Principles of programming languages

Sub Code: 20CYS312 **Name: Chitra Harini**

Roll No:CH.EN.U4CYS22010 **Date: 10-01-2025**

GITHUB LINK: **Haskell/Haskell_Lab6 at main · Harini-chitra/Haskell**

Lab 6

Objective: To implement basic mathematical and functional operations in Haskell, covering topics like Higher-Order Functions, Currying, Lambdas, Maps, Filters, Folds, and the IO Monad.

Exercise 1: Currying

1. **Curried Function with Complex Logic:** Define a curried function that takes a multiplier and a list of numbers. It returns a function that, when applied to another list, multiplies each element by the multiplier.

Haskell Code:

```
createMultiplier :: Int -> [Int] -> [Int]
createMultiplier multiplier = map (* multiplier)
exampleCreateMultiplier = createMultiplier 3 [1, 2, 3]
main :: IO ()
main = do
   print exampleCreateMultiplier
```

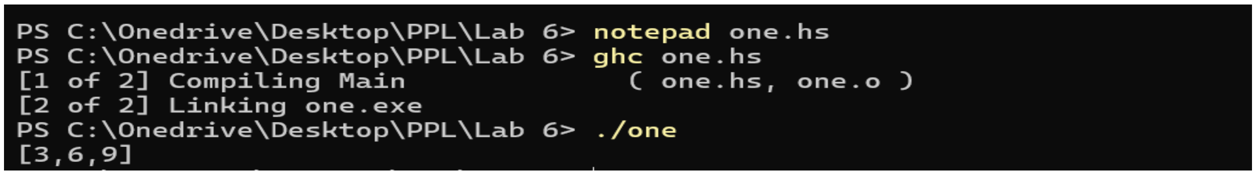
Explanation of code:

- createMultiplier is a curried function that takes a multiplier and applies it to a list of numbers.
- The map (* multiplier) multiplies each element in the list by the multiplier.

I/O Examples:

createMultiplier 3 [1, 2, 3] -- returns [3, 6, 9]

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad one.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc one.hs
[1 of 2] Compiling Main             ( one.hs, one.o )
[2 of 2] Linking one.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./one
[3,6,9]
```

2. **Partial Application in Higher-Order Functions:** Write a curried function that takes an operation (e.g., addition or multiplication) and applies it to a list of numbers. Partially apply this function to the operation (+) and apply it to the list [1, 2, 3, 4] to compute the sum.

Haskell Code:

```
applyOperation :: (Int -> Int -> Int) -> [Int] -> Int
applyOperation op = foldl op 0
exampleApplyOperation = applyOperation (+) [1, 2, 3, 4]
main :: IO ()
main = do
   print exampleApplyOperation
```

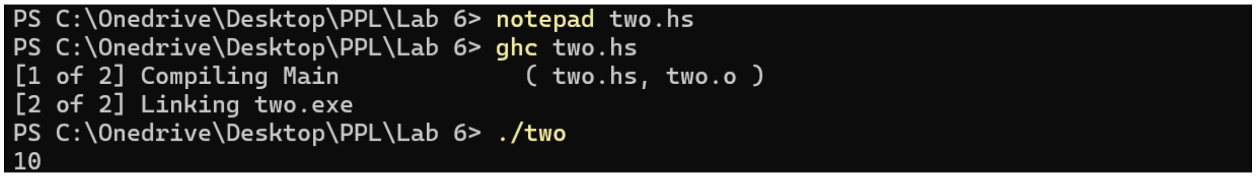
Explanation of code:

- applyOperation is a higher-order function that takes a binary operation and applies it to a list using foldl.
- Partial application with (+) allows summing the list.

I/O Examples:

```
applyOperation (+) [1, 2, 3, 4] -- returns 10
```

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad two.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc two.hs
[1 of 2] Compiling Main             ( two.hs, two.o )
[2 of 2] Linking two.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./two
10
```

3. **Currying with Function Composition:** Implement a curried function compose that takes two functions and returns their composition. Use this function to compose two operations: one that doubles a number and another that adds 5 to it. Then, apply the composed function to the list [1, 2, 3, 4].

