

20CYS312 - Principles of Programming Languages - Lab Exercise 6

Currying, Map, and Fold

Objective:

The goal of this exercise is to implement a curried function `applyOp` that takes an operation (either addition or multiplication) and a list of numbers, then applies the operation to the list and returns the final result. The task is to compute the sum of the squares of all even numbers from the list `[1, 2, 3, 4, 5, 6]`. This should be done by first filtering out the even numbers, squaring them, and then computing the sum.

Program Code:

```
applyOp :: (Int → Int → Int) → [Int] → Int
applyOp op xs = foldl op 0 xs

sumOfSquaresOfEvens :: [Int] → Int
sumOfSquaresOfEvens xs = applyOp (+) (map (^2) (filter even xs))

main = do
  print (sumOfSquaresOfEvens [1, 2, 3, 4, 5, 6])
  print (sumOfSquaresOfEvens [1, 3, 5])
  print (sumOfSquaresOfEvens [2, 4, 6, 8, 10])
  print (sumOfSquaresOfEvens [1, 2, 3, 4])
```

Explanation of the Code:

- `applyOp` Function:

- `applyOp` is a curried function that takes an operation (such as addition or multiplication) and a list of integers as inputs.
- It applies the operation (passed as `op`) to each element in the list using `foldl`, which accumulates the result starting from an initial value of `0`.
- `foldl` is a left-fold function that reduces the list to a single value by applying the operation across all elements in the list.
- **`sumOfSquaresOfEvens` Function:**
 - This function first filters the even numbers from the input list using the `filter even xs` function.
 - It then squares each even number using `map (^2)`, which applies the squaring operation to each element in the list.
 - Finally, it applies the `applyOp (+)` function to compute the sum of these squared numbers.
- **`main` Function:**
 - In the `main` function, we call `sumOfSquaresOfEvens` with various input lists and print the result for each case.

Input/Output Examples:

Here are the sample inputs tested in the program along with their corresponding outputs:

- **Input 1:** `sumOfSquaresOfEvens [1, 2, 3, 4, 5, 6]`
Output 1: `56`
- **Input 2:** `sumOfSquaresOfEvens [1, 3, 5]`
Output 2: `0`
- **Input 3:** `sumOfSquaresOfEvens [2, 4, 6, 8, 10]`
Output 3: `220`
- **Input 4:** `sumOfSquaresOfEvens [1, 2, 3, 4]`
Output 4: `20`

Output:

```
asecomputerlab@ase-computer-lab: ~/ppl/ex6
File Edit View Search Terminal Help
asecomputerlab@ase-computer-lab:~/ppl/ex6$ gedit ques1.hs
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ghc -o 1 ques1.hs
[1 of 1] Compiling Main                ( ques1.hs, ques1.o )
Linking 1 ...
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ./1
56
0
220
20
asecomputerlab@ase-computer-lab:~/ppl/ex6$
```

Conclusion:

In this exercise, we implemented a curried function `applyOp` that allows us to apply an operation (such as addition) to a list of numbers. We used this function to calculate the sum of the squares of even numbers from a given list. The solution involves using common functional programming concepts in Haskell such as currying, higher-order functions (`map`, `foldl`), and filtering data. The multiple examples help demonstrate how the function works with different inputs and scenarios.

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.

Program Code:

```
sumOfSquares :: [Int] → Int
sumOfSquares xs = sum (map (^2) (filter (<= 10) xs))
```

```
main = do
  print (sumOfSquares [5, 12, 9, 20, 15])
  print (sumOfSquares [2, 3, 7, 8])
  print (sumOfSquares [15, 18, 12, 20])
  print (sumOfSquares [1, 2, 3])
```

Input/Output Examples:

- **Input 1:** `sumOfSquares [5, 12, 9, 20, 15]`
Output 1: `106`
- **Input 2:** `sumOfSquares [2, 3, 7, 8]`
Output 2: `118`
- **Input 3:** `sumOfSquares [15, 18, 12, 20]`
Output 3: `0`
- **Input 4:** `sumOfSquares [1, 2, 3]`
Output 4: `14`

Output:

```
asecomputerlab@ase-computer-lab: ~/ppl/ex6
File Edit View Search Terminal Help
asecomputerlab@ase-computer-lab:~/ppl/ex6$ gedit ques2.hs
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ghc -o 2 ques2.hs
[1 of 1] Compiling Main                ( ques2.hs, ques2.o )
Linking 2 ...
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ./2
106
126
0
14
asecomputerlab@ase-computer-lab:~/ppl/ex6$
```

Conclusion:

In this exercise, we used `map` and `filter` to filter out numbers greater than 10, square the remaining numbers, and calculate their sum. The implementation effectively demonstrates the power of functional programming in Haskell.

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]`.

