Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2012/2013

19 giugno 2013

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

```
Pattern p = Pattern.compile("\"(\\\\[btfnr]|[^\\\\\"])*\"");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

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

ii. `assert p.matcher("\"\"").matches();`

iii. `assert p.matcher("\"abc\"").matches();`

iv. `assert p.matcher("\"01\"\"").matches();`

v. `assert p.matcher("\"10\\h\"").matches();`

vi. `assert p.matcher("\"10\\n\"").matches();`

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= 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` **rimanga invariato**.

2. Considerare la funzione `no_seq_dup : 'a list -> 'a list` così definita: `no_seq_dup` l restituisce la lista ottenuta da l eliminando tutte le ripetizioni consecutive di un elemento, lasciando inalterata la posizione relativa degli altri elementi.

Esempio:

```
# no_seq_dup [1;1;1;2;3;3;2;1;1;2;4];;
- : int list = [1; 2; 3; 2; 1; 2; 4]
```

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

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

(c) Definire la funzione `no_seq_dup` 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 seguenti dichiarazioni di classi Java:

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

```
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) `h.m(1)`

(c) `h.m((Integer) 1)`

(d) `h.m((Number) 3.2)`

(e) `h.m((Double) 3)`

(f) `h.m(1,2)`

4. Considerare i package `ast` e `visitor` che implementano abstract syntax tree e visite su di essi per espressioni aritmetiche formate a partire da variabili, literal interi e gli operatori binari di addizione e moltiplicazione.

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

```
package ast;
public interface Variable extends Exp {
        String getName();
}
```

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

```
package ast;
import visitor.Visitor;
public class IdentExp extends AbsExp implements Variable {
        private final String name;
        public IdentExp(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
        public void accept(Visitor v) {
                // completare
        }
}
```

```
package ast;
import visitor.Visitor;
public class NumLit extends AbsExp {
        final private int value;
        public NumLit(int value) {
                this.value = value;
        }
        public int getValue() {
                return value;
        }
        public void accept(Visitor v) {
                // completare
        }
}
```

```
package ast;
import visitor.Visitor;
public class AddExp extends AbsExp {
        public AddExp(Exp exp1, Exp exp2) {
                // completare
        }
        public void accept(Visitor v) {
                // completare
        }
}
```

```
package ast;
import visitor.Visitor;
public class MulExp extends AbsExp {
        public MulExp(Exp exp1, Exp exp2) {
                // completare
        }
        public void accept(Visitor v) {
                // completare
        }
}
```

(a) Completare le definizioni delle classi `AbsExp`, `IdentExp`, `NumLit`, `AddExp` e `MulExp`.

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

```
package visitor;
import ast.*;
public interface Visitor {
        void visit(IdentExp e);
        void visit(NumLit e);
        void visit(AddExp e);
        void visit(MulExp e);
}
```

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

```
package visitor;
import java.util.HashSet;
public class FreeVarVisitor extends AbstractVisitor<Set<String>> {
        public FreeVarVisitor() {
                result = new HashSet<>();
        }
        @Override
        public void visit(IdentExp exp) {
                // completare
        }
        @Override
        public void visit(NumLit exp) {
                // completare
        }
        @Override
        public void visit(AddExp exp) {
                // completare
        }
        @Override
        public void visit(MulExp exp) {
                // completare
        }
}
```

```
package visitor;
import java.util.List;
public class SimplifyVisitor extends AbstractVisitor<Exp> {
        private boolean isLit(Exp exp, int val) {
                return exp instanceof NumLit && ((NumLit) exp).getValue() == val;
        }
        @Override
        public void visit(IdentExp exp) {
                // completare
        }
        @Override
        public void visit(NumLit exp) {
                // completare
        }
        @Override
        public void visit(AddExp exp) {
                // completare
        }
        @Override
        public void visit(MulExp exp) {
                // completare
        }

}
```

- la classe `FreeVarVisitor` calcola l'insieme dei nomi di tutte le variabili contenute nell'espressione. Esempio:

```
Exp exp = new AddExp(new NumLit(0), new MulExp(new AddExp(new IdentExp(
                "x"), new IdentExp("y")), new AddExp(new IdentExp("y"),
                new IdentExp("z"))));
FreeVarVisitor fvv = new FreeVarVisitor();
exp.accept(fvv);
Set<String> names = fvv.getResult();
assert names.contains("x");
assert names.contains("y");
assert names.contains("z");
assert names.size() == 3;
```

- la classe `SimplifyVisitor` semplifica l'espressione applicando le identità

$$\begin{aligned} e+0 &= e = 0+e \\ e\cdot 1 &= e = 1\cdot e \end{aligned}$$

Esempio:

```
SimplifyVisitor sv = new SimplifyVisitor();
Exp exp = new MulExp(new AddExp(new NumLit(1), new NumLit(0)), new MulExp(
                new IdentExp("x"), new NumLit(1)));
exp.accept(sv);
exp = sv.getResult();
assert exp instanceof IdentExp;
assert ((IdentExp) exp).getName().equals("x");
```