Data in Haskell

Principles of Programming Languages

Modules in Haskell

- The Haskell standard library is split into modules, each of them contains functions and types that are somehow related and serve some common purpose.
- The syntax for importing modules in a Haskell script is import <module name>.
- The import must be done before defining any functions, so imports are usually done at the top of the file.

Data.list

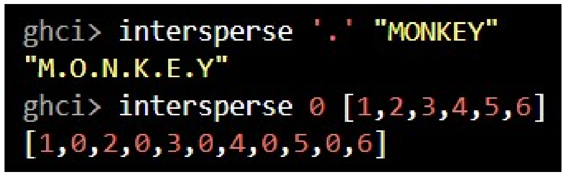
Data.List

- Data.List module, which has a bunch of useful functions for working with lists.
- You don't have to import Data.List via a qualified import because it doesn't clash with any Prelude names

Data.List functions

intersperse

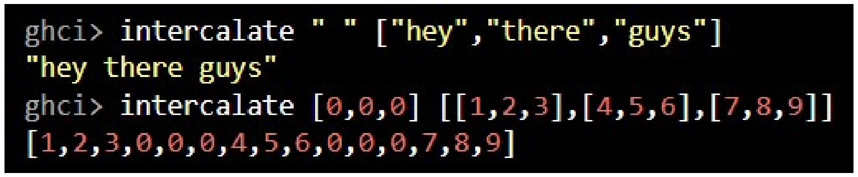
- takes an element and a list and then puts that element in between each pair of elements in the list.



```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

intercalate

- takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

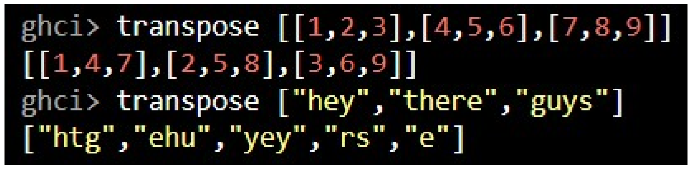


```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

Data.List functions

transpose

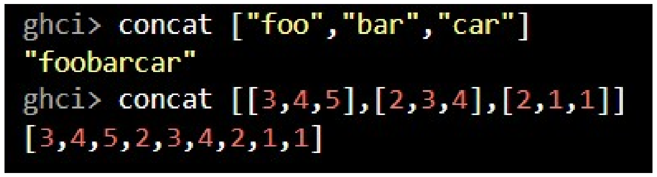
- transposes a list of lists.



```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

concat

- flattens a list of lists into just a list of elements.

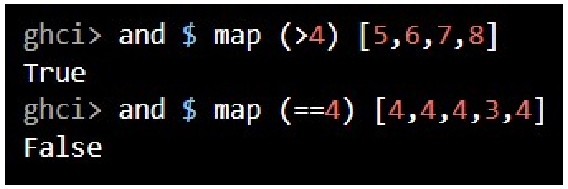


```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

Data.List functions

and

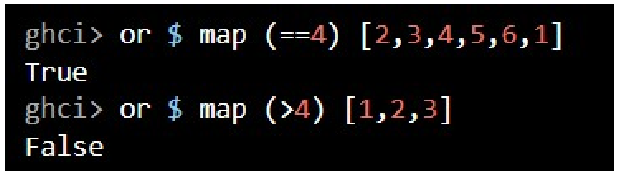
- takes a list of boolean values and returns True only if all the values in the list are True.



```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

or

- is like and, only it returns True if any of the boolean values in a list is True.

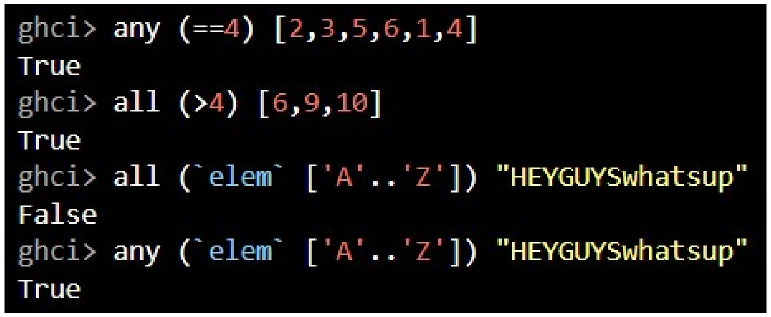


```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

Data.List functions

any and all

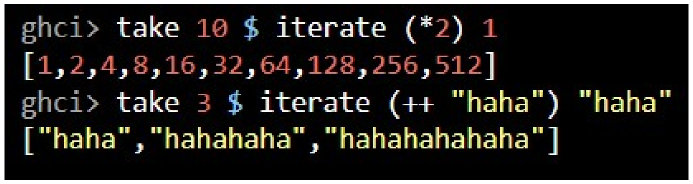
- take a predicate and then check if any or all the elements in a list satisfy the predicate, respectively.



```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

iterate

- takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.

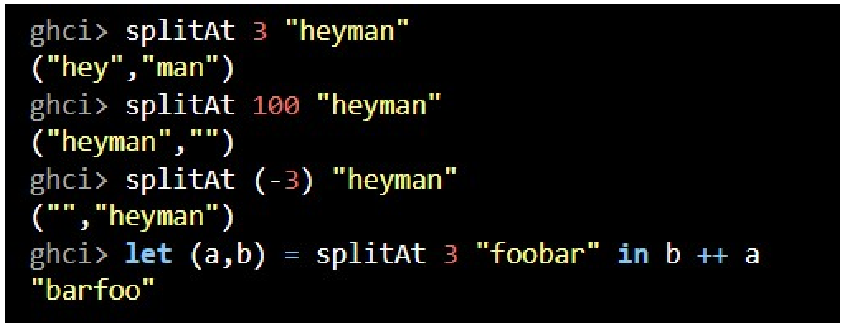