Program Code:

```
compose :: (b → c) → (a → b) → a → c
compose f g = \x → f (g x)

main = do
    let multiplyBy2 = (* 2)
```

```
let subtract3 = subtract 3
let composedFunc = compose multiplyBy2 subtract3
print (map composedFunc [1, 2, 3, 4])
print (map composedFunc [5, 6, 7, 8])
print (map composedFunc [0, -1, 2, 4])
print (map composedFunc [10, 15, 20])
```

Input/Output Examples:

- **Input 1:** `map composedFunc [1, 2, 3, 4]`

Output 1: `[-1, 1, 3, 5]`

Explanation:

- For each element:

- $(1 - 3) * 2 = -4$

- $(2 - 3) * 2 = -2$

- $(3 - 3) * 2 = 0$

- $(4 - 3) * 2 = 2$

- **Input 2:** `map composedFunc [5, 6, 7, 8]`

Output 2: `[4, 6, 8, 10]`

Explanation:

- For each element:

- $(5 - 3) * 2 = 4$

- $(6 - 3) * 2 = 6$

- $(7 - 3) * 2 = 8$

- $(8 - 3) * 2 = 10$

- **Input 3:** `map composedFunc [0, -1, 2, 4]`

Output 3: `[-6, -8, -2, 2]`

Explanation:

- For each element:

- $(0 - 3) * 2 = -6$

- $(-1 - 3) * 2 = -8$
 - $(2 - 3) * 2 = -2$
 - $(4 - 3) * 2 = 2$
- **Input 4:** `map composedFunc [10, 15, 20]`

Output 4: `[14, 24, 34]`

Explanation:

- For each element:

- $(10 - 3) * 2 = 14$
- $(15 - 3) * 2 = 24$
- $(20 - 3) * 2 = 34$

Output:

```

asecomputerlab@ase-computer-lab: ~/ppl/ex6
File Edit View Search Terminal Help
asecomputerlab@ase-computer-lab:~/ppl/ex6$ gedit ques3.hs
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ghc -o 3 ques3.hs
[1 of 1] Compiling Main             ( ques3.hs, ques3.o )
Linking 3 ...
asecomputerlab@ase-computer-lab:~/ppl/ex6$ ./3
[-4,-2,0,2]
[4,6,8,10]
[-6,-8,-2,2]
[14,24,34]
asecomputerlab@ase-computer-lab:~/ppl/ex6$

```

Conclusion:

In this exercise, we implemented a curried `compose` function that allows function composition. By composing the operations of multiplying by 2 and subtracting

3, we applied this composed function to various input lists, demonstrating the versatility of function composition and currying in Haskell.

4. Currying, Filter, and Fold

Problem:

Compute the sum of all odd numbers in the list `[1, 2, 3, 4, 5, 6]`.

Code:

```
filterAndFold :: (a → Bool) → (b → a → b) → b → [a] → b
filterAndFold p f z = foldl f z . filter p

sumOddNumbers :: [Int] → Int
sumOddNumbers a = filterAndFold odd (+) 0 a

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

Explanation:

- `filterAndFold`: This is a curried function that takes a filtering predicate `p`, a folding function `f`, an initial value `z`, and a list. It first filters the list using the predicate and then applies the folding function (`f`) cumulatively to compute the result.
- `sumOddNumbers`: This function computes the sum of all odd numbers in the list by using `filterAndFold` with the predicate `odd` and the folding function `+`.

Input:

`[1, 2, 3, 4, 5, 6]`

Output:


```
input
[1 of 1] Compiling Main                ( main.hs, main.o )
Linking a.out ...
9

...Program finished with exit code 0
Press ENTER to exit console.
```

5. Map, Filter, and Fold Combination

Problem:

Filter out numbers greater than 10 from the list `[5, 12, 9, 20, 15]`, double each remaining number, and compute the product of these doubled numbers using `foldl`.

Code:

```
productDoublesLessThan10 :: [Int] → Int
productDoublesLessThan10 = foldl (*) 1 . map (*2) . filter (<= 10)

main :: IO ()
main = print (productDoublesLessThan10 [5, 12, 9, 20, 15])
```

Explanation:

- `productDoublesLessThan10`: This function filters out numbers greater than 10, doubles the remaining numbers using `map (2)`, and computes their product using `foldl (*) 1`, which multiplies the elements from left to right, starting with an initial value of 1.

Input:

```
[5, 12, 9, 20, 15]
```

Output:

```
input
[1 of 1] Compiling Main                ( main.hs, main.o )
Linking a.out ...
180

...Program finished with exit code 0
Press ENTER to exit console.
```

6. Currying, Map, and Filter

Problem:

Filter all even numbers from the list `[1, 2, 3, 4, 5, 6]`, double them, and return the result.

Code:

```
filterAndMap :: (a → Bool) → (a → b) → [a] → [b]
filterAndMap p f = map f . filter p

doubleEvenNumbers :: [Int] → [Int]
doubleEvenNumbers = filterAndMap even (*2)

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

Explanation:

- `filterAndMap` : This is a curried function that first filters the list based on a condition `p`, then applies a mapping function `f` to each of the filtered elements.
- `doubleEvenNumbers` : This function filters out even numbers and doubles them using `filterAndMap`.

Input:

```
[1, 2, 3, 4, 5, 6]
```

Output:

```
input
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking a.out ...
[4,8,12]

...Program finished with exit code 0
Press ENTER to exit console.[]
```

7. Map, Fold, and Lambda

Problem:

Convert a list of strings to their lengths using `map`, then compute the sum of all string lengths using `foldl`.

