

20CYS312 - Principles of Programming Languages

S. Shyam Balaji

CH.EN.U4CYS22045

DATE:

13/12/24

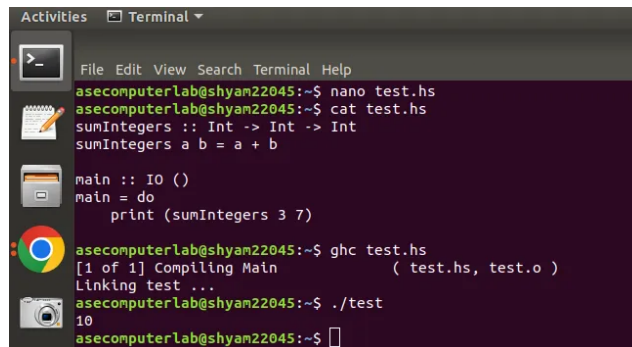
1. Basic Data Types

a. Sum of Two Integers

Objective:

To define a function that calculates the sum of two integers and returns the result.

Function Definition:



```
Activities Terminal
File Edit View Search Terminal Help
asecomputerlab@shyan22045:~$ nano test.hs
asecomputerlab@shyan22045:~$ cat test.hs
sumIntegers :: Int -> Int -> Int
sumIntegers a b = a + b

main :: IO ()
main = do
    print (sumIntegers 3 7)

asecomputerlab@shyan22045:~$ ghc test.hs
[1 of 1] Compiling Main          ( test.hs, test.o )
Linking test ...
asecomputerlab@shyan22045:~$ ./test
10
asecomputerlab@shyan22045:~$
```

Explanation:

The function `sumIntegers` takes two arguments of type `Int` and returns their sum.

Example:

```
sumIntegers 4 6
-- Output: 10
```

Conclusion:

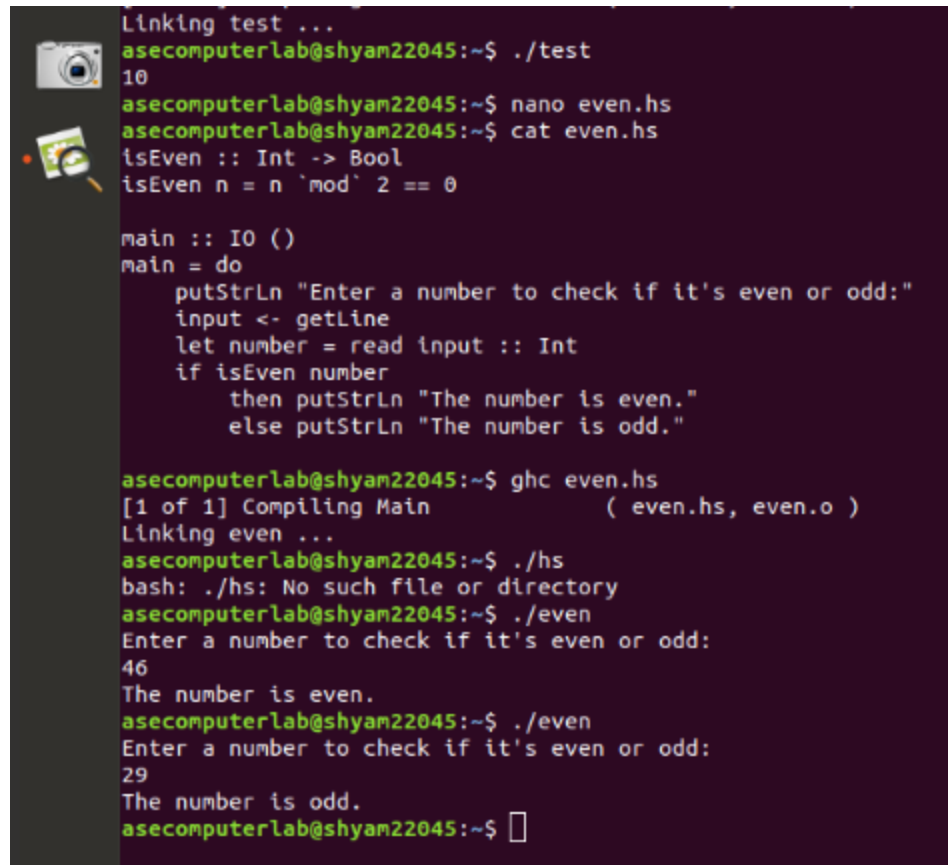
This function demonstrates a simple arithmetic operation in Haskell.

b. Check if a Number is Even or Odd

Objective:

To define a function that checks if a given number is even.

Function Definition:

A terminal window with a dark purple background and light green text. It shows the process of creating and testing a Haskell program. The user enters commands like './test', 'nano even.hs', 'cat even.hs', 'ghc even.hs', './hs', './even', and './even' to compile and run the program. The program prompts for a number and outputs whether it is even or odd based on the 'isEven' function.

```
Linking test ...
asecomputerlab@shyam22045:~$ ./test
10
asecomputerlab@shyam22045:~$ nano even.hs
asecomputerlab@shyam22045:~$ cat even.hs
isEven :: Int -> Bool
isEven n = n `mod` 2 == 0

main :: IO ()
main = do
    putStrLn "Enter a number to check if it's even or odd:"
    input <- getLine
    let number = read input :: Int
    if isEven number
    then putStrLn "The number is even."
    else putStrLn "The number is odd."

asecomputerlab@shyam22045:~$ ghc even.hs
[1 of 1] Compiling Main             ( even.hs, even.o )
Linking even ...
asecomputerlab@shyam22045:~$ ./hs
bash: ./hs: No such file or directory
asecomputerlab@shyam22045:~$ ./even
Enter a number to check if it's even or odd:
46
The number is even.
asecomputerlab@shyam22045:~$ ./even
Enter a number to check if it's even or odd:
29
The number is odd.
asecomputerlab@shyam22045:~$
```

Explanation:

The `isEven` function determines whether a number is divisible by 2 without a remainder.

Example:

```
isEven 46
-- Output: True
```

Conclusion:

The function showcases how to perform logical checks using Haskell.

c. Absolute Value

Objective:

To define a function that computes the absolute value of a given number.

Function Definition:

```
asecomputerlab@shyam22045:~$ nano float.hs
asecomputerlab@shyam22045:~$ cat float.hs
absolute :: Float -> Float
absolute x = if x < 0 then -x else x

main :: IO ()
main = do
    print (absolute (-3.5))
    print (absolute 4.2)

asecomputerlab@shyam22045:~$ ghc float.hs
[1 of 1] Compiling Main                ( float.hs, float.o )
Linking float ...
asecomputerlab@shyam22045:~$ ./float
3.5
4.2
asecomputerlab@shyam22045:~$
```

Explanation:

The `absolute` function checks if a number is negative and converts it to its positive equivalent.

Example:

```
absolute (-3.5)
-- Output: 3.5
```

Conclusion:

This function demonstrates conditional logic in Haskell.

2. List Operations

a. Sum of All Elements

Objective:

To compute the sum of all elements in a list.

Function Definition:

```
asecomputerlab@shyan22045:~$ nano sumlist.hs
asecomputerlab@shyan22045:~$ cat sumlist.hs
sumList :: [Int] -> Int
sumList xs = sum xs

main :: IO ()
main = do
    print (sumList [1, 2, 3, 4, 5])
    print (sumList [10, -3, 7])

asecomputerlab@shyan22045:~$ ghc sumlist.hs
[1 of 1] Compiling Main                ( sumlist.hs, sumlist.o )
Linking sumlist ...
asecomputerlab@shyan22045:~$ ./sumlist
15
14
asecomputerlab@shyan22045:~$
```

Explanation:

The `sumList` function utilizes Haskell's built-in `sum` function to add all integers in a list.

Example:

```
sumList [1, 2, 3, 4, 5]
sumList [10, -3, 7]
-- Output: 15
-- Output: 14
```

Conclusion:

This function efficiently calculates the sum of elements in a list.

b. Filter Even Numbers

Objective:

To filter out only even numbers from a given list.

Function Definition:

```

asecomputerlab@shyan22045:~$ nano gh.c
asecomputerlab@shyan22045:~$ nano gh.hs
asecomputerlab@shyan22045:~$ ghc gh.hs
[1 of 1] Compiling Main                ( gh.hs, gh.o )
Linking gh ...
asecomputerlab@shyan22045:~$ cat gh.hs
filterEven :: [Int] -> [Int]
filterEven xs = filter even xs

main :: IO ()
main = do
    print (filterEven [1, 2, 3, 4, 5, 6])
    print (filterEven [10, 15, 20, 25, 30])

asecomputerlab@shyan22045:~$ ./gh
[2,4,6]
[10,20,30]
asecomputerlab@shyan22045:~$ 

```

Explanation:

The function `filterEven` applies the `filter` function with the predicate `even` to keep only even numbers.

Example:

```

filterEven [1, 2, 3, 4, 5, 6]
-- Output: [2, 4, 6]

```

Conclusion:

This function demonstrates list filtering in Haskell.

c. Reverse a List

Objective:

To reverse the order of elements in a list.

Function Definition:

```

(shyam@LAPTOP-7K4E7JT9)~$ nano revers.hs

(shyam@LAPTOP-7K4E7JT9)~$ cat revers.hs
reverseList :: [a] -> [a]
reverseList xs = reverse xs

main :: IO ()
main = do
    let originalList = [1, 2, 3, 4, 5]
    let reversedList = reverseList originalList
    putStrLn "Original List:"
    print originalList
    putStrLn "Reversed List:"
    print reversedList

(shyam@LAPTOP-7K4E7JT9)~$ ghc revers.hs

(shyam@LAPTOP-7K4E7JT9)~$ ./revers
Original List:
[1,2,3,4,5]
Reversed List:
[5,4,3,2,1]

(shyam@LAPTOP-7K4E7JT9)~$

```

Explanation:

The function `reverseList` uses the built-in `reverse` function to reorder the elements of a list.

