**Academic Year 2024-25 Even**

**19CSE313 – Principles of Programming Language**

**B.Tech CSE 2022-26 F Section**

**Practice Set 10 – Closure, Currying, Higher-order Functions in Scala**

1. scala> (x: Int) => x + 1
2. scala> (10)
3. scala> var increase = (x: Int) => x + 1
4. scala> increase(10)
5. scala> increase = (x: Int) => x + 9999
6. scala> increase(10)
7. scala> var increase = (x:Int)=>{

| println("We")

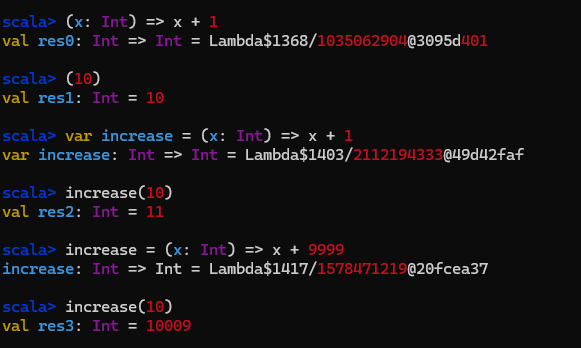
| println("are")

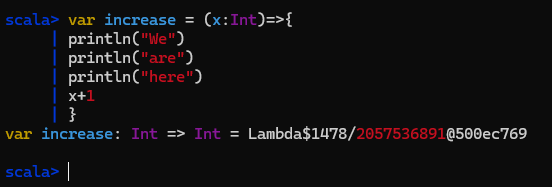
| println("here")

| x+1

| }

**Note: The copy-paste option will create issues**.





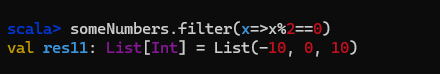
1. scala> increase(10)
2. scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
3. scala> someNumbers.foreach((x: Int) => println(x))
4. scala> someNumbers
5. scala>someNumbers.filter((x: Int) => x > 0)
6. scala>someNumbers.filter((x) => x > 0)
7. scala>someNumbers.filter(x => x > 0)
8. scala>someNumbers.filter(\_ > 0)



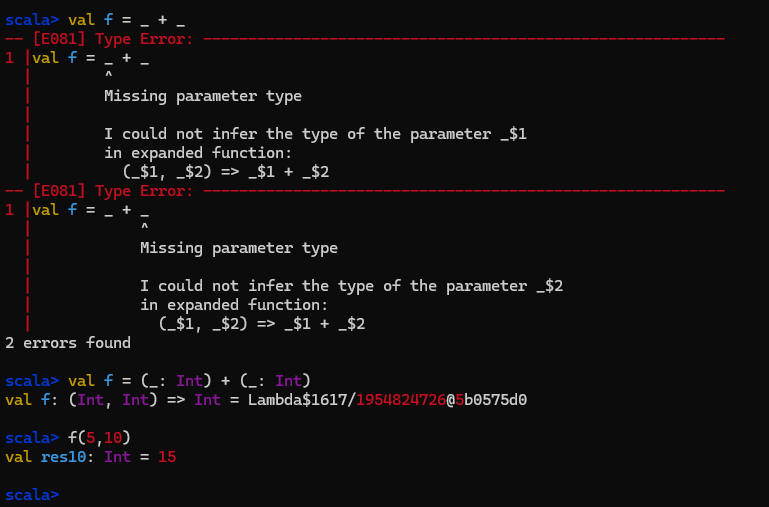
1. What did you observe with respect to the scripts in 12, 13, 14 and 15?

