20CYS312 - Principles of Programing Languages - Lab Exercise 5

Roll number: CH.EN.U4CYS22002

Name: S. ASWIN SREE RAM

6th-sem-labs/PPL at main · lvexol/6th-sem-labs

Contribute to lvexol/6th-sem-labs development by creating an account on GitHub.

https://github.com/lvexol/6th-sem-labs/tree/main/PPL

lvexol/**Labs**

1 Contributor · 0 Issues · 0 Stars · 0 Forks

Ex01. Define a function `isZero :: Int -> String` that checks if a number is zero or not.

Objective of the Exercise:

The goal is to learn basic pattern matching with integers in Haskell and use it to implement a function that checks if a given integer is zero.

Program Code:

```
isZero :: Int → String
isZero 0 = "Zero"
isZero _ = "Not Zero"

main :: IO ()
main = do
  print (isZero 0)
  print (isZero 5)
  print (isZero (-3))
```

Explanation of the Code:

The `isZero` function uses pattern matching to handle two cases:

1. When the input is `0`, it returns `"Zero"`.
2. For any other value (represented by `_`), it returns `"Not Zero"`.

The `main` function demonstrates the usage of `isZero` with three examples: `0`, `5`, and `-3`.

Input/Output Examples:

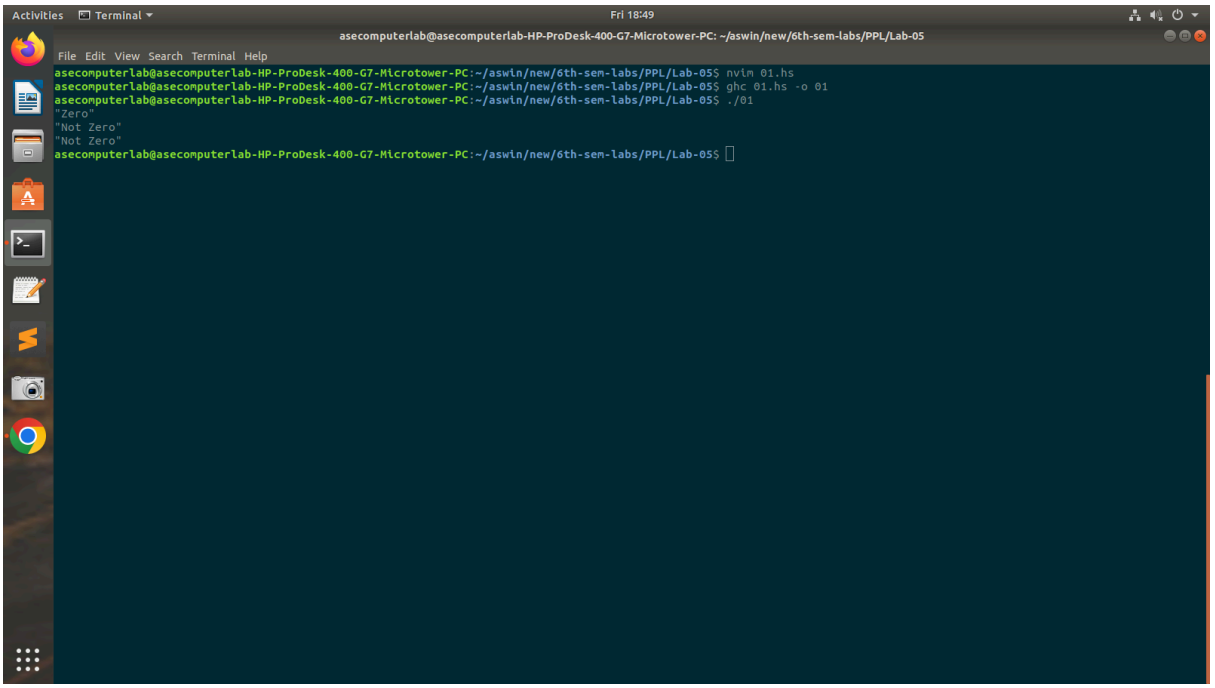
Input:

```
isZero 0
isZero 5
isZero (-3)
```

Output:

```
"Zero"
"Not Zero"
"Not Zero"
```

Screenshots:



Conclusion:

This exercise provided a simple introduction to pattern matching in Haskell, demonstrating how to differentiate between specific cases (`0`) and general cases (`_`) in a concise and readable manner.

Ex02. Define a function `countElements :: [a] -> Int` that counts the number of elements in a list.

Objective of the Exercise:

The goal is to use pattern matching on lists in Haskell to implement a function that recursively counts the number of elements in a list.

