Department of Cyber Security

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Chennai Campus

Principals of Programming Languages

Subject Code: 20CYS312 Date:2025/01/03

Name: Sushant Yadav RN:CH.EN.U4CYS22067

LAB – 5

Exercise 1: Simple Pattern Matching with Integers

Objective: Basic pattern matching with integers.

Write a function isZero :: Int -> String that:

• Returns "Zero" if the number is 0.

• Returns "Not Zero" if the number is anything other than 0.

Example Input:

isZero 0 -- Expected Output: "Zero"

isZero 5 -- Expected Output: "Not Zero"

Code:

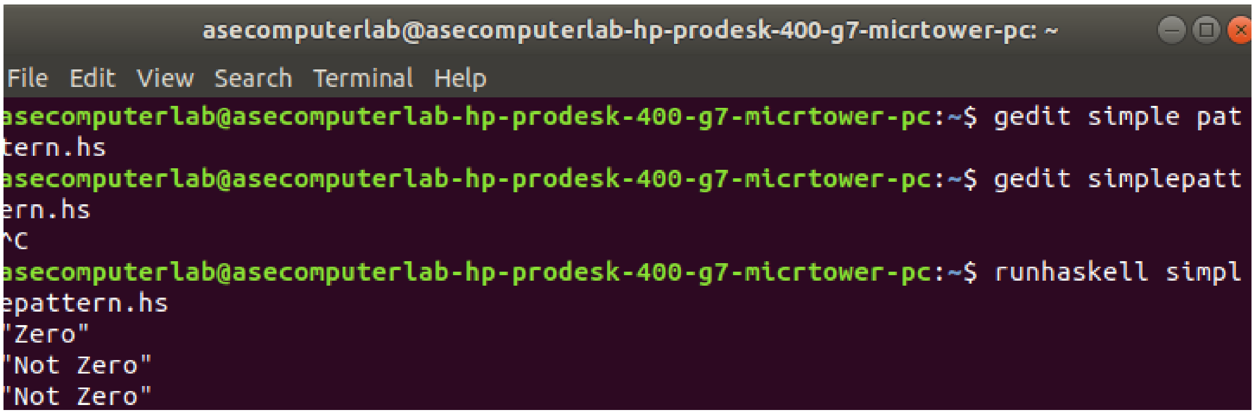
simplepattern.hs

```haskell
isZero :: Int -> String
isZero 0 = "Zero"
isZero _ = "Not Zero"
main :: IO ()
main = do
    print (isZero 0) -- Output: "Zero"
    print (isZero 5) -- Output: "Not Zero"
    print (isZero (-1)) -- Output: "Not Zero"
```

Explanation:

- The first pattern isZero 0 matches when the input is exactly 0 and returns "Zero".
- The second pattern isZero _ matches any other integer (denoted by _, which is a wildcard) and returns "Not Zero".

Output:



Conclusion:

The isZero function in Haskell demonstrates the simplicity and power of pattern matching for handling specific cases of input values. By explicitly matching 0 and using a wildcard for all other values, the code is concise and readable.

This exercise highlights:

1. **Pattern Matching:** A fundamental feature in Haskell for matching specific cases.
2. **Conciseness:** Avoids verbose conditional logic, making the function straightforward.
3. **Expressiveness:** Clearly differentiates behavior based on input values.

The function is an excellent starting point for understanding pattern matching, which is widely used in Haskell to simplify logic and improve code clarity.

Exercise 2: Pattern Matching on Lists

Objective: Use pattern matching on lists to count the number of elements.

Write a function countElements :: [a] -> Int that returns the number of elements in a list using pattern matching.

Example Input:

countElements [1, 2, 3] -- Expected Output: 3

countElements [] -- Expected Output: 0

Code:

Open listcount.ha ~/ Save

```haskell
countElements :: [a] -> Int
countElements [] = 0
countElements (_:xs) = 1 + countElements xs
main :: IO ()
main = do
    print (countElements [1, 2, 3])  -- Output: 3
    print (countElements [])         -- Output: 0
    print (countElements ["a", "b"]) -- Output: 2
```

Explanation:

1. **Base Case:**
2. The first pattern countElements [] = 0 matches an empty list and returns 0, as there are no elements to count.
3. **Recursive Case:**

The second pattern countElements (_:xs) matches a non-empty list. It splits the list into:

a. _: The head of the list (the first element, which is ignored in this case).
b. xs: The tail of the list (the rest of the list).

