Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 7 luglio 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("(zero|one)|(\\s+)|([A-Z]([a-zA-Z]*))");
7          Matcher m = regEx.matcher("zero One");
8          m.lookingAt();
9          assert !(m.group(1) == null);
10         assert !(m.group(0) == null);
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(1) == null;
14         assert m.group(0) != null;
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group(3).equals("One");
18         assert m.group(4).equals("ne");
19     }
20 }
```

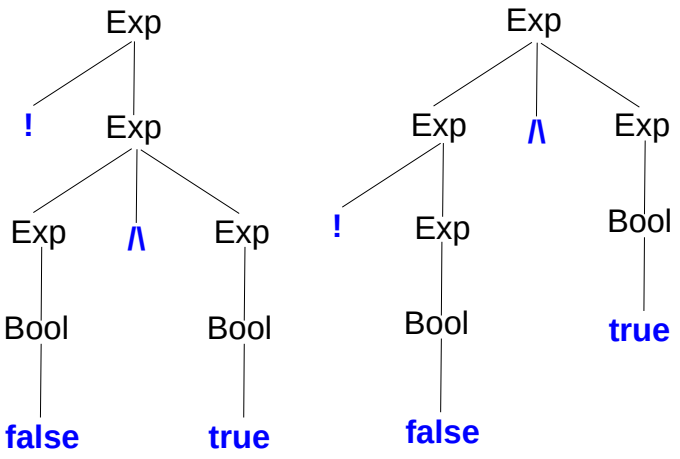
Soluzione:

- `assert !(m.group(1) == null);` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `zero One` 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 `zero` (appartenente ai soli gruppi 0 e 1), quindi il metodo restituisce un oggetto non `null` e l'asserzione ha successo;
- `assert !(m.group(0) == null);` (linea 10): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(0)` restituisce un oggetto non `null` e l'asserzione ha successo;
- `assert m.group(1) == null;` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `zero` (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 2, quindi `m.group(1)` restituisce `null` e l'asserzione ha successo;
- `assert m.group(0) != null;` (linea 14): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(0)` restituisce una stringa e l'asserzione ha successo;
- `assert m.group(3).equals("One");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `O`) e l'invocazione del metodo `lookingAt()` ha successo poiché la stringa `One` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3 (qualsiasi stringa non vuota che inizia con una lettera maiuscola seguita da zero o più lettere maiuscole o minuscole); per tale motivo, `m.group(0)` restituisce tale stringa e l'asserzione ha successo;
- `assert m.group(4).equals("ne");` (linea 18): lo stato del matcher non è cambiato rispetto alla linea sopra, il gruppo 4 individua la sotto-espressione regolare `[a-zA-Z]*` (sequenza di zero o più lettere maiuscole o minuscole) che contribuisce al match della sotto-stringa `ne`, quindi l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ! Exp | Exp ∧ Exp | ( Exp ) | Bool
Bool ::= false | true
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `!false ∧ true`



(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 del non-terminale `Bang` per poter attribuire la precedenza all'operatore unario `!` e forzare l'associatività (a sinistra nella grammatica) dell'operatore binario `∧`.

```
Exp ::= Exp ∧ Bang | Bang
Bang ::= ! Bang | ( Exp ) | Bool
Bool ::= false | true
```

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

   Esempio:

```
# merge (fun (x,y) -> x+y) [(1,2);(3,4);(5,6)];;
- : int list = [3; 7; 11]
# merge (fun (x,y) -> x+String.length y)  [(1,"one");(2,"two");(3,"three")];;
- : int list = [4; 5; 8]
```

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

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

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

   Soluzione: Vedere il file `soluzione.ml`.

3. Completare la seguente classe `OddOnlyIterator` di iteratori definiti a partire da un iteratore di base `baseIterator` che restituiscono solo gli elementi di posizione dispari di `baseIterator` (considerando dispari la posizione del primo elemento).

```
public class OddOnlyIterator<E> implements Iterator<E> {

    private final Iterator<E> baseIterator; // non opzionale
    private boolean returnNext = true; /* stabilisce se il prossimo elemento di
        baseIterator va restituito; inizialmente true poiche' il primo
        elemento di baseIterator e' in posizione dispari */

    public OddOnlyIterator(Iterator<E> baseIterator) { /* completare */ }
    public boolean hasNext() { /* completare */ }
    public E next() { /* completare */ }
}
```

Per esempio, nel codice sottostante viene creato l'iteratore `altIt` a partire dall'iteratore `it` della lista `[2, 4, 6, 8]` e l'iterazione su `altIt` restituisce solo gli elementi 2 e 6.

```
Iterator<Integer> it = asList(2, 4, 6, 8).iterator();
Iterator<Integer> altIt = new OddOnlyIterator<>(it);
while (altIt.hasNext())
    System.out.println(altIt.next()); // stampa 2\n6\n
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(long l) { return "P.m(long)"; }
    String m(int i) { return "P.m(int)"; }
}
public class H extends P {
    String m(long l) { return super.m(l) + " H.m(long)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
}
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(42L)`

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

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

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

(a) Il literal `42` ha tipo statico **`int`** e il tipo statico di `p` è `P`, quindi entrambi i metodi di `P` sono accessibili è applicabili per sottotipo, ma `m(int)` è più specifico poiché **`int`** $\leq$ **`long`**. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(int)` in `P`.

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

(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(int)` ereditato da `H` e viene stampata la stringa `"P.m(int)"`.

(c) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`, i metodi applicabili per sotto-tipo sono gli stessi dei due punti precedenti visto che **`int`**$\not\leq$`Integer`, quindi viene selezionato il metodo `m(int)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa `"P.m(int)"`.

(d) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `p` è `P`, quindi `m(long)` è l'unico metodo accessibile e applicabile per sotto-tipo (dato che **`long`** $\not\leq$ **`int`**).

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`.

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

(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(`**`long`**`)` ridefinito in `H`; l'invocazione **`super`**`.m(l)` viene risolta come al punto precedente poiché `l` ha tipo statico **`long`**; viene stampata la stringa `"P.m(long) H.m(long)"`.

(f) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `h` è `H`, il metodo applicabile per sotto-tipo è `"m(long)"` come per i due punti precedenti visto che **`long`**$\not\leq$`Integer`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa `"P.m(long) H.m(long)"`.