Experiment No: 02

Aim: Tutorial Compile and run Haskell Program.

Lab Objective: Design and implement declarative programs in functional and logic programming languages.

Lab Outcomes: Design and Develop solution based on declarative programming paradigm using functional and logic programming (LO2)

Requirements: Any Text Editor and Glasgow Haskell Compiler 8.0+ Version

Theory: Haskell is a purely functional programming language. We can trace roots of Functional Programming concepts in Lambda Calculus theory proposed by mathematician Alonzo Church in 1930s. Lambda Calculus (also written as λ – calculus) is formal system in Mathematical Logic for expressing computation on the basis of function abstraction and function application using variable binding and substitution.

The λ calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines. However, the λ calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

The central concept in λ calculus is the "expression". A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters a, b, c, . . .

An expression is defined recursively as follows:

<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>

An expression can be surrounded with parenthesis for clarity, that is, if E is an expression, (E) is the same expression. The only keywords used in the language are λ and the dot. In order to avoid cluttering expressions with parenthesis, we adopt the convention that function application associates from the left, that is, the expression E_1 E_2 E_3 . . . E_n is evaluated applying the expressions as follows: (. . . ((E1 E2)E3) . . . En)

As can be seen from the definition of λ expressions given above, a single identifier is a λ expression. An example of a function is λx.x This expression defines the identity function. The name after the λ is the identifier of the argument of this function. The expression after the point (in this case a single x) is called the "body" of the definition. Functions can be applied to expressions. An example of an application is (λx.x)y this is the identity function applied to y. Parenthesis are used for clarity in order to avoid ambiguity. Function applications are evaluated by substituting the value of the argument x (in this case with y) in the body of the function definition, i.e. (λx.x)y = [y/x]x = y

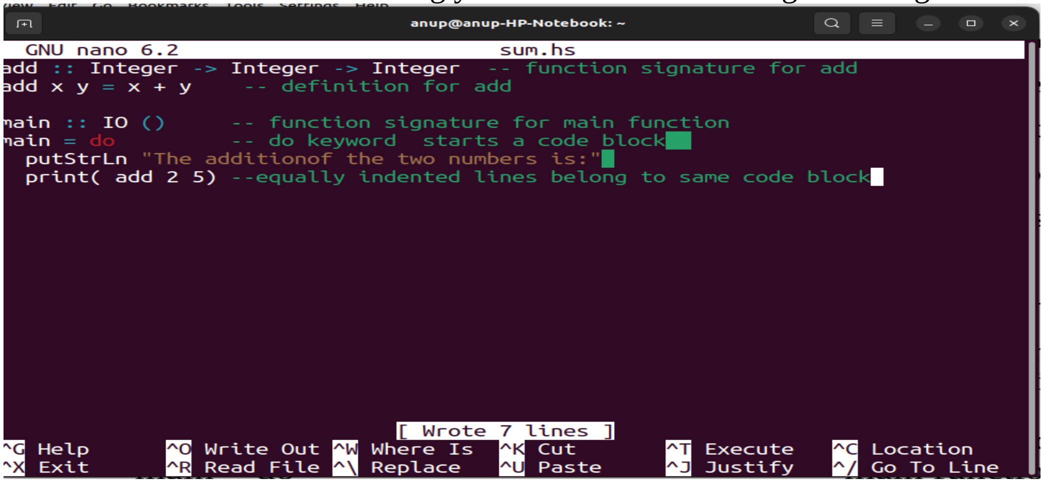
In this transformation the notation [y/x] is used to indicate that all occurrences

of x are substituted by y in the expression to the right. The names of the arguments in function definitions do not carry any meaning by themselves. They are just “place holders”, that is, they are used to indicate how to rearrange the arguments of the function when it is evaluated. We use the symbol “≡” to indicate that when A ≡ B, A is just a synonym of B.
To summarize

Performance: [Note: While writing the write up student need to change the wording such that it coveys that students have done all following steps. Also where ever output is generated the output must be written by the student]
Students need to perform following steps to complete this Experiment:

1. Understand how to create Haskell code with main function and run it in ghci
2. Understand how to compile and execute haskell code having main using ghc compiler
3. Understand more standard Prelude functions

