20CYS312 - Principles of Programming Languages

R. Subramanian
20/12/24
CH.EN.U4CYS22043

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

Sum of Squares of Even Numbers

Input: `[1, 2, 3, 4, 5, 6]`
Output: `56`

Explanation:

```haskell
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

Filter and Square Numbers ≤ 10

Input: `[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)`, and 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

Compose Multiply and Subtract Functions

Input: `[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

Sum of Odd Numbers

Input: `[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.

IO Monad and Currying

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.

Summary

Through this exercise, we explored various functional programming techniques in Haskell, including higher-order functions, currying, function composition, lambda expressions, and the IO Monad. By integrating these concepts, we were able to perform complex data transformations in a concise and modular way. This reinforces the power and expressiveness of Haskell in functional programming paradigms.