```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

Data.List functions

splitAt

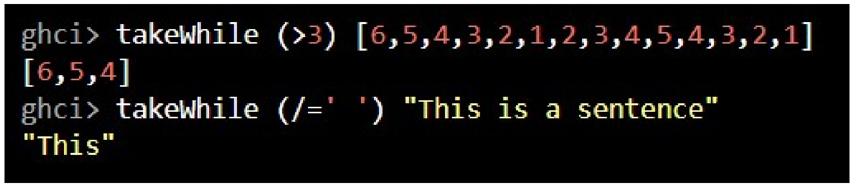
- takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.



```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

takeWhile

- It takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it's cut off.

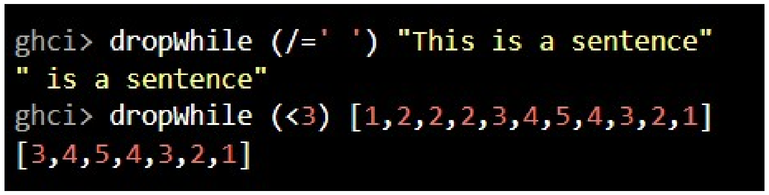


```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Data.List functions

dropWhile

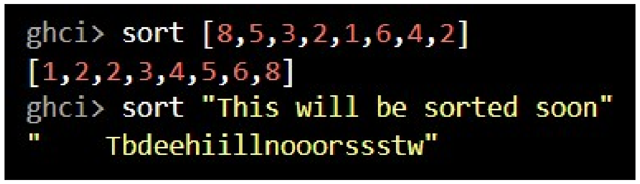
- it drops all the elements while the predicate is true. Once predicate equates to False, it returns the rest of the list.



```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

sort

- Sorts a list. The type of the elements in the list has to be part of the Ord typeclass, because if the elements of a list can't be put in order, then the list can't be sorted.

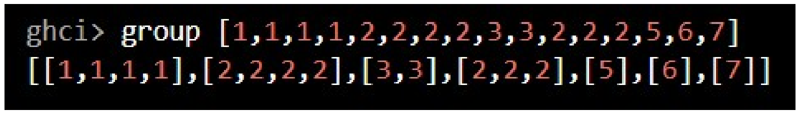


```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

Data.List functions

group

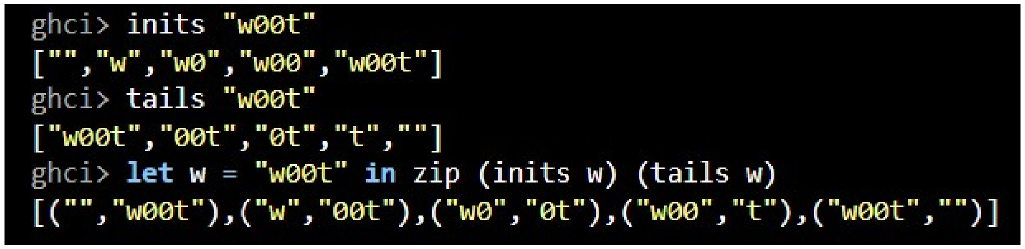
- takes a list and groups adjacent elements into sublists if they are equal.



```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

inits and tails

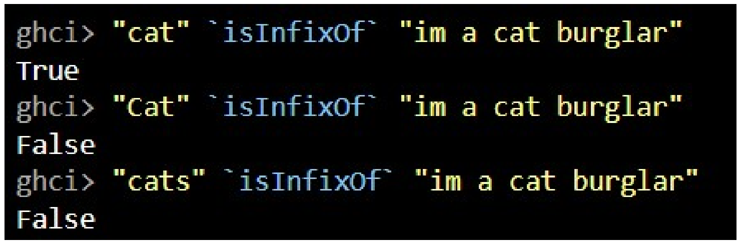
- Similar to init and tail, only that they recursively apply that to a list until there's nothing left.



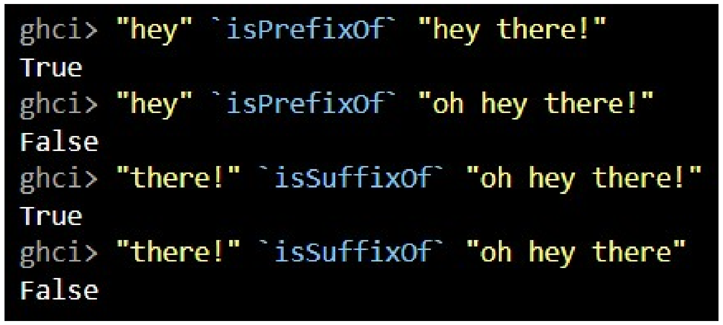
```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Data.List functions

isInfixOf, isPrefixOf and isSuffixOf



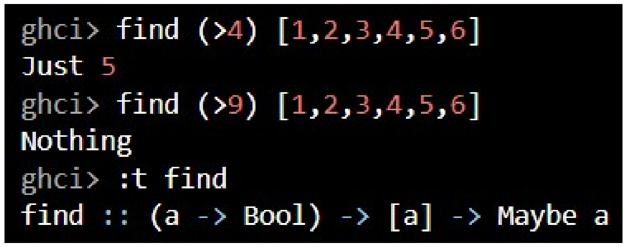
```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```



```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

find

- Takes a list and a predicate and returns the first element that satisfies the predicate. But it returns that element wrapped in a Maybe value.



