UNIT-III: Declarative Programming Paradigm: Functional Programming

Faculty In-charge

Mrinmoyee Mukherjee
Assistant Professor (IT Dept.)
email: mrinmoyeemukherjee@sfit.ac.in
Mob: 9324378409
Academic Year: 2023-24

DATA TYPES

- Bool
- Char
- Int
- Float
- Double
- List
- Tuple
- Function

- In Haskell all computations are done via the evaluation of expressions
- Examples of expressions include atomic values (built-in) such as
  - the integer 5,
  - the character 'a', and
  - the function \x -> x+1, as well as structured values such as
  - the list [1,2,3] and
  - the pair ('b',4).

Clear screen command CNTL+L
Exit GHCI command CNTL+D
Exit Prelude :n
https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-invocation.html

Types (set of Values)

- Bool
- Char
- Int (64 bit)
- Integer (Superset of Int)
- Float
- Double
- List
- Tuple
- Function

Examples

- :type True
- :type "hi"
- :type 5
- :type 5.34
- :type (True, False)

Type Class

- **EQ**
  - Type class is an interface which provides the functionality to test the equality of an expression.
- **Num and Fractional**
  - This type class is used for numeric operations. Types such as Int, Integer, Float, and Double come under this Type class.
- **Integral**
  - sub-class of the Num Type Class.
  - Int and Integer are the types under this Type class.
- **Floating**
  - sub-class of the Num Type Class.
  - Float and Double come under this type class.

DATA TYPES

Command Prompt - ghci

```
C:\Users\Mrinmoyee>ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 2
2
Prelude> :type 2
2 :: Num p => p
Prelude> :type 2.1
2.1 :: Fractional p => p
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
Prelude> :type "Hello"
"Hello" :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]
Prelude> :type [2.1,3.1,4.1]
[2.1,3.1,4.1] :: Fractional a => [a]
Prelude> _
```

Type here to search

Near record

DATA TYPES

Command Prompt - ghci

```
Prelude> :type ["hi","hello","nameste"]
["hi","hello","nameste"] :: [[Char]]
Prelude> :type (2,3,4)
(2,3,4) :: (Num a, Num b, Num c) => (a, b, c)
Prelude> :type (2,3,"Hi")
(2,3,"Hi") :: (Num a, Num b) => (a, b, [Char])
Prelude>
```

ARITHMETIC OPERATORS

Command Prompt - ghci

```
Prelude> 2+3
5
Prelude> 2.1+2.1
4.2
Prelude> 2.1+5
7.1
Prelude> :type 2.1+5
2.1+5 :: Fractional a => a
Prelude> x=2
Prelude> :type x
x :: Num p => p
Prelude> x=2.1+5
Prelude> :type x
x :: Fractional a => a
Prelude> True
True
Prelude> :type it
it :: Bool
Prelude>
```

LOGICAL OPERATORS

Command Prompt - ghci

```
Prelude> False&&False
False
Prelude> True&&False
False
Prelude> False&&True
False
Prelude> False||False
False
Prelude> False||True
True
Prelude> True||False
True
Prelude> True||True
True
Prelude> (True&&True)||(False&&False)
True
Prelude> not((True&&True)||(False&&False))
False
Prelude>
```

Type here to search

COMPARATIVE OPERATORS

Command Prompt - ghci

```
Prelude> 2==2
True
Prelude> 2==3
False
Prelude> not(2==3)
True
Prelude> not(not(2==3))&&True
False
Prelude> not(not(2==3))&&True+2

<interactive>:36:17: error:
    * No instance for (Num Bool) arising from a use of `+'
    * In the second argument of `(&&)', namely `True + 2'
      In the expression: not (not (2 == 3)) && True + 2
      In an equation for `it': it = not (not (2 == 3)) && True + 2
Prelude> _
```

Type here to search

29°C Haze

Command Prompt - ghci

```
Prelude> 2/=2
False
Prelude> 2/=3
True
Prelude> 2<3
True
Prelude> 2>3
False
Prelude> not(2>3)
True
Prelude> 2^3
8
Prelude> 2**3
8.0
Prelude> 2^4
16
Prelude> 2**4
16.0
Prelude>
```

Command Prompt - ghci

```
Prelude> "Hi"=="Hi"
True
Prelude> "Hi"=="hi"
False
Prelude> not("Hi"=="hi")&&True
True
Prelude> not(not("Hi"=="hi")&&True)||False
False
Prelude> "Hello world"=="Hello world"
True
Prelude>
```

LIST

- Much like shopping lists in the real world, lists in Haskell are very useful.
- It's the most used data structure
- In Haskell, lists are a **homogenous** data structure.
- It stores several elements of the same type.

