Functional Programming Concepts

- *Functional programming* defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state
- Functional Programming Languages
  - Miranda,
  - Haskell,
  - pH,
  - Sisal, and
  - Single Assignment C are purely functional.
  - Erlang is nearly functional
  - Most others include imperative features.

Functional Programming Concepts

Functional languages provide a number of features that are often missing in imperative languages:

- First-class function values and higher-order functions: A higher order function takes a function as an argument, or returns a function as a result.
- Extensive polymorphism
- List types and operators
- Structured(arrays) function returns
- Constructors (aggregates) for structured objects(lambda expression)
- Garbage collection

Several of features (recursion, structured function returns, constructors, garbage collection) can be found in some but not all imperative languages.

Haskell Programming

Haskell Programming

- Haskell is a widely used purely functional language.
- Functional programming is based on mathematical functions.
- Besides Haskell, some of the other popular Functional languages :
  - Lisp, Python, Erlang, Racket, F#, Clojure, etc.

Core idea:

- Haskell, de-emphasizes the code that modifies data.
- It focuses on functions that take immutable values as input and produce new values as output. Given the same inputs, these functions always return the same results.

Characteristics

- **Functional Language** − In conventional programing language, we instruct the compiler a series of tasks which is nothing but telling the computer "what to do" and "how to do?" But in Haskell we tell our computer "what it is?“
- **Laziness** − Haskell is a lazy language. By lazy, we mean that Haskell won't evaluate any expression without any reason. When the evaluation engine finds that an expression needs to be evaluated, then it creates a data structure to collect all the required information for that specific evaluation and a pointer to that data structure.

Characteristics

- Example: Let's say we want to find the k least-valued elements of an unsorted list.
- Traditional approach - sort the list and take the first k elements
  - Expensive approach
- Haskell - laziness ensures that the list will only be sorted enough to find the k minimal elements.

```
minima k xs = take k (sort xs)
```

- An important aspect of Haskell's power lies in the compactness of the code we write.

Characteristics

- Modularity − A Haskell application is nothing but a series of functions. We can say that a Haskell application is a collection of numerous small Haskell applications.
- Statically Typed −
  - Conventional programing - define a series of variables along with their type. , Haskell - strictly typed language. Haskell compiler is intelligent enough to figure out the type of the variable declared,
  - No need to mention the type of the variable used.
- Maintainability − Haskell applications are modular and hence, it is very easy and cost-effective to maintain them.

Installation

- Go to
- https://www.haskell.org/downloads/
- There are three widely used ways to install the Haskell toolchain on supported platforms. These are:
- Minimal installers: Just GHC (the compiler), and build tools (primarily Cabal and Stack) are installed globally on your system, using your system's package manager.
- Stack: Installs the stack command globally: a project-centric build tool to automatically download and manage Haskell dependencies on a project-by-project basis.
- Haskell Platform: Installs GHC, Cabal, and some other tools, along with a starter set of libraries in a global location on your system.

Installation

Stack

What it is

Stack is a cross-platform build tool for Haskell that handles management of the toolchain (including the GHC compiler and MSYS2 on Windows), building and registering libraries, and more.

What you get

- Once downloaded, it has the capacity to download and install GHC and other core tools.
- Project development is isolated within sandboxes, including automatic download of the right version of GHC for a given project.
- It manages all Haskell-related dependencies, ensuring reproducible builds.
- It fetches from a curated repository of over a thousand packages by default, known to be mutually compatible.
- It can optionally use Docker to produce standalone deployments.

How to get it

The install and upgrade page describes how to download Stack on various platforms, although the main three are repeated here:

- Ubuntu Linux
- OS X
- Windows

Instructions for other Linux distributions, including Debian, Fedora, Red Hat, Nix OS, and Arch Linux, are also available.

Installation

- Click on Stack and then on windows
- Click on Installer
- There is an option for manual download too but don't go for it.
- Download
- Run it
- Select defaults
- Go to command prompt: type **stack**
- Now type, **stack setup**
- Open command prompt again, type **stack ghci**
- This opens interactive interpreter, and we are good to go

Basic Data Model

Numbers:

- Haskell is intelligent enough to decode some number as a number.
- Therefore, one need not mention its type externally
- Go to prelude command prompt and just run "2+2" and hit enter.

Basic Data Model

1. prelude> 2 + 15
2. 17
3. prelude> 49 * 100
4. 4900
5. prelude> 1892 - 1472
6. 420
7. prelude> 5 / 2
8. 2.5
9. prelude>

Basic Data Model

1. prelude> (50 * 100) - 4999
2. 1
3. prelude> 50 * 100 - 4999
4. 1
5. prelude> 50 * (100 - 4999)
6. -244950

Basic Data Model

Characters:

- Haskell can intelligently identify a character given in as an input to it.
- Go to Haskell command prompt and type any character with double or single quotation.

