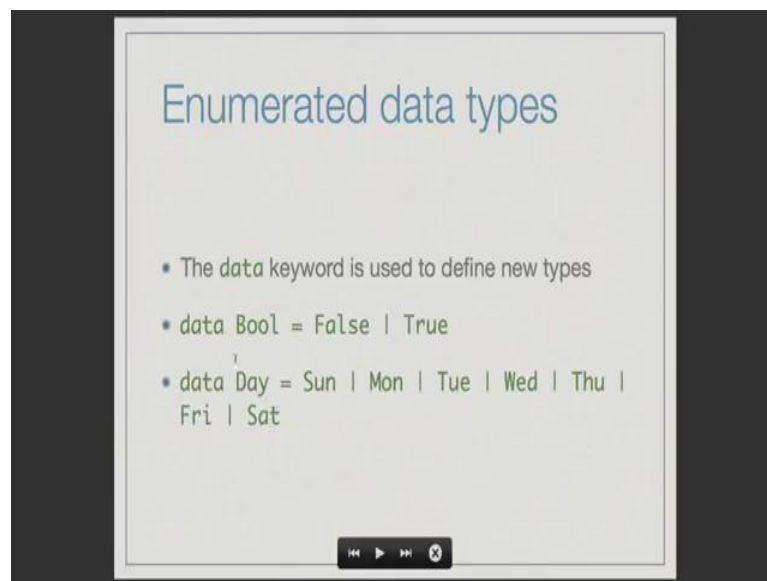


Functional Programming in Haskell
Prof. Madhavan Mukund & S.P Suresh
Chennai Mathematical Institute

Module # 05
Lecture - 01
User Defined Data Types

Welcome to week 5 of the NPTEL course on Functional Programming in Haskell.

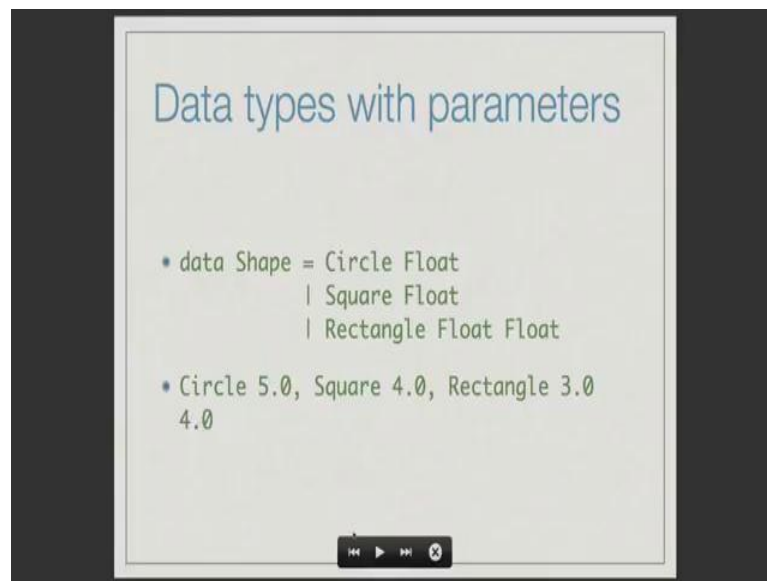
(Refer Slide Time: 00:19)



I am S.P. Suresh and I shall be taking over from Professor Madhavan Mukund for the next few weeks. In today's lecture, we shall be looking at user defined data types in Haskell. The simplest way to define new data types in Haskell is the so called enumerated data types, here is an example `data Bool = False | True`. You define the new data type using the `data` keyword and give the name of the type here `Bool`.

