20CYS312 - Principles of Programming Languages Lab - 4

R Subramanian CH.EN.U4CYS22043
DATE: 20/12/24

GitHub Repository: https://github.com/subra123/PPL-lab-work

1. Function: swapTuple

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

Program Code:

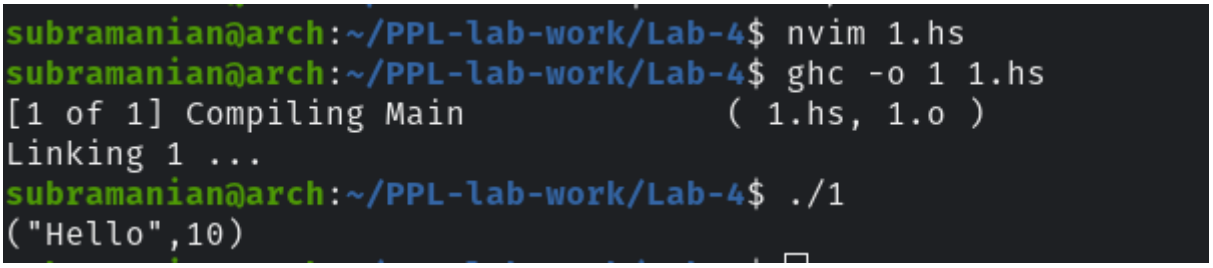
```haskell
swapTuple :: (a, b) -> (b, a)
swapTuple (a, b) = (b, a)

main :: IO ()
main = print $ swapTuple (10, "Hello")
```

Explanation of the Code:
`swapTuple` takes a tuple (a, b) and swaps the positions of its elements to return (b, a).

Input/Output Examples:

- Input: (10, "Hello")
- Output: ("Hello", 10)

Screenshots:



Conclusion:
`swapTuple` – This function highlights the simplicity and power of tuple manipulation in Haskell, where elements in a tuple are swapped using pattern matching.

---

2. Function: multiplyElements

Objective: 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.

Program Code:

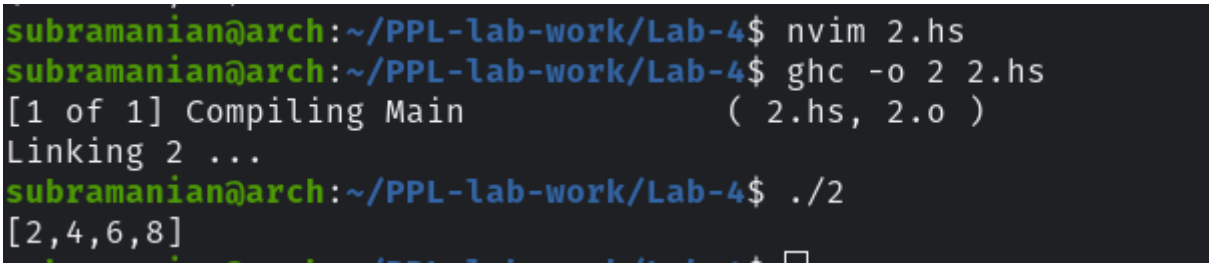
```haskell
multiplyElements :: Num a => [a] -> a -> [a]
multiplyElements lst n = [x * n | x <- lst]

main :: IO ()
main = print $ multiplyElements [1, 2, 3, 4] 2
```

Explanation of the Code:
`multiplyElements` multiplies each element of a list `lst` by a given number `n`. This is done using list comprehension to generate a new list.

Input/Output Examples:

- Input: [1, 2, 3, 4], 2
- Output: [2, 4, 6, 8]

Screenshots:



Conclusion:
`multiplyElements` – By utilizing list comprehension, this function efficiently multiplies each element in a list by a given number, showcasing Haskell's concise syntax for working with lists.