Code:

```
sumStringLengths :: [String] → Int
sumStringLengths = foldl (+) 0 . map length

main :: IO ()
main = print (sumStringLengths ["hello", "world", "haskell"])
```

Explanation:

- `sumStringLengths`: This function first maps each string to its length using `map length` and then computes the sum of all string lengths using `foldl (+) 0`.

Input:

```
["hello", "world", "haskell"]
```

Output:

```
input
[1 of 1] Compiling Main          ( main.hs, main.o )
Linking a.out ...
17

...Program finished with exit code 0
Press ENTER to exit console.
```

8. Filter, Map, and Function Composition

Problem:

Filter out numbers greater than 5 from the list `[3, 7, 2, 8, 4, 6]`, then square the remaining numbers.

Code:

```
composeFilterMap :: (a -> Bool) -> (a -> b) -> [a] -> [b]
composeFilterMap p f = map f . filter p

squareNumbersLessThan5 :: [Int] -> [Int]
squareNumbersLessThan5 = composeFilterMap (<= 5) (^2)

main :: IO ()
main = print (squareNumbersLessThan5 [3, 7, 2, 8, 4, 6])
```

Explanation:

- `composeFilterMap`: This function first filters the list based on a predicate `p` and then maps each element using the function `f`.
- `squareNumbersLessThan5`: This function filters out numbers greater than 5 and squares the remaining numbers.

Input:

```
[3, 7, 2, 8, 4, 6]
```

Output:

```
input
[1 of 1] Compiling Main                ( main.hs, main.o )
Linking a.out ...
[9,4,16]

...Program finished with exit code 0
Press ENTER to exit console.
```

9. Map, Filter, and Fold Combination

Problem:

Filter all odd numbers from the list `[1, 2, 3, 4, 5, 6]`, square them using `map`, and compute the product of the squared numbers using `foldl`.

Code:

```
productOfSquaredOdds :: [Int] → Int
productOfSquaredOdds = foldl (*) 1 . map (^2) . filter odd

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

Explanation:

- `productOfSquaredOdds`: This function filters out odd numbers, squares them using `map (^2)`, and computes their product using `foldl (*) 1`.

Input:

```
[1, 2, 3, 4, 5, 6]
```

Output:

```
input
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking a.out ...
225

...Program finished with exit code 0
Press ENTER to exit console.
```

10. IO Monad and Currying

Problem:

Ask the user for two numbers, then apply 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.

Code:

```
applyOp :: (Int → Int → Int) → [Int] → Int
applyOp op (x:xs) = foldl op x xs
applyOp _ [] = 0

main :: IO ()
main = do
  putStrLn "Enter operation (+ or *):"
  op ← getLine
  putStrLn "Enter two numbers:"
  num1 ← readLn
  num2 ← readLn
  let operation = if op == "+" then (+) else (*)
  print (applyOp operation [num1, num2])
```

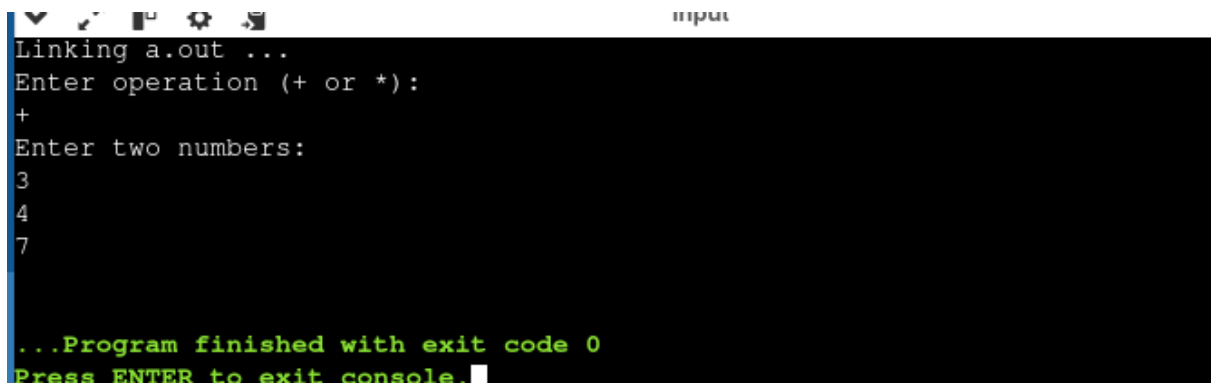
Explanation:

- `applyOp`: This is a curried function that takes an operation (`op`) and a list of integers, then applies the operation on the list using `foldl`.
- The program interacts with the user, asking for the operation and two numbers, then applies the appropriate operation (`+` or `*`) to the numbers.

Input:

`+, 3, 5`

Output:



```
Linking a.out ...
Enter operation (+ or *):
+
Enter two numbers:
3
4
7

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

This lab exercise helped in understanding and applying functional programming techniques such as currying, map, filter, fold, and function composition. These concepts are essential for manipulating lists and handling higher-order functions efficiently in functional languages like Haskell.