> ❖ A list in Haskell can be written using square brackets with commas separating the list's individual values.
>
> ❖ Has only homogeneous values

- That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

```
Prelude> let list1=[1,2,3,4]
Prelude> print list1
[1,2,3,4]
Prelude> let list2=[1,2,3,'a','b']
```

```
Prelude> [1,2,3,4]++[5,6,7,8]
[1,2,3,4,5,6,7,8]
Prelude> "hello"++ " "++"world"
"hello world"
Prelude> ['w','o']++['o','t']
"woot"
Prelude>
```

- Internally, Haskell has to walk through the whole list on the left side of **++**
- That's not a problem when dealing with lists that aren't too big
- But putting something at the end of a list that's fifty million entries long is going to take a while
- However, putting something at the beginning of a list using the **:** operator (also called the cons operator) is instantaneous.

```
Prelude> 'A':" Small Cat"
"A Small Cat"
Prelude> 5:[1,2,3,4]
[5,1,2,3,4]
Prelude>
```

LIST ....Continues....

```
Prelude> []
[]
Prelude> [[]]
[[]]
Prelude> [[[]]]
[[[]]]
Prelude>
```

1. An empty list
2. List of one empty list
3. List of two empty list

- If you want to get an element out of a list by index, use **!!**. The indices start at 0
- But if you try to get the sixth element from a list that only has four elements, you'll get an error so be careful!

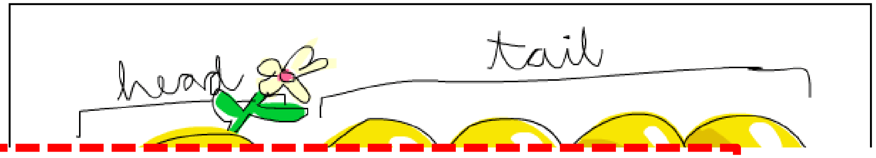
```
[[[]]]
Prelude> "Steve Buscemi" !! 6
'B'
```

LIST ....Continues....

Lists can also contain lists. They can also contain lists that contain lists that contain lists …

```
[[[]]]
Prelude> "Steve Buscemi" !! 6
'B'
Prelude> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
Prelude> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b !! 2
[1,2,2,3,4]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude>
```

LIST ....Continues....



`head` takes a list and returns its head. The head of a list is basically its first element.

```
ghci> head [5,4,3,2,1]
5
```

`tail` takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

`last` takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

`init` takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

LIST ....Continues....

`length` takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

`null` checks if a list is empty. If it is, it returns `True`, otherwise it returns `False`. Use this function instead of `xs == []` (if you have a list called `xs`)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` takes number and a list. It extracts that many elements from the beginning of the list. Watch.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
```

POST IAT-1 EXAM

LIST ....Continues....

drop works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

maximum takes a list of stuff that can be put in some kind of order and returns the biggest element.

minimum returns the smallest.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

sum takes a list of numbers and returns their sum.

product takes a list of numbers and returns their product.

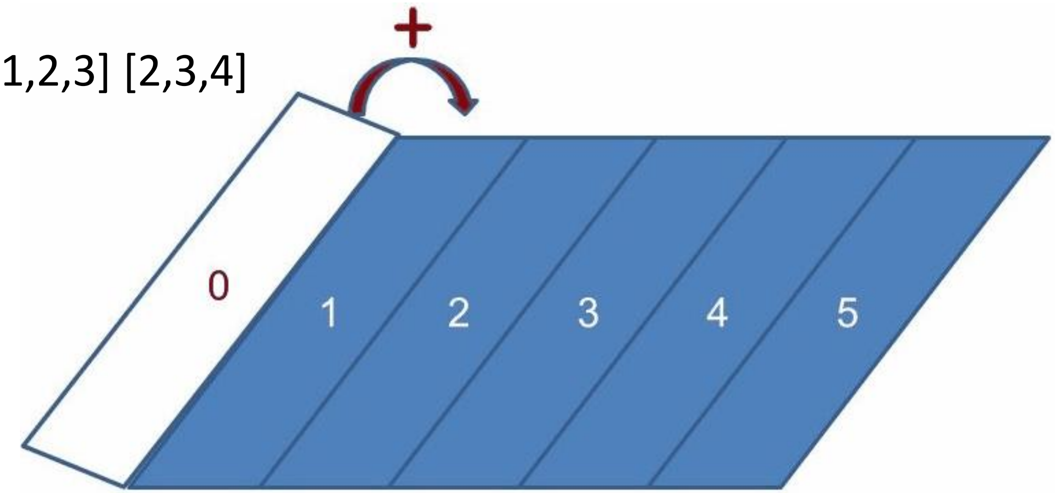
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

LIST ....Continues....

`elem` takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

zipWith (*) [1,2,3] [2,3,4]



LIST ....Continues....(Ranges)

Command Prompt - ghci

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
Prelude> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Prelude> ['a'..'p']
"abcdefghijklmnop"
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [3,6..30]
[3,6,9,12,15,18,21,24,27,30]
Prelude> [5,10..50]
[5,10,15,20,25,30,35,40,45,50]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
Prelude> [100,90..1]
[100,90,80,70,60,50,40,30,20,10]
Prelude> [0.1,0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
Prelude> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
Prelude> take 12 (cycle "LOL ")
"LOL LOL LOL "
Prelude> take 12 (cycle "LOL")
"LOLLOLLOLLOL"
Prelude> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
Prelude> replicate 3 10
[10,10,10]
Prelude>
```

LIST ….Comprehension (Ranges)

- A basic comprehension for a set that contains the first ten even natural numbers is

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$$

- The part before the pipe is called the output function
- **x** is the variable
- **N** is the input set and **x <= 10** is the predicate.
- That means that the set contains the doubles of all natural numbers that satisfy the predicate.

```
Prelude> [x*2|x<-[1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x*2|x<-[1..10],x*2>=12]
[12,14,16,18,20]
Prelude> [x*3|x<-[1..10],x*2>=12]
[18,21,24,27,30]
Prelude> [x*3|x<-[1..10],x*2>=9]
[15,18,21,24,27,30]
Prelude>
```

LIST ….Comprehension (Ranges)

- How about if we wanted all numbers from 50 to 100 whose remainder when divided with the number 7 is 3?

