CS331
Haskell Tutorial 03

A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Factorial Example
- Higher Order Function
- Currying, Composition
- Lazy Evaluation
- List Comprehension

Factorial I

```
fact n =
   if n == 0 then 1
   else n * fact (n - 1)
```

This is an extremely conventional definition.

Factorial II

```
fact n
    | n == 0        = 1
    | otherwise = n * fact (n - 1)
```

Each | indicates a “guard.”

Notice where the equal signs are.

Factorial III

```
fact n = case n of
   0 -> 1
   n -> n * fact (n - 1)
```

This is essentially the same as the last definition.

Factorial IV

You can introduce new variables with

`let` ***declarations*** `in` ***expression***

```
fact n
   | n == 0    = 1
   | otherwise = let m = n - 1 in n * fact m
```

Factorial V

You can also introduce new variables with

expression `where` ***declarations***

```
fact n
   | n == 0        = 1
   | otherwise = n * fact m
   where m = n - 1
```

Higher−Order Functions

- **Higher-order programming** treats functions as first-class,
  - Allowing them to be passed as parameters, returned as results or stored into data structures.
- This concept supports generic coding,
  - and allows programming to be carried out at a more abstract level.
- Genericity can be applied to a function
  - by letting specific operation/value in the function body to become parameters.

Higher order Functions

- Functions can be written in two main ways:

```
add x y       = x+y
add2(x,y)     = x+y
```

- The first version allows a function to be returned as result after applying a single argument.

```
inc   =  add 1
```

```
Prelude> add x y = x+y
Prelude> inc = add 1
Prelude > inc 5
6
Prelude>
```

Higher order Functions

- The second version needs all arguments. Same effect requires a lambda abstraction:

```
add2(x,y)      = x+y
inc     = \x -> add2(x,1)
```

```
Prelude> add2 (x+y) = x+y
Prelude> inc =  \x -> add2(x, 1)
Prelude > inc 5
6
Prelude>
```

Functions

- Functions can also be passed as parameters. Example:

```
map              :: (a->b) -> [a] -> [b]
map f []         = []
map f (x:xs)     = (f x) : (map f xs)
```

- Such higher-order function aids code reuse.

```
map (add 1) [1, 2, 3]      ) [2, 3, 4]
map add [1, 2, 3]          ) [add 1, add 2, add 3]
```

- Alternative ways of defining functions:

```
add              = \ x -> \ y -> x+y
add              = \ x y -> x+y
```

Haskell Brooks Curry

- Haskell Brooks Curry (September 12, 1900 – September 1, 1982)
- Developed Combinatorial Logic, the basis for Haskell and many other functional languages

Currying

- **Technique named after: logician Haskell Curry**
- **Currying** absorbs an argument into a function, returning a new function that takes one fewer argument
- `f a b = (f a) b`, where `(f a)` is a curried function
  - For example, if `avg = \x y -> (x + y) / 2` then `(avg 6)` returns a function
    - This new function takes one argument (`y`) and returns the average of that argument with `6`
  - Consequently, we can say that in Haskell, every function takes exactly one argument

Currying

- For example, if `avg = \x y -> (x + y) / 2` then `(avg 6)` returns a function
  - This new function takes one argument (`y`) and returns the average of that argument with `6`

```
Prelude> avg = \x y -> (x + y) / 2
Prelude> (avg 6) 20
```

Currying example

- “And”, `&&`, has the type `Bool -> Bool -> Bool`
- `x && y` can be written as `(&&) x y`
- If `x` is `True`,
  `(&&)x` is a function that returns the value of `y`
- If `x` is `False`,
  `(&&)x` is a function that returns `False`
  - It accepts `y` as a parameter, but doesn’t use its value

Slicing

- `negative = (< 0)`

```
Main> negative 5
False
Main> negative (-3)
True
Main> :type negative
negative :: Integer -> Bool
Main>
```

List

- List creation/declaration

```
myData = [1,2,3,4,5,6,7]
```

Operations on Lists I

head	[a] -> a	First element
tail	[a] -> [a]	All but first
:	a -> [a] -> [a]	Add as first
last	[a] -> a	Last element
init	[a] -> [a]	All but last
reverse	[a] -> [a]	Reverse

