Department of Cyber Security

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Chennai Campus

Principals of Programming Languages

Subject Code: 20CYS312 Date:2024/12/06

Name: Sushant Yadav Roll Number:CH.EN.U4CYS22067

---

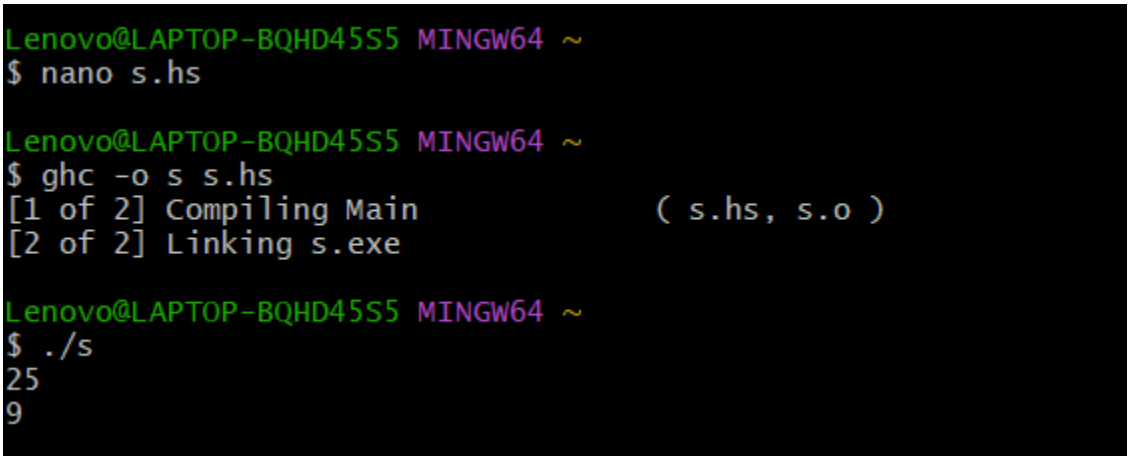
1. Functions and Types

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

Code:

```
square :: Int -> Int

square x = x * x

main :: IO ()

main = do

  print (square 5)  -- This will output 25

  print (square (-3))  -- This will output 9
```

Output:

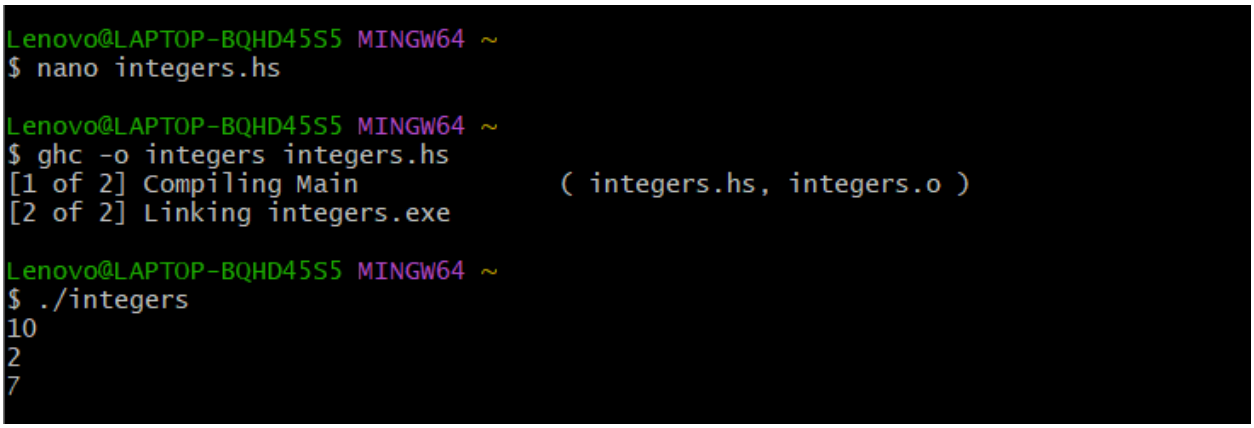


2. Define a function maxOfTwo :: Int -> Int -> Int that takes two integers and returns the larger one.

Code:

```
maxOfTwo :: Int -> Int -> Int

maxOfTwo x y = if x > y then x else y

main :: IO ()

main = do

  print (maxOfTwo 5 10)   -- This will output 10

  print (maxOfTwo (-3) 2)  -- This will output 2

  print (maxOfTwo 7 7)    -- This will output 7
```

