20CYS312 - Principles of Programming Languages

S.Shyam Balaji **CH.EN.U4CYS22045** **DATE:** 12/06/24

---

1. Functions and Types

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

Explanation:

The function `square` takes an integer, multiplies it by itself, and returns the result.

Input:

```
7
```

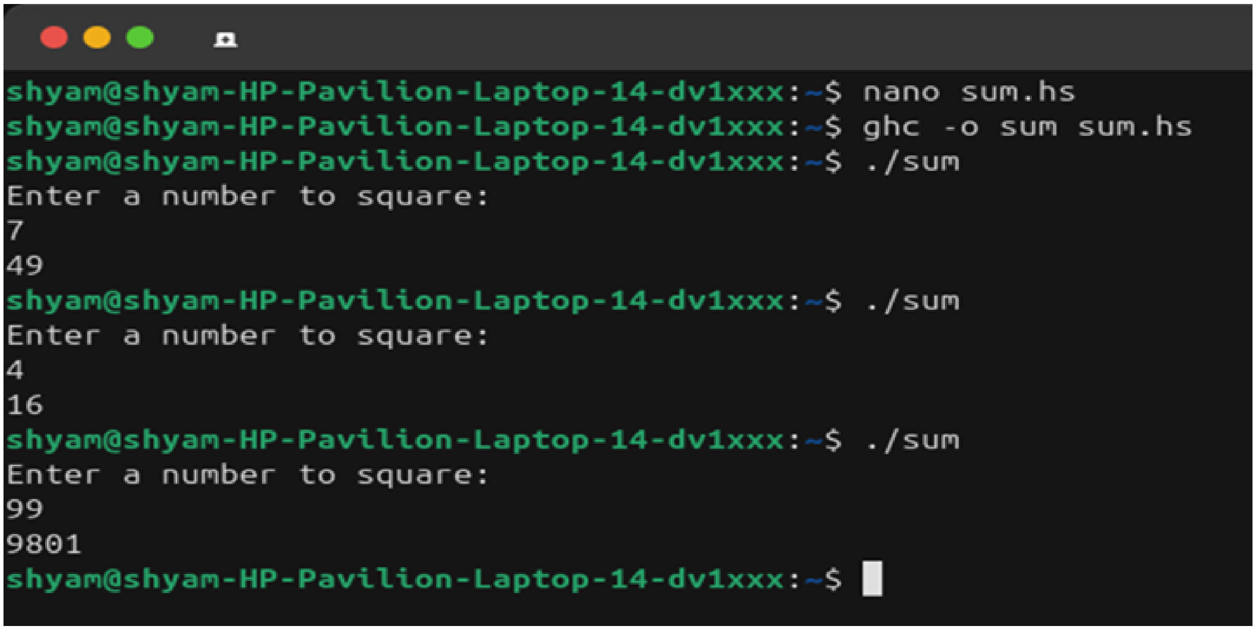
Output:

```
49
```

Code:

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

main :: IO ()
main = do
    putStrLn "Enter a number to square:"
    input <- getLine
```

```
    let number = read input :: Int
    print (square number)
```



Conclusion:

Learned to define and apply a simple function for mathematical operations.

---

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

Code:

```
maxoftwo :: Int -> Int -> Int
maxoftwo x y
  | x > y = x
  | otherwise = y

main :: IO ()
main = print(maxoftwo 10 5)
```

Explanation:

The function `maxoftwo` uses guards (`|`) to compare two integers and returns the larger one.

- **Input**: 10, 5

Output:

```
10
```

```
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ nano max.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ghc -o max max.hs
[1 of 2] Compiling Main             ( max.hs, max.o )
[2 of 2] Linking max
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./max
15
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$
```

Conclusion:

Learned to use guards to make decisions based on conditions.

2. Functional Composition

1. Objective: Define a function `doubleAndIncrement :: [Int] -> [Int]` that doubles each number in a list and increments it by 1 using function composition.

Code:

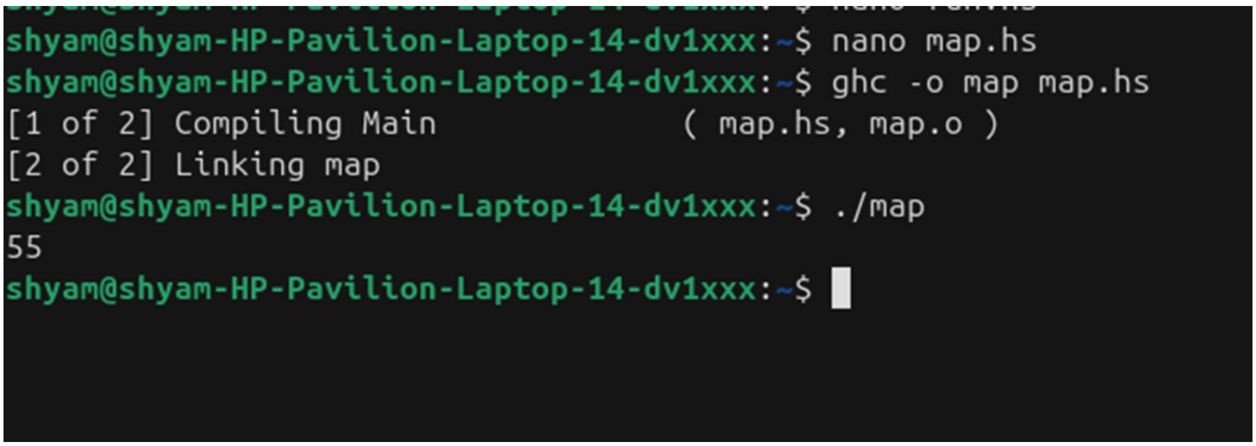
```
doubleAndIncrement :: [Int] -> [Int]
doubleAndIncrement = map ((+1) . (*2))

main :: IO ()
```

```
main = do
  print (doubleAndIncrement [1, 2, 3, 4])
```

Explanation:

Uses `map` and function composition (`(+1) . (*2)`) to process each list element.

- **Input**: `[1, 2, 3, 4]`
- **Output**: `[3, 5, 7, 9]`



Conclusion:

Learned functional composition for efficient list processing.

---

2. Objective: Write a function `sumOfSquares :: [Int] -> Int` that takes a list of integers, squares each element, and returns the sum of the squares using composition.

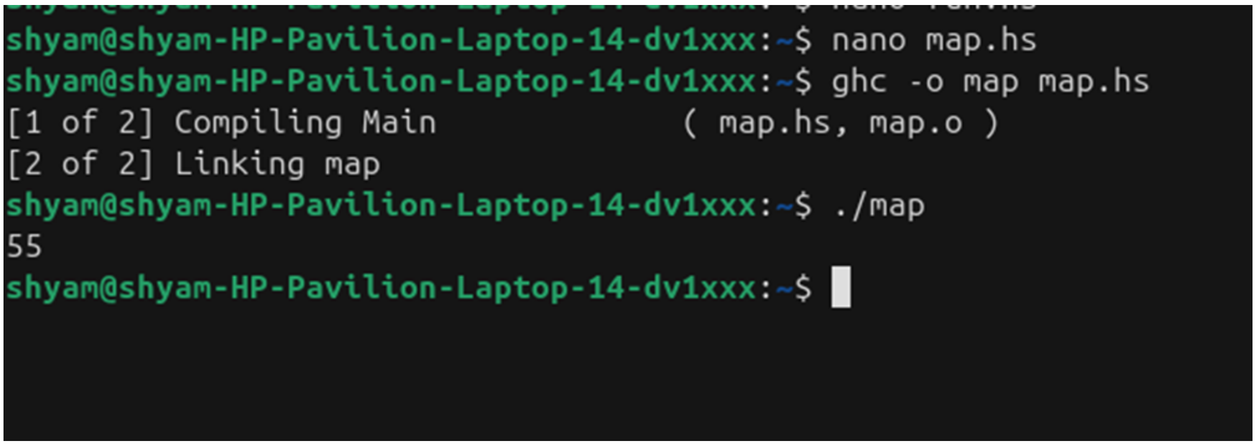
Code:

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

main :: IO ()
main = do
  print (sumOfSquares [1, 2, 3, 4])
```

Explanation:

Function squares each list element using `map (^2)` and sums them using `sum`.