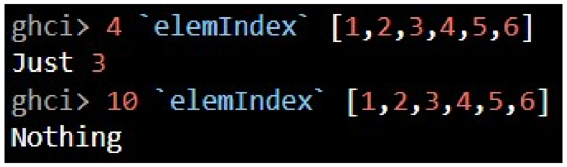
```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

**The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing).*

Data.List functions

elemIndex

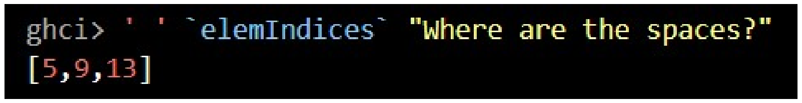
- Returns the index of the element we're looking for. If that element isn't in our list, it returns a Nothing.



```
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

elemIndices

- Returns a list of indices, in case the element we're looking for crops up in our list several times.

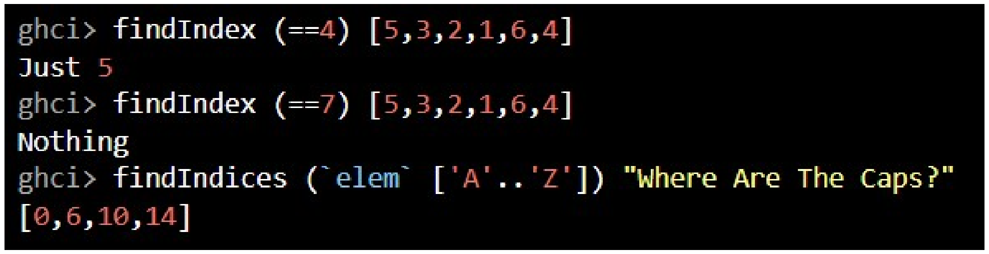


```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

Data.List functions

findIndex and findIndices

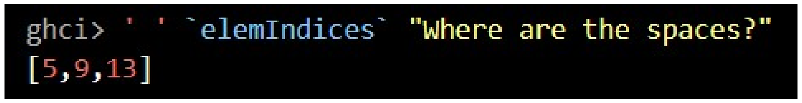
- First one returns the index of the first element that satisfies the predicate. The later one returns the indices of all elements that satisfy the predicate in the form of a list.



```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

elemIndices

- Returns a list of indices, in case the element we're looking for crops up in our list several times.

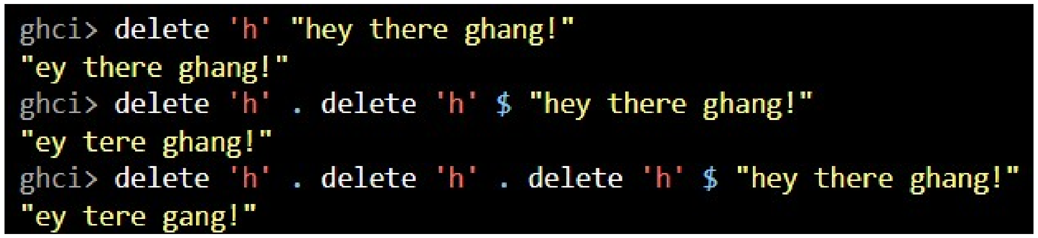


```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

Data.List functions

delete

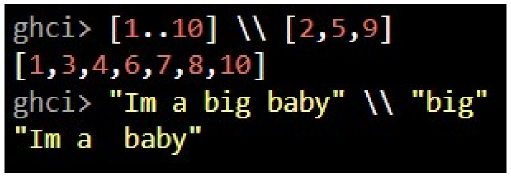
- takes an element and a list and deletes the first occurence of that element in the list.



```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

\

- is the list difference function. It acts like a set difference, basically. For every element in the right-hand list, it removes a matching element in the left one.

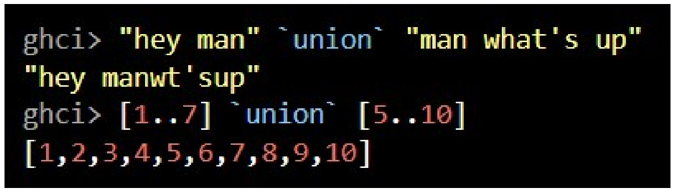


```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

Data.List functions

union

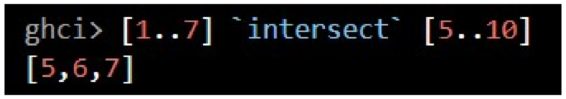
- It returns the union of two lists.



```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

intersect

- It returns only the elements that are found in both lists.

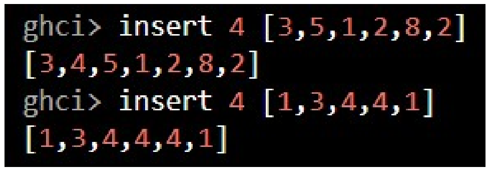


```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

Data.List functions

insert

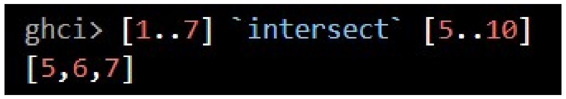
- takes an element and a list of elements that can be sorted and inserts it into the last position where it's still less than or equal to the next element.