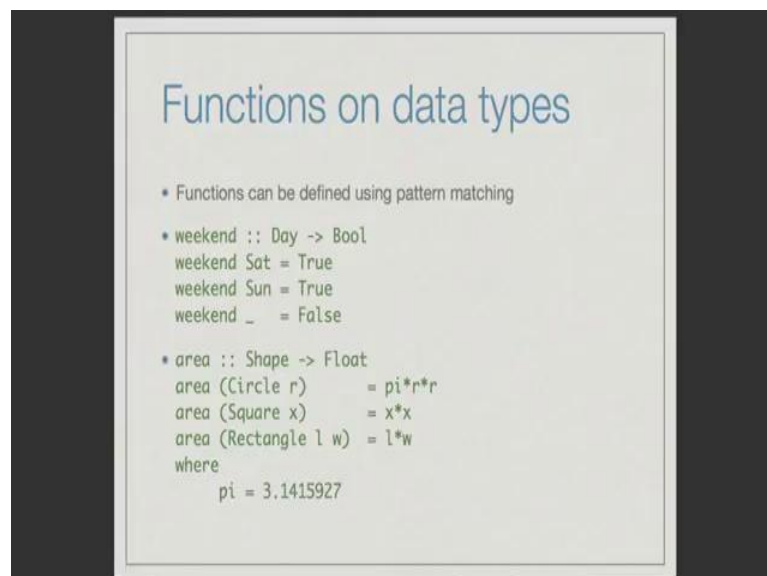
Notice the capitalization and then, you enumerate all the values that in the part of the type, in this case it is either `False` or `True`, notice the capitalization again. Another example is `Day`, you declare it by saying `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`. These are all the possible values of the type `Day`.

(Refer Slide Time: 01:15)



Here is the another example, data shape equals either a circle or a square or a rectangle, but a circle has a parameter namely the radius and a square comes with the parameter, namely the length of the side. Similarly, a rectangle also has a parameter namely the length and breadth. This enables us to declare new data types, which can take infinitely many values, you have circle 1, circle 2.0 or circle 3.0 and so on, one circle objects for every float value. Here are some examples, circle 5.0, square 4.0, rectangle 3.0, 4.0; these are rectangle with breadth 3 and length 4.

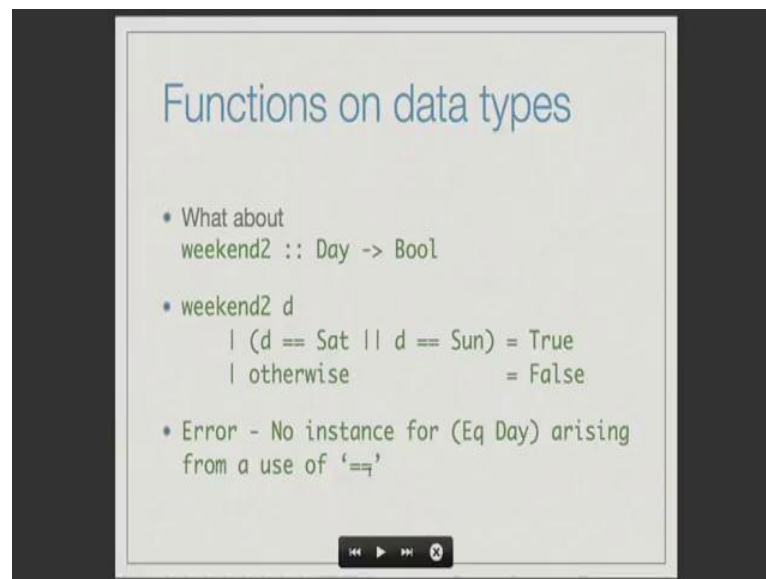
(Refer Slide Time: 02:15)



You can define functions on user defined data types in the usual manner, here are some example functions defined using pattern matching. For instance, you can define what a

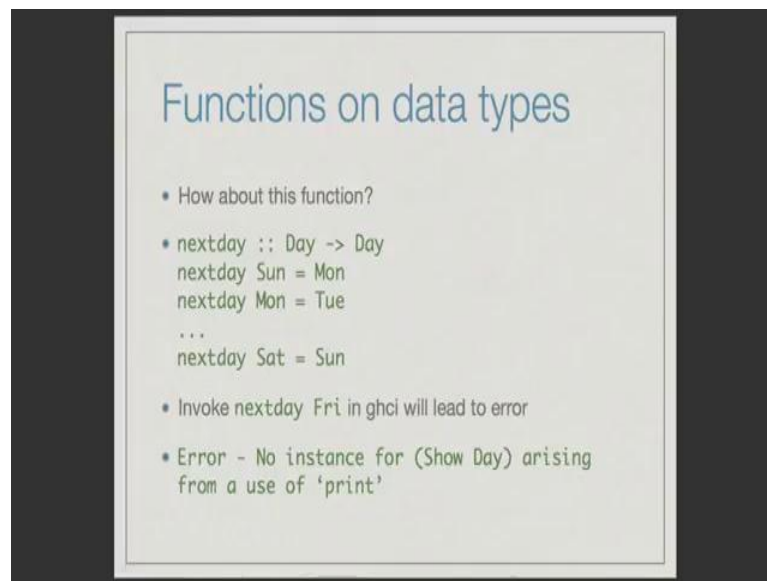
weekend is. Weekend is a function from day to bool and here is the definition, weekend of Saturday is true, weekend of Sunday is true and weekend of anything else is false. Similarly, you can define an area function, which is a function from shape to float, area of a circle with radius r is πr^2 , area of a square with length of square x is x^2 and area of rectangle with length l and width w is $l \times w$. Or there is other ways to define functions on user defined data types, yes.