---

3. Function: filterEven

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

Program Code:

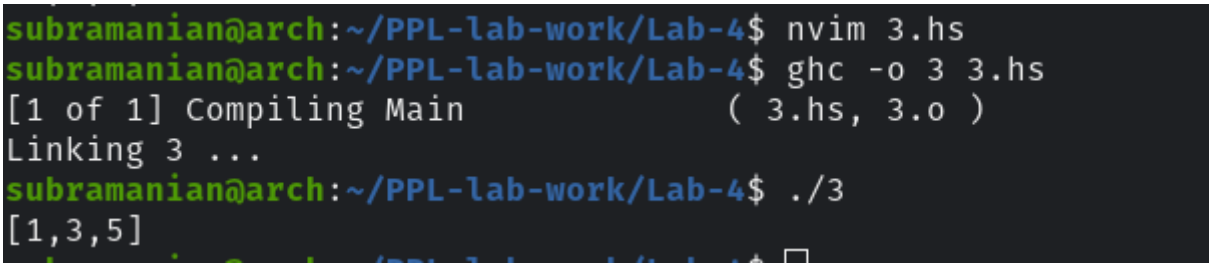
```haskell
filterEven :: [Int] -> [Int]
filterEven = filter odd

main :: IO ()
main = print $ filterEven [1, 2, 3, 4, 5, 6]
```

Explanation of the Code:
`filterEven` filters out even numbers from the list. This is done by using the `filter` function and the `odd` predicate (which keeps odd numbers).

Input/Output Examples:

- Input: [1, 2, 3, 4, 5]
- Output: [1, 3, 5]

Screenshots:



Conclusion:
`filterEven` – Using the `filter` function with a predicate, this function demonstrates how Haskell handles filtering elements in a list in a declarative manner, making it easy to process collections.

---

4. Function: listZipWith

Objective: 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].

Program Code:

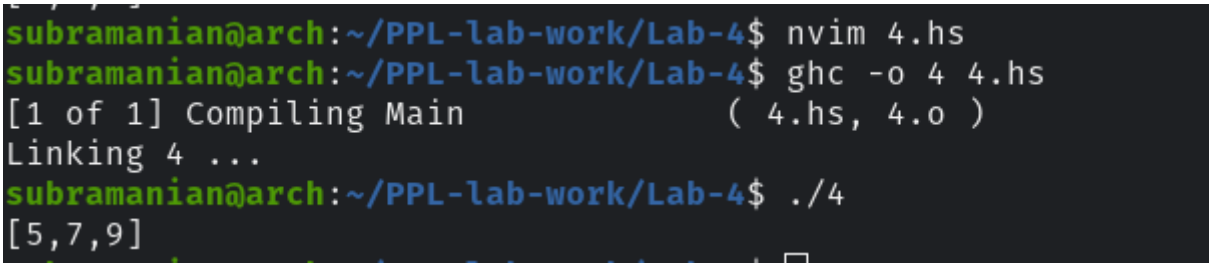
```
listZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
listZipWith _ [] _ = []
listZipWith _ _ [] = []
listZipWith f (x:xs) (y:ys) = f x y : listZipWith f xs ys

main :: IO ()
main = print $ listZipWith (+) [1, 2, 3] [4, 5, 6]
```

Explanation of the Code:
`listZipWith` takes a function `f` and two lists `xs` and `ys`. It zips the two lists together and applies the function `f` to each pair of elements, returning a new list with the results.

Input/Output Examples:

- Input: (+), [1, 2, 3], [4, 5, 6]
- Output: [5, 7, 9]

Screenshots:



Conclusion:
`listZipWith` – This function illustrates Haskell's ability to combine two lists element by element using higher-order functions, similar to the `zipWith` function in Haskell. It shows the functional approach to applying a function to corresponding elements of two lists.

---

5. Function: reverseList

Objective: 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].

