Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta dell'11 luglio

a.a. 2015/2016

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("([A-Z][A-Z$]*)|([0-9]+(\\.[0-9]*[eE]-?[0-9]+)?)|<=|<|(\\s+)");
7        Matcher m = regEx.matcher("V$<=3.14e00");
8        m.lookingAt();
9        assert m.group(1).equals("V$");
10       m.region(m.end(), m.regionEnd());
11       assert m.lookingAt();
12       assert m.group(0).equals("<");
13       m.region(m.end(), m.regionEnd());
14       assert m.lookingAt();
15       assert m.group(2).equals("3.14e00");
16       assert m.group(3).equals(".14e00");
17    }
18 }
```

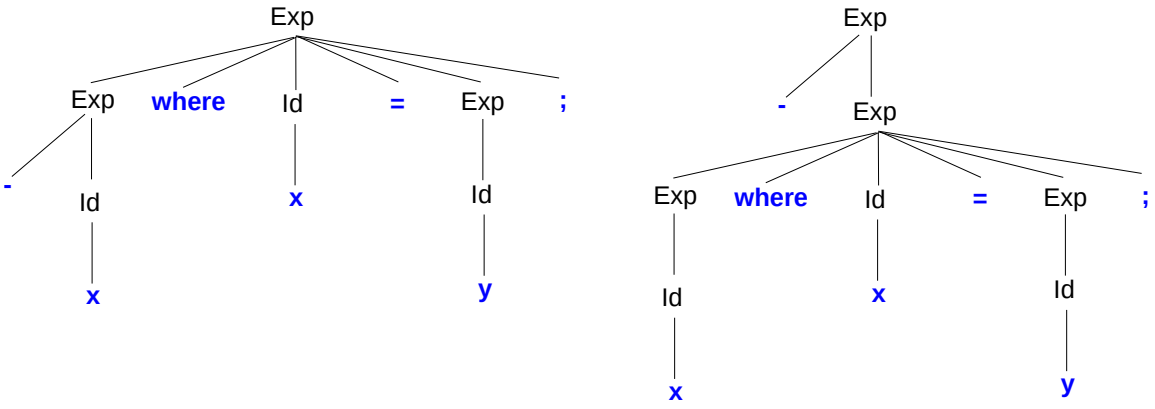
Soluzione:

- `assert m.group(1).equals("V$");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `V$<=3.14e00` 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 `V$` (stringa appartenente al gruppo di indice 1), quindi l'asserzione ha successo;
- `assert m.lookingAt();` (linea 11): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `V$` (ossia `<`); l'invocazione di `lookingAt()` restituisce **`true`** poiché `<=` appartiene all'espressione regolare (solo gruppo 0), quindi l'asserzione ha successo;
- `assert m.group(0).equals("<");` (linea 12): per le motivazioni del punto precedente l'asserzione fallisce, dato che la stringhe `"<"` e `"<="` sono diverse;
- `assert m.lookingAt();` (linea 14): alla linea 13 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `<=` (ossia `3`) e l'invocazione del metodo `lookingAt()` ha successo poiché `3.14e00` appartiene all'espressione regolare (gruppi 0 e 2, mentre la sottostringa `.14e00` appartiene anche al gruppo 3), quindi l'asserzione ha successo;
- `assert m.group(2).equals("3.14e00");` (linea 15): per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(3).equals(".14e00");` (linea 16): per i motivi del punto 4 l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp where Id = Exp ; | - Exp  | ( Exp ) | Id
Id   ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `-x where x=y;`



(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 `Minus` per poter attribuire precedenza all'operatore unario `-`.

```
Exp   ::= Exp where Id = Exp ; | Minus
Minus ::= - Minus | ( Exp ) | Id
Id    ::= x | y | z
```

2. Considerare la funzione `remove : ('a -> bool) -> 'a list -> 'a list` che rimuove da una data lista tutti gli elementi che verificano il predicato p. Esempio:

```
# remove (fun x -> x < 0) [-1;-2;1;2;-3]
- : int list = [1; 2]
```

(a) Definire la funzione `remove` senza uso di parametri di accumulazione.

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

(c) Definire la funzione `remove` come specializzazione della funzione `it_list` così definita:

