Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 4 giugno 2018

a.a. 2017/2018

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

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class MatcherTest {
5      public static void main(String[] args) {
6          Pattern regEx = Pattern.compile("('[^']*')|(0[0-7]*)|(\\s+)");
7          Matcher m = regEx.matcher("'00' 00");
8          m.lookingAt();
9          assert m.group(1).equals("00");
10         assert m.group(2) != null;
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(3).equals("'");
14         assert m.group(3) == null;
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group(0).equals("0");
18         assert m.group(2) == null;
19     }
20 }
```

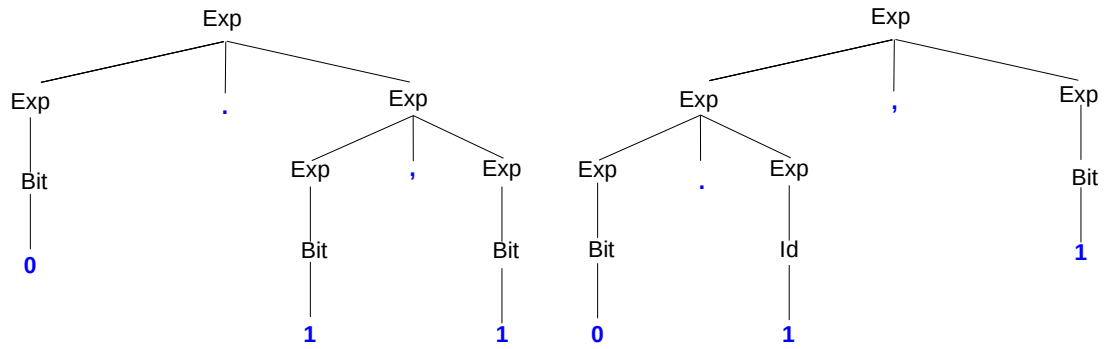
Soluzione:

- `assert m.group(1).equals("00");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `'00' 00` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `'00'` (stringa di zero o più caratteri diversi da `'`, delimitata da `'`, appartenente ai soli gruppi 0 e 1), quindi l'asserzione fallisce;
- `assert m.group(2) != null;` (linea 10): lo stato del matcher non è cambiato rispetto alla linea precedente, quindi per i motivi del punto precedente `m.group(2)` restituisce `null` e l'asserzione fallisce;
- `assert m.group(3).equals("'");` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `'00'` (ossia uno spazio bianco) e l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3, quindi `m.group(3)` restituisce una stringa di spazi bianchi che è diversa dalla stringa `'` e l'asserzione fallisce;
- `assert m.group(3) == null;` (linea 14): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(3)` restituisce una stringa e l'asserzione fallisce;
- `assert m.group(0).equals("0");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `0`) e l'invocazione del metodo `lookingAt()` ha successo poiché la stringa `00` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 2 (stringa che inizia con `0` seguita da zero o più cifre ottali); per tale motivo, `m.group(0)` restituisce tale stringa e l'asserzione fallisce;
- `assert m.group(2) == null;` (linea 18): lo stato del matcher non è cambiato rispetto alla linea sopra, per i motivi del punto precedente `m.group(2)` restituisce una stringa, quindi l'asserzione fallisce.

(b) Mostrare che la seguente grammatica è ambigua.

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

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `0 . 1 , 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**.

Soluzione: Una possibile soluzione consiste nell'aggiunta dei non-terminali `Op` e `Atom` per poter attribuire associatività a sinistra e stessa precedenza a entrambi gli operatori `.` e `,`.

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

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`.

Soluzione: Vedere il file `soluzione.ml`.

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);
```

Soluzione: Vedere il file `soluzione.jar`.

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)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché **`double`** $\not\leq$ `Float, Double`, mentre l'unico metodo accessibile e applicabile per boxing conversion è quello con segnatura `m(Double)` poiche `Double` $\not\leq$ `Float`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Double)` in `P`.
Viene stampata la stringa `"P.m(Double)"`.

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Double)` ridefinito in `H`; l'invocazione **`super`**`.m(d)` viene risolta come al punto precedente poiché `d` ha tipo statico `Double` e l'unico metodo accessibile e applicabile per sottotipo è quello con segnatura `m(Double)`; viene stampata la stringa `"P.m(Double) H.m(Double)"`.

(c) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`, quindi nessun metodo di `H` accessibile è applicabile per sottotipo poiché **`double`** $\not\leq$ **`float`**`, Float, Double`, mentre l'unico metodo applicabile per boxing conversion è quello con segnatura `m(Double)` poiché `Double` $\not\leq$ `Float`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Double)` di `H` come nel punto precedente; viene quindi stampata la stringa `"P.m(Double) H.m(Double)"`.

(d) Il literal `42f` ha tipo statico **`float`** e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché **`float`** $\not\leq$ `Float, Double`, mentre l'unico metodo accessibile e applicabile per boxing conversion è quello con segnatura `m(Float)` poiche `Float` $\not\leq$ `Double`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Float)` in `P`.

Viene stampata la stringa `"P.m(Float)"`.

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(Float)` ereditato da `H` e viene stampata la stringa `"P.m(Float)"`.

(f) Il literal `42f` ha tipo statico **`float`** e il tipo statico di `h` è `H`, quindi solo il metodo accessibile di `H` con segnatura `m(`**`float`**`)` è applicabile per sottotipo poiché **`float`** $\not\leq$ `Float, Double`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(`**`float`**`)` di `H`; l'invocazione **`super`**`.m(f)` viene risolta come al punto precedente poiché `f` ha tipo statico **`float`**; viene stampata la stringa `"P.m(Float) H.m(float)"`.