Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2020/2021

3 giugno 2021

1. (a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).

```
(0[0-7]+[lL]?)|([A-Za-z]+(?:\.[A-Za-z]+)*)|(\s+)|(\.mv|\.add|\.sub)
```

   i. `"a.b7"`
   ii. `"a."`
   iii. `"mv.sub"`
   iv. `".sub"`
   v. `"0L"`
   vi. `"00L"`

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp Exp | Atom
Atom ::=  a | b | ( Exp )
```

   (c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Exp` **resti invariato**.

2. Sia `fuse : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` la funzione così specificata:

   `fuse` f `[`x_1`; ... ;`x_k`] [`y_1`; ... ;`y_n`]` $=$ `[`$f\ x_1\ y_1$`; ... ;`$f\ x_m\ y_m$`]`, con $k, n \geq 0$ e $m = \min(n, k)$.

   Esempi:

```
fuse max [1;2;3] [3;1;4] = [3;2;4]
fuse max [1;2;3] [3] = [3]
fuse max [4] [3;5;7] = [4]
fuse (+) [1;2;3] [3;1;4] = [4;3;7]
fuse (+) [1;2;3] [3] = [4]
fuse (+) [4] [3;5;7] = [7]
```

   (a) Definire `fuse` senza uso di parametri di accumulazione.

   (b) Definire `fuse` usando un parametro di accumulazione affinché la ricorsione sia di coda.

3. Completare la seguente classe di iteratori `RangeIterator` per generare sequenze di interi da un estremo (`start`) incluso fino a un estremo (`end`) escluso con passo (`step`) diverso da zero.

   Esempio:

```
for (var i : new RangeIterator(3)) // start=0 end=3 step=1
   System.out.println(i); // prints 0 1 2
for (var i : new RangeIterator(3, -1)) // start=3 end=-1 step=1
   System.out.println(i); // no printed values
for (var i : new RangeIterator(3, -1, -3)) // start=3 end=-1 step=-3
   System.out.println(i); // prints 3 0
```

Codice da completare:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

class RangeIterator implements Iterator<Integer>, Iterable<Integer> {

   // dichiarare i campi mancanti
   // invariant step!=0

   // ranges from start (inclusive) to end (exclusive) with step!=0
   public RangeIterator(int start, int end, int step) {
      // completare
   }

   // ranges from start (inclusive) to end (exclusive) with step 1
   public RangeIterator(int start, int end) { this(start, end, 1); }

   // ranges from 0 (inclusive) to end (exclusive) with step 1
   public RangeIterator(int end) { this(0, end); }

   @Override
   public boolean hasNext() {
      // completare
   }

   @Override
   public Integer next() {
      // completare
   }

   @Override
   public Iterator<Integer> iterator() { return this; }
}
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Short s) { return "P.m(Short)"; }
   String m(Number n) { return "P.m(Number)"; }
}
public class H extends P {
   String m(short s) { return "H.m(short)"; }
   String m(float f) { return "H.m(float)"; }
}
public class Test {
    public static void main(String[] args) {
        P p = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato, assumendo che tutte le classi siano dichiarate nello stesso package.

Per ogni caso fornire due o tre righe di spiegazione così strutturate: se c'è un errore in fase di compilazione, specificare esattamente quale; se invece la compilazione va a buon fine spiegare brevemente perché e descrivere cosa avviene al momento dell'esecuzione, anche qui spiegando brevemente perché.

(a) `p.m(42)`

(b) `p2.m(42)`

(c) `h.m(42)`

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`