Program Code:

```
countElements :: [a] → Int
countElements [] = 0
countElements (_:xs) = 1 + countElements xs

main :: IO ()
main = do
  print (countElements [1, 2, 3])
  print (countElements [])
  print (countElements ["a", "b", "c", "d"])
```

Explanation of the Code:

The `countElements` function uses pattern matching to handle two cases:

1. When the list is empty (`[]`), it returns `0`.
2. When the list is non-empty (`(_:xs)`), it splits the list into the head (`_`) and the tail (`xs`). It increments the count by `1` and recursively calls `countElements` on the tail.

The `main` function demonstrates the usage of `countElements` with examples including a list of integers, an empty list, and a list of strings.

Input/Output Examples:

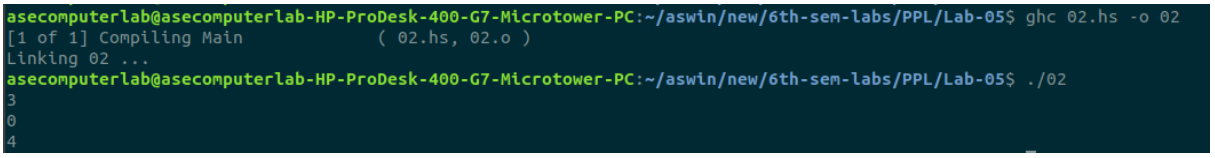
Input:

```
countElements [1, 2, 3]
countElements []
countElements ["a", "b", "c", "d"]
```

Output:

```
3
0
4
```

Screenshots:



Conclusion:

This exercise demonstrated how to use pattern matching on lists in Haskell, showcasing recursive techniques to count the elements in a list. It highlights how Haskell's pattern matching simplifies working with data structures like lists.

Ex03. Define a function `sumTuple :: (Int, Int) -> Int` to sum elements of a tuple.

Objective of the Exercise:

Learn to match and process tuple elements using Haskell's pattern matching.

Program Code:

```
sumTuple :: (Int, Int) → Int
sumTuple (x, y) = x + y

main :: IO ()
main = do
  print (sumTuple (3, 5))
  print (sumTuple (10, 20))
```

Explanation of the Code:

The `sumTuple` function extracts two integers `x` and `y` from the tuple and returns their sum. Pattern matching simplifies accessing tuple elements.

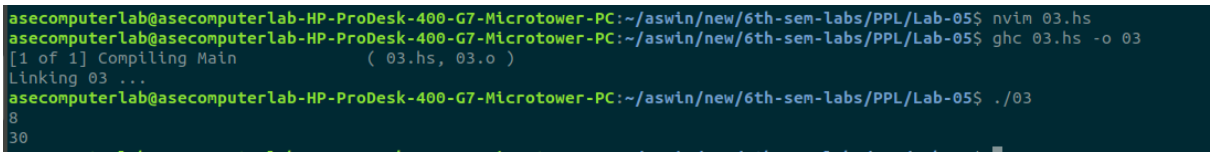
Input/Output Examples:

Input:

```
sumTuple (3, 5)
sumTuple (10, 20)
```

Output:

```
8
30
```



Ex04. Define a data type `Color` and a function `describeColor :: Color -> String`.

Objective of the Exercise:

Understand custom data types and pattern matching in Haskell.

Program Code:

```
data Color = Red | Green | Blue

describeColor :: Color → String
describeColor Red = "This is Red"
describeColor Green = "This is Green"
describeColor Blue = "This is Blue"

main :: IO ()
main = do
  print (describeColor Red)
  print (describeColor Blue)
```

Explanation of the Code:

The `Color` data type defines three values: `Red`, `Green`, and `Blue`. The `describeColor` function matches these values and returns descriptive strings.

Input/Output Examples:

Input:

```
describeColor Red
describeColor Blue
```

Output:

```
"This is Red"
"This is Blue"
```

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ nvim 04.hs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ ghc 04.hs -o 04
[1 of 1] Compiling Main             ( 04.hs, 04.o )
Linking 04 ...
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ ./04
"This is Red"
"This is Blue"
```

Ex05. Define a function `firstElement :: [a] -> String`.

Objective of the Exercise:

Practice head-tail pattern matching on lists.

Program Code:

```
firstElement :: [a] → String
firstElement [] = "Empty list"
firstElement (x:_) = "First element is " ++ show x

main :: IO ()
main = do
  print (firstElement [1, 2, 3])
  print (firstElement ([] :: [Int]))
```

Explanation of the Code:

The function matches an empty list (`[]`) and a non-empty list (`x:_`). If non-empty, it extracts the first element (`x`).

Input/Output Examples:

Input:

```
firstElement [1, 2, 3]
firstElement []
```

Output:

```
"First element is 1"
"Empty list"
```

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ nvim 05.hs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ ghc 05.hs -o 05
[1 of 1] Compiling Main             ( 05.hs, 05.o )
Linking 05 ...
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/aswin/new/6th-sem-labs/PPL/Lab-05$ ./05
"First element is 1"
"Empty list"
```

Ex06. Define a function `firstTwoElements :: [a] -> [a]`.