- **Input**: `[1, 2, 3, 4]`
- **Output**: `55`



Conclusion:

Learned to combine higher-order functions for list operations.

---

3. Numbers

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

Code:

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

main :: IO ()
main = do
  print (factorial 5)
```

Explanation:

Uses recursion: For `n > 0`, computes `n * factorial (n-1)`. Base case: `factorial 0 = 1`.

- **Input**: `5`
- **Output**: `120`

```
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ nano rec1.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ghc -o rec1 rec1.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./rec1
120
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$
```

Conclusion:

Understood recursion for mathematical computations.

---

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

Code:

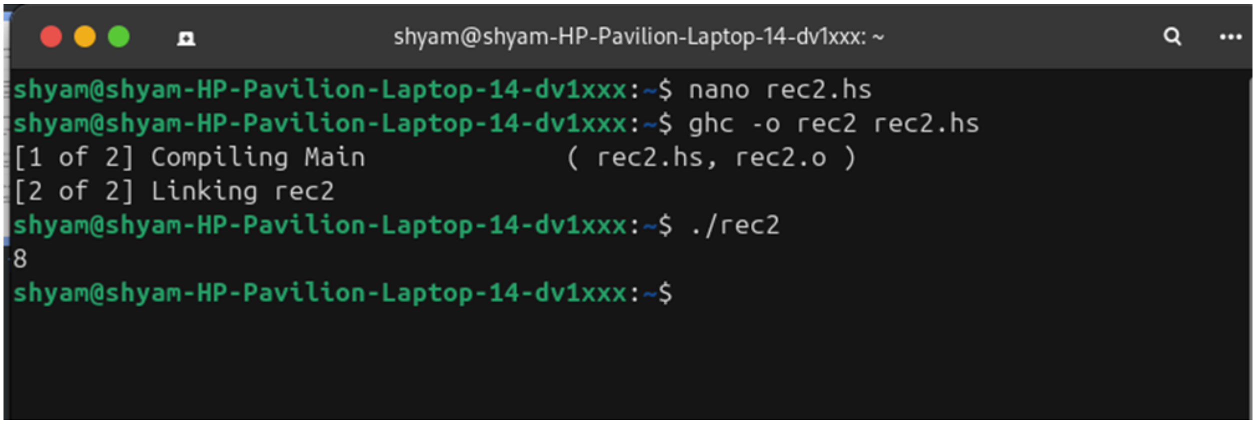
```
power :: Int -> Int -> Int
power _ 0 = 1
power base exp = base * power base (exp - 1)

main :: IO ()
main = print (power 2 3)
```

Explanation:

Uses recursion: For `exp > 0`, computes `base * power base (exp-1)`. Base case: `power _ 0 = 1`.

- **Input**: `2,3`

- **Output**: `8`



Conclusion:

Learned to implement recursive functions for exponentiation.

---

4. Lists

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

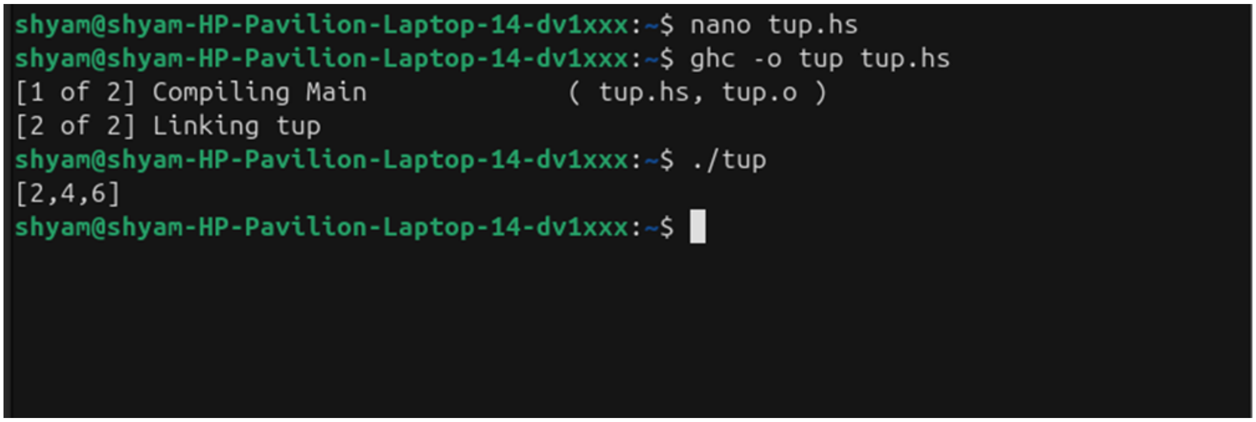
Code:

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

main :: IO ()
main = print (removeOdd [1, 2, 3, 4, 5, 6])
```

Explanation:

Uses recursion: Checks if the number is even. If true, includes it in the result; otherwise skips it.

