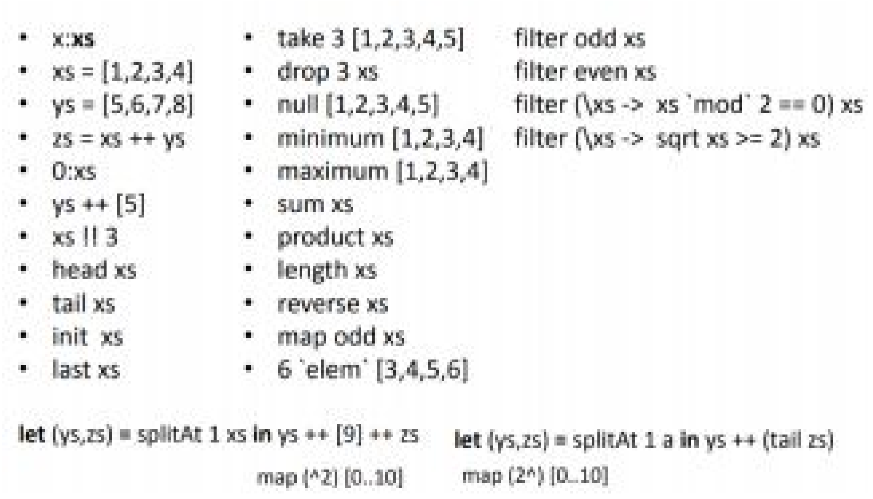
Name: Yash Mahajan Class: SE IT B Roll No.:04 PID:191061

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

Experiment – 6

1. Aim: To implement list comprehensions using Haskell Programming
Program Code

1. Implement the following list comprehensions using Haskell compiler at prelude prompt



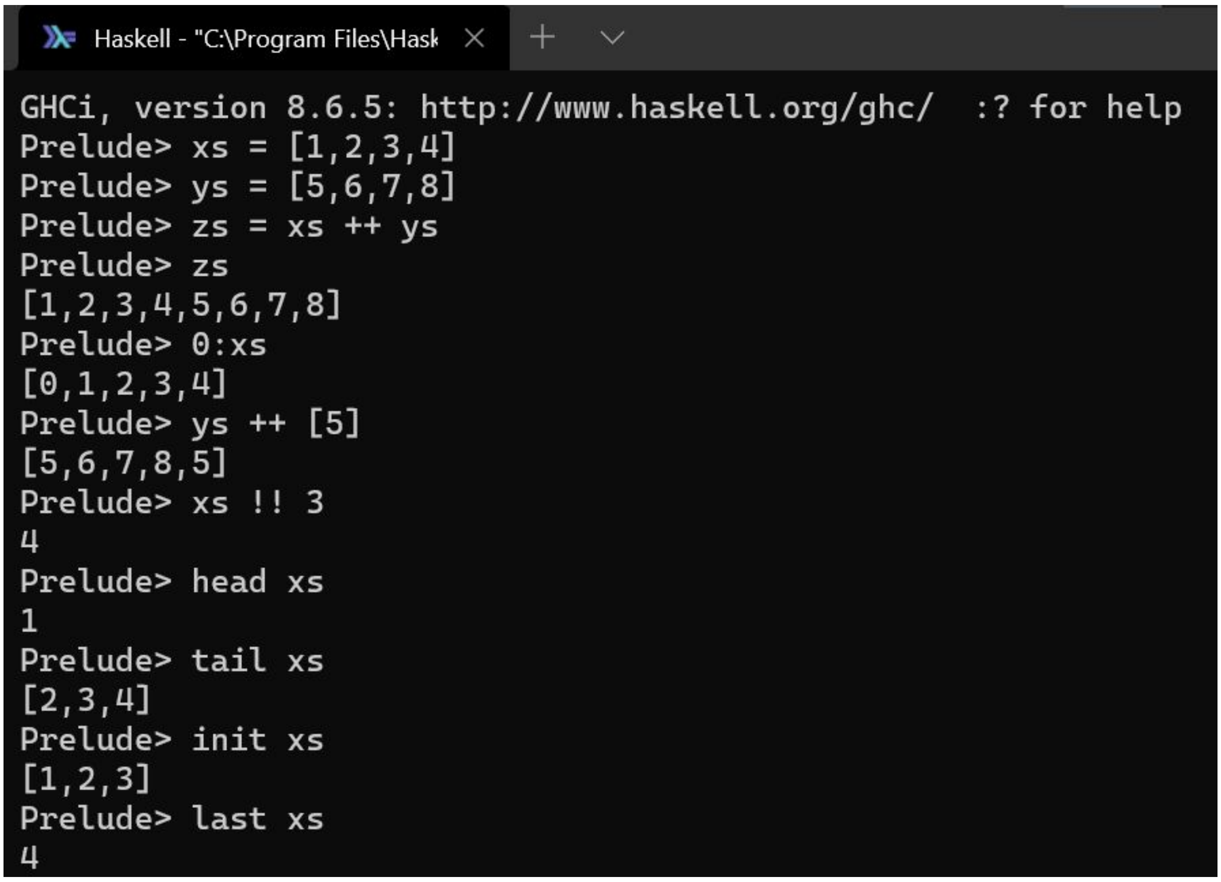
- x:xs
- xs = [1,2,3,4]
- ys = [5,6,7,8]
- zs = xs ++ ys
- 0:xs
- ys ++ [5]
- xs !! 3
- head xs
- tail xs
- init xs
- last xs

