Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 5 giugno

a.a. 2014/2015

9 giugno 2015

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

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class MatcherTest {
5      public static void main(String[] args) {
6          Pattern regEx = Pattern
7              .compile("(for|while|(?<FIRST>[a-zA-Z@]))(?<REST>[a-zA-Z0-9]*)|
8                  (?<NUM>[0-9]+\\.?[0-9]*)|(?<SKIP>\\s+)");
9          Matcher m = regEx.matcher("@for 42.");
10         assert m.lookingAt();
11         assert m.group("REST").equals("for");
12         assert m.group("FIRST").equals("@");
13         m.region(m.end(), m.regionEnd());
14         m.lookingAt();
15         assert m.group("SKIP") != null;
16         m.region(m.end(), m.regionEnd());
17         m.lookingAt();
18         assert m.group("NUM").equals("42.0");
19         assert Double.parseDouble(m.group("NUM")) == 42.0;
20     }
21 }
```

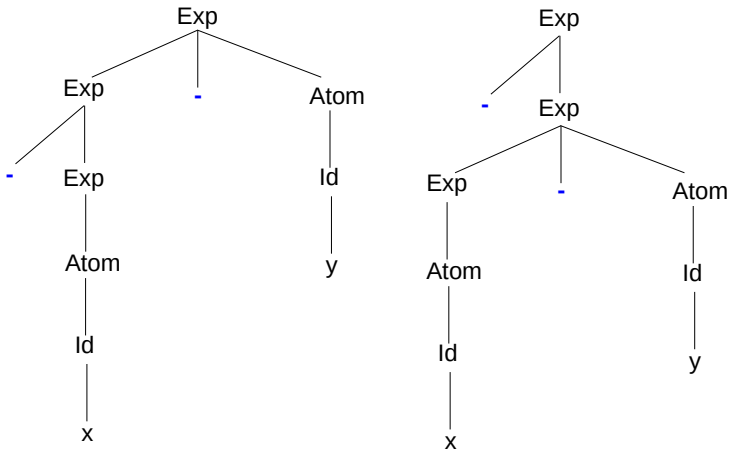
Soluzione:

- `assert m.lookingAt();` (linea 10): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `"@for 42."` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `"@for"`, ottenuta come concatenazione di `"@"` (gruppo `FIRST`) e `"for"` (gruppo `REST`), quindi l'asserzione ha successo;
- `assert m.group("REST").equals("for");` (linea 11): `m.group("REST")` restituisce la stringa `"for"` per i motivi indicati al punto precedente, per cui l'asserzione ha successo;
- `assert m.group("FIRST").equals("@");` (linea 12): `m.group("FIRST")` restituisce la stringa `"@"` per i motivi indicati al punto 1, per cui l'asserzione ha successo;
- `assert m.group("SKIP") != null;` (linea 15): alla linea 13 l'inizio della regione viene spostato al carattere immediatamente successivo a `"@for"` che è uno spazio bianco e appartiene al gruppo `SKIP`, quindi `m.group("SKIP")` restituisce la stringa `" "` e l'asserzione ha successo;
- `assert m.group("NUM").equals("42.0");` (linea 18): alla linea 16 l'inizio della regione viene spostato al carattere immediatamente successivo allo spazio bianco (carattere `0`), l'invocazione `m.lookingAt()` alla linea 17 ha successo poiché la stringa `"42."` appartiene al gruppo `NUM`, quindi `m.group("NUM")` restituisce la stringa `"42."` e l'asserzione fallisce;
- `assert Double.parseDouble(m.group("NUM")) == 42.0;` (linea 19): `m.group("NUM")` restituisce la stringa `"42."` appena riconosciuta, per cui l'asserzione ha successo, visto che la stringa rappresenta il valore di tipo `double` `42.0`.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  - Exp | Exp - Atom | Atom
Atom  ::= Id | ( Exp )
Id   ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `- x - y`



(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**.

Soluzione: La soluzione più semplice consiste nell' aggiungere il non terminale `Minus` per dare precedenza all'operatore unario −.

```
Exp   ::=  Exp - Atom | Minus
Minus ::= - Minus | Atom
Atom  ::= Id | ( Exp )
Id    ::= x | y | z
```

La grammatica può essere semplificata nella seguente equivalente:

```
Exp   ::=  Exp - Atom | Atom
Atom  ::= Id | - Atom | ( Exp )
Id    ::= x | y | z
```

2. Considerare la funzione `eval : int -> int list -> int` che valuta un polinomio a una variabile e a coefficienti interi rappresentato dalla lista dei suoi coefficienti in ordine crescente di grado.

   Esempio:

```
# eval 4 [1;0;1];; (* valuta 1+x^2 per x=4 *)
- : int = 17
# eval 5 [1;-4;4];; (* valuta 1-4x+4x^2 per x=5 *)
- : int = 81
```

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

   (b) Definire la funzione `eval` direttamente, usando un parametro di accumulazione affinché la ricorsione sia di coda. Per rovesciare la lista utilizzare la funzione di libreria `List.rev`.

   (c) Definire la funzione `eval` 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>
```

   Per rovesciare la lista utilizzare la funzione di libreria `List.rev`.