Different applications of filter

1. **Filter out even numbers from the someNumbers list.**

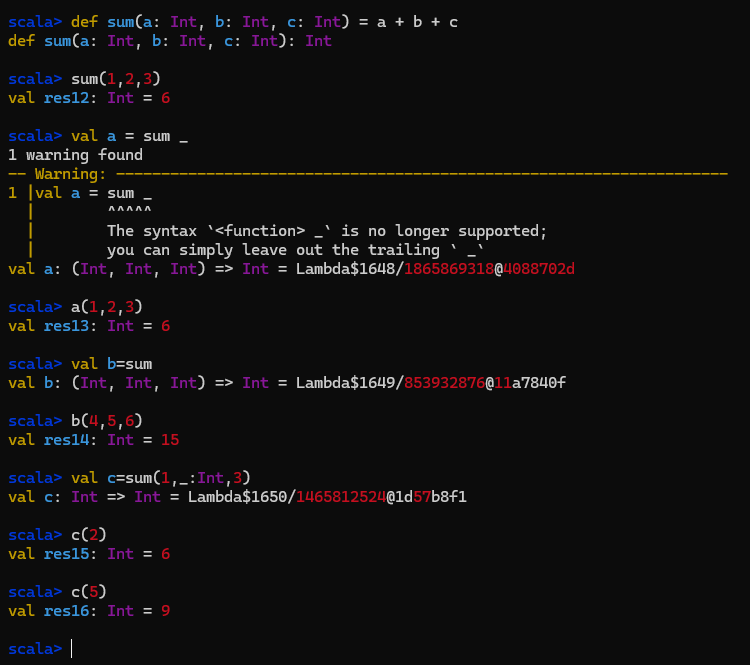
****

1. scala> val f = \_ + \_
2. scala> val f = (\_: Int) + (\_: Int)
3. scala>f(5, 10)



**Partially Applied Functions**

1. scala> def sum(a: Int, b: Int, c: Int) = a + b + c
2. scala> sum(1,2,3)
3. scala> val a = sum \_
4. scala> a(1,2,3)
5. scala> val b = sum
6. scala> b(4,5,6)
7. scala> val c = sum(1, \_: Int, 3)
8. scala> c(2)
9. scala> c(5)



1. Partial.scala //Source file

object Partial1{

def main(args:Array[String])={

val sum = (a: Int, b: Int, c: Int) => a + b + c

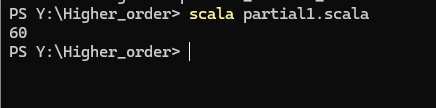
val f = sum(10,\_:Int,30)

println(f(20));

}

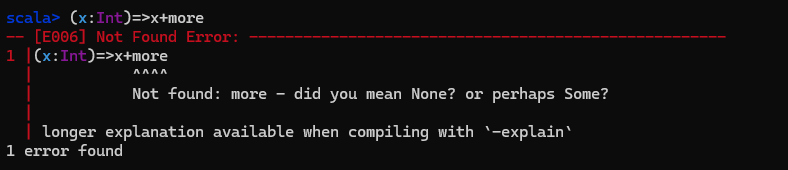
}

Execute: scala Partial1.scala

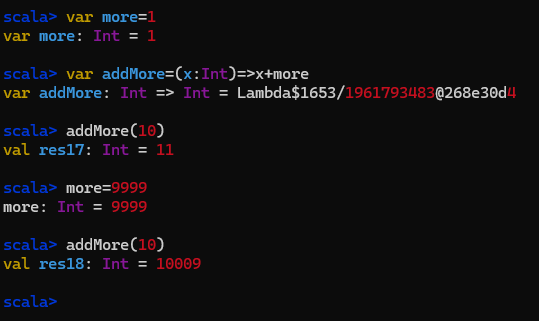


**Scala Closure**

1. scala> (x: Int) => x + more



1. scala> var more = 1
2. scala> val addMore = (x: Int) => x + more
3. scala> addMore(10)
4. scala> more = 9999
5. scala> addMore(10)



1. Briefly describe Scala Closures

Scala Closures are functions which uses one or more free variables and the return value of this function is dependent of these variable. The free variables are defined outside of the Closure Function and is not included as a parameter of this function. So the difference between a closure function and a normal function is the free variable. A free variable is any kind of variable which is not defined within the function and not passed as the parameter of the function. A free variable is not bound to a function with a valid value. The function does not contain any values for the free variable.

