Name: SATYAJIT GHANA Registration Number: 17ETCS002159

Laboratory 6

Title of the Laboratory Exercise: Introductory exercises in Haskell

1. Introduction and Purpose of Experiment

   Students get familiar with the Haskell platform using a set of simple exercises.

2. Aim and Objectives

   Aim

   - To develop Haskell programs using GHCi

   Objectives

   At the end of this lab, the student will be able to

   - Use GHCi to develop Haskell programs
   - Develop Haskell programs edit and execute them successfully

3. Experimental Procedure

   Students are given a set of instructions to be executed on the computer. The instructions should be edited and executed and documented by the student in the lab manual. They are expected to answer questions posed in section 5 based on their experiment.

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

Documentation:	
a. Procedure and Algorithm(s):	**Procedure:** 1. Creating a new Haskell File Haskell source files are ASCII based text file with a `.hs` extension. The execution of the Haskell Project usually begins from `Main.hs` and hence we name the file Main.hs, it can be anything if its going to a simple project, for the sake of organization the file that contains the driver function or main function is inside the Main.hs. 2. Design the functions required Since Haskell is a functional programming language, the required functions need to be decomposed into sub functions, the operations that needs to be done in the function also needs to be purely functional, and side-effects needs to be avoided as much as possible. 3. Documentation Write documentation for the functions that are used in the source file along with its input and outputs, write the function signatures to enforce typing in Haskell. 4. Running and testing The functions written can be individually tested in GHCI, which is an interactive version of the Glasgow Haskell Compiler, this will us know test atomic functions, which are a part of more functions, and make it easier to debug the programs, since there are no side-effects testing becomes easier. Load the `.hs` file into GHCI by typing `:l Main.hs` , this will compile and load the function written in the file and each of the function can be called specifically by just typing the name of the function, to see the type of function `:t` can be used.
b. Conclusions :	Haskell is an Interpreted Language which uses gcc as its backend to compile to machine code. Through this experiment, the use of GHCi

	shell was learnt. The knowledge of creating user defined functions and running then in GHCi was learnt. HUnit testing can be used by software developers to test and debug Haskell code, since we have already done functional decomposition it is easy to debug such functions.

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

Results and Discussions:

Screenshot:

```haskell
numericOperations :: IO ()
numericOperations = do
    putStr "2+15 = "
    print $ 2+15
    putStr "49*100 = "
    print $ 49*100
    putStr "5/2 = "
    print $ 5/2
    putStr "50*(100-4999) = "
    print $ 50 * (100 - 4999)

logicalOperations :: IO ()
logicalOperations = do
    putStr "True && False = "
    print $ True && False
    putStr "True && True = "
    print $ True && True
    putStr "False || True = "
    print $ False || True
    putStr "not False = "
    print $ not False
    putStr "not (True && True) = "
    print $ not (True && True)

comparisonOperations :: IO ()
comparisonOperations = do
    putStr "5 == 5 = "
    print $ 5 == 5
    putStr "1 == 0 = "
    print $ 1 == 0
    putStr "5 /= 0 = "
    print $ 5 /= 4
    putStr "hello == hello = "
    print $ "hello" == "hello"
```

OUTPUT :

```
*Main> numericOperations
2+15 = 17
49*100 = 4900
5/2 = 2.5
50*(100-4999) = -244950

*Main> logicalOperations
True && False = False
True && True = True
False || True = True
not False = True
not (True && True) = False
```

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

```
*Main> comparisonOperations
5 == 5 = True
1 == 0 = False
5 /= 0 = True
hello == hello = True
```

Discussion:

Here we demonstrate the numeric, logical and comparison operations built in to Haskell.

1. numericOperations

numeric operations are in-built in Haskell, and since it's an interpreted language it can be directly typed into the GHCi and the output is displayed in the REPL terminal. To display the result the `print` function is used that converts the output to a string and displays it in the terminal. `putStr` is a function that prints a string to the terminal.

2. logicalOperations

logical operations such as and, or are represented as `(&&)`, `(||)` and not is represented as `not`.

