Haskell Data Types, Pattern Matching, Type Classes

Haskell provides a way to represent and work with shapes using data types. In this lecture, we will explore how to use Haskell data types and compare functional programming with object-oriented programming.

A Haskell Solution Using Data Types First, let's define a data type `Shape` that can represent circles, squares, and right triangles:

```
data Shape
  = Circle         Double Double Double
  | Square         Double Double Double
  | RightTriangle  Double Double Double Double
  deriving Show
```

Next, we'll define an `area` function that computes the area of a shape:

```
area :: Shape -> Double
area shape =
  case shape of
    Circle x y r          -> pi * r * r
    Square x y s          -> s * s
    RightTriangle x y l w -> l * w
```

Finally, we'll define a `shift` function that shifts a shape by a given amount in the x and y directions:

```
shift :: Shape -> Double -> Double -> Shape
shift shape delta_x delta_y =
  case shape of
    Circle x y r -> Circle (x + delta_x) (y + delta_y) r
    Square x y s -> Square (x + delta_x) (y + delta_y) s
    RightTriangle x y l w ->
    RightTriangle (x + delta_x) (y + delta_y) l w
```

Functional Programming vs Object Oriented Programming When comparing functional programming (FP) and object-oriented programming (OOP), it's important to understand that they have different points of view.

In OOP, objects know how to perform different operations on themselves. For example, a circle knows how to compute its own area and perform a shift.

In FP, functions know how to compute over different data types. For instance, the `shift` function knows how to compute a shifted square, circle, or right triangle.

These two points of view are orthogonal:

- Object-Oriented Programming: An object knows how to perform different operations on itself.
- Functional Programming: A function knows how to compute over different data types.

The key takeaway is that Haskell data types are not classes. Instead, they provide a way to represent and work with data in a functional programming style.

Simple Data Types in Haskell

Haskell provides a variety of simple data types, some of which include:

Value	Type	Description
1, 2, 100000000000, -42, ...	Integer	Integer numbers
3.14, 3.2831, -2.718, ...	Double	Floating-point numbers
True, False	Bool	Boolean values
'a', 'z', ...	Char	Character values
"hello", "world"	String	String values
[1, 2, 3, 4]	[Integer]	List of integers
(True, 0.5)	(Bool, Double)	Tuple of Bool and Double

All type names in Haskell are upper-case, but not all upper-case names are types.

Data Type Simple Examples

```haskell
data CoinFlip = Heads | Tails
```

`Heads` and `Tails` are values that have type `CoinFlip`.

```haskell
data CardSuit = Clubs | Diamonds | Hearts | Spades
```

`Clubs`, `Diamonds`, `Hearts`, and `Spades` are values that have type `CardSuit`.

```haskell
data ThermostatSetting = Off | Cooling | Heating
```

`Off`, `Cooling`, and `Heating` are values that have type `ThermostatSetting`.

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

`True` and `False` are values that have type `Bool`.

Functions for Data Types: Simple Example

We have a simple `ThermostatSetting` data type defined as follows:

```haskell
data ThermostatSetting
= Off
    | Cooling
```

```haskell
  | Heating
  deriving Show
```

Now, let's create a function `isRunning` that takes a `ThermostatSetting` and returns a `Bool`:

```haskell
isRunning :: ThermostatSetting -> Bool
```

Defining Functions for Data Types

Style 1: Pattern Matching the Function Arguments

```haskell
isRunning :: ThermostatSetting -> Bool

isRunning Off      = False
isRunning Cooling  = True
isRunning Heating  = True
```

Style 2: Pattern Matching Inside a Case Expression in the Function Body

```haskell
isRunning :: ThermostatSetting -> Bool

isRunning setting =
  case setting of
    Off     -> False
    Cooling -> True
    Heating -> True
```

Style 3: Pattern Matching the Function Arguments with a Wildcard (_)

```haskell
isRunning :: ThermostatSetting -> Bool

isRunning Off = False
isRunning _   = True
```

Style 4: Pattern Matching in a Case Expression in the Function Body with a Wildcard

```haskell
isRunning :: ThermostatSetting -> Bool

isRunning setting =
  case setting of
    Off -> False
    _   -> True
```

Style 5: If-Then-Else Expression in the Function Body

```
isRunning :: ThermostatSetting -> Bool

isRunning setting =
  if setting == Off
  then False
  else True
```