- If we wanted all numbers from 10 to 20 that are not 13, 15 or 19

- If we have two lists, **[2,5,10]** and **[8,10,11]** and we want to get the products of all the possible combinations between numbers in those lists

```
Prelude> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
Prelude> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

```
Prelude> [x*1|x<-[1..20]]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [x*2|x<-[1..20]]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40]
Prelude> [x*2|x<-[1..20], x/=10]
[2,4,6,8,10,12,14,16,18,22,24,26,28,30,32,34,36,38,40]
Prelude> [x*2|x<-[1..20], x/=13]
[2,4,6,8,10,12,14,16,18,20,22,24,28,30,32,34,36,38,40]
Prelude> [x^2|x<-[1..20]]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361,400]
Prelude> [2^x|x<-[1..20]]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,524288,1048576]
```

TUPLES…

- Are used when you know exactly how many values you want to combine
- Its type depends on how many components it has and the types of the components.
- They are denoted with parentheses and their components are separated by commas.

- Unlike a list, a tuple can contain a combination of several types.

(represent someone's name and age in Haskell),**("Christopher", "Walken", 55)**

- Use tuples when you know in advance how many components some piece of data should have. There are singleton lists, there's no such thing as a singleton tuple.

- Like lists, tuples can be compared with each other if their components can be compared

- Only you can't compare two tuples of different sizes, whereas you can compare two lists of different sizes.

```
Prelude> fst (8,9)
8
Prelude> fst (8,"wow")
8
Prelude> snd (8,9)
9
Prelude> snd (8,"wow")
"wow"
```

Tuple

- Like Lists, Tuples are used to store the values or group the values together.
- List in Haskell store same type of data inside it.
- Tuple can store different type of data inside it.

- Ex. of tuple

("String", Boolean, numbers)

- Functions used for tuples

fst (1,2,"a")

snd(1,"a",2)

Ex.

```
main = do
  print("Tuple in Haskell")
  let tuple1 =  (100, 200)
  let tuple2  =  (10.5, "hello")
  let tuple3  =  ("hello", "world")
  let tuple4  =  ("i am first", "i am second")
  let tuple5  =  ("i am string", "i am second
                   string")
  let tuple6 =  (260, 270)
  print(fst(tuple1))
  print(snd(tuple2))
```

Decision Making (If..else)

- if<Condition> then <True-Value> else <False-Value>

```
main = do
  let var = 23
  if var == 0
    then putStrLn "Number is zero"
  else if var `rem` 2 == 0
    then putStrLn "Number is Even"
  else putStrLn "Number is Odd"
```

Evaluation Order

- **Expressions:**
- Haskell has no statements, only expression.
- Pure functional programming languages don't have any statements—no assignments, no jumps
- All computations are performed by evaluating expressions

- Ex:

3+4 --> 7

3+4*5 is equivalent to 3 + (4*5) --> 23

(3+4)*5 is equivalent to 7*5 --> 35

```
Prelude> sum ([3,7])
10
Prelude> sum ([3,7]++[5])
15
Prelude> 2
2
Prelude> 3+4
7
Prelude> 3+4*5
23
Prelude> (3+4)*5
35
```

Built-in Functions

- succ 6
- succ 7 * 8
- succ (7 * 8)
- min 4 9
- max 4 9
- div 4 3
- div 3 4
- mod 3 2
- 4/3
- reverse “hello”
- return True
- x <- return 35
- print x
- let y = 35
- print y
- putStrLn "hello“

Command Prompt - ghci

```
Prelude>
Prelude> gcd 12 3
3
Prelude> lcm 12 3
12
Prelude> even 4
True
Prelude> odd 4
False
Prelude> succ 6
7
Prelude> pred 6
5
Prelude> pred 0
-1
```

Command Prompt - ghci

```
               (Foldable t
               t3
Prelude> succ 6
7
Prelude> succ 7*8
64
Prelude> succ (7*8)
57
Prelude> min 4 9
4
Prelude> max 4 9
9
Prelude> div 4 3
1
Prelude> 4/3
1.3333333333333333
Prelude> mod 4 3
1
```

Command Prompt - ghci

```
Prelude> div 10 3
3
Prelude> mod 10 3
1
Prelude> 10/3
3.3333333333333335
Prelude> reverse "HELLO"
"OLLEH"
Prelude> x<-return 35
Prelude> print x
35
Prelude> let y=45
Prelude> print y
45
```

THANK YOU