Output:



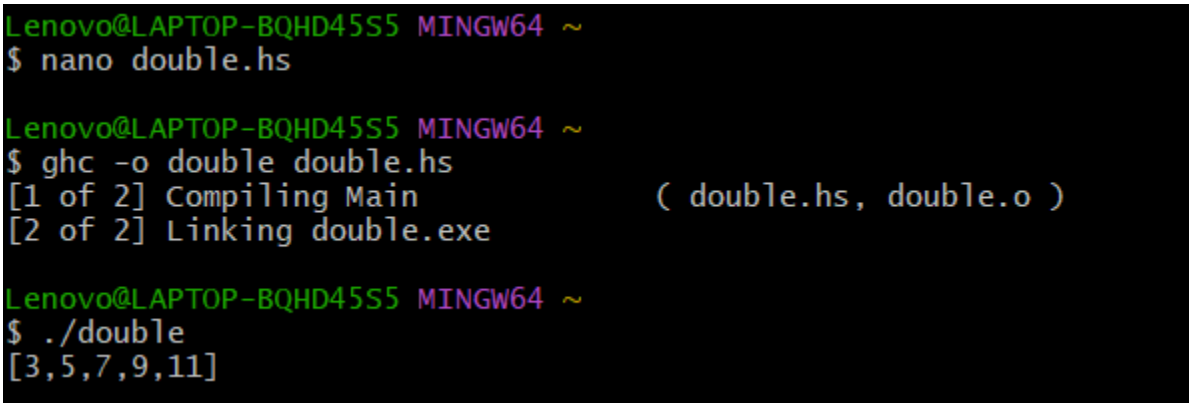
2. Functional Composition

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

Code:

```
doubleAndIncrement :: [Int] -> [Int]

doubleAndIncrement = map (+1) . map (*2)


main :: IO ()

main = do

  let numbers = [1, 2, 3, 4, 5]

  let doubledAndIncrementedNumbers = doubleAndIncrement numbers

  print doubledAndIncrementedNumbers
```

Output:

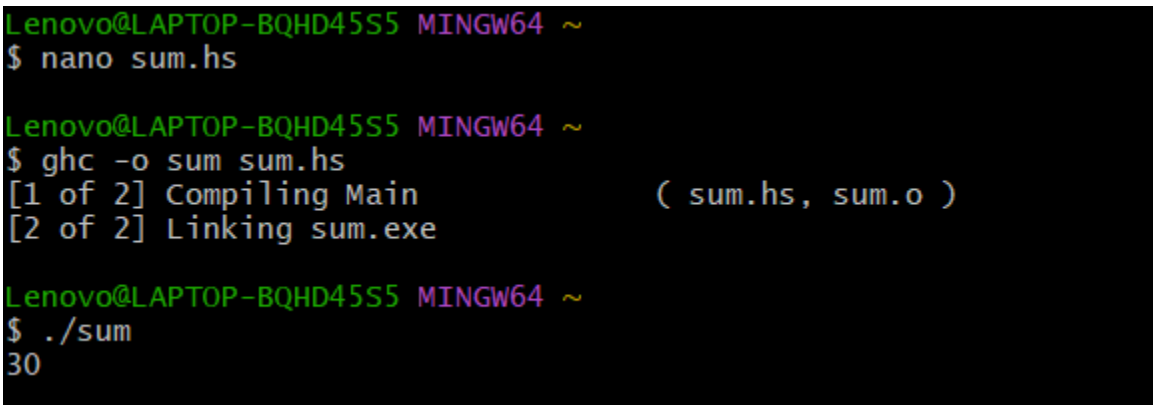


2. Write a function sumOfSquares :: [Int] -> Int that takes a list of integers, squares each element, and returns the sum of the squares using composition.

