Experiment Number: 04

Aim: Tutorial on introduction to how to compile and execute Haskell code

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

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

Requirements: Any text editor to be able to edit Haskell code and Glasgow Haskell Compiler 8.0+ version.

Theory:

Hakell is a purly functional programming language. We can trace many of the concepts in functional programming to the theorotical basis of Lambda Calculus proposed by the mathematician Alonzo Church in the 1930s.

Lambda calculus (also written as **λ-calculus**) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

Lambda calculus consists of constructing lambda terms and performing reduction operations on them. In the simplest form of lambda calculus, terms are built using only the following rules:

Syntax	Name	Description
x	Variable	A character or string representing a parameter or mathematical/logical value.
(λ*x*.*M*)	Abstraction	Function definition (*M* is a lambda term). The variable *x* becomes bound in the expression.
(*M N*)	Application	Applying a function to an argument. M and N are lambda terms.

The reduction operations include:

Operation	Name	Description
(λ*x*.*M*[*x*]) → (λ*y*.*M*[*y*])	α-conversion	Renaming the bound variables in the expression. Used to avoid name collisions.
((λ*x*.*M*) *E*) → (*M*[*x* := *E*])	β-reduction	Replacing the bound variables with the argument expression in the body of the abstraction.

Functions in Haskell:

Functions in Hakell are first class members that is one can work with functions as objects that can be passes on to and returned from other functions. This leads to many advaced features of Haskell like higher order functions, compositions of functions and anonymous (lambda) functions etc. Haskell has a strong type system which can imply the types of input output parameters of a function, still it is good practice that progammar specifies the type signature while defining a function in Haskell.

Consider following Haskell code having 2 functions defined in a sum.hs file

```
add :: Integer -> Integer -> Integer            --function signature declaration
add x y = x + y                                 --function definition

main:: IO ()                                     --main function signature
main = do                                       --main function definition
       putStrLn "The addition of the two numbers is:"
       print(add 2 5)                           --evaluation of a function
```

Line number 1 and are respectively the signatures for functions add and main. Type signatures of function reads function named add has type Integer data mapped to Integer data mapped to Integer. This indicates the add function is a binary function that accepts two Integer arguments and returns an Integer value. This double mapping is due to the fact that in haskell functions are curried i.e. arguments of functions are applied one at a time to the mapping from right to left.

e.g.

add 5 4 will be evaluated in two steps first argument 4 is bound to variable y to produce partial application x+4. In this partial application we then bound the second argument x with 5 to generate 5+4 which is evaulated as 9 which is returned as answer to application of 5 4 to add function.

Performance steps:

Part 1: Learn how to write **Haskell program cotaining a function and run it:**

1. First we will type above code for add function in a add.hs file.

   To do so execute

   a. Execute on terminal prompt **nano add.hs**

   b. Type the code, to save the file with name add.hs use CTRL+x when propted say y and press enter.

   c. Check if file content is correct using command **cat add.hs**

2. To run the code in add.hs execute command **runghc add.hs** and chek if output is corectly obtained.

3. To genearate executable file with above haskell code execute **ghc add.hs -o add**

name of the executable is add which can be run as **./add**

Part 2: Learn how to evaluate standard prelude functions in ghci

To complete this section student need to start ghci and execute folloing function, note output and explain what does the function do i.e. create a three column table.

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”

Part 3: Write Haskell Program to do following

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: We leared how to compile and execute Haskell code

Reference:

[1] Glasgow Haskell project home page, https://www.haskell.org/
[2] Learn you Hakell for greate good. A biginner’s guide, http://learnyouahaskell.com/
[3] Michael L Scott, “ Programming Language Pragmatics”, Third edition, Elsevier publication