Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2011/2012

20 giugno 2012

1. Sia $\mathcal{L}$ il linguaggio generato dalla seguente grammatica a partire dal simbolo non terminale `Rat`.

```
Rat  ::= Sign Int Frac
Sign ::= - | ϵ
Int  ::= Dg | Int Dg
Dg   ::= 0 | ... | 9
Frac ::= ϵ | . | . Int
```

   (a) Definire una grammatica regolare destra che generi il linguaggio $\mathcal{L}$ *(una grammatica* (T, N, P) *è detta* regolare destra *se ogni produzione in* P *è della forma* S `::=` u *oppure* S `::=` u R *oppure* S `::=` ϵ*, dove* $S, R \in N$*,* $u \in T$*).*

   (b) Completare la seguente espressione Java in modo che si valuti in un oggetto di `java.util.regex.Pattern` che riconosca il linguaggio $\mathcal{L}$.

```
java.util.regex.Pattern.compile("...");
```

2. Sia `intexp` il seguente tipo user-defined

```
type intexp = Num of int | Var of string | Add of intexp * intexp | Mul of intexp * intexp
```

corrispondente alle espressioni di tipo `int` costruite a partire da costanti numeriche (`Num`), variabili (`Var`), e dagli operatori di addizione (`Add`) e moltiplicazione (`Mul`).

   (a) Definire in modo diretto la funzione `eval : (string -> int) -> intexp -> int` che valuta un'espressione, rispetto a una funzione `get_val : string -> int` che restituisce il valore associato a ogni variabile.

   Per esempio, se `get_val "x"= 3`, `get_val "y"= 4`, allora
   `eval get_val (Mul(Add(Var "x",Var "y"),Num 5))` si valuta in 35.

   (b) Definire in modo diretto la funzione `applysub : (string -> intexp) -> intexp -> intexp` che data un'espressione `e` e una funzione `sub : string -> intexp`, restituisce l'espressione intera ottenuta sostituendo ogni variabile `Var s` che compare in `e` con l'espressione intera `sub s`.

   Per esempio, se `sub "x"= Add(Var "x",Num 1)` e `sub "y"= Num 2`, allora
   `applysub sub (Mul(Var "x",Mul(Var "y",Var "y")))` si valuta in
   `Mul (Add (Var "x", Num 1), Mul (Num 2, Num 2))`.

   (c) Sia

```
morph : (int -> 'a) -> (string -> 'a) -> ('a * 'a -> 'a) -> ('a * 'a -> 'a) -> intexp -> 'a
```

   la funzione definita nel seguente modo:

```
let rec morph num var add mul = function
     Num n -> num n
   | Var s -> var s
   | Add(e1,e2) -> add(morph num var add mul e1,morph num var add mul e2)
   | Mul(e1,e2) -> mul(morph num var add mul e1,morph num var add mul e2);;
```

   Definire le funzioni `eval` e `applysub` specificate nei punti precedenti, come opportune specializzazioni della funzione `morph`.

3. Considerare le seguenti classi e interfacce:

   - `Exp`, `AbstractExp`, `NumLit`, `AddExp`, `MulExp`: classi e interfacce che definiscono l'abstract syntax tree di un'espressione di tipo **`int`**, costruita a partire da costanti numeriche (`NumLit`), e dagli operatori di addizione (`AddExp`) e moltiplicazione (`MulExp`).
   - `Visitor`, `AbstractVisitor`: interfaccia e classe astratta che definiscono un generico visitor di `Exp`.
   - `EvalVisitor`: classe che implementa un visitor che restituisce come risultato il valore in cui si valuta l'espressione visitata.
   - `SwapVisitor`: classe che implementa un visitor che restituisce come risultato una nuova espressione ottenuta da quella visitata sostituendo gli operatori di addizione con quelli di moltiplicazione e viceversa.

```
public interface Exp {
    void accept(Visitor v);
}
public abstract class AbstractExp implements Exp {
    private final Exp[] children;
    public AbstractExp(Exp... children) {
        this.children = children;
    }
    public Exp[] getChildren() {
        return children;
    }
}
public final class NumLit extends AbstractExp {
    private final int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}
public final class AddExp extends AbstractExp {
    AddExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}
public final class MulExp extends AbstractExp {
    MulExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}
public interface Visitor {
    // completare
}
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}
public class EvalVisitor extends AbstractVisitor<Integer> {
    // completare
}
public class SwapVisitor extends AbstractVisitor<Exp> {
    // completare
}
```

   (a) Completare i metodi `accept` delle classi `NumLit`, `AddExp` e `MulExp`.

   (b) Completare la definizione dell'interfaccia `Visitor`.

   (c) Completare la definizione della classe `EvalVisitor`.

   (d) Completare la definizione della classe `SwapVisitor`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
package pck1;
public class C1 {
    private void m() {
        System.out.println("C1.m");
    }
    public void q() {
        System.out.println("C1.q");
        m();
    }
}
-----------------------------------------
package pck1;
public class C2 extends C1 {
    void m() {
        System.out.println("C2.m");
    }
}
-----------------------------------------
package pck2;
public class C3 extends pck1.C2 {
    protected void m(Object... objects) {
        System.out.println("C3.m");
    }
}
-----------------------------------------
package pck1;
public class C4 extends pck2.C3 {
    public void m() {
        super.m();
        System.out.println("C4.m");
    }
    public void q() {
        System.out.println("C4.q");
        super.q();
    }
}
-----------------------------------------
package pck2;
import pck1.*;
public class Main {
    public static void main(String[] args) {
        C1 c1 = new C1();
        C2 c2 = new C2();
        C3 c3 = new C3();
        C4 c4 = new C4();
        ...
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Main` l'espressione indicata.

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) `c2.m();`

(b) `c3.m();`

(c) `c4.m();`

(d) `((C3) c4).m();`

(e) `c1.q();`

(f) `c4.q();`