- take 3 [1,2,3,4,5]
- drop 3 xs
- null [1,2,3,4,5]
- minimum [1,2,3,4]
- maximum [1,2,3,4]
- sum xs
- product xs
- length xs
- reverse xs
- map odd xs
- 6 `elem` [3,4,5,6]

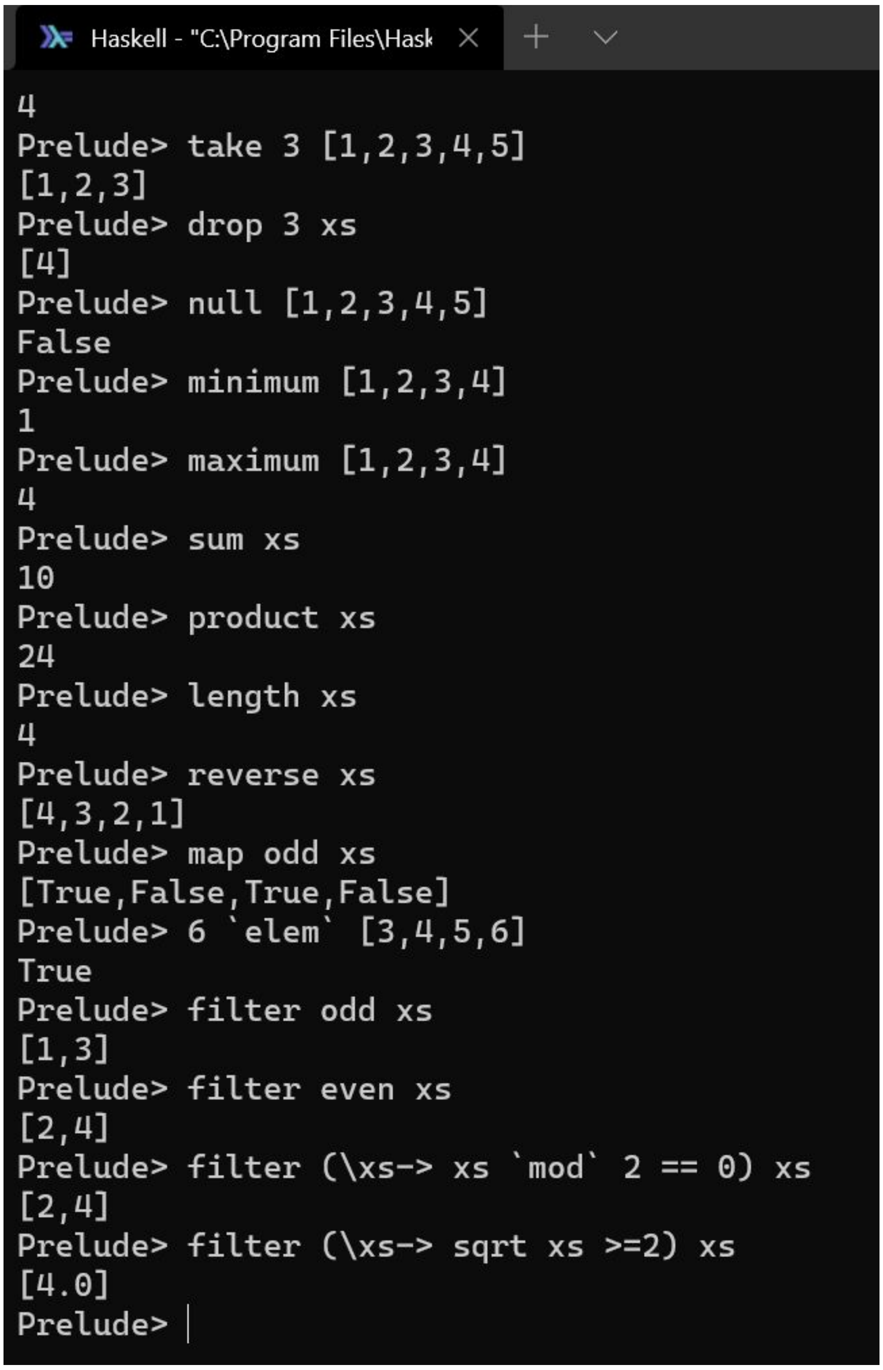
```
filter odd xs
filter even xs
filter (\xs -> xs `mod` 2 == 0) xs
filter (\xs -> sqrt xs >= 2) xs
```

```
let (ys,zs) = splitAt 1 xs in ys ++ [9] ++ zs
let (ys,zs) = splitAt 1 a in ys ++ (tail zs)
map (^2) [0..10]
map (2^) [0..10]
```

