Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2012/2013

25 gennaio 2013

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

```
Pattern p = Pattern.compile("([1-9][0-9]*|0)(\\.([0-9]*[1-9])?)?");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

i. `assert p.matcher("13.0.9").matches();`

ii. `assert p.matcher("03.").matches();`

iii. `assert p.matcher("150.0").matches();`

iv. `assert p.matcher("00").matches();`

v. `assert p.matcher("3.").matches();`

vi. `assert p.matcher(".30").matches();`

(b) Mostrare che la seguente grammatica è ambigua.

```
Dec  ::= Type Id ; | Dec Dec
Id ::= x | y | z
Type ::= int | bool
```

(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Dec` **rimanga lo stesso**.

2. Considerare la funzione `add_after_if : ('a -> bool) -> 'a -> 'a list -> 'a list` tale che `add_after_if` p e l restituisce la lista ottenuta da l aggiungendo e immediatamente dopo ogni elemento x di l per cui $p(x)$ è vero.

Esempio:

```
# add_after_if (fun x -> x > 0) 42 [-1;0;1;2];;
- : int list = [-1; 0; 1; 42; 2; 42]
```

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

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

(c) Definire la funzione `add_after_if` 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(Object o) {
        return "P.m(Object)";
    }
    public String m(int i) {
        return "P.m(int)";
    }
}
public class H extends P {
    public String m(Object o) {
        return "H.m(Object)";
    }
    public String m(int i) {
        return super.m(i) + '\n' + "H.m(int)";
    }
    public String m(Number... n) {
        return super.m(n) + '\n' + "H.m(Number...)";
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        P p1 = 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.

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) `p1.m(1)`

(b) `h.m(1)`

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

(d) `h.m(3.2)`

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

(f) `((H) p1).m(1,2)`

4. Considerare i package `ast` e `visitor` per rappresentare tramite abstract syntax tree espressioni e implementare visite sui corrispondenti abstract syntax tree; le espressioni considerate sono costruite a partire da:

   - literal di tipo `List<Integer>`;
   - identificatori di tipo `List<Integer>`;
   - un operatore unario che presa una lista l restituisce una nuova lista ottenuta rovesciando l;
   - un operatore binario che prese due liste l_1 ed l_2 restituisce una nuova lista ottenuta concatenando l_1 ed l_2.

```
package visitor;
import ast.*;
public interface Visitor {
    void visit(AppendExp e);
    void visit(IdentExp e);
    void visit(ReverseExp e);
    void visit(ListLit e);
}
-----------------------------------------------------------------
package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}
-----------------------------------------------------------------
package ast;
import visitor.Visitor;
public interface Exp {
    Exp[] getChildren();
    void accept(Visitor v);
}
-----------------------------------------------------------------
package ast;
public abstract class AbsExp implements Exp {
    private final Exp[] children;
    protected AbsExp(Exp... children) {
        this.children = children;
    }
    @Override
    public Exp[] getChildren() {
        return children;
    }
}
-----------------------------------------------------------------
package ast;
public interface Ident extends Exp {
    String getName();
}
-----------------------------------------------------------------
package ast;
import visitor.Visitor;
public class IdentExp extends AbsExp implements Ident {
    private final String name;
    public IdentExp(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}
-----------------------------------------------------------------
package ast;
import java.util.List;
import visitor.Visitor;
public class ListLit extends AbsExp {
    protected final List<Integer> value;
    public ListLit(List<Integer> value) { /* completare */ }
    public List<Integer> getValue() { /* completare */ }
    public void accept(Visitor v) { /* completare */ }
}
-----------------------------------------------------------------
package ast;
import visitor.Visitor;
public class ReverseExp extends AbsExp {
    public ReverseExp(Exp exp) { /* completare */ }
    public void accept(Visitor v) { /* completare */ }
-----------------------------------------------------------------
package ast;
import visitor.Visitor;
public class AppendExp extends AbsExp {
    public AppendExp(Exp leftExp, Exp rightExp) { /* completare */ }
    public void accept(Visitor v) { /* completare */ }
}
```

(a) Completare le definizioni delle classi `ListLit`, `ReverseExp` e `AppendExp`.

(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione di `EvaluationVisitor` che permette di valutare un'espressione a partire da un ambiente dinamico `DynamicEnv`. La visita deve sollevare un'eccezione di tipo `RuntimeException` se l'espressione contiene una variabile non definita nell'ambiente dinamico.

Il metodo `read` di `DynamicEnv` solleva un'eccezione di tipo `RuntimeException` se l'identificatore non è definito.

```
package visitor;
import java.util.List;
import ast.Ident;
public interface DynamicEnv {
    List<Integer> read(Ident id);
}
-----------------------------------------------------------------
package visitor;
import java.util.List;
import java.util.ArrayList;
import static java.util.Collections.reverse;
import test.ast.*;

public class EvaluationVisitor extends AbstractVisitor<List<Integer>> {
    // completare
}
```

(c) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione di `SubstitutionVisitor` che permette di produrre una nuova espressione ottenuta a partire da quella visitata applicando una sostituzione. Una sostituzione è una funzione da identificatori a espressioni.

Il metodo `read` restituisce sempre una nuova copia dell'espressione associata all'identificatore `id`.

```
package visitor;
import test.ast.*;
public interface Substitution {
    Exp read(Ident id);
}
-----------------------------------------------------------------
package visitor;
import ast.*;
public class SubstitutionVisitor extends AbstractVisitor<Exp> {
    // completare
}
```

Per esempio, se la sostituzione associa all'identificatore `"x"` l'espressione `new ListLit(Arrays.asList(5,6,7))`, allora la visita dell'albero corrispondente all'espressione

```
new ReverseExp(new AppendExp(new IdentExp("x"),new IdentExp("x")))
```

restituisce un nuovo albero corrispondente all'espressione

```
new ReverseExp(new AppendExp(new ListLit(Arrays.asList(5,6,7)),new ListLit(Arrays.asList(5,6,7))))
```