Haskell Code:

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g x = f (g x)
double :: Int -> Int
double x = x * 2
addFive :: Int -> Int
addFive x = x + 5
exampleCompose = map (compose double addFive) [3]
main :: IO ()
main = do
   print exampleCompose
```

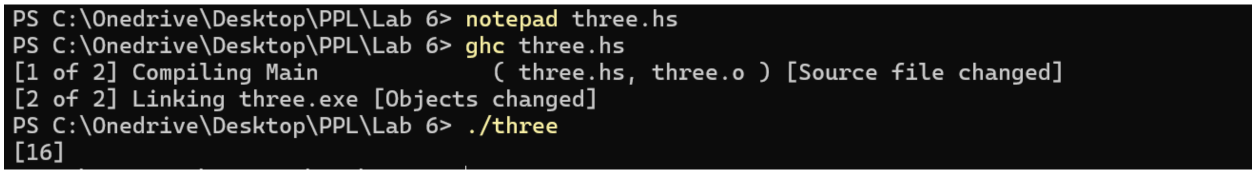
Explanation of code:

- compose combines two functions into one.
- double multiplies by 2, and addFive adds 5.
- The composed function compose double addFive is applied to each list element.

I/O Examples:

compose (*2) (+5) 3 -- returns 16 (First 3 + 5 = 8, then 8 * 2 = 16)

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad three.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc three.hs
[1 of 2] Compiling Main             ( three.hs, three.o ) [Source file changed]
[2 of 2] Linking three.exe [Objects changed]
PS C:\Onedrive\Desktop\PPL\Lab 6> ./three
[16]
```

Exercise 2: Lambdas

1. **Lambda for Counting Vowels:** Write a lambda function that counts the number of vowels in a string. Apply this function to a list of strings ["hello", "world", "haskell"] to count the vowels in each string.

Haskell Code:

```
countVowels :: [String] -> [Int]
countVowels = map (\s -> length (filter (\c -> c `elem` "aeiou") s))
exampleCountVowels = countVowels ["hello", "world", "haskell"]
main :: IO ()
main = do
  print exampleCountVowels
```

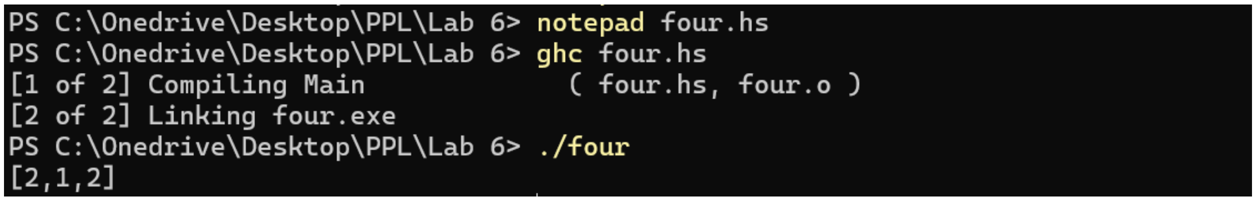
Explanation of code:

- The lambda function \c -> c elem "aeiou" checks if a character is a vowel.
- The function filter selects vowels from the string, and length counts them.
- map applies this logic to each string in the list.

I/O Examples:

countVowels ["hello", "world", "haskell"] -- returns [2, 1, 2].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad four.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc four.hs
[1 of 2] Compiling Main             ( four.hs, four.o )
[2 of 2] Linking four.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./four
[2,1,2]
```

2. **Lambda for Swapping Tuples:** Given a list of tuples [(1, 2), (3, 4), (5, 6)], use a lambda function to swap the elements in each tuple.

Haskell Code:

