Lambda Functions:

- It is an **anonymous function**, which is a function without giving it a name.
- A lambda function is denoted by the "****" character.
- Syntax: **\(var) -> (expression)**
  E.g.

```
main = do
    putStr("The next number after 5 is ")
    print((\x -> x + 1) 5)
```

```
*Main> main
The next number after 5 is 6
```

- **Note:** Lambda functions can be used as a substitute for missing parameters.
- If you intend 2 parameters, the Haskell culture is to model it as a nested function: **\x -> (\y -> 2*x - 3*y)** (those parentheses can be omitted). This creates a function that maps the 1st parameter to a function that takes the 2nd parameter. Doing this is called **currying**.
  The shorthand way of doing it is: **\x y -> 2*x - 3*y**
  E.g.

```
main2 = do
    putStr("The value of 2*3 - 3*1 is ")
    print((\x -> (\y -> 2*x - 3*y))3 1)

main3 = do
    putStr("The value of 2*3 - 3*1 is ")
    print((\x y -> 2*x - 3*y)3 1)
```

```
*Main> main2
The value of 2*3 - 3*1 is 3
*Main> main3
The value of 2*3 - 3*1 is 3
```

  Notice that both ways work and give the same result.
- Recall from earlier **diffSq x y = (x - y) * (x + y)**. This can be written as **diffSq = \x y -> (x - y) * (x + y)** or even **diffSq x = \y -> (x - y) * (x + y)**.
  E.g.

```
-- Example of diffSq using lambda functions.
-- The first example doesn't use currying while the second example does.
diffSq = \x y -> (x - y) * (x + y) -- Doesn't use currying.
diffSq2 x = \y -> (x - y) * (x + y) -- Uses currying.
```

```
*Main> diffSq 4 5
-9
*Main> diffSq2 4 5
-9
```

- When applying a function to 2 parameters, such as doing function a b, that's shorthand for (function a) b.

E.g.
diffSq 10 5 is shorthand for **(diffSq 10) 5**.

```
*Main> (diffSq 4) 5
-9
```

Compare this with the diffSq examples from above and notice that you get the same result.

- Note: It is possible to use "diffSq 10" alone. This is called a **partial application**. **Partial application** is when you decide to use a function but not give it all of the needed parameters. When it is evaluated, here is what happens:
  diffSq 10
  → (\x y -> (x - y) * (x + y)) 10
  → \y -> (10 - y) * (10 + y)
- Typewise, **X -> Y -> A** is shorthand for **X -> (Y -> A)**.

Higher Order Function:

- **Higher Order Functions** are a unique feature of Haskell where you can use a function as an input or output argument.
  Note: We use () to show that a function takes a function as an input.
- E.g.

```
four_plus_seven :: (Int -> Int) -> Int
four_plus_seven f = f 4 + f 7
```

```
*Main> four_plus_seven (\x -> x*2)
22
*Main> four_plus_seven (\x -> x^2)
65
*Main> four_plus_seven (\x -> x+2)
15
```

The first function multiplies each variable by 2. 4*2 + 7*2 = 22.
The second function squares each variable. $4^2 + 7^2 = 65$.
The third function increases each variable by 2. 4+2 + 7+2 = 15.
In the first picture above, **(Int -> Int)** shows that four_plus_seven takes in a function as an input and that function takes in an Int as an input and outputs something of type Int.

- E.g.

```
four_plus_seven :: (Int -> Int) -> Int
four_plus_seven f = f 4 + f 7

random_function :: Int -> Int
random_function x = 5*x + 12
```

```
*Main> four_plus_seven (random_function)
79
```

5*4 + 12 = 32
5*7 + 12 = 47
32 + 47 = 79

Parametric Polymorphism:

- A **polymorphic** function is a function that works for many different types.
- **Polymorphic:** Can become one of many types.

- **Monomorphic:** Stuck with being one single type.
- Also known as **generics** in other languages.
- **Type variables** always begin in lowercase whereas **concrete types** like Int or String always start with an uppercase letter.
- Just as a variable represents some value of a given type, a **type variable** represents some type. A **type variable** represents one type across the type signature and function definition in the same way a variable represents a value throughout the scope it's defined in.
- E.g.

```
add :: Int -> Int -> Int
add x y = x + y

add2 :: Num x => x -> x -> x
add2 a b = a + b
```

```
*Main> add 2 3
5
*Main> add 2.0 3.0

<interactive>:124:5: error:
    • No instance for (Fractional Int) arising from the literal ‘2.0’
    • In the first argument of ‘add’, namely ‘2.0’
      In the expression: add 2.0 3.0
      In an equation for ‘it’: it = add 2.0 3.0
*Main> add2 2 3
5
*Main> add2 2.0 3.0
5.0
*Main> add2 2.1 3.1
5.2
*Main> add2 2 3.4
5.4
*Main> add2 2 (-1.5)
0.5
```

In the add function, only Integers are allowed. Hence, when I tried doing add 2.0 3.0, I got an error. However, in add2, as long as the inputs are numbers, I can add integers, floats or a mix.

