Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 22 gennaio

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-zA-Z][a-zA-Z0-9]*)|(\"([^\"]*)\")|(\\s+)");
7          Matcher m = regEx.matcher("x3   \"x4\"");
8          m.lookingAt();
9          assert m.group(1).equals("x3");
10         assert m.group(2) == null;
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(4) == null;
14         m.region(m.end(), m.regionEnd());
15         assert m.lookingAt();
16         assert m.group(2).equals("\"x4\"");
17         assert m.group(3).equals("x4");
18     }
19 }
```

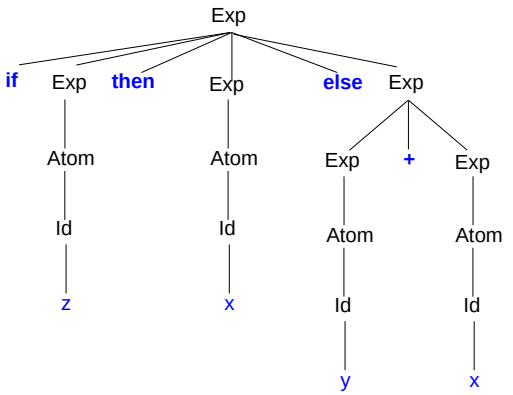
Soluzione:

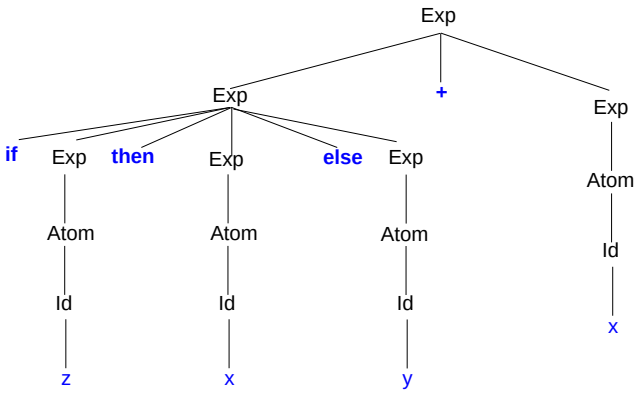
- `assert m.group(1).equals("x3");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `x3  "x4"` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `x3` (stringa interamente appartenente al gruppo di indice 1), quindi l'asserzione ha successo;
- `assert m.group(2) == null;` (linea 10): poiché il gruppo di indice 2 è disgiunto dal gruppo di indice 1 l'asserzione ha successo;
- `assert m.group(4) == null;` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `x3` (spazio bianco) e alla linea seguente l'invocazione di `lookingAt()` restituisce **`true`** poiché la sequenza di spazi bianchi appartiene al gruppo di indice 4, quindi l'asserzione fallisce;
- `assert m.lookingAt();` (linea 15): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo alla sequenza di spazi bianchi (carattere `"`). L'asserzione ha successo poiché la stringa `"x4"` appartiene al gruppo di indice 2;
- `assert m.group(2).equals("\"x4\"");` (linea 16): l'asserzione ha successo per motivazioni date al punto precedente;
- `assert m.group(3).equals("x4");` (linea 17): il gruppo 3 corrisponde alle stringhe del gruppo 2 private dei due delimitatori di stringa, quindi l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  if Exp then Exp else 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 `if z then x else y + x`





(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 `If` per poter attribuire precedenza all'operatore condizionale e imporre che l'addizione associ da sinistra.

```
Exp ::=  Exp + If | If
If ::= if Exp then Exp else If | ( Exp ) | Id
Id ::= x | y | z
```

2. Considerare la funzione `flat_map : ('a -> 'b list) -> 'a list -> 'b list` tale che `flat_map` f `[`e_1`;`e_2`;...;`e_n`]` restituisce la lista fe_1`@`fe_2`@...@`fe_n.

   Esempi:

```
# flat_map (fun x -> [x;x]) ['a';'b';'c']

- : char list = ['a'; 'a'; 'b'; 'b'; 'c'; 'c']

# flat_map (fun x -> [x-1;x;x+1]) [1;2;3]

- : int list = [0; 1; 2; 1; 2; 3; 2; 3; 4]
```

   (a) Definire la funzione `flat_map` direttamente, senza uso di parametri di accumulazione.

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

   (c) Definire la funzione `flat_map` 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. (a) Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni formate dagli operatori binari di addizione e moltiplicazione, dai literal di tipo intero e dagli identificatori di variabile.

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

public interface Visitor<T> {
    T visitAdd(Exp left, Exp right);
    T visitMul(Exp left, Exp right);
    T visitVarIdent(String name);
    T visitIntLiteral(int value);
}
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 class Add extends BinaryOp {
    public Add(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 31 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Add))
            return false;
        Add other = (Add) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}
public class Mul extends BinaryOp {
    public Mul(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 37 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Mul))
            return false;
        Mul other = (Mul) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}
public class IntLiteral implements Exp {
    protected final int value;
    protected IntLiteral(int value) { /* completare */ }
    public int getValue() { return value; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return value; }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof IntLiteral))
            return false;
        return value == ((IntLiteral) obj).value;
    }
}
public class VarIdent implements Exp {
    private final String name;
    public VarIdent(String name) { /* completare */ }
    public String getName() { return name; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return name.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarIdent))
            return false;
        return name.equals(((VarIdent) obj).name);
    }
}
```

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

(c) Completare le definizioni dei metodi `accept` delle classi `Add`, `Mul`, `IntLiteral`, e `VarIdent`.