Output:



```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> xs = [1,2,3,4]
Prelude> ys = [5,6,7,8]
Prelude> zs = xs ++ ys
Prelude> zs
[1,2,3,4,5,6,7,8]
Prelude> 0:xs
[0,1,2,3,4]
Prelude> ys ++ [5]
[5,6,7,8,5]
Prelude> xs !! 3
4
Prelude> head xs
1
Prelude> tail xs
[2,3,4]
Prelude> init xs
[1,2,3]
Prelude> last xs
4
```



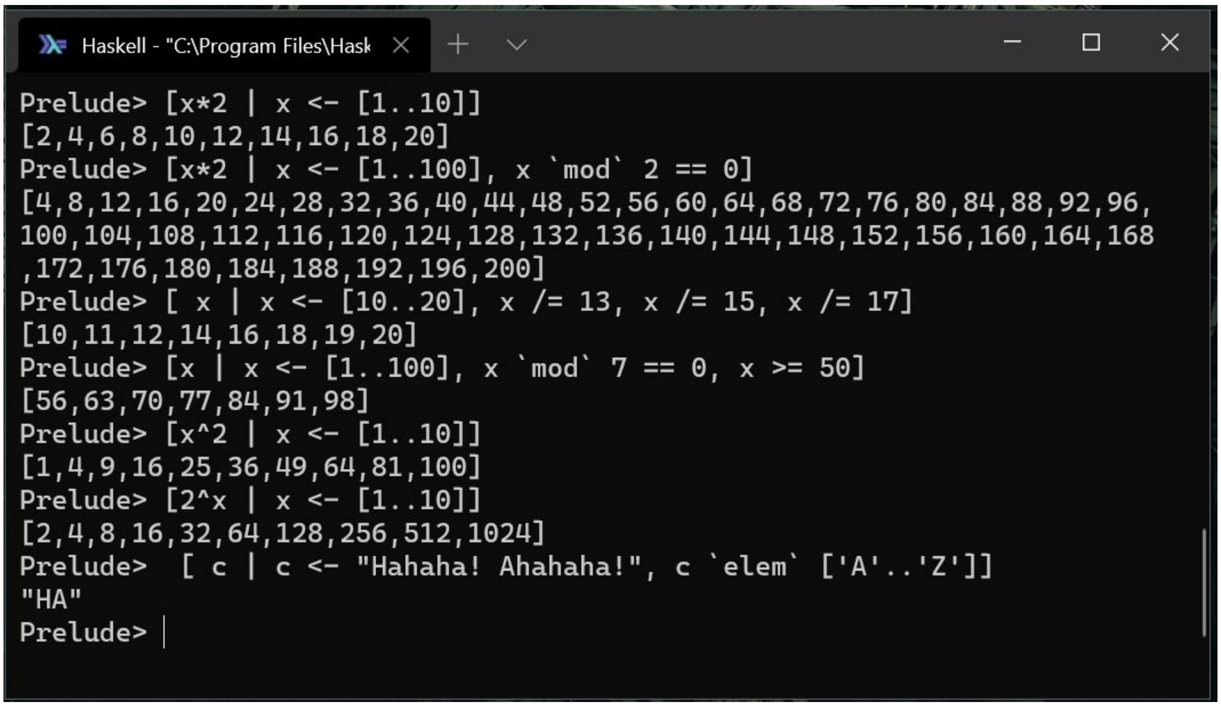
```
4
Prelude> take 3 [1,2,3,4,5]
[1,2,3]
Prelude> drop 3 xs
[4]
Prelude> null [1,2,3,4,5]
False
Prelude> minimum [1,2,3,4]
1
Prelude> maximum [1,2,3,4]
4
Prelude> sum xs
10
Prelude> product xs
24
Prelude> length xs
4
Prelude> reverse xs
[4,3,2,1]
Prelude> map odd xs
[True,False,True,False]
Prelude> 6 `elem` [3,4,5,6]
True
Prelude> filter odd xs
[1,3]
Prelude> filter even xs
[2,4]
Prelude> filter (\xs-> xs `mod` 2 == 0) xs
[2,4]
Prelude> filter (\xs-> sqrt xs >=2) xs
[4.0]
Prelude>
```