**Higher-order Functions**

1. First Example

Source Code – highre1.scala

object higher1

{

def math(x: Double, y:Double, fn: (Double,Double)=>Double): Double = fn(x,y);

 def main(args:Array[String])={

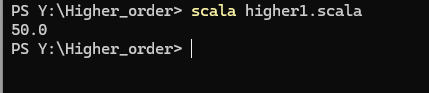
val result = math(50,20,(x,y)=>x max y);

println(result);

}

}

Execution: scala higher1.scala



1. Second Example

Source code: higher2.scala

object higher2

{

def math(x:Double,y:Double,z:Double,fn:(Double,Double)=>Double):Double = fn(fn(x,y),z);

def main(args:Array[String])={

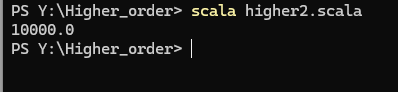
val result = math(50,20,10,(x,y)=>x \* y);

println(result);

}

}

Execution: scala higher2.scala



Q) given three strings write a higher order function to find the maximum of those trings

the higher order function should take 3 strings and a function which takes 2 strings

object StringMax {

// Higher-order function that takes 3 strings and a comparison function

def findMaxOfThree(str1: String, str2: String, str3: String, compare: (String, String) => String): String = {

// Apply the comparison function to find the maximum of the first two strings

val maxOfFirstTwo = compare(str1, str2)

// Compare the result with the third string to get the maximum of all three

compare(maxOfFirstTwo, str3)

}

def main(args: Array[String]): Unit = {

// Define a comparison function to compare two strings (lexicographically)

val compareStrings = (s1: String, s2: String) => if (s1 > s2) s1 else s2

// Call the findMaxOfThree function with three strings and the comparison function

val result = findMaxOfThree("apple", "banana", "cherry", compareStrings)

// Print the result

println(result) // Outputs: cherry

}

}

**Currying in Scala**

1. Example

Source code: curry.scala

object curry

{

def add(x:Int,y:Int) = x + y

def add2(x:Int) = (y:Int) => x + y; //curried version

def add3(x:Int)(y:Int) = x + y;//Simpler scala version

def main(args:Array[String])={

println(add(20,10));

println(add2(20)(10));//calling curried add2

val sum40 = add2(40);

println(sum40(100));//Partial application

println(add3(100)(200));//calling simpler scala version

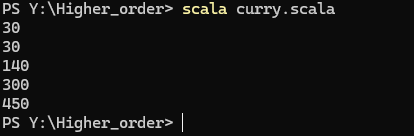
val sum50 = add3(50)\_;//partial application of simple version

println(sum50(400));

}

}

Execution: scala curry.scala



1. Briefly describe the concept of currying and different options of writing curried functions in Scala.

Currying in Scala is a technique where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. This allows for partial application, where some arguments can be fixed to create specialized functions.

Defining a curried function:

def add(x: Int)(y: Int): Int = x + y

**Map, Filter, FlatMap, ReduceLeft/Right, FOLD/SCAN**

1. Map

Source File: mymap.scala

object mymap

{

val lst = List(1,2,3,5,7,10,13);

def main(args:Array[String])={

println(lst.map(\_\*2));//method 1

println(lst.map(x=>x+3));//method 2 - using nameless function

println(lst.map(x => "hi"+ x));//string concatenation

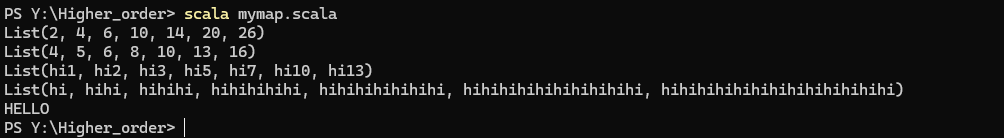
println(lst.map(x => "hi" \* x));//string multiplication