(Refer Slide Time: 03:11)



For instance, here is another way to define the weekend function called `weekend2`, the definition is here. `Weekend2` of `d` is true if `d` is equal to Saturday or `d` is equal to Sunday and it is equal to false otherwise. But, if you enter this program as it is in `ghci`, you will get an error, the error message will be something like this. No instance for `Eq Day` arising from a use of the equality operator; let us see how to fix this later.

(Refer Slide Time: 03:49)

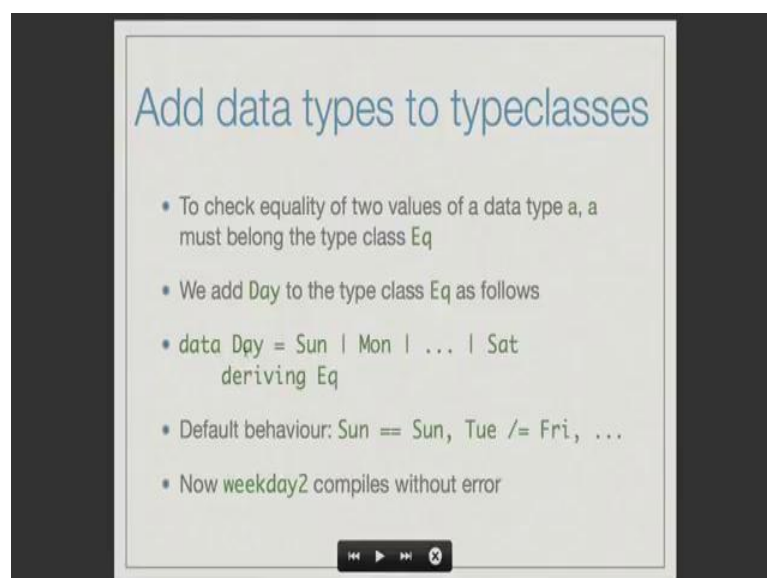


Functions on data types

- How about this function?
- ```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Sat = Sun
```
- Invoke `nextday Fri` in `ghci` will lead to error
- Error - No instance for (Show Day) arising from a use of 'print'

Here is another function you can write, which is a function from day to day, it gives the nextday. For instance, nextday of Sunday equals Monday, nextday of Monday equals Tuesday and so on, nextday of Saturday equals Sunday. Now, if you load this function in `ghci` and invoke nextday of Friday, it will again lead to an error and it is the following error. No instance for show day arising from a use of print, we will see what this means and how to fix this in a later slide.

(Refer Slide Time: 04:26)



### Add data types to typeclasses

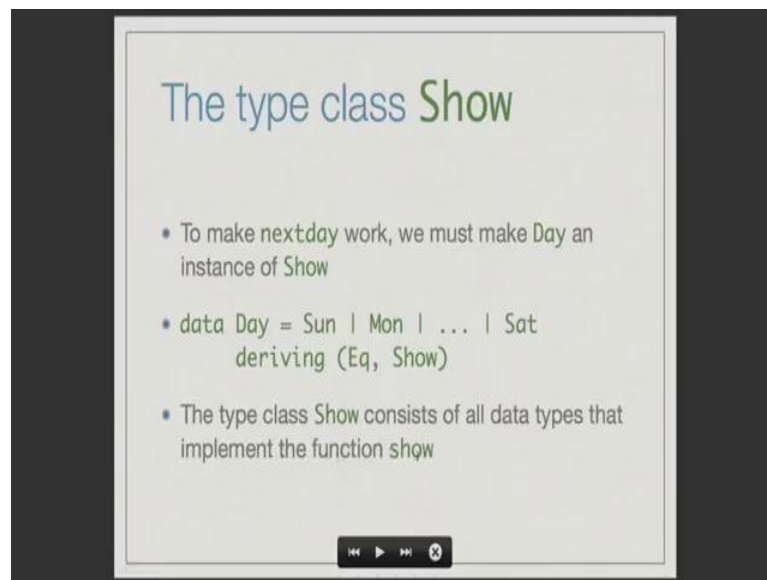
- To check equality of two values of a data type `a`, `a` must belong to the type class `Eq`
- We add `Day` to the type class `Eq` as follows
- ```
data Day = Sun | Mon | ... | Sat
  deriving Eq
```
- Default behaviour: `Sun == Sun`, `Tue /= Fri`, ...
- Now `weekday2` compiles without error