Operations on Lists II

!!	[a] -> Int -> a	Index (from 0)
take	Int -> [a] -> [a]	First n elements
drop	Int -> [a] -> [a]	Remove first n
nub	[a] -> [a]	Remove duplicates
length	[a] -> Int	Number of elements

Operations on Lists III

elem, notElem	a -> [a] -> Bool	Membership
concat	[[a]] -> [a]	Concatenate lists

Operations on Tuples

`fst`	`(a, b) -> a`	First of two elements
`snd`	`(a, b) -> b`	Second of two elements

…and nothing else, really.

Finite and Infinite Lists

[a..b]	All values a to b	[1..4] = [1, 2, 3, 4]
[a..]	All values a and larger	[1..] = positive integers
[a, b..c]	a step (b-a) up to c	[1, 3..10] = [1,3,5,7,9]
[a, b..]	a step (b-a)	[1, 3..] = positive odd integers

List Comprehensions-0

- Notation for constructing new lists from old:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to "set comprehension"

$$\{ x \mid x \in Odd \ \wedge \ x > 6 \}$$

List Comprehensions I

- `[` ***expression_using_x*** `| x <-` ***list*** `]`
  - read: <expression> where x is in <list>
  - `x <-` ***list*** is called a generator
- Example: `[ x * x | x <- [1..] ]`
  - This is the list of squares of positive integers
- `take 5 [x * x | x <- [1..]]`
  - `[1,4,9,16,25]`

List Comprehensions II

- `[` ***expression_using_x_and_y*** `| x <- `***list***`, y <- `***list***`]`
- `take 10 [x*y | x <- [2..], y <- [2..]]`
  - `[4,6,8,10,12,14,16,18,20,22]`
- `take 10 [x * y | x <- [1..], y <- [1..]]`
  - `[1,2,3,4,5,6,7,8,9,10]`
- `take 5 [(x,y) | x <- [1,2], y <- "abc"]`
  - `[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b')]`

List Comprehensions III

- [*expression_using_x* | *generator_for_x*, *test_on_x*]

- `take 5 [x*x | x <- [1..], even x]`
  - `[4,16,36,64,100]`

List Comprehensions IV

- `[x+y | x <- [1..5], even x, y <- [1..5], odd y]`
  - `[3,5,7,5,7,9]`
- `[x+y | x <- [1..5], y <- [1..5], even x, odd y]`
  - `[3,5,7,5,7,9]`
- `[x+y | y <- [1..5], x <- [1..5], even x, odd y]`
  - `[3,5,5,7,7,9]`

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1...5\}\}$$

The set $\{1,4,9,16,25\}$ of all numbers x^2 such that x is an element of the set $\{1...5\}$.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x ← [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note: # Lists Comprehensions

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.

- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Lists Comprehensions

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

Lists Comprehensions

- For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Factorial VI : Revisited

```
product [] = 1
product (a:x) = a * product x

fact n = product [1..n]
```

Dependent Generators

Later generators can <u>depend</u> on the variables that are introduced by earlier generators.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and y ≥ x.

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Guards

Using a guard we can define a function that maps a positive integer to its list of <u>factors</u>:

```
factors  :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

Guards

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime   :: Int → Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes   :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

A useful library function is <u>zip</u>, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

```
pairs    :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted    :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y | (x,y) ← pairs xs]
```

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all <u>positions</u> of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x’,i) ← zip xs [0..], x == x’]
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

