

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

Experiment – 5

1. Aim: To implement functions in Haskell Programming

Program Code

1. Implement the following inbuilt functions using Haskell compiler at prelude prompt

- | | |
|-----------------------------|----------------------|
| 1. succ 6 | 20. x=45 |
| 2. succ (succ 5) | 21. print x |
| 3. min 5 6 | 22. return True |
| 4. max 5 6 | 23. return False |
| 5. max 101 101 | 24. x <- return 35 |
| 6. succ 9 + max 5 4 + 1 | 25. print x |
| 7. (max 5 4)+(succ 9)+1 | 26. putStrLn "hello" |
| 8. (succ 9) + (max 5 4) + 1 | |
9. We wanted to get the successor of the product of numbers 9 and 10. we couldn't write succ 9*10 because that would get the successor of 9, which would then be multiplied by 10
10. succ 9*10
 11. succ (9*10)
 12. div 92 10
 13. div 3 4
 14. div 4 3
 15. 4/3
 16. mod 7 5
 17. mod 3 1
 18. mod 7 2
 19. reverse "hello"

```
Haskell - "C:\Program Files\Hask × + ∨
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
Prelude> succ 6
7
Prelude> succ(succ 5)
7
Prelude> min 5 6
5
Prelude> max 5 6
6
Prelude> max 101 101
101
Prelude> succ 9 + max 5 4 + 1
16
Prelude> (max 5 4)+(succ 9)+1
16
Prelude> succ 9*10
100
Prelude> succ (9*10)
91
Prelude> div 92 10
9
Prelude> div 3 4
0
Prelude> div 4 3
1
Prelude> 4/3
1.3333333333333333
Prelude> mod 7 5
2
Prelude> mod 3 1
0
Prelude> mod 7 2
```

```

Haskell - "C:\Program Files\Hask
Prelude> reverse "hello"
"olleh"
Prelude> x=45
Prelude> print x
45
Prelude> return True
True
Prelude> return False
False
Prelude> x<-return 35
Prelude> print x
35
Prelude> putStrLn "hello"
hello
Prelude> |

```

2. Write Haskell function to –

a. Print 'Hello World'

Program:

```

main::IO()

main = do
    putStrLn "Hello World"

```

Output:

```

Haskell - "C:\Program Files\Hask
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l helloworld.hs
[1 of 1] Compiling Main                ( helloworld.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello World

```

b. Add two numbers (both int and float)

Program:

```
addInt :: Int->Int->Int
addInt x y = x + y
```

Output:

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l addInt.hs
[1 of 1] Compiling Main                ( addInt.hs, interpreted )
Ok, one module loaded.
*Main> addInt 123 678
801
*Main> |
```

Program:

```
addFloat :: Float->Float->Float
addFloat x y = x + y
```

Output:

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l addFloat.hs
[1 of 1] Compiling Main                ( addFloat.hs, interpreted )
Ok, one module loaded.
*Main> addFloat 1.23 6.78
8.01
*Main> |
```

c. Subtract two numbers (both int and float)

Program:

```
subtractInt :: Int->Int->Int
subtractInt x y = x - y
```

Output:

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l subtractInt.hs
[1 of 1] Compiling Main                ( subtractInt.hs, interpreted )
Ok, one module loaded.
*Main> subtractInt 678 123
555
*Main> |
```

Program:

```
subtractFloat :: Float->Float->Float
subtractFloat x y = x - y
```

Output:

```

GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l subtractFloat.hs
[1 of 1] Compiling Main                ( subtractFloat.hs, interpreted )
Ok, one module loaded.
*Main> subtractFloat 6.78 1.23
5.55
*Main> |

```

d. Exponent of two numbers

Program:

```

expo :: Float -> Float -> Float
expo x y = x**y

```

Output:

```

GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l expo.hs
[1 of 1] Compiling Main                ( expo.hs, interpreted )
Ok, one module loaded.
*Main> expo 2 8
256.0
*Main> |

```

e. Square root of a number

Program:

