Amrita Vishwa Vidyapeetham

Amrita School Of Computing

Principles of Programming Languages Lab

---

Course Code: 20CYS312 Name:Yash Yashuday

Roll No.:CH.EN.U4CYS22055 Date: 07-12-2024

---

1.Functions and types:

i) Define a function square :: Int -> Int that takes an integer and returns its square

A) Code:

```haskell
square :: Int -> Int
square x = x * x
```

Output:

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l square.hs
[1 of 1] Compiling Main             ( square.hs, interpreted )
Ok, one module loaded.
ghci> square 12
144
ghci> square 24
576
ghci> square 22
484
ghci> square 21
441
ghci> square 198
39204
```

Explanation:
The `square` function takes a single integer input `x` and computes its square by multiplying `x` by itself (`x * x`). This operation is straightforward and works efficiently because multiplication of integers is a basic operation in Haskell.

## Conclusion:

This function provides a simple and direct way to calculate the square of an integer, which can be useful in various mathematical or computational tasks. The concise implementation demonstrates Haskell's expressiveness.

ii)Define a function maxOfTwo :: Int -> Int -> Int that takes two integers and returns the

larger one.

Code:

```haskell
maxOfTwo :: Int -> Int -> Int
maxOfTwo x y = if x > y then x else
```

Output:

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l maxOfTwo.hs
[1 of 1] Compiling Main             ( maxOfTwo.hs, interpreted )
Ok, one module loaded.
ghci> maxOfTwo 2 4
4
ghci> maxOfTwo 123 11
123
ghci> maxOfTwo 165 1342
1342
ghci> maxOfTwo 165521 1342
165521
ghci> maxOfTwo 165521 124122
165521
```

Explanation:

The `maxOfTwo` function takes two integers as inputs, `x` and `y`.

if `x` is greater than `y`, it returns `x`.Otherwise, it returns `y`.

## Conclusion:

This function effectively determines the larger of two integers. It demonstrates the power of guards for simple comparison logic. For example:

2. Functional Composition

1. Define a function doubleAndIncrement :: [Int] -> [Int] that doubles each number in a

list and increments it by 1 using function composition

Code:

```haskell
doubleAndIncrement :: [Int] -> [Int]
doubleAndIncrement = map ((+1) . (*2))
```

Output:

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l doubleAndIncrement.hs
[1 of 1] Compiling Main             ( doubleAndIncrement.hs, interpreted )
Ok, one module loaded.
ghci> doubleAndIncrement [1,2,4,3,2,1]
[3,5,9,7,5,3]
ghci> doubleAndIncrement [11,21,41,31,21,11]
[23,43,83,63,43,23]
ghci> doubleAndIncrement [111,211,411,311,211,111]
[223,423,823,623,423,223]
ghci> doubleAndIncrement [1111,2211,4111,3211,2111,1121]
[2223,4423,8223,6423,4223,2243]
```

Explanation:

The doubleAndIncrement function uses map to apply a transformation to each element in the input list. Function composition (.) is used to combine the doubling operation ((*2)) with the increment operation ((+1)). This ensures that each number is first doubled and then incremented by 1.

Conclusion:

This implementation leverages Haskell's concise syntax for function composition, making the code both readable and efficient

2. Write a function sumOfSquares :: [Int] -> Int that takes a list of integers, squares

each element, and returns the sum of the squares using composition.

Code:

```haskell
sumOfSquares :: [Int] -> Int
sumOfSquares= sum . map (^2)
```

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l factorial.hs
[1 of 1] Compiling Main             ( factorial.hs, interpreted )
Ok, one module loaded.
ghci> factorial 1
1
ghci> factorial 2
2
ghci> factorial 3
6
ghci> factorial 4
24
ghci> factorial 12
479001600
ghci> factorial 10
3628800
```

Output:

Explaination:

The `sumOfSquares` function uses function composition to combine two operations:

1. `map (^2)`: Squares each element of the input list.
2. `sum`: Adds up all the squared values.

By composing these operations, the function efficiently processes the list in a concise manner.

## Conclusion:

This function is a clean and functional approach to computing the sum of squares of a list of integers.

3. Numbers

   1. Write a function factorial :: Int -> Int that calculates the factorial of a given number using recursion.

```haskell
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Output:

Explanation:

The `factorial` function is defined recursively:

- The base case is when `n` is 0, where the factorial is defined as 1.
- For any other positive integer `n`, the factorial is calculated as `n * factorial (n - 1).`

This recursive approach leverages Haskell's ability to handle function calls naturally and elegantly.

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l power.hs
[1 of 1] Compiling Main             ( power.hs, interpreted )
Ok, one module loaded.
ghci> power 2 3
8
ghci> power 3 4
81
ghci> power 4 5
1024
ghci> power 5 6
15625
ghci> power 6 7
279936
```

## Conclusion:

The recursive implementation of the factorial function is intuitive and demonstrates Haskell's functional nature.

2. Write a function power :: Int -> Int -> Int that calculates the power of a number (base raised to exponent) using recursion.

Code:

```haskell
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * power x (n-1)
```


Explanation:

The `power` function uses recursion to compute the result of raising a base to an exponent:

## Conclusion:

This recursive approach elegantly handles the calculation of powers, showcasing Haskell's simplicity in expressing mathematical operations.

