Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

9 settembre 2019, a.a. 2018/2019

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali falliscono, motivando la risposta.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherTest {
    public static void main(String[] args) {
        Pattern regEx =
            Pattern.compile("(\\s+)|([0-3]?[0-9]/[0-1]?[0-9]/[0-9][0-9])|(0[xX]([A-Fa-f0-9]+))");
        Matcher m = regEx.matcher("9/9/19 0XFF");
        m.lookingAt();
        assert m.group(2).equals("09/09/19");
        assert m.group(0).equals("9/9/19");
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(2) != null;
        assert m.group(0).equals("0XFF");
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group(3).equals("0XFF");
        assert m.group(4).equals("FF");
    }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp * Atom | Exp + Atom | Atom
Atom ::=  [ ExpSeq ] | Id
ExpSeq ::= ExpSeq Exp | Exp
Id ::= a | b
```

   (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` **resti invariato**.

2. Sia `count_zeros : ('a -> int) -> 'a list -> int` la funzione così specificata:

   `count_zeros` f l restituisce il numero di elementi e della lista l per i quali vale l'uguaglianza $f(e) = 0$.

   Esempi:

```
# count_zeros (fun x->(x-1)*(x-2)*(x+3)) [-3;1;2;0;4]
- : int = 3
# count_zeros (fun x->(x-1)*(x-2)*(x+3)) [-1;0;4]
- : int = 0
```

   (a) Definire `count_zeros` senza uso di parametri di accumulazione.

   (b) Definire `count_zeros` usando un parametro di accumulazione affinché la ricorsione sia di coda.

   (c) Definire `count_zeros` come specializzazione della funzione `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

3. Sia dato il seguente programma per la valutazione di espressioni formate da literal di tipo intero e tipo stringa, e dall'operatore binario di prodotto tra una stringa e un intero:

```
public interface Exp { <T> T accept(Visitor<T> visitor); }
public interface Visitor<T> {...}
public interface Value {...}
public abstract class PrimVal<T> implements Value {...}
public class IntVal extends PrimVal<Integer> {...}
public class StringVal extends PrimVal<String> {...}
public class Eval implements Visitor<Value> {...}
```

(a) Completare le seguenti classi che implementano i valori di tipo intero e stringa.

```
public interface Value {
    public default String asString() {throw new RuntimeException("Expecting a string");}
    public default int asInt() {throw new RuntimeException("Expecting an integer");}
}
// valori primitivi generici
public abstract class PrimVal<T> implements Value {
    protected T val;
    // invariante di classe: val!=null
    protected PrimVal(T val) { /* completare */ }
}
public class IntVal extends PrimVal<Integer> {
    protected IntVal(Integer val) { /* completare */ }
    @Override public int asInt() { /* completare */ }
}
public class StringVal extends PrimVal<String> {
    protected StringVal(String val) { /* completare */ }
    @Override public String asString() { /* completare */ }
}
```

(b) Completare la classe `Eval` per la valutazione delle espressioni. Nel metodo `visitTimes(Exp left, Exp right)` (prodotto tra stringhe e interi) l'operando `left` si deve valutare in una stringa s e `right` in un intero i; il risultato calcolato è la concatenazione di s ripetuta i volte. Viene sollevata `RuntimeException` se i valori degli operandi non sono del tipo giusto e `IllegalArgumentException` se $i < 0$.

```
public class Eval implements Visitor<Value> {
    @Override public Value visitIntLit(int val) { /* completare */ }
    @Override public Value visitStringLit(String val) { /* completare */ }
    @Override public Value visitTimes(Exp left, Exp right) { /* completare */ }
}
```

Suggerimento: per il calcolo della concatenazione, utilizzare il seguente metodo della classe predefinita `String`:

```
// Returns a string whose value is the concatenation of this string repeated count times.
public String repeat(int count)
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number... o) { return "P.m(Number...)"; }
    String m(Object... o) { return "P.m(Object...)"; }
}
public class H extends P {
    String m(String s) { return super.m(s) + " H.m(String)"; }
    String m(Double d1, Double d2) { return super.m(d1, d2) + " H.m(Double,Double)";    }
}
public class Test {
    public static void main(String[] args) {
        P p = 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, assumendo che tutte le classi siano dichiarate nello stesso package.

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("42")`

(b) `p2.m("42")`

(c) `h.m("42")`

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`