In order to use the `==` operator, we need to derive `Eq` for the `ThermostatSetting` data type:

```
data ThermostatSetting
= Off
    | Cooling
    | Heating
    deriving (Show, Eq)

isRunning :: ThermostatSetting -> Bool
isRunning setting =
  if setting == Off
  then False
  else True
```

Style 6: A "One-Liner" That Does All the Work in a Single Expression

```
isRunning :: ThermostatSetting -> Bool

isRunning setting = (setting /= Off)
```

Style 7: A "One-Liner" Using Currying with (/=)

```
isRunning :: ThermostatSetting -> Bool

isRunning = ((/=) Off)
```

Discussion

Which style do you personally prefer? Are there situations in which one way is definitely better or worse than another? Consider readability, writability, and maintainability when evaluating each style.

In general, you might not be able to tell the difference between different implementations just by calling the `isRunning` function, but the choice of implementation style can impact how easy it is to understand and maintain the code. Choose a style that best suits your preferences and the specific situation.

Data Types and Associated Values

Simple Data Type Examples

Here are some simple data type examples:

```
data CoinFlip = Heads | Tails deriving (Show, Eq)

data Suit = Clubs | Diamonds | Hearts | Spades
       deriving (Show, Eq)

data ThermostatSetting
= Off
    | Cooling
    | Heating
    deriving (Show, Eq)
```

Data Types with Associated Values

Let's consider a more complex `ThermostatSetting` data type:

```
data ThermostatSetting
= Off
    | CoolTo Int
    | HeatTo Int
    | OutOfService String
    deriving (Show, Eq)
```

Some example values of the `ThermostatSetting` data type are:

```
Off                            is a value of type     ThermostatSetting
CoolTo 27                      is a value of type     ThermostatSetting
HeatTo 35                      is a value of type     ThermostatSetting
OutOfService "Maintenance"     is a value of type     ThermostatSetting
```

You can create instances of these values:

```
setting_1 = Off
setting_2 = CoolTo 20
setting_3 = OutOfService "Under repair"
```

These constructors have the following types:

```
Off             is a value of type              ThermostatSetting
CoolTo          is a (function) value of type  Int -> ThermostatSetting
HeatTo          is a (function) value of type  Int -> ThermostatSetting
OutOfService    is a (function) value of type  String -> ThermostatSetting
```

General Haskell Data Types

A Haskell datatype declaration has the following form:

```
data <TypeName>
    = <ConstructorName> <Type> <Type> ... <Type>
    | <ConstructorName> <Type> <Type> ... <Type>
    ...
    | <ConstructorName> <Type> <Type> ... <Type>
```

You can also add type class constraints using the `deriving` keyword:

```
data <TypeName>
    = <ConstructorName> <Type> <Type> ... <Type>
    | <ConstructorName> <Type> <Type> ... <Type>
    ...
    | <ConstructorName> <Type> <Type> ... <Type>
    deriving (<TypeClass1>, <TypeClass2>, ...)
```

`Show` and `Eq` are examples of type classes.

Data Types, Associated Values, and Functions

Functions with Data Types and Associated Values

Here is an example of a function working with the `ThermostatSetting` data type:

```
isRunning :: Int -> ThermostatSetting -> Bool

isRunning temp Off                = False
isRunning temp (OutOfService msg) = False
isRunning temp (CoolTo t)         = temp > t
isRunning temp (HeatTo t)         = temp < t
```

Alternatively, you can use a wildcard `_` to ignore the unneeded values:

```
isRunning :: Int -> ThermostatSetting -> Bool

isRunning _ Off              = False
isRunning _ (OutOfService _) = False
isRunning temp (CoolTo t)    = temp > t
isRunning temp (HeatTo t)    = temp < t
```

Recursive Data Types

Creating Custom Lists with Recursive Data Types

A list can be an empty list or an element and another list:

```haskell
data IntList
    = Empty
    | Cons Int IntList
    deriving (Show)
```

Examples of IntList

```haskell
list1 = Empty
list2 = Cons 6 Empty
list3 = Cons 10 (Cons 20 list2)
list4 = Cons (-4) list3
list5 = Cons 100 (Cons 13 list4)
list6 = Cons 100 (Cons 13 list5)
```