The function then recursively counts the elements in the tail (xs) and adds 1 for the head.

Output:

Conclusion:

This exercise demonstrates the recursive nature of pattern matching in Haskell:

- Pattern matching simplifies operations on data structures like lists.
- The function uses recursion to traverse the list and count elements, emphasizing Haskell's declarative and functional programming style.

Exercise 3: Pattern Matching with Tuples

Objective: Matching tuples with simple patterns.

Write a function sumTuple :: (Int, Int) -> Int that takes a tuple of two integers and
returns the sum of the integers.
Example Input:
sumTuple (3, 5) -- Expected Output: 8

sumTuple (10, 20) -- Expected Output: 30

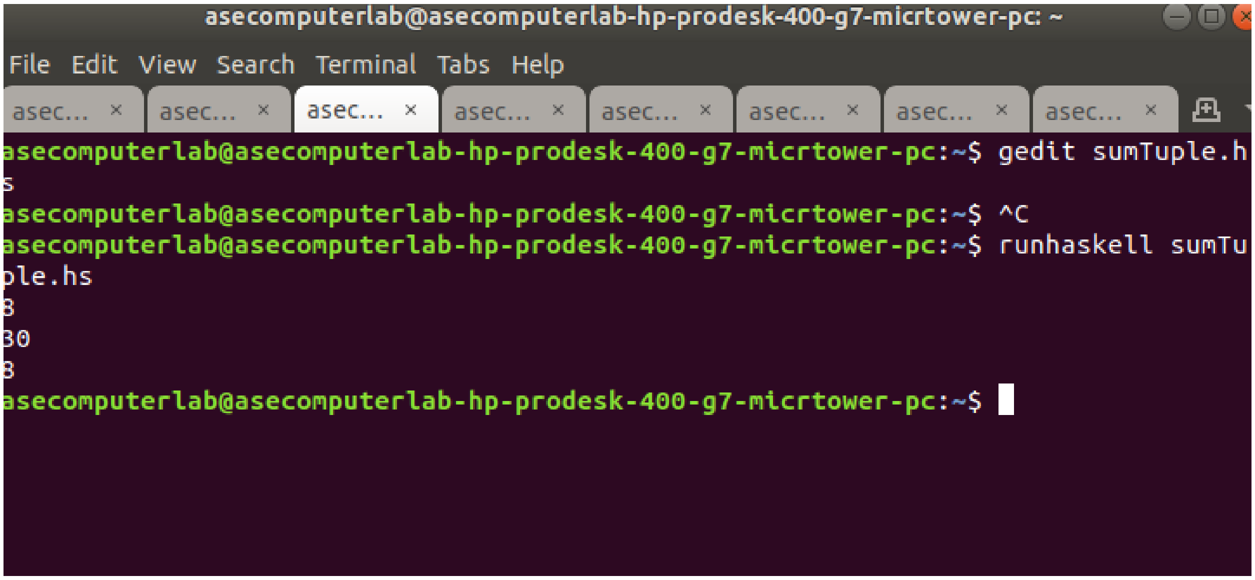
Code:

```haskell
sumTuple :: (Int, Int) -> Int
sumTuple (a, b) = a + b
main :: IO ()
main = do
    print (sumTuple (3, 5))  -- Output: 8
    print (sumTuple (10, 20)) -- Output: 30
    print (sumTuple (-7, 15)) -- Output: 8
```

Explanation:

1. **Pattern Matching on Tuples:**
   a. The pattern (a, b) matches a tuple with two integers.
   b. The variables a and b extract the first and second integers from the tuple.
2. **Computation:**
   a. The function computes the sum of a and b using a + b.

Output:



Conclusion:

This function demonstrates the use of pattern matching with tuples:

- It extracts elements of a tuple for computation in a concise and readable manner.
- This approach avoids manual indexing or destructuring, adhering to Haskell's functional and declarative paradigm.

Exercise 4: Pattern Matching on a Custom Data Type

Objective: Define a simple custom data type and pattern match on it.

Define a data type Color to represent basic colors:

data Color = Red | Green | Blue

Write a function describeColor :: Color -> String that:

- Returns "This is Red" if the color is Red.
- Returns "This is Green" if the color is Green.
- Returns "This is Blue" if the color is Blue.

