Principles of Programming

20CYS312 - Principles of Programming Languages

S. Shyam Balaji **CH.EN.U4CYS22045** **DATE:** 20/12/24

Objective of the Exercise

The objective of this lab exercise is to explore higher-order functions, currying, lambdas, maps, filters, folds, and IO monad in Haskell. By implementing these concepts, we aim to deepen our understanding of Haskell's functional programming paradigm and its use in solving real-world problems.

Exercise Solutions

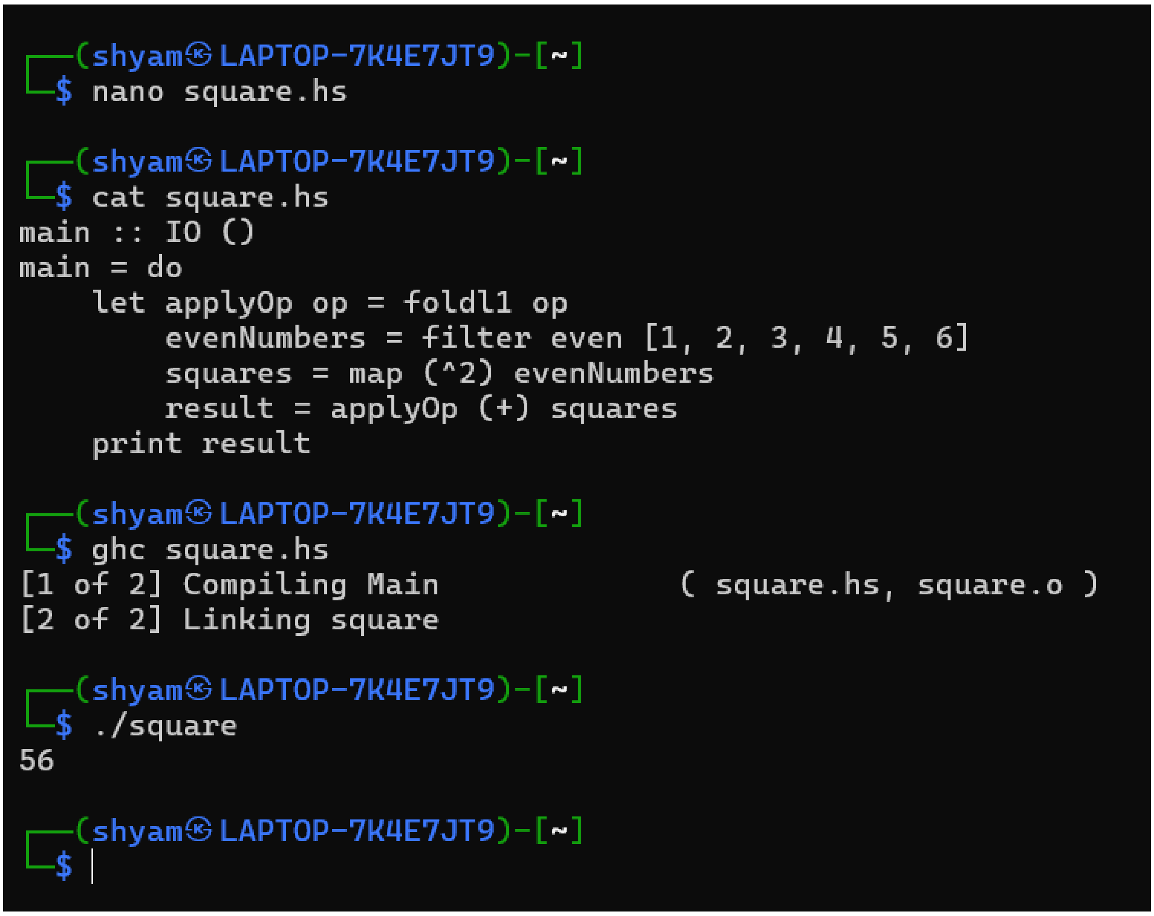
Currying, Map, and Fold

1. **Sum of Squares of Even NumbersInput:** [1, 2, 3, 4, 5, 6]**Output:** `56` **Explanation:**

```
main :: IO ()
main = do
    let applyOp op = foldl1 op
        evenNumbers = filter even [1, 2, 3, 4, 5, 6]
        squares = map (^2) evenNumbers
        result = applyOp (+) squares
    print result
```

