Principles of programming languages

Sub Code: 20CYS312 **Name: Chitra Harini**

Roll No:CH.EN.U4CYS22010 **Date: 20-12-2024**

Github link: https://github.com/Harini-chitra/Haskell/tree/main/Haskell_Lab4

Lab 4

Objective: To implement basic Haskell functions demonstrating tuple manipulation, list comprehensions, filtering, recursion, and functional programming concepts to solve real-world problems efficiently.

1. Implement a function swapTuple that takes a tuple (a, b) and swaps its elements, i.e., returns the tuple (b, a).

Haskell Code:

```
swapTuple :: (a, b) -> (b, a)
swapTuple (x, y) = (y, x)

main :: IO ()
main = do
  putStrLn "Testing swapTuple:"
  print (swapTuple (1, "a"))
  print (swapTuple ('x', True))
  print (swapTuple (42, 3.14))
```

Explanation of code:

- **swapTuple Function**: Takes a tuple (a, b) and returns a tuple (b, a) by pattern matching and swapping elements.
- **main Function**: Tests the swapTuple function with different data types, including integers, strings, characters, and booleans.

I/O Examples:

```
swapTuple (1, "a")      -- returns ("a", 1)
swapTuple ('x', True)   -- returns (True, 'x')
swapTuple (42, 3.14)    -- returns (3.14, 42)
```

Output Screenshot:

```
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit swaptuple.hs
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ghc swaptuple.hs
[1 of 1] Compiling Main             ( swaptuple.hs, swaptuple.o )
Linking swaptuple ...
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ./swaptuple
Testing swapTuple:
("a",1)
(True,'x')
(3.14,42)
```

2. Write a function multiplyElements that takes a list of numbers and a multiplier n, and returns a new list where each element is multiplied by n. Use a list comprehension for this task.

Haskell Code:

```
multiplyElements :: Num a => [a] -> a -> [a]
multiplyElements xs n = [x * n | x <- xs]

main :: IO ()
main = do
  putStrLn "Testing multiplyElements:"
  print (multiplyElements [1, 2, 3] 2)
  print (multiplyElements [5, -3, 7] 3)
  print (multiplyElements [] 10)
```

Explanation of code:

- **multiplyElements Function**: Uses a list comprehension to multiply each element in the list xs by the given multiplier n.
- **main Function**: Tests multiplyElements with various lists, including empty and mixed-sign numbers.

I/O Examples:

```
multiplyElements [1, 2, 3] 2      -- returns [2, 4, 6]
multiplyElements [5, -3, 7] 3     -- returns [15, -9, 21]
multiplyElements [] 10            -- returns []
```

Output Screenshot:

```
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit mul2ele.hs
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ghc mul2ele.hs
[1 of 1] Compiling Main             ( mul2ele.hs, mul2ele.o )
Linking mul2ele ...
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ./mul2ele
Testing multiplyElements:
[2,4,6]
[15,-9,21]
[]
```

3. Write a function filterEven that filters out all even numbers from a list of integers using the filter function.

Haskell Code:

```
filterEven :: [Int] -> [Int]
filterEven xs = filter odd xs

main :: IO ()
main = do
  putStrLn "Testing filterEven:"
  print (filterEven [1, 2, 3, 4, 5])
  print (filterEven [2, 4, 6, 8])
  print (filterEven [])
```

