

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

Experiment – 7

1. Aim: To implement higher order functions using Haskell Programming

2. Objective: After performing the experiment, the students will be able to understand and implement–

- Higher order functions
- Build higher order 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:

A higher-order function is defined as a function that takes other functions as arguments or returns a function as result.

- Function application is evaluated left to right, i.e. is left associative.
- This means that a function f of two arguments $f\ x\ y$ is correctly interpreted as f being applied to x to give a function which is applied to y , i.e. $f\ x\ y == (f\ x)\ y$
- The type signature of f hints at this as well.
- $f :: a \rightarrow b \rightarrow c$
- This says that f takes a value of type a and returns a value of type $b \rightarrow c$, which is a function that takes a value of type b and returns a value of type c .
- Here, the \rightarrow symbol in the type signature is applied right to left, i.e. it is right associative
- $f :: a \rightarrow (b \rightarrow c)$
- The first or innermost function application corresponds to the outermost \rightarrow
- Putting a space between two things is simply function application
- The space is sort of like an operator and it has the highest precedence
- Parentheses indicate that the first parameter is a function that takes something and returns that same thing.

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

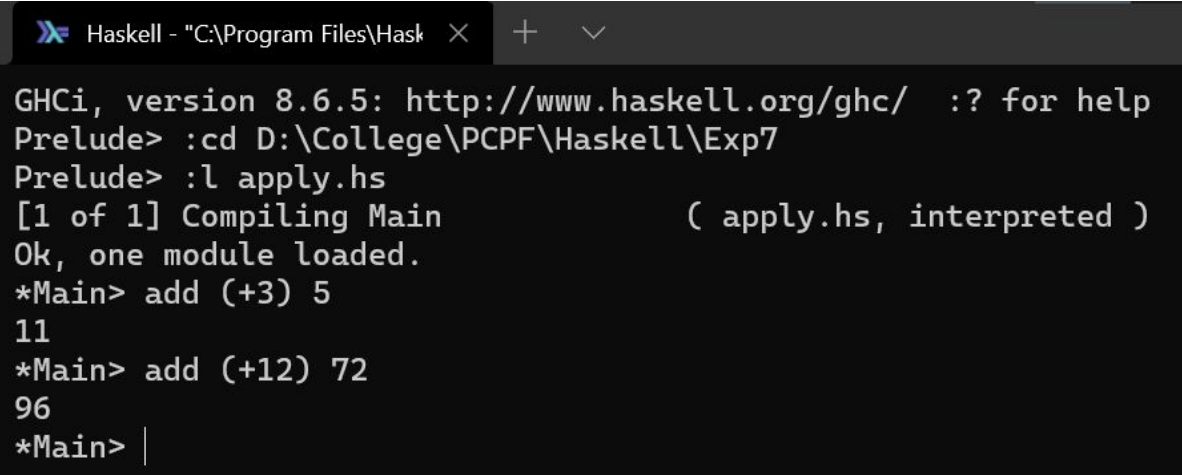
1. Write a higher order function to –

a. Create a function to add two numbers. Take the function output as the argument to create the higher order function to add the number twice.

Program:

```
add :: (Int -> Int) -> Int -> Int
add f x = f(f x)
```

Output:



```
Haskell - "C:\Program Files\Hask  ×  +  ∨
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l apply.hs
[1 of 1] Compiling Main                ( apply.hs, interpreted )
Ok, one module loaded.
*Main> add (+3) 5
11
*Main> add (+12) 72
96
*Main> |
```

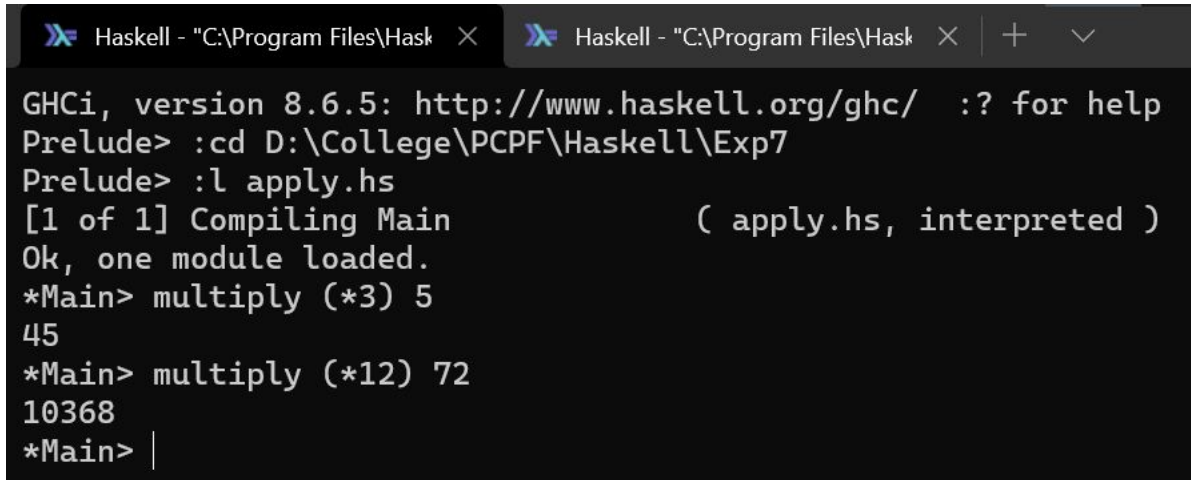
From the above output we can see that this is an example of higher order function. Here function add takes 2 arguments, +3 and 5, here +3 is a function itself of the type Int->Int, and 5 is an Int. We just use the parameter f as a function, applying x to it by separating them with a space and then applying the result to f again.

