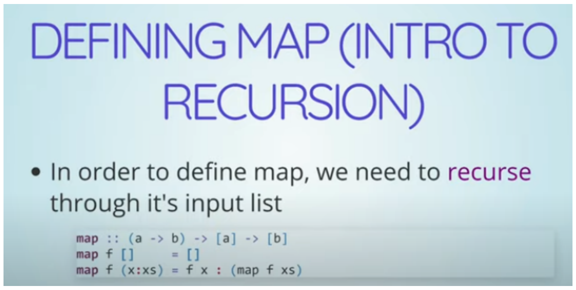
Presentation Week04: Recursion & Data types

RECURSION:

Using the function you are defining within itself.
For every recursive definition you NEED a base case. This means that at one point, the function must define some sort of behavior that does not use itself in the definition to end the recursive call. Without a base case, the function would call itself recursively indefinitely.




```
Map f [] = []              ---->      BASE CASE
Map f(x:xs)               -----> Recursive portion
```

Example :

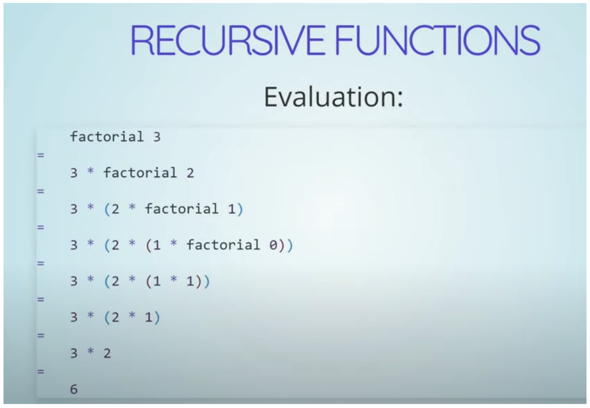
```
F:    add x = x + 1

x:xs = [1,2,3]

map f (x:xs) = add [1] : map add [2,3]
            =add [1] : add[2] : map add [3]
            =add [1] : add[2] : add[3] : map add [].    ---This is why a base case is needed, if there was
            no definition 'map f [] = [] '
                                                           the function would not stop calling itself.
            =add [1] : add[2] : add[3] : []
            =add [1] : add[2] : [4] : []
            =add [1] : [3] : [4] : []
            = [2] : [3] : [4] : []
            =[2,3,4]
```

** remember colon operator adds an element to the beginning of a list.
** Pattern matches to one of the definitions of map at every call of itself, until a base case is reached

Recursive definition of Factorial Function:

```
factorial :: (Eq a, Num a) => a -> a
factorial 0 = 1
factorial x = x * factorial(x-1)
```

Factorial definition pattern matches on the input, and gives output



Another Example:

```
pow :: (Eq a ,Num a) => a -> a -> a
pow m 0 = 1
pow m n = m * pow m (n-1)
```

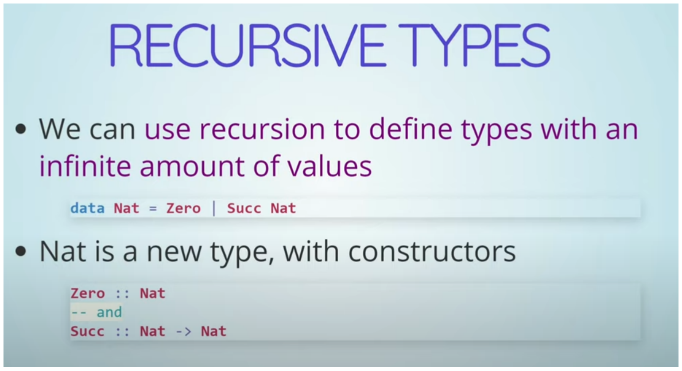
i.e. produces m^n output, recursively.

What happens if didn't have the base case? Well..

```
                  Pow 3 2 = 3 * pow 3 1
                        = 3 * (3* pow 3 0)
                        = 3 * (3* (3 * pow 3 -1) < -- never ends!
```

Recursive Types

-- Can define own types by the 'data' keyword.



I.e. Nat rebuilds natural number (any positive whole number)

So now, 'Zero' and 'Succ Nat' are of type 'Nat'

Zero = 0

Succ (Succ Zero) = 2

*When defining out own data types must write: 'deriving Show' In order for ghci to print.

i.e.

```
data Nat = Zero | Succ Nat -- where successor is plus 1 of another Natural Number
     deriving Show --must write this for ghci to print, must turn into string
-- makes an instant of the show class
```

Zero	0
Succ Zero	1
Succ Succ Zero	2

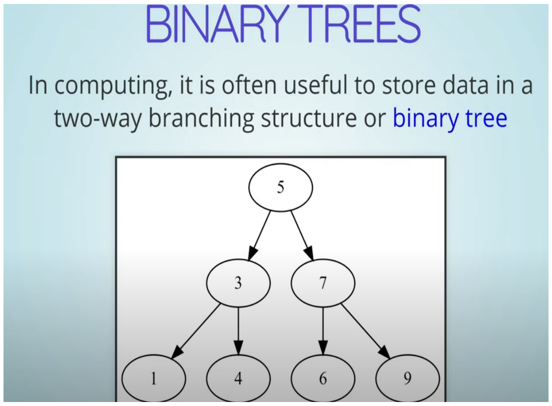
```
nat2int :: Num p => Nat -> p
nat2int Zero = 0
nat2int (Succ n) = 1 + (nat2int n)
```

Defining nat2int recursively, since nat is a recursive type with infinite amount of values.
I.e.

```
Nat2int (Succ Succ Zero)  = 1 + nat2int (Succ Zero)
                      = 1 + 1 + nat2int Zero
                      = 1 + 1 + 0
                      = 2
```