println("hello".map(\_.toUpper));//String is a list of characters

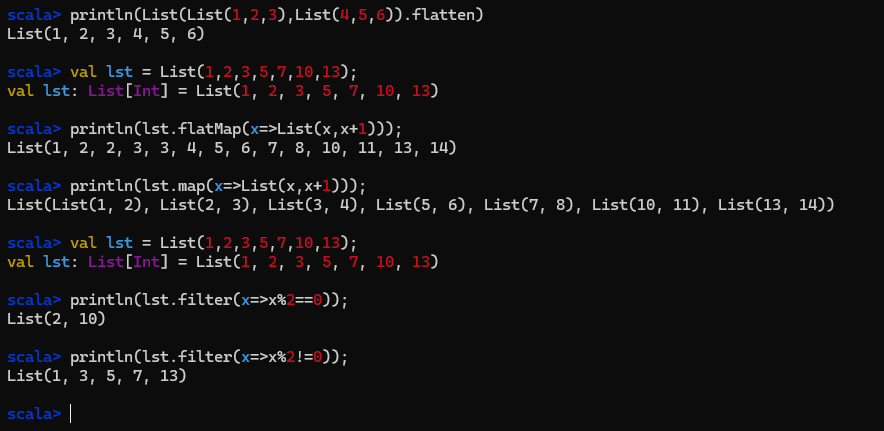
}

}

Execution: scala mymap.scala



1. scala> println(List(List(1,2,3),List(4,5,6)).flatten)
2. scala> val lst = List(1,2,3,5,7,10,13);
3. scala> println(lst.flatMap(x=>List(x,x+1)));
4. scala> println(lst.map(x=>List(x,x+1)));
5. What is the difference between map and flatMap?
6. scala> val lst = List(1,2,3,5,7,10,13);
7. scala> println(lst.filter(x=>x%2==0));
8. scala>println(lst.filter(x=>x%2!=0));



1. Create a Scala program with the following source code and write down your observations regarding the output based on the functions used

A screenshot of a computer code

AI-generated content may be incorrect.

object myreduce{

val lst=List(1,2,3,5,7,10,13);

val lst2=List("A","B","C");

def main(args:Array[String])={

println(lst.reduceLeft(\_+\_));

println(lst.reduceLeft((x,y)=>{println(x+","+y);x+y;}))

println(lst2.reduceLeft(\_+\_));

println(lst.reduceRight(\_+\_));

println(lst.reduceLeft(\_-\_));

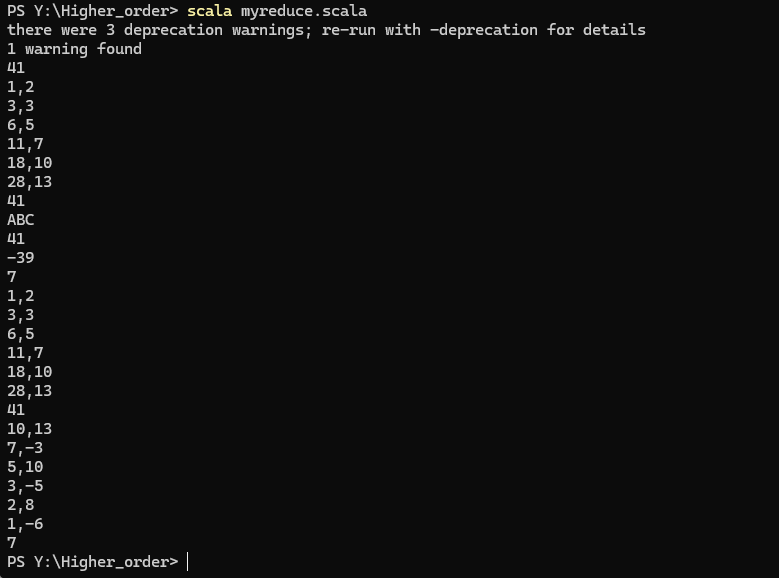
println(lst.reduceRight(\_-\_));

println(lst.reduceLeft((x,y)=>{println(x+","+y);x+y;}))

println(lst.reduceRight((x,y)=>{println(x+","+y);x-y;}))

}

}



1. Create a Scala program with the following source code and write down your observations regarding the output based on the functions used

A screenshot of a computer program

AI-generated content may be incorrect.

object myfoldscan

{

val lst=List(1,2,3,5,7,10,13);

val lst2=List("A","B","C");

def main(args:Array[String])={

println(lst.foldLeft(0)(\_+\_));

println(lst.foldLeft(100)(\_+\_));

println(lst.foldLeft("z")(\_+\_));

println(lst.foldLeft(100)(\_+\_));

println(lst.foldLeft("z")(\_+\_));

}

}

