Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2012/2013

3 settembre 2013

1. (a) Data la seguente linea di codice Java

```
Pattern p = Pattern.compile("0_*[0-7]([0-7_]*[0-7])?[lL]?");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

i. `assert p.matcher("0").matches();`

ii. `assert p.matcher("042__").matches();`

iii. `assert p.matcher("04_2L_").matches();`

iv. `assert p.matcher("04_2l").matches();`

v. `assert p.matcher("04_2L").matches();`

vi. `assert p.matcher("0__4__2").matches();`

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= App | Exp + App
App  ::= Term | App Term
Term ::= Id | + Term
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` **rimanga invariato**.

2. Considerare la funzione `delete : ('a -> bool) -> 'a list -> 'a list` così definita: `delete` p l restituisce la lista ottenuta eliminando da l tutti gli elementi che soddisfano il predicato p e lasciando invariato l'ordine degli elementi restanti.

Esempio:

```
# delete (fun x -> x mod 3 = 0) [0; 1; 2; 3; 4; 5; 6; 6; 5; 4; 3; 2; 1; 0];;
- : int list = [1; 2; 4; 5; 5; 4; 2; 1]
```

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

(b) Definire la funzione `delete` direttamente, usando un parametro di accumulazione in modo che la ricorsione sia di coda (usando eventualmente la funzione `reverse`).

(c) Definire la funzione `delete` come specializzazione della funzione `it_list` (usando eventualmente la funzione `reverse`):

```
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 seguenti dichiarazioni di classi Java:

```
public class P {
      public String m(int i) {
            return "P.m(int)";
      }

      public String m(long l) {
            return "P.m(long)";
      }

      public String m(double d) {
            return "P.m(double)";
      }

}
public class H extends P {

      public String m(Integer i) {
            return "H.m(Integer) " + super.m(i);
      }

      public String m(Long l) {
            return "H.m(Long) " + super.m(l);
      }

      public String m(Double d) {
            return "H.m(Double) " + super.m(d);
      }

}

public class Test {
       public static void main(String[] args) {
              H h = new H();
              P p = h;
              System.out.println(...);
       }

}
```

Dire, per ognuno dei casi sotto elencati, 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) `p.m(1)`

(b) `p.m((Long) 1L)`

(c) `h.m((Long) 1L)`

(d) `h.m(4.2)`

(e) `h.m(4.2F)`

(f) `h.m((Double) 4.2)`

4. Considerare i package `ast` e `visitor` che implementano abstract syntax tree e visite su di essi per espressioni booleane formate a partire da variabili, l'operatore unario di negazione e quelli binari di congiunzione e disgiunzione logica.

```
package ast;
import visitor.Visitor;
public interface Exp {
    Iterable<Exp> getChildren();
    void accept(Visitor v);
}
```

```
package ast;
import static java.util.Arrays.asList;
public abstract class AbsExp implements Exp {
    private final Iterable<Exp> children;
    protected AbsExp(Exp... children) {
        this.children = asList(children);
    }
    @Override
    public Iterable<Exp> getChildren() {
        return children;
    }
}
```

```
package ast;
import visitor.Visitor;
public class BoolLit extends AbsExp {
    final private boolean value;
    // completare
}
```

```
package ast;
import visitor.Visitor;
public class AndExp extends AbsExp {
    // completare
}
```

```
package ast;
import visitor.Visitor;
public class OrExp extends AbsExp {
    // completare
}
```

```
package ast;
import visitor.Visitor;
public class NotExp extends AbsExp {
    // completare
}
```

(a) Completare le definizioni delle classi `BoolLit`, `AndExp`, `OrExp` e `NotExp`.

(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione delle classi `EvalVisitor` e `NegateVisitor`.

```
package visitor;
import ast.*;
public interface Visitor {
    void visit(BoolLit e);
    void visit(AndExp e);
    void visit(OrExp e);
    void visit(NotExp e);
}
```

```
package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}
```

```
package visitor;
import java.util.Iterator;
public class EvalVisitor extends AbstractVisitor<Boolean> {
    // completare
}
```

```
package visitor;
import java.util.Iterator;
public class NegationVisitor extends AbstractVisitor<Exp> {
    // completare
}
```

- la classe `EvalVisitor` valuta l'espressione visitata.
  Esempio:

```
Exp exp = new AndExp(new BoolLit(true), new NotExp(new OrExp(
        new BoolLit(false), new BoolLit(true))));
EvalVisitor ev = new EvalVisitor();
exp.accept(ev);
assert !ev.getResult();
```

- la classe `NegateVisitor` genera l'espressione corrispondente alla negazione dell'espressione visitata costruita applicando le leggi di De Morgan:

$$
\begin{aligned}
\neg true &= false \\
\neg false &= true \\
\neg(e_1 \wedge e_2) &= (\neg e_1) \vee (\neg e_2) \\
\neg(e_1 \vee e_2) &= (\neg e_1) \wedge (\neg e_2) \\
\neg\neg e &= e
\end{aligned}
$$

  Esempio: dopo l'esecuzione del seguente frammento di codice, la variable `exp` contiene l'abstract syntax tree corrispondente all'espressione $false \vee (false \vee true)$.

```
Exp exp = new AndExp(new BoolLit(true), new NotExp(new OrExp(
        new BoolLit(false), new BoolLit(true))));
NegationVisitor nv = new NegationVisitor();
exp.accept(nv);
exp = nv.getResult();
```