Note: **Num x =>** just means that x must be of type Num, or x must be a number. I need to put this or else I get an error. This is because if I don’t specify the type, I could, theoretically, add 2 non-numbers, which would cause an error. Hence, Haskell mandated that I put the **Num x =>** part.

- E.g.

rep2 :: a -> [a]

In **a -> [a]**, the “a” there is a **type variable** or **type parameter**. Names of type variables are up to you, doesn't have to be “a”, but does have to start with lowercase.
E.g. element, myElementType, etc

Note: Type constants/Concrete types, names of built-in types and defined types, start with uppercase.
E.g. Integer, Bool, String

- **Note:** The choice of the type is up to the user, not the provider. Furthermore, in parametric polymorphism, the "parametric" part means that the provider is not told what the user chooses. As a result, the code can be inflexible. However, it's easy to test your code.
- Generally, flexibility for the implementer is in direct conflict with predictability for the user and vice versa.

Map:

- A **map** is the name of a higher-order function that applies a given function to each element of a functor, such as a list, returning a list of results in the same order. It is often called apply-to-all when considered in functional form.
- Can be written in 2 ways:
    1. map :: (a -> b) -> [a] -> [b]
    2. map :: (a -> b) -> ([a] -> [b])
- E.g.

```
square :: Int -> Int
square x = x ^ 2
```

```
*Main> map square [1,2,3,4,5,6,7,8,9,10]
[1,4,9,16,25,36,49,64,81,100]
*Main> square [1,2,3,4,5,6,7,8,9,10]

<interactive>:13:8: error:
    • Couldn't match expected type ‘Int’ with actual type ‘[Integer]’
    • In the first argument of ‘square’, namely ‘[1, 2, 3, 4, ....]’
      In the expression: square [1, 2, 3, 4, ....]
      In an equation for ‘it’: it = square [1, 2, 3, ....]
```

By having the word "map", it allowed me to use the square function on a list.

- E.g.

```
import Data.Char
ascii_conversion x = ord x
```

```
*Main> ascii_conversion ['a', 'b']

<interactive>:16:18: error:
    • Couldn't match expected type ‘Char’ with actual type ‘[Char]’
    • In the first argument of ‘ascii_conversion’, namely ‘['a', 'b']’
      In the expression: ascii_conversion ['a', 'b']
      In an equation for ‘it’: it = ascii_conversion ['a', 'b']
*Main> map ascii_conversion ['a', 'b']
[97,98]
```

Notice that when I put the keyword "map" at the beginning, I can run ascii_conversion on a list.

- What it does by example:
  map ord ['a', 'b', 'c']
  = [ord 'a', ord 'b', ord 'c']
  = [97, 98, 99]
- Consider this: map (map ord) [['a', 'b', 'c'], ['x', 'y', 'z']] :: [[Int]]
  Detailed type breakdown:
  - inner map :: (Char -> Int) -> ([Char] -> [Int])
    (I'm choosing a=Char, b=Int)
  - map ord :: [Char] -> [Int]
  - outer map :: ([Char] -> [Int]) -> [[Char]] -> [[Int]]
    (I'm choosing a=[Char], b=[Int])
  - map (map ord) :: [[Char]] -> [[Int]]
- What it does by example:
  map (map ord) [['a', 'b', 'c'], ['x', 'y', 'z']]
  = [map ord ['a', 'b', 'c'], map ord ['x', 'y', 'z']]
  = [[ord 'a', ord 'b', ord 'c'], [ord 'x', ord 'y', ord 'z']]
  = [[97, 98, 99], [120, 121, 122]]
- E.g.

```
Prelude Data.Char> map (map (map ord)) [[['a', 'b', 'c'], ['x', 'y', 'z']]]
[[[97,98,99],[120,121,122]]]
```

- **Note:** To use ord, you need to do import Data.Char.

Type-specific behaviour preview:

- Consider the code below:

```
square :: a -> a
square x = x^2
```

The square function takes in a number and returns the square of that number. Since any number, not just integers, can be squared, we want to use parametric polymorphism. However, there's an issue. What happens if the user enters a string or boolean? To avoid this problem, we have to do the following:

```
square :: Num a => a -> a
square x = x^2
```

By putting the Num a => part, we are saying that "a" must be a number.

User-defined types:

- Also called **algebraic data types**.
- We can define our own types using the keyword **data**.
- Each option must start with an uppercase letter.
- We use | to say alternatively.
- There needs to be at least one case, and each case can have 0 or more fields.