```
"abc" :: String
```

Means ['a', 'b', 'c'] :: [Char].

Because strings are just special kinds of lists, any <u>polymorphic</u> function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string:

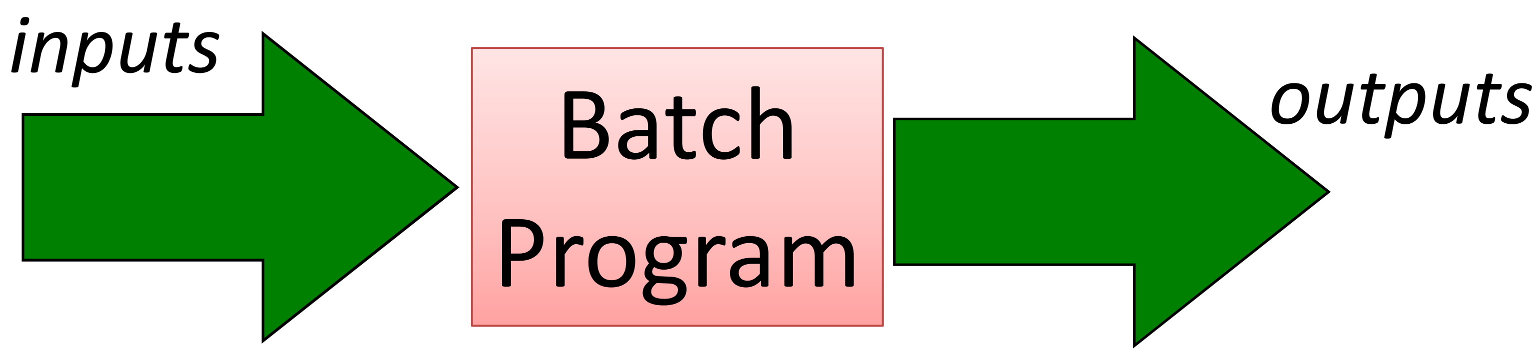
```
count      :: Char → String → Int
count x xs =
   length [x’ | x’ ← xs, x == x’]
```

For example:

```
> count ’s’ "Mississippi"
4
```

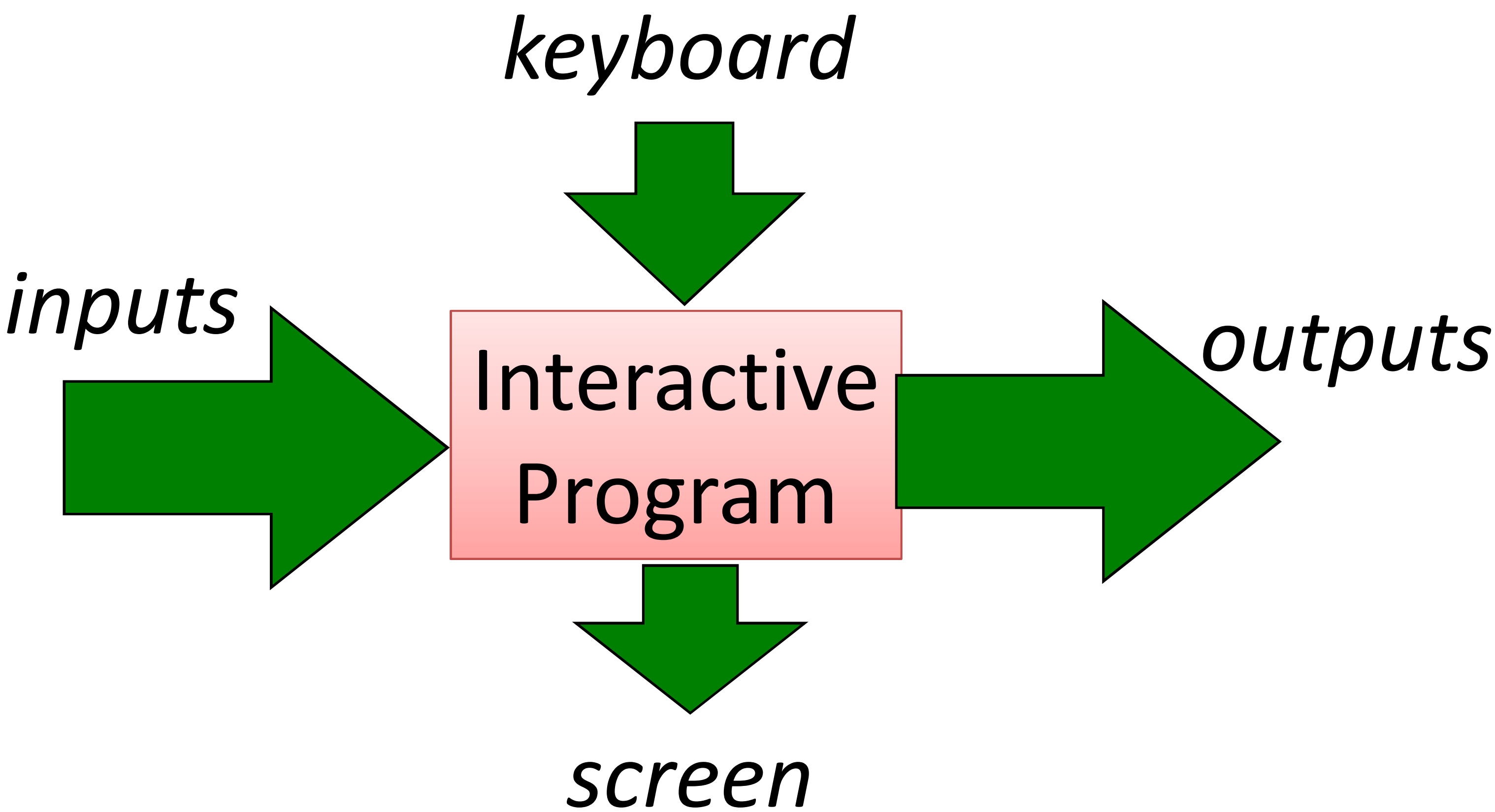
Haskell Batch Program

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.



Haskell Interactive Program

However, we would also like to use Haskell to write <u>interactive</u> programs that read from the keyboard and write to the screen, as they are running.



The Problem

Haskell programs are pure mathematical functions:

Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

Interactive programs have side effects.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure <u>actions</u> that may involve side effects.