Example Input:

describeColor Red -- Expected Output: "This is Red"

describeColor Blue -- Expected Output: "This is Blue"

Code:

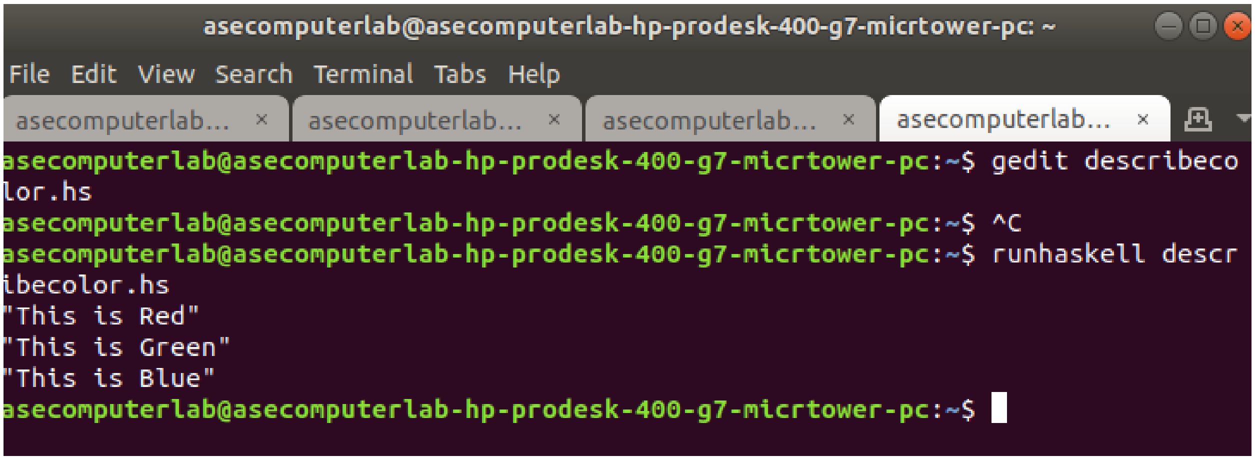
```haskell
-- Define the Color data type
data Color = Red | Green | Blue

-- Define the describeColor function
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)    -- Output: "This is Red"
    print (describeColor Green)  -- Output: "This is Green"
    print (describeColor Blue)   -- Output: "This is Blue"
```

Explanation:

1. **Custom Data Type Definition:**
   a. The Color data type defines three possible values: Red, Green, and Blue.
   b. Each value is a constructor for the Color type.
2. **Pattern Matching in the Function:**
   a. The function describeColor takes a Color value as input.
   b. It matches the input against the constructors Red, Green, and Blue.

c. For each match, it returns the corresponding descriptive string.

Output:



Conclusion:

This exercise demonstrates how to:

1. **Define a Custom Data Type:** A core feature in Haskell for modeling specific domains.
2. **Pattern Match on Constructors:** Simplifies logic by directly handling each case based on the data type's constructors.
3. **Use Readability and Type Safety:** Leveraging Haskell's type system ensures only valid Color values can be passed to describeColor.

Exercise 5: Pattern Matching with Lists (Head and Tail)

Objective: Use head-tail pattern matching on lists.

Write a function firstElement :: [a] -> String that returns:

• "Empty list" if the list is empty.

• "First element is X" if the list has at least one element, where X is the first element.

Example Input:

firstElement [1, 2, 3] -- Expected Output: "First element is 1"

firstElement [] -- Expected Output: "Empty list"

