

**St. Francis Institute of Technology
Borivli (West), Mumbai-400103
Department of Information Technology**

Experiment – 8

- 1. Aim:** To implement recursive functions using Haskell Programming
- 2. Objective:** After performing the experiment, the students will be able to understand and implement– Recursive functions
- 3. Lab objective mapped:** To understand and implement declarative programming paradigm through functional programming (PSO2) (PO1)
- 4. Prerequisite:** Understanding of basic operators (arithmetic, comparator and logical), lists and simple functions
- 5. Requirements:** The following are the requirements – Haskell Compiler

6. Pre-Experiment Theory:

Recursion is important to Haskell because unlike imperative languages, you do computations in Haskell by declaring what something is instead of declaring how you get it. That's why there are no while loops or for loops in Haskell and instead we many times have to use recursion to declare what something is. Some functions, such as factorial, are simpler to define in terms of other functions. Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Example

```
fac 1 = 1
fac n = n * fac (n-1)
fac 3
= 3 * fac 2
= 3 * (2 * fac 1)
= 3 * (2 * (1))
= 6
```

7. Laboratory Exercise

A. Steps to be implemented

1. Open command prompt
2. Change the directory where you have saved your .hs file at the prelude prompt
3. Open notepad, write your code and save as .hs file
4. At the prelude prompt, compile the file using the command ‘:l file_name.hs’
5. After successful compilation, run the code using the command ‘Main> main [if using main function]’
6. Main> name of function arguments [if using functions]

B. Program Code**1. Write Haskell program to find Fibonacci series using the principle of recursion**

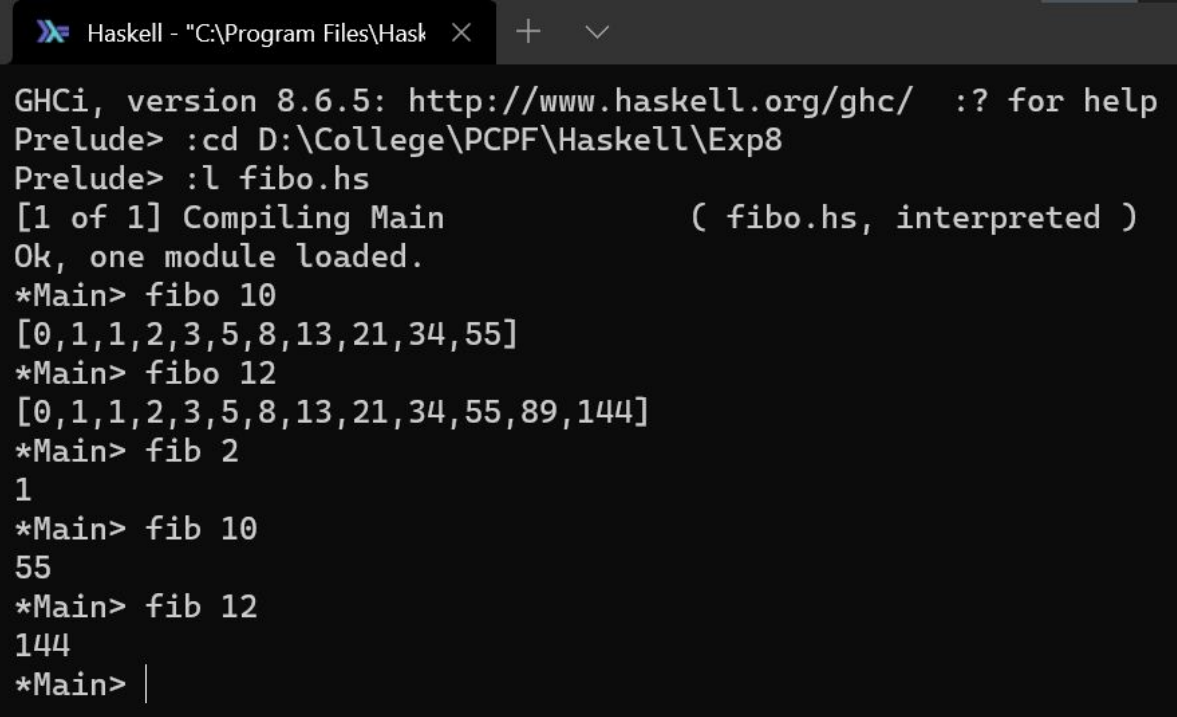
```

fib::Integer->Integer
fib 0=0
fib 1=1
fib n=fib (n-1)+fib (n-2)    --takes an integer n as input and returns nth
fibonacci number

fibonacci::Integer->[Integer]
fibonacci n=[fib x|x<-[0..n]]    --Takes an integer n as input and returns a
list of fibonacci numbers upto n