b. Create a function to multiply two numbers. Take the function output as the argument to create the higher order function to multiply the number twice.

Program:

```
multiply :: (Int -> Int) -> Int -> Int
multiply f x = f(f x)
```

Output:



```
Haskell - "C:\Program Files\Hask  X Haskell - "C:\Program Files\Hask  X + v
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l apply.hs
[1 of 1] Compiling Main                                ( apply.hs, interpreted )
Ok, one module loaded.
*Main> multiply (*3) 5
45
*Main> multiply (*12) 72
10368
*Main> |
```

From the above output we can see that this is an example of higher order function. Here function add takes 2 arguments, *3 and 5, here *3 is a function itself of the type `Int->Int`, and 5 is an `Int`. We just use the parameter `f` as a function, applying `x` to it by separating them with a space and then applying the result to `f` again.

c. Write a program in Haskell to display area of triangle and circle

Program:

```
main =do
    -- prompt the user and get their input
    putStrLn "Enter height:"
    height<-getLine    --height::String

    putStrLn "Enter base:"
    base<-getLine      --base::String

    let h =read height :: Double  --h::Double
    let b =read base  :: Double   --b::Double

    -- displaying the output
    putStrLn "Area of triangle is:"
    print (0.5*h*b)
```

Output:

```
Haskell - "C:\Program Files\Hask  ×  +  ∨

GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l triangle.hs
[1 of 1] Compiling Main                  ( triangle.hs, interpreted )
Ok, one module loaded.
*Main> main
Enter height:
12
Enter base:
15
Area of triangle is:
90.0
*Main> main
Enter height:
6
Enter base:
10
Area of triangle is:
30.0
*Main> |
```

From the output we can infer that the function prompts the user to enter values of height and base of the triangle, calculates the area and prints it in the command prompt.

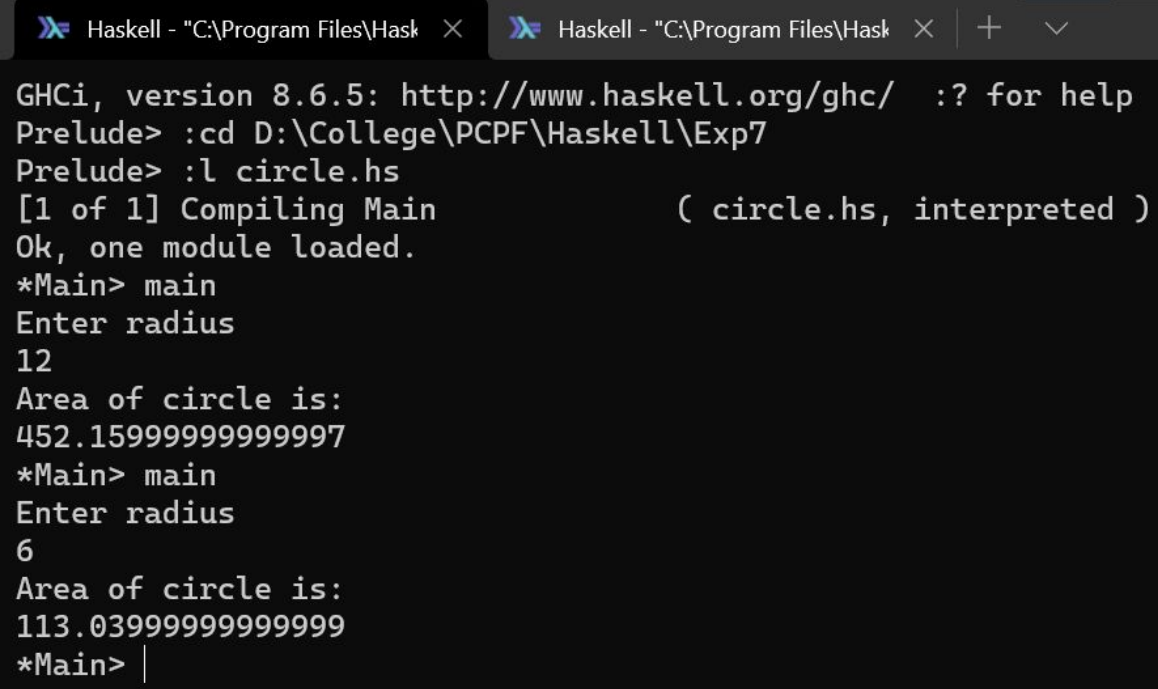
Program:

```
main = do
    -- prompt the user and get their input
    putStrLn "Enter radius "
    radius<-getLine    --radius::String

    let r=read radius :: Double    --r::Double

    -- displaying the output
    putStrLn "Area of circle is:"
    print (3.14*r*r)
```