To check equality of two values of a data type `a`, `a` must be declared to belong to the type class `Eq`. We have seen the notion of a type class in earlier lectures and we have also seen the type class `Eq` in detail. So, to get `weekday2` to work we need to add `Day` to the type

class Eq and we add Day to the type class Eq as follows, data Day equals Sunday or Monday or Tuesday or Wednesday or Friday or Thursday or Saturday deriving Eq, the keyword is derived.

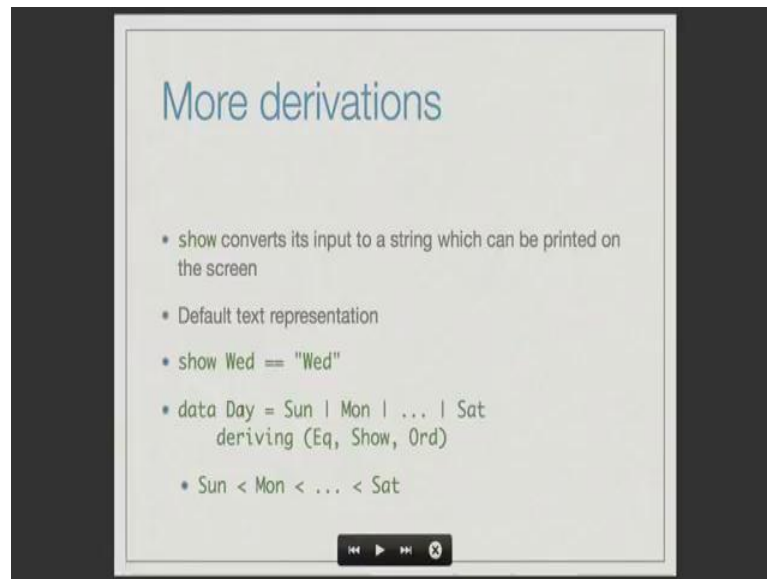
When we declare Day to derive from equality, the equality operator has the default behavior. Sunday equals Sunday, Tuesday is not equal to Friday, Monday equals Monday, etcetera. Once you declare Day to be deriving from equality, then weekday2 compiles without error.

(Refer Slide Time: 05:32)



To make the next day work, we must make day an instance of the type class called Show with a capital S. We have to declare it as follows, data Day equals Sunday, Monday or Tuesday, etcetera deriving Eq comma Show. The type class Show consists of all data types that implement the function show with a small s.

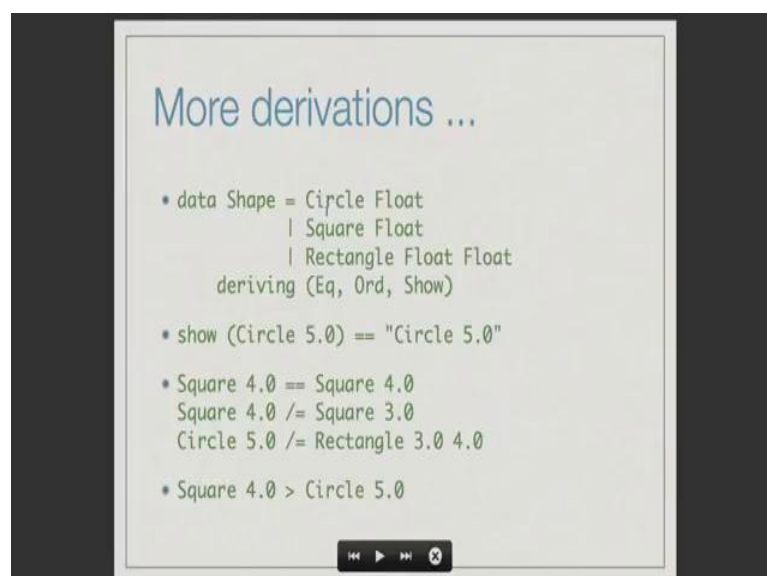
(Refer Slide Time: 05:59)



