Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 17 gennaio

a.a. 2013/2014

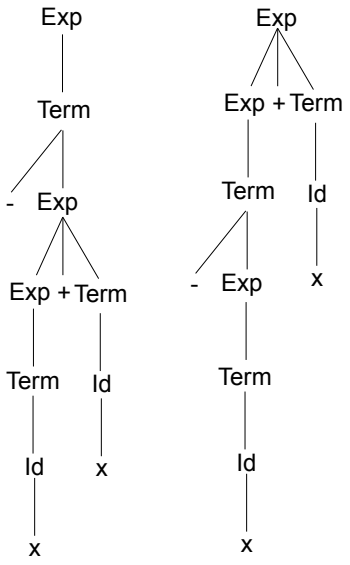
1. (a) Dato il seguente frammento di codice Java, indicare quali delle asserzioni contenute in esso falliscono, motivando la risposta.

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

   Esistono due diversi alberi di derivazione per la stringa `-x+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**.

```
Exp  ::=  Term | Exp + Term
Term ::= Id | - Term | ( Exp )
Id ::= x | y | z
```

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.

```
let rec get_all x = function
    (k,v)::l -> let l2=get_all x l in if k=x then v::l2 else l2
  | _ -> [];;
```

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

```
let get_all x l=
  let rec aux acc = function
      (k,v)::l -> aux (if k=x then v::acc else acc) l
    | _ -> acc
  in List.rev (aux [] l);;
```

(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>

let get_all x l =
  List.rev (it_list (fun acc (k,v) -> if k = x then v::acc else acc) [] l);;
```

3. (a)
```
class Point extends Shape {
    private final int x, y;
    public Point(Color color, int x, int y) {
        super(color);
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}
class Rectangle extends Shape { /* ... */
    @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}
class Group extends Shape { /* ... */
    @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}
```

(b)
```
class FindColor implements ShapeVisitor<Boolean> {
    private Color color;
    public FindColor(Color color) { this.color = color; }
    @Override public Boolean visit(Point p) {
        return p.getColor().equals(this.color);
    }
    @Override public Boolean visit(Rectangle r) {
        return r.getColor().equals(this.color);
    }
    @Override public Boolean visit(Group g) {
        if ( g.getColor().equals(this.color))
            return Boolean.TRUE;
        for (Shape s : g.getShapes())
            if (s.accept(this))
                return Boolean.TRUE;
        return Boolean.FALSE;
    }
}
```

(c)
```
class CalculateBoundingBox implements ShapeVisitor<BoundingBox> {
    @Override public BoundingBox visit(Point p) {
        final int px = p.getX();
        final int py = p.getY();
        return new BoundingBox(px, py, px, py);
    }
    @Override public BoundingBox visit(Rectangle r) {
        return new BoundingBox(r.getX1(), r.getY1(), r.getX2(), r.getY2());
    }
    @Override public BoundingBox visit(Group g) {
        Shape[] shapes = g.getShapes();
        assert shapes.length>0;
        BoundingBox result = shapes[0].accept(this);
        for (int i=1; i<shapes.length; ++i)
            result = result.union(shapes[i].accept(this));
        return result;
    }
}
```

4. (a) I tipi statici di `h` ed `s` sono, rispettivamente, `H` e `Shape`. Cercando, a partire da `H`, i metodi che si chiamano `m` e sono applicabili (per sottotipo) a due argomenti di tipo `Shape` non troviamo nulla. Idem considerando anche le conversioni. A questo punto consideriamo anche i metodi con un numero variabile di argomenti, e risulta applicabile solo `P.m(Shape...)`, che è ovviamente anche il più specifico. Quindi, la compilazione va a buon fine. A runtime, siccome il tipo statico e dinamico di `h` coincidono, viene stampato `3`.

   (b) I tipi statici di `h` e `g` sono, rispettivamente, `H` e `Group`. Cercando, a partire da `H`, i metodi che si chiamano `m` e sono applicabili (per sottotipo) a due argomenti di tipo `Group` troviamo `H.m(Group, Shape)` e `P.m(Shape, Group)`. Poiché nessuno dei due è più specifico dell'altro, l'invocazione è ambigua e abbiamo un errore di compilazione.

   (c) Il tipo statico di `p` è `P`. Cercando, a partire da `P`, i metodi che si chiamano `m` e sono applicabili (per sottotipo) all'argomento **`null`** troviamo `P.m(BoundingBox)`, `P.m(Shape)`, `P.m(Group)` e `P.m(Shape [])`. Poiché nessuno è più specifico degli altri, l'invocazione è ambigua e abbiamo un errore di compilazione.

   (d) I tipi statici di `h` e `g` sono, rispettivamente, `H` e `Group`. Cercando, a partire da `H`, i metodi che si chiamano `m` e sono applicabili (per sottotipo) a un argomento di tipo `Group` e due **`null`** non troviamo nulla. Idem considerando anche le conversioni. A questo punto consideriamo anche i metodi con un numero variabile di argomenti, e risultano applicabili `H.m(Group...)` e `P.m(Shape...)`. Il primo è più specifico del secondo, quindi, la compilazione va a buon fine e runtime, siccome il tipo statico e dinamico di `h` coincidono, viene stampato `4`.

   (e) I tipi statici di `p` e `g` sono, rispettivamente, `P` e `Group`. Il cast è staticamente corretto e l'espressione ha tipo statico `Shape`. Cercando, a partire da `P`, i metodi che si chiamano `m` e sono applicabili (per sottotipo) a un argomento di tipo `Shape` troviamo solo `P.m(Shape)`, quindi la compilazione va a buon fine. A runtime, siccome il tipo dinamico di `p` è `H`, viene stampato `5`.