Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)
Department of Computer Science and Engineering (Data Science)

JHANVI PAREKH

60009210033

CSE(Data Science)

Experiment No 5

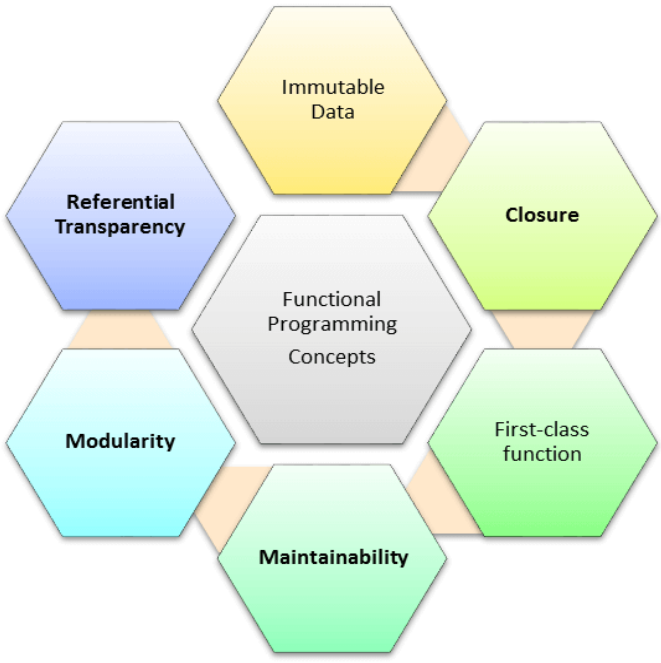
Aim: Implement Functional Programming using Haskell

Theory:

Functional Programming

Functional programming (also called FP) is a way of thinking about software construction by creating pure functions. Functional languages emphasize on expressions and declarations rather than execution of statements. Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function. Functional programming method focuses on results, not the process. Emphasis is on what is to be computed. Data is immutable. Functional programming Decompose the problem into 'functions. It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation. It does not support iteration like loop statements and conditional statements like If-Else.

Basic Functional Programming Terminology and Concepts



Haskell

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

Haskell is more intelligent than other popular programming languages such as Java, C, C++, PHP, etc. Haskell is a Functional Programming Language that has been specially designed to handle symbolic computation and list processing applications.

List in Haskell

There are ***two major differences*** in Haskell lists, compared to other languages, especially dynamically typed languages, like Python, Ruby, PHP, and JavaScript.

1. First, lists in Haskell are ***homogenous***. This means that a Haskell list can only hold elements of the ***same type***
2. Second, lists in Haskell are (internally) implemented as ***linked lists***. This is different from many other languages, where the word "list" and "array" is used interchangeably. Linked lists and arrays have very different performance characteristics when operating on large amounts of data. Keep this in mind for the future. If you're dealing with very large lists, where you want to randomly access the i-th There are two major differences in Haskell lists, compared to other languages, especially dynamically typed languages, like Python, Ruby, PHP, and JavaScript.

Example of list in Haskell

1. **let** lostNumbers = [4,8,15,16,23,42]
2. lostNumbers
3. [4,8,15,16,23,42]

As you can see, lists are denoted by square brackets and the values in the lists are separated by commas. If we tried a list like **[1,2,'a',3,'b','c',4]**, Haskell would complain that characters (which are, by the way, denoted as a character between single quotes) are not numbers. Speaking of characters, strings are just lists of characters. **"hello"** is just syntactic sugar for **['h','e','l','l','o']**.

Problem 1

Find the last element of a list.

Example in Haskell:

```
λ> myLast [1,2,3,4]
4
λ> myLast ['x','y','z']
'z'
```

Problem 2

(*) Find the K'th element of a list. The first element in the list is number 1.

Example:

```
* (element-at '(a b c d e) 3)
```

```
c
```

Example in Haskell:

```
λ> elementAt [1,2,3] 2
2
λ> elementAt "haskell" 5
'e'
```

Function in Haskell

Functions play a major role in Haskell, as it is a functional programming language. Like other languages, Haskell does have its own functional definition and declaration.

- Function declaration consists of the function name and its argument list along with its output.
- Function definition is where you actually define a function.