Program Code:

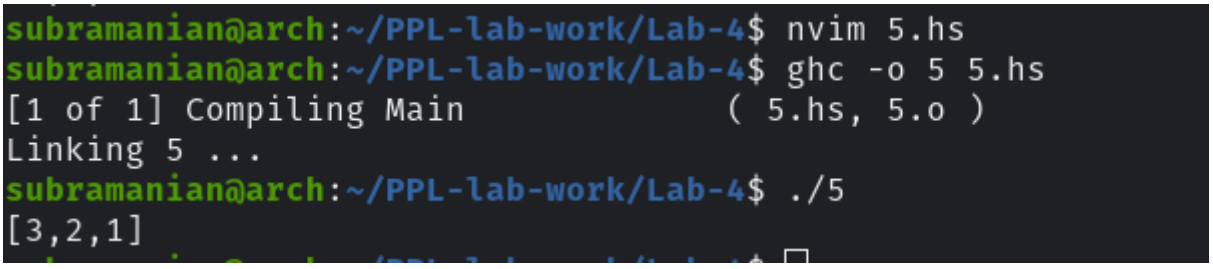
```haskell
reverseList :: [a] -> [a]
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]

main :: IO ()
main = print $ reverseList [1, 2, 3]
```

Explanation of the Code:
`reverseList` is a recursive function that reverses a list. It recursively processes the list, reversing the tail and appending the head to the result.

Input/Output Examples:

- Input: [1, 2, 3]
- Output: [3, 2, 1]

Screenshots:



Conclusion:
`reverseList` – By implementing recursion, this function demonstrates how Haskell efficiently processes lists in a recursive manner, a fundamental feature of functional programming.

---

6. Function: averageMarks

Objective: 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.

Program Code:

```
type Student = (String, Int, [Int])

averageMarks :: [Int] -> Double
averageMarks marks = fromIntegral (sum marks) / fromIntegral
(length marks)

main :: IO ()
main = do
    let students = [("R Subramanian", 101, [85, 90, 78]),
                    ("Manisk", 102, [70, 75, 80]),
                    ("Johnrex", 103, [95, 88, 92])]
    mapM_ (\(name, _, marks) ->
            putStrLn $ name ++ "'s average marks: " ++ show
(averageMarks marks)) students
```

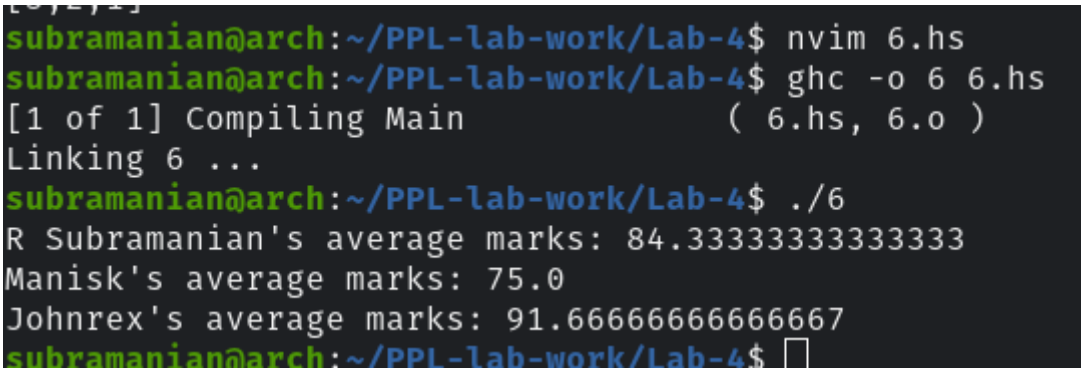
Explanation of the Code:
`averageMarks` calculates the average marks of a student by summing their marks and dividing by the length of the list. The `main` function iterates over a list of students and prints each student's name along with their average marks.

Input/Output Examples:

- Input:
    - R Subramanian: Roll number 101, marks: [85, 90, 78]
    - Manisk: Roll number 102, marks: [70, 75, 80]
    - Johnrex: Roll number 103, marks: [95, 88, 92]
- Output:
    - R Subramanian's average marks: 84.33333333333333
    - Manisk's average marks: 75.0

- Johnrex's average marks: 91.66666666666667

Screenshots:



Conclusion:

`averageMarks` – This function calculates the average of a student's marks and prints it in a formatted way. It exemplifies how Haskell handles numeric operations, lists, and data processing recursively and functionally.
