

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: 2021-22



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  
True
```

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

```
Prelude>
```

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> _
```



```
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>
```

```
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.
- 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.

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

❖ Has only homogeneous values

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

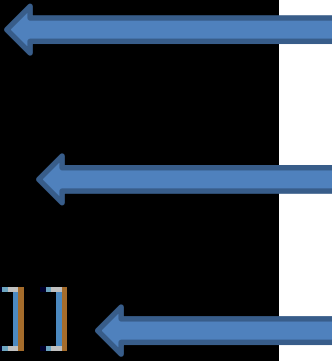
- Internally, Haskell has to walk through the whole list on the left side of ++
- That's not a problem when dealing with lists that aren't too big
- But putting something at the end of a list that's fifty million entries long is going to take a while
- However, putting something at the beginning of a 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]
```



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.

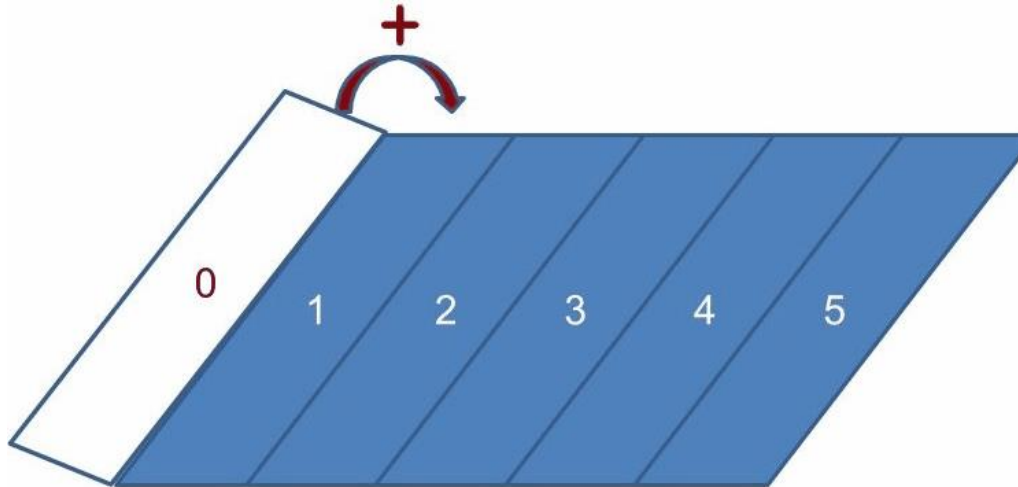
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
```


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]



What if we want a list of all numbers between 1 and 200?



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"
Prelude> zip [1,2,3,4,5] [6,6,6,6,6]
[(1,6),(2,6),(3,6),(4,6),(5,6)]
Prelude> zip [1,2,3,4,5] [6,6,6,6,6]
[(1,6),(2,6),(3,6),(4,6)]
Prelude> zip [1,2,3,4] [6,6,6,6,6]
[(1,6),(2,6),(3,6),(4,6)]
Prelude> zip [1..5] ["a","b","c","d","e"]
[(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")]
Prelude> zip [a..f] ["a","b","c","d","e"]

<interactive>:25:6: error: Variable not in scope: a

<interactive>:25:9: error: Variable not in scope: f
Prelude> zip ['a'..'f'] ["a","b","c","d","e"]
[('a',"a"),('b',"b"),('c',"c"),('d',"d"),('e',"e")]
Prelude>

```