```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

intersect

- It returns only the elements that are found in both lists.



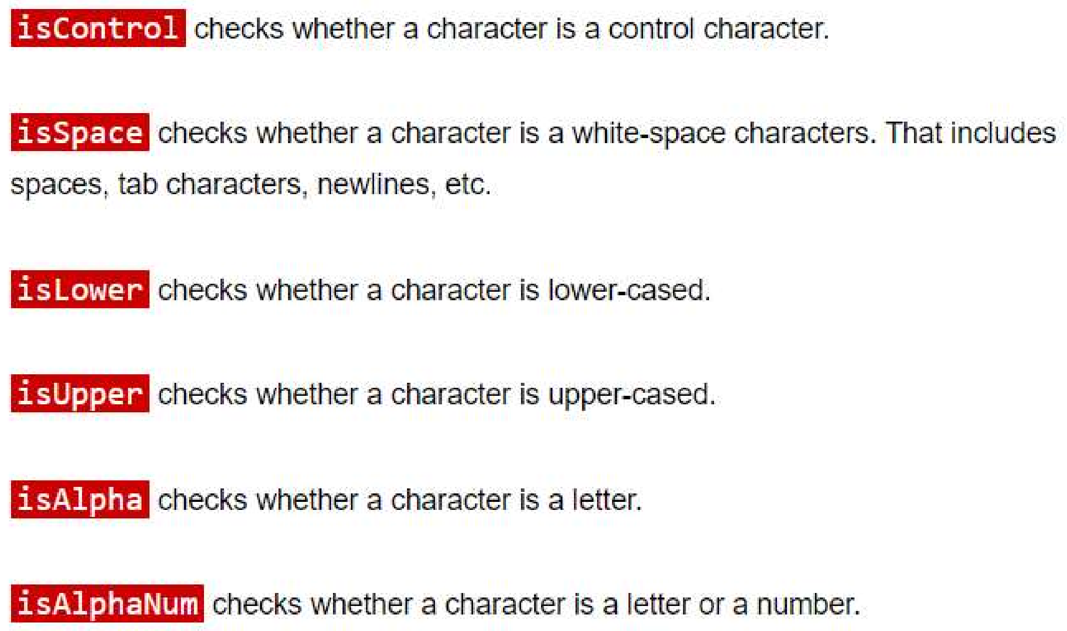
```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

Data.char

Data.char

- The Data.Char module exports functions that deal with characters.
- It's also helpful when filtering and mapping over strings because they're just lists of characters.
- Data.Char exports a bunch of predicates over characters. That is, functions that take a character and tell us whether some assumption about it is true or false. Here's what they are:

Data.char



`isControl` checks whether a character is a control character.

`isSpace` checks whether a character is a white-space characters. That includes spaces, tab characters, newlines, etc.

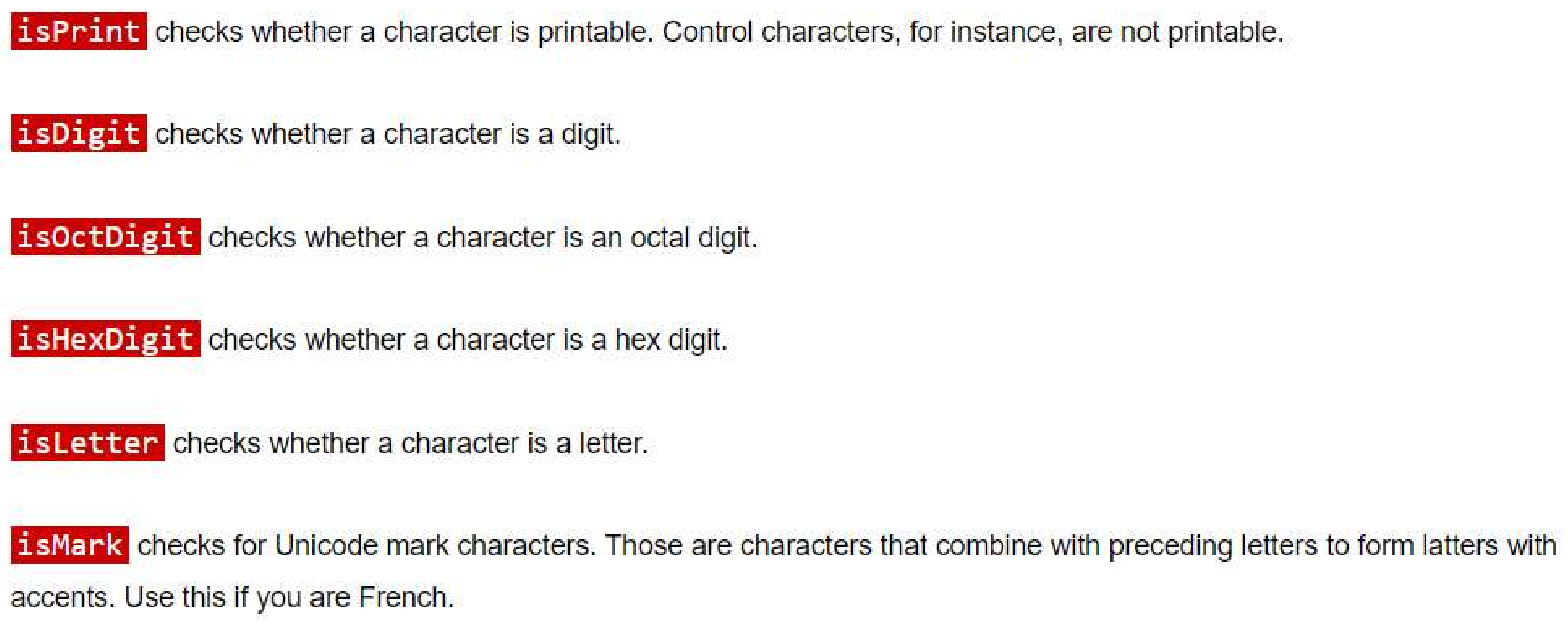
`isLower` checks whether a character is lower-cased.

`isUpper` checks whether a character is upper-cased.

`isAlpha` checks whether a character is a letter.