- E.g.

```
-- Created a new data type called Area.
data Area
    = Circle Float -- Case 1: Circle. It has one field, float.
    | Square Float Float -- Case 2: Square. It has 2 fields, float, float.
    | Triangle Float Float -- Case 3: Triangle. It has 2 fields, float, float.

surface :: Area -> Float
surface (Circle r) = pi * r ^ 2
surface (Square b h) = b * h
surface (Triangle b h) = b * h / 2
```

```
*Main Data.Char> surface $ Circle 10
314.15927
*Main Data.Char> surface $ Square 2 2
4.0
*Main Data.Char> surface $ Triangle 2 3
3.0
```

```
*Main Data.Char> :t Circle
Circle :: Float -> Area
*Main Data.Char> :t Square
Square :: Float -> Float -> Area
*Main Data.Char> :t Triangle
Triangle :: Float -> Float -> Area
```

Here, the type name is Area.
Circle, Square and Triangle are called **data constructors** or **tags**. As stated before, all these data constructors must be capitalized.
Note: These are not OOP constructors. It's only labelling, not arbitrary initialization code.
Note: These are not OOP subclasses/subtypes either. Circle is not a subtype, it's a term and value.

Recursive types:

- A recursive data type is a data definition that refers to itself.
- This lets us define even more interesting data structures such as linked lists and trees.
- The line, **deriving (Eq, Show)**, is called the **deriving clause**. It specifies that we want the compiler to automatically generate instances of the Eq and Show classes. The EQ type class is an interface which provides the functionality to test the equality of an expression. The Show type class has a functionality to print its argument as a String. Whatever may be its argument, it always prints the result as a String.

- E.g.

```haskell
data IntList = EndOfIntList | ValAndNext Integer IntList
  deriving(Show, Eq)

example = ValAndNext 0 (ValAndNext 1 (ValAndNext 2 (ValAndNext 3 (ValAndNext 4 (ValAndNext 5 EndOfIntList)))))

-- numList makes a list of numbers from 0 to n.
numList :: Integer -> IntList
numList n = make 0
    where
        make i | i > n = EndOfIntList
               | otherwise = ValAndNext i (make (i+1))

myISum :: IntList -> Integer
myISum EndOfIntList = 0
myISum (ValAndNext x xs) = x + myISum xs
```

```
*Main Data.Char> numList 5
ValAndNext 0 (ValAndNext 1 (ValAndNext 2 (ValAndNext 3 (ValAndNext 4 (ValAndNext 5 EndOfIntList)))))
*Main Data.Char> myISum example
15
```

Recursion & Lists:

- E.g. Consider the example below:

  A value of type MyIntegerList is one of:
  1. INil
  2. ICons x xs, if x::Integer and xs::MyIntegerList

```
data MyIntegerList = INil | ICons Integer MyIntegerList
  deriving (Show, Eq)

exampleMyIntegerList = ICons 4 (ICons (-10) INil)

-- `from0to n` builds a MyIntegerList from 0 to n-1
from0to :: Integer -> MyIntegerList
from0to n = make 0
 where
   make i | i >= n = INil
          | otherwise = ICons i (make (i+1))

mylSum :: MyIntegerList -> Integer
mylSum INil = 0
mylSum (ICons x xs) = x + mylSum xs
```

Recursion & Binary Trees:

- E.g. Consider the example below:

  A value of type IntegerBST is one of:
  1. IEmpty
  2. INode lt x rt, if lt::IntegerBST, x::Integer, rt::IntegerBST

```
data IntegerBST = IEmpty | INode IntegerBST Integer IntegerBST
  deriving Show

exampleIntegerBST = INode (INode IEmpty 3 IEmpty) 7 (INode IEmpty 10 IEmpty)

ibstInsert :: Integer -> IntegerBST -> IntegerBST
ibstInsert k IEmpty =
  INode IEmpty k IEmpty
ibstInsert k inp@(INode left key right)
  | k < key = INode (ibstInsert k left) key right
  | k > key = INode left key (ibstInsert k right)
  | otherwise = inp   -- INode left key right
```

Note: Since this is functional programming with immutable trees, “insert” means produce a new tree that is like the input tree but with the new key. Maybe it's better to say “the tree plus k”.

Polymorphic Types:

- Consider the example below:

```
data MyList a = Nil | Cons a (MyList a) deriving (Eq, Show)

exampleMyListI :: MyList Integer
exampleMyListI = Cons 4 (Cons (-10) Nil)

exampleMyListS :: MyList String
exampleMyListS = Cons "albert" (Cons "bart" Nil)
```

  These are homogeneous lists. They can't have different item types in the same list. For example, Cons "albert" (Cons True Nil) is illegal because what would be its type, MyList String? MyList Bool?
- Some polymorphic algebraic data types from the standard library as further examples:
- Maybe:

```
data Maybe a = Nothing | Just a
-- Great for: Sometimes there is no answer
```

- Either:

```
data Either a b = Left a | Right b
-- Great for: Having two possible types.
```