Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 20 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("(\\s+)|(false|true)|(/[^/]*/([gim]))");
7          Matcher m = regEx.matcher("false /true/i");
8          m.lookingAt();
9          assert m.group(0).equals("false");
10         assert m.group(2) != null;
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(0).equals("/");
14         assert m.group(1) == null;
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group(0).equals("/true/i");
18         assert m.group(4).equals("i");
19     }
20 }
```

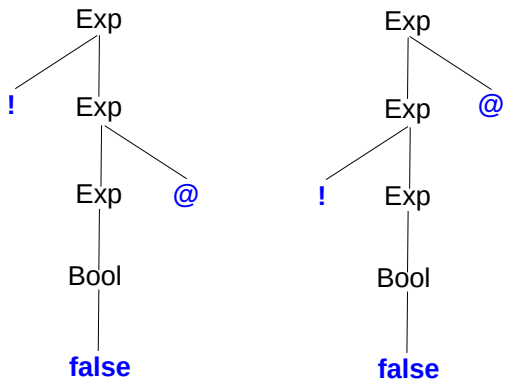
Soluzione:

- **assert** `m.group(0).equals("false");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `false /true/i` 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 `false` (appartenente ai soli gruppi 0 e 2), quindi l'asserzione ha successo;
- **assert** `m.group(2) != null;` (linea 10): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(2)` restituisce la stringa `false` e l'asserzione ha successo;
- **assert** `m.group(0).equals("/");` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `false` (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 1, quindi `m.group(0)` restituisce una stringa di spazi bianchi che è diversa dalla stringa `/` e l'asserzione fallisce;
- **assert** `m.group(1) == null;` (linea 14): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(1)` restituisce una stringa e l'asserzione fallisce;
- **assert** `m.group(0).equals("/true/i");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `/`) e l'invocazione del metodo `lookingAt()` ha successo poiché la stringa `/true/i` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3 (qualsiasi stringa di caratteri diversi da `/` delimitata da `/` e opzionalmente terminata da `g`, `i` o `m`); per tale motivo, `m.group(0)` restituisce tale stringa e l'asserzione ha successo;
- **assert** `m.group(4).equals("i");` (linea 18): lo stato del matcher non è cambiato rispetto alla linea sopra, il gruppo 4 individua la sotto-espressione regolare `[gim]` che specifica il carattere opzionale che segue il secondo delimitatore `/` e corrisponde, in questo caso, a `i`, quindi l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
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@`



(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 `!`.

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

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

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

   (a) Definire `swap` senza uso di parametri di accumulazione.
   (b) Definire `swap` usando un parametro di accumulazione affinché la ricorsione sia di coda.
   (c) Definire `swap` come specializzazione della funzione `map:('a -> 'b) -> 'a list -> 'b list`.

Soluzione: Vedere il file `soluzione.ml`.

3. Completare la seguente classe `CondIterator` che permette di creare iteratori a partire da un iteratore di base `baseIterator` e un predicato `cond` che decide se l'iterazione di `baseIterator` può continuare.

```
import java.util.Iterator;
import java.util.function.Predicate; // public interface Predicate<E> {boolean test(E el);}
public class CondIterator<E> implements Iterator<E> {
    private final Iterator<E> baseIterator; // non opzionale
    private final Predicate<E> cond; // non opzionale
    private E cachedNext; // memorizza il prossimo elemento di baseIterator
    private boolean nextIsCached; /* true se e solo cachedNext contiene
                                     gia' il prossimo elemento di baseIterator;
                                     campo necessario perche' baseIterator
                                     potrebbe avere elementi null */
    // svuota la cache
    private void resetCache() { cachedNext = null; nextIsCached = false; }
    /* salva nella cache il prossimo elemento di baseIterator
       nota bene: da usare solo se tale elemento esiste */
    private void cacheNext() { cachedNext = baseIterator.next(); nextIsCached = true; }
    public CondIterator(Iterator<E> baseIterator, Predicate<E> cond) { /* completare */ }
    /*
     * restituisce true se e solo se
     * baseIterator ha un prossimo elemento el e cond.test(el) restituisce true
```

```
     * nota bene: il prossimo elemento el potrebbe gia' essere in cache;
     * se non in cache, allora se esiste viene salvato in cache
     * */
    public boolean hasNext() { /* completare */ }
    /*
     * lancia NoSuchElementException se non esiste un prossimo elemento,
     * altrimenti restituisce l'elemento in cache e fa reset della cache
     * */
    public E next() { /* completare */ }
}
```

Per esempio, il codice sottostante crea un iteratore `condIt` a partire dall'iteratore di stringhe `it`, in modo che `condIt` continui a restituire gli elementi `el` di `it` finché la condizione `!"three".equals(el)` è verificata.

```
Iterator<String> it = asList("one", "two", "three", "four").iterator();
Predicate<String> notEqThree = ...; // test(String el){return !"three".equals(el);}
CondIterator<String> condIt = new CondIterator<>(it, notEqThree);
String elem = null;
while (condIt.hasNext())
    elem = condIt.next();
assert "two".equals(elem);
it = asList("one", "two", "four").iterator();
condIt = new CondIterator<>(it, notEqThree);
while (condIt.hasNext())
    elem = condIt.next();
assert "four".equals(elem);
```

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(Integer i) {
        return "P.m(Integer)";
    }
}
public class H extends P {
    String m(Long l) {
        return super.m(l) + " H.m(Long)";
    }
    String m(int i) {
        return super.m(i) + " 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(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 nessun metodo di `P` accessibile è applicabile per sottotipo poiché **`int`** $\not\leq$ `Long, Integer`, mentre l'unico metodo accessibile e applicabile per boxing conversion è quello con segnatura `m(Integer)` poiche `Integer` $\not\leq$ `Long`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `P`.

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

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

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

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

(d) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `p` è `P`, quindi nessun metodo di `P` accessibile è applicabile per sottotipo poiché **`long`** $\not\leq$ `Long, Integer`, mentre l'unico metodo accessibile e applicabile per boxing conversion è quello con segnatura `m(Long)` poiche `Long` $\not\leq$ `Integer`. 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` e l'unico metodo accessibile e applicabile per sottotipo è quello con segnatura `m(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`, quindi nessun metodo di `H` accessibile è applicabile per sottotipo poiché **`long`** $\not\leq$ **`int`**`, Long, Integer`, mentre l'unico metodo applicabile per boxing conversion è quello con segnatura `m(Long)` poiché `Long` $\not\leq$ `Integer`.

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