Department of Cyber Security

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Chennai Campus

Principals of Programming Languages

Subject Code: 20CYS312	Date:2024/12/20
Name: Sushant Yadav	RN:CH.EN.U4CYS22067

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

Code:

```haskell
swapTuple :: (a, b) -> (b, a)
swapTuple (a, b) = (b, a)
main :: IO ()
main = do
  print (swapTuple (1, "hello"))  -- Output: ("hello", 1)
  print (swapTuple ("apple", 3))  -- Output: (3, "apple")
```

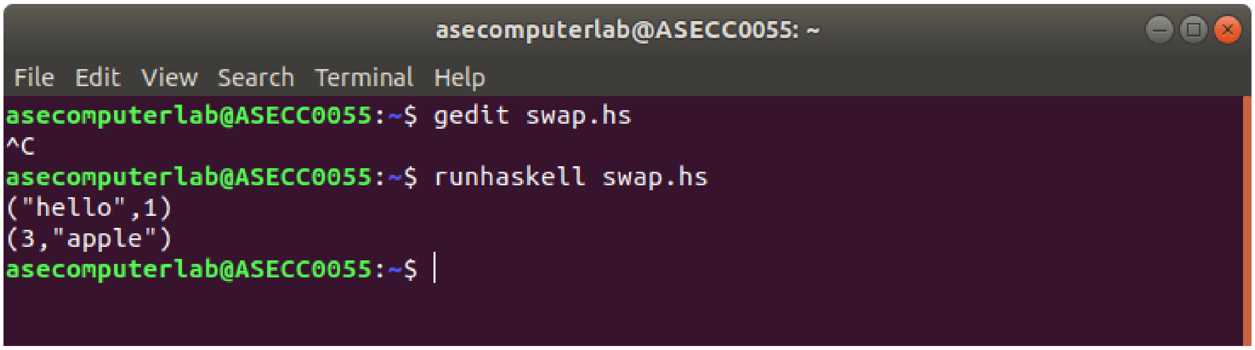
Explanation:

- **Function Name**: swapTuple
- **Type Signature**: (a, b) -> (b, a)

This indicates that the function takes a tuple with two elements of possibly different types and returns a tuple with the order of elements swapped.

- **Implementation**: The function pattern matches the input tuple (a, b) and directly returns (b, a).

Output:

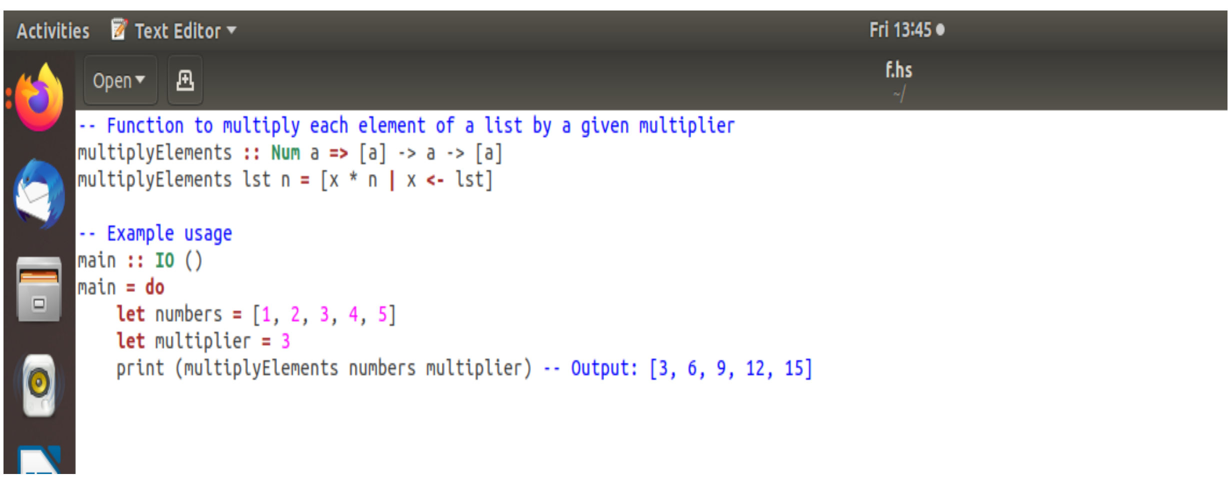


Conclusion

The swapTuple function in Haskell is a simple and effective utility that demonstrates pattern matching and tuple manipulation. By taking a tuple (a, b) and returning (b, a), it highlights Haskell's concise syntax and strong type system.