`isAlphaNum` checks whether a character is a letter or a number.

Data.char



`isPrint` checks whether a character is printable. Control characters, for instance, are not printable.

`isDigit` checks whether a character is a digit.

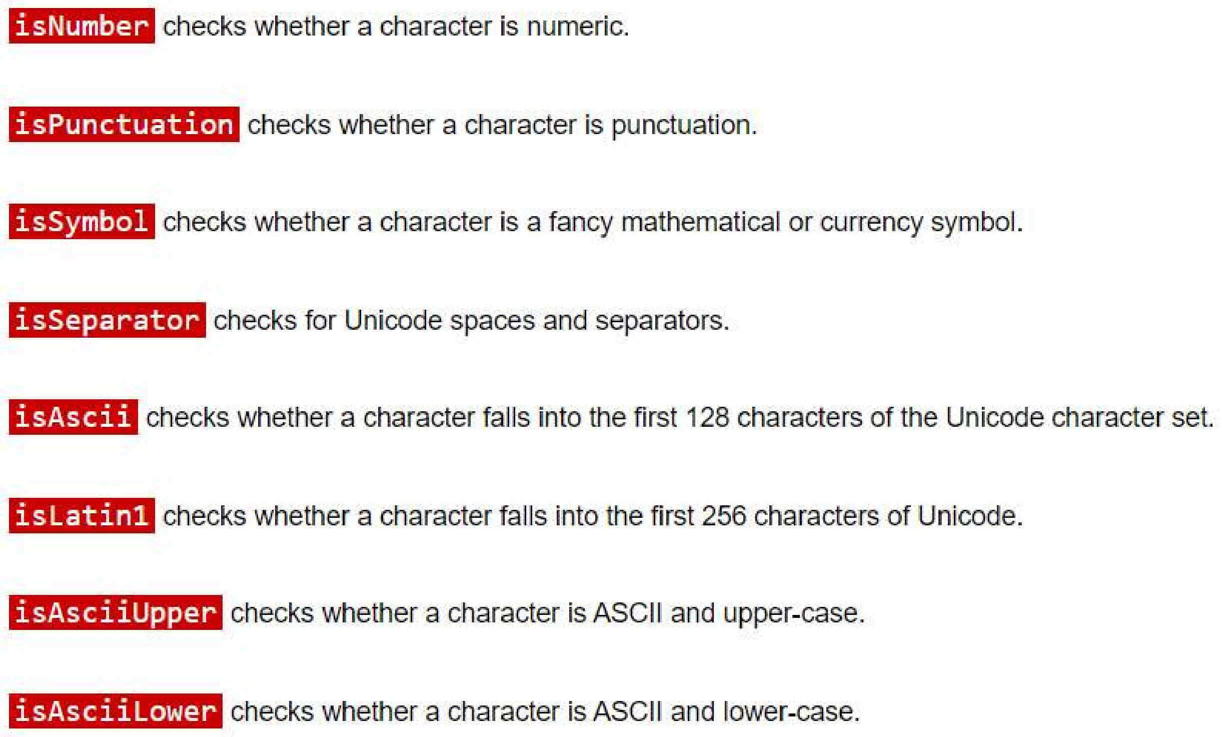
`isOctDigit` checks whether a character is an octal digit.

`isHexDigit` checks whether a character is a hex digit.

`isLetter` checks whether a character is a letter.

`isMark` checks for Unicode mark characters. Those are characters that combine with preceding letters to form latters with accents. Use this if you are French.

Data.char



`isNumber` checks whether a character is numeric.

`isPunctuation` checks whether a character is punctuation.

`isSymbol` checks whether a character is a fancy mathematical or currency symbol.

`isSeparator` checks for Unicode spaces and separators.

`isAscii` checks whether a character falls into the first 128 characters of the Unicode character set.

`isLatin1` checks whether a character falls into the first 256 characters of Unicode.