Functions on IntList

Length

```haskell
intListLength :: IntList -> Int

intListLength Empty = 0
intListLength (Cons x xs) = 1 + intListLength xs
```

Head

```haskell
intListHead :: IntList -> Int

intListHead Empty = undefined
intListHead (Cons x xs) = x
```

Tail

```haskell
intListTail :: IntList -> IntList

intListTail Empty = undefined
intListTail (Cons x xs) = xs
```

Map

```haskell
intListMap :: (Int -> Int) -> IntList -> IntList

intListMap f Empty = Empty
intListMap f (Cons x xs) = Cons (f x) (intListMap f xs)
```

Sum

```haskell
intListSum :: IntList -> Int
```

```haskell
intListSum Empty = 0
intListSum (Cons x xs) = x + intListSum xs
```

String Binary Trees with Recursive Data Types

A `StringBinaryTree` is either a leaf with a `String` in it or a node with a `String` and two `StringBinaryTrees`:

```haskell
data StringBinaryTree
  = Leaf String
  | Node String StringBinaryTree StringBinaryTree
  deriving (Show)
```

Examples of StringBinaryTree

```haskell
tree1 = Leaf "a"
tree2 = Leaf "b"
tree3 = Leaf "c"
tree4 = Node "f" (Leaf "d") (Leaf "e")
tree5 = Node "g" tree1 tree2
tree6 = Node "h" tree5 tree4
tree7 = Node "i" tree3 tree6
```

Functions on StringBinaryTree

Size

```haskell
treeSize :: StringBinaryTree -> Int

treeSize (Leaf _) = 1
treeSize (Node _ left right) = 1 + (treeSize left) + (treeSize right)
```

Height

```haskell
treeHeight :: StringBinaryTree -> Int

treeHeight (Leaf _) = 0
treeHeight (Node _ left right) = 1 + max (treeHeight left) (treeHeight right)
```

Map

```haskell
treeMap :: (String -> String) -> StringBinaryTree -> StringBinaryTree

treeMap f (Leaf s) = Leaf (f s)
treeMap f (Node s left right) = Node (f s) (treeMap f left) (treeMap f right)
```

Example: Excited Tree

```
excitedTree = treeMap (++ "!") tree7
```

Type Classes

Values, Types, and Type Classes

We've already learned a lot about values and types. Values are instances of Types, and this relationship is similar to set membership, where values are members of types.

Example values:

- 'C'
- True
- [True, False]
- "Hello"

The Types of these values, respectively, are:

- Char
- Bool
- [Bool]
- [Char]

Haskell's Families of Types — Type Classes

Types belong to type classes:

- `Eq` defines the `==` and `/=` operators
- `Ord` defines comparison operators, such as `<` (Note that any type in `Ord` must also be in `Eq`)
- `Read` defines the `read` operator which turns a `String` into a value
- `Show` defines the `show` operator which turns a value into a `String`
- `Num` defines `+`, `-`, `/`, etc.

There are many more type classes, and we'll introduce them as needed.

Simple Data Type Examples

```
data CardSuit = Clubs | Diamonds | Hearts | Spades
```

Here, `Clubs`, `Diamonds`, `Hearts`, and `Spades` are values that have the `CardSuit` type.

Data Type Examples with Type Classes

```
data CardSuit = Clubs | Diamonds | Hearts | Spades
    deriving (Show, Eq, Ord, Read)
```

`Clubs`, `Diamonds`, `Hearts`, and `Spades` are values that have the `CardSuit` type.

Type Classes are NOT OOP Classes

Type classes in Haskell are not the same as the concept of classes from object-oriented programming. They both use the word "class," but they mean totally different things.

Bonus: Combining Two (or more) Data Types

Combining Data Types

```haskell
data CardSuit = Clubs | Diamonds | Hearts | Spades
  deriving (Show, Eq, Ord)

data FaceValues
  = Two   | Three   | Four   | Five   | Six   | Seven   | Eight
  | Nine   | Ten   | Jack | Queen   | King | Ace
  deriving (Show, Eq, Ord)

type Card = (FaceValues, CardSuit)

data CardList
  = Empty
  | Hand Card CardList
  deriving (Show, Eq, Ord)
```

Example of a `CardList`:

```haskell
cardList1 = Hand (Two, Hearts) (Hand (Ace, Diamonds) (Hand (Ten, Spades) Empty))
```