Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2017/2018

10 settembre 2018

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

```
Pattern regEx = Pattern.compile("(\\s+)|([a-z][_a-zA-Z]*)|(True|False)");
Matcher m = regEx.matcher("is_False True");
m.lookingAt();
assert m.group(2) == null;
assert m.group(0).equals("is");
m.region(m.end(), m.regionEnd());
m.lookingAt();
assert m.group(1) == null;
assert m.group(0).equals("");
m.region(m.end(), m.regionEnd());
m.lookingAt();
assert m.group(3) == null;
assert m.group(0).equals("False");
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp ? | Exp in Exp | ( Exp ) | Bool
Bool ::= false | true
```

   (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 `filter_map : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list` la funzione così specificata:

   `filter_map` p f l restituisce la lista ottenuta da l eliminando gli elementi che non soddisfano il predicato p e applicando, nell'ordine, la funzione f ai restanti.

   Esempio:

```
# filter_map (fun x->x>=0.0) sqrt [-1.0;0.0;-4.0;4.0];;
- : float list = [0.0; 2.0]
```

   (a) Definire `filter_map` senza uso di parametri di accumulazione.

   (b) Definire `filter_map` usando un parametro di accumulazione affinché la ricorsione sia di coda.

   (c) Definire `filter_map` come specializzazione della funzione `it_list` o `List.fold_left`:
   `it_list:('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

3. (a) Completare le classi `IntLit` e `Add` che rappresentano i nodi di un albero della sintassi astratta corrispondenti, rispettivamente, a literal interi e all'operazione aritmetica di addizione.

```
public interface AST { <T> T accept(Visitor<T> v); }

public interface Visitor<T> {
   T visitIntLit(int i);
   T visitAdd(AST left, AST right);
}

public class IntLit implements AST {
   private final int value;
   public IntLit(int value) { /* completare */  }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Add implements AST {
   private final AST left, right;
   public Add(AST left, AST right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

(b) Completare le classi `Eval` e `ToString` che implementano visitor su oggetti di tipo `AST`.

```
/* Un visitor Eval restituisce il valore dell'espressione visitata,
   calcolato secondo le regole convenzionali */
public class Eval implements Visitor<Integer> {
   public Integer visitIntLit(int i) { /* completare */}
   public Integer visitAdd(AST left, AST right) { /* completare */}
}

/* Un visitor ToString restituisce la stringa che rappresenta l'espressione visitata
   secondo la sintassi convenzionale senza parentesi */
public class ToString implements Visitor<String> {
   public String visitIntLit(int i) { /* completare */}
   public String visitAdd(AST left, AST right) { /* completare */}
}

// Classe di prova
public class Test {
   public static void main(String[] args) {
        AST i1 = new IntLit(1), i2 = new IntLit(2), i3 = new IntLit(3);
        AST i1_plus_i2_plus_i3 = new Add(new Add(i1, i2), i3);
        assert i1_plus_i2_plus_i3.accept(new Eval()) == 6;
        assert i1_plus_i2_plus_i3.accept(new ToString()).equals("1+2+3");
   }
}
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Object o) { return "P.m(Object)"; }
   String m(String s) { return "P.m(String)"; }
}
public class H extends P {
   String m(String s) { return super.m(s) + " H.m(String)"; }
}
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) `h.m(42)`

(d) `p.m("42")`

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

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