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.

Code:

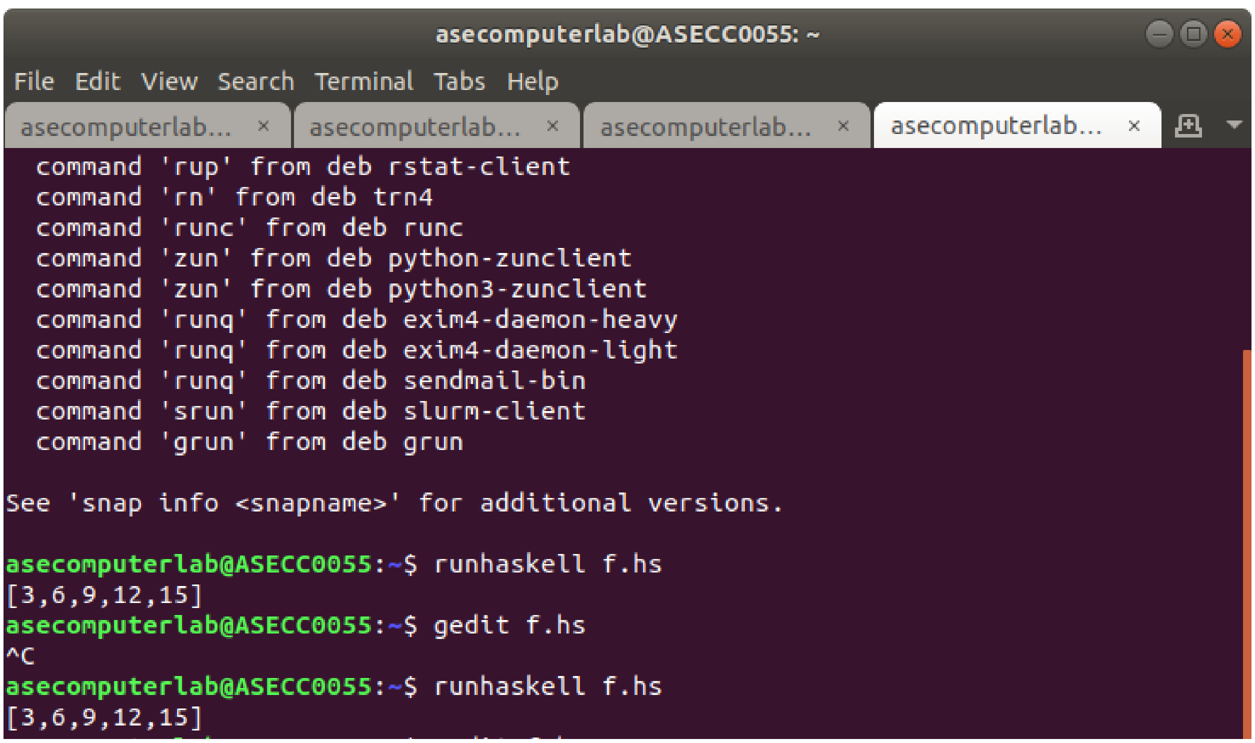


Explanation:

1. **Function Name**: multiplyElements

2. **Type Signature**: Num a => [a] -> a -> [a]
    a. The function takes a list of numbers ([a]) and a multiplier (a).
    b. It returns a new list of numbers ([a]), where each element is the product of the original element and the multiplier.
    c. The Num a constraint ensures that the elements and the multiplier are numeric.
3. **Implementation**: Uses a **list comprehension** to iterate over each element x in the list lst and calculates x * n.