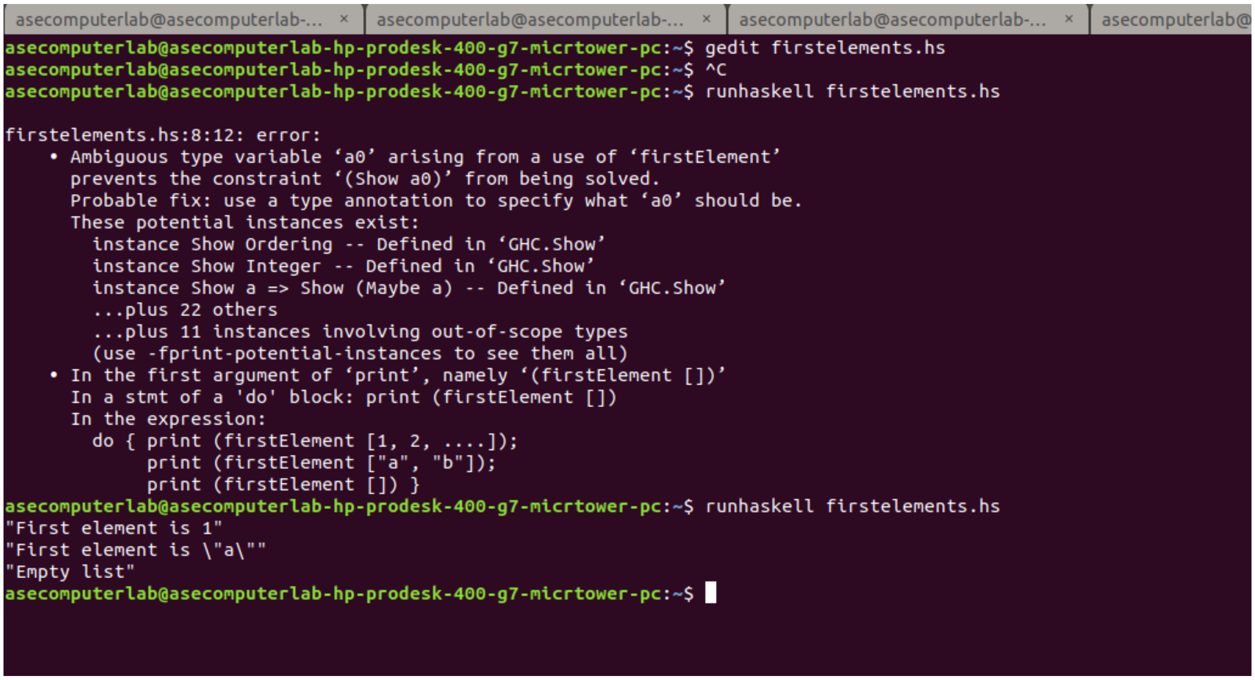
Code:

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

main :: IO ()
main = do
    print (firstElement [1, 2, 3])        -- Output: "First element is 1"
    print (firstElement ["a", "b"])       -- Output: "First element is \"a\""
    print (firstElement ([] :: [Int]))    -- Output: "Empty list"
```

Explanation:

1. **Base Case (Empty List):**
    a. The pattern [] matches an empty list.
    b. When the list is empty, the function returns "Empty list".
2. **Head-Tail Pattern Matching:**
    a. The pattern (x:_) matches a non-empty list.
    b. x is the head (first element) of the list, and _ is a wildcard for the tail (remaining elements), which is ignored.
    c. The function uses show x to convert the first element to a string (works for any type that is an instance of the Show typeclass).

Output:



Conclusion:

This exercise demonstrates:

1. **Head-Tail Pattern Matching:** A powerful way to access specific parts of a list in Haskell.
2. **Polymorphism with Typeclasses:** By requiring Show a, the function can handle lists of any type that can be converted to a string.
3. **Readability:** The function is concise and clearly distinguishes between empty and non-empty lists.

Exercise 6: Pattern Matching with Simple List Processing
Objective: Process lists using pattern matching.

Write a function firstTwoElements :: [a] -> [a] that:

• Returns the first two elements of the list if it has two or more elements.

• Returns the entire list if it has fewer than two elements.

Example Input:

firstTwoElements [1, 2, 3] -- Expected Output: [1, 2]

firstTwoElements [10] -- Expected Output: [10]

firstTwoElements [] -- Expected Output: []

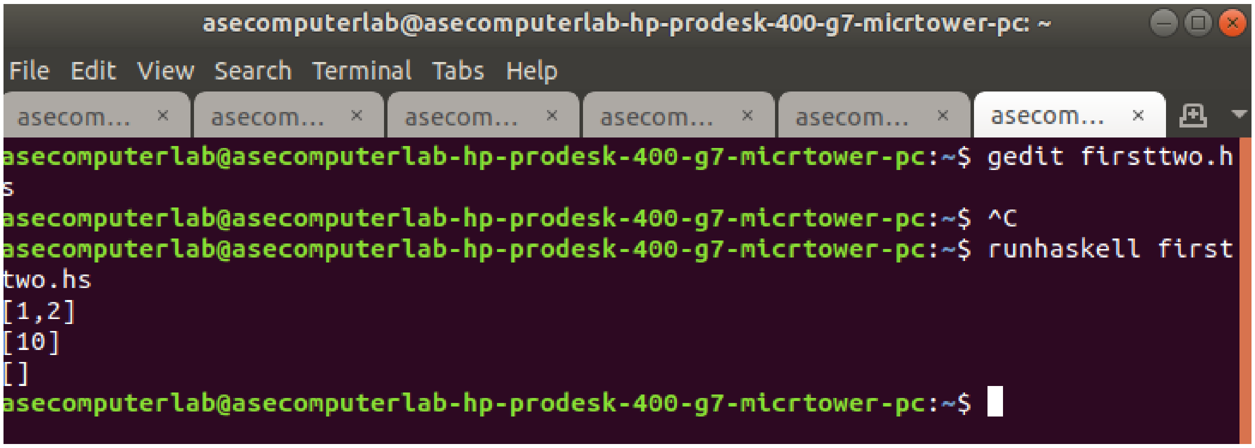
Code:

```haskell
firstTwoElements :: [a] -> [a]
firstTwoElements (x:y:_) = [x, y]  -- Case for lists with two or more elements
firstTwoElements xs = xs          -- Case for lists with fewer than two elements

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

Explanation:

1. **Pattern Matching:**
    a. (x:y:_): Matches lists with at least two elements.
        i. x is the first element.
        ii. y is the second element.
        iii. _ is a wildcard for the remaining elements, which are ignored.
        iv. The function returns [x, y], a list containing the first two elements.
    b. xs: Matches any list not already matched by the first pattern (i.e., empty lists or lists with fewer than two elements).
        i. The function simply returns the list xs as-is.
2. **Default Case:**
    a. The second pattern acts as a catch-all for cases not matched by the first pattern, ensuring the function handles all possible inputs.

Output:



Conclusion:

This exercise demonstrates:

1. **Selective Processing with Pattern Matching:** Enables concise handling of specific cases like lists with at least two elements.
2. **Catch-All Pattern:** Ensures the function handles all inputs without failure.
3. **Simplicity and Readability:** The function clearly expresses its intent with minimal code.

Exercise 7: Pattern Matching with Multiple Cases

Objective: Match against multiple patterns.

Write a function describePair :: (Int, Int) -> String that:

• Returns "Origin" if the pair is (0, 0).

• Returns "X-Axis" if the first element is 0 and the second element is any non-zero value.

• Returns "Y-Axis" if the second element is 0 and the first element is any non-zero value.

• Returns "Other" for all other pairs.

Example Input:

describePair (0, 0) -- Expected Output: "Origin"
describePair (0, 5) -- Expected Output: "X-Axis"
describePair (3, 0) -- Expected Output: "Y-Axis"
describePair (2, 3) -- Expected Output: "Other"