`isAsciiUpper` checks whether a character is ASCII and upper-case.

`isAsciiLower` checks whether a character is ASCII and lower-case.

Data.char

`toUpper` converts a character to upper-case. Spaces, numbers, and the like remain unchanged.

`toLower` converts a character to lower-case.

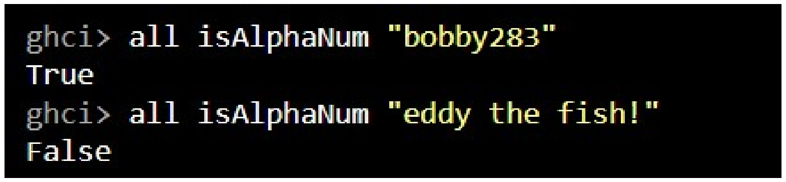
`toTitle` converts a character to title-case. For most characters, title-case is the same as upper-case.

`digitToInt` converts a character to an `Int`. To succeed, the character must be in the ranges `'0'..'9'`, `'a'..'f'` or `'A'..'F'`.

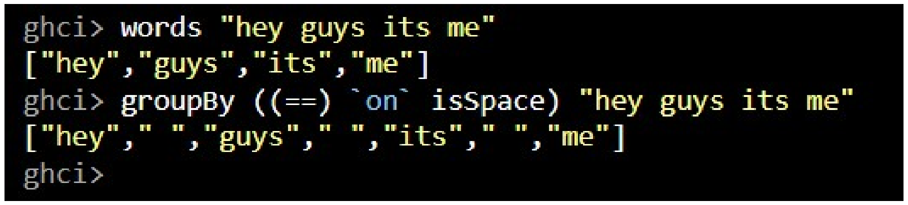
`intToDigit` is the inverse function of `digitToInt`. It takes an `Int` in the range of `0..15` and converts it to a lower-case character.

The `ord` and `chr` functions convert characters to their corresponding numbers and vice versa:

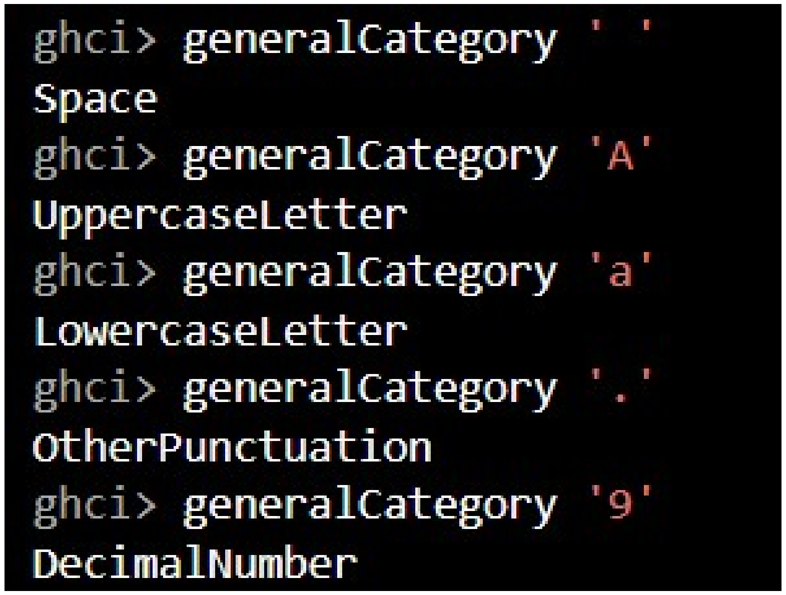
Example of Data.char functions



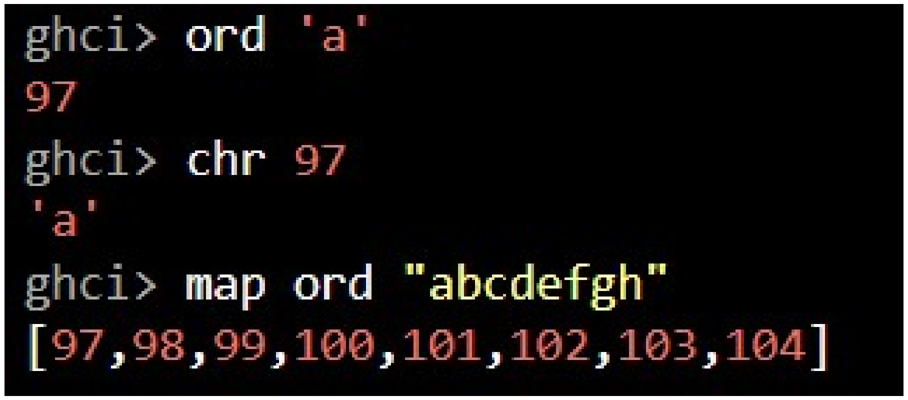
```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```



