Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 15 febbraio 2017

a.a. 2016/2017

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.compile("([a-zA-Z][a-zA-Z0-9]*)|([0-9]+([eE][0-9]+)?)|(\\s+)");
7        Matcher m = regEx.matcher("x1E0 42e04");
8        m.lookingAt();
9        assert m.group(1).equals("x");
10       assert m.group(0).equals("x1E0");
11       m.region(m.end(), m.regionEnd());
12       m.lookingAt();
13       assert m.group(4) != null;
14       m.region(m.end(), m.regionEnd());
15       m.lookingAt();
16       assert m.group(2).equals("42");
17       assert m.group(3).equals("04");
18       assert m.group(0).equals("42e04");
19    }
20 }
```

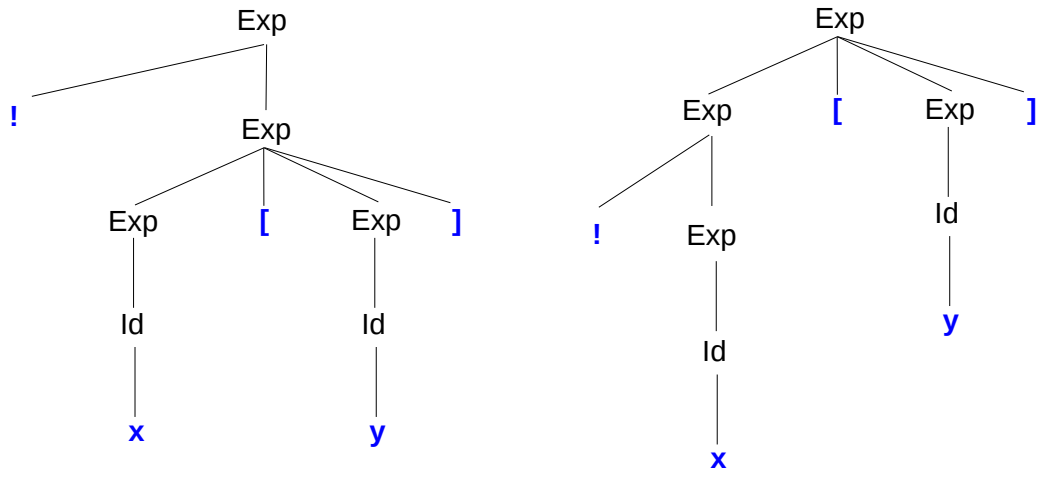
Soluzione:

- `assert m.group(1).equals("x");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `x1E0 42e04` 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 `x1E0` (stringa appartenente al gruppo di indice 1), quindi l'asserzione fallisce;
- `assert m.group(0).equals("x1E0");` (linea 10): lo stato del matcher non è cambiato rispetto alla linea precedente, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(4) != null;` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `x1E0` (ossia lo spazio bianco) e l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene all'espressione regolare (gruppo 4), quindi l'asserzione ha successo;
- `assert m.group(2).equals("42");` (linea 16): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo alla successione di spazi bianchi (ossia `4`) e l'invocazione del metodo `lookingAt()` ha successo poiché `42e04` appartiene all'espressione regolare (gruppo 2, la sottostringa `e04` appartiene al gruppo 3), quindi l'asserzione fallisce;
- `assert m.group(3).equals("04");` (linea 17): per i motivi del punto precedente l'asserzione fallisce;
- `assert m.group(0).equals("42e04");` (linea 18): il gruppo 0 corrisponde all'intera espressione regolare, quindi per i motivi del punto precedente l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ! Exp | Exp [ Exp ] | ( Exp ) | Id
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: Una possibile soluzione consiste nell'aggiunta del non-terminale `Bang` per poter attribuire precedenza all'operatore unario `!`.

```
Exp  ::= Exp [ Exp ] | Bang
Bang ::= ! Bang | ( Exp ) | Id
Id   ::= x | y | z
```

2. Considerare la funzione `replace : 'a -> 'a -> 'a list -> 'a list` tale che `replace` x y l sostituisce nella lista l tutte le occorrenze di x con y, lasciando gli altri elementi invariati. Esempio:

```
# replace 'L' 'l' ['H';'e';'L';'L';'o'];;
- : char list = ['H'; 'e'; 'l'; 'l'; 'o']
```

(a) Definire la funzione `replace` senza uso di parametri di accumulazione.

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare la seguente implementazione di successioni finite di interi tale che **new** `IntSeq(`*min*`, `*max*`, `*step*`)` rappresenta l'insieme di interi $\{min + k{\cdot}step \mid k \geq 0$ e $min + k \cdot step \leq max\}$.

   Per esempio, **new** `IntSeq(1, 10, 4)` corrisponde all'insieme $\{1, 5, 9\}$.