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



Conclusion:

Learned to filter elements from a list based on conditions.

---

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

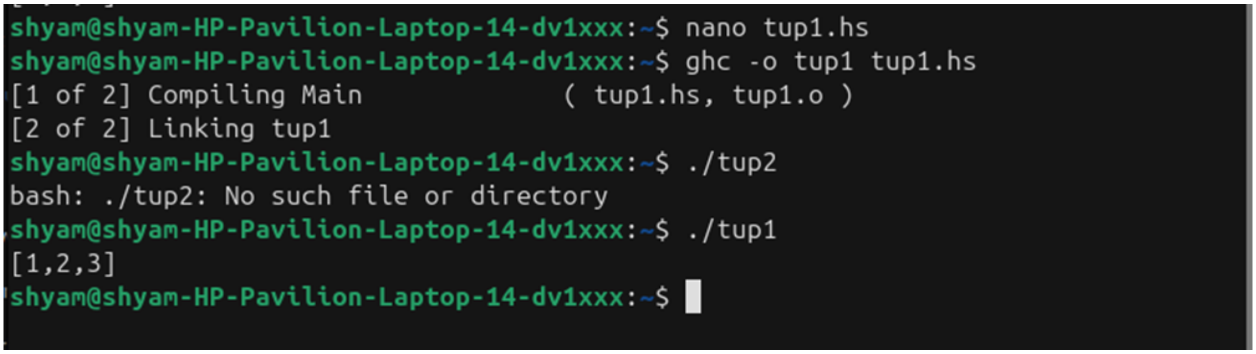
Code:

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

main :: IO ()
main = print (firstNElements 3 [1, 2, 3, 4, 5])
```

Explanation:

Uses recursion: Extracts the first `n` elements by repeatedly splitting the list.

- **Input**: `3, [1, 2, 3, 4, 5]`
- **Output**: `[1, 2, 3]`



```
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ nano tup1.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ghc -o tup1 tup1.hs
[1 of 2] Compiling Main             ( tup1.hs, tup1.o )
[2 of 2] Linking tup1
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./tup2
bash: ./tup2: No such file or directory
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./tup1
[1,2,3]
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$
```

Conclusion:

Learned to extract subsets from lists using recursion.

---

5. Tuples

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

Code:

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

main :: IO ()
main = print (swap (1, "shyam"))
```

Explanation:

Directly swaps elements of the tuple using pattern matching.

- **Input**: `(1, "shyam")`
- **Output**: `("shyam", 1)`

```
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ nano swap.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ghc -o swap swap.hs
[1 of 2] Compiling Main             ( swap.hs, swap.o ) [Source file changed]
[2 of 2] Linking swap [Objects changed]
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./swap
("shyam",1)
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$
```

Conclusion:

Learned tuple manipulation using pattern matching.

---

2. Objective: 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:

```
addPairs :: [(Int, Int)] -> [Int]
addPairs = map (uncurry (+))

main :: IO ()
main = print (addPairs [(1, 2), (3, 4), (5, 6)])
```

Explanation:

Recursively processes each tuple `(x, y)`, sums the pair, and appends the result to the list.

- **Input**: `[(1, 2), (3, 4), (5, 6)]`
- **Output**: `[3, 7, 11]`

```
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ nano swap1.hs
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ghc -o swap1 swap1.hs
[1 of 2] Compiling Main             ( swap1.hs, swap1.o )
[2 of 2] Linking swap1
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$ ./swap1
[3,7,11]
shyam@shyam-HP-Pavilion-Laptop-14-dv1xxx:~$
```

Conclusion:

Learned to process lists of tuples and generate new lists based on operations.