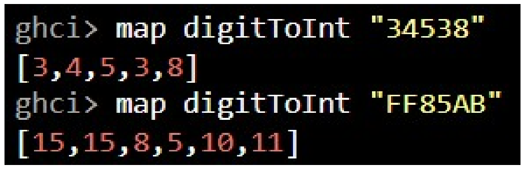
```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
ghci>
```



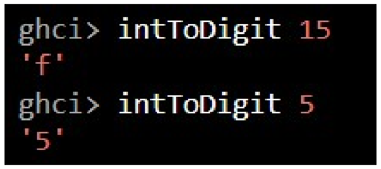
```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
```



```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```



```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

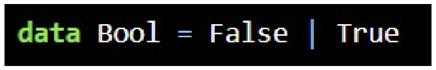


```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

User defined data type - Data

User defined Types - Data

- The data keyword can be used to define a type.

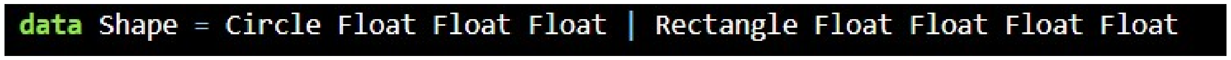


```haskell
data Bool = False | True
```

- data means that we're defining a new data type.
- The part before the = denotes the type, which is Bool.
- The parts after the = are value constructors. They specify the different values that this type can have.
- The | is read as or.
- We can read this as: *the Bool type can have a value of True or False.* *Both the type name and the value constructors must be capital cased*.

Data - Example

- Let's think about how we would represent a shape in Haskell.
  - One way would be to use tuples. A circle could be denoted as (43.1, 55.0, 10.4) where the first and second fields are the coordinates of the circle's center, and the third field is the radius.
- A better solution would be to make our own type to represent a shape. Let's say that a shape can be a circle or a rectangle.

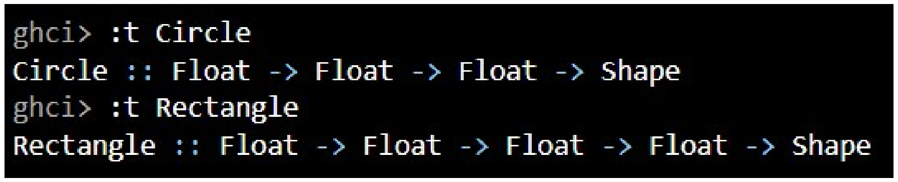


```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

- The Circle value constructor has three fields, which take floats. So when we write a value constructor, we can optionally add some types after it and those types define the values it will contain. The Rectangle value constructor has four fields which accept floats.

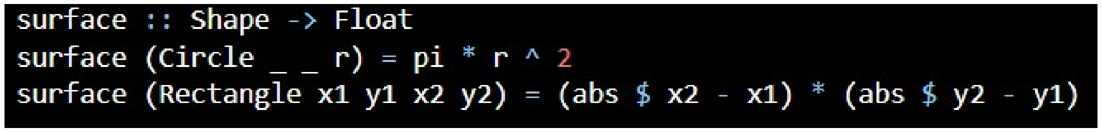
Data - Example

- Type Signatures of the Shape :-



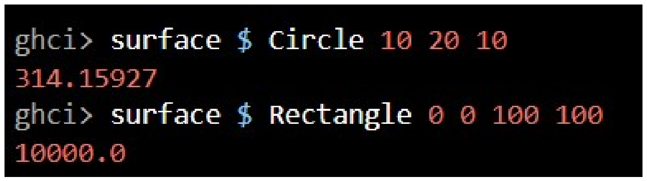
```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

- Let's make a function that takes a shape and returns its surface.



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

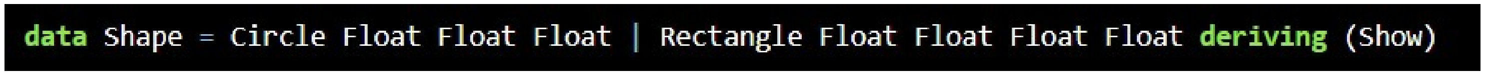
- Note that the function takes a shape and returns a float.



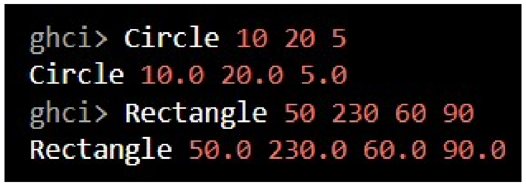
```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Data - Example

- If we try to just print out Circle 10 20 5 in the prompt, we'll get an error. Haskell doesn't know how to display our data type as a string (yet).
- Haskell first runs the show function to get the string representation of our value and then it prints that out to the terminal. To make our Shape type part of the Show typeclass, we modify it like this:



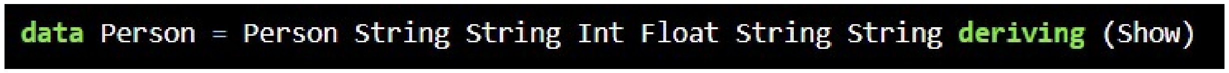
```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```



```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

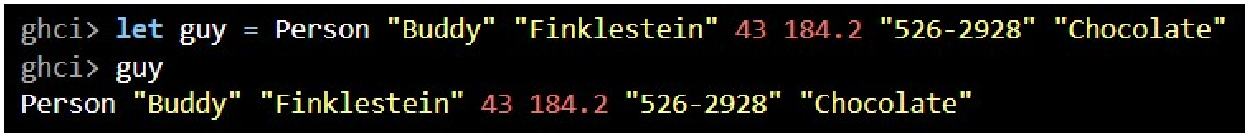
Data - Example

- Let’s assume we want to store about that person is: first name, last name, age, height, phone number, and favorite ice-cream flavor.



```haskell
data Person = Person String String Int Float String String deriving (Show)
```