```
swapTuples :: [(Int, Int)] -> [(Int, Int)]
swapTuples = map (\(x, y) -> (y, x))
exampleSwapTuples = swapTuples [(1, 2), (3, 4), (5, 6)]
main :: IO ()
main = do
   print exampleSwapTuples
```

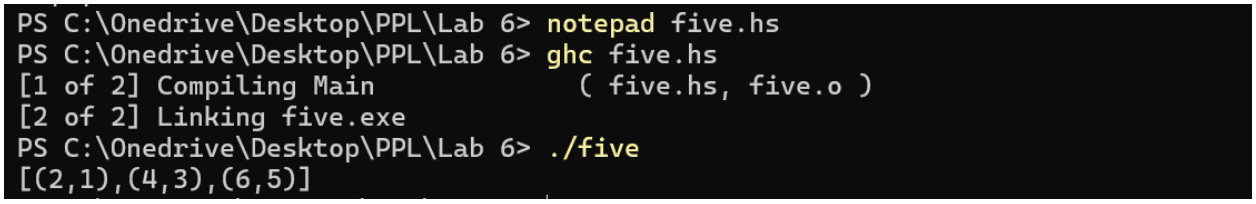
Explanation of code:

- The lambda function \(x, y) -> (y, x) swaps the elements in each tuple.
- map applies this lambda function to all tuples in the list.

I/O Examples:

swapTuples [(1, 2), (3, 4), (5, 6)] -- returns [(2, 1), (4, 3), (6, 5)].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad five.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc five.hs
[1 of 2] Compiling Main             ( five.hs, five.o )
[2 of 2] Linking five.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./five
[(2,1),(4,3),(6,5)]
```

3. **Lambda for Filtering Numbers Greater Than a Threshold:** Define a lambda function that filters out numbers greater than a given threshold from a list. Use it on the list [10, 25, 15, 8, 30] with a threshold of 20.

Haskell Code:

```
filterGreaterThan :: Int -> [Int] -> [Int]
filterGreaterThan threshold = filter (\x -> x > threshold)
exampleFilterGreaterThan = filterGreaterThan 20 [10, 25, 15, 8, 30]
main :: IO ()
main = do
   print exampleFilterGreaterThan
```

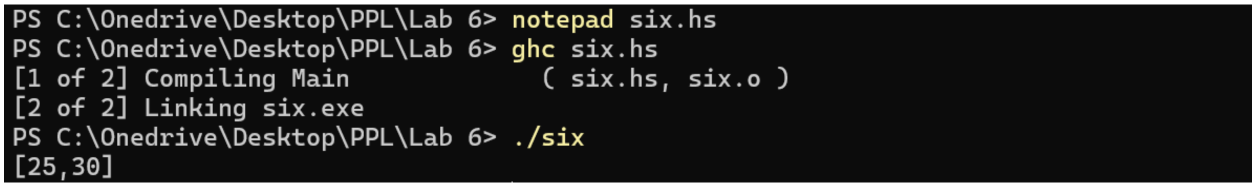
Explanation of code:

- The lambda function \x -> x > threshold checks if a number is greater than the given threshold.
- The function filter selects numbers that satisfy this condition.

I/O Examples:

filterGreaterThan 20 [10, 25, 15, 8, 30] -- returns [25, 30].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad six.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc six.hs
[1 of 2] Compiling Main             ( six.hs, six.o )
[2 of 2] Linking six.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./six
[25,30]
```

Exercise 3: Maps

1. **Map for Squaring Numbers:** Define a function square and use map to apply it to the list [1, 2, 3, 4, 5] to return a list of squares.

Haskell Code:

```
squareList :: [Int] -> [Int]
squareList = map (\x -> x * x)
exampleSquareList = squareList [1, 2, 3, 4, 5]
main :: IO ()
main = do
  print exampleSquareList
```

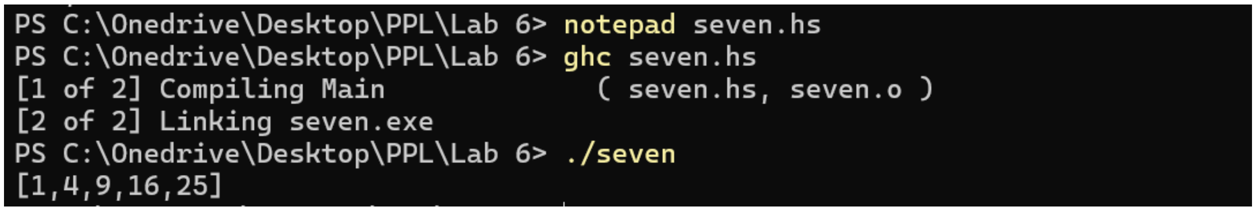
Explanation of code:

- The lambda function \x -> x * x computes the square of a number.
- map applies this function to all elements of the list.

I/O Examples:

squareList [1, 2, 3, 4, 5] -- returns [1, 4, 9, 16, 25].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad seven.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc seven.hs
[1 of 2] Compiling Main             ( seven.hs, seven.o )
[2 of 2] Linking seven.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./seven
[1,4,9,16,25]
```

2. **Map and Function Composition:** Use map to apply the function (x -> (x * 3) + 5) to each element of the list [1, 2, 3, 4].

Haskell Code:

```
transformList :: [Int] -> [Int]
transformList = map (\x -> (x * 3) + 5)
exampleTransformList = transformList [1, 2, 3, 4]
main :: IO ()
main = do
   print exampleTransformList
```

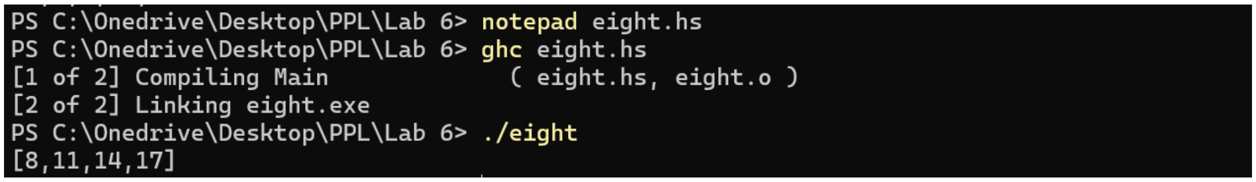
Explanation of code:

- The lambda function \x -> (x * 3) + 5 transforms each element by first multiplying it by 3, then adding 5.
- map applies this transformation to each element of the list.

I/O Examples:

transformList [1, 2, 3, 4] -- returns [8, 11, 14, 17].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad eight.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc eight.hs
[1 of 2] Compiling Main             ( eight.hs, eight.o )
[2 of 2] Linking eight.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./eight
[8,11,14,17]
```

3. **Map for Flattening and Squaring Nested Lists:** Given a list of lists [[1, 2, 3], [4, 5, 6], [7, 8]], square each element inside the sublists.

Haskell Code:

```
squareNestedLists :: [[Int]] -> [[Int]]
squareNestedLists = map (map (^2))
exampleSquareNestedLists = squareNestedLists [[1, 2, 3], [4, 5, 6], [7, 8]]
main :: IO ()
main = do
   print exampleSquareNestedLists
```

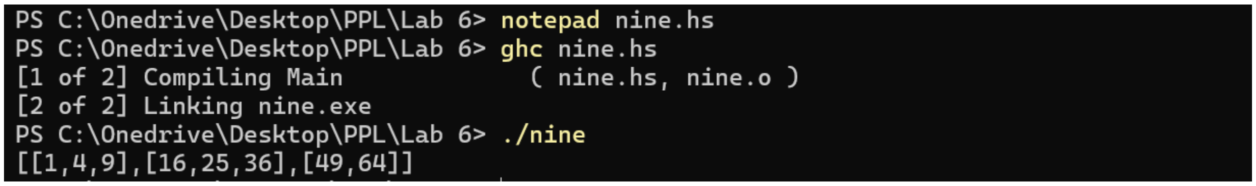
Explanation of code:

- The inner map (^2) squares each element in a sublist.
- The outer map applies this logic to all sublists.

I/O Examples:

squareNestedLists [[1, 2, 3], [4, 5, 6], [7, 8]] -- returns [[1, 4, 9], [16, 25, 36], [49, 64]].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad nine.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc nine.hs
[1 of 2] Compiling Main             ( nine.hs, nine.o )
[2 of 2] Linking nine.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./nine
[[1,4,9],[16,25,36],[49,64]]
```

Exercise 4: Filters

1. **Filter for Even Numbers:** Use filter to extract all even numbers from the list [1, 2, 3, 4, 5, 6].

Haskell Code:

```
filterEven :: [Int] -> [Int]
filterEven = filter (\x -> x `mod` 2 == 0)
exampleFilterEven = filterEven [1, 2, 3, 4, 5, 6]
main :: IO ()
main = do
   print exampleFilterEven
```

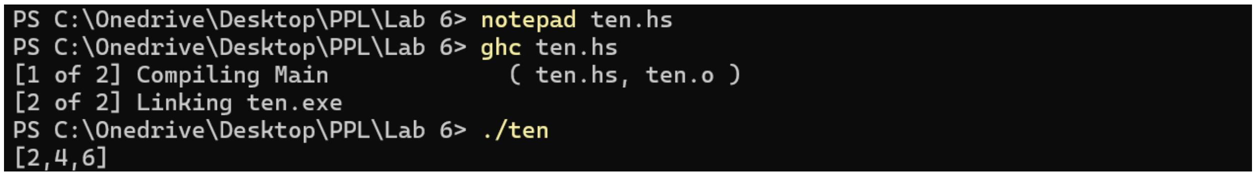
Explanation of code:

- The lambda function \x -> x mod 2 == 0 checks if a number is even.
- filter selects only even numbers from the list.

I/O Examples:

filterEven [1, 2, 3, 4, 5, 6] -- returns [2, 4, 6].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad ten.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc ten.hs
[1 of 2] Compiling Main             ( ten.hs, ten.o )
[2 of 2] Linking ten.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./ten
[2,4,6]
```

2. **Filter for Positive Numbers:** Given the list [-1, 2, -3, 4, 5, -6], use filter to return all positive numbers.

Haskell Code:

```
filterPositive :: [Int] -> [Int]
filterPositive = filter (\x -> x > 0)
exampleFilterPositive = filterPositive [-1, 2, -3, 4, 5, -6]
main :: IO ()
main = do
   print exampleFilterPositive
```

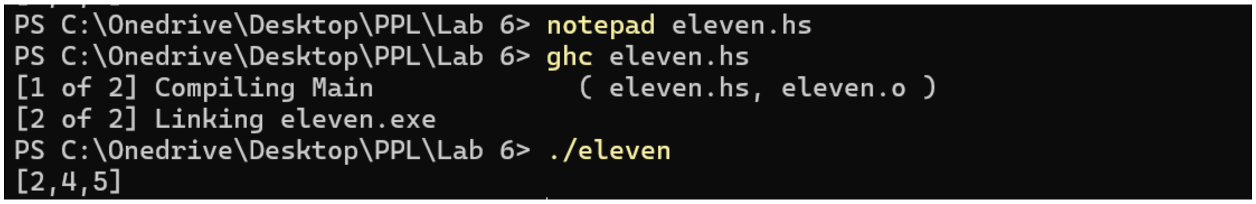
Explanation of code:

- The lambda function \x -> x > 0 checks if a number is positive.
- filter selects numbers that satisfy this condition.

I/O Examples:

filterPositive [-1, 2, -3, 4, 5, -6] -- returns [2, 4, 5].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad eleven.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc eleven.hs
[1 of 2] Compiling Main             ( eleven.hs, eleven.o )
[2 of 2] Linking eleven.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./eleven
[2,4,5]
```

3. **Filter Strings Based on Length:** Use filter to extract strings longer than 4 characters from the list ["apple", "dog", "banana", "cat", "elephant"].