Code:

```
sumOfSquares :: [Int] -> Int

sumOfSquares = sum . map (^2)

main :: IO ()

main = do

  let numbers = [1, 2, 3, 4]

  print (sumOfSquares numbers)  -- Output will be 30 (1^2 + 2^2 + 3^2 + 4^2)
```

Output:



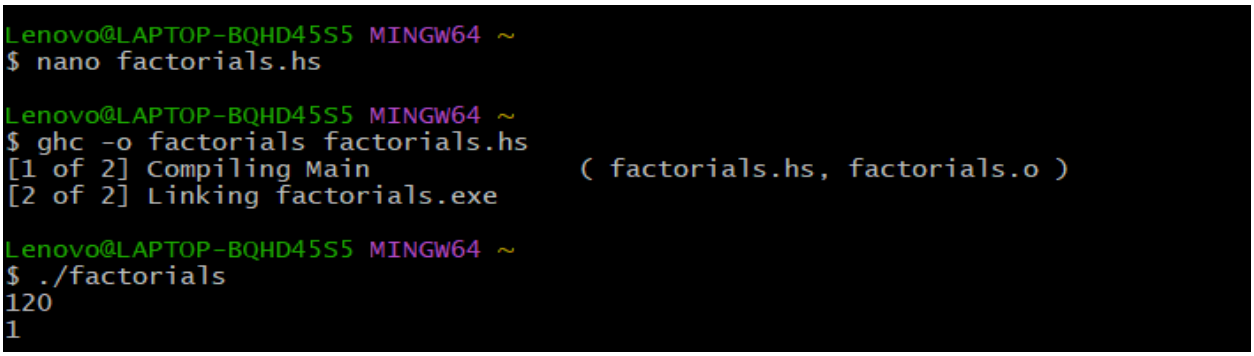
3. Numbers

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

Code:

```
module Main where

-- Function to calculate factorial

factorial :: Int -> Int

factorial 0 = 1 -- Base case: the factorial of 0 is 1

factorial n

| n < 0 = error "Factorial is not defined for negative numbers" -- Handle negative input

| otherwise = n * factorial (n - 1) -- Recursive case

-- Main function to test the factorial function

main :: IO ()

main= do

print (factorial 5) -- Output will be 120 (5! = 5 * 4 * 3 * 2 * 1)

print (factorial 0) -- Output will be 1 (0! = 1)

-- Uncommenting the next line will raise an error

-- print (factorial (-1)) -- Error: Factorial is not defined for negative numbers
```

Output:



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

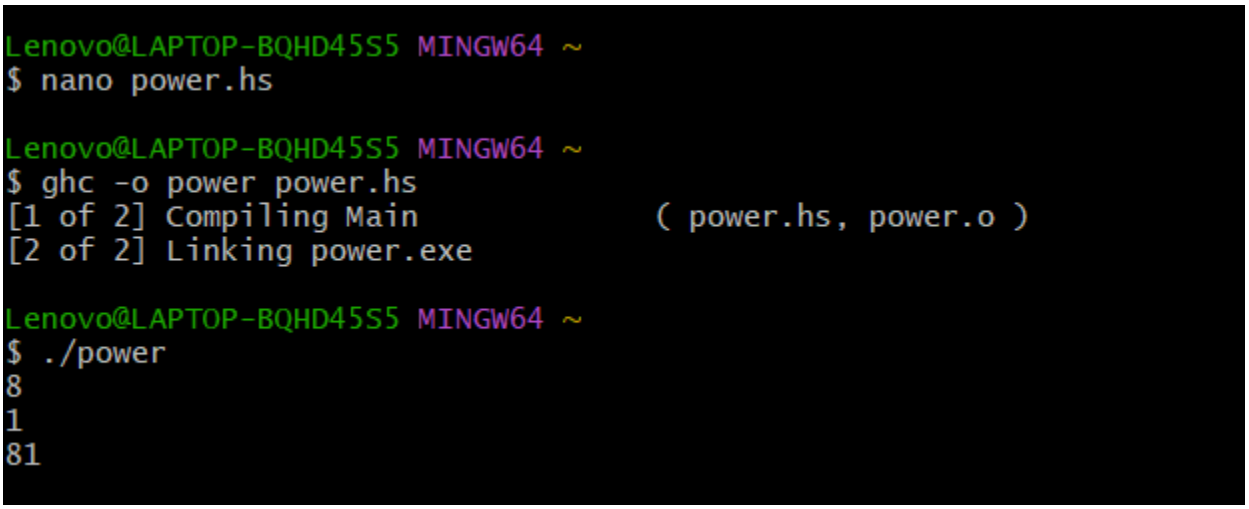
Code:

```
module Main where

-- Function to calculate power

power :: Int -> Int -> Int

power _ 0 = 1 -- Any number raised to the power of 0 is 1

power base exp

| exp < 0 = error "Exponent must be non-negative" -- Handle negative exponent
| otherwise = base * power base (exp - 1) -- Recursive case

-- Main function to test the power function

main :: IO ()

main = do

print (power 2 3) -- Output will be 8 (2^3 = 2 * 2 * 2)

print (power 5 0) -- Output will be 1 (5^0 = 1)

print (power 3 4) -- Output will be 81 (3^4 = 3 * 3 * 3 * 3)
```