```
IO a
```

The type of actions that return a value of type a.

For example : No side effect IO functions

`IO Char`

The type of actions that return a character.

`IO ()`

The type of purely side effecting actions that return <u>no</u> result value.

() is the type of tuples with no components.

Primitive Actions

The standard library provides a number of actions, including the following three primitives:

- The action getChar

```
getChar :: IO Char
```

  - reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

- The action putChar c

```
putChar :: Char → IO ()
```

  - writes the character c to the screen, and returns no result value:

- The action return v

```
return :: a → IO a
```

  - simply returns the value v, without performing any interaction:

Sequencing Actions

A sequence of actions can be combined as a single composite action using the keyword <u>do</u>.

For example:

```
getTwo :: IO (Char,Char)
getTwo  = do x ← getChar
             y ← getChar
             return (x,y)
```

Sequencing Actions

- Each action must begin in precisely the same column. That is, the layout rule applies;

- The values returned by intermediate actions are discarded by default, but if required can be named using the $\leftarrow$ operator;

- The value returned by the last action is the value returned by the sequence as a whole.

Other Library Actions

Reading a string from the keyboard:

```
getLine :: IO String
getLine  = do x ← getChar
              if x == '\n' then
                 return []
               else
                 do xs ← getLine
                    return (x:xs)
```

Other Library Actions

Writing a string to the screen:

```
putStr          :: String → IO ()
putStr []        = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Writing a string and moving to a new line:

```
putStrLn   :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

Example

```
> strlen

Enter a string: hello there
The string has 11 characters
```

Evaluating an action <u>executes</u> its side effects, with the final result value being discarded.

Defining Types: Data Declarations

A new type can be defined by specifying its set of values using a <u>data declaration</u>.

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

Bool is a new type, with two new values False and True.

Defining Types: Data Declarations

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer → Answer
flip Yes      = No
flip No       = Yes
flip Unknown = Unknown
```

Defining Types: Data Declarations

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square              :: Float → Shape
square n             = Rect n n

area                :: Shape → Float
area (Circle r)      = pi * r^2
area (Rect x y)      = x * y
```

Defining Types: Data Declarations