Output:



Conclusion:

The multiplyElements function leverages Haskell's concise syntax and list comprehensions to process lists in an elegant and efficient way. It demonstrates Haskell's power in functional programming for transforming data structures.

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

Code:

Open filter.hs ~/

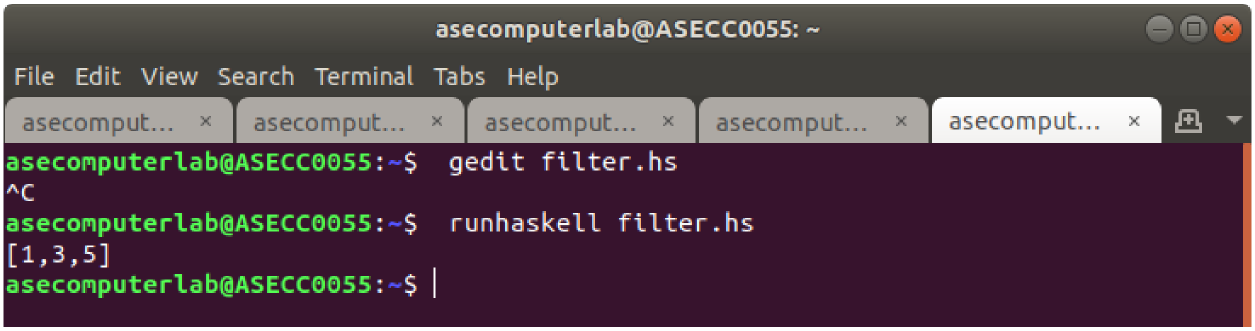
```haskell
-- Function to filter out even numbers from a list
filterEven :: [Int] -> [Int]
filterEven lst = filter odd lst

-- Example usage
main :: IO ()
main = do
    let numbers = [1, 2, 3, 4, 5, 6]
    print (filterEven numbers) -- Output: [1, 3, 5]
```

Explanation:

1. **Function Name**: filterEven
2. **Type Signature**: [Int] -> [Int]
    a. The function takes a list of integers ([Int]) and returns a list of integers with all even numbers removed.
3. **Implementation**:
    a. The filter function is used to create a new list.
    b. The predicate odd is applied to each element of the input list. Only elements for which the predicate returns True (odd numbers) are included in the result.

Output:



Conclusion:

The filterEven function is an excellent example of how Haskell’s filter function and built-in predicates like odd can simplify tasks like filtering elements in a list. It highlights Haskell's expressiveness and power in functional programming.

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].

Code:

Open ▾ listZip.hs

filter.hs ×

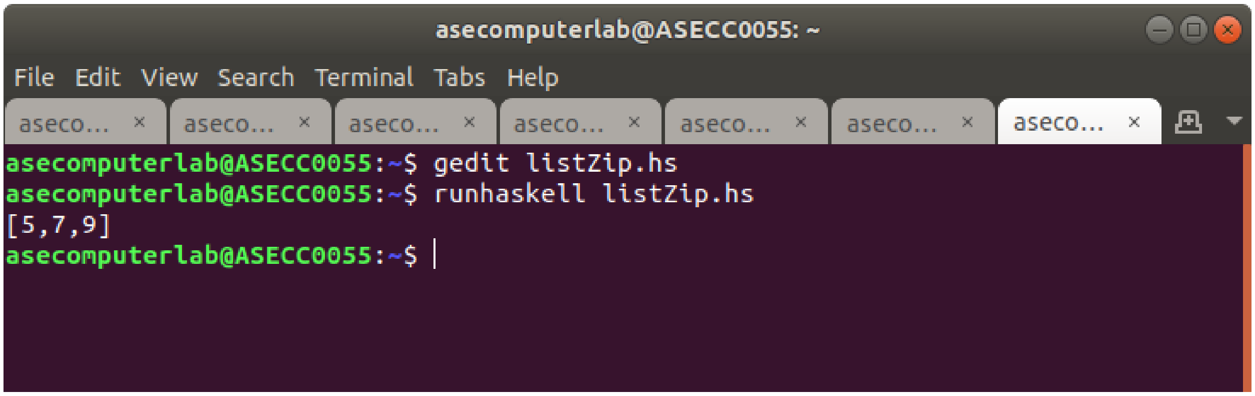
```haskell
-- Function that behaves like zipWith
listZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
listZipWith _ [] _ = []
listZipWith _ _ [] = []
listZipWith f (x:xs) (y:ys) = f x y : listZipWith f xs ys

-- Example usage
main :: IO ()
main = do
    let list1 = [1, 2, 3]
    let list2 = [4, 5, 6]
    let result = listZipWith (+) list1 list2
    print result -- Output: [5, 7, 9]
```

Explanation:

1. **Function Name**: listZipWith
2. **Type Signature**: (a -> b -> c) -> [a] -> [b] -> [c]
    a. Takes a function f of type (a -> b -> c), which combines elements of type a and b to produce elements of type c.
    b. Also takes two lists [a] and [b] and returns a new list [c].
3. **Implementation**:
    a. The function uses recursion to iterate through both input lists simultaneously.
    b. If either list is empty, the result is an empty list.
    c. Otherwise, the function applies f to the heads of the lists (x and y) and recursively processes the tails (xs and ys).

Output:



Conclusion:

The listZipWith function demonstrates Haskell's elegance in processing lists with recursion. It mirrors the behavior of Haskell's built-in zipWith function, providing a flexible tool for combining elements of two lists using any user-defined function.

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].

Code:

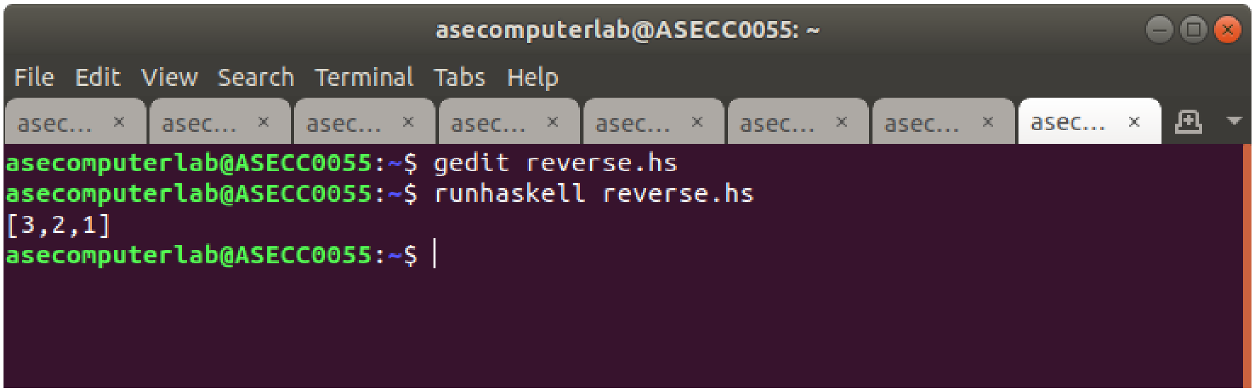
```haskell
-- Recursive function to reverse a list
reverseList :: [a] -> [a]
reverseList [] = []              -- Base case: an empty list is already reversed
reverseList (x:xs) = reverseList xs ++ [x]  -- Recursive case: reverse the tail and append the head

-- Example usage
main :: IO ()
main = do
    let list = [1, 2, 3]
    print (reverseList list)  -- Output: [3, 2, 1]
```

Explanation:

1. **Function Name**: reverseList
2. **Type Signature**: [a] -> [a]
   a. The function takes a list of elements of any type a and returns the list in reverse order.
3. **Implementation**:
   a. The base case: If the list is empty ([]), the reversed list is also empty.
   b. The recursive case: For a list (x:xs), the function reverses the tail (xs) and then appends the head (x) to the reversed tail. This results in the list being reversed one element at a time.

Output:



Conclusion:

The reverseList function is a simple yet effective example of recursion in Haskell. It demonstrates how recursion can be used to reverse a list by breaking it down into smaller subproblems: reversing the tail and then reassembling the list with the head at the end. This approach highlights the power of recursion in functional programming.

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.

Code:

```haskell
-- Type definition for a student (Name, Roll number, Marks)
type Student = (String, Int, [Int])

-- Function to calculate the average of a student's marks
averageMarks :: [Int] -> Double
averageMarks [] = 0  -- If the list of marks is empty, return 0
averageMarks marks = fromIntegral (sum marks) / fromIntegral (length marks)

-- Function to display student names and their average marks
displayStudentAverages :: [Student] -> IO ()
displayStudentAverages [] = return ()  -- Base case: if no students, do nothing
displayStudentAverages ((name, _, marks):rest) = do
    let avg = averageMarks marks
    putStrLn $ name ++ ": " ++ show avg
    displayStudentAverages rest  -- Recursive call for the rest of the students

-- Example usage
main :: IO ()
main = do
    let students = [("Alice", 1, [85, 90, 78]),
                    ("Bob", 2, [88, 76, 92]),
                    ("Charlie", 3, [70, 80, 85])]
    displayStudentAverages students
```

Explanation:

1. **Type Definition**:
2. The Student type is defined as a tuple (String, Int, [Int]), where:
    a. String represents the student's name.
    b. Int represents the student's roll number.
    c. [Int] is a list of integers representing the marks in various subjects.
3. **averageMarks Function**:

This function calculates the average of a list of integers:

a. The base case handles an empty list of marks, returning 0.
b. It sums the marks and divides by the length of the list, converting the values to Double for precision.

4. **displayStudentAverages Function**:

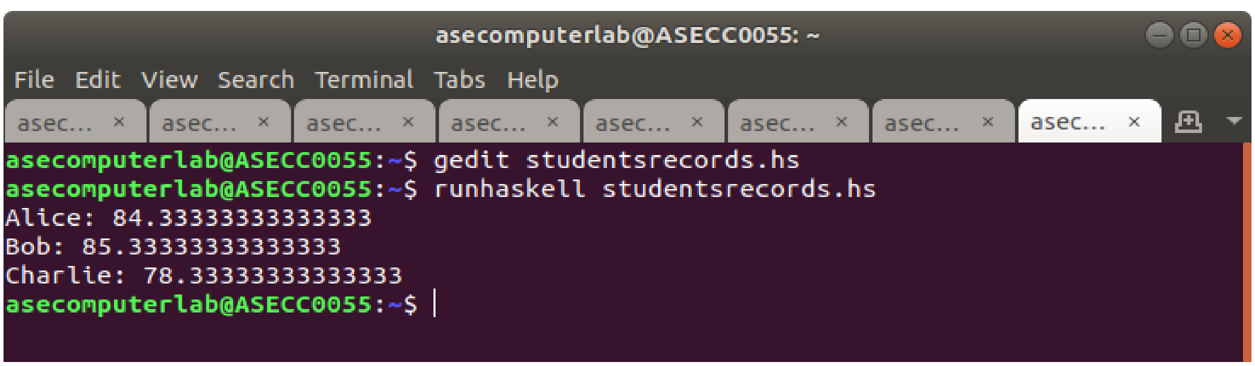
This recursive function processes a list of students:

a. It extracts the student's name and marks, calculates the average, and prints it.
b. The function then recursively processes the rest of the students.

5. **main Function**:

This provides sample student data and calls displayStudentAverages to display each student's name and their average marks.

Output:



Conclusion:

This program demonstrates how to manage student records, calculate averages, and display the results. It uses recursion to process a list of students, showcasing Haskell's ability to handle recursive tasks efficiently. The program calculates average marks, making it a useful tool for educational purposes or student management systems.