```

sqr :: Float -> Float
sqr x = x**0.5

```

Output:

```

GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l sqrt.hs
[1 of 1] Compiling Main                ( sqrt.hs, interpreted )
Ok, one module loaded.
*Main> sqrt 64
8.0
*Main> |

```

8. Post Experimental Exercise

Questions:

1. List important characteristics of functions in Haskell

Functions play a major role in Haskell, as it is a functional programming language.

First, consider this definition of a function which adds its two arguments:

```
add :: Integer -> Integer -> Integer
```

```
add x y = x + y
```

This is an example of a curried function

The name curry derives from the person who popularized the idea: Haskell Curry. To get the effect of an uncurried function, we could use a tuple, as in:

```
add (x,y) = x + y
```

But then we see that this version of add is really just a function of one argument. An application of add has the form $\text{add } e_1 \ e_2$, and is equivalent to $(\text{add } e_1) \ e_2$, since function application associates to the *left*. In other words, applying add to one argument yields a new function which is then applied to the second argument. This is consistent with the type of add, $\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$, which is equivalent to $\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$; i.e. \rightarrow associates to the *right*. Indeed, using add, we can define inc in a different way from earlier:

```
inc = add 1
```

This is an example of the partial application of a curried function, and is one way that a function can be returned as a value. Let's consider a case in which it's useful to pass a function as an argument. The well-known map function is a perfect example:

```
map :: (a->b) -> [a] -> [b]
```

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

[Function application has higher precedence than any infix operator, and thus the right-hand side of the second equation parses as $(f \ x) : (\text{map } f \ xs)$.] The map function is polymorphic and its type indicates clearly that its first argument is a function; note also that the two a's must be instantiated with the same type (likewise for the b's). As an example of the use of map, we can increment the elements in a list:

```
map (add 1) [1,2,3] => [2,3,4]
```

These examples demonstrate the first-class nature of functions, which when used in this way are usually called higher-order functions.

2. Write the lambda calculus abstraction functions for-

▪ Unity operation

```
( $\lambda x. x$ ) a
```

```
a
```

x is the formal parameter, function body is x, the function is $\lambda x. x$, argument to this function is 'a'.

▪ Addition operation

```
(( $\lambda x. (\lambda y. x+y)$ ) a) b)
```

```
( $\lambda y. a+y$ ) b
```

```
a+b
```

The outermost function is $\lambda x. (\lambda y. x+y)$, where x is the formal parameter, $\lambda y. x+y$ is the function body, 'a' is the argument to the function. For the function $\lambda y. x+y$ the formal parameter is y and

the function body is $x+y$, and the argument to this function is 'b'.

▪ **Exponential operation**

```
((λx. (λy.x**y) a) b)
(λy.a**y)b
a**b
```

The outermost function is $\lambda x. (\lambda y.x**y)$, where x is the formal parameter, $\lambda y.x**y$ is the function body, 'a' is the argument to the function. For the function $\lambda y.x**y$ the formal parameter is y and the function body is $x**y$, and the argument to this function is 'b'.

▪ **Square-root operation**

```
(λx.x**0.5)a
a**0.5
```

Here the function is $\lambda x.x**0.5$, function body is $x**0.5$, formal parameter is x , and the argument to this function is 'a'.

▪ **Multiplication by 2 operation**

```
(λx.x*2) a
a*2
```

Here the function is $\lambda x.x*2$, function body is $x*2$, formal parameter is x , and the argument to this function is 'a'.

3. Write Haskell function to-

- Add two numbers and find their square root

