Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

23 gennaio 2020, a.a. 2018/2019

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali falliscono, motivando la risposta.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherTest {
    public static void main(String[] args) {
        Pattern regEx = Pattern.compile("(\\s+)|([*+/-])|(0[0-7]*)|([a-zA-Z][0-9]*)");
        Matcher m = regEx.matcher("077+x42");
        m.lookingAt();
        assert "077".equals(m.group(3));
        assert "".equals(m.group(4));
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert null != m.group(2);
        assert "+".equals(m.group(0));
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert "x".equals(m.group(4));
        assert "42".equals(m.group(3));
    }
}
```

   (b) Mostrare che nella seguente grammatica la stringa `a+b*a` è ambigua rispetto a `Exp`.

```
Exp ::= Exp * Atom | Atom + Exp | Atom
Atom ::=  ( Exp ) | Id
Id ::= a | b
```

   (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 `zip : 'a list -> 'b list -> ('a * 'b) list` la funzione così specificata (con $n, k \geq 0$):

   - `zip` $[x_1;\ldots;x_{n+k}]$ $[y_1;\ldots;y_n]$ restituisce la lista di coppie $[(x_1,y_1);\ldots;(x_n,y_n)]$;
   - `zip` $[x_1;\ldots;x_n]$ $[y_1;\ldots;y_{n+k}]$ restituisce la lista di coppie $[(x_1,y_1);\ldots;(x_n,y_n)]$.

   Esempi:

```
zip [1;2;3] ["one";"two";"three"]=[(1, "one"); (2, "two"); (3, "three")];;
zip [1;2] ["one";"two";"three"]=[(1, "one"); (2, "two")];;
zip [1;2;3] ["one";"two"]=[(1, "one"); (2, "two")];;
```

   (a) Completare la seguente definizione di `zip` senza uso di parametri di accumulazione.

```
let rec zip l1 l2 = match l1,l2 with (* completare *)
```

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

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
  String m(long i) {return "P.m(long)";}
  String m(long... i) {return "P.m(long...)";}
}
public class H extends P {
  String m(int i) {return super.m(i) + " H.m(int)";}
  String m(int... i) {return super.m(i[0],i[1]) + " H.m(int...)";}
}
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(42,42)` (e) `p2.m(42,42)` (f) `h.m(42,42)`

4. (a) Completare la classe `PairImp` che implementa coppie di valori diversi da **`null`**.

```
public interface Pair<T1, T2> {
  T1 getFirst();
  T2 getSecond();
}
public class PairImp<T1, T2> implements Pair<T1, T2> {
  public final T1 first; /* invariante: first != null */
  public final T2 second; /* invariante: second != null */

  public PairImp(T1 first, T2 second) { /* completare */ }
  public T1 getFirst() { /* completare */ }
  public T2 getSecond() { /* completare */ }
  public String toString() { /* completare */ } // restituisce "("first","second")"
}
```

(b) Completare la classe `Zipper` che definisce iteratori zipper a partire da due iteratori `iterator1` e `iterator2` inizialmente ottenuti da due oggetti `iterable1` e `iterable2` di tipo `Iterable`. A ogni iterazione il metodo `next()` dello zipper restituisce la coppia di elementi rispettivamente ottenuti dai due iteratori `iterator1` e `iterator2` tramite il metodo `next()`; l'iterazione termina quando uno dei due iteratori `iterator1` e `iterator2` non ha più elementi da restituire.

```
public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {
  private final Iterator<T1> iterator1;
  private final Iterator<T2> iterator2;

  public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) { /* completare */ }
  public boolean hasNext() { /* completare */ }
  public Pair<T1, T2> next() { /* completare */ }
}
```

Il seguente codice mostra un semplice esempio di uso della classe `Zipper`. Il metodo statico `java.util.Arrays.asList` crea una nuova lista a partire dagli argomenti passati; per esempio, `asList(1, 2, 3)` restituisce la lista `[1,2,3]`.

```
import static java.util.Arrays.asList;

public class Test {
  public static void main(String[] args) {
    Zipper<Integer, String> zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two", "three"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two) (3,three)
    zipper = new Zipper<>(asList(1, 2), asList("one", "two", "three"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two)
    zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two)
  }
}
```