Programmazione Funzionale e Parallela (A.A. 2015-2016)
Corso di Laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

Esempio di compito di esonero – Durata 1h 30'

Inserire nome, cognome e matricola nel file `studente.txt`. E' possibile usare Eclipse oppure un qualsiasi editor di testo usando `scalac`/`scala` da riga di comando.

Esercizio 1

Si scriva in un file `X1.scala` una funzione Scala **ricorsiva** `numEqual` che, date due funzioni f2 ed f2 e due interi a e b, calcola il numero di punti in [a,b] su cui le due funzioni coincidono. Assumendo che le funzioni f1 ed f2 richiedano tempo costante, la funzione `numEqual` deve richiedere tempo O(b-a) e spazio O(1).

La soluzione deve poter usare il seguente programma di prova `X1Main.scala`:

```
object X1Main extends App {
    val t1:Int = X1.numEqual(x=>2*x, x=>2*x)(1, 10)
    printf("test1: %d [corretto=10]\n", t1)

    val pf1:(Int,Int)=>Int = X1.numEqual(x=>x*x, x=>2*x)
    val t2:Int = pf1(0, 3)
    printf("test2: %d  [corretto=2]\n", t2)

    val t3:Int = pf1(3, 10)
    printf("test2: %d  [corretto=0]\n", t3)
}
```

La soluzione non deve usare alcun costrutto della programmazione imperativa e in particolare alcuna variabile `var`.

Esercizio 2

Si vuole estendere la classe `String` con un metodo `|||` che alterna i caratteri della prima stringa con quelli della seconda. Scrivere la soluzione in un file `X2.scala` in modo che sia possibile compilare ed eseguire correttamente il seguente programma di prova `X2Main.scala`:

```
import X2._

object X2Main extends App {
    val s1:String = "Leia" ||| "Luke"
    println(s1+""" [corretto="LLeuikae"]""")

    val s2:String = "Obi-Wan" ||| "Kenobi"
    println(s2+""" [corretto="OKbein-oWbain"]""")

    val s3:String = "Anakin" ||| "Skywalker"
    println(s3+""" [corretto="ASnkaykwianlker"]""")
}
```

La soluzione non deve usare alcun costrutto della programmazione imperativa e in particolare alcuna variabile `var`. Suggerimento: usare `map` e `reduce`.

Esercizio 3

Completare il seguente file `X3.scala`:

```
sealed abstract class Expr()
case class X() extends Expr()
case class Y() extends Expr()
case class And(e1:Expr, e2:Expr) extends Expr()
case class Or(e1:Expr, e2:Expr) extends Expr()
case class Not(e:Expr) extends Expr()
```

in modo da poter compilare ed eseguire correttamente il seguente programma di prova `X3Main.scala`:

```
object X3Main extends App {
    val xor:Expr = (X() || Y()) && !(X() && Y())

    val b0:Boolean = xor(false,false)
    println(b0+" [correct: false]")

    val b1:Boolean = xor(true,false)
    println(b1+"  [correct: true]")

    val b2:Boolean = xor(false,true)
    println(b2+"  [correct: true]")

    val b3:Boolean = xor(true,true)
    println(b3+" [correct: false]")
}
```

Nota: per definire in Scala il metodo unario `!` basta definire un metodo senza parametri dal nome `unary_!`.

La soluzione non deve usare alcun costrutto della programmazione imperativa e in particolare alcuna variabile `var`.