Haskell Code:

```
filterLongStrings :: [String] -> [String]
filterLongStrings = filter (\s -> length s > 4)
exampleFilterLongStrings = filterLongStrings ["apple", "dog", "banana", "cat", "elephant"]
main :: IO ()
main = do
   print exampleFilterLongStrings
```

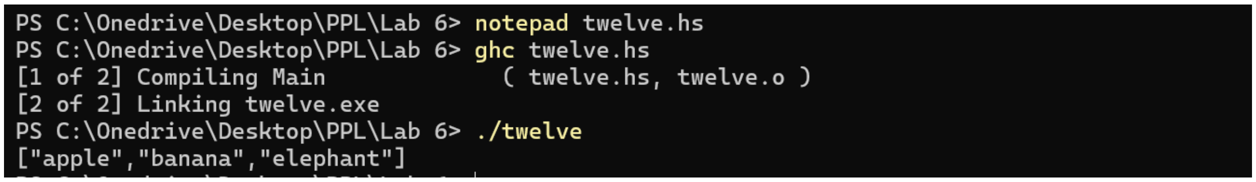
Explanation of code:

- The lambda function \s -> length s > 4 checks if the length of a string is greater than 4.
- filter selects strings that satisfy this condition.

I/O Examples:

filterLongStrings ["apple", "dog", "banana", "cat", "elephant"] -- returns ["apple", "banana", "elephant"].

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad twelve.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc twelve.hs
[1 of 2] Compiling Main             ( twelve.hs, twelve.o )
[2 of 2] Linking twelve.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./twelve
["apple","banana","elephant"]
```

Exercise 5: Folds

1. **Fold to Calculate Sum:** Use foldl to calculate the sum of the list [1, 2, 3, 4, 5].

Haskell Code:

```
calculateSum :: [Int] -> Int
calculateSum = foldl (+) 0
exampleCalculateSum = calculateSum [1, 2, 3, 4, 5]
main :: IO ()
main = do
   print exampleCalculateSum
```

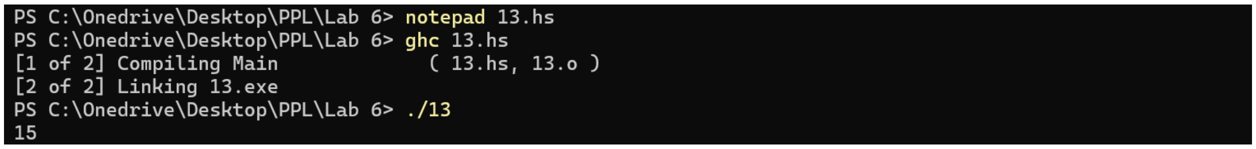
Explanation of code:

- foldl (+) 0 adds all elements of the list, starting with an initial value of 0.

I/O Examples:

calculateSum [1, 2, 3, 4, 5] -- returns 15.

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 13.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 13.hs
[1 of 2] Compiling Main             ( 13.hs, 13.o )
[2 of 2] Linking 13.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./13
15
```

2. **Fold to Concatenate Strings:** Use foldr to concatenate the list of strings ["Haskell", "is", "fun"] into a single string with spaces in between.

Haskell Code:

```
concatenateStrings :: [String] -> String
concatenateStrings = foldr (\x acc -> x ++ " " ++ acc) ""
exampleConcatenateStrings = concatenateStrings ["Haskell", "is", "fun"]
main :: IO ()
main = do
   print exampleConcatenateStrings
```

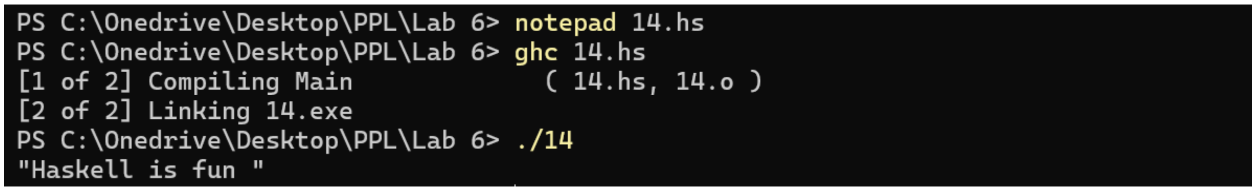
Explanation of code:

- foldr (\x acc -> x ++ " " ++ acc) "" concatenates strings with spaces in between.

I/O Examples:

concatenateStrings ["Haskell", "is", "fun"] -- returns "Haskell is fun".

Output Screenshot:


```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 14.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 14.hs
[1 of 2] Compiling Main             ( 14.hs, 14.o )
[2 of 2] Linking 14.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./14
"Haskell is fun "
```

3. **Fold to Find Maximum:** Use foldl to find the maximum number in the list [10, 20, 30, 15, 25].

Haskell Code:

```
findMax :: [Int] -> Int
findMax = foldl max 0
exampleFindMax = findMax [10, 20, 30, 15, 25]
main :: IO ()
main = do
   print exampleFindMax
```

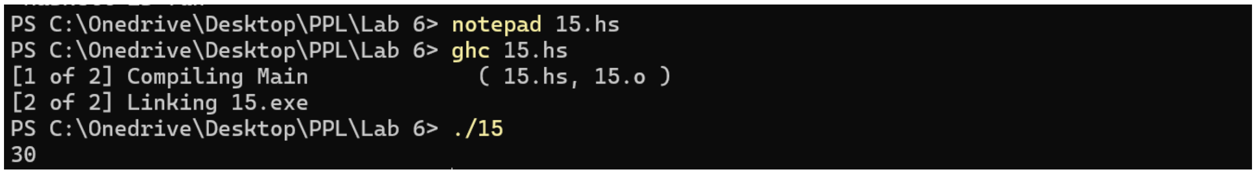
Explanation of code:

- foldl max 0 iterates through the list, keeping track of the maximum value.

I/O Examples:

findMax [10, 20, 30, 15, 25] -- returns 30.

Output Screenshot:


```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 15.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 15.hs
[1 of 2] Compiling Main             ( 15.hs, 15.o )
[2 of 2] Linking 15.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./15
30
```

Exercise 6: IO Monad

1. **Simple IO: User Input and Output:** Write a program that asks the user for their name, then prints "Hello, !".

Haskell Code:

```
main :: IO ()
main = do
   putStrLn "Enter your name:"
   name <- getLine
   putStrLn ("Hello, " ++ name ++ "!")
```

Explanation of code:

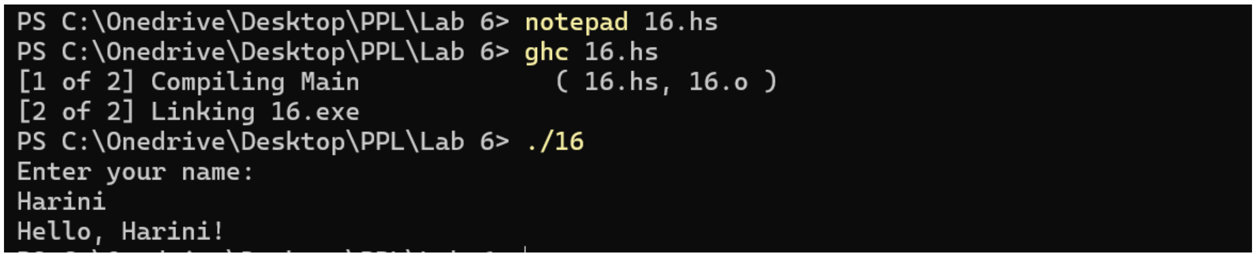
- getLine reads user input, and putStrLn outputs the greeting.

I/O Examples:

User inputs: "Harini"

Output: "Hello, Harini!"