The function `show` converts its input to a string which can be printed on the screen, we need not define it explicitly for the data type `Day`. If we do not give an explicit definition, there is a default definition that Haskell provides, which is just to give a default text representation for the data value. For instance, `show Wed` is just the string `Wed`, we can also derive `Day` as an instance of the type class `Ord`, `Ord` which is an ordinal type.

When we do this, an order is defined on the data type `Day` as follows, Sunday is less than Monday is less than Tuesday etcetera is less than Saturday and this order is determined by the order in which the data values are enumerated in the data type declaration.

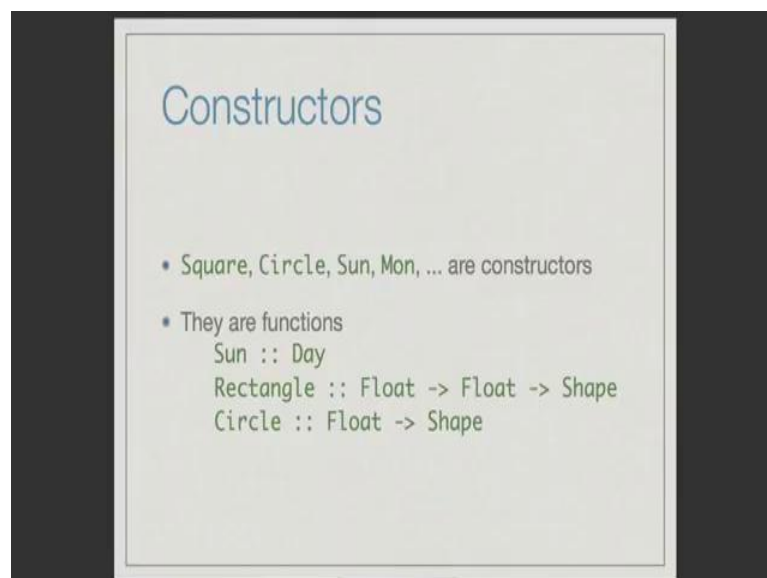
(Refer Slide Time: 07:01)



We can also derive shape to belong to various classes, we consider data Shape equals circle float or square float or rectangle float, float deriving Eq, Ord and Show. Now, you can use all these functions on data on values of type Shape. For instance, Show circle 5.0 will just be the string that say circle 5.0, square 4.0 is equal to square 4.0, the equality check is derived from the equality on floating point numbers as well as the names, square or circle or rectangle.

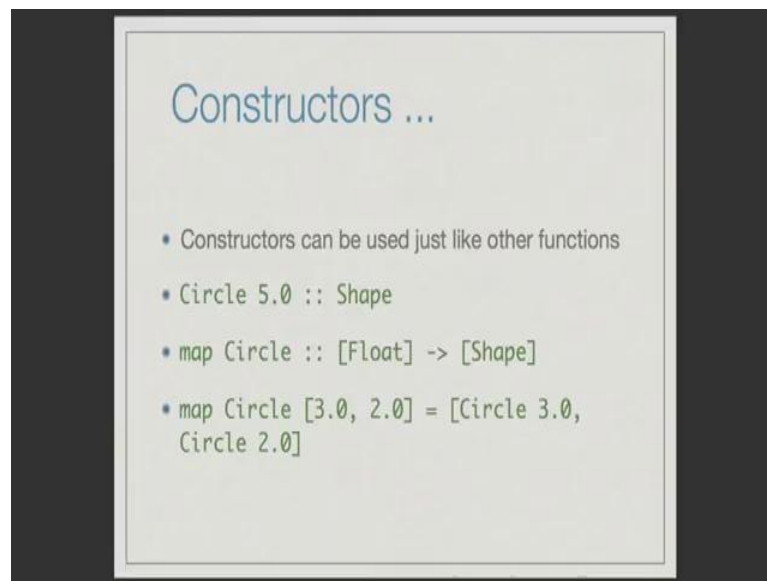
For instance, square 4.0 is equal to square 4.0, because both the parameters and the name here are equal, square 4.0 is not equal to square 3.0. Because, even though the names are equal the parameters are different, circle 5.0 is not equal to rectangle 3.0, 4.0, because there are two different types of shape, one is a circle and other is a rectangle and we have also derived it to belong to the type class Ord. So, there is an order defined on shapes, square 4.0 is for instance greater than circle 5.0, because square comes later in the declaration than circle.

