Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta dell'11 settembre 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  public class MatcherTest {
2     public static void main(String[] args) {
3        Pattern regEx = Pattern.compile("([$a-zA-Z][a-zA-Z0-9]+)|([0-9]+\\.?[0-9]*|\\.[0-9]+)|(\\s+)");
4        Matcher m = regEx.matcher(".09 12.");
5        m.lookingAt();
6        assert m.group(2).equals(".09");
7        assert m.group(3) == null;
8        m.region(m.end(), m.regionEnd());
9        m.lookingAt();
10       assert m.group(0) != null;
11       assert m.group(0).length() > 0;
12       m.region(m.end(), m.regionEnd());
13       m.lookingAt();
14       assert m.group(2).equals("12.");
15       assert m.group(0).length() == 3;
16    }
17 }
```

Soluzione:

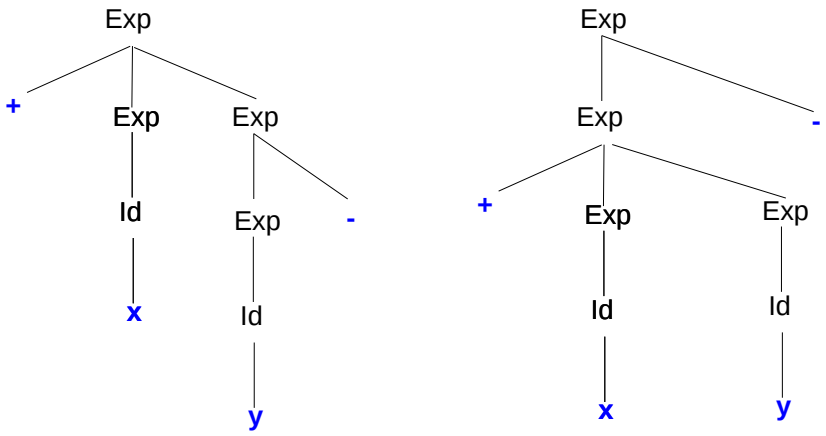
- `assert m.group(2).equals(".09");` (linea 6): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `.09 12.` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa è `.09` ed appartenente ai soli gruppi di indice 0 e 2 (carattere `'.'` seguito da successione non vuota di cifre decimali), quindi l'asserzione ha successo;
- `assert m.group(3) == null;` (linea 7): 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(0) != null;` (linea 10): alla linea 8 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `.09` (uno spazio bianco), l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene all'espressione regolare (soli gruppi 0 e 3), quindi l'asserzione ha successo;
- `assert m.group(0).length() > 0;` (linea 11): 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(2).equals("12.");` (linea 14): alla linea 12 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo allo spazio bianco (carattere `"`) e l'invocazione di `lookingAt()` ha successo poiché la stringa `"12."` appartiene ai soli gruppi 0 e 2 (successione non vuota di cifre decimali opzionalmente terminata dal carattere `'.'` seguito da successione anche vuota di cifre decimali); per tali motivi l'asserzione ha successo;
- `assert m.group(0).length() == 3;` (linea 15): lo stato del matcher non è cambiato rispetto alla linea precedente, quindi per i motivi del punto precedente l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
1  Exp  ::= + Exp Exp | Exp - | ( Exp ) | Id
2  Id   ::= x | y | z
```

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



(d) 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'attribuire maggiore priorità all'operatore unario `-`, introducendo il non terminale `Minus`:

```
1  Exp   ::= + Exp Exp | Minus
2  Minus ::= Minus - | ( Exp ) | Id
3  Id    ::= x | y | z
```

2. Sia `odd : 'a list -> 'a list` la funzione così specificata:
`odd` l restituisce la lista che contiene, nello stesso ordine, i soli elementi di l di posizione dispari (ossia, il primo, il terzo, ecc.).

Esempio:

```
# odd []
- : 'a list = []

# odd [1]
- : int list = [1]

# odd [1;2]
- : int list = [1]

# odd [1;2;3;4;5]
- : int list = [1; 3; 5]
```

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. La seguente classe `FilteredIterator` permette di iterare sugli elementi appartenenti a una lista di tipo `ArrayList<E>` che soddisfano un predicato di tipo `Predicate<E>`.

```
public interface Predicate<E> {
        boolean test(E t);
}
public class FilteredIterator<E> implements Iterator<E> {
   private final Predicate<E> pred;
   private final ArrayList<E> list;
   private int curr; // index of the current element

   public FilteredIterator(Predicate<E> pred, ArrayList<E> list) { /* da completare */ }
   public boolean hasNext() { /* da completare */  }
   public E next() { /* da completare */  }
}
```

Per esempio, le **`assert`** nel seguente frammento di codice sono sempre verificate:

```
Predicate<Integer> odd = ... // predicato che testa se un intero e' dispari
ArrayList<Integer> list = ... // lista  [1, 2, 4, 3, 5, 6]
FilteredIterator<Integer> fit = new FilteredIterator<>(odd, list);
int elem = 1;
int count = 0;
while (fit.hasNext()) {
   assert fit.next().equals(elem);
   elem += 2;
   count++;
}
assert count == 3;
```

(a) Completare le definizioni del costruttore della classe.