From the output we can study the different operations and functions that can be performed on a list in Haskell.

2. Perform the following list comprehensions using Haskell compiler at the prelude prompt and explain each comprehension in your own words

a. [x*2 | x <- [1..10]]
b. [x*2 | x <- [1..100], x ‘mod’ 2 == 0]
c. [x | x <- [10..20], x /= 13, x /= 15, x /= 17]
d.[x | x <- [1..100], x ‘mod’ 7 == 0, x >= 50]
e. [x^2 | x <- [1..10]]
f. [2^x | x <- [1..10]]
g. [c | c <- “Hahaha! Ahahaha!", c `elem` ['A'..'Z']]

Name: Yash Mahajan Class: SE IT B Roll No.:04 PID:191061



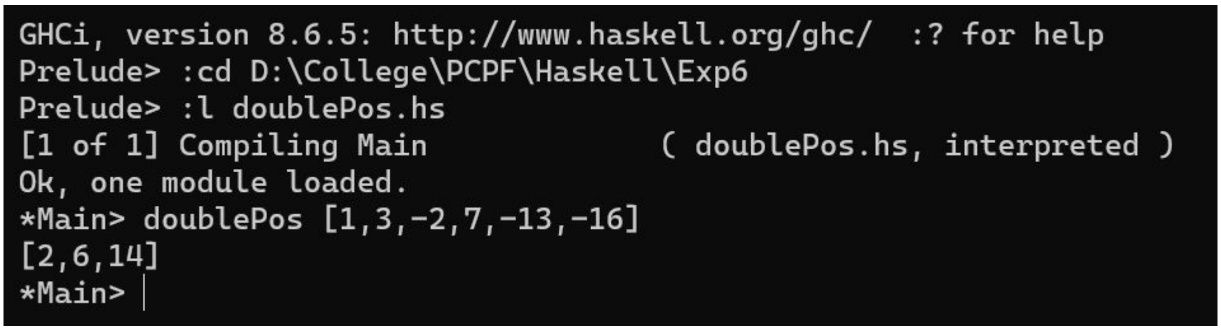
```
Prelude> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x*2 | x <- [1..100], x `mod` 2 == 0]
[4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96,
100,104,108,112,116,120,124,128,132,136,140,144,148,152,156,160,164,168
,172,176,180,184,188,192,196,200]
Prelude> [ x | x <- [10..20], x /= 13, x /= 15, x /= 17]
[10,11,12,14,16,18,19,20]
Prelude> [x | x <- [1..100], x `mod` 7 == 0, x >= 50]
[56,63,70,77,84,91,98]
Prelude> [x^2 | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
Prelude> [2^x | x <- [1..10]]
[2,4,8,16,32,64,128,256,512,1024]
Prelude>  [ c | c <- "Hahaha! Ahahaha!", c `elem` ['A'..'Z']]
"HA"
Prelude>
```

From the output we can see that lists can be written as a basic comprehension, example, [x*2 | x <- [1..100], x 'mod' 2 == 0] , here part before the pipe i.e. x*2 is called the output function , x is variable, x 'mod' 2 == 0 is the predicate and [1..100] is the range of numbers from 1 to 100.

