Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2015/2016

8 settembre 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$_]*)|(0[bB]([01]|[01][01_]*[01])[lL]?)|->|-|(\\s+)");
      Matcher m = regEx.matcher("b0_0->0B1L");
      m.lookingAt();
      assert m.group(1).equals("b0_0");
      m.region(m.end(), m.regionEnd());
      assert m.lookingAt();
      assert m.group(0).equals("->");
      m.region(m.end(), m.regionEnd());
      assert m.lookingAt();
      assert m.group(2).equals("0B1");
      assert m.group(3).equals("0");
   }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= if Exp then Exp else 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 `values : ('a -> bool) -> ('a * 'b) list -> 'b list` che, preso un predicato p e una lista di coppie $(chiave, valore)$, restituisce la lista dei valori associati alle chiavi che soddisfano il predicato p, mantenendo l'ordine iniziale e le possibili ripetizioni.

   Esempi:

```
# values (fun k -> k>0) [(1,"one");(0,"zero");(2,"two")]

- : string list = ["one"; "two"]

# values (fun k -> k>0) [(1,"b");(2,"b");(0,"a")]

- : string list = ["b"; "b"]
```

   (a) Definire la funzione `values` senza uso di parametri di accumulazione.

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

   (c) Definire la funzione `values` 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 la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni che denotano insiemi di interi e sono formate dall'operatore binario di unione, dall'operatore unario di complementazione, dai literal di tipo insieme e dagli identificatori di variabile.

```
public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
   T visitComplement(Exp exp);
   T visitUnion(Exp left, Exp right);
   T visitVarId(String name);
   T visitSetLit(java.util.Set<Integer> value);
}
public abstract class UnaryOp implements Exp {
   protected final Exp exp;
   protected UnaryOp(Exp exp) { /* completare */   }
}
public abstract class BinaryOp implements Exp {
   protected final Exp left;
   protected final Exp right;
   protected BinaryOp(Exp left, Exp right) { /* completare */   }
}
public abstract class AbstractLit<V> implements Exp {
   protected final V value;
   protected AbstractLit(V value) { /* completare */ }
   public int hashCode() { return value.hashCode(); }
}
public class Union extends BinaryOp {
   public Union(Exp left, Exp right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class Complement extends UnaryOp {
   public Complement(Exp exp) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
public class SetLit extends AbstractLit<java.util.Set<Integer>> {
   public SetLit(java.util.Set<Integer> value) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
   public final boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (!(obj instanceof SetLit))
         return false;
      return value.equals(((SetLit) obj).value);
   }
}
public class VarId implements Exp {
   private final String name;
   public VarId(String name) { /* 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 `Union`, `Complement`, `SetLit`, e `VarId`.

   (c) Completare la classe `Eval` che valuta un'espressione restituendo un valore booleano secondo le seguenti regole:

   - l'operatore di complementazione corrisponde alla negazione logica;
   - l'operatore di unione corrisponde alla disgiunzione logica;
   - i literal vengono valutati in **`true`** se e solo se l'insieme rappresentato non è vuoto;
   - le variabili si valutano sempre in **`true`**.

   Per esempio, la seguente asserzione ha successo:

```
exp = new Complement(new Union(new SetLit(new java.util.HashSet<>()), new VarId("x")));
assert ! exp.accept(new Eval());

public class Eval implements Visitor<Boolean> {
   public Boolean visitComplement(Exp exp) { /* completare */ }
   public Boolean visitUnion(Exp left, Exp right) { /* completare */ }
   public Boolean visitVarId(String name) { /* completare */ }
   public Boolean visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}
```

   (d) Completare la classe `ReplaceVar` che costruisce una nuova espressione ottenuta da quella visitata rimpiazzando le variabili con il literal che rappresenta l'insieme vuoto. Per esempio, la seguente asserzione ha successo:

```
Exp exp = new Complement(new Union(new SetLit(new java.util.HashSet<>()), new VarId("x")));
assert exp.accept(new ReplaceVar()).accept(new Eval());

public class ReplaceVar implements Visitor<Exp> {
   private static final Exp emptyLit = new SetLit(new java.util.HashSet<>());
   public Exp visitComplement(Exp exp) { /* completare */ }
   public Exp visitUnion(Exp left, Exp right) { /* completare */ }
   public Exp visitVarId(String name) { /* completare */ }
   public Exp visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Number n) {
      return "P.m(Number)";
   }
   String m(Double d) {
      return "P.m(Double)";
   }
}
public class H extends P {
   String m(Integer i) {
      return super.m(i) + " H.m(Integer)";
   }
   String m(int i) {
      return super.m(i) + " H.m(int)";
   }
   String m(Number n) {
      return super.m(n) + " H.m(Number)";
   }
}
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(4.2)`

(e) `p2.m(4.2)`

(f) `h.m(4,0)`