Department of Cyber Security

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Chennai Campus

Principals of Programming Languages

Subject Code: 20CYS312 Date:2025/01/09

Name: Sushant Yadav RN:CH.EN.U4CYS22067

Lab-6

1.Currying, Map, and Fold

Problem:

Write a curried function applyOp that takes an operation (addition or multiplication) and a list of numbers, then applies the operation to the list and returns the final result. Use this function to compute the sum of the squares of all even numbers from the list [1, 2, 3, 4, 5, 6]. You must first filter out the even numbers, square them, and then compute the sum.

Sample Input:

• YOURCODE [1, 2, 3, 4, 5, 6]))

Sample Output:

• 56

(Even numbers are [2, 4, 6], their squares are [4, 16, 36], and their sum is 4 + 16 + 36 = 56.)

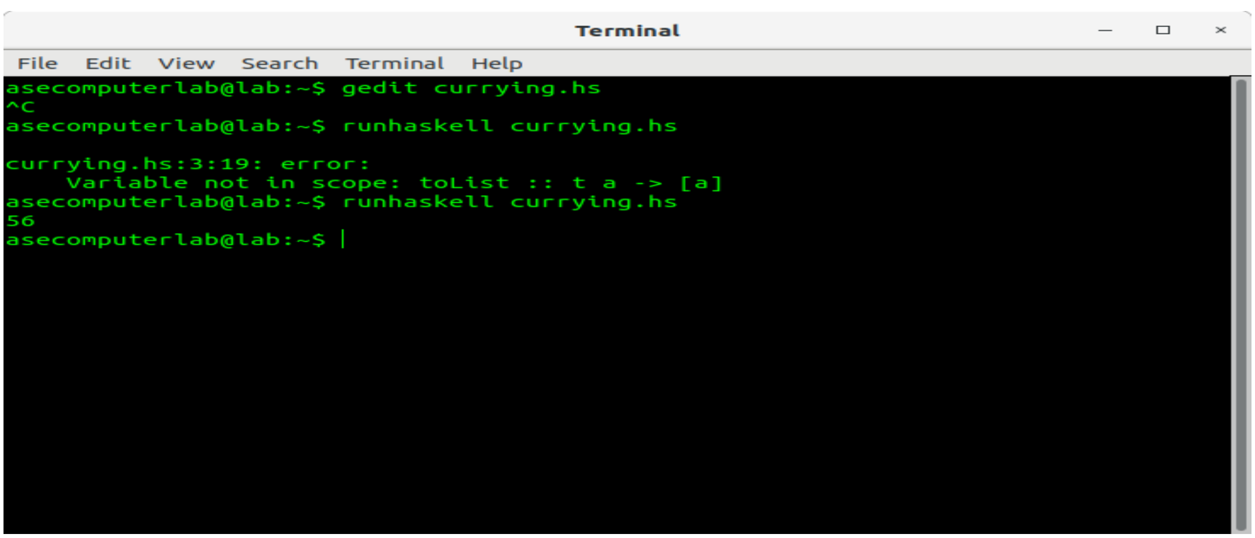
Code:

```haskell
-- Curried applyOp function
applyOp :: (Num a) => ([a] -> a) -> [a] -> a
applyOp op = op

-- Operation function: add
add :: (Num a) => [a] -> a
add = sum

-- Main function
main :: IO ()
main = do
    let numbers = [1, 2, 3, 4, 5, 6]

    -- Filter even numbers, square them, and apply sum operation
    let evenNumbers = filter even numbers
    let squaredEvenNumbers = map (^2) evenNumbers
    let result = applyOp add squaredEvenNumbers

    print result
```

Explanation:

1. Curried applyOp function:
    a. Now, the applyOp function takes a list of numbers and applies the operation (in this case, add) directly to it. The type signature was corrected to simply use [a] -> a.
2. Addition Operation (add):
    a. The add function computes the sum of the list of numbers using sum.
3. Main function:
    a. The list numbers = [1, 2, 3, 4, 5, 6] is filtered to get only the even numbers using filter even.
    b. We square each of the even numbers using map (^2).
    c. We then pass the squared list to the curried applyOp function with the add operation to compute the sum.