3. Perform the following using Haskell lists

a. Define a function doublePos that doubles the positive elements in the list of integers

```haskell
doublePos::[Int]->[Int]
doublePos [] = []
doublePos (x:xs)|0<x=(2*x):doublePos xs
                |otherwise = doublePos xs
```

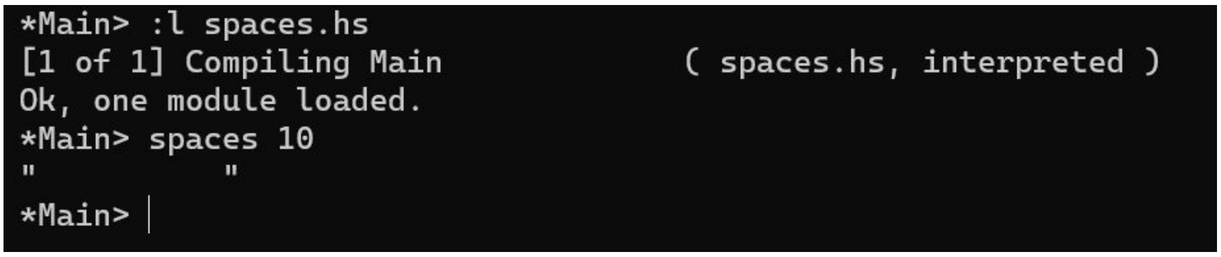


```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :cd D:\College\PCPF\Haskell\Exp6
Prelude> :l doublePos.hs
[1 of 1] Compiling Main             ( doublePos.hs, interpreted )
Ok, one module loaded.
*Main> doublePos [1,3,-2,7,-13,-16]
[2,6,14]
*Main>
```

From this output we can see that the function doublePos takes an input list, traverses through the list to find positive numbers and returns a list including all the positive doubled.

b. Define a function spaces n which returns a string of n spaces

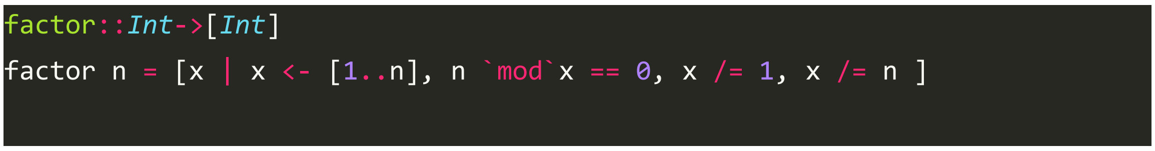
```haskell
spaces::Int->String
spaces n = [' '|x<-[1..n]]
```



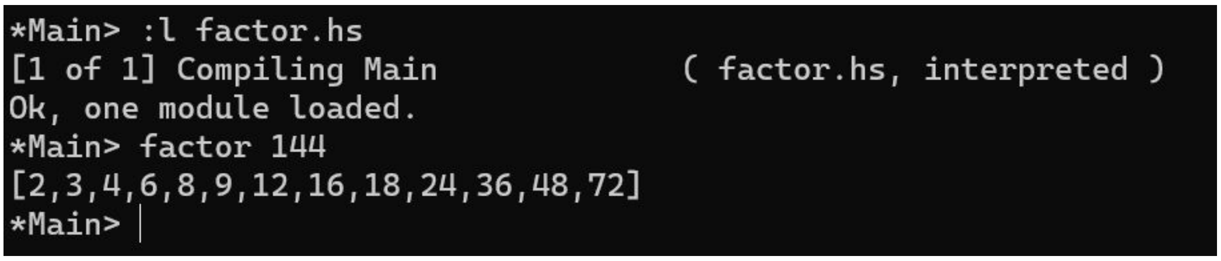
```
*Main> :l spaces.hs
[1 of 1] Compiling Main             ( spaces.hs, interpreted )
Ok, one module loaded.
*Main> spaces 10
"          "
*Main>
```

From the output we can see that the function spaces n takes an integer value as input and returns a output string containing n spaces.

c. Define a function factor n which returns a list of integers that divide n. Omit the trivial factors of 1 and n