Basic Data Model

String::

- A **string** is nothing but a collection of characters.
- Haskell follows the conventional style of representing a string with double quotation

Basic Data Model

Boolean:

- We need not mention that "True" and "False" are the Boolean values.
- Haskell itself can decode it and do the respective operations.

Basic Data Model

1. prelude> True && False
2. False
3. prelude> True && True
4. True
5. prelude> False || True
6. True
7. prelude> not False
8. True
9. prelude> not (True && True)
10. False

List

- List is a collection of same data type separated by comma.
- 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
  - let lostNumbers = [4,8,15,16,23,42]
- Lists are denoted by square brackets and the values in the lists are separated by commas.
  - **[1,2,'a',3,'b','c',4]** – **error**
- A common task is putting two lists together. This is done by using the **++** operator.

List: ++

- A common task is putting two lists together(append).
- This is done by using the **++** operator.
- prelude> [1,2,3,4] ++ [9,10,11,12]

  [1,2,3,4,9,10,11,12]
- prelude> "hello" ++ " " ++ "world"

  "hello world"
- prelude> ['w','o'] ++ ['o','t']

  "woot"

List:Cons Operator

- When we put together two lists (even if you append a singleton list to a list, for instance: [1,2,3] ++ [4]), internally, Haskell has to walk through the whole list on the left side of ++.
- Putting something at the end of a list that contain fifty million entries is time consuming
- Instead put something at the beginning of a list using the **:** operator (called the **cons** operator

List:Cons Operator

- prelude> 'A':" SMALL CAT"

  "A SMALL CAT"
- prelude> 5:[1,2,3,4,5]

  [5,1,2,3,4,5]

List:Cons Operator

- : takes a **number** and a **list of numbers** or

  a **character** and a **list of characters**,
- Whereas ++ takes two lists.
- Even if we are adding an element to the end of a list with ++, we have to surround it with square brackets so it becomes a list.

List:!!

- If we want to get an element out of a list by index,we use **!!.**
- The indices start at 0.
- Prelude> "Steve Jobs" !! 6

  'J'
- Prelude> [1,2,3,4,5,6] !! 3

  4

List: List within list

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

List: Basic functions on lists

- **head** takes a list and returns its head.
- The head of a list is basically its first element.
- prelude> 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.
- prelude> tail [5,4,3,2,1]

  [4,3,2,1]

List: Basic functions on lists

- last takes a list and returns its last element.
- prelude> last [5,4,3,2,1]

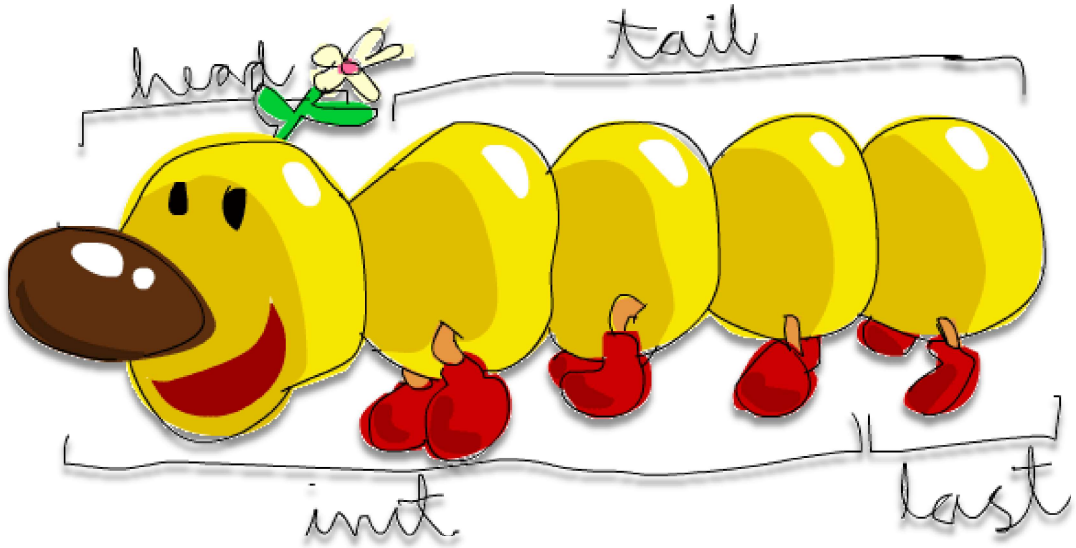
  1

- init takes a list and returns everything except its last element.
- prelude> init [5,4,3,2,1]