(Refer Slide Time: 08:37)



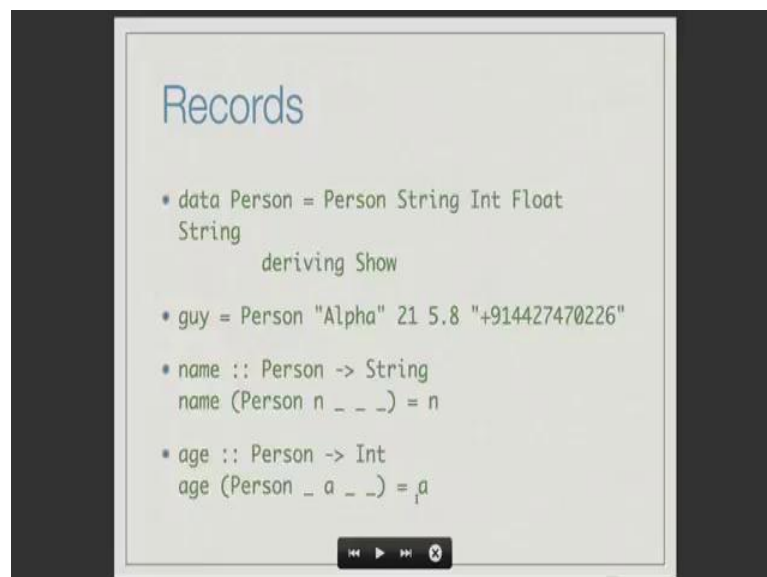
The names square, circle, Sunday, Monday, etcetera that we have used are called constructors. They are nothing but, functions; Sunday for instance is a function that is of type Day. It is a function that accepts no input, but produces an output. Rectangle is a function with two parameters, so it takes two floats as an inputs and it produces a shape as output. So, rectangle is a function whose type is float arrow float arrow shape, similarly circle is a function whose type is float arrow shape.

(Refer Slide Time: 09:21)



These constructors can be used just like any other function, for instance circle can be invoked on the input 5.0 to give a shape. You can map the circle function over a list of floats to get a list of shapes. For instance, map circle on the list 3.0 and 2.0 will give you the list consisting of circle 3.0 and circle 2.0.

(Refer Slide Time: 09:49)

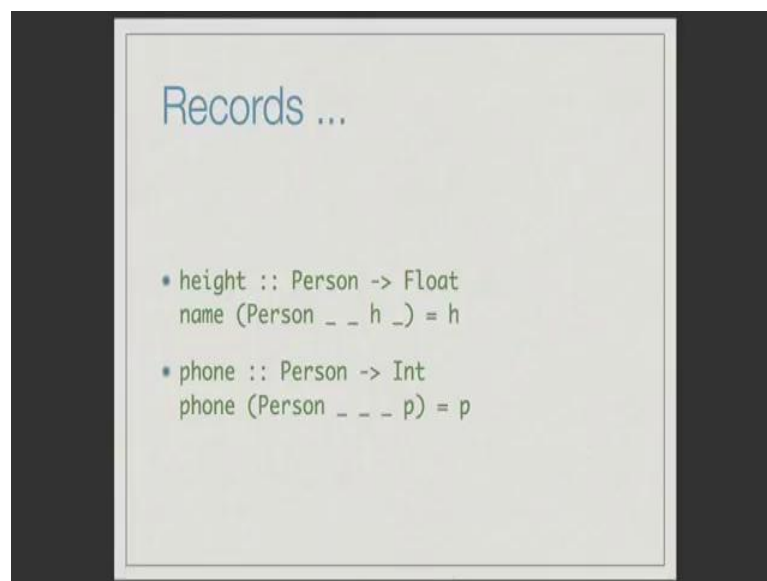


Here is another way of defining types, data Person equals Person String Int Float String. Now, you will see two occurrences of the word person here, one on the left one on the right. On the left, it denotes the name of the type; on the right it is the name of the constructor. Now, unlike in a type like shape where we had three constructors circle, square, rectangle, here we have only one constructor, though we have many parameters.