```
prelude>data Shape = Circle Float Float Float | Rectangle
                                    Float Float Float Float
Prelude> surface (Circle _ _ r) = pi * r ^ 2
Prelude> surface (Circle 10 20 10)
314.15927
Prelude>surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) *
(abs $ y2 - y1)
Prelude>
Prelude> surface (Rectangle 0 0 100 100)
10000.0
Prelude>surface (Circle 10 20 10)
Err….
```

Defining Types: Data Declarations

- if we add deriving (Show) at the end of a *data* declaration
- Haskell automagically makes that type part of the Show typeclass

```
Prelude>data Shape = Circle Float Float Float | Rectangle
Float Float Float Float deriving (Show)
Prelude>Circle 10 20 5
Circle 10.0 20.0 5.0
Prelude> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
Prelude>  map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6
.0,Circle 10.0 20.0 6.0]
```

If we want a list of concentric circles with different radii, we can do this.

Defining Types: Records

Prelude>data Person = Person String String Int Float String String de
riving (Show)
Prelude>let guy = Person "Buddy" "Finklestein" 43 184.2 "526-
2928" "Chocolate"
Prelude> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
Prelude>firstName (Person firstname _ _ _ _ _) = firstname
Prelude>lastName (Person _ lastname _ _ _ _) = lastname
Prelude>age (Person _ _ age _ _ _) = age
Prelude>height (Person _ _ _ height _ _) = height
Prelude>phoneNumber (Person _ _ _ _ number _) = number
Prelude> flavor (Person _ _ _ _ _ flavor) = flavor

Defining Types: Records

```
Prelude> data Person =
Person { firstName :: String,  lastName :: String,  age :: Int ,  height :: Flo
at ,  phoneNumber :: String , flavor :: String } deriving (Show)
Prelude>    :t flavor
flavor :: Person -> String
Prelude>data Car = Car String String Int deriving (Show)
Prelude>  Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
Prelude> data Car a b c = Car { company :: a, model :: b
                 , year :: c    } deriving (Show)
Prelude>tellCar (Car {company = c, model = m, year = y}) = "This " ++ c
++ " " ++ m ++ " was made in " ++ show y
Prelude> let stang = Car {company="Ford", model="Mustang", year=1967}
Prelude>tellCar stang
"This Ford Mustang was made in 1967"
```

Data Type vector Example

```
Prelude>  data Vector a = Vector a a a deriving (Show)
Prelude>
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)
Prelude> (Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)
Prelude>(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

```
Prelude> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
Prelude>Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
12 9 19
Prelude>Vector 3 9 7 `vectMult` 10
Vector 30 90 70
Prelude> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
Prelude> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

Data Type : True False Ordering

Prelude> **data Bool = False | True deriving (Ord)**

- Because the False value constructor is specified first and the True value constructor is specified after it, we can consider True as greater than False.

Prelude> True `compare` False

GT

Prelude> True > False

True

Data Type : May be and Just

Prelude> **data Maybe a = Just a | Nothing z**

That declaration defines a type, Maybe a, which is parameterized by a type variable a, which just means that you can use it with any type in place of a.

```
lend amount balance =
           let reserve = 100
            newBalance = balance - amount
          in if balance < reserve then Nothing
          else Just newBalance
```

Defining Types: Data Declarations

Similarly, data declarations themselves can also have parameters. For example, given

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

we can define:

```
return         :: a → Maybe a
return x        = Just x

(>>=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing >>= _ = Nothing
Just x  >>= f = f x
```

Data Type : May be and Just

The Nothing value constructor is specified before the Just value constructor

```
Prelude> Nothing < Just 100
True
Prelude> Nothing > Just (-49999)
False
Prelude> Just 3 `compare` Just 2
GT
Prelude>  Just 100 > Just 50
True
```

BST Creation in Haskell

```
data Tree a = Nil | Node (Tree a) a (Tree a) deriving Show
-- Checking Empty function
empty Nil = True
empty  _  = False
-- Insert an  element to  the Tree
insert Nil x = Node Nil x Nil
insert (Node t1 v t2) x
        | v == x = Node t1 v t2
        | v  < x = Node t1 v (insert t2 x)
        | v  > x = Node (insert t1 x) v t2
```

BST Creation in Haskell

