

Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2015/2016

22 gennaio 2016

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali falliscono, motivando la risposta.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MatcherTest {
    public static void main(String[] args) {
        Pattern regex = Pattern.compile("[a-zA-Z][a-zA-Z0-9]*|\\\"([^\"]*)\\\"|\\\"(\\\\s+)\\\"");
        Matcher m = regex.matcher("x3  \"x4\"");
        m.lookingAt();
        assert m.group(1).equals("x3");
        assert m.group(2) == null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(4) == null;
        m.region(m.end(), m.regionEnd());
        assert m.lookingAt();
        assert m.group(2).equals("\"x4\"");
        assert m.group(3).equals("x4");
    }
}
```

- (b) Mostrare che la seguente grammatica è ambigua.

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

- (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**.

2. Considerare la funzione `flat_map : ('a -> 'b list) -> 'a list -> 'b list` così definita:
`flat_map f [e1; e2; ...; en]` restituisce la lista `f e1 @ f e2 @ ... @ f en`.

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>
```

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 */ }
}

```

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)`