The convention in Haskell is that, if a data type has only one constructor then you use the same name for both the type and the constructor. So, the person that appears on the right is a constructor and the person that appears on the left is the name of the data type. So, here we say that data Person equals person string int float string, the intention is that the first string is the name of the person, this int here is the age of the person, the float here is let us say the height of the person and the last string here is the phone number.

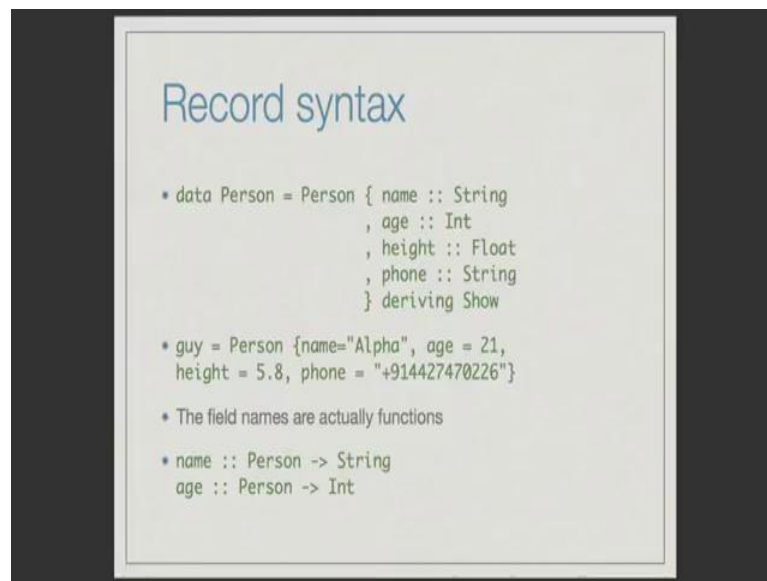
So, for instance I might say guy equals person alpha 21, 5.8 and some phone number. How do you extract the name of a person object? You write a function name, whose input is person and the output is string and the definition is this. Name, person n, any age, any height and any phone number equals n, here is a function that extracts the age of a person, age is a function from person to int and the definition is age, person, anything a, anything, anything equals a.

(Refer Slide Time: 11:45)



You can write a height function which says height person anything, anything h, anything is height, is h and here is a function that extracts phone number of a person; phone, person, anything, anything, anything p equals p. There is a pattern matching and there is a do not care patterns in all these definitions, but this kind of definition is quite common. So, Haskell offers an alternative easier syntax which is also familiar from other languages.

(Refer Slide Time: 12:21)



Record syntax

- `data Person = Person { name :: String, age :: Int, height :: Float, phone :: String } deriving Show`
- `guy = Person {name="Alpha", age = 21, height = 5.8, phone = "+914427470226"}`
- The field names are actually functions
- `name :: Person -> String`
`age :: Person -> Int`

You can define the person type as for this, `data Person` equals `person` and within braces you say `name` which is of type `string`, `age` which is of type `int`, `height` which is of type `float` and `phone` which is of type `string`. So, you are naming all the fields directly in the data definition itself and these names here, `name`, `age`, `height`, `phone` etcetera are really nothing but, the functions that we defined earlier. `Name` is a function from `person` to `string`, `age` is a function from `person` to `int`.

You declare new objects of type `person` in this syntax as follows, `guy` equals `person`, within braces you say `name` equals `alpha`, `age` equals `21`, `height` equals `5.8` and `phone` equals `phone number`. This is an easier syntax that Haskell offers.

(Refer Slide Time: 13:24)



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`

To summarize, if we seen simple ways of defining new data types, the keyword `data` is used to declare new data types. The key word `deriving` is used to derive the data type as an instance of a type class and typically, the standard functions that are supposed to be defined on types of the type class are defined to be default. For instance equality, order, show etcetera, then you can also define data types with parameters as in the example of `shape`, `person`, etcetera.

You have two different types of user defined data types, one you might called the sum type or union type, where there are multiple constructors on the right. The example here is `Day` or `Shape` or the other is the product type or the struct type that you might be familiar from other languages, where you have only one constructor on the right, the example of this is a `Person`.