```
--Contain : if element is present return true
contains Nil _ = False
contains (Node t1 v t2) x
        | x == v = True
        | x  < v = contains t1 x
        | x  > v = contains t2 x
-- Creation of Tree from list of number
ctree [] = Nil
ctree (h:t) = ctree2 (Node Nil h Nil) t
       where
             ctree2 tr [] = tr
             ctree2 tr (h:t) = ctree2 (insert tr h) t
-- Creation of Tree from list of number Example
ctree [1,2,4,5,8,7]
```

Advanced Data Types in Haskell (Program not tested in GHCI)

Recursive Types

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

Define Natural Number

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors

Zero :: Nat and Succ :: Nat $\longrightarrow$ Nat.

Recursive Types

- A value of type Nat is
  - either Zero, or of the form Succ n
  - where n :: Nat.
- That is, Nat contains the following infinite sequence of values:

```
Zero
```

```
Succ Zero
```

```
Succ (Succ Zero)
```

⋮

Recursive Types

- We can think of values of type Nat as <u>natural numbers</u>, where Zero represents 0, and Succ represents the successor function (1 +).
- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))  =  3
```

Recursive Types

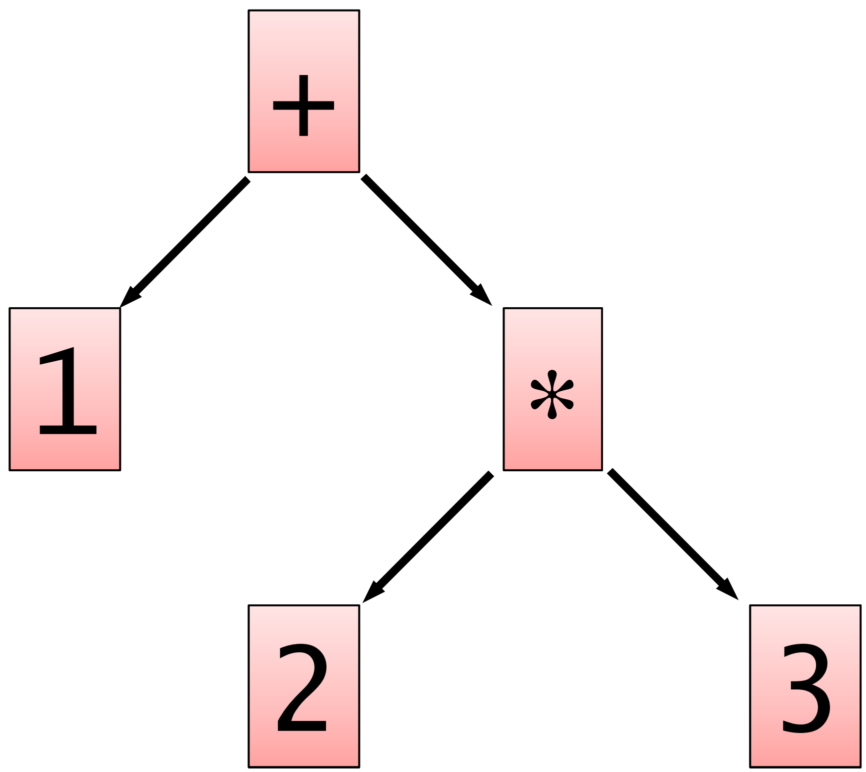
Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int              :: Nat → Int
nat2int Zero          = 0
nat2int (Succ n) = 1 + nat2int n

int2nat              :: Int → Nat
int2nat 0             = Zero
int2nat n      = Succ (int2nat (n-1))
```

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Arithmetic Expressions

Using recursion, a suitable new type to represent such expressions can be defined by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Arithmetic Expressions

Using recursion, it is now easy to define functions that process expressions. For example:

```
size            :: Expr → Int
size (Val n)    = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval            :: Expr → Int
eval (Val n)    = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Thanks