Let us take small example of **add** function to understand this concept in detail.

```
add: Integer -> Integer -> Integer   --function declaration

add x y = x + y                      --function definition

main = do

  putStrLn "The addition of the two numbers is:"

  print (add 9 10)                   --calling a function
```

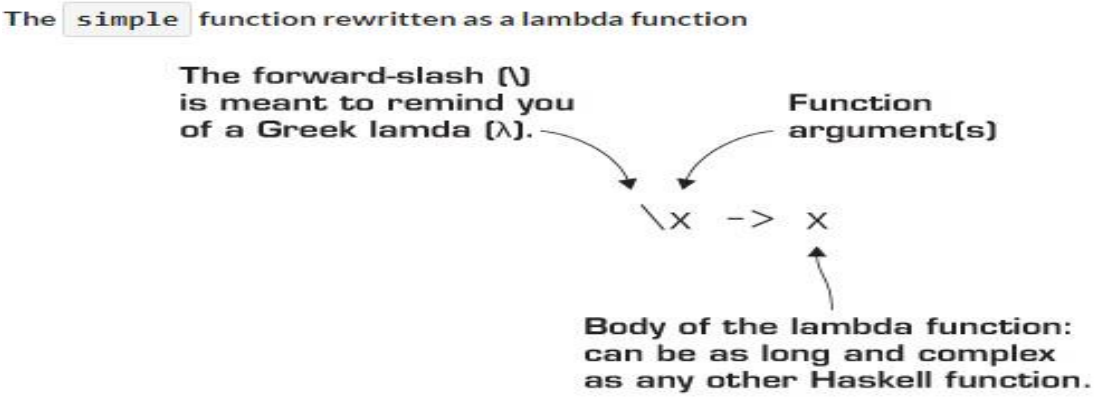
Here, we have declared our function in the first line and in the second line, we have written our actual function that will take two arguments and produce one integer type output. Like most other languages, Haskell starts compiling the code from the **main** method. Our code will generate the following output −

```
The addition of the two numbers is:
7
```

Lambda Function in Haskell

One of the most foundational concepts in functional programming is a function without a name, called a lambda function (hence lambda calculus). Lambda functions are often referred to using the lowercase Greek letter λ. Another common name for a lambda function is an anonymous function. You can use a lambda function to redefine your simple function, only

without a name. To do this, you use Haskell's lambda syntax, shown in figure below.



Lab Assignments to complete in this session

Implement Functional Programming using Haskell

1) Write the program in Haskell to perform following arithmetic operations: -
Addition
Subtraction
Multiplication ‘
Division

main.hs

```haskell
-- Addition
addition :: Num a => a -> a -> a
addition x y = x + y

-- Subtraction
subtraction :: Num a => a -> a -> a
subtraction x y = x - y

-- Multiplication
multiplication :: Num a => a -> a -> a
multiplication x y = x * y

-- Division
division :: Fractional a => a -> a -> a
division x y = x / y

-- Example usage
main :: IO ()
main = do
  putStrLn "Enter the first number: "
  input1 <- getLine
  let x = read input1 :: Double

  putStrLn "Enter the second number: "
  input2 <- getLine
  let y = read input2 :: Double

  putStrLn $ "Addition: " ++ show (addition x y)
  putStrLn $ "Subtraction: " ++ show (subtraction x y)
  putStrLn $ "Multiplication: " ++ show (multiplication x y)
  putStrLn $ "Division: " ++ show (division x y)
```

```
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking a.out ...
Enter the first number:
5
Enter the second number:
6
Addition: 11.0
Subtraction: -1.0
```

```
Addition: 11.0
Subtraction: -1.0
Multiplication: 30.0
Division: 0.8333333333333334


...Program finished with exit code 0
Press ENTER to exit console.
```

