Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2013/2014

17 gennaio 2014

1. (a) Dato il seguente frammento di codice Java, indicare quali delle asserzioni contenute in esso falliscono, motivando la risposta.

```
Pattern p = Pattern.compile("(begin|end|until|(?<HEAD>[a-zA-Z_$]))(?<TAIL>[a-zA-Z0-9_]*)");
Matcher m = p.matcher("end0");
m.matches();
assert m.group("HEAD") == null;
assert m.group("TAIL").equals("0");
m = p.matcher("until");
m.matches();
assert "u".equals(m.group("HEAD"));
assert m.group("TAIL").equals("");
m = p.matcher("$begin");
m.matches();
assert m.group("HEAD").equals("$");
assert m.group("TAIL").equals("$begin");
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Term | Exp + Term
Term ::= Id | - Exp | ( Exp )
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 `get_all : 'a -> ('a * 'b) list -> 'b list`, così specificata: `get_all` k l restituisce la lista di tutti i valori v, eventualmente ripetuti, per i quali esiste una coppia (k, v) nella lista l; la lista dei valori deve rispettare l'ordine della lista l passata come argomento: se in l la coppia (k, v_1) precede la coppia (k, v_2), allora in `get_all` k l il valore v_1 precederà il valore v_2.

   Esempio:

```
# get_all_it 0 [(0,"a");(1,"bc");(0,"az");(2,"");(0,"az")];;
- : string list = ["a"; "az"; "az"]
```

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

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

   (c) Definire la funzione `get_all` 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. Considerare le dichiarazioni dei seguenti tipi, che modellano dei raggruppamenti arbitrari di figure colorate e implementano il design pattern *visitor*. Solo per curiosità, il sistema di coordinate usato è quello "a schermo", dove l'asse delle Y è orientato verso il basso (tutto quello che vi chiederemo di implementare è indipentente dall'orientamento degli assi).

   L'interfaccia generica `ShapeVisitor<T>` rappresenta un *visitor* che effettua un'operazione che restituisce un oggetto di tipo `T`. La classe `Point` rappresenta i punti colorati; un oggetto di tipo `Point` deve poter essere costruito a partire da un colore `Color color` e le due coordinate **`int`** `x` e **`int`** `y`, e deve offrire degli opportuni *getter*.

   La classe `Rectangle` rappresenta i rettangoli colorati, dove le coordinate `(x1,y1)` rappresentano l'angolo in alto a sinistra e `(x2, y2)` l'angolo in basso a destra.

   Infine, la classe `Group` rappresenta un raggruppamento, non vuoto, di figure; il colore di un gruppo corrisponde al colore del suo bordo, mentre le figure raggruppate all'interno mantengono i loro colori. Il costruttore, già implementato, si limita a salvare le figure raggruppate nell'array **`this`**`.shapes` (la scelta di usare due argomenti, `firstShape` e `otherShapes`, impedisce a compile-time la costruzione di `Group` vuoti).

```
interface ShapeVisitor<T> {
  T visit(Point p);
  T visit(Rectangle r);
  T visit(Group g);
}

abstract class Shape {
  private final Color color;
  public Shape(Color color) { this.color = color; }
  public Color getColor() { return color; }
  abstract <T> T accept(ShapeVisitor<T> v);
}

class Point extends Shape { /* DA COMPLETARE (1) */ }

class Rectangle extends Shape {
  private final int x1, y1, x2, y2;
  public int getX1() { return x1; }
  public int getY1() { return y1; }
  public int getX2() { return x2; }
  public int getY2() { return y2; }
  public Rectangle(Color color, int x1, int y1, int x2, int y2) {
    super(color);
    if (x1>x2 || y1>y2)
      throw new IllegalArgumentException();
    this.x1 = x1;
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
  }
  @Override <T> T accept(ShapeVisitor<T> v) {  /* DA COMPLETARE (2) */ }
}

class Group extends Shape {
  private final Shape[] shapes;
  public Group(Color borderColor, Shape firstShape, Shape... otherShapes) {
    super(borderColor);
    final int initialCapacity = otherShapes.length+1;
    List<Shape> l = new ArrayList<>(initialCapacity);
    l.add(firstShape);
    l.addAll(Arrays.asList(otherShapes));
    this.shapes = l.toArray(new Shape[initialCapacity]);
  }
  public Shape[] getShapes() { return shapes; }
  @Override <T> T accept(ShapeVisitor<T> v) {  /* DA COMPLETARE (3) */  }
}
```

   (a) Completare le definizioni delle classi `Point`, `Rectangle` e `Group`.

   (b) Completare la seguente definizione del *visitor* "predicato" `FindColor`. Un oggetto di tipo `FindColor`, costruito a partire da un certo colore `c`, deve implementare la visita che restituisce un valore di verità che corrisponde al fatto che il colore `c` sia utilizzato, o meno, all'interno di una figura.

```
class FindColor implements ShapeVisitor<Boolean> {
  private Color color;
  public FindColor(Color color) { this.color = color; }
  @Override public Boolean visit(Point p) { /* DA COMPLETARE (4) */  }
  @Override public Boolean visit(Rectangle r) {  /* DA COMPLETARE (5) */  }
  @Override public Boolean visit(Group g) {  /* DA COMPLETARE (6) */  }
}
```

(c) Date le seguenti dichiarazioni di classe, completare la definizione del *visitor* `CalculateBoundingBox`, che deve permettere di calcolare il *bounding-box*, rappresentato da un'istanza della classe `BoundingBox`, di una figura. Il bounding-box di una figura è, per definizione, il rettangolo più piccolo che contiene tutti gli elementi della figura. Quindi, per un punto corrisponde al punto stesso (ovvero, `upperLeftX==bottomRightX` e `upperLeftY==bottomRightY`), per un rettangolo corrisponde a un bounding-box con le stesse coordinate del rettangolo, e per un gruppo di figure corrisponde all'unione dei boundig-box. Notate che l'operazione di unione di bounding-box è già implementata dal metodo `union(BoundingBox)`.

```
class BoundingBox {
  private final int upperLeftX, upperLeftY, bottomRightX, bottomRightY;
  public int getUpperLeftX()   { return upperLeftX;   }
  public int getUpperLeftY()   { return upperLeftY;   }
  public int getBottomRightX() { return bottomRightX; }
  public int getBottomRightY() { return bottomRightY; }
  public BoundingBox(int upperLeftX, int upperLeftY, int bottomRightX, int bottomRightY) {
    if (upperLeftX>bottomRightX || upperLeftY>bottomRightY)
      throw new IllegalArgumentException();
    this.upperLeftX = upperLeftX;
    this.upperLeftY = upperLeftY;
    this.bottomRightX = bottomRightX;
    this.bottomRightY = bottomRightY;
  }
  BoundingBox union(BoundingBox other) {
    return new BoundingBox(
        min(this.upperLeftX, other.upperLeftX),
        min(this.upperLeftY, other.upperLeftY),
        max(this.bottomRightX, other.bottomRightX),
        max(this.bottomRightY, other.bottomRightY));
  }
  @Override public String toString() {
    return String.format("(%d, %d)->(%d, %d)", this.upperLeftX, this.upperLeftY,
                                               this.bottomRightX, this.bottomRightY);
  }
}

class CalculateBoundingBox implements ShapeVisitor<BoundingBox> {  /* DA COMPLETARE (7) */ }
```

4. Considerare le seguenti dichiarazioni di classi Java (che utilizzano le classi dichiarate nell'esercizio precedente):

```
class P {
  int  m(BoundingBox b)    { return 0;   }
  int  m(Shape s)          { return 1;   }
  int  m(Group g)          { return 2;   }
  char m(Shape s, Group g) { return 'A'; }
  int  m(Shape... ss)      { return 3;   }
}

class H extends P {
  int  m(Group... ps)      { return 4;   }
  int  m(Shape s)          { return 5;   }
  int  m(Group g, Shape s) { return 6;   }
}

public class Test {
  public static void main(String[] args) {
    H h = new H();
    P p = h;
    Group g = new Group(Color.red, new Point(Color.green, 0, 0));
    Shape s = g;
    System.out.println(...);
  }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

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) `h.m(s,s)`

(b) `h.m(g,g)`

(c) `p.m(null)`

(d) `h.m(g, null, null)`

(e) `p.m((Shape)g)`