-- Uncommenting the next line will raise an error

-- print (power 2 (-3)) -- Error: Exponent must be non-negative

Output:



4. Lists

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

Code:

module Main where

-- Function to remove odd numbers from a list

removeOdd :: [Int] -> [Int]

removeOdd [] = [] -- Base case: an empty list returns an empty list

removeOdd (x:xs)

```
  | odd x    = removeOdd xs  -- If the head is odd, skip it
  | otherwise = x : removeOdd xs  -- If the head is even, keep it


-- Main function to test the removeOdd function
main :: IO ()
main = do
  print (removeOdd [1, 2, 3, 4, 5, 6])  -- Output will be [2, 4, 6]
  print (removeOdd [7, 8, 9, 10])       -- Output will be [8, 10]
  print (removeOdd [1, 3, 5])           -- Output will be []
  print (removeOdd [2, 4, 6])           -- Output will be [2, 4, 6]
  print (removeOdd [])                  -- Output will be []
```

Output:

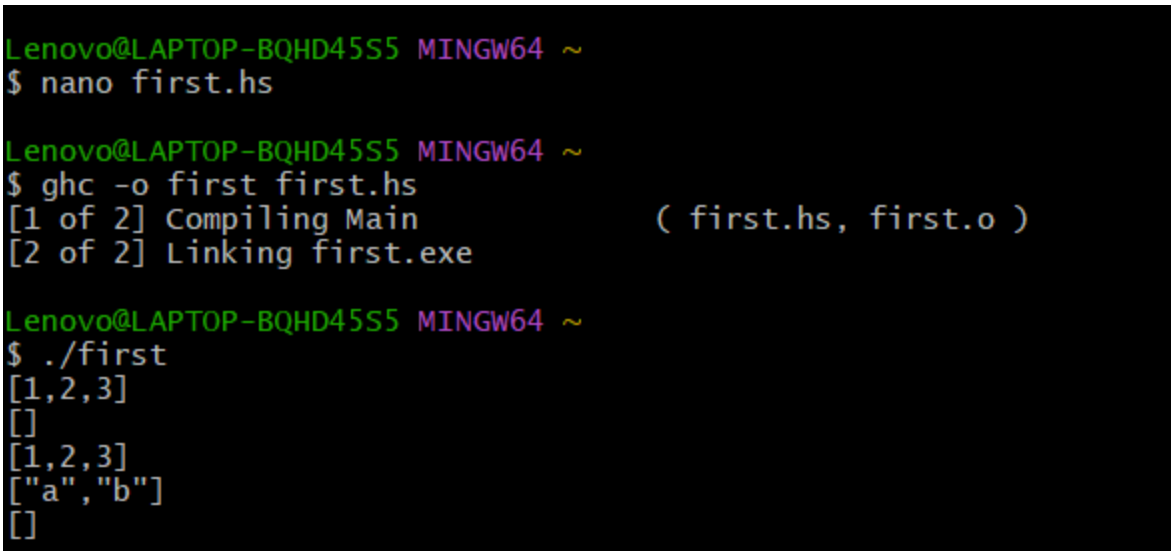
```
Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ nano odd.hs

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ghc -o odd odd.hs
[1 of 2] Compiling Main             ( odd.hs, odd.o )
[2 of 2] Linking odd.exe

Lenovo@LAPTOP-BQHD45S5 MINGW64 ~
$ ./odd
[2,4,6]
[8,10]
[]
[2,4,6]
[]
```

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.