** now that we defined data type, we can use this data type and create functions.

Binary Tree




```
Binary trees contain Nodes
nodes with no branches at all, called leaves
Each node has two branches, child nodes; HENCE BINARY
```

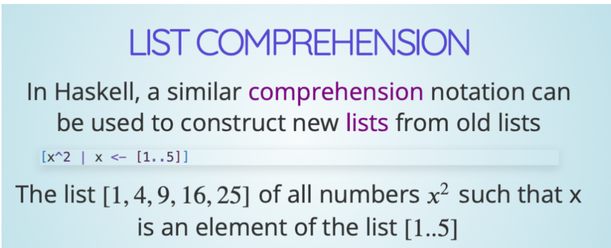
Sorted Trees : Every node/leaf on left side of Node is LESS THAN node.
: Every node /leaf on right side of Node is GREATER THAN node

```
         4
    |                  |
    2                   5
|                  |
1                  3

[1,2,3,4,5]

--   d
-- / \
-- b f
-- / \ / \
-- a c e g

Tree1  =  Node  ‘b’ (Leaf ‘a’) (Leaf ‘c’)

Node 'd' (Tree1 ) (Tree t2)

Node 'd' (Node  b (Leaf a) (Leaf c) ) (Tree t2)

      b
    |  |
    a  c
```

Lists:

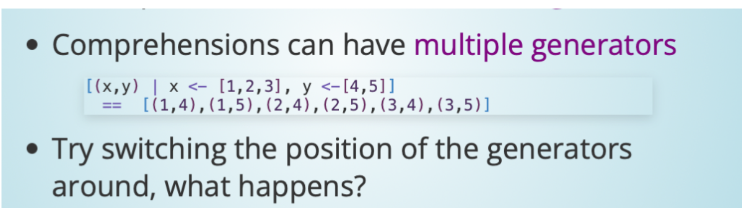
- Groups together values
- Lists are created by putting values inside square brackets separated by commas [1,2,3,4,5]

List comprehension



- Can specify new lists from old lists

X<-[1..5] called a GENERATOR ! List comprehensions can have multiple generators



i.e. [(x,y) | y<-[4,5], x<-[1,2,3]]
== [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]. <<--- changed the order

Guards

Restrict values produced by generator.

i.e.

List = [x | x< [1..10], even x]

What will be produced? ->> [2,4,6,8,10]

Type Declaration

A new name for an existing type
i.e. Data declarations CREATE NEW types, will type declarations RENAME EXISTING types

i.e. type Pos = (Int , Int)

String = [Char] -> list of char ; type alias or type synonym

- Main use for them would be to make code more readable.

```
origin :: Pos
origin = (0,0)
```

Could've easily had it as:

```
origin :: (Int, Int)
origin :: (0,0)
```

Increases readability!!

Type Parameters:

```
type Pair a = (a, a) -- can now utilize polymorphism
```

Nested Types:

```
type Trans = Pos -> Pos
```

We created the type "pos" and nested it creating another type "Trans"

When creating data types: i.e. :t Circle (w/o the Float) would return signature Float -> Shape , treats it as if it were a function definition. This type must be completed with a float to be of type 'Shape'

Data Parameters

- Data declarations can have parameters (can involve polymorphic variables)

i.e. data Maybe a = Nothing | Just a

*This will take in a value of any type 'a' and return either Nothing OR Just 'a'; i.e. can create multiple varieties of this type.

DERIVING

Deriving (Show, Eq) will automatically create instances of the type class mentioned i.e.

```
data Maybe1 a = Nothing1 | Just1 a
    deriving (Show, Eq)
```

```
instance [safe] Eq a => Eq (Maybe1 a)
  -- Defined at /Users/alessandraguerinoni/Desktop/1JC3/Haskell Tutorial Projects/Week04Tut/src/Lib.hs:284:21
```

Type Constructors vs Data Constructors:

Constructor:

Constructor can mean:
Type constructor
Data constructor (or value constructor)

```
Data Bool = True | False
```

A **type constructor is the thing on the left hand side of the equals sign.**
The **data constructor(s) are the things on the right hand side of the equals sign.**
You use type constructors where a type is expected & you use data constructors where a value is expected.

Data constructors

```
data Colour = Red | Green | Blue
```

Here, we have three data constructors.
Colour is a type, and Green is a constructor that contains a value of type Colour.

A data constructor either contains a value like a variable would, or takes other values as its argument and creates a new value.

```
data SBTree = Leaf String
            | Branch String SBTree SBTree
```

type SBTree that contains two data constructors.
there are two functions (namely Leaf and Branch) that will construct values of the SBTree type.

Type constructors

BTree is a type constructor.

```
data BTree a = Leaf a
             | Branch a (BTree a) (BTree a)
```

type variable 'a' as a parameter to the type constructor.
In this declaration, BTree has become a function. It takes a type as its argument and it returns a new type.

If we pass in, say, Bool as an argument to BTree, it returns the type BTree Bool,

If you want to, you can view BTree as a function with the kind
BTree :: * -> * -- these are called "Kinds"

BTree is from a concrete type -> concrete type.

Concrete type (examples include Int, [Char] and Maybe Bool) which is a type that can be assigned to a value

Wrapping up

A data constructor is a "function" that takes 0 or more values and gives you back a new value.
A type constructor is a "function" that takes 0 or more types and gives you back a new type.

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

Here, Maybe is a type constructor that returns a concrete type.
Just is a data constructor that returns a value.
Nothing is a data constructor that contains a value.

Maybe :: * -> *
In other words, Maybe takes a concrete type and returns a concrete type.