Output:



```

Haskell - "C:\Program Files\Hask... × Haskell - "C:\Program Files\Hask... × + ▾
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l circle.hs
[1 of 1] Compiling Main                                ( circle.hs, interpreted )
Ok, one module loaded.
*Main> main
Enter radius
12
Area of circle is:
452.15999999999997
*Main> main
Enter radius
6
Area of circle is:
113.03999999999999
*Main> |

```

From the output we can infer that the function prompts the user to enter the value of radius , calculates the area and prints it in the command prompt.

8. Post Experimental Exercise

A. Questions:

1. What are higher order functions?

At the heart of functional programming is the idea that functions are just like any other value. The power of functional style comes from handling functions themselves as regular values, i.e. by passing functions to other functions and returning them from functions. Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function.

2. State and explain the syntax of higher order function. Explain with example

Functions can take functions as parameters and also return functions. To illustrate this, we're going to make a function that takes a function and then applies it twice to something.

```

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

```

Here (a->a) signifies that one of the parameters is a function that takes something and returns that same thing. The second parameter is something of that same type also and the return value is also of the same type. The function takes two parameters, the first parameter is a function (of type a -> a) and the second is of the same type a. The function can also be Int -> Int or String -> String or whatever. But then, the second parameter also has to be of that type. In the body of the function we use the parameter f as a function, applying x to it by separating them with a space and then applying the result to f again.

The output can be given as:

```

Haskell - "C:\Program Files\Hask  x  +  v
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l applyTwice.hs
[1 of 1] Compiling Main                ( applyTwice.hs, interpreted )
Ok, one module loaded.
*Main> applyTwice (+3) 10
16
*Main> applyTwice (== " HAHA") "HEY"
"HEY HAHA HAHA"
*Main> |

```

Here we can see that the type of the parameters does not matter as long as they are the same, we can use this function on any data type.

3. State the lambda abstraction of the above functions of part 7B (program code)

a) $(\lambda x y \rightarrow x + y + x) \ 3 \ 5$

```

Haskell - "C:\Program Files\Hask  x  Haskell - "C:\Program Files\Hask  x  +  v
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l apply.hs
[1 of 1] Compiling Main                ( apply.hs, interpreted )
Ok, one module loaded.
*Main> add (+3) 5
11
*Main> add (+12) 72
96
*Main> (\x y->x+y+x) 3 5
11
*Main> (\x y->x+y+x) 12 72
96
*Main> |

```

b) $(\lambda x y \rightarrow x * y * x) \ 3 \ 5$

```

Haskell - "C:\Program Files\Hask  x  Haskell - "C:\Program Files\Hask  x  +  v
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp7
Prelude> :l apply.hs
[1 of 1] Compiling Main                ( apply.hs, interpreted )
Ok, one module loaded.
*Main> multiply (*3) 5
45
*Main> multiply (*12) 72
10368
*Main> (\x y->x*y*x) 3 5
45
*Main> (\x y->x*y*x) 12 72
10368
*Main> |

```

C. Conclusion:

In this Experiment we have written Haskell programs to implement and understand Higher order functions. A function that takes another function (or several functions) as an argument is called a higher-order function. The power of functional style comes from handling functions themselves as regular values, i.e. by passing functions to other functions and returning them from functions. Throughout this experiment we have studied different ways in which we can use Higher order functions instead of regular 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.

If we want to define computations by defining what the computation will do instead of defining steps that change some state and maybe looping them, higher order functions are indispensable. They're a really powerful way of solving problems and thinking about programs.

D. References:

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

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

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