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 16.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 16.hs
[1 of 2] Compiling Main             ( 16.hs, 16.o )
[2 of 2] Linking 16.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./16
Enter your name:
Harini
Hello, Harini!
```

2. **Chaining IO Actions:** Write an IO program that asks the user for their first and last name, then prints "Hello, !".

Haskell Code:

```
main :: IO ()
main = do
  putStrLn "Enter your first name:"
  firstName <- getLine
  putStrLn "Enter your last name:"
  lastName <- getLine
  putStrLn ("Hello, " ++ firstName ++ " " ++ lastName ++ "!")
```

Explanation of code:

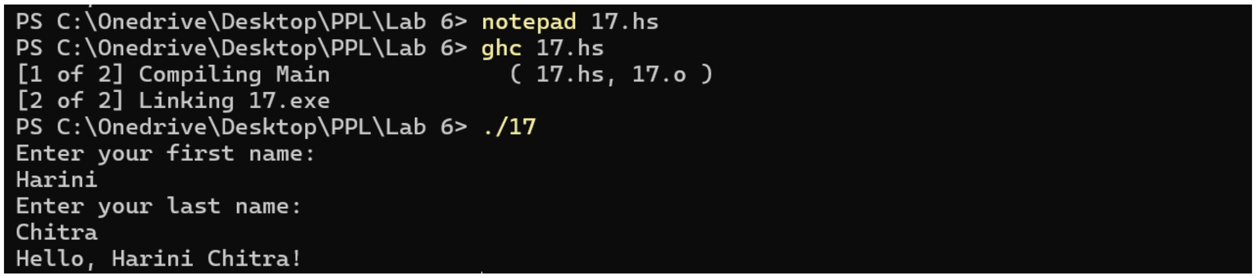
- The first getLine gathers the user's first name, while the second one gathers the last name.
- The putStrLn combines and displays a personalized greeting.

I/O Examples:

User inputs: "Harini", "Chitra"

Output: "Hello, Harini Chitra!"

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 17.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 17.hs
[1 of 2] Compiling Main             ( 17.hs, 17.o )
[2 of 2] Linking 17.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./17
Enter your first name:
Harini
Enter your last name:
Chitra
Hello, Harini Chitra!
```

3. **Reading and Writing Files:** Write a program that reads the contents of a file input.txt, counts the number of lines, and writes the result to output.txt.

Haskell Code:

```
main :: IO ()
main = do
  writeFile "output.txt" "This is a sample output text."
  content <- readFile "output.txt"
  putStrLn ("File content: " ++ content)
```

Explanation of code:

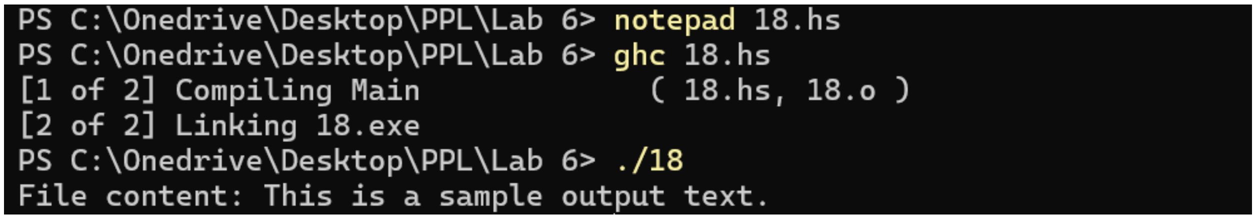
- writeFile creates or overwrites a file with the specified text.
- readFile reads the contents of the file and returns it as a string.
- putStrLn displays the content read from the file.

I/O Examples:

File "output.txt" is created with the content: "This is a sample output text."

Output: "File content: This is a sample output text."

Output Screenshot:



```
PS C:\Onedrive\Desktop\PPL\Lab 6> notepad 18.hs
PS C:\Onedrive\Desktop\PPL\Lab 6> ghc 18.hs
[1 of 2] Compiling Main             ( 18.hs, 18.o )
[2 of 2] Linking 18.exe
PS C:\Onedrive\Desktop\PPL\Lab 6> ./18
File content: This is a sample output text.
```