  [5,4,3,2]

List

If we think of a list as a monster



List: Basic functions on lists

- **length** takes a list and returns its length.
- prelude> length [5,4,3,2,1]

  5

- **null** checks if a list is empty. If it is, it returns True, otherwise it returns False.
- prelude> null [1,2,3]

  False
- prelude> null []

  True

List: Basic functions on lists

- **reverse** reverses a list.
- prelude> 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.
- prelude> take 3 [5,4,3,2,1]

  [5,4,3]
- prelude> take 1 [3,9,3]

  [3]
- prelude> take 5 [1,2]

  [1,2]
- prelude> take 0 [6,6,6]

  []

List: Basic functions on lists

- **drop** drops the number of elements from the beginning of a list.

- prelude> drop 3 [8,4,2,1,5,6]

  [1,5,6]
- prelude> drop 0 [1,2,3,4]

  [1,2,3,4]
- prelude> drop 100 [1,2,3,4]

  []

List: Basic functions on lists

- **maximum** takes a list of stuff that can be put in some kind of order and returns the biggest element.
- **minimum** returns the smallest.
- prelude> minimum [8,4,2,1,5,6]

  1
- prelude> 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.

List:Range

- Range:
- Prelude> [1..20]

  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
- Prelude> ['a'..'h']

  "abcdefgh"
- Prelude> ['A'..'Z']

  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

List:Range

- if we want all even numbers between 1 and 20
- [2,4..20]
- every third number between 1 and 20
- [3,6..20]
- List from 20 to 1
- [20,19..1]

List:cycle

- **cycle** takes a list and cycles it into an infinite list.
- 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 "

List:Repeat

- **repeat** takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

- ghci> take 10 (repeat 5)
- [5,5,5,5,5,5,5,5,5,5]

List Comprehension

- List comprehension is the process of generating a list using mathematical expression.
- format - [output | range ,condition]
- Prelude> [x*2| x<-[1..10]]

  [2,4,6,8,10,12,14,16,18,20]
- Prelude> [x*2| x<-[1..5]]

  [2,4,6,8,10]
- Prelude> [x| x<-[1..5]]

  [1,2,3,4,5]

List Comprehension

- List comprehension is the process of generating a list using mathematical expression.
- format - [output | range ,condition]
- Prelude> [x*2| x<-[1..10]]

  [2,4,6,8,10,12,14,16,18,20]
- Prelude> [x*2| x<-[1..5]]

  [2,4,6,8,10]
- Prelude> [x| x<-[1..5]]

  [1,2,3,4,5]

List Comprehension

- We want elements that when doubled are greater than equal to 12

- Prelude> [x*2 | x <- [1..10], x*2 >= 12]

  [12,14,16,18,20]

List Comprehension

- All numbers from 50 to 100 whose remainder, when divided with the number 7, is 3

- Prelude> [x | x <- [50..100], x `mod` 7 == 3]

  [52,59,66,73,80,87,94]

List Comprehension

- We can include several predicates.
- If we wanted all numbers from 10 to 20 that are not 13, 15 or 19
- Prelude> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]

  [10,11,12,14,16,17,18,20]

List Comprehension

- We can also draw from several lists.

- Prelude> [x*y | x <- [2,5,10], y <- [8,10,11]]
  [16,20,22,40,50,55,80,100,110]

Zip function

- Zip- It takes two lists and then zips them together into one list by joining the matching elements into pairs.
- Prelude> zip [1,2,3,4,5] [5,5,5,5,5]
- [(1,5),(2,5),(3,5),(4,5),(5,5)]
- It pairs up the elements and produces a new list.
- The first element goes with the first, the second with the second, etc.

Zip function

- If the length do not match, the longer list simply gets cut off to match the length of the shorter one
- Prelude> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
- [(5,"im"),(3,"a"),(2,"turtle")]
- Prelude> zip [1..] ["apple", "orange", "cherry", "mango"]
- [(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]

Tuples

- Haskell provides another way to declare multiple values in a single data type.
- Tuple is heterogeneous in nature, because a Tuple may contain different type of data inside it.
- Tuples are represented by single parenthesis.
- Prelude> (1,2.5,'a',"hello",True)

  (1,2.5,'a',"hello",True)

Tuples

- A tuple can be used as a dictionary
- [(“one”,1), (“two”,2), (“three”,3)]
- We can look up the value of type string and it will return the number in it.
- Let dict= [(“one”,1), (“two”,2), (“three”,3)]
- lookup “one” dict
- lookup “two” dict

