> Scorelist -> String
f n sl =
    case lookup n sl of
        Nothing -> "Looks like you were absent!"
        Just x  -> "Your score is " ++ show x
```

Either

- **Either** is the simplest union type constructor

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

Either

- **Either** is the simplest union type constructor

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

- Handle **Either** objects also using **case**

```
f :: (Show a, Show b) => Either a b -> String
f = case value of
    Left x  -> "You have a left "  ++ show x
    Right y -> "You have a right " ++ show y
```

Summary

- The keyword `data` is used to declare new data types

Summary

- The keyword `data` is used to declare new data types
- The keyword `deriving` to derive as an instance of a type class

Summary

- The keyword `data` is used to declare new data types
- The keyword `deriving` to derive as an instance of a type class
- Data types with parameters – `Shape`, `Person`

Summary

- The keyword `data` is used to declare new data types
- The keyword `deriving` to derive as an instance of a type class
- Data types with parameters – `Shape`, `Person`
- Sum type or `union` – `Day`, `Shape`

Summary

- The keyword **`data`** is used to declare new data types
- The keyword **`deriving`** to derive as an instance of a type class
- Data types with parameters – `Shape`, `Person`
- Sum type or `union` – `Day`, `Shape`
- Product type or `struct` – `Person`

Summary

- The keyword **`data`** is used to declare new data types
- The keyword **`deriving`** to derive as an instance of a type class
- Data types with parameters – `Shape`, `Person`
- Sum type or `union` – `Day`, `Shape`
- Product type or `struct` – `Person`
- Type constructors – **`Maybe`**, **`Either`**, `Stack`