```
public class IntSeq implements Iterable<Integer> {
    /* implements the set { min+k*step | k >= 0 and min+k*step <= max } */
    private final int min;
    private final int max;
    private final int step; // invariant: step > 0

    public IntSeq(int min, int max) { // default step is 1
        ...
    }
    public IntSeq(int min, int max, int step) {
        ...
    }
    public int getMin() {
        ...
    }
    public int getMax() {
        ...
    }
    public int getStep() {
        ...
    }
    public Iterator<Integer> iterator() {
        ...
    }
}
class IntSeqIterator implements Iterator<Integer> {
    private int min;
    private final int max;
    private final int step;

    public IntSeqIterator(IntSeq seq) {
        ...
    }
    public boolean hasNext() {
        ...
    }
    public Integer next() {
        ...
    }
}
```

   (a) Completare le definizioni dei costruttori e dei metodi *getter* della classe `IntSeq`.

   (b) Completare la definizione del metodo `iterator()` della classe `IntSeq`.

   (c) Completare la definizione del costruttore della classe `IntSeqIterator`.

   (d) Completare le definizioni dei metodi `hasNext()` e `next()` della classe `IntSeqIterator`.

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(long l) {
        return "P.m(long)";
    }
    String m(double d) {
        return "P.m(double)";
    }
    String m(Object... os) {
        return "P.m(Object...)";
    }
}
public class H extends P {
    String m(long l) {
        return super.m((double) l) + " H.m(long)";
    }
    String m(Double d) {
        return super.m(d, d + 1) + " H.m(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(42L)`

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

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

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(Double.valueOf(42.0))`

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

(a) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `p` è `P`, quindi esistono due metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(`**`long`**`)` è più specifico del metodo con segnatura `m(`**`double`**`)` poiché **`long`** $\leq$ **`double`**.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(`**`long`**`)` in `P`.

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

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al 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 con segnatura `m(`**`long`**`)` ridefinito in `H`. L'espressione `(`**`double`**`) l` è staticamente corretta poiché i cast tra tipi numerici sono sempre ammessi; dato che il tipo statico dell'argomento è **`double`**, per l'invocazione **`super`**`.m((`**`double`**`) l)` esiste in `P` un solo metodo accessibile e applicabile per sottotipo, quello con segnatura `m(`**`double`**`)`.

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

(c) L'espressione `42L` ha tipo statico **`long`**, mentre il tipo statico di `h` è `H`; poiché il metodo di `H` con segnatura `m(Double d)` non è applicabile per sottotipo, l'invocazione viene risolta come al punto precedente. Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa `"P.m(double) H.m(long)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`, quindi il solo metodo accessibile in `P` e applicabile per sottotipo ha segnatura `m(`**`double`**`)`.

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

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

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al 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 di `P` con segnatura `m(`**`double`**`)`, visto che la sottoclasse `H` non ridefinisce il metodo, ma lo eredita da `P`.

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

(f) Il literal `42.0` ha tipo statico **`double`** e l'unico metodo della classe `Double` accessibile e applicabile per sottotipo è il metodo statico con segnatura `valueOf(`**`double`**`)` e tipo di ritorno `Double`. Il tipo statico di `h` è `H` e l'unico metodo accessibile e applicabile per sottotipo è quello con segnatura `m(Double)`. Gli argomenti dell'invocazione **`super`**`.m(d, d + 1)` hanno, rispettivamente, tipo `Double` e **`double`**[1]; nella classe `P` non esistono metodi accessibili e applicabili solo per sottotipo o sottotipo e boxing/unboxing, mentre il metodo con segnatura `m(Object...)` è applicabile per arità variabile e per boxing del secondo argomento e conseguente widening reference conversion.

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

[1] Per il secondo argomento viene applicata una conversione implicita per unboxing da `Double` a **`double`** per la variabile `d` e una conversione implicita per promotion da **`int`** a **`double`** per il literal `1`.