Tuples

- We can create list of tuples
- We can also create tuples of tuples
- Prelude> let a= [(1,2,3),(4,5,6)]
- Prelude> a
- [(1,2,3),(4,5,6)]
- Prelude> let a= [(1,2,3),("two","three")]
- Error
- let a= ((1,2,3),("four","five"))
- Prelude> a
- ((1,2,3),("four","five"))

Tuples

- A tuple of size two (also called a pair) is its own type
- Two useful functions that operate on pairs:
- **fst** takes a pair and returns its first component.
- Prelude> let a= (1,"one")
- Prelude> fst a
- 1
- **snd** takes a pair and returns its second component.
- Prelude> snd a
- "one"
- Prelude>

Comments

- Single line comment

  -- ^

- Multi line comment

  {-|

  -}

Functions

- Write the function in notepad
- Save this as **something.hs**
- Go to command prompt and to the respective folder
- Run GHCI from there
- Once inside GHCI, type **:l something**
- Call the function
  - Function name parameter names

Functions

- Define the function
- doubleMe x = x + x
- doubleUs x y = x*2 + y*2
- doubleUs x y = doubleMe x + doubleMe y
- doubleSmallNumber x = if x > 100
  then x
  else x*2

Main method

- Main:
- main = do
  statements

Print statement

- If you have a String that you want to print to the screen, you should use putStrLn. If you have something other than a String that you want to print, you should use print.

Function Composition

- **Function Composition** is the process of using the output of one function as an input of another function.
- In mathematics, **composition** is denoted by **f{g(x)}**
  - where **g()** is a function and its output is used as an input to another function, that is, **f()**.
- The output type of one function should match with the input type of the second function.
- We use the dot operator (.) to implement function composition in Haskell.

Function Composition

```
eveno x = if x `rem` 2 == 0
   then True
else False
noto x = if x == True
   then "This is an even Number"
else "This is an ODD number"

main = do
   putStrLn "Example of Haskell Function composition"
   print ((noto.eveno)(17))
```

Function Composition

- Here, in the **main** function, we are calling two functions, **noto** and **eveno**, simultaneously.
- The compiler will first call the function **"eveno()"** with **16** as an argument.
- Thereafter, the compiler will use the output of the **eveno** method as an input of **noto()** method.

Modules

- Haskell can be considered as a collection of **modules**
- A Haskell module is a collection of related functions, types and typeclasses.

Advantages.

- If a module is generic enough, the functions it exports can be used in a multitude of different programs
- More manageable code
- Reuse

Modules

- The syntax for importing modules in a Haskell script is

import <module name>.

import Data.List

- This must be done before defining any functions, so imports are usually done at the top of the file.
- When you do **import Data.List**, all the functions that **Data.List** exports become available

Data.List

- **intersperse** takes an element and a list and then puts that element in between each pair of elements in the list.
- Prelude> import Data.List
- Prelude Data.List> intersperse '.' "MONKEY"

  "M.O.N.K.E.Y“

- Prelude Data.List> intersperse 0 [1,2,3,4,5,6]

  [1,0,2,0,3,0,4,0,5,0,6]

Data.List

- **intercalate** takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.
- Prelude Data.List> intercalate " " ["hey","there","guys"]

  "hey there guys"
- Prelude Data.List> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]

  [1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]

Data.List

- **transpose** transposes a list of lists.
- Prelude Data.List> transpose [[1,2,3],[4,5,6]]

  [[1,4],[2,5],[3,6]]

Data.List

- **concat** flattens a list of lists into just a list of elements
- Prelude Data.List> concat [[3,4,5],[2,3,4],[2,1,1]]

  [3,4,5,2,3,4,2,1,1]
- Prelude Data.List> concat ["foo","bar","car"]

  "foobarcar"

Data.List

- **any** and **all** take a predicate and then check if any or all the elements in a list satisfy the predicate, respectively.
- Prelude Data.List> any (==4) [2,3,5,6,1,4]

  True
- Prelude Data.List> all (>4) [6,9,10]

  True

Data.List

- **sort** simply sorts a list.
- Prelude Data.List> sort [8,5,3,2,1,6,4,2]
  [1,2,2,3,4,5,6,8]
- Prelude Data.List> sort "This will be sorted soon"
  " Tbdeehiillnooorssstw"
- Prelude Data.List> sort "Z A h b"
  " AZbh"
- Prelude Data.List> sort "ZAhb"
  "AZbh"
- Prelude Data.List> sort ["Ayush", "Zainab", "Manu"]
  ["Ayush","Manu","Zainab"]

Data.List

- **find** takes a list and a predicate and returns the first element that satisfies the predicate.
- Prelude Data.List> find (>4) [1,2,3,4,5,6]

  Just 5
- Prelude Data.List> find (>9) [1,2,3,4,5,6]

