Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2016/2017

5 giugno 2017

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][0-9]+)|((0[0-7]*)|([1-9][0-9]*))|(\\s+)");
        Matcher m = regEx.matcher("017 17 a01");
        m.lookingAt();
        assert m.group(2).equals("017");
        assert m.group(3).equals("017");
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(2).equals("17");
        assert m.group(3) != null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(0).equals("a01");
        assert m.group(1).equals("a01");
        assert m.group(2) != null;
    }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= * 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. 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>
```

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 */ }
}
```

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)`