- The program filters even numbers from the input list using `filter even`.
- Each even number is squared using `map (^2)`.
- The squares are then summed using `foldl1 (+)`, a curried version of `foldl`.

Conclusion:
This program demonstrates how currying, map, and fold can be used together to perform data processing tasks in a functional programming style.



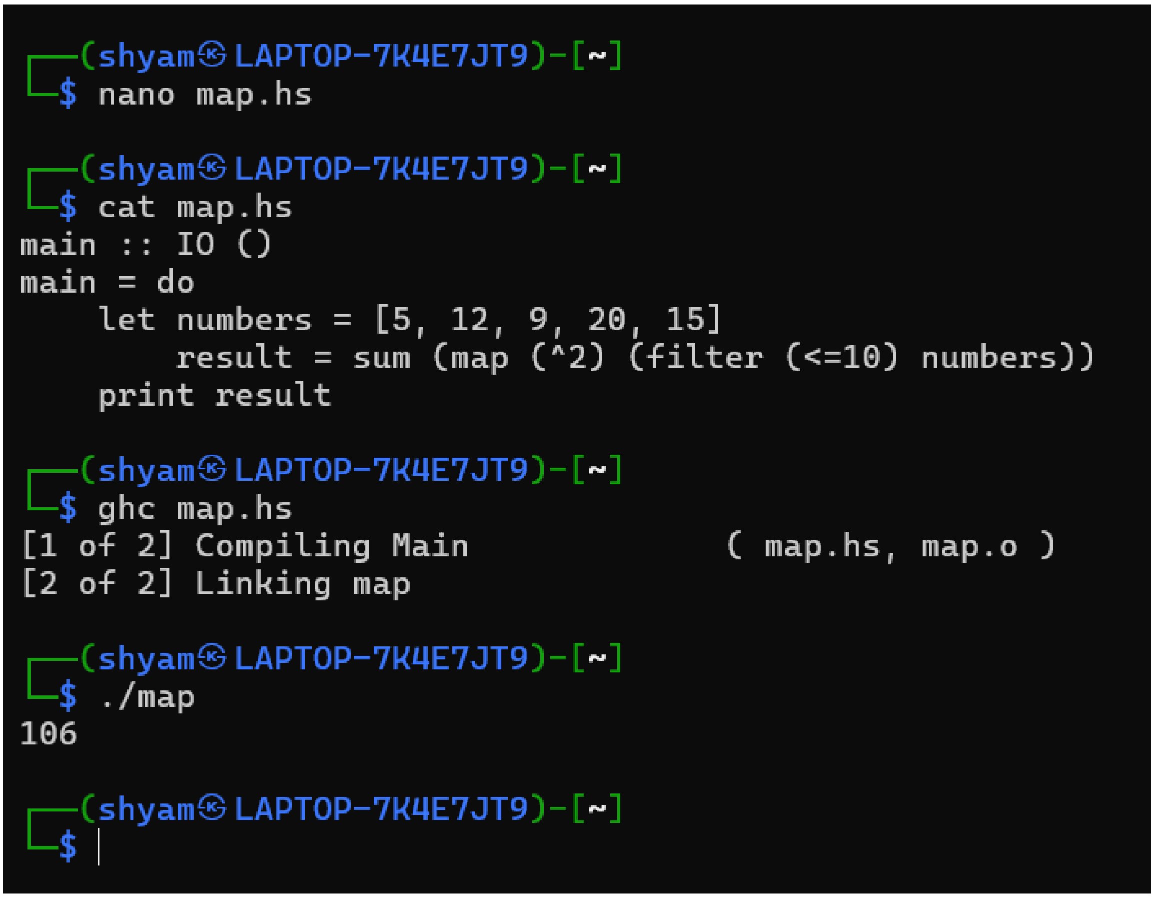
Map, Filter, and Lambda

1. **Filter and Square Numbers ≤ 10Input:** [5, 12, 9, 20, 15]**Output:** `106` **Explanation:**

```
main :: IO ()
main = do
    let numbers = [5, 12, 9, 20, 15]
        result = sum (map (^2) (filter (<=10) numbers))
    print result
```

- Filters the list to include only numbers less than or equal to 10.
- Squares each filtered number using `map (^2)`.

- Sums up the squared values using `sum`.**Conclusion:**
  This program shows how map and filter can be combined with lambda functions to process numerical data efficiently.



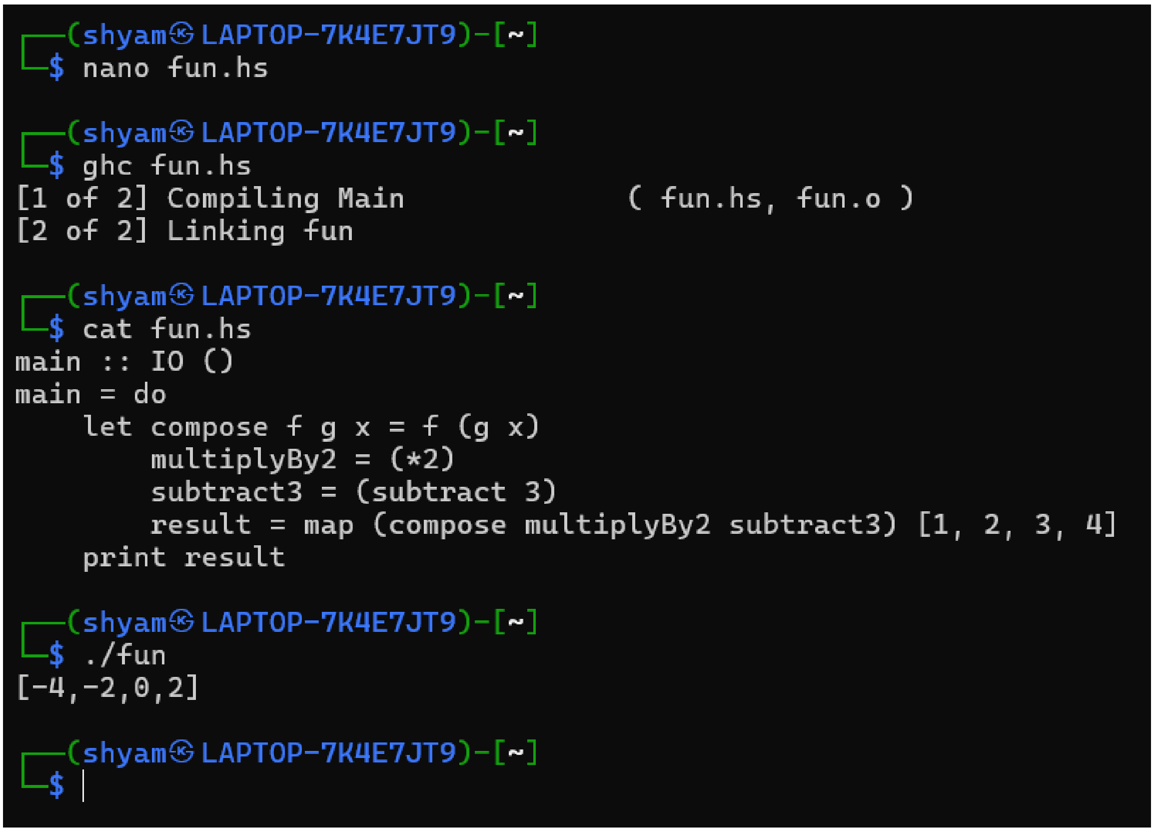
Currying, Function Composition, and Map

1. **Compose Multiply and Subtract FunctionsInput:** [1, 2, 3, 4]**Output:** `[-1, 1, 3, 5]` **Explanation:**

```
main :: IO ()
main = do
    let compose f g x = f (g x)
        multiplyBy2 = (*2)
        subtract3 = (subtract 3)
        result = map (compose multiplyBy2 subtract3) [1,
2, 3, 4]
    print result
```

- Defines a custom compose function to combine two functions.

- Each number is first reduced by 3 using `subtract3`, and then multiplied by 2 using `multiplyBy2`.
- The composed function is applied to each number using `map`.**Conclusion:**
  This demonstrates the power of function composition in Haskell to create and reuse concise operations.



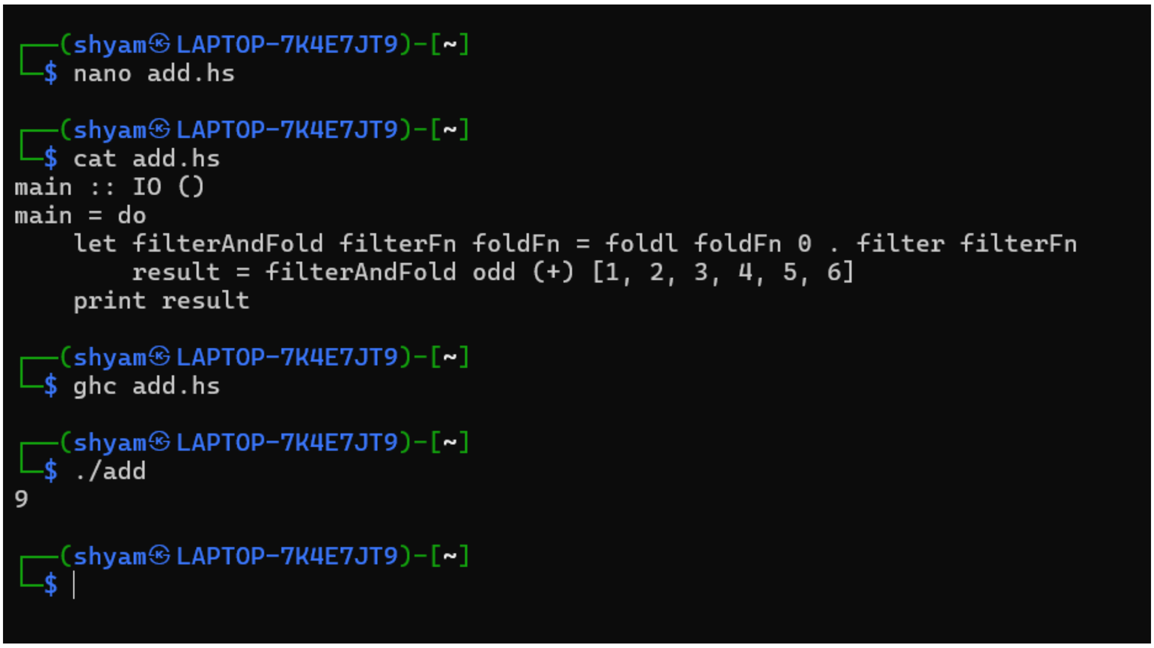
Currying, Filter, and Fold

1. **Sum of Odd NumbersInput:** [1, 2, 3, 4, 5, 6]**Output:** `9`**Explanation:**

```
main :: IO ()
main = do
    let filterAndFold filterFn foldFn = foldl foldFn 0 .
filter filterFn
        result = filterAndFold odd (+) [1, 2, 3, 4, 5,
6]
    print result
```

- Filters out odd numbers using `filter odd`.

- Sums the filtered numbers using a curried fold function.**Conclusion:** This program demonstrates how currying can be used to create reusable and concise higher-order functions.



Map, Filter, and Fold Combination

1. **Filter, Double, and Compute ProductInput:** [5, 12, 9, 20, 15]**Output:** `180` **Explanation:**

```
main :: IO ()
main = do
    let numbers = [5, 12, 9, 20, 15]
        result = foldl (*) 1 (map (*2) (filter (<=10) numbers))
    print result
```

- Filters numbers less than or equal to 10.
- Doubles each filtered number using `map (*2)`.
- Computes the product of the doubled numbers using `foldl (*) 1`.**Conclusion:**
  This program illustrates how to integrate multiple transformations and aggregations in a functional style.

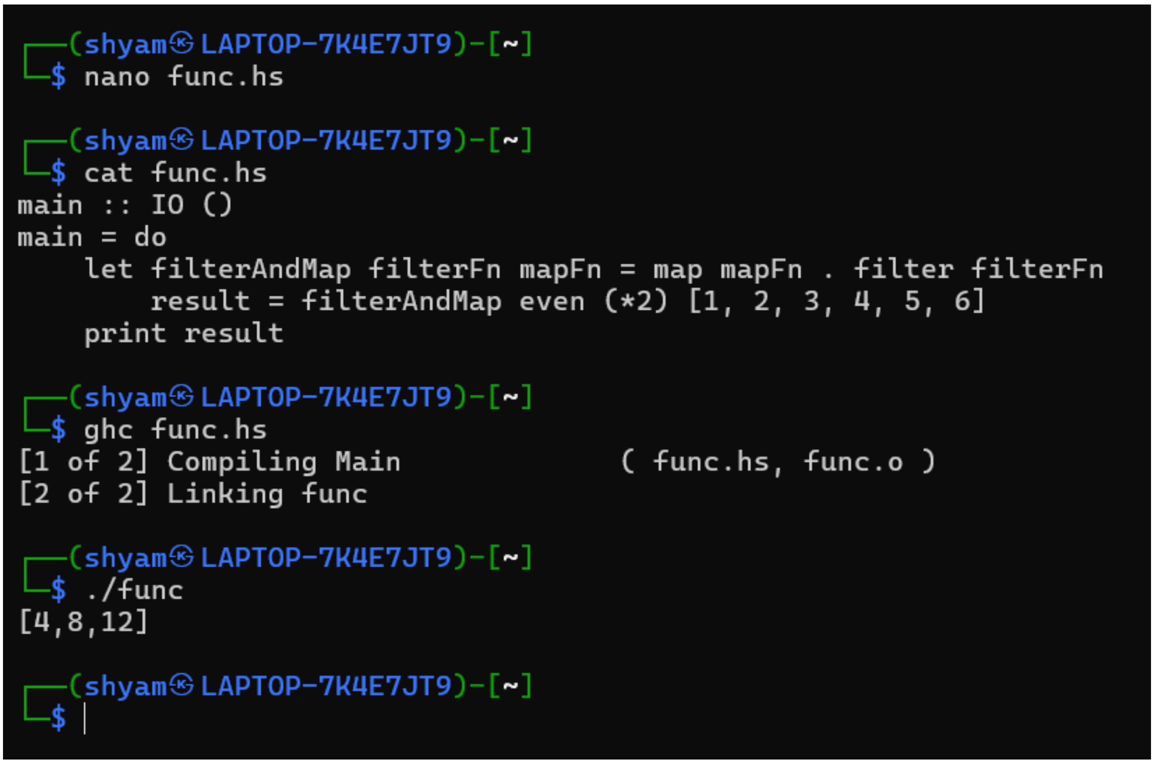
```
┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ nano double.hs

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ cat double.hs
main :: IO ()
main = do
    let numbers = [5, 12, 9, 20, 15]
        result = foldl (*) 1 (map (*2) (filter (<=10) numbers))
    print result

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ghc double.hs
[1 of 2] Compiling Main             ( double.hs, double.o )
[2 of 2] Linking double

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ./double
180

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ |
```