Objective of the Exercise:

Use pattern matching for simple list processing.

Program Code:

```
firstTwoElements :: [a] → [a]
firstTwoElements (x,y,_) = [x, y]
firstTwoElements xs = xs

main :: IO ()
main = do
  print (firstTwoElements [1, 2, 3])
  print (firstTwoElements [10])
  print (firstTwoElements []:: [Int]))
```

Explanation of the Code:

The function matches lists with two or more elements (`x:y:_`) and returns the first two. For shorter lists, it returns the original list.

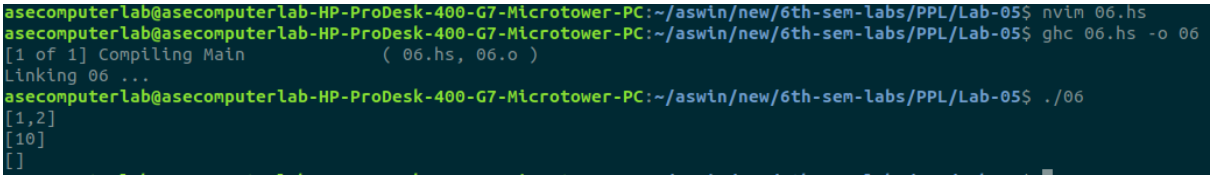
Input/Output Examples:

Input:

```
firstTwoElements [1, 2, 3]
firstTwoElements [10]
firstTwoElements []
```

Output:

```
[1, 2]
[10]
[]
```



Ex07. Define a function `describePair :: (Int, Int) -> String`.

Objective of the Exercise:

Match multiple cases with tuples.

Program Code:

```
describePair :: (Int, Int) → String
describePair (0, 0) = "Origin"
describePair (0, _) = "X-Axis"
describePair (_, 0) = "Y-Axis"
describePair _ = "Other"

quad ::(Int,Int)→ String
quad (x,y)=
   case (x,y) in
   x>0 && y<0

main :: IO ()
main = do
  print (describePair (0, 0))
  print (describePair (0, 5))
```

```
  print (describePair (3, 0))
  print (describePair (2, 3))
```

Explanation of the Code:

The function matches specific cases of `(0, 0)`, `(0, _)`, and `(_, 0)` to provide descriptive labels for the pair.

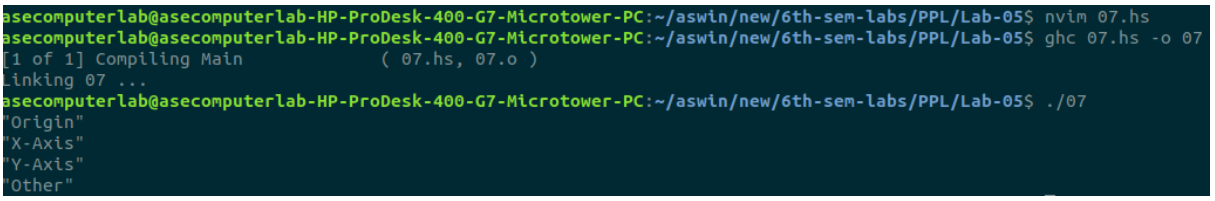
Input/Output Examples:

Input:

```
describePair (0, 0)
describePair (0, 5)
describePair (3, 0)
describePair (2, 3)
```

Output:

```
"Origin"
"X-Axis"
"Y-Axis"
"Other"
```



Ex08. Define a function `listLength :: [a] -> Int`.

Objective of the Exercise:

Calculate list length using recursion and pattern matching.

Program Code:

```
listLength :: [a] → Int
listLength [] = 0
listLength (_:xs) = 1 + listLength xs
```

```
main :: IO ()
main = do
  print (listLength [1, 2, 3])
  print (listLength ([]:: [Int]))
```

Explanation of the Code:

The function recursively matches an empty list (`[]`) and non-empty lists (`_:xs`). It counts each element and sums them.

Input/Output Examples:

Input:

```
listLength [1, 2, 3]
listLength []
```

Output:

```
3
0
```