```
addSqrt :: Float -> Float -> Float
addSqrt x y = (x+y)**0.5
```

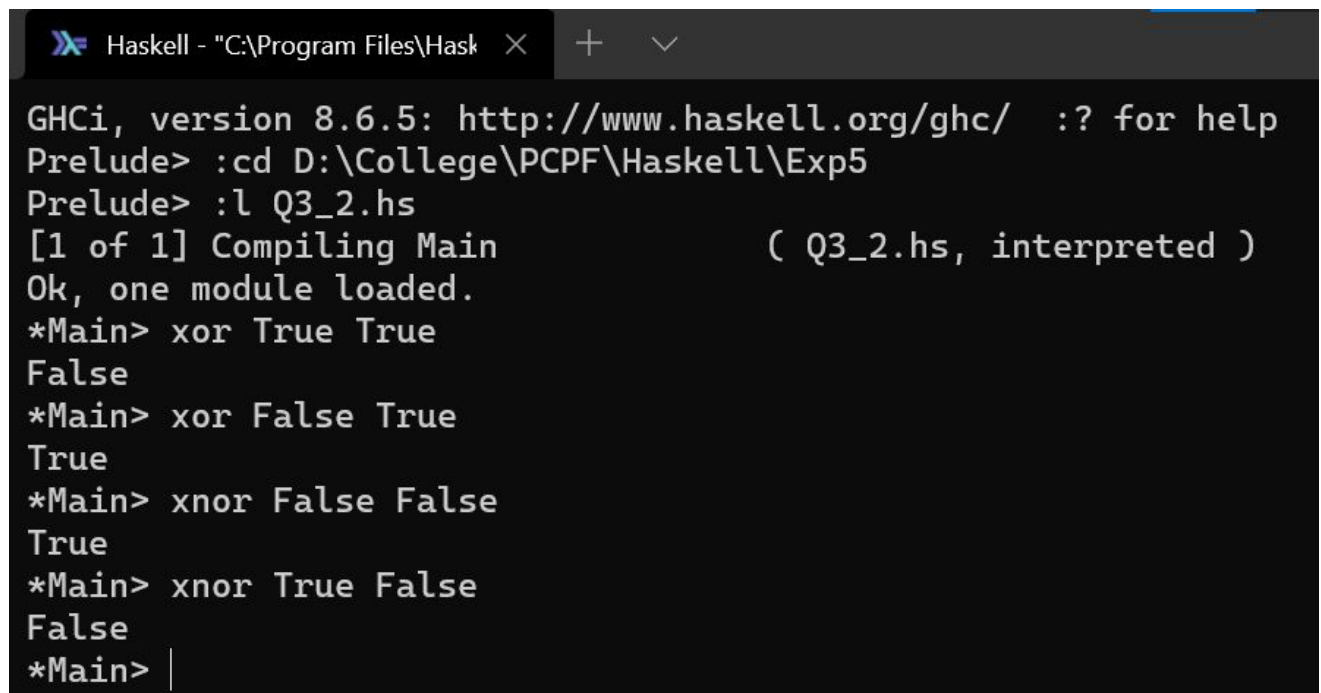
```
Haskell - "C:\Program Files\Hask
+  v

GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp5
Prelude> :l Q3_1.hs
[1 of 1] Compiling Main                ( Q3_1.hs, interpreted )
Ok, one module loaded.
*Main> addSqrt 12 4
4.0
*Main> |
```

- Perform XNOR and XOR operation

```
xor :: Bool -> Bool -> Bool
xor x y | x==True && y ==True = False
        | x==False && y ==False = False
        | otherwise = True

xnor :: Bool -> Bool -> Bool
xnor x y | x==True && y ==True = True
        | x==False && y ==False = True
        | otherwise = False
```

```
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\Exp5
Prelude> :l Q3_2.hs
[1 of 1] Compiling Main                ( Q3_2.hs, interpreted )
Ok, one module loaded.
*Main> xor True True
False
*Main> xor False True
True
*Main> xnor False False
True
*Main> xnor True False
False
*Main> |
```

Conclusion:

Functions play a major role in Haskell, as it is a functional programming language. Functional programming languages are specially designed to handle symbolic computation and list processing applications. In this experiment we have written programs to implement various functions in Haskell.

To perform this experiment Visual Studio Code was used as a code editing software and Glasgow Haskell compiler was used to compile the codes

From this experiment we infer how functions are implemented in Haskell language. We have also studied the various properties of functions in Haskell like currying, partial functions, lambda expressions, higher order functions, etc.

References:

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

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

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