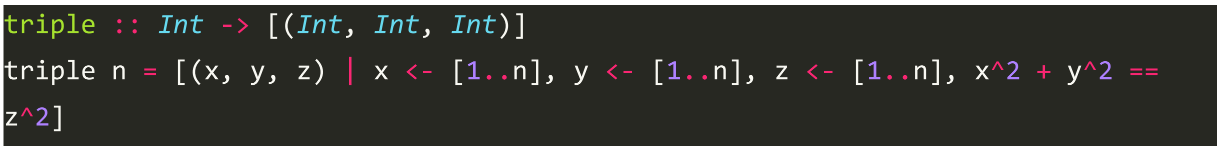
```haskell
factor::Int->[Int]
factor n = [x | x <- [1..n], n `mod`x == 0, x /= 1, x /= n ]
```



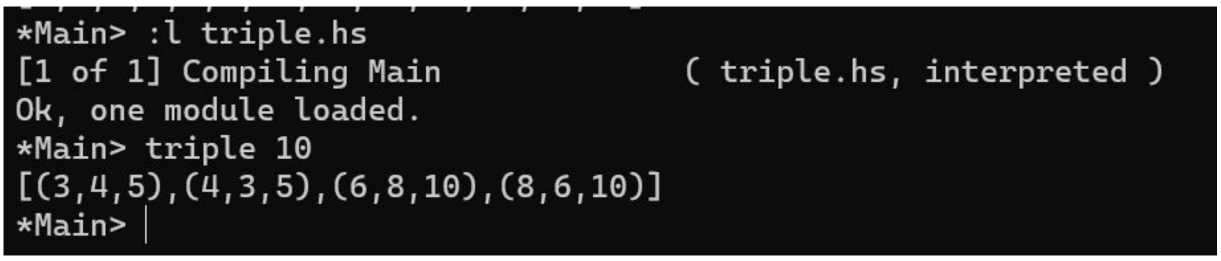
```
*Main> :l factor.hs
[1 of 1] Compiling Main             ( factor.hs, interpreted )
Ok, one module loaded.
*Main> factor 144
[2,3,4,6,8,9,12,16,18,24,36,48,72]
*Main>
```

From the output we can see that the function factor n takes an integer number as input and returns a list of the factors of number excluding 1 and the number itself.

d. Generate a list of triples (x,y,z) such that $x^2+y^2=z^2$



```haskell
triple :: Int -> [(Int, Int, Int)]
triple n = [(x, y, z) | x <- [1..n], y <- [1..n], z <- [1..n], x^2 + y^2 ==
z^2]
```



```
*Main> :l triple.hs
[1 of 1] Compiling Main             ( triple.hs, interpreted )
Ok, one module loaded.
*Main> triple 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
*Main>
```

From the above output we can see that the function triple n takes an integer value as input and returns a list of 3 numbers upto n, such that the sum of squares of the first two numbers is equal to the square of the third number.

8. Post Experimental Exercise
Questions:
1. List important characteristics of Lists in Haskell

List is a fundamental data structure in Haskell for managing several values. The square brackets delimit the list, and individual elements are separated by commas. The only important restriction is that all elements in a list must be of the same type. Trying to define a list with mixed-type elements results in a typical type error. In addition to specifying the whole list at once using square brackets and commas, you can build them up piece by piece using the (:) operator pronounced "cons". The process of building up a list this way is often referred to as consing.

2. Write a Haskell snippet to create a list of first 10 numbers in numbers

```
[x|x<-[1..10]]
```

3. Write a Haskell snippet to filter the elements from the list whose square root is greater than 7.

```
filter(\x->sqrt(fromIntegral x)>=7) x
```

4. Write a Haskell snippet to implement Fibonacci series. Define an expression fibs :: [Integer] that generates this infinite sequence.

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Conclusion:

In this experiment we have written Haskell programs to understand List data structure in Haskell and the various types of operations that can be performed on the Lists. We have also studied basic List comprehension through this experiment.

To perform inbuilt List operations on the list we have used the prelude prompt of the Glasgow Haskell compiler. We also have used Visual Studio Code to write all the user defined functions.

From this experiment we have inferred that List comprehension returns a list of elements created by evaluation of generators. They can have one or more input sets, and one or more predicates. The input set is a list of values which are fed, in order, to the output function. Ultimately, the generated (output) list will consist of all of the values of the input set, which, once fed through the output function, satisfy the predicate.

References:

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

[2] http://learnyouahaskell.com/

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