Currying, Map, and Filter

1. **Filter and Double Even NumbersInput:** [1, 2, 3, 4, 5, 6]**Output:** `[4, 8, 12]` **Explanation:**

    ```
    main :: IO ()
    main = do
        let filterAndMap filterFn mapFn = map mapFn . filter
    filterFn
            result = filterAndMap even (*2) [1, 2, 3, 4, 5,
    6]
        print result
    ```

    - Filters even numbers using `filter even`.
    - Doubles the filtered numbers using `map (*2)`.**Conclusion:** This program demonstrates how filtering and mapping can be modularized and reused through higher-order functions.



Map, Fold, and Lambda

1. **Sum of String LengthsInput:** ["hello", "world", "haskell"]**Output:** `18` **Explanation:**

```
main :: IO ()
main = do
    let strings = ["hello", "shyam", "Balaji"]
        lengths = map length strings
        totalLength = foldl (+) 0 lengths
    print totalLength
```

- Calculates the length of each string in the list using `map length`.
- Computes the sum of lengths using `foldl (+) 0`.**Conclusion:** This task demonstrates how map and fold can be used to process string-based data in Haskell.

```
┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ nano hello.hs

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ cat hello.hs
main :: IO ()
main = do
    let strings = ["hello", "shyam", "Balaji"]
        lengths = map length strings
        totalLength = foldl (+) 0 lengths
    print totalLength

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ghc hello.hs
[1 of 2] Compiling Main             ( hello.hs, hello.o )
[2 of 2] Linking hello

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ./hello
16

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ 
```

Filter, Map, and Function Composition

1. **Filter ≤ 5 and SquareInput:** [3, 7, 2, 8, 4, 6]**Output:** `[9, 4, 16]`**Explanation:**

```
main :: IO ()
main = do
    let composeFilterMap filterFn mapFn = map mapFn . filter filterFn
        result = composeFilterMap (<=5) (^2) [3, 7, 2, 8, 4, 6]
    print result
```

- Filters numbers less than or equal to 5.
- Squares each filtered number using `map (^2)`.**Conclusion:**
  This program highlights the use of function composition to achieve clean and modular logic.