(b) Completare le definizioni dei metodi `hasNext()` e `next()`.

(c) Utilizzando la classe `FilteredIterator`, completare il metodo `find` che restituisce il primo elemento di una lista di tipo `ArrayList<E>` che verifica un predicato di tipo `Predicate<E>`, se tale elemento esiste, o solleva l'eccezione `NoSuchElementException` altrimenti.

```
public static <E> E find(Predicate<E> pred, ArrayList<E> list) { /* completare */ }
```

Per esempio, la seguente **`assert`** ha successo:

```
Predicate<Integer> positive = ... // predicato che testa se un numero intero e' positivo
ArrayList<Integer> list = ... // lista  [-1, -2, -3, -4, 5, 6]
assert find(positive, list).equals(5);
```

Soluzione: Vedere il file `soluzione.jar`.

4. Assumere che le seguenti dichiarazioni di classi Java siano contenute nello stesso package:

```
public class P {
   String m(Double d) {
      return "P.m(Double)";
   }
   String m(Long l) {
      return "P.m(Long)";
   }
}
public class H extends P {
   String m(double d) {
      return super.m(d) + " H.m(double)";
   }
   String m(long l) {
      return super.m(l) + " H.m(long)";
   }
}
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) `h.m(42)`

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

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

(e) `p.m(42.)`

(f) `p2.m(42.)`

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

(a) Il literal `42` ha tipo statico **`int`**, mentre `p` ha tipo statico `P`; non esistono metodo di `P` accessibili e applicabili né per sottotipo, né per boxing, dato che **`int`**$\not\leq$`Long`, **`int`**$\not\leq$`Double` `Integer`$\not\leq$`Long` e `Integer`$\not\leq$`Double`, quindi viene emesso un errore di compilazione.

(b) Il literal `42` ha tipo statico **`int`**, mentre `h` ha tipo statico `H`; esistono solo due metodi di `H`, `m(`**`long`**`)` e `m(`**`double`**`)`, entrambi accessibili e applicabili per sottotipo (**`int`**$\leq$**`long`**, **`int`**$\leq$**`double`**, **`int`**$\not\leq$`Long` e **`int`**$\not\leq$`Double`); viene selezionato il più specifico `m(`**`long`**`)` dato che **`long`**$\leq$**`double`**.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(`**`long`**`)` di `H`. L'invocazione **`super`**`.m(l)` viene risolta con il metodo `m(Long)` di `P` per boxing dato che `l` ha tipo statico **`long`** e **`long`**$\not\leq$`Long`, **`long`**$\not\leq$`Double`, `Long`$\not\leq$`Double` e `Long`$\leq$`Long`. Viene stampata la stringa `"P.m(Long) H.m(long)"`.

(c) Il literal `42L` ha tipo statico **`long`**, mentre `p` ha tipo statico `P`; essendo i tipi statici gli stessi di **`super`**`.m(l)` del caso precedente, l'invocazione viene risolta con il metodo `m(Long)`.

A runtime `p` contiene un'istanza di `P`, quindi viene eseguito il metodo `m(Long)` di `P` e viene stampata la stringa `"P.m(Long)"`.

(d) Il literal `42L` ha tipo statico **`long`**, mentre `h` ha tipo statico `H`; esistono solo due metodi di `H`, `m(`**`long`**`)` e `m(`**`double`**`)`, entrambi accessibili e applicabili per sottotipo (**`long`**$\leq$**`long`**, **`long`**$\leq$**`double`**, **`long`**$\not\leq$`Long` e **`long`**$\not\leq$`Double`); viene selezionato il più specifico `m(`**`long`**`)` come per il caso (b).

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(`**`long`**`)` di `H`. Analogamente al caso (b), viene stampata la stringa `"P.m(Long) H.m(long)"`.

(e) Il literal `42.` ha tipo statico **`double`**, mentre `p` ha tipo statico `P`; non esistono metodi di `P` accessibili e applicabili per sottotipo (**`double`**$\not\leq$`Long` e **`double`**$\not\leq$`Double`)), mentre `m(Double)` è l'unico accessibile e applicabile per boxing (`Double`$\not\leq$`Long` e `Double`$\leq$`Double`).

A runtime `p` contiene un'istanza di `P`, quindi viene eseguito il metodo `m(Double)` di `P` e viene stampata la stringa `"P.m(Double)"`.

(f) L'invocazione viene risolta con il metodo `m(Double)` come per il caso precedente, dato che i tipi statici sono gli stessi.

A runtime `p2` contiene un'istanza di `H`, ma poiché il metodo `m(Double)` non viene ridefinito in `H`, viene eseguito il metodo ereditato da `P` e viene stampata la stringa `"P.m(Double)"` come nel caso precedente.