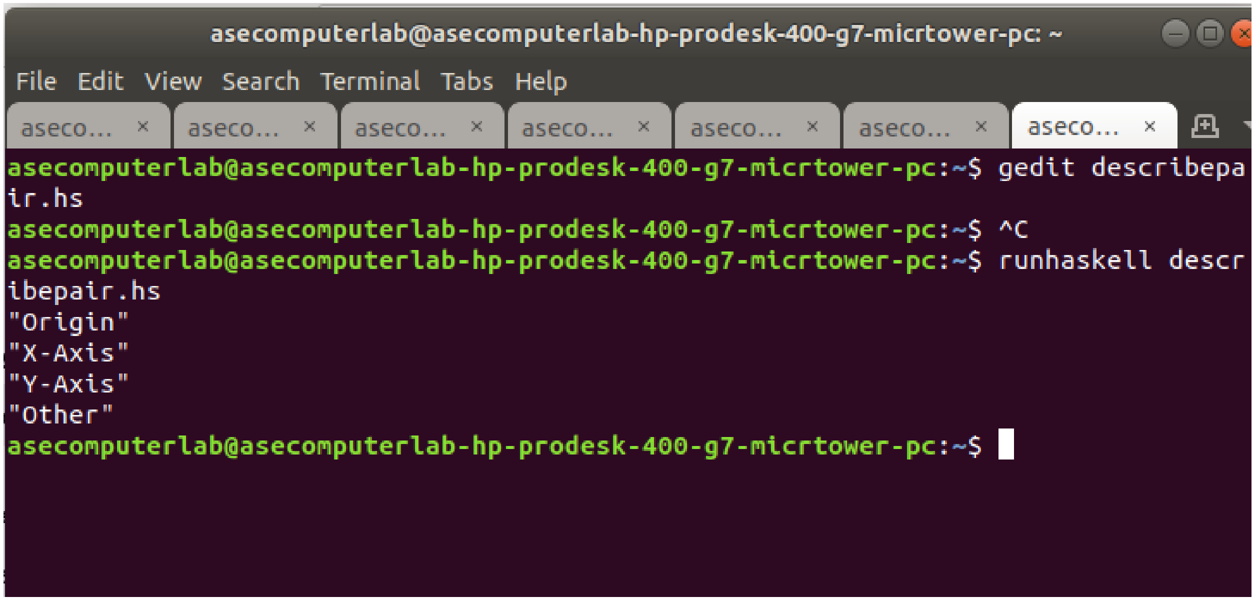
Code:

```
describePair :: (Int, Int) -> String
describePair (0, 0) = "Origin"                    -- Case for (0, 0)
describePair (0, _) = "X-Axis"                    -- Case where the first
element is 0
describePair (_, 0) = "Y-Axis"                    -- Case where the second
element is 0
describePair _ = "Other"                          -- Catch-all case for all
other pairs
main :: IO ()
main = do
    print (describePair (0, 0)) -- Output: "Origin"
    print (describePair (0, 5)) -- Output: "X-Axis"
    print (describePair (3, 0)) -- Output: "Y-Axis"
    print (describePair (2, 3)) -- Output: "Other"
```

Explanation:

1. **Pattern Matching Cases:**
    a. (0, 0): Matches the specific case where both elements are 0, returning "Origin".
    b. (0, _): Matches pairs where the first element is 0 and the second element can be any value (_ is a wildcard), returning "X-Axis".
    c. (_, 0): Matches pairs where the second element is 0 and the first element can be any value, returning "Y-Axis".
    d. _: A catch-all pattern for all other pairs, returning "Other".
2. **Order of Patterns:**
    a. Patterns are evaluated in order. The most specific patterns (e.g., (0, 0)) are placed first to ensure they match before broader patterns like (0, _) or (_, 0).

Output:



Conclusion:

This exercise highlights:

1. **Multiple Case Handling:** Pattern matching in Haskell simplifies handling distinct scenarios based on input values.
2. **Wildcard (_) Usage:** Enables flexibility for values that don't need to be explicitly checked.
3. **Readable and Declarative Code:** The function's logic is clear and concise, directly reflecting the problem requirements.

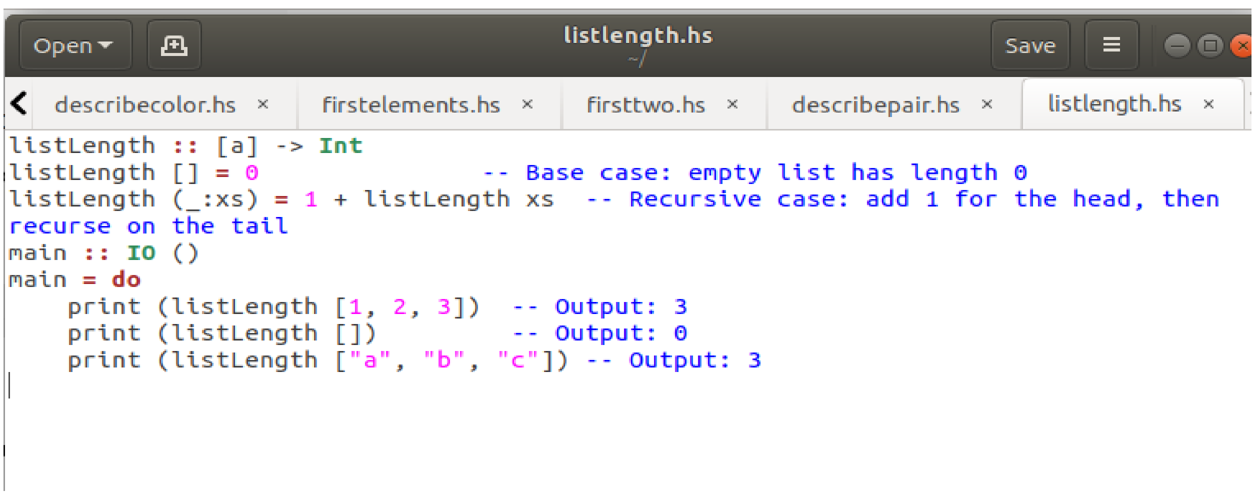
Exercise 8: Pattern Matching for List Recursion

Objective: Use recursion to work with lists.