```

Output:



```

Haskell - "C:\Program Files\Hask
+  v
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp8
Prelude> :l fibo.hs
[1 of 1] Compiling Main                ( fibo.hs, interpreted )
Ok, one module loaded.
*Main> fibo 10
[0,1,1,2,3,5,8,13,21,34,55]
*Main> fibo 12
[0,1,1,2,3,5,8,13,21,34,55,89,144]
*Main> fib 2
1
*Main> fib 10
55
*Main> fib 12
144
*Main> |

```

From this output we can clearly see that the function fib n takes an integer value n as input and returns the nth element of the fibonacci series. This is achieved by recursively calling itself as fib (n-1)+fib (n-2) until the nth element is found. The function fibo n takes an integer value as input and returns a list of fibonacci numbers upto n.

2. WAP to find the product of all numbers in a list using recursion

```

prod::[Int]->Int
prod [ ] = 1
prod (x:xs) =x*prod xs    --Takes an integer list as input and returns the
product of all elements in the list

```

Output:

```
Haskell - "C:\Program Files\Hask
*Main> :l product.hs
[1 of 1] Compiling Main                ( product.hs, interpreted )
Ok, one module loaded.
*Main> prod [1,2,3,4,5]
120
*Main> prod [12,23,0,14]
0
```

From the result we can see that the function takes an input integer list and returns the product of each element of the list. This is done as the function recursively calls itself until it has visited all the elements of the list. The piece of code explains us exactly how it is done $\text{prod } (x:xs) = x * \text{prod } xs$.

3. WAP to find the length of a list using recursion

```
len :: [Int] -> Int
len [] = 0
len (x:xs) = 1 + len xs  --Takes an integer list as an input and returns
it's length
```

Output:

```
Haskell - "C:\Program Files\Hask
*Main> :l length.hs
[1 of 1] Compiling Main                ( length.hs, interpreted )
Ok, one module loaded.
*Main> len [1,2,3,4,5,6]
6
*Main> len[12,24,36,48]
4
*Main> |
```

From the output we can see that the function len takes an integer list as input and returns the length of that list. This is done recursively as the function traverses the entire list and keeps on adding 1 to the result as an element is visited. After all the elements are visited it returns the result.

8. Post Experimental Exercise

A. Questions:

1. Explain the principle of recursion in Haskell? How is recursion useful?

The functional programming paradigm doesn't allow the use of loops for two simple reasons. First, a loop requires the maintenance of state, and the functional programming paradigm doesn't allow