Code:

```
module Main where


-- Function to get the first n elements of a list

firstNElements :: Int -> [a] -> [a]

firstNElements _ [] = []  -- If the list is empty, return an empty list

firstNElements n _ | n <= 0 = []  -- If n is less than or equal to 0, return an empty list

firstNElements n (x:xs) = x : firstNElements (n - 1) xs  -- Include the head and recurse



-- Main function to test the firstNElements function

main :: IO ()

main = do

  print (firstNElements 3 [1, 2, 3, 4, 5] :: [Int])  -- Output will be [1, 2, 3]

  print (firstNElements 0 [1, 2, 3, 4, 5] :: [Int])  -- Output will be []

  print (firstNElements 5 [1, 2, 3] :: [Int])        -- Output will be [1, 2, 3]

  print (firstNElements 2 ["a", "b", "c"] :: [String]) -- Output will be ["a", "b"]

  print (firstNElements 4 ([] :: [Int]))             -- Output will be []
```

Output:



5. Tuples

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

Code:

-- Define the swap function

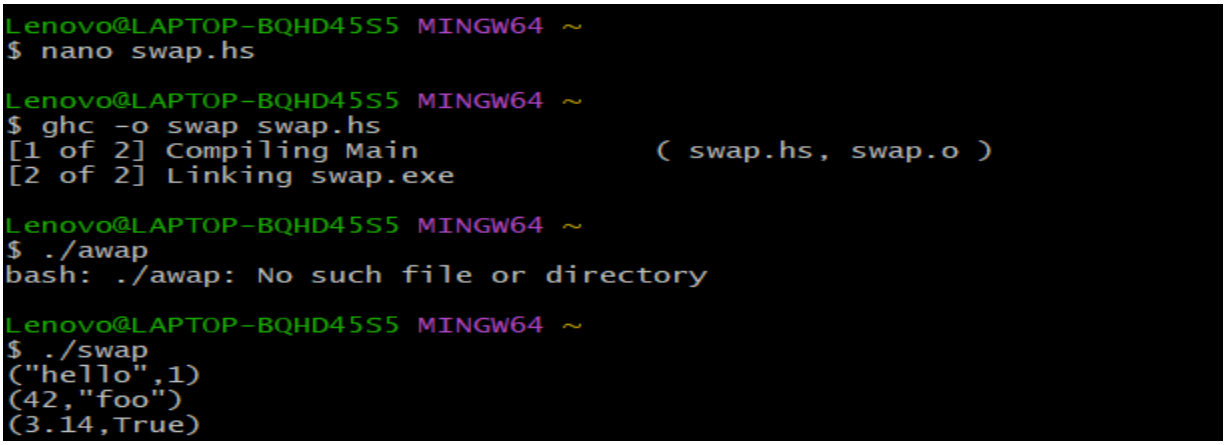
swap :: (a, b) -> (b, a)

swap (x, y) = (y, x)

-- Main function to test the swap function

```
main :: IO ()

main = do

print (swap (1, "hello")) -- Output will be ("hello", 1)

print (swap ("foo", 42)) -- Output will be (42, "foo")

print (swap (True, 3.14)) -- Output will be (3.14, True)
```

Output:



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:

```
-- Define the addPairs function

addPairs :: [(Int, Int)] -> [Int]

addPairs pairs = [x + y | (x, y) <- pairs]

-- Main function to test the addPairs function
```

```
main :: IO ()

main = do

let pairs = [(1, 2), (3, 4), (5, 6)]

print (addPairs pairs) -- Output will be [3, 7, 11]
```

Output: