UNIT-III: Declarative Programming Paradigm: Functional Programming

Faculty In-charge

Mrinmoyee Mukherjee
Assistant Professor (IT Dept.)
email: mrinmoyeemukherjee@sfit.ac.in
Mob: 9324378409

Academic Year: 2023-24

Functions

- Functions play a major role in Haskell, as it is a functional programming language.
- Function declaration consists of the function name and its argument list along with its output. Function definition is where you actually define a function. **Examples**:

Ex1:

```
add :: Integer -> Integer -> Integer      --function declaration
add x y =  x + y                          --function definition
main = do
  putStrLn "The addition of the two numbers is:"
  print(add 2 5)                          --calling a function
```

Lambda abstraction: `\x -> \y -> x+y` or `\x y -> x+y`

Curried function: `(add 2) 5`

Ex2:

```
add :: (Num a) => a -> a -> a
add x y = x+y
main = do
   print(add 2 4)
```

Output = 6

or

```
add :: (Floating a) => a -> a -> a
add x y = x+y
```

Output = 6.0

Program 1: Hello world

helloworld.hs - Notepad

File Edit Format View Help

```
{-Name:Mrinmoyee
File:helloworld.hs
Description: Hello world program
-}

main::IO()
main=putStrLn "Hello World"
```

Pay Attention to multiline comment

Pay Attention to multiline comment

Main is the top level function. By lambda calculus the function is defined first

```
6
*Main> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello World
*Main>
```

Desktop OneDrive

Program 2: Addition Operation

addn.hs - Notepad

File Edit Format View Help

```
{-Name: Mrinmoyee
  Date: 16/9/2020
-}

main::IO()
main=do
     let var1=6
     let var2=7
     putStrLn "The addition is"
     print(var1+var2)
```

Use space and not tab

addwithinput.hs - Notepad

File Edit Format View Help

```
main:: IO()
main = do
 putStrLn "Insert the first value: "
 one <- getLine
 putStrLn "Insert the second value: "
 two <- getLine
 putStrLn "The result is:"
 print ((read one) + (read two))
```

getLine produces a string, you'll need to convert it to the correct type. We can use read to get a number from it

getLine returns a String in the IO monad and Strings cannot be added.
readLn has polymorphic return type, and the compiler infers that the return type is Integer (in the IO monad) so you can add them.

```
addwithinput.hs:2:9: error:
    Unexpected do block in f
        do putStrLn "Insert
    You could write it with
    Or perhaps you meant to
  |
2 |  main = do putStrLn "Ins

  |         ^^^^^^^^^^^^^^^^
Failed, no modules loaded.
Prelude> :l addwithinput
[1 of 1] Compiling Main
hs, interpreted )
Ok, one module loaded.
*Main> main
Insert the first value:
12
Insert the second value:
13
The result is:
25
*Main>
```

Functions

Ex3:

```
inc :: Integer ->  Integer        --function declaration
inc x =  x + 1                    --function definition
main = do
  putStrLn "The incremented value is:"
  print(inc 5)                     --calling a function
```

Lambda abstraction: \x-> x+1

Examples of Functions

- **Ex4**:

```
double :: Int -> Int
double n = 2*n
main = do
   putStrLn "The double of the number is:"
   print(double 5)    --calling a function
```

Lambda abstraction:

```
\n-> 2*n
```

- **Ex5**:

```
DoubleUs :: Int -> Int -> Int
DoubleUs x y = 2*x + 2*y
main = print(DoubleUs  4 5)
```

```
\x y-> 2*x + 2*y
```

- **Ex6**:

```
mysum1  :: Int -> Int
mysum1  n = sum [1..n]
main = print(mysum1   10)
```

```
\n -> sum [1..n]
```

```
doubleme1::Int->Int
doubleme1 x=x*2

main=print(doubleme1 10)
```

```
    * No instance for (Show (Int -> Int)) arising from a use of `print'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
*Main> doubleme1 3
6
*Main> :l doubleme1
[1 of 1] Compiling Main             ( doubleme1.hs, interpreted )
Ok, one module loaded.
*Main> main
20
```

```
doubleus::Int->Int->Int
doubleus x y=x*2+y*2

main=print(doubleus 10 20)
```

Command Prompt - ghci

```
'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
*Main> doubleme1 3
6
*Main> :l doubleme1
[1 of 1] Compiling Main             ( doubleme1.hs, interpreted )
Ok, one module loaded.
*Main> main
20
*Main> :l doubleus
[1 of 1] Compiling Main             ( doubleus.hs, interpreted )
Ok, one module loaded.
*Main> doubleus 10 20
60
*Main> :l doubleus
[1 of 1] Compiling Main             ( doubleus.hs, interpreted )
Ok, one module loaded.
*Main> main
60
```

Examples of Functions

- **Ex7**:

```
sayTwice :: String -> String
sayTwice s = s ++ s
-- or sayTwice s = s ++ " " ++ s
main = print (sayTwice "hello")
```

Lambda abstraction:

```
\s-> s ++ s
```

- **Ex8**:

```
nand :: Bool -> Bool -> Bool
nand False False = True
nand False True  = True
nand True  False = True
nand True  True  = False
main = do
   print(nand True True)          --calling a function
```

Pattern Matching in Recursive Function

- In recursion the function is applied inside its own definition.
- Use of Pattern Matching in récursive function to match one or more expressions
- **Ex2**:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
main = do
    putStrLn "The factorial of 5 is:"
    print (fact 4)
```

Output = 24

- **Ex3**:

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n =  x * (power x (n-1))
main = print (power 2 3)
```

Output = 8.0

Higher Order Function Ex6

```
divideByTen :: (Floating a) => a -> a
divideByTen x = (/10) x

main = do
    print(divideByTen 400)
```

Output:

40.0

Higher Order Function Ex7

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x

main = do
   print(compareWithHundred 2)
   print(compareWithHundred 102)
```

Output:

GT

LT

The function returns "LT" if the first argument is less than the second one, "EQ" if the arguments are equal, and "GT" if the first argument is greater than the second one.

Higher Order Function Ex8

```
apply :: (a -> a) -> a -> a
apply f x = f x
main = do
    print(apply (+5) 7)
```

Output = 12

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

main = do
    print(applyTwice (+5) 7)
```

Output: 17