  Nothing

Data.List

- **elemIndex** returns the index of the element we're looking for. If that element isn't in our list, it returns a Nothing.
- Prelude Data.List> 4 `elemIndex` [1,2,3,4,5,6]

  Just 3

Data.List

- **lines** is a useful function when dealing with files or input from somewhere. It takes a string and returns every line of that string in a separate list.
- Prelude Data.List> lines "first line\nsecond line\nthird line"

  ["first line","second line","third line"]

Data.List

- **words** and **unwords** are for splitting a line of text into words or joining a list of words into a text.
- Prelude Data.List> words "hey these are the words in this sentence"

  ["hey","these","are","the","words","in","this","sentence"]
- Prelude Data.List> unwords ["hey","there","mate"]

  "hey there mate"

Data.List

- **nub** takes a list and weeds out the duplicate elements
- Prelude Data.List> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]

  [1,2,3,4]

Data.List

- **union** returns the union of two lists. It goes over every element in the second list and appends it to the first one if it isn't already in yet. Duplicates are removed from the second list
- Prelude Data.List> [1] `union` [2]

  [1,2]
- Prelude Data.List> "hey" `union` "how"

  "heyow"

Data.List

- **intersect** works like set intersection. It returns only the elements that are found in both lists.
- Prelude Data.List> [1..7] `intersect` [5..10]

  [5,6,7]
- Prelude Data.List> [2,3] `intersect` [3,5]

  [3]
- Prelude Data.List> "how" `intersect` "hey"

  "h"

Data.List

- To unimport a module
- :module -Data.List

Data.Char Module

- isLower checks whether a character is lower-cased
- isUpper checks whether a character is upper-cased
- isAlpha checks whether a character is a letter.
- isAlphaNum checks whether a character is a letter or a number
- isDigit checks whether a character is a digit.
- isLetter checks whether a character is a letter.

Data.Char

- Prelude Data.Char> isAlpha 's'
- True
- Prelude Data.Char> isAlpha '2'
- False
- Prelude Data.Char> isAlpha '&'
- False
- Prelude Data.Char> isLetter 'u'
- True
- Prelude Data.Char> isLetter '8'
- False

Data.Char

- Prelude Data.Char> isLower 'd'
- True
- Prelude Data.Char> isLower 'A'
- False
- Prelude Data.Char> isDigit '6'
- True
- Prelude Data.Char> isDigit 'y'
- False

Data.Char

- Problem: A program that takes a username and the username can only be comprised of alphanumeric characters.
- Solution: We can use the Data.List function all in combination with the Data.Char predicates to determine if the username is alright.
- Prelude Data.Char> all isAlphaNum "swati222"
- True
- Prelude Data.Char> all isAlphaNum "swati222!"
- False

Data.Map

- An efficient implementation of maps from keys to values (dictionaries).
- **Map** is an unsorted value-added pair type data type.
- Association lists (also called dictionaries) are lists that are used to store key-value pairs
- E.g. we might use an association list to store phone numbers, where phone numbers would be the values and people's names would be the keys
- Because **Data.Map** exports functions that clash with the **Prelude** and **Data.List**, we do a qualified import.
- **import qualified Data.Map as Map**

Data.Map

- The **fromList** function takes an association list and returns a map with the same associations
- Prelude Data.Map Map> jobs= fromList[("Ajay", "TCS"), ("Divya", "Infosys"), ("Insha", "Wipro")]
- **lookup**: Lookup the value at a key in the map
- Prelude Data.Map Map> Map.lookup("Ajay") jobs
- Just "TCS"

Data.Map

- **size**: The number of elements in the map.
- Prelude Data.Map Map> Map.size jobs
- 3
- **member**: Is the key a member of the map?
- Prelude Data.Map Map> member "Ajay" jobs
- True

Data.Map

- **elems**: Return all elements of the map in the ascending order of their keys
- Prelude Data.Map Map> elems jobs
- ["TCS","Infosys","Wipro"]
- **keys**: Return all keys of the map in ascending order.
- Prelude Data.Map Map> keys jobs
- ["Ajay","Divya","Insha"]

Data.Map

- **insert**: Insert a new key and value in the map.
- Prelude Data.Map Map> insert "Devansh" "AWS" jobs
- fromList [("Ajay","TCS"),("Devansh","AWS"),("Divya","Infosys"),("Insha","Wipro")]
- **delete**: Delete a key and its value from the map.
- Prelude Data.Map Map> delete "Ajay" jobs
- fromList [("Divya","Infosys"),("Insha","Wipro")]