2) Write the above program using Functions in Haskell

```haskell
-- Addition
addition :: Num a => a -> a -> a
addition x y = x + y

-- Subtraction
subtraction :: Num a => a -> a -> a
subtraction x y = x - y

-- Multiplication
multiplication :: Num a => a -> a -> a
multiplication x y = x * y

-- Division
division :: Fractional a => a -> a -> a
division x y = x / y

-- Function to perform arithmetic operations based on user input
performOperations :: IO ()
performOperations = do
  putStrLn "Enter the first number: "
  input1 <- getLine
  let x = read input1 :: Double

  putStrLn "Enter the second number: "
  input2 <- getLine
  let y = read input2 :: Double

  putStrLn $ "Addition: " ++ show (addition x y)
  putStrLn $ "Subtraction: " ++ show (subtraction x y)
  putStrLn $ "Multiplication: " ++ show (multiplication x y)
  putStrLn $ "Division: " ++ show (division x y)
```

```haskell
subtraction :: Num a => a -> a -> a
subtraction x y = x - y

-- Multiplication
multiplication :: Num a => a -> a -> a
multiplication x y = x * y

-- Division
division :: Fractional a => a -> a -> a
division x y = x / y

-- Function to perform arithmetic operations based on user input
performOperations :: IO ()
performOperations = do
  putStrLn "Enter the first number: "
  input1 <- getLine
  let x = read input1 :: Double

  putStrLn "Enter the second number: "
  input2 <- getLine
  let y = read input2 :: Double

  putStrLn $ "Addition: " ++ show (addition x y)
  putStrLn $ "Subtraction: " ++ show (subtraction x y)
  putStrLn $ "Multiplication: " ++ show (multiplication x y)
  putStrLn $ "Division: " ++ show (division x y)

-- Main function
main :: IO ()
main = performOperations

```

```
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking a.out ...
Enter the first number:
6
Enter the second number:
6
Addition: 12.0
Subtraction: 0.0
```

```
Addition: 12.0
Subtraction: 0.0
Multiplication: 36.0
Division: 1.0


...Program finished with exit code 0
Press ENTER to exit console.
```

3)Write a program to perform following on list in Haskell

1. Find first element
2. Find Length
3. Add element at The End of the list
4. Add element at the Head of the List
5. Reverse The List

```haskell
-- Find the first element of a list
findFirstElement :: [a] -> Maybe a
findFirstElement [] = Nothing
findFirstElement (x:_) = Just x

-- Find the length of a list
findLength :: [a] -> Int
findLength = length

-- Add an element at the end of a list
addElementAtEnd :: [a] -> a -> [a]
addElementAtEnd xs x = xs ++ [x]

-- Add an element at the head of a list
addElementAtHead :: [a] -> a -> [a]
addElementAtHead xs x = x : xs

-- Reverse a list
reverseList :: [a] -> [a]
reverseList = reverse

-- Example usage
main :: IO ()
main = do
  let myList = [1, 2, 3, 4, 5]

  putStrLn $ "List: " ++ show myList
  putStrLn $ "First element: " ++ show (findFirstElement myList)
  putStrLn $ "Length: " ++ show (findLength myList)

  let newList = addElementAtEnd myList 6
```

```haskell
-- Add an element at the end of a list
addElementAtEnd :: [a] -> a -> [a]
addElementAtEnd xs x = xs ++ [x]

-- Add an element at the head of a list
addElementAtHead :: [a] -> a -> [a]
addElementAtHead xs x = x : xs

-- Reverse a list
reverseList :: [a] -> [a]
reverseList = reverse

-- Example usage
main :: IO ()
main = do
  let myList = [1, 2, 3, 4, 5]

  putStrLn $ "List: " ++ show myList
  putStrLn $ "First element: " ++ show (findFirstElement myList)
  putStrLn $ "Length: " ++ show (findLength myList)

  let newList = addElementAtEnd myList 6
  putStrLn $ "List with element added at the end: " ++ show newList

  let newList' = addElementAtHead myList 0
  putStrLn $ "List with element added at the head: " ++ show newList

  let reversedList = reverseList myList
  putStrLn $ "Reversed list: " ++ show reversedList
```

```
Length: 5
List with element added at the end: [1,2,3,4,5,6]
List with element added at the head: [0,1,2,3,4,5]
Reversed list: [5,4,3,2,1]


...Program finished with exit code 0
Press ENTER to exit console.
```

4)Write a Function in Haskell to find the length of the list

main.hs

```haskell
module Main where

-- Find the length of a list
findLength :: [a] -> Int
findLength [] = 0
findLength (_:xs) = 1 + findLength xs

-- Main function
main :: IO ()
main = do
  let myList = [1, 2, 3, 4, 5]
  putStrLn $ "List: " ++ show myList
  putStrLn $ "Length: " ++ show (findLength myList)

```

```
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking a.out ...
List: [1,2,3,4,5]
Length: 5


...Program finished with exit code 0
Press ENTER to exit console.
```