```
let rec it_list f a = function x::l -> it_list f (f a x) l | _ -> a;;
val it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni su stringhe formate dall'operatore binario di concatenzione, dall'operatore unario *reverse*, dai literal di tipo stringa e dagli identificatori di variabile.

```
public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
   T visitReverse(Exp exp);
   T visitCocat(Exp left, Exp right);
   T visitVarId(String name);
   T visitStringLit(String value);
}
public abstract class UnaryOp implements Exp {
   protected final Exp exp;
   protected UnaryOp(Exp exp) { /* completare */   }
   public Exp getLeft() { return exp; }
}
public abstract class BinaryOp implements Exp {
   protected final Exp left;
   protected final Exp right;
   protected BinaryOp(Exp left, Exp right) { /* completare */   }
   public Exp getLeft() { return left; }
   public Exp getRight() { return right; }
}
public abstract class AbstractLit<V> implements Exp {
   protected final V value;
   protected AbstractLit(V value) { /* completare */ }
   public V getValue() { return value; }
   public int hashCode() { return value.hashCode(); }
}
public class Concat extends BinaryOp {
   public Concat(Exp left, Exp right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class Reverse extends UnaryOp {
   public Reverse(Exp exp) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class StringLit extends AbstractLit<String> {
   public StringLit(String value) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
   public final boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (!(obj instanceof StringLit))
         return false;
      return value == ((StringLit) obj).value;
   }
}
public class VarId implements Exp {
   private final String name;
   public VarId(String name) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
   public String getName() { return name; }
}
```

   (a) Completare le definizioni dei costruttori di tutte le classi.

   (b) Completare le definizioni dei metodi `accept` delle classi `Concat`, `Reverse`, `StringLit`, e `VarId`.

(c) Completare la classe `ContainsVarId` che controlla se un'espressione contiene una data variabile. Per esempio, le seguenti asserzioni hanno successo:

```
Exp exp = new Concat(new StringLit("one"), new Reverse(new VarId("x")));
assert exp.accept(new ContainsVarId(new VarId("x")));
assert !exp.accept(new ContainsVarId(new VarId("y")));

public class ContainsVarId implements Visitor<Boolean> {
   private final String varName;
   public ContainsVarId(VarId var) { /* completare */ }
   public Boolean visitReverse(Exp exp) { /* completare */ }
   public Boolean visitCocat(Exp left, Exp right) { /* completare */ }
   public Boolean visitVarId(String name) { /* completare */ }
   public Boolean visitStringLit(String value) { /* completare */ }
}
```

(d) Completare la classe `CountStringLit` che conta quanti literal contiene un'espressione. Per esempio, la seguente asserzione ha successo:

```
Exp exp = new Concat(new StringLit("one"), new Concat(new VarId("x"), new StringLit("one")));
assert exp.accept(new CountStringLit()).equals(2);

public class CountStringLit implements Visitor<Integer> {
   public Integer visitReverse(Exp exp) { /* completare */ }
   public Integer visitCocat(Exp left, Exp right) { /* completare */ }
   public Integer visitVarId(String name) { /* completare */ }
   public Integer visitStringLit(String value) { /* completare */ }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(int i) {
      return "P.m(int)";
   }
   String m(double d) {
      return "P.m(double)";
   }
}
public class H extends P {
   String m(Integer i) {
      return super.m(i) + " H.m(Integer)";
   }
   String m(Double l) {
      return super.m(l) + " H.m(Double)";
   }
   String m(Integer... ia) {
      return super.m(ia[0]) + " 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(4.2)`

(e) `p2.m(4.2)`

(f) `h.m(42,0)`

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`; entrambi i metodi di `P` sono accessibili e applicabili per sottotipo, ma il metodo con segnatura `m(int)` è più specifico poiché **`int`** è sottotipo di **`double`**.

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 `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(int)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(int)"`.

(c) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`; poichè nessuno dei metodi dichiarati in `H` è applicabile per sottotipo, gli unici due metodi accessibili e applicabili per sottotipo sono quelli ereditati da `P`, quindi la risoluzione dell'overloading e il comportamento a runtime sono gli stessi del punto precedente e viene stampata la stringa `"P.m(int)"`.

(d) Il literal `4.2` ha tipo statico **`double`** e il tipo statico di `p` è `P`; l'unico metodo di `P` accessibile e applicabile per sottotipo, ha segnatura `m(double)`.

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)"`.

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(double)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(double)"`.

(f) I literal `42` e `0` hanno tipo statico **`int`** e il tipo statico di `h` è `H`; dato che gli argomenti sono due, nessun metodo è applicabile per sottotipo o per conversione boxing/unboxing e l'unico metodo accessibile e applicabile è quello di arità variabile e segnatura `m(Integer...)`.

Il tipo dinamico dell'oggetto in `h` è `H` e nel corpo del metodo di segnatura `m(Integer...)` l'espressione `ia[0]` ha tipo statico `Integer`; per l'invocazione **`super`**`.m(ia[0])` non esistono metodi in `P` accessibili e applicabili per sottotipo, mentre entrambi i metodi di `P` sono applicabili per unboxing e sottotipo, ma il metodo con segnatura `m(int)` è più specifico poiché **`int`** è sottotipo di **`double`**.

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