Data.Map

- **singleton** : A map with a single element.
- Prelude Data.Map Map> b= singleton 1 'a'
- Prelude Data.Map Map> b
- **filter**: Filter all values that satisfy the predicate
- Prelude Data.Map Map> Map.filter (>'a') b
- fromList []

Data.Map

- **split**: the expression (split k map) is a pair (map1,map2) where the keys in map1 are smaller than k and the keys in map2 are larger than k. Any key equal to k is found either in map1 or map2
- Prelude Data.Map Map> split 2 (fromList [(5,"a"), (3,"b")])
- (fromList [],fromList [(3,"b"),(5,"a")])
- **elemAt**: Retrieve an element by index
- Prelude Data.Map Map> elemAt 1 jobs
- ("Divya","Infosys")

Data.Map

- findMin & findMax:
- Prelude Data.Map Map> findMin jobs
- ("Ajay","TCS")
- Prelude Data.Map Map> findMax jobs
- ("Insha","Wipro")

Data.Set

- The Data.Set module offers sets, like sets from mathematics.
- All the elements in a set are unique and ordered
- Because the names in **Data.Set** clash with a lot of **Prelude** and **Data.List** names, we do a qualified import.
- import qualified Data.Set as Set

Data.Set

- The **fromList** function takes a list and converts it into a set.
- Prelude Set> let set1=Set.fromList "This is a set"
- Prelude Set> let set2=Set.fromList "It will rain today"
- Prelude Set> set1
- fromList " Taehist"
- Prelude Set> set2
- fromList " ladilnortwy“
- The items are ordered, and each element is unique.

Data.Set

- **intersection** function gives the elements they both share.
- Prelude Set> let set1=Set.fromList "This is a set"
- Prelude Set> let set2=Set.fromList "It will rain today“
- Prelude Set> Set.intersection set1 set2
- fromList " ait“
- **union** combines unique elements of both the sets
- Prelude Set> Set.union set1 set2
- fromList " ITadehilnorstwy"

Data.Set

- **difference** function is used to see which letters are in the first set but aren't in the second one and vice versa
- Prelude Set> set1
- fromList " Taehist"
- Prelude Set> set2
- fromList " ladilnortwy"
- Prelude Set> Set.difference set1 set2
- fromList "Tehs"
- Prelude Set> Set.difference set2 set1
- fromList "ldlnorwy"

Data.Set

- **Insert**:
- Prelude Set> Set.insert '4' set1
- fromList " 4Taehist"
- Prelude Set> Set.insert 'k' set1
- fromList " Taehikst"

Data.Set

- **delete**:
- Prelude Set> set2
- fromList " ladilnortwy"
- Prelude Set> Set.delete 'n' set2
- fromList " ladilortwy"

Recursion

- Recursion is a way of defining functions in which the function is applied inside its own definition.

  Or

- In recursion, a function calls itself.
- Definitions in mathematics are often given recursively.
- For instance, the Fibonacci sequence is defined recursively.
- When we write recursive function, we must have an **edge condition** to force the function to return without the recursive call being executed or else function will never return

Sum of Natural numbers

- natSum :: Int -> Int
- natSum 0 = 0
- natSum n= n + natSum (n-1)

Factorial

- fact:: Int -> Int
- fact 0 = 0
- fact 1 =1
- fact n = n * fact (n-1)

Fibonacci term

- import Prelude
- fib :: Int -> Int
- fib 0 =0
- fib 1=1
- fib n = fib (n-1) + fib (n-2)

Higher order functions

- Haskell functions can take functions as parameters and return functions as return values.
- A function that does either of these is called a higher order function.

Higher order functions

- applyTwice :: (a -> a) -> a -> a
- applyTwice f x = f (f x)

- Parantheses indicate that the first parameter is a function that takes something and returns that same thing.
- The second parameter is something of that type also and the return value is also of the same type.

Higher order functions

- ghci> applyTwice (+3) 10
- 16
- ghci> applyTwice (++ " HAHA") "HEY"
- "HEY HAHA HAHA"
- ghci> applyTwice ("HAHA " ++) "HEY"
- "HAHA HAHA HEY"

Higher order functions

- flip' :: (a -> b -> c) -> (b -> a -> c)
- flip' f y x = f x y

- flip' zip [1,2,3,4,5] "hello"
- [('h',1),('e',2),('l',3),('l',4),('o',5)]

Inbuilt functions: Maps and filters

- **map** takes a function and a list and applies that function to every element in the list, producing a new list.
- :type map
- ghci> map (+3) [1,5,3,1,6]
- [4,8,6,4,9]

Inbuilt functions: Maps and filters

- **filter** is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate.

- *Main> filter (>3) [1,5,3,2,1,6,4,3,2,1]
- [5,6,4]

Lambda Function

- An anonymous function is a function without a name.
- Also called as Lambda function
- E.g. \x -> x + 1
- Backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda.
- That is a nameless function which increments its parameter, x.