4. Lists

1. Write a function removeOdd :: [Int] -> [Int] that removes all odd numbers from a list.

```haskell
removeOdd :: [Int] -> [Int]
removeOdd = filter even
```

```
❯ n removeOdd.hs
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l removeOdd.hs
[1 of 1] Compiling Main             ( removeOdd.hs, interpreted )
Ok, one module loaded.
ghci> removeOdd [1,2,3,4,5,6,7]
[2,4,6]
ghci> removeOdd [1,2,3,4,5,6,7,10,12,34,12,1]
[2,4,6,10,12,34,12]
ghci> removeOdd [1,3,5,7,9]
[]
ghci> removeOdd [121,311,51,71,91,10]
[10]
```

Explanation:

The `removeOdd` function filters a list to remove all odd numbers:

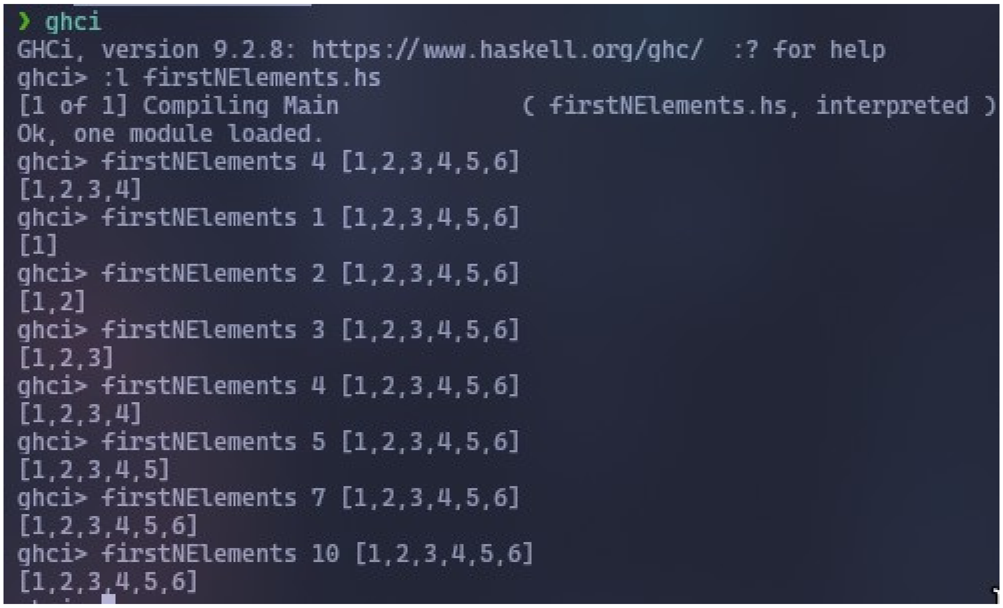
- It uses the `filter` function with the predicate `even`, which retains only the even numbers in the list.

## Conclusion:

This function is a concise and efficient way to remove odd numbers from a list in Haskell.

2. Write a function firstNElements :: Int -> [a] -> [a] that takes a number n and a list and returns the first n elements of the list.

```haskell
firstNElements :: Int -> [a] -> [a]
firstNElements n xs = take n xs
```

Explanation: 


- `take n xs` is a built-in Haskell function that returns the first `n` elements of the list `xs`.
- If `n` is greater than the length of the list, it simply returns the whole list.
- If `n` is `0`, it returns an empty list.

**Conclusion:**

This implementation of `firstNElements` is an effective and idiomatic Haskell solution for retrieving the first `n` elements from a list. Using Haskell's built-in functions enhances readability and ensures the function is both concise and efficient.

5. Tuples

1. Define  a function swap :: (a,b) → (b,a) that swaps the elements of the pair(tuple with 2 elements)

Code:

```
swap :: (a,b) → (b,a)
swap (x,y) = (y,x)
```

```
ghci> swap (11,21)
(21,11)
ghci> swap (111,211)
(211,111)
ghci> swap (1211,2111)
(2111,1211)
```

xplanation:

- The function `swap` takes a tuple `(x, y)` of type `(a, b)` as input.
- It returns a tuple `(y, x)` where the elements are swapped, thus converting the pair `(a, b)` into `(b, a)`.

  Conclusion:

The `swap` function provides a concise and efficient way to swap the elements of a tuple. It is a useful utility in many situations where pairs of values need to be reversed.

2.Write a function addPairs :: [(Int, Int)] -> [Int] that takes a list of tuples containing pairs of integers and returns a list of their sums.

Code:

```
❯ ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
ghci> :l addPairs.hs
[1 of 1] Compiling Main             ( addPairs.hs, interpreted )
Ok, one module loaded.
ghci> Here's the implementation of the function `addPairs` in Haskell:

<interactive>:2:27: error: parse error on input ‘of’
ghci>
ghci> addPairs [(1, 2), (3, 4), (5, 6)]
[3,7,11]
ghci> addPairs [(11, 21), (31, 41), (51, 61)]
[32,72,112]
ghci> addPairs [(112, 21), (31, 412), (512, 61)]
[133,443,573]
ghci> addPairs [(1412, 241), (311, 412), (512, 61)]
[1653,723,573]
```

Explaination:
**`map`**: This function applies the given function to each element of the list. In this case, the list `pairs` consists of tuples of integers.

- **Anonymous Function (`\ (x, y) -> x + y`)**: This is a lambda function that takes a tuple `(x, y)` as input and returns the sum of `x` and `y`. It works by pattern matching on the tuple and summing the two elements.
- **`pairs`**: This is the list of tuples, where each tuple contains two integers. The function applies the lambda function to each tuple in the list.

Conclusion:

Your implementation of `addPairs` is a clean and straightforward solution for summing the integers in a list of pairs. By using `map` with a lambda function, you can efficiently process each tuple and generate a list of their sums. The function is simple, clear, and flexible, making it a good choice for this task.