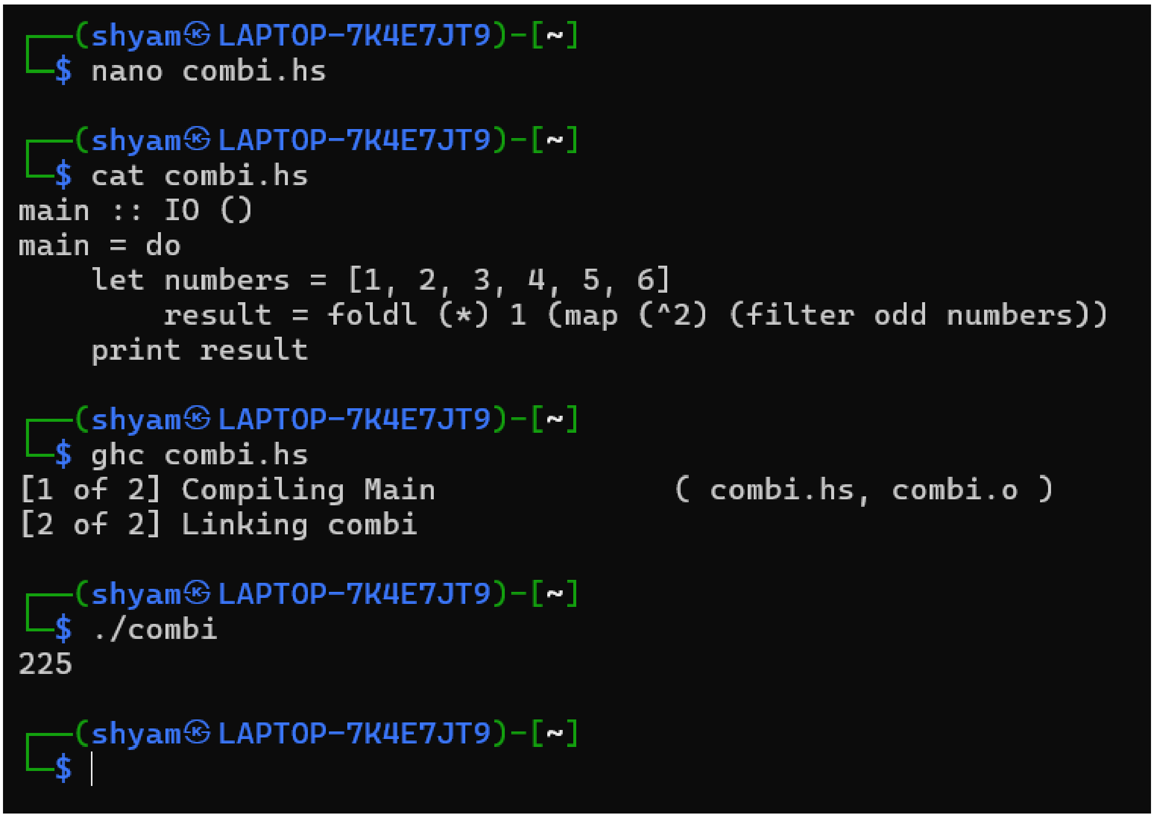
```
┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ nano fil.hs

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ cat fil.hs
main :: IO ()
main = do
    let composeFilterMap filterFn mapFn = map mapFn . filter filterFn
        result = composeFilterMap (<=5) (^2) [3, 7, 2, 8, 4, 6]
    print result

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ghc fil.hs
[1 of 2] Compiling Main             ( fil.hs, fil.o )
[2 of 2] Linking fil

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ./fil
[9,4,16]

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ |
```

Map, Filter, and Fold Combination

1. **Product of Squares of Odd NumbersInput:** [1, 2, 3, 4, 5, 6]**Output:** `225` **Explanation:**

    ```
    main :: IO ()
    main = do
        let numbers = [1, 2, 3, 4, 5, 6]
            result = foldl (*) 1 (map (^2) (filter odd numbers))
        print result
    ```

    - Filters odd numbers using `filter odd`.
    - Squares the filtered numbers using `map (^2)`.
    - Computes the product using `foldl (*) 1`.**Conclusion:**
      This task shows how Haskell's functional programming primitives can be combined to process and aggregate data.



IO Monad and Currying

1. **User-Input-Based Operation**

```
main :: IO ()
main = do
    putStrLn "Enter operation (+ or *):"
    op <- getLine
    putStrLn "Enter two numbers:"
    num1 <- readLn
    num2 <- readLn
    let applyOp "+" = (+)
        applyOp "*" = (*)
        result = applyOp op num1 num2
    print result
```

Input: "+" and 23, 45

Output: 68

Explanation:

- Takes input for an operation (`+` or ` `) and two numbers.

- Applies the corresponding operation using a curried function.
- Prints the result.**Conclusion:**
  This program illustrates interactive programming in Haskell using the IO Monad and curried functions for dynamic behavior.

```
┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ nano two.hs

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ cat two.hs
main :: IO ()
main = do
    putStrLn "Enter operation (+ or *):"
    op <- getLine
    putStrLn "Enter two numbers:"
    num1 <- readLn
    num2 <- readLn
    let applyOp "+" = (+)
        applyOp "*" = (*)
        result = applyOp op num1 num2
    print result

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ghc two.hs

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ ./two
Enter operation (+ or *):
+
Enter two numbers:
23
45
68

┌──(shyam㉿LAPTOP-7K4E7JT9)-[~]
└─$ 
```