Write a function listLength :: [a] -> Int that calculates the length of a list using recursion
and pattern matching.

Example Input:
listLength [1, 2, 3] -- Expected Output: 3
listLength [] -- Expected Output: 0

Code:



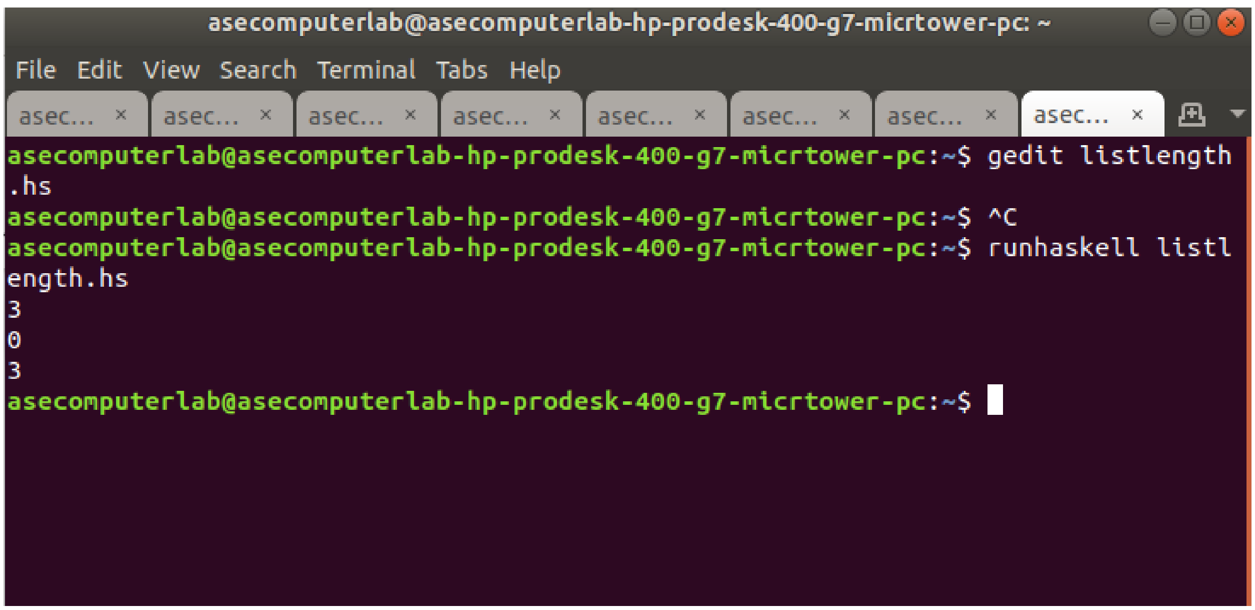

```haskell
listLength :: [a] -> Int
listLength [] = 0                    -- Base case: empty list has length 0
listLength (_:xs) = 1 + listLength xs   -- Recursive case: add 1 for the head, then
recurse on the tail
main :: IO ()
main = do
    print (listLength [1, 2, 3])  -- Output: 3
    print (listLength [])         -- Output: 0
    print (listLength ["a", "b", "c"]) -- Output: 3
```

Explanation:

1. **Base Case:**
   a. listLength [] = 0: When the list is empty ([]), the length is 0.
2. **Recursive Case:**
   a. listLength (_:xs) = 1 + listLength xs: The pattern (_:xs) matches any non-empty list, where _ is the head (first element) and xs is the tail (the rest of the list).

b. For each element in the list, we add 1 to the result of recursively calculating the length of the tail (xs).

Output:



Conclusion:

This exercise demonstrates:

1. **Recursion with Lists:** The function listLength uses recursion to traverse through the list and count its elements.
2. **Pattern Matching in Recursion:** The recursive case relies on the pattern (_:xs) to break the list down into its head and tail, while the base case handles the empty list.

3. **Functional Style:** This approach is typical in functional programming, where recursion replaces traditional loops for list processing.