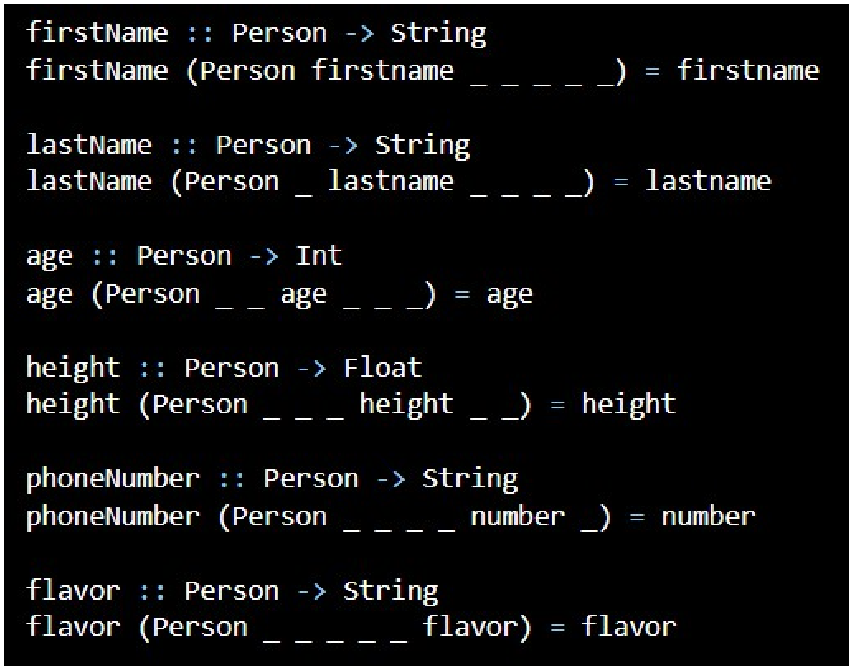
- Let’s make a person



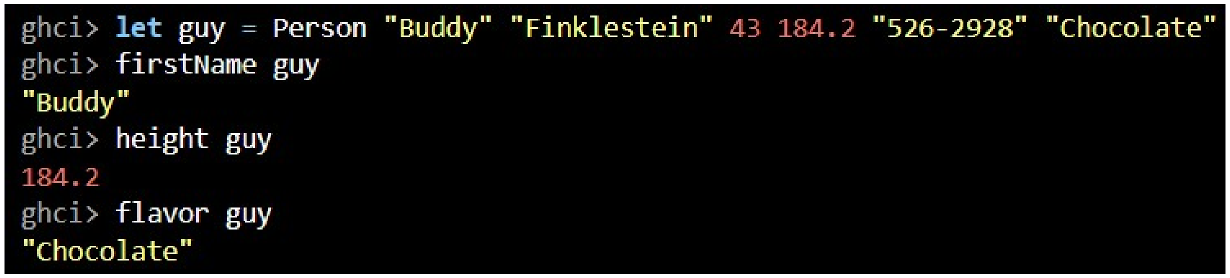
```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Data - Example

- What if we want to create a function to get separate info from a person.



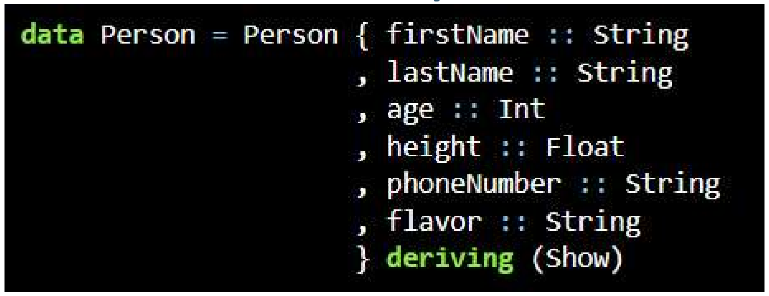
```haskell
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

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

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```



```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

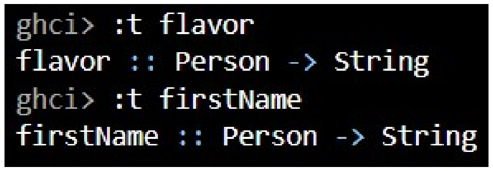
Data – using Record Syntax

- An alternative way to write data types - with record syntax.



```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

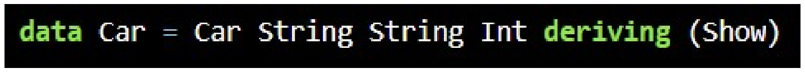
- The main benefit of this is that it creates functions that lookup fields in the data type. By using record syntax to create this data type, Haskell automatically made these functions: firstName, lastName, age, height, phoneNumber and flavor.



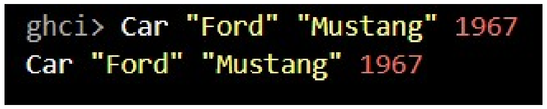
```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Data – using Record Syntax

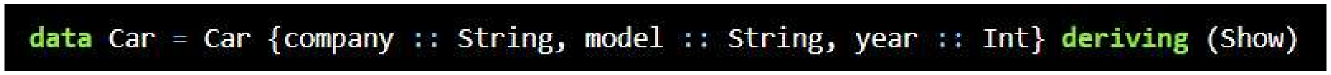
- Another benefit to using record syntax - when we derive Show for the type, it displays it differently if we use record syntax to define and instantiate the type.
- Say we have a type that represents a car. We want to keep track of the company that made it, the model name and its year of production. See the difference below :-



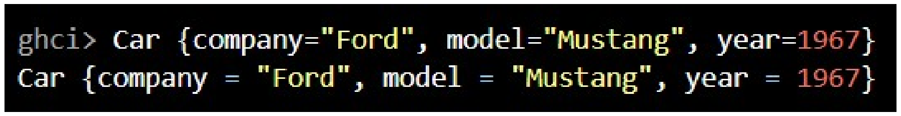
```haskell
data Car = Car String String Int deriving (Show)
```



```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```



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

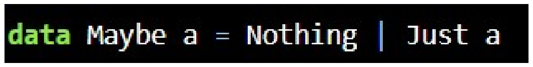


```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

Use record syntax when a constructor has several fields and it's not obvious which field is which.

Type parameter

- A value constructor can take some values parameters and then produce a new value.



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

- The a here is the type parameter. And because there's a type parameter involved; we call Maybe a type constructor.
- If we pass Char as the type parameter to Maybe, we get a type of Maybe Char. The value Just 'a' has a type of Maybe Char