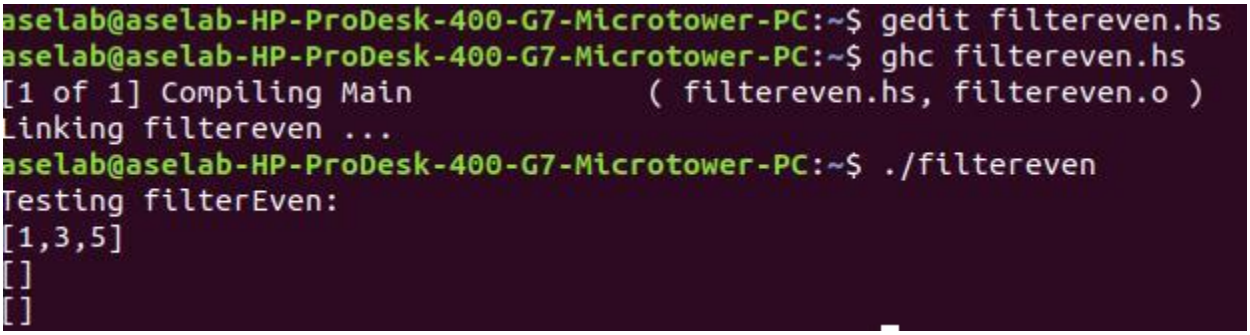
Explanation of code:

- **filterEven Function**: Uses the filter function with the odd predicate to exclude even numbers from the list.
- **main Function**: Tests filterEven with lists containing odd, even, and empty lists.

I/O Examples:

```
filterEven [1, 2, 3, 4, 5]      -- returns [1, 3, 5]
filterEven [2, 4, 6, 8]         -- returns []
filterEven []                   -- returns []
```

Output Screenshot:



4. Implement a function listZipWith that behaves similarly to zipWith in Haskell. It should take a function and two lists, and return a list by applying the function to corresponding elements from both lists. For example, given the function + and the lists [1, 2, 3] and [4, 5, 6], the result should be [5, 7, 9].

Haskell Code:

```
listZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
listZipWith _ [] _ = [] listZipWith _ _ [] = []
listZipWith f (x:xs) (y:ys) = f x y :
```

```
listZipWith f xs ys
main :: IO ()
main = do
  putStrLn "Testing listZipWith:"
  print (listZipWith (+) [1, 2, 3] [4, 5, 6])
  print (listZipWith (*) [1, 2] [10, 20, 30])
  print (listZipWith (++) ["a", "b"] ["x", "y", "z"])
```

Explanation of code:

- **listZipWith Function**: Recursively applies a binary function to corresponding elements of two lists. Stops when either list is exhausted.
- **main Function**: Tests listZipWith with addition, multiplication, and string concatenation.

I/O Examples:

```
listZipWith (+) [1, 2, 3] [4, 5, 6]          -- returns [5, 7, 9]
listZipWith (*) [1, 2] [10, 20, 30]          -- returns [10, 40]
listZipWith (++) ["a", "b"] ["x", "y", "z"]  -- returns ["ax", "by"]
```

Output Screenshot:

5. Write a recursive function reverseList that takes a list of elements and returns the list in reverse order. For example, given [1, 2, 3], the output should be [3, 2, 1].

Haskell Code:

```
reverseList :: [a] -> [a]
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]

main :: IO ()
main = do
  putStrLn "Testing reverseList:"
  print (reverseList [1, 2, 3])
  print (reverseList "hello")
  print (reverseList ([] :: [Int]))
```

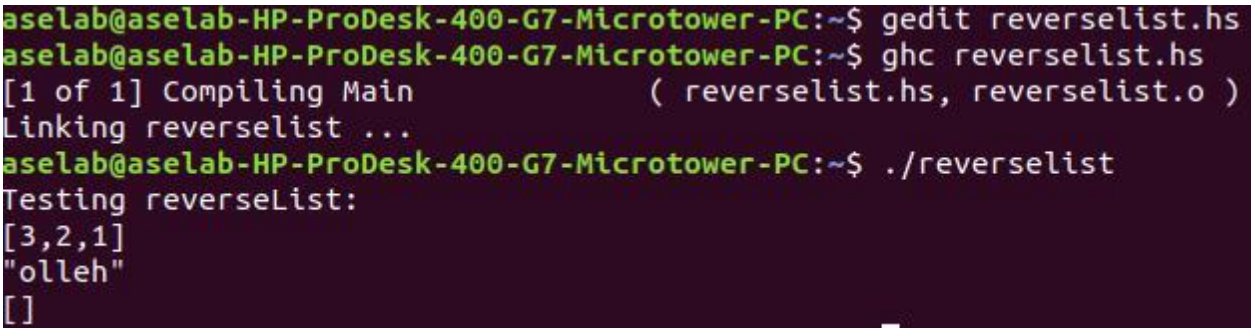
Explanation of code:

- **reverseList Function**:
- If the input list is empty ([]), return an empty list.
- If the list is non-empty (x:xs), recursively reverse the tail (xs) and append the head (x) to the end.
- **main Function**:
- Tests the reverseList function with various inputs:
    - A list of integers [1, 2, 3].
    - A string "hello" (strings in Haskell are lists of characters).
    - An empty list with an explicitly specified type ([] :: [Int]).

I/O Examples:

```
reverseList [1, 2, 3]          -- returns [3, 2, 1]
reverseList "hello"            -- returns "olleh"
reverseList ([] :: [Int])      -- returns []
```

Output Screenshot:



6. You are tasked with developing a program to manage and analyze student records. Each student is represented as a tuple (String, Int, [Int]), where the first element is the student's name (a string), the second is their roll number (an integer), and the third is a list of integers representing their marks in various subjects. Write a recursive function averageMarks to calculate the average of a student's marks. Display all student names and their average marks.

Haskell Code:

```
type Student = (String, Int, [Int])
averageMarks :: [Int] -> Double
averageMarks [] = 0
averageMarks marks = fromIntegral (sum marks) / fromIntegral (length marks)
displayAverages :: [Student] -> [(String, Double)]
displayAverages students = [(name, averageMarks marks) | (name, _, marks) <- students]

main :: IO ()
main = do
  let students = [("Alice", 1, [80, 90, 85]), ("Bob", 2, [70, 75, 80]), ("Charlie", 3, [])]
```

putStrLn "Student Averages:"
mapM_ print (displayAverages students)

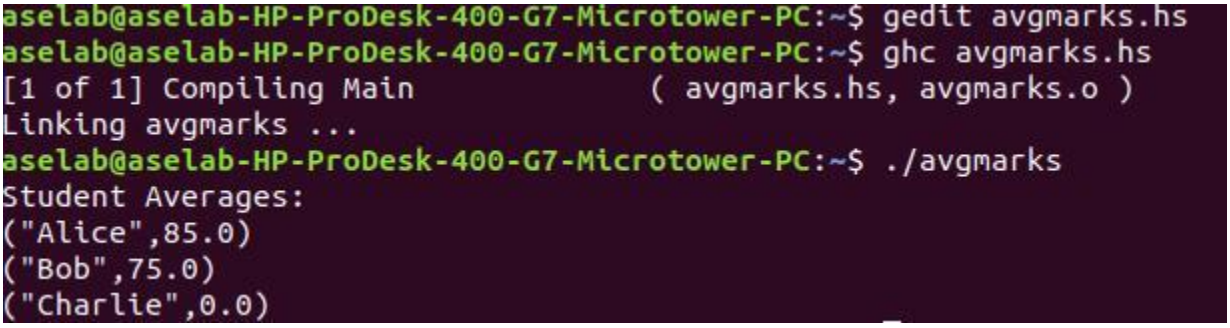
Explanation of code:

- **averageMarks Function**: Computes the average marks of a student by dividing the sum by the count of marks. Handles empty lists gracefully.
- **displayAverages Function**: Extracts student names and their average marks as a list of tuples.
- **main Function**: Demonstrates the program with a sample list of students.

I/O Examples:

students = [("Alice", 1, [80, 90, 85]), ("Bob", 2, [70, 75, 80]), ("Charlie", 3, [])]
displayAverages students -- returns [("Alice", 85.0), ("Bob", 75.0), ("Charlie", 0.0)]

Output Screenshot:



Conclusion: These problems enhance understanding of Haskell's functional paradigms, recursion, and list operations, building foundational skills for solving computational tasks elegantly.