Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2017/2018

4 giugno 2018

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("('[^']*')|(0[0-7]*)|(\\s+)");
        Matcher m = regEx.matcher("'00' 00");
        m.lookingAt();
        assert m.group(1).equals("00");
        assert m.group(2) != null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(3).equals("'");
        assert m.group(3) == null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(0).equals("0");
        assert m.group(2) == null;
    }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp . Exp | Exp , Exp | ( Exp ) | Bit
Bit   ::= 0 | 1
```

   (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 `first : ('a * 'b) list -> 'a list` la funzione così specificata: `first` $[(a_1, b_1); \ldots; (a_n, b_n)]$ restituisce la lista $[a_1; \ldots; a_n]$. Esempio:

```
# first [(1,"one");(2,"two");(3,"three")];;
- : int list = [1; 2; 3]
```

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

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

   (c) Definire `first` come specializzazione della funzione `map:('a -> 'b) -> 'a list -> 'b list`.

3. (a) Completare la seguente classe `LimitIterator` che permette di creare iteratori con un limite massimo di elementi.

```
import java.util.Iterator;

public class LimitIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final int limit; // limite massimo elementi
    private int items = 0; // numero elementi restituiti

    public LimitIterator(Iterator<E> baseIterator, int limit) { /* completare */ }
    public boolean hasNext() { /* completare */ }
    public E next() { /* completare */ }
}
```

   Per esempio, il codice sottostante crea un iteratore `lim_it` a partire dall'iteratore di numeri interi `it`, in modo che `lim_it` restituisca solo i primi 5 elementi che restituirebbe `it`.

```
Iterator<Integer> it = asList(1, 2, 3, 4, 5, 6, 7, 8, 9).iterator();
Iterator<Integer> lim_it = new LimitIterator<>(it, 5);
int i = 0;
while (lim_it.hasNext())
    assert ++i == lim_it.next();
assert i == 5;
```

(b) Utilizzando la classe `LimitIterator`, implementare il seguente metodo `found`.

```
/*
 * restituisce true se e solo se it contiene tra i suoi primi limit elementi un
 * oggetto uguale a elem
 */
public static <E> boolean found(E elem, Iterator<E> it, int limit) {
    /* completare usando LimitIterator */
}
```

Per esempio, se un iteratore della classe `GenBinLit` genera la sequenza infinita di stringhe binarie `"0"`, `"1"`, `"10"`, `"11"`, `"100"`, ..., allora il metodo `found` si comporta nel seguente modo:

```
assert !found("1111", new GenBinLit(), 7);
assert found("1111", new GenBinLit(), 15);
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Double d) {
        return "P.m(Double)";
    }
    String m(Float f) {
        return "P.m(Float)";
    }
}
public class H extends P {
    String m(Double d) {
        return super.m(d) + " H.m(Double)";
    }
    String m(float f) {
        return super.m(f) + " 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(42.0)`

(b) `p2.m(42.0)`

(c) `h.m(42.0)`

(d) `p.m(42f)`

(e) `p2.m(42f)`

(f) `h.m(42f)`