Soluzione: Vedere il file `soluzione.ml`.

3. (a) Completare la definizione della classe `ArrayPoly` che implementa i polinomi a una variabile rappresentandoli con l'array dei loro coefficienti in ordine decrescente di grado. Per esempio, **new** `ArrayPoly(1.,0.5,-2.,0.)` rappresenta il polinomio $x^3 + \frac{1}{2}x^2 - 2x$, mentre **new** `ArrayPoly(-2.,0.,0.)` rappresenta $-2x^2$.

```
public interface Polynomial extends Iterable<Double> {
    Iterator<Double> revIterator();
    int degree();
    double eval(double val);
    Polynomial add(Polynomial other);
}
public abstract class AbstractPoly implements Polynomial {
    @Override
    public double eval(double val) {
        ...
    }
    @Override
    public Polynomial add(Polynomial other) {
        ...
    }
}
public class ArrayPoly extends AbstractPoly {
    private final double[] coeffs;
    /** new ArrayPoly() deve essere equivalente a new ArrayPoly(0.) */
    public ArrayPoly(double... coeffs) {
        // da completare
    }
    /** itera sui coefficienti a partire da quello di grado massimo */
    public Iterator<Double> iterator() {
        return new PolyIterator(coeffs);
    }
    /** itera sui coefficienti a partire da quello di grado 0 */
    public Iterator<Double> revIterator() {
        return new RevPolyIterator(coeffs);
    }
    /** restituisce il grado del polinomio */
    @Override
    public int degree() {
        // da completare
    }
}
```

(b) Completare i metodi `eval` e `add` della classe astratta `AbstractPoly`. Il metodo `eval` valuta il polinomio sostituendo all'incognita il valore `val`.

Per esempio, `assert` **new** `ArrayPoly(4., -4., 1.).eval(3.) == 25.` ha successo.

Il metodo `add` restituisce il polinomio ottenuto sommando il polinomio **this** con il polinomio `other`. Per esempio, **new** `ArrayPoly(4., -4., 1.).add(`**new** `ArrayPoly(1.,0.,0.,-1.))` deve restituire un'instanza di `ArrayPoly` che rappresenta il polinomio $x^3 + 4x^2 - 4x$.

```
public abstract class AbstractPoly implements Polynomial {
    @Override
    public double eval(double val) {
        // da completare
    }
    @Override
    public Polynomial add(Polynomial other) {
        // da completare
    }
}
```

(c) Completare la classe `PolyIterator` che itera su un array di **double**.

```
class PolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    PolyIterator(double[] coeffs) {
        if (coeffs == null)
            throw new NullPointerException();
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Double next() {
        // da completare
    }
}
```

(d) Completare la classe `RevPolyIterator` che itera su un array di **`double`** in ordine inverso.

```
class RevPolyIterator implements Iterator<Double> {
    private int index;
    private final double[] coeffs;
    RevPolyIterator(double[] coeffs) {
        index = coeffs.length - 1;
        this.coeffs = coeffs;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Double next() {
        // da completare
    }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

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

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(Integer.valueOf(42))`

(e) `p2.m(new double[]{4.2})`

(f) `h.m(new double[]{4.2})`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42.` ha tipo statico **`double`**; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(double d)` in `P`.

Viene stampata la stringa `"P.m(double)"`.

(b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`. Poiché `d` ha tipo statico **`double`**, per l'invocazione `super.m(d)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.

Viene stampata la stringa `"P.m(double) H.m(double)"`.

(c) Il literal `42.` ha tipo statico **`double`**; il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(double d)`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`.

Per gli stessi motivi del punto precedente, viene stampata la stringa `"P.m(double) H.m(double)"`.

(d) Il literal `42` ha tipo statico **`int`** e per l'invocazione `Integer.valueOf(42)` esiste un solo metodo statico nella classe `Integer` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Integer`. Il tipo statico di `p` è `P`, non esistono metodi in `P` accessibile e applicabile per sotto-tipo, mentre se l'argomento viene convertito in un valore di tipo **`int`** tramite unboxing i due metodi di `P String m(double d)` e `String m(int i)` sono entrambi accessibili e applicabili, ma il secondo è più specifico.

Viene stampata la stringa `"P.m(int)"`.

(e) L'espressione **`new double`**`[]{4.2}` ha tipo statico **`double`**`[]`, il tipo statico di `p2` è `P` e non esiste alcun metodo di `P` accessibile e applicabile, quindi viene segnalato un errore durante la compilazione.

(f) L'espressione **`new double`**`[]{4.2}` ha tipo statico **`double`**`[]`, il tipo statico di `h` è `H` e l'unico metodo di `H` che è accessibile e applicabile è `String m(Object... os)`, poiché nessun metodo è applicabile per sotto-tipo o per boxing/unboxing, mentre `Object...` corrisponde a `Object` e **`double`**`[]`$\leq$`Object`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Object... os)` in `H`. La conversione da **`int`** a **`double`** è senza perdita di informazione.

Viene stampata la stringa `"H.m(Object...)"`.