Lambda Function

- Prelude> (\x -> x+1) 5
- 6

- Prelude> (\x y -> x+y) 5 6
- 11

Pattern Matching

- Pattern matching consists of specifying patterns to which some data should conform, then checking to see if it does and de-constructing the data according to those patterns.
- When defining functions, you can define separate function bodies for different patterns.
- One can pattern match on any data type – numbers, characters, lists, tuples, etc.

Pattern Matching

- A function that checks if the number we supplied to it is a 7 or not

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

Pattern Matching

- Integral is a typeclass.
- A typeclass is a sort of interface that defines some behavior.
- If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes.
- Num typeclass does not provide a division operator which is provided in Integral
- Num includes all numbers, including real numbers and integral numbers
- Integral includes only integral (whole) numbers.
- Integral typeclass includes Int and Integer.

Pattern Matching

- => separates two parts of a type signature:
- On the left, typeclass constraints
- On the right, the actual type

Pattern Matching

- When you call lucky, the patterns will be checked from top to bottom and when the argument of the function conforms to a pattern, the corresponding function body will be used.

Pattern Matching

- We want a function that
  - says the numbers from 1 to 5 and
  - says “Not between 1 and 5 for any other number

Pattern Matching

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Pattern Matching

- If we moved the last pattern (the catch-all one) to the top, it would always say "Not between 1 and 5", because it would catch all the numbers and they wouldn't have a chance to fall through and be checked for any other patterns.

Guards

- Patterns are a way of making sure a value conforms to some form and deconstructing it
- Guards are a way of testing whether some property of a value are true or false.
- Guards are indicated by pipes that follow a function's name and its parameters.
- A guard is a boolean expression.
- If it evaluates to True, then the corresponding function body is used.
- If it evaluates to False, checking drops through to the next guard and so on.
- Many times, the last guard is **otherwise**
- otherwise is defined simply as otherwise = True and catches everything

Guards

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = " You're underweight!"
    | bmi <= 25.0 = "You're supposedly normal!"
    | bmi <= 30.0 = "You're fat!"
    | otherwise   = “Something wrong!"
```

Note that there's no = right after the function name and its parameters, before the first guard.

Case expressions

- Pattern matching on function parameters can only be done when defining functions, whereas case expressions can be used anywhere.

case expression **of** pattern -> result

pattern -> result

pattern -> result

...

- expression is matched against the patterns.
- The pattern matching action is the same as expected: the first pattern that matches the expression is used.
- If it falls through the whole case expression and no suitable pattern is found, a runtime error occurs.

Case expressions

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
Main> describeList [1,2,3]
"The list is a longer list."
*Main> describeList []
"The list is empty."
*Main> describeList [1]
"The list is a singleton list."
```

I/O

- main = putStrLn "hello, world"

ghci> :t putStrLn

putStrLn :: String -> IO ()

- putStrLn takes a string and returns an I/O action that has a result type of ()
- (i.e. the empty tuple)
- An I/O action is something that, when performed, will carry out an action with a side-effect
- It will also contain some kind of return value inside it.
- Printing a string to the terminal doesn't really have any kind of meaningful return value, so a dummy value of () is used.

I/O

- **do** syntax to glue together several I/O actions into one.

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

I/O

- name <- getLine.
- it reads a line from the input and stores it into a variable called name.

ghci> :t getLine

getLine :: IO String

- getLine is an I/O action that contains a result type of String
- It will wait for the user to input something at the terminal and then that something will be represented as a string.
- getLine has a type of IO String, so name will have a type of String

I/O

- **putStr** is much like putStrLn in that it takes a string as a parameter and returns an I/O action that will print that string to the terminal, only putStr doesn't jump into a new line after printing out the string while putStrLn does
- **putChar** takes a character and returns an I/O action that will print it out to the terminal.
- **print** takes a value of any type

I/O

```
*Main> putStrLn [1,2,3]

<interactive>:16:11: error:
    * No instance for (Num Char) arising from the literal `1'
    * In the expression: 1
      In the first argument of `putStrLn', namely `[1, 2, 3]'
      In the expression: putStrLn [1, 2, 3]
*Main> print([1,2,3])
[1,2,3]
*Main> print $[1,2,3]
[1,2,3]
```

