Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2013/2014

luglio? settembre? 2014

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

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

Indicare quali delle seguenti asserzioni hanno successo e quali falliscono, motivando la risposta.

i. `assert p.matcher("42.0.42").matches();`
ii. `assert p.matcher(".").matches();`
iii. `assert p.matcher("420.042").matches();`
iv. `assert p.matcher("0042").matches();`
v. `assert p.matcher("42.").matches();`
vi. `assert p.matcher(".42").matches();`

(b) Mostrare che la seguente grammatica è ambigua.

```
Const ::= Id = Exp ; | Const Const
Id ::= x | y | z
Exp ::= false | true | Exp && false | Exp && true
```

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

2. Considerare la funzione `compose : 'a -> ('a -> 'a) list -> 'a` tale che

`compose` x `[`f_1`; ...;` f_n`]` $= f_1(\ldots f_n(x)\ldots)$.

Esempi:

```
# compose 2 [(fun x -> x*2);(fun x -> x+1)];;
- : int = 6
# compose 2 [(fun x -> x+1);(fun x -> x*2)];;
- : int = 5
```

Nei seguenti esercizi è ammesso utilizzare la funzione predefinita `List.rev`.

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

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

(c) Definire la funzione `compose` 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 S {
    public String f(Object o) {
        return "S.f(Object)";
    }
    public String f(double i) {
        return "S.f(double)";
    }
}
public class C extends S {
    public String f(Object o) {
        return "C.f(Object)";
    }
    public String f(double d) {
        return super.f(d) + " C.f(double)";
    }
    public String f(Number... n) {
        return super.f(n) + " C.f(Number...)";
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        S s1 = new S();
        C c = new C();
        S s2 = c;
        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) `s1.f(42)`

(b) `s2.f(42)`

(c) `c.f((Double) 42.)`

(d) `c.f(4.2)`

(e) `c.f(4, 2)`

(f) `((C) s1).f(4, 2)`

4. Considerare le due interfacce generiche `Visitor` e `BinTree`.

```
public interface Visitor<E, R> {
    // E type of node labels, R type of the result of the visit

    public R visitLeaf(E e);

    public R visitBranch(E e, BinTree<E> left, BinTree<E> right);

}

public interface BinTree<E> {
    <R> R accept(Visitor<E, R> v);
}
```

(a) Completare la definizione della classe `BinTreeFactory`.

```
public class BinTreeFactory {
    private BinTreeFactory() {
    }

    public static <T> BinTree<T> leaf(final T e) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                        // DA COMPLETARE
            }
        };
    }

    public static <T> BinTree<T> branch(final T e,final BinTree<T> l, final BinTree<T> r) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                        // DA COMPLETARE
            }
        };
    }
}
```

(b) Completare la classe `Depth` che calcola la profondità di un albero (ossia, la lunghezza di uno dei cammini massimi dalla radice a una foglia).

```
public class Depth<E> implements Visitor<E, Integer> {

    @Override
    public Integer visitLeaf(E elt) {
        // DA COMPLETARE
    }

    @Override
    public Integer visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        // DA COMPLETARE
    }

}
```

(c) Completare la classe `InOrder` che calcola la stringa (tramite un oggetto di tipo `StringBuilder`) corrispondente alla visita in-order di un albero.

```
public class InOrder<E> implements Visitor<E, StringBuilder> {
    final private StringBuilder stb = new StringBuilder();

    @Override
    public StringBuilder visitLeaf(E e) {
        // DA COMPLETARE
    }

    @Override
    public StringBuilder visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        // DA COMPLETARE
    }
}
```