Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 5 giugno 2017

a.a. 2016/2017

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-zA-Z][0-9]+)|((0[0-7]*)|([1-9][0-9]*))|(\\s+)");
7          Matcher m = regEx.matcher("017 17 a01");
8          m.lookingAt();
9          assert m.group(2).equals("017");
10         assert m.group(3).equals("017");
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         m.region(m.end(), m.regionEnd());
14         m.lookingAt();
15         assert m.group(2).equals("17");
16         assert m.group(3) != null;
17         m.region(m.end(), m.regionEnd());
18         m.lookingAt();
19         m.region(m.end(), m.regionEnd());
20         m.lookingAt();
21         assert m.group(0).equals("a01");
22         assert m.group(1).equals("a01");
23         assert m.group(2) != null;
24     }
25 }
```

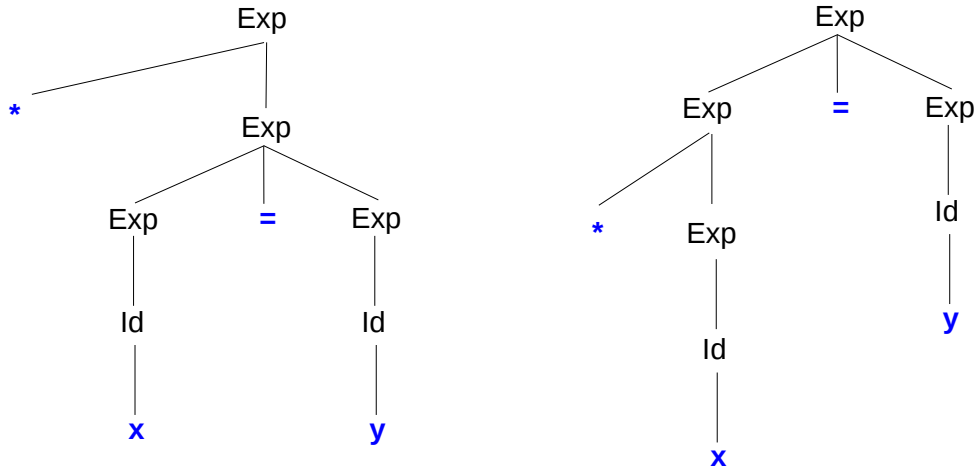
Soluzione:

- `assert m.group(2).equals("017");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `017 17 a01` 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 `017` (stringa appartenente ai gruppi di indice 0, 2 e 3), quindi l'asserzione ha successo;
- `assert m.group(3).equals("017");` (linea 10): lo stato del matcher non è cambiato rispetto alla linea precedente, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(2).equals("17");` (linea 15): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `017` (uno spazio bianco), l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene all'espressione regolare (gruppo 5), quindi la seguente invocazione del metodo `region` sposta l'inizio della regione all'inizio della stringa `17 a01`. La successiva invocazione del metodo `lookingAt()` ha successo poiché la stringa `17` appartiene ai gruppi di indice 0, 2 e 4 (ma non 3, visto che la prima cifra non è 0), quindi l'asserzione ha successo;
- `assert m.group(3) != null;` (linea 16): l'asserzione fallisce per i motivi del punto precedente, dato che lo stato del matcher è rimasto invariato;
- `assert m.group(0).equals("a01");` (linea 21): alla linea 17 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `17` (uno spazio bianco), l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene all'espressione regolare (gruppo 5), quindi la seguente invocazione del metodo `region` sposta l'inizio della regione all'inizio della stringa `a01`. La successiva invocazione del metodo `lookingAt()` ha successo poiché la stringa `a01` appartiene al gruppo di indice 0 e 1, quindi l'asserzione ha successo;
- `assert m.group(1).equals("a01");` (linea 22): l'asserzione ha successo per i motivi del punto precedente, dato che lo stato del matcher è rimasto invariato;
- `assert m.group(2) != null;` (linea 23): l'asserzione fallisce per i motivi del punto precedente, dato che lo stato del matcher è rimasto invariato.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= * Exp | 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=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 `Deref` per poter attribuire precedenza all'operatore unario `*` e determinare l'associatività dell'operatore `=`.

```
Exp   ::= Exp = Deref | Deref
Deref ::= * Deref | ( Exp ) | Id
Id    ::= x | y | z
```

2. Sia `replace : ('a -> bool) -> 'a -> 'a list -> 'a list` la funzione tale che `replace` p x l sostituisce con x tutti gli elementi della lista l che verificano il predicato p, lasciando i restanti elementi invariati. Esempio:

```
# replace (fun x->x<0) 0 [-1;2;3;-4;-5]
- : int list = [0; 2; 3; 0; 0]
```

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

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

(c) Definire la funzione `replace` come specializzazione della funzione `map` così definita:

```
# let rec map f = function [] -> [] | h::t -> f h::map f t
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare la seguente implementazione degli alberi della sintassi astratta (AST) di un semplice linguaggio di espressioni booleane formate a partire dagli operatori standard (and, or e not), dai literal booleani e dalle variabili.

```
public interface Exp { <T> T accept(Visitor<T> visitor); }
public interface Visitor<T> {
   T visitLit(boolean value);
   T visitVar(String name);
   T visitNot(Exp exp);
   T visitAnd(Exp left, Exp right);
   T visitOr(Exp left, Exp right);
}
public abstract class BinOp implements Exp {
        final protected Exp left, right;
        protected BinOp(Exp left, Exp rigth) { /* completare */ }
}
public class LitExp implements Exp {
   private final boolean value;
   public LitExp(boolean value) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class VarExp implements Exp {
   private final String name;
   public VarExp(String name) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class NotExp implements Exp {
   private final Exp exp;
   public NotExp(Exp exp) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class OrExp extends BinOp {
   public OrExp(Exp left, Exp right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class AndExp extends BinOp {
   public AndExp(Exp left, Exp right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

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

   (b) Completare le definizioni dei metodi `accept` delle classi `LitExp`, `VarExp`, `NotExp`, `OrExp` e `AndExp`.

   (c) Completare la classe `Display`, i cui visitor restituiscono la stringa corrispondente alla sintassi concreta dell'espressione rappresentata dall'AST visitato, usando le convenzioni usuali: operatori binari infissi `&&` e `||` con parentesi tonde per evitare problemi di precedenza tra operatori, operatore unario `!` prefisso, nessuno spazio tra i vari lessemi. Esempio:

```
Exp e = new OrExp(new AndExp(new LitExp(true), new VarExp("x")),
   new NotExp(new AndExp(new VarExp("y"), new VarExp("x"))));
System.out.println(e.accept(new Display())); // stampa ((true&&x)||!(y&&x))

public class Display implements Visitor<String> {
        public String visitLit(boolean value) { /* completare */ }
        public String visitVar(String name) { /* completare */ }
        public String visitNot(Exp exp) { /* completare */ }
        public String visitAnd(Exp left, Exp right) { /* completare */ }
        public String visitOr(Exp left, Exp right) { /* completare */ }
}
```

   (d) Completare la classe `Subst`, i cui visitor costruiscono un nuovo AST ottenuto da quello visitato rimpiazzando le occorrenze dei nodi variabile identificati da `name` con il nodo literal che rappresenta il valore `value`. Esempio:

```
Exp e = new OrExp(new AndExp(new LitExp(true), new VarExp("x")),
   new NotExp(new AndExp(new VarExp("y"), new VarExp("x"))));
e = e.accept(new Subst("x", false));
System.out.println(e.accept(new Display())); // stampa ((true&&false)||!(y&&false))

public class Subst implements Visitor<Exp> {
   private final String name;
   private final boolean value;
   public Subst(String name, boolean value) { /* completare */ }
   public Exp visitLit(boolean value) { /* completare */ }
   public Exp visitVar(String name) { /* completare */ }
   public Exp visitNot(Exp exp) { /* completare */ }
   public Exp visitAnd(Exp left, Exp right) { /* completare */ }
   public Exp visitOr(Exp left, Exp right) { /* completare */ }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Object o) {
        return "P.m(Object)";
    }
    String m(Number n) {
        return "P.m(Number)";
    }
    String m(Object... os) {
        return "P.m(Object...)";
    }
}
public class H extends P {
    String m(Number n) {
        return super.m(n) + " H.m(Number)";
    }
    String m(double d) {
        return super.m(d) + " H.m(double)";
    }
}
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(Double.valueOf(42.0))`

(b) `p2.m(Double.valueOf(42.0))`

(c) `h.m(Double.valueOf(42.0))`

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`

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

(a) Il literal `42.0` ha tipo statico **`double`** e l'unico metodo statico `valueOf` della classe `Double` accessibile e applicabile restituisce un valore di tipo `Double`. Il tipo statico di `p` è `P`, quindi esistono due metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(Number)` è più specifico del metodo con segnatura `m(Object)` poiché `Number` $\leq$ `Object`.

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

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

(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`, quindi viene eseguito il metodo con segnatura `m(Number)` ridefinito in `H`. L'argomento `n` ha tipo statico `Number`, quindi per ragioni analoghe a quelle del punto precedente, per l'invocazione **`super`**`.m(n)` il metodo accessibile, applicabile e più specifico in `P` ha segnatura `m(Number)`.

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

(c) Il tipo statico di `h` è `H` e come nei casi precedenti, l'argomento del metodo `m` ha tipo statico `Double`. Rispetto ai casi precedenti, i metodi applicabili e accessibili sono gli stessi, quindi l'overloading viene risolto come al punto precedente.

Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa `"P.m(Number) H.m(Number)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`, non esistono metodi in `P` accessibili e applicabili per sottotipo, mentre i due metodi con segnatura `m(Number)` e `m(Object)` sono accessibili e applicabili per boxing e widening reference conversion. Il primo è più specifico poiché `Number` $\leq$ `Object`.

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

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

(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`, quindi il comportamento è lo stesso di quello al punto (b).

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

(f) Il literal `42.0` ha tipo statico **`double`**, il tipo statico di `h` è `H` e l'unico metodo della classe `H` accessibile e applicabile per sottotipo è il metodo con segnatura `m(`**`double`**`)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(`**`double`**`)` in `H`. L'argomento `d` ha tipo statico **`double`**, quindi per ragioni analoghe a quelle del punto precedente, per l'invocazione **`super`**`.m(d)` il metodo accessibile, applicabile e più specifico in `P` ha segnatura `m(Number)`.

Viene stampata la stringa `"P.m(Number) H.m(double)"`.