3. comparsionOperations

string, numeric type comparisons can be made using the equals `(==)` and the not equals `(/=)` operations, which have the following type signatures respectively,

```
Prelude> :i (==)
class Eq a where
  (==) :: a -> a -> Bool
  ...
        -- Defined in 'GHC.Classes'
infix 4 ==
Prelude> :i (/=)
class Eq a where
  ...
  (/=) :: a -> a -> Bool
        -- Defined in 'GHC.Classes'
infix 4 /=
```

numbers, lists can be compared using these operators.

Screenshot:

```haskell
callingFunctions :: IO ()
callingFunctions = do
    putStr "succ 8 = "
    print $ succ 8
    putStr "min 9 10 = "
    print $ min 9 10
    putStr "min 3.4 3.2 = "
    print $ min 3.4 3.2
    putStr "max 100 101 = "
```

```
    print $ max 100 101
    putStr "succ 9 + max 5 4 + 1 = "
    print $ succ 9 + max 5 4 + 1
    putStr "92 div 10 = "
    print $ 92 `div` 10

doubleMe :: Num a => a -> a
doubleMe x = x + x

doubleUs :: Num a => a -> a -> a
doubleUs x y = doubleMe x + doubleMe y

doubleSmallNumber :: (Num a, Ord a) => a -> a
doubleSmallNumber x = if x > 100 then x else x + 2
```

OUTPUT:

```
*Main> callingFunctions
succ 8 = 9
min 9 10 = 9
min 3.4 3.2 = 3.2
max 100 101 = 101
succ 9 + max 5 4 + 1 = 16
92 div 10 = 9
*Main> doubleMe 2
4
*Main> doubleUs 2 3
10
*Main> doubleSmallNumber 101
101
```

Discussion:

Function can be called in Haskell by providing the function name along with the parameters the function expects. succ is a function that returns the next successive number, we pass the parameter 8 to it, and it returns 9, which is the successor to 8. min is a function that expects two parameters and returns the minimum of the two numbers, we pass 9 and 10 to it and the minimum of the two, which is 9 is the result obtained. max is similar and returns the maximum of the two numbers.

doubleMe is a user defined function that returns the double of a number, it expects a argument of type Num and returns a Num. if we call doubleMe with parameter 2, we get 4 as the result.

doubleUs is defined as a function that expects two arguments, both of type Num and returns the sum of the double of the two parameters passed. Here we pass 2 3 as the parameters to the doubleUs function and it returns (2+2)+(3+3) which is 10.

doubleSmallNumber is a function that expects an argument of type Num which is comparable to other Num, if the number is greater than 100 then the same number is returned, else the double of the number is returned, here we pass 101 as the argument and since it is greater than 100, the same number is returned back.

Screenshot:

```
listOperations :: IO ()
listOperations = do
    header
    let myList = [x | x <- [1..100], x `mod` 7 == 0]
    putStr "myList = "
    print myList
    putStrLn "Concatenate a element at the BEGINNING of List using (:)"
    print . show $ 10:myList
    putStrLn "Concatenate two Lists using (++)"
    print . show $ myList++[3, 5..15]
    putStrLn "Accessing List Elements"
    print . show $ myList
    putStrLn $ "4th Element in List is "++ show (myList!!4)
    putStrLn $ "head myList = " ++ show (head myList)
    putStrLn $ "tail myList = " ++ show (tail myList)
    putStrLn $ "last myList = " ++ show (last myList)
    putStrLn $ "init myList = " ++ show (init myList)
    putStrLn $ "length myList = " ++ show (length myList)
    putStrLn $ "reverse myList = " ++ show (reverse myList)
    putStrLn "Comparing Lists"
    putStr "[3, 2, 1] > [2, 1, 0] = "
    print $ [3, 2, 1] > [2, 1, 0]
    putStr "[3, 4, 2] < [3, 4, 3] = "
    print $ [3, 4, 2] < [3, 4, 3]
    putStr "[3, 4, 2] == [3, 4, 2] = "
    print $ [3, 4, 2] == [3, 4, 2]

nestedLists :: IO()
nestedLists = do
    let xxs = [[1,3,5,2,3,1,2,4,5], [1..9],[1,2]]
    print . show  $ xxs
    putStrLn "Only Evens from the nested List "
    print . show  $ [[x | x <- xs, even x] | xs <- xxs]

removeNonUpperCase :: String -> String
removeNonUpperCase st = [c | c <- st, c `elem` ['A'..'Z']]
```

OUTPUT:

```
*Main> listOperations
--------------------------------------------------
myList = [7,14,21,28,35,42,49,56,63,70,77,84,91,98]
```

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

```
Concatenate a element at the BEGINNING of List using (:)
"[10,7,14,21,28,35,42,49,56,63,70,77,84,91,98]"
Concatenate two Lists using (++)
"[7,14,21,28,35,42,49,56,63,70,77,84,91,98,3,5,7,9,11,13,15]"
Accessing List Elements
"[7,14,21,28,35,42,49,56,63,70,77,84,91,98]"
4th Element in List is 35
head myList = 7
tail myList = [14,21,28,35,42,49,56,63,70,77,84,91,98]
last myList = 98
init myList = [7,14,21,28,35,42,49,56,63,70,77,84,91]
length myList = 14
reverse myList = [98,91,84,77,70,63,56,49,42,35,28,21,14,7]
Comparing Lists
[3, 2, 1]  > [2, 1, 0] = True
[3, 4, 2]  < [3, 4, 3] = True
[3, 4, 2] == [3, 4, 2] = True

*Main> nestedLists
"[[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2]]"
Only Evens from the nested List
"[[2,2,4],[2,4,6,8],[2]]"

*Main> removeNonUpperCase "IdontLIKEFROGS"
"ILIKEFROGS"
```

Discussion:

Lists in Haskell are homogeneous data types, i.e. they can strictly contain data of the same types. Lists are surrounded by square brackets and the values are separated by commas.

List Comprehension is a way to filter, compare and combine lists, here we generate a list using the list generator from 1 to 100, with increment of 1, and filter those numbers which are divisible by 7.

To push an element to the list the "cons" or the `(:)` operator is used. Here we push the number 10 to the already generated list above.

To concatenate two lists, we use the `(++)` operator, this combines the two lists and returns a list which has elements of both the lists.

There are some other useful functions on list such as head, that returns the first element in the list. Tail returns the rest of the elements of the list except the first element. Init return the list with the last element removed, and last returns the last element of the list.

Lists can also be compared using the comparison operators, > , < , ==, and /=.

Nested lists can be created using a list inside a list, list operators can be performed on these also.

Screenshot:

```
zipping :: [(Integer, String)]
zipping = zip [1..] ["apple", "orange", "cherry", "mango"]
```

OUTPUT:

```
*Main> zipping
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
*Main> fst (1, "apple")
1
*Main> snd (1, "apple")
"apple"
*Main> zip [1] ["apple"]
[(1,"apple")]
*Main>
```

Discussion:

Another data type in Haskell is a Tuple, which is a fixed size data structure with heterogeneous data elements, here we have two separate list of data, first one is a infinite list of integers starting from 1, and the second one is a list of strings which are fruit name, we would like to create a new list with tuples with every nth element of list 1 mapped to the nth element of list 2, so we use the function zip, which does exactly this. Some other functions related to tuples is `fst` and `snd`, if there is a tuple with just 2 elements then `fst` returns the first element in the tuple and `snd` returns the second element in the tuple. In `(1, "apple")`, 1 is the first element and "apple" is the second element.

Screenshot:

```
findRightTriangle :: [(Integer, Integer, Integer)]
findRightTriangle = [(a, b, c)
    | a <- [1..10], b <- [1..10], c <- [1..10],
    c^2 == a^2 + b^2]
```

OUTPUT:

```
*Main> findRightTriangle
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
*Main>
```

Discussion:

Name: SATYAJIT GHANA Registration Number: 17ETCS002159

`findRightTriangle` is a function that takes no arguments and returns a list of tuples with 3 integers which are the lengths of a right-angle triangle. To do this we use list comprehension to generate this list, the comprehension has the generator part which decides which all values the list can take and the predicate which filters the values that are required, here we generate values from 1 to 10 and assign it to a, b, c. the predicate here is that the sum of squares of a and b should equal the square of c, this will filter only those values that form a right-angled triangle.