Example:

```

reverseList [1, 2, 3, 4, 5]
-- Output: [5, 4, 3, 2, 1]

```

Conclusion:

This function showcases how to manipulate the order of list elements.

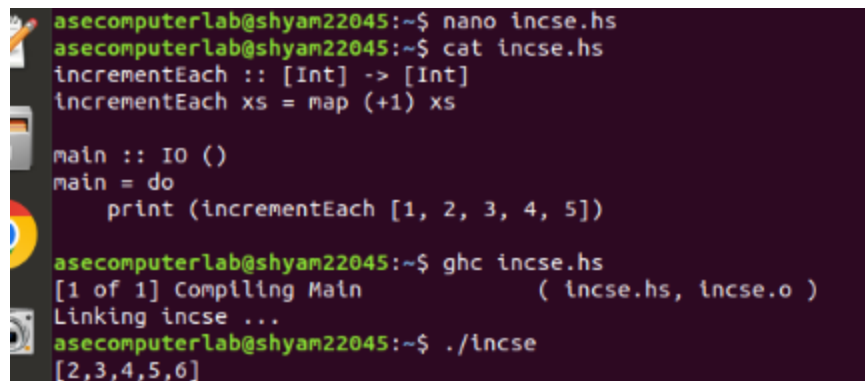
3. Basic Functions

a. Increment Each Element

Objective:

To increment each element in a list by 1.

Function Definition:



```
asecomputerlab@shyan22045:~$ nano incse.hs
asecomputerlab@shyan22045:~$ cat incse.hs
incrementEach :: [Int] -> [Int]
incrementEach xs = map (+1) xs

main :: IO ()
main = do
    print (incrementEach [1, 2, 3, 4, 5])

asecomputerlab@shyan22045:~$ ghc incse.hs
[1 of 1] Compiling Main                ( incse.hs, incse.o )
Linking incse ...
asecomputerlab@shyan22045:~$ ./incse
[2,3,4,5,6]
```

Explanation:

The `incrementEach` function uses `map` to apply the addition of 1 to each element of the list.

Example:

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

Conclusion:

This function illustrates how to use `map` for element-wise operations.

b. Square a Number

Objective:

To calculate the square of a given number.

Function Definition:

```
asecomputerlab@shyam22045:~$ nano play.hs
asecomputerlab@shyam22045:~$ nano play.hs
asecomputerlab@shyam22045:~$ cat play.hs
square :: Int -> Int
square x = x * x

main :: IO ()
main = do
    print (square 4)

asecomputerlab@shyam22045:~$ ghc play.hs
[1 of 1] Compiling Main                ( play.hs, play.o )
Linking play ...
asecomputerlab@shyam22045:~$ ./play
16
```

Explanation:

The `square` function multiplies a number by itself to find its square.

Example:

```
square 4
-- Output: 16
```

Conclusion:

This function highlights basic arithmetic operations in Haskell.

4. Function Composition

a. Compose Functions to Add and Multiply

Objective:

To compose two functions that add and multiply numbers.

Function Definition:


```

asecomputerlab@shyam22045:~$ nano mul.hs
asecomputerlab@shyam22045:~$ cat mul.hs
addThenMultiply :: Int -> Int -> Int -> Int
addThenMultiply x y z = (x + y) * z

main :: IO ()
main = do
    print (addThenMultiply 3 4 5) -- (3 + 4) * 5 = 35

asecomputerlab@shyam22045:~$ ghc mul.hs
[1 of 1] Compiling Main             ( mul.hs, mul.o )
Linking mul ...
asecomputerlab@shyam22045:~$ ./mul
35
asecomputerlab@shyam22045:~$ 

```

Explanation:

The function first adds `x` and `y` and then multiplies the result by `z`.

Example:

```

addThenMultiply 3 4 5
-- Output: 35

```

Conclusion:

This function demonstrates the power of combining operations in Haskell.

b. Apply Multiple Transformations

Objective:

To apply multiple transformations to elements in a list.

Function Definition:

```

asecomputerlab@shyam22045:~$ nano tran.hs
asecomputerlab@shyam22045:~$ cat tran.hs
transformList :: [Int] -> [Int]
transformList = map (+10) . map (^2)

main :: IO ()
main = do
    print (transformList [1, 2, 3, 4]) -- [11, 14, 19, 26]

asecomputerlab@shyam22045:~$ ghc tran.hs
[1 of 1] Compiling Main             ( tran.hs, tran.o )
Linking tran ...
asecomputerlab@shyam22045:~$ ./trans
bash: ./trans: No such file or directory
asecomputerlab@shyam22045:~$ ./tran
[11,14,19,26]
asecomputerlab@shyam22045:~$ 

```

Explanation:

The `transformList` function first squares each element and then adds 10 using a lambda function.

Example:

```

transformList [1, 2, 3] -- [11, 14, 19, 26]
-- Output: [11, 14, 19, 26]

```

Conclusion:

This function combines mathematical transformations using `map`.