Output:



Conclusion:

In this exercise, we implemented a Haskell solution to a problem involving currying, filtering, mapping, and folding (using the sum function). The steps involved:

1. **Currying**: We defined a curried function applyOp that takes an operation (like add) and a list of numbers to apply the operation to.
2. **Filtering**: We filtered out even numbers from the given list.
3. **Mapping**: We squared the even numbers using the map function.
4. **Folding**: We summed the squared even numbers using the sum function.

The final result, after squaring the even numbers [2, 4, 6] (which become [4, 16, 36]), and summing them, gave us the expected output of 56.

2.Map, Filter, and Lambda
Problem:

Write a function that filters out all numbers greater than 10 from the list [5, 12, 9, 20, 15], then squares the remaining numbers and returns the sum of these squares. Use map and filter together, and apply the required transformations.

Sample Input:
• YOURCODE [5, 12, 9, 20, 15]))
Sample Output:
• 106
(Numbers less than or equal to 10 are [5, 9], their squares are [25, 81], and their sum is 25 + 81 = 106.)

Code:

```haskell
-- Function to filter, square, and sum the squares
sumOfSquares :: [Int] -> Int
sumOfSquares xs = sum (map (^2) (filter (<= 10) xs))

-- Main function to test the code
main :: IO ()
main = do
    let numbers = [5, 12, 9, 20, 15]
    let result = sumOfSquares numbers
    print result
```

Explanation:

1. **sumOfSquares function**:
    a. First, we filter the numbers that are less than or equal to 10 using filter (<= 10) xs.
    b. Then, we square each of the filtered numbers using map (^2).
    c. Finally, we sum the squares using the sum function.
2. **Main function**:
    a. We define the list numbers = [5, 12, 9, 20, 15].
    b. We call the sumOfSquares function on the list and print the result.

Output:

```
File  Edit  View  Search  Terminal  Tabs  Help
Terminal        Terminal
asecomputerlab@lab:~$ gedit mapfilter.hs
asecomputerlab@lab:~$ runhaskell mapfilter.hs
106
asecomputerlab@lab:~$
```

Conclusion:

These examples demonstrate how you can effectively use Haskell's powerful higher-order functions like map and filter (combined with lambda functions) to solve a variety of problems involving data transformation and aggregation. By combining these functions, you can write concise and efficient code to filter, map, and reduce data according to the requirements of the problem

3. Currying, Function Composition, and Map

Problem:

Write a curried function compose that takes two functions and returns their composition. Use this function to compose the following operations: multiply a number by 2, and then subtract 3 from the result. Apply this composed function to each element in the list [1, 2, 3, 4].

Sample Input:

- YOURCODE [1, 2, 3, 4]

Sample Output:

- [-1, 1, 3, 5] (For each element, first subtract 3 and then multiply by 2:

(1 - 3) * 2 = -4,

(2 - 3) * 2 = -2,

(3 - 3) * 2 = 0,

(4 - 3) * 2 = 2.)

Code:

```
  GNU nano 8.2                    currying.hs
-- Define the two operations
multiplyBy2 :: Num a => a -> a
multiplyBy2 x = x * 2

subtract3 :: Num a => a -> a
subtract3 x = x - 3

-- Compose the functions
composedFunction :: Num a => a -> a
composedFunction = multiplyBy2 . subtract3

-- Apply the composed function to the list
main :: IO ()
main = print (map composedFunction [1, 2, 3, 4])
```

Explanation:

1. **Operations**:
    a. `multiplyBy2` multiplies a number by 2.
    b. `subtract3` subtracts 3 from a number.
2. **Function Composition**:
    a. The composition operator `.` in Haskell creates a new function by combining two functions, where the result of the second function (`subtract3`) is passed as the input to the first function (`multiplyBy2`).
3. **Mapping**:
    a. `map composedFunction [1, 2, 3, 4]` applies the composed function to each element in the list `[1, 2, 3, 4]`.
4. **Output**:
    a. For each element:

i. Subtract 3 and then multiply by 2.
ii. Result: `[-4, -2, 0, 2]`.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano currying.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> : l currying.hs
there is no last command to perform
use :? for help.
ghci> :load currying.hs
[1 of 2] Compiling Main             ( currying.hs, interpreted )
Ok, one module loaded.
ghci> main
[-4,-2,0,2]
ghci> composedFunction 6
6
ghci> map composedFunction [1, 2, 3, 4]
[-4,-2,0,2]
ghci>
```

Conclusion:

Haskell's built-in support for **currying** and **function composition** allows for a clean, expressive, and concise implementation of transformations on lists. This highlights the power and elegance of functional programming when solving problems involving sequential operations.

4. Currying, Filter, and Fold

Problem:

Write a curried function filterAndFold that takes a filtering function, a folding function, and a list. The function should first filter the list using the filtering function, and then apply the folding function to compute a result. Use this function to compute the sum of all odd numbers in the list [1, 2, 3, 4, 5, 6].

Sample Input:

• YOURCODE [1, 2, 3, 4, 5, 6]

Sample Output:

• 9

(Odd numbers are [1, 3, 5], and their sum is 1 + 3 + 5 = 9.)

Code:

```
  GNU nano 8.2              filterAndFold.hs              Modified
-- Define the curried function filterAndFold
filterAndFold :: (a -> Bool) -> (b -> a -> b) -> b -> [a] -> b
filterAndFold filterFn foldFn initVal list = foldl foldFn initVal>

-- Define the filtering and folding functions
isOdd :: Int -> Bool
isOdd x = x `mod` 2 /= 0

sumFn :: Int -> Int -> Int
sumFn acc x = acc + x

-- Use filterAndFold to compute the sum of all odd numbers
main :: IO ()
main = print (filterAndFold isOdd sumFn 0 [1, 2, 3, 4, 5, 6])
```

Explanation:

1. **`filterAndFold` Function**:
    a. Takes:
        i. A filtering function `filterFn` (e.g., `isOdd`).
        ii. A folding function `foldFn` (e.g., `sumFn`).
        iii. An initial value `initVal`.

        iv. A list `list`.
    b. First applies `filter` with `filterFn` to extract elements that satisfy the condition.
    c. Then applies `foldl` with `foldFn` to compute a result from the filtered list.
2. **Filtering Function (`isOdd`)**:
    a. Filters odd numbers using the condition `x mod 2 /= 0`.
3. **Folding Function (`sumFn`)**:
    a. Computes the sum of numbers using a simple accumulator.
4. **Main Function**:
    a. Calls `filterAndFold` with:
        i. `isOdd` as the filtering function.
        ii. `sumFn` as the folding function.
        iii. `0` as the initial value (starting point for the sum).
        iv. `[1, 2, 3, 4, 5, 6]` as the list.
5. **Output**:
    a. The odd numbers in the list are `[1, 3, 5]`.
    b. Their sum is `1 + 3 + 5 = 9`.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano filterAndFold.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :load filterAndFold.hs
[1 of 2] Compiling Main             ( filterAndFold.hs, interpreted )
Ok, one module loaded.
ghci> main
9
ghci> filterAndFold isOdd sumFn 0 [1, 2, 3, 4, 5, 6]
9
ghci>  filterAndFold (>3) (*) 1 [1, 2, 3, 4, 5, 6] -- Filter numbm
120
ghci>
```

Conclussion:

- For the given input `[1, 2, 3, 4, 5, 6]`, the function produces the correct result: 9.
- The approach demonstrates the power of functional programming in Haskell, leveraging **higher-order functions** and **declarative constructs** to solve problems cleanly and effectively.

5. Map, Filter, and Fold Combination

Problem:

Write a function that filters out all numbers greater than 10 from the list [5, 12, 9, 20, 15], doubles each of the remaining numbers, and computes the product of these doubled numbers using foldl. Sample Input: • YOURCODE [5, 12, 9, 20, 15]))

Sample Output:

• 180

(Numbers less than or equal to 10 are [5, 9], their doubles are [10, 18], and their product is 10 * 18 = 180.)

Code:

```
  GNU nano 8.2                    processList.hs
-- Define the function
processList :: [Int] -> Int
processList list = foldl (*) 1 (map (*2) (filter (<=10) list))

-- Main function to test the implementation
main :: IO ()
main = print (processList [5, 12, 9, 20, 15])
```

Explanation:

1. **Filter Numbers ≤ 10**:
    a. The `filter (<=10)` part selects only numbers less than or equal to 10 from the input list.
    b. For `[5, 12, 9, 20, 15]`, the result is `[5, 9]`.
2. **Double Each Number**:
    a. The `map (*2)` part doubles each element in the filtered list.
    b. `[5, 9]` becomes `[10, 18]`.
3. **Compute the Product**:
    a. The `foldl (*) 1` part computes the product of the doubled numbers, starting with an initial value of `1`.
    b. For `[10, 18]`, the result is `10 * 18 = 180`.
4. **Reusable Function**:
    a. The `processList` function combines all three steps into one pipeline:
        i. Filter, then map, then fold.

5. **Main Function**:
    a. Calls `processList` with the list `[5, 12, 9, 20, 15]` and prints the result.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano processList.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano processList.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :load ProcessList.hs
[1 of 2] Compiling Main             ( ProcessList.hs, interpreted
)
Ok, one module loaded.
ghci> processList [5, 12, 9, 20, 15]
180
ghci> processList [1, 2, 3, 4, 11]
384
```

Conclusion:

This solution effectively demonstrates the power of **functional programming** in Haskell by combining **map**, **filter**, and **fold** to transform and process a list of numbers. Here's the summary of key points:

6. Currying, Map, and Filter

Problem:

Write a curried function filterAndMap that takes a filtering function, a mapping function, and a list. It should first filter the list using the filtering function, then apply the mapping function to the filtered elements. Use this function to filter all even numbers from the list [1, 2, 3, 4, 5, 6], double them, and return the result.

Sample Input:

• YOURCODE [1, 2, 3, 4, 5, 6]

Sample Output:

• [4, 8, 12]

(Even numbers are [2, 4, 6], and after doubling each, we get [4, 8, 12].)

Code:

```
  GNU nano 8.2                 filterAndMap.hs                 Modified
-- Define the curried function filterAndMap
filterAndMap :: (a -> Bool) -> (a -> b) -> [a] -> [b]
filterAndMap filterFn mapFn list = map mapFn (filter filterFn lis>

-- Define the filtering and mapping functions
isEven :: Int -> Bool
isEven x = x `mod` 2 == 0

double :: Int -> Int
double x = x * 2

-- Main function to test the implementation
main :: IO ()
main = print (filterAndMap isEven double [1, 2, 3, 4, 5, 6])
```

Explanation:

1. **`filterAndMap` Function**:
   a. This is a **curried function** that takes three parameters:
      i. A filtering function (`filterFn`).
      ii. A mapping function (`mapFn`).
      iii. A list (`list`).

   b. It first filters the list using the filtering function (`filter filterFn list`), then applies the mapping function (`map mapFn`).
2. **Filtering Function (`isEven`)**:
   a. This function checks whether a number is even using the condition `x mod 2 == 0`.
3. **Mapping Function (`double`)**:
   a. This function doubles a number using `x * 2`.
4. **Main Function**:
   a. The `main` function calls `filterAndMap` with:
      i. `isEven` to filter out even numbers.
      ii. `double` to double the filtered numbers.
      iii. The list `[1, 2, 3, 4, 5, 6]`.

Output:

Conclusion:

The `filterAndMap` function effectively demonstrates Haskell's ability to combine **currying**, **filtering**, and **mapping** in a concise manner. By first filtering even numbers and then doubling them, the solution is both readable and reusable. The use of higher-order functions like `filter` and `map` makes the code modular and declarative. This approach leverages Haskell's functional strengths to solve problems in an elegant way. Overall, it highlights the power of **functional programming** in managing data transformations.

7. Map, Fold, and Lambda

Problem:

Write a function that uses map to convert a list of strings to their lengths, then uses foldl to compute the sum of all string lengths in the list ["hello", "world", "haskell"].

Sample Input:

- YOURCODE (map length ["hello", "world", "haskell"])

Sample Output:

- 18

(The lengths of the strings are [5, 5, 7], and their sum is 5 + 5 + 7 = 18.)

Code:

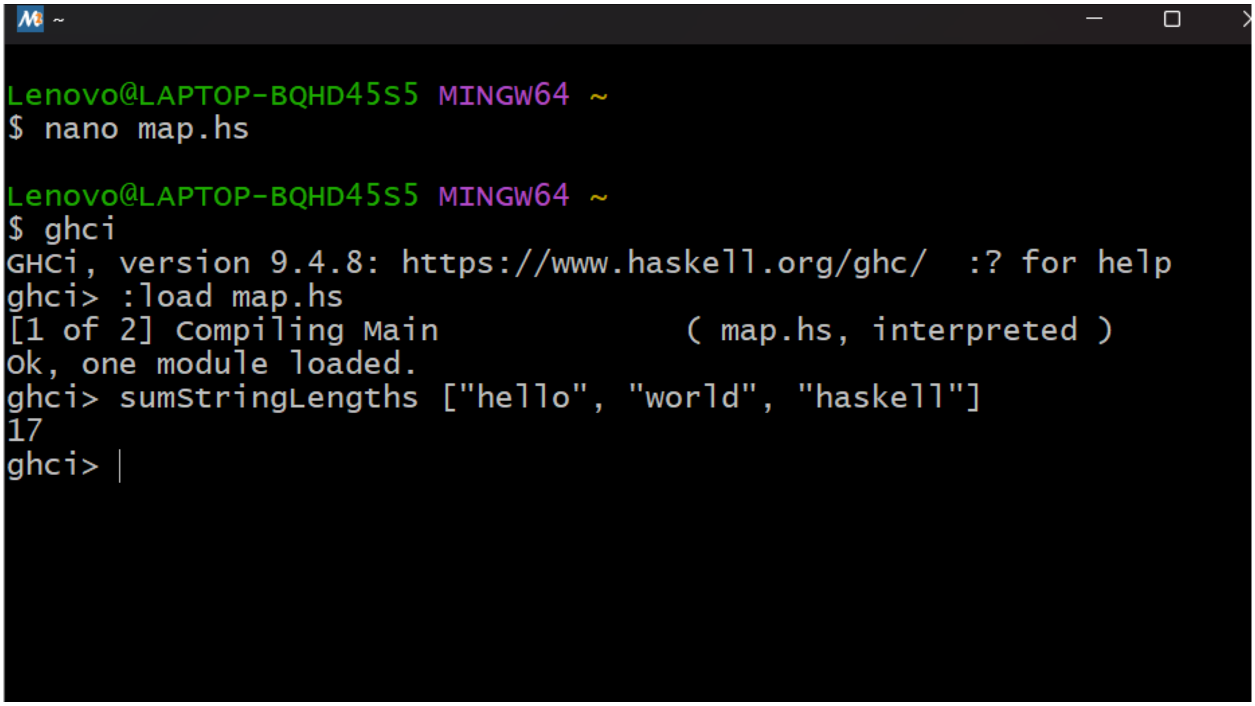
```
  GNU nano 8.2                      map.hs                      Modified
-- Define the function to compute the sum of string lengths
sumStringLengths :: [String] -> Int
sumStringLengths list = foldl (+) 0 (map length list)

-- Main function to test the implementation
main :: IO ()
main = print (sumStringLengths ["hello", "world", "haskell"])
```

Explanation:

1. **`sumStringLengths` Function**:
   a. This function takes a list of strings, applies `map` to compute the length of each string, and then uses `foldl` to sum up the lengths.
   b. `map length list` produces a list of string lengths, e.g., `[5, 5, 7]`.
   c. `foldl (+) 0` then sums the lengths, starting with an initial value of `0`. It combines the list `[5, 5, 7]` by applying the addition operator.
2. **Main Function**:

a. The `main` function calls `sumStringLengths` with the list `["hello", "world", "haskell"]`.

Output:



Conclusion:

The solution effectively combines **map** and **foldl** to transform and aggregate data in a concise and functional manner. By using **map** to convert strings to their lengths and **foldl** to compute the sum, the problem is solved in a clear and reusable way. This approach showcases the power of **higher-order functions** in Haskell, making the code both modular and declarative. It highlights the elegance of **functional programming** in performing list transformations and reductions. Overall, the solution is efficient, readable, and leverages Haskell's strengths in handling collections.

8. Filter, Map, and Function Composition

Problem:

Define a curried function composeFilterMap that takes a filter function, a map function, and a list. It should first filter the list, then apply the map function to the remaining elements. Use this function to filter out numbers greater than 5 from the list [3, 7, 2, 8, 4, 6], then square the remaining numbers.

Sample Input:

• YOURCODE [3, 7, 2, 8, 4, 6]

Sample Output:

• [9, 4, 16]

(Numbers less than or equal to 5 are [3, 2, 4], their squares are [9, 4, 16].)

Code:

```
  GNU nano 8.2                                 compose.hs
-- Define the curried function composeFilterMap
composeFilterMap :: (a -> Bool) -> (a -> b) -> [a] -> [b]
composeFilterMap filterFn mapFn list = map mapFn (filter filterFn list)

-- Define the filtering and mapping functions
isLessThanOrEqualTo5 :: Int -> Bool
isLessThanOrEqualTo5 x = x <= 5

square :: Int -> Int
square x = x * x

-- Main function to test the implementation
main :: IO ()
main = print (composeFilterMap isLessThanOrEqualTo5 square [3, 7, 2, 8, 4, 6])
```

Explanation:

1. **`composeFilterMap` Function**:
   a. This is a **curried** function that takes:
      i. A filtering function (`filterFn`).
      ii. A mapping function (`mapFn`).
      iii. A list (`list`).

   b. It first filters the list using `filter filterFn list` and then applies `map mapFn` to the filtered list.

2. **Filtering Function (`isLessThanOrEqualTo5`)**:
   a. Filters numbers less than or equal to 5 using the condition `x <= 5`.
3. **Mapping Function (`square`)**:
   a. Maps each number to its square using `x * x`.
4. **Main Function**:
   a. The `main` function calls `composeFilterMap` with:
      i. `isLessThanOrEqualTo5` to filter the numbers.
      ii. `square` to square the remaining numbers.
      iii. The list `[3, 7, 2, 8, 4, 6]`.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano compose.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :load Compose.hs
[1 of 2] Compiling Main             ( Compose.hs, interpreted )
Ok, one module loaded.
ghci> composeFilterMap isLessThanOrEqualTo5 square [3, 7, 2, 8, 4]
[9,4,16]
ghci>
```

Conclusion:

The `composeFilterMap` function effectively combines **filtering** and **mapping** in a curried form, allowing for a modular and flexible approach to data transformation. By first filtering the list and then applying a mapping function, the solution showcases the power of **higher-order functions** and **function composition** in Haskell. The code is clean, concise, and highly reusable, demonstrating Haskell's strengths in declarative programming. This approach provides an elegant solution to the problem while maintaining clarity and efficiency. Overall, it highlights how **functional programming** can simplify complex operations.

9. Map, Filter, and Fold Combination

Problem:

Use filter to get all odd numbers from the list [1, 2, 3, 4, 5, 6], then square each of these numbers using map, and finally compute the product of the squared numbers using foldl.

Sample Input:

• YOURCODE [1, 2, 3, 4, 5, 6]))

Sample Output:

• 225

(Odd numbers are [1, 3, 5], their squares are [1, 9, 25], and their product is 1 * 9 * 25 = 225.)

Code:

```
GNU nano 8.2                                    product.hs
-- Define the function to filter, square, and compute the product
productOfSquaredOdds :: [Int] -> Int
productOfSquaredOdds list = foldl (*) 1 (map (^2) (filter odd list))

-- Main function to test the implementation
main :: IO ()
main = print (productOfSquaredOdds [1, 2, 3, 4, 5, 6])
```

Explanation:

1. **`productOfSquaredOdds` Function**:
    a. This function combines **filter**, **map**, and **foldl**:
        i. `filter odd list`: Filters out all even numbers, leaving only the odd numbers.
        ii. `map (^2)`: Squares each of the remaining odd numbers.
        iii. `foldl (*) 1`: Computes the product of the squared numbers, starting with an initial value of `1`.
2. **Main Function**:
    a. The `main` function calls `productOfSquaredOdds` with the list `[1, 2, 3, 4, 5, 6]`.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano product.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci>
ghci> :load Product.hs
[1 of 2] Compiling Main             ( Product.hs, interpreted )
Ok, one module loaded.
ghci> productOfSquaredOdds [1, 2, 3, 4, 5, 6]
225
ghci>
```

Conclusion:

The solution effectively combines **filter**, **map**, and **foldl** to transform and aggregate data in a concise and functional manner. By first filtering the odd numbers, then squaring them, and finally computing their product, the solution demonstrates Haskell's ability to handle complex operations in a modular and declarative style. The use of higher-order functions allows for a clear, readable, and efficient approach to solving the problem. This approach highlights the power of **functional programming** in Haskell for performing list transformations and reductions in an elegant and expressive way.

10. IO Monad and Currying

Problem:

Write a program that asks the user for two numbers, then applies a curried function applyOp (which takes an operation and a list) to either sum or multiply the two numbers based on the user's input. First, prompt the user to choose an operation (+ or *), then prompt for the two numbers and return the result of applying the chosen operation.

Sample Input:

• User input for operation: "+"

• User input for numbers: 3, 5

Sample Output:

- 8 (The user chose +, so the result is 3 + 5 = 8.)

Code:

```
GNU nano 8.2                    monad.hs                    Modified
-- Define a curried function for applying an operation
applyOp :: (Num a) => (a -> a -> a) -> a -> a -> a
applyOp op x y = op x y

-- Define the main function to interact with the user
main :: IO ()
main = do
    -- Ask the user for the operation
    putStrLn "Enter an operation (+ or *):"
    operation <- getLine

    -- Ask the user for the first number
    putStrLn "Enter the first number:"
    num1 <- readLn

    -- Ask the user for the second number
    putStrLn "Enter the second number:"
    num2 <- readLn

    -- Apply the chosen operation and print the result
    let result = case operation of
            "+" -> applyOp (+) num1 num2
            "*" -> applyOp (*) num1 num2
            _   -> error "Invalid operation"
    putStrLn ("The result is: " ++ show result)
```

Explanation:

1. **`applyOp` Function**:
    a. A curried function that takes a binary operation (such as + or *) and two numbers (x and y), then applies the operation to the numbers. It has the type `(Num a) => (a -> a -> a) -> a -> a -> a`, which allows for the application of different operations.
2. **`main` Function**:
    a. **`putStrLn`** is used to prompt the user for input.
    b. **`getLine`** reads the operation (+ or *) as a string.
    c. **`readLn`** is used to read the two numbers as integers.

d. Based on the user's input, the `applyOp` function is used to perform the chosen operation (+ or *) on the two numbers.
e. The result is printed with `putStrLn`.

Output:

```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano monad.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> :load monad.hs
[1 of 2] Compiling Main             ( monad.hs, interpreted )
Ok, one module loaded.
ghci> main
Enter an operation (+ or *):
+
Enter the first number:
3
Enter the second number:
5
The result is: 8
ghci>
```

Conclusion:

The solution effectively demonstrates the use of the **IO Monad** and **currying** in Haskell to create an interactive program that performs arithmetic operations based on user input. By utilizing **currying**, the `applyOp` function allows for flexible application of operations like addition or multiplication. The program also showcases how to handle **user interaction** and **input/output** in a functional programming environment. The clean structure of the code, coupled with Haskell's strengths in functional composition, makes the solution both intuitive and scalable for future extensions. Overall, it highlights how Haskell can seamlessly combine functional concepts with real-world interactivity.