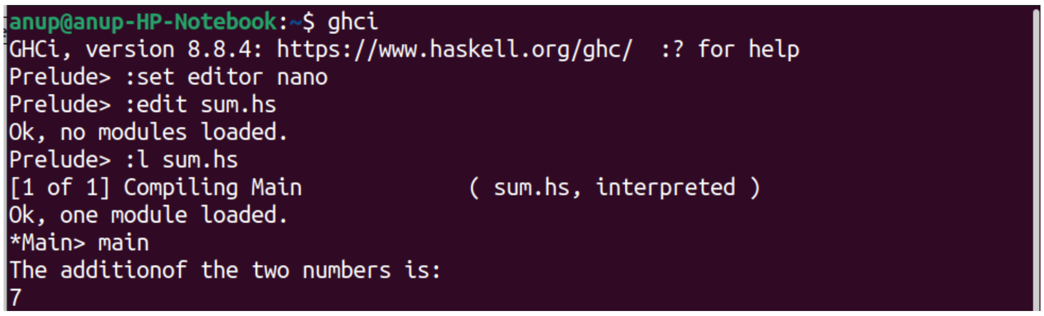
Create a file named : sum.hs using your favorite editor having following code



```
anup@anup-HP-Notebook: ~
  GNU nano 6.2                         sum.hs
add :: Integer -> Integer -> Integer   -- function signature for add
add x y = x + y      -- definition for add

nain :: IO ()       -- function signature for main function
nain = do           -- do keyword  starts a code block
  putStrLn "The additionof the two numbers is:"
  print( add 2 5) --equally indented lines belong to same code block




                                       [ Wrote 7 lines ]
^G Help         ^O Write Out ^W Where Is  ^K Cut        ^T Execute   ^C Location
^X Exit         ^R Read File ^\ Replace   ^U Paste      ^J Justify   ^/ Go To Line
```

Part 1. Running sum.hs in ghci

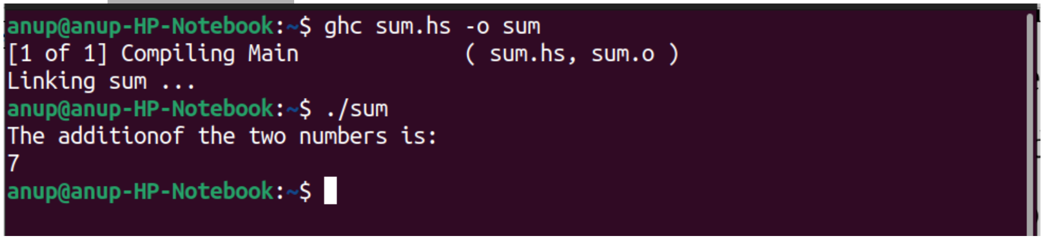
1. Open ghci and set the editor to nano using command
   Prelude> :set editor nano <*enter*>
2. Open file sum.hs for editing and type above code in nano editor, save the buffer using CTRL+O and exit the editor.
3. To load the file use command
   Prelude> :load sum.hs<*enter*>
4. If successfully compiled just run main code block using command
   *Main> main<*enter*>
5. Note the output.



```
anup@anup-HP-Notebook:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> :set editor nano
Prelude> :edit sum.hs
Ok, no modules loaded.
Prelude> :l sum.hs
[1 of 1] Compiling Main             ( sum.hs, interpreted )
Ok, one module loaded.
*Main> main
The additionof the two numbers is:
7
```

Part 2. Compile and execute sum.hs using ghc compiler

1. Consider you have already above code in sum.hs file
2. Open terminal and go do folder that has file sum.hs
3. To compile sum.hs to sum binary executable type
   $ *ghc sum.hs -o sum <enter>*
4. To exucute code sum type commands
   $./sum<enter>



```
anup@anup-HP-Notebook:~$ ghc sum.hs -o sum
[1 of 1] Compiling Main             ( sum.hs, sum.o )
Linking sum ...
anup@anup-HP-Notebook:~$ ./sum
The additionof the two numbers is:
7
anup@anup-HP-Notebook:~$ 
```

Part 3. Understand use of standard Prelude functions

Execute following Haskell prelude library functions in ghci and note the output create a table having four columns name of function, use, output and explanation. [Write this on unruled pages]

1. succ 6	2. succ (succ 6)	3. min 5 6
4. max 5 6	5. max 101 101	6. succ 9 + max 5 4 + 1
7. (max 5 4)+(succ 9)+1	8. (succ 9) + (max 5 4) + 1	10. succ 9*10
11. succ (9*10)	12. div 92 10	13. div 3 4
14. mod 7 5	15. x=45	16. print x
17. x=45.5	18. print x	19. x = “hi”
20. print x	21. putStrLn “Hello”	22. print “Helllo”

Exersize: Write following codes and run it using ghci or ghc:

a. print Hello World Message
b. add two numbers either both Int or Float or mixed
c. subtract two numbers either both Int or Float or mixed
d. multiply two numbers either both Int or Float or mixed
e. find square root of a number

Conclusion: Thus we have learned how to compile and run Haskell code using ghci and ghc.

Reference:

1. Glasgow Haskell Project Home Page. https://www.haskell.org/
2. Learn You a Haskell for Great Good ! A Beginner's Guide http://learnyouahaskell.com/
3. Michael L Scott, ‘Programming Language Pragmatics’, 3rd Edition, Elsevier Publication.
4. Introduction to Lambda Calculus,