(d) Completare la classe `DisplayPostfix` che permette di visualizzare l'espressione in forma polacca post-fissa. Per esempio, la seguente asserzione ha successo:

```
assert new Mul(new IntLiteral(42), new Add(new VarIdent("x"), new VarIdent("y"))).
    accept(new DisplayPostfix()).equals("42 x y + *").

public class DisplayPostfix implements Visitor<String> {
        public String visitAdd(Exp left, Exp right) { /* completare */ }
        public String visitMul(Exp left, Exp right) { /* completare */ }
        public String visitVarIdent(String name) { /* completare */ }
        public String visitIntLiteral(int value) { /* completare */ }
}
```

(e) Completare la classe `SimplifyNeutral` che permette di semplificare un'espressione applicando le seguenti identità: $e + 0 = 0 + e = e$, $e * 1 = 1 * e = e$. Per esempio, la seguente asserzione ha successo:

```
Exp exp1 = new Mul(new Add(new VarIdent("x"), new IntLiteral(1)),
    new Add(new IntLiteral(1), new IntLiteral(0)));
Exp exp2 = new Add(new VarIdent("x"), new IntLiteral(1));
assert exp1.accept(new SimplifyNeutral()).equals(exp2);

public class SimplifyNeutral implements Visitor<Exp> {
        public Exp visitAdd(Exp left, Exp right) { /* completare */ }
        public Exp visitMul(Exp left, Exp right) { /* completare */ }
        public Exp visitVarIdent(String name) { /* completare */ }
        public Exp visitIntLiteral(int 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(long l) { return "P.m(long)"; }
}
public class H extends P {
    String m(int i) { return super.m(i) + " H.m(int)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Integer... l) { return (l.length > 0 ? super.m(l[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) `p.m(new Long(42))`

(d) `h.m(new Long(42))`

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

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

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

(a) Il literal `42` ha tipo statico `int`; il tipo statico di `p` è `P` ed entrambi i metodi di `P` sono accessibili e applicabili per sotto-tipo, ma quello con segnatura `m(int)` è più specifico.

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 per esattamente lo stesso motivo del 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(int)` ridefinito in `H`. Poiché `i` ha tipo statico **`int`**, l'overloading per l'invocazione **`super`**`.m(i)` viene risolta come al punto precedente e, quindi, viene chiamato il metodo della classe `P` con segnatura `m(int)`.

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

(c) L'espressione **`new`** `Long(42)` ha tipo statico `Long` poiché `42` ha tipo **`int`** che è sotto-tipo del tipo **`long`** del parametro dell'unico costruttore applicabile per sotto-tipo di `Long` (l'altro ha tipo `String`); il tipo statico di `p` è `P`. Nessun metodo della classe `P` è applicabile per sotto-tipo, mentre per unboxing l'unico metodo accessibile e applicabile è quello con segnatura `m(long)`.

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)"`.

(d) L'espressione **`new`** `Long(42)` ha tipo statico `Long` per gli stessi motivi del punto precedente e il tipo statico di `h` è `H`; nessun metodo in `H` è applicabile per sotto-tipo o per unboxing, quindi l'overloading viene risolto come al punto precedente.

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

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

(e) Il literal `42` ha tipo statico **`int`**; il tipo statico di `p2` è `P` e nessun metodo accessibile di `P` è applicabile visto che entrambi i metodi hanno arità costante 1, quindi verrà segnalato un errore di tipo.

(f) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`; nessun metodo di `H` e `P` è applicabile per sotto-tipo o per boxing, dato che in entrambi i casi tutti i metodi considerati hanno arità costante 1. Il metodo in `H` con arità variabile e segnatura `m(Integer...)` è applicabile visto che **`int`** può essere convertito a `Integer` tramite boxing.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Integer...)`. Poiché in questo caso vengono passati due argomenti `l.length` si valuta in **`true`** e viene eseguita l'invocazione **`super`**`.m(l[0])`; l'argomento `l[0]` ha tipo statico `Integer`, nessun metodo accessibile in `P` è applicabile per sotto-tipo, mentre entrambi i metodi sono applicabili per unboxing, ma quello con segnatura `m(int)` è più specifico.

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