I/O

```
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

I/O

- **getChar** is an I/O action that reads a character from the input.
- **getContents** is an I/O action that reads everything from the standard input until it encounters an end-of-file character.

I/O

- **read** casts strings to another type
- read "12"::Int
- read "12"::Double

*Main> :t read

read :: Read a => String -> a

I/O

- **readFile** has a type signature of readFile :: FilePath -> IO String
- readFile takes a path to a file and returns an I/O action that will read that file and bind its contents to something as a string.

I/O

- **writeFile** has a type of writeFile :: FilePath -> String -> IO ().
- It takes a path to a file and a string to write to that file and returns an I/O action that will do the writing.

I/O

- **appendFile** has a type signature that's just like writeFile, only appendFile doesn't truncate the file to zero length if it already exists but it appends stuff to it.

Functors

- Functor is a typeclass in Haskell.

- The Functor type class provides a generic interface for applying functions to values in a container(List) or context(Maybe)

What is Maybe?

- The Maybe type encapsulates an optional value. A value of type Maybe either contains a value (represented as Just a, a is some data type), or it is empty (represented as Nothing).

Functors

- In order for a data type to be an instance of the Functor typeclass, it must implement a single function: **fmap**.
- fmap :: Functor f => (a -> b) -> f a -> f b
- fmap :: (a -> b) -> f a -> f b
- The fmap function takes two inputs
  - A function between two data types.
  - The second parameter is some container of the first type.
- The output then is a container of the second type.

Functors

- Functor typeclass is basically for things that can be mapped over.
  - list type is part of the Functor typeclass

```
class Functor f where
fmap :: (a -> b) -> f a -> f b
```

- Functor defines one function, fmap, and doesn't provide any default implementation for it.

Functors

- For lists, fmap is just map

- We get the same results when using them on lists.

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Functors

- Maybe is a functor

```
instance Functor Maybe where
fmap f (Just x) = Just (f x)
fmap f Nothing = Nothing
```

If we have no value in the first place, then the result is Nothing. If we do have a value, simply apply the function to the value, and rewrap it in Just

Applicative Functors

- Functors allows us to run transformations on data regardless of how the data is wrapped
- But we cannot combine 2 wrapped data using functors
- If we just have to multiply by a constant value we can use the functor approach.
- (Just 4) * (Just 5)
  - Will give error
- Solution: Applicative Functors

Applicative Functors

- It has two main functions:

- pure :: a -> f a
- (<*>) :: f (a -> b) -> f a -> f b
- The pure function takes some value and wraps it in a minimal context
- The <*> function, called sequential application, takes two parameters.
- First, it takes a function wrapped in the context.
- Second, a wrapped value.
- Its output is the result of applying the function to the value, rewrapped in the context.

Applicative Functors

- It is called an applicative functor because it allows us to apply a wrapped function.
- Since sequential application takes a wrapped function, we often begin by wrapping something with either pure or fmap
- If we want to multiply 2 maybe values, we start by wrapping the bare multiplication function in pure, then we sequentially apply both Maybe values:
- pure (*) <*> (Just 4) <*> (Just 5)
- The pure function may simply wrap a value with Just

Applicative Functors

- Applicative instance for Maybe

```
instance Applicative Maybe where
  pure = Just
  (<*>) Nothing = Nothing
    (<*>) (Just f) (Just x) = Just (f x)
```

Applicative Functors

- Applicative instance for Lists

```
instance Applicative [] where
  pure a = [a]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Applicative Functors

- When we have only one function, this results in familiar mapping behavior.
- >> pure (4 *) <*> [1,2,3]
- [4,8,12]
- >> [(1+), (5*), (10*)] <*> [1,2,3]
- [2,3,4,5,10,15,10,20,30]

Applicative Functors

- when we have multiple functions
- >> pure (*) <*> [1,2,3] <*> [10,20,30]
- [10,20,30,20,40,60,30,60,90]

Monads

- In Haskell a monad is represented as a type constructor
- A Monad wraps a value or a computation with a particular context
- A monad must define both a means of wrapping normal values in the context, and a way of combining computations within the context
- It has two functions

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Monads

- The return function specifies how to wrap values in the monad's context.
- The >>= operator, which we call the "bind" function, specifies how to combine two operations within the context.

Monads

- (>>=)-the basic intuition is that it combines two computations into one larger computation.
- The first argument, m a, is the first computation.
- The second argument to (>>=) has type a -> m b: a function of this type, given a result of the first computation, can produce a second computation to be run.
- x >>= k is a computation which runs x, and then uses the result(s) of x to decide what computation to run second, using the output of the second computation as the result of the entire computation.

Monads

- Just as Maybe is a functor and an applicative functor, it is also a monad.

```
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  Just a >>= f = f a
```

Monads

- The context the Maybe monad describes is simple.
- Computations in Maybe can either fail or succeed with a value.
- We can take any value and wrap it in this context by calling the value a "success".
- We do this with the Just constructor.
- We represent failure by Nothing.