AMRITA VISHWA VIDYAPEETHAM, CHENNAI

20CYS312 - Principles of Programming Languages

Exercise - 02

Student Details

- **Name**: R. Subramanian
- **Roll Number**: CH.EN.U4CYS22043

- **Date**: 07 December 2024

---

1. FUNCTIONS AND TYPES

---

1. Define a function `square :: Int -> Int` that takes an integer and returns its square.

OBJECTIVE:
To calculate the square of a number using a simple function.

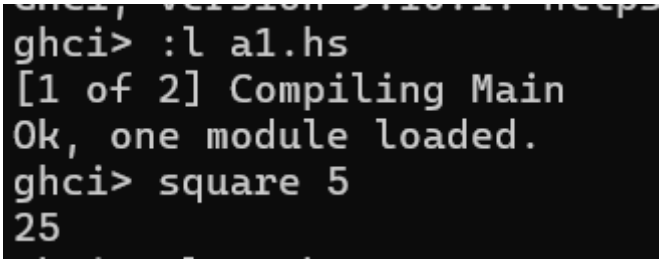
CODE:

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

EXPLANATION OF THE CODE:
The `square` function multiplies a number by itself to calculate its square.

Example:
Input: `square 5` → Output: `25`

SAMPLE INPUT & OUTPUT:



CONCLUSION:
Learned how to define a single-parameter function for basic arithmetic.

---

2. Define a function `maxOfTwo :: Int -> Int -> Int` that takes two integers and returns the larger one.

OBJECTIVE:
To determine the larger number between two integers.

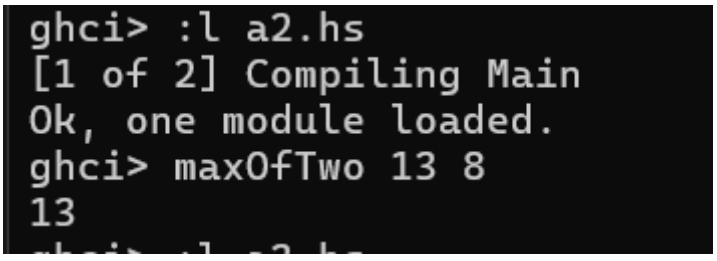
CODE:

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

EXPLANATION OF THE CODE:
The `maxOfTwo` function compares two numbers and returns the larger one.

Example:
Input: `maxOfTwo 13 8` → Output: `13`

SAMPLE INPUT & OUTPUT:



CONCLUSION:
Understood how to compare values using a function.

---

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.

OBJECTIVE:
To transform a list by doubling each element and adding 1, using function composition.

CODE:

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

EXPLANATION OF THE CODE:
This function doubles each number in the list and then adds 1 to each result using function composition.

Example:
Input: `doubleAndIncrement [3, 4, 5]` → Output: `[7, 9, 11]`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a3.hs
[1 of 2] Compiling Main             ( a3.hs, interpreted )
Ok, one module loaded.
ghci> doubleAndIncrement [3, 4, 5]
[7,9,11]
```

CONCLUSION:
Learned how to use composition to chain operations on a list.

---

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.

OBJECTIVE:
To compute the sum of squares of all elements in a list using composition.

CODE:

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

EXPLANATION OF THE CODE:
The `sumOfSquares` function squares each number in the list and adds them together.

Example:
Input: `sumOfSquares [2, 1, 5]` → Output: `30`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a4.hs
[1 of 2] Compiling Main             ( a4.hs, interpreted )
Ok, one module loaded.
ghci> sumOfSquares [2, 1, 5]
30
```

CONCLUSION:
Practiced applying a function to a list and aggregating the results with composition.

---

3. NUMBERS

---

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

OBJECTIVE:
To calculate the factorial of a number using recursion.

CODE:

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

EXPLANATION OF THE CODE:
The `factorial` function calculates the product of all numbers from 1 to the given number using recursion.

Example:
Input: `factorial 6` → Output: `720`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a5.hs
[1 of 2] Compiling Main             ( a5.hs, interpreted )
Ok, one module loaded.
ghci> factorial 6
720
```

CONCLUSION:
Gained understanding of recursive functions for repetitive calculations.

---

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

OBJECTIVE:
To compute the result of a number raised to a power using recursion.

CODE:

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

EXPLANATION OF THE CODE:
The `power` function computes the result of raising a base to a given exponent using recursion.

Example:
Input: `power 3 4` → Output: `81`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a6.hs
[1 of 2] Compiling Main             ( a6.hs, interpreted )
Ok, one module loaded.
ghci> power 3 4
81
```

---

4. LISTS

---

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

OBJECTIVE:
To filter out odd numbers from a list, keeping only the even ones.

CODE:

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

EXPLANATION OF THE CODE:
The `removeOdd` function removes all odd numbers from the given list, keeping only the even ones.

Example:
Input: `removeOdd [1, 2, 3, 4, 2]` → Output: `[2, 4, 2]`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a7.hs
[1 of 2] Compiling Main             ( a7.hs, interpreted )
Ok, one module loaded.
ghci>  removeOdd [1, 2, 3, 4, 2]
[2,4,2]
```

CONCLUSION:
Learned how to manipulate lists by removing specific elements.

---

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.

OBJECTIVE:
To extract the first `n` elements from a list.

CODE:

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

EXPLANATION OF THE CODE:
The `firstNElements` function extracts the first `n` elements from a list.

Example:
Input: `firstNElements 3 [100, 200, 300, 500]` → Output: `[100, 200, 300]`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a8.hs
[1 of 2] Compiling Main             ( a8.hs, interpreted )
Ok, one module loaded.
ghci> firstNElements 3 [100, 200, 300, 500]
[100,200,300]
```

CONCLUSION:
Practiced slicing lists to retrieve a specific portion.

---

5. TUPLES

---

1. Define a function `swap :: (a, b) -> (b, a)` that swaps the elements of a pair (tuple with two elements).

OBJECTIVE:
To exchange the elements of a tuple.

CODE:

```haskell
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

EXPLANATION OF THE CODE:
The `swap` function takes a pair and switches the two elements' positions.

Example:
Input: `swap (True, 54)` → Output: `(54, True)`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a9.hs
[1 of 2] Compiling Main             ( a9.hs, interpreted )
Ok, one module loaded.
ghci> swap (True, 54)
(54,True)
```

CONCLUSION:
Gained experience working with tuples and swapping their elements.

---

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.

OBJECTIVE:
To add the elements of each tuple in a list and return the sums.

CODE:

```haskell
addPairs :: [(Int, Int)] -> [Int]
addPairs = map (\(x, y) -> x + y)
```

EXPLANATION OF THE CODE:
The `addPairs` function takes a list of number pairs and adds each pair, returning a new list of sums.

Example:
Input: `addPairs [(5, 10), (20, 30), (15, 25)]` → Output: `[15, 50, 40]`

SAMPLE INPUT & OUTPUT:

```
ghci> :l a10.hs
[1 of 2] Compiling Main             ( a10.hs, interpreted )
Ok, one module loaded.
ghci>  addPairs [(5, 10), (20, 30), (15, 25)]
[15,50,40]
ghci>
```

CONCLUSION:
Learned how to process lists of tuples and apply arithmetic operations.