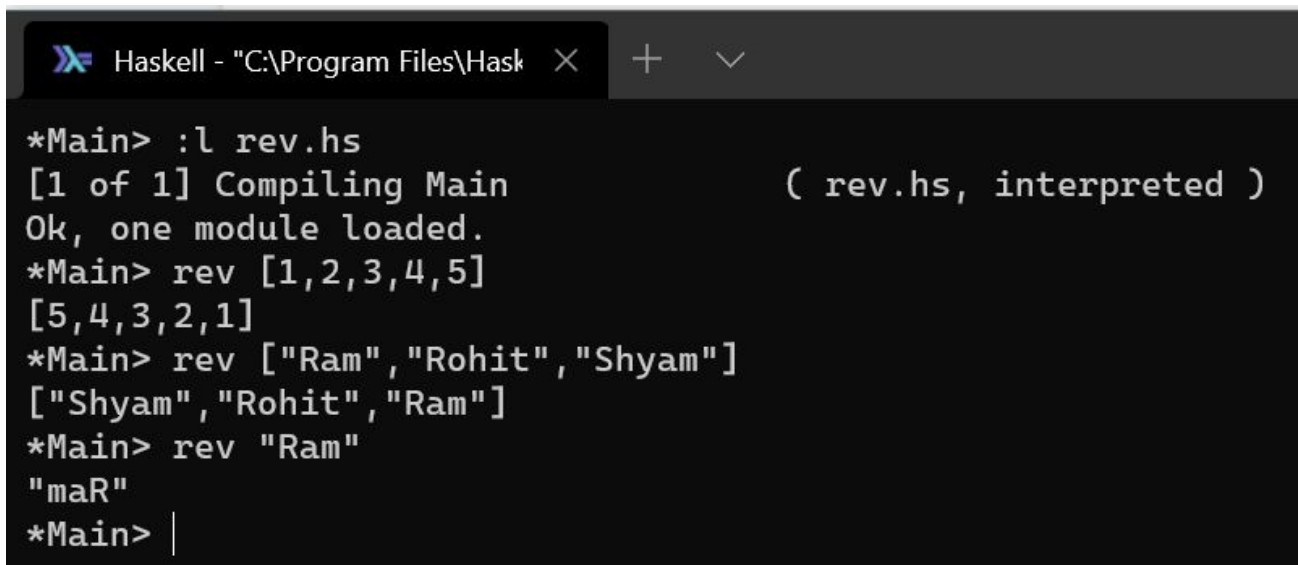
state. Second, loops generally require mutable variables so that the variable can receive the latest data updates as the loop continues to perform its task.

To overcome this we can use recursion. Recursion is actually a way of defining functions in which the function is applied inside its own definition. One of the most interesting aspects of using Higher Order functions in the functional programming paradigm is that you can now pass functions rather than variables to enable recursion. This capability makes recursion in the functional programming paradigm significantly more powerful than it is in other paradigms because the function can do more than a variable, and by passing different functions, you can alter how the main recursion sequence works.

2. WAP to find the reverse of a list using recursion

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]  --Takes any list and reverses it
```

Output:



```
Haskell - "C:\Program Files\Hask  X + v
*Main> :l rev.hs
[1 of 1] Compiling Main                ( rev.hs, interpreted )
Ok, one module loaded.
*Main> rev [1,2,3,4,5]
[5,4,3,2,1]
*Main> rev ["Ram","Rohit","Shyam"]
["Shyam","Rohit","Ram"]
*Main> rev "Ram"
"maR"
*Main> |
```

From the output we can see that the function `rev` takes an input list of any type and reverses it. This is done by storing the first element of the list into the output list and then inserting the next element of the list in the beginning of the output list. This process is done recursively until all the elements of the input list are visited.

C. Conclusion:

In this experiment we have studied and implemented the concept of recursion in Haskell by writing programs that implement recursion. As loops are not supported in Functional programming, recursion is used to perform repetitive tasks in Haskell. Throughout this experiment we have studied how recursion can be implemented with different functions.

To perform this experiment and write all the codes Visual Studio Code was used as a text editor and to compile all the programs Glasgow Haskell Compiler was used.

Haskell being a functional language does not support mutable variables and state. This makes the use of loops impossible. The lack of state and mutable variables makes recursion the perfect tool for

performing repetitive tasks. Also support for higher order functions makes recursion in the functional programming paradigm significantly more powerful than it is in other paradigms because the function can do more than a variable, and by passing different functions, you can alter how the main recursion sequence works.

D. References:

[1] <https://www.haskell.org/>

[2] <http://learnyouahaskell.com/>

[3] Michael L Scott, "Programming Language Pragmatics", Third edition, Elsevier publication