Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 25 gennaio 2018

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("(0[0-7]*)|([1-9][0-9]*)|(\\s+)");
4        Matcher m = regEx.matcher("01 7");
5        m.lookingAt();
6        assert m.group(1).equals("01");
7        assert m.group(2) == null;
8        m.region(m.end(), m.regionEnd());
9        m.lookingAt();
10       assert !m.group(3).equals(null);
11       assert m.group(3).length() >= 1;
12       m.region(m.end(), m.regionEnd());
13       m.lookingAt();
14       assert m.group(2).equals("7");
15       assert !m.group(2).equals(7);
16    }
17 }
```

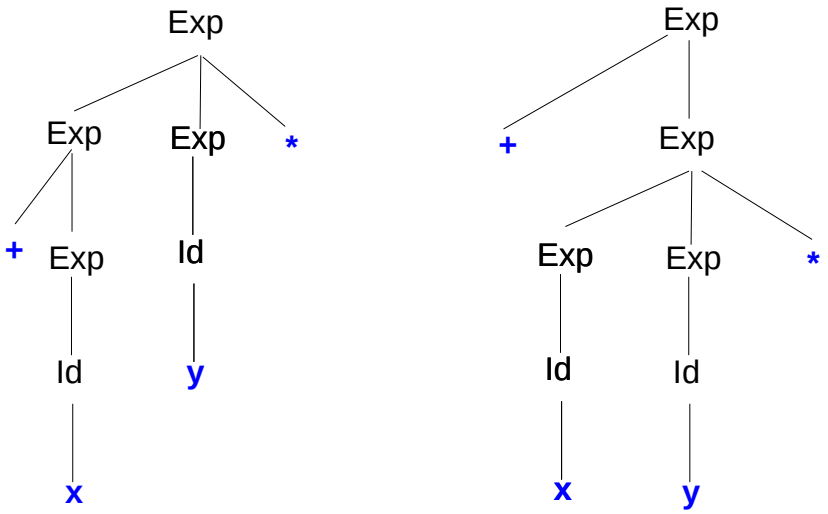
Soluzione:

- `assert m.group(1).equals("01");` (linea 6): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `01 7` 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 `01` appartenente ai soli gruppi 0 e 1 (carattere `'0'` seguito da successione non vuota di cifre ottali), quindi l'asserzione ha successo;
- `assert m.group(2) == 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(3).equals(null);` (linea 10): alla linea 8 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `01` (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(3).length() >= 1;` (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("7");` (linea 14): alla linea 12 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo allo spazio bianco (carattere `'7'`) e l'invocazione di `lookingAt()` ha successo poiché la stringa `7` appartiene ai soli gruppi 0 e 2 (successione non vuota di cifre decimali che non iniziano con `'0'`); per tali motivi l'asserzione ha successo;
- `assert m.group(2).equals("7");` (linea 15): lo stato del matcher non è cambiato alla linea 14, quindi per i motivi del punto precedente il metodo `group` restituisce un oggetto della classe `String` che rappresenta la stringa `7`, mentre l'argomento di `equals` viene convertito tramite boxing a un oggetto della classe `Integer` che rappresenta il numero intero 7, quindi 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'attribuire maggiore priorità all'operatore unario +, introducendo il non terminale `Plus`:

```
Exp  ::= Exp Exp * | Plus
Plus ::= + Plus | ( Exp ) | Id
Id   ::= x | y | z
```

2. Sia `list_gen : ('a -> 'a) -> 'a -> int -> 'a list` la funzione così specificata:
`list_gen` f i n restituisce la lista di lunghezza n dove il primo elemento è i, il secondo è $f(i)$, il terzo $f(f(i))$ e così via. Esempi:

```
# list_gen (fun x->x+1) 0 3;;
- : int list = [0; 1; 2]

# list_gen (fun x->x*2) 1 4;;
- : int list = [1; 2; 4; 8]

# list_gen (fun x->"a"^x) "" 5;;
- : string list = [""; "a"; "aa"; "aaa"; "aaaa"]
```

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. Completare il costruttore e i metodi della classe `FunIterator<T>` che permette di creare iteratori infiniti a partire da un oggetto di tipo `Function<T,T>` e un valore iniziale di tipo `T`; i valori di tipo `Function<T,T>` rappresentano funzioni da valori di tipo `T` a valori di tipo `T`.

```
// functions from T to R
public interface Function<T, R> {
        R apply(T t); // applies the function to t
}

public class FunIterator<T> implements Iterator<T> {
   private final Function<T, T> fun;
   private T first; // allowed to be null

   public FunIterator(Function<T, T> fun, T first) {
      // completare
   }

   public boolean hasNext() {
      // completare
   }

   public T next() {
      // completare
   }
}
```

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

```
Function<String, String> append = // oggetto che rappresenta la funzione OCaml fun x -> "a"^x
Iterator<String> it = new FunIterator<String>(append, "");
assert it.next().equals("");
assert it.next().equals("a");
assert it.next().equals("aa");
```

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(Number n) {
      return "P.m(Number)";
   }
   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(Integer i) {
      return super.m(i) + " H.m(Integer)";
   }
}
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(new Integer(42))`

(b) `p2.m(new Integer(42))`

(c) `h.m(new Integer(42))`

(d) `p.m(new Double(42))`

(e) `p2.m(new Double(42))`

(f) `h.m(new Double(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`; la classe `Integer` ha un costruttore pubblico con parametro di tipo **`int`** che è l'unico applicabile per sottotipo, quindi l'argomento dell'invocazione ha tipo statico `Integer`. Poiché `Integer`≤`Number`, `Integer`≰**`double`** l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number)"`.

(b) Per analogia con il caso precedente i tipi statici coinvolti sono identici, quindi anche in questo caso viene selezionato il metodo con segnatura `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi la ricerca del metodo inizia dalla classe `H`; non essendo ridefinito in `H`, viene eseguito come nel caso precedente il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number)"`.

(c) Come nei casi precedenti, l'argomento dell'invocazione ha tipo statico `Integer`, mentre `h` ha tipo statico `H`. Poiché `Integer`≤`Integer`, `Integer`≤`Number`, `Integer`≰`Double`, `Integer`≰**`double`**, esistono due metodi di `H` accessibili e applicabili, aventi segnatura `m(Integer)` e `m(Number)`; viene selezionato il più specifico `m(Integer)`, dato che `Integer`≤`Number`.

A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo `m(Integer)` di `H`. La chiamata **`super`**`.m(i)` viene risolta come ai punti precedenti visto che il tipo statico dell'oggetto receiver è `P` e il tipo statico dell'argomento `i` è `Integer`, quindi viene invocato il metodo `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number) H.m(Integer)"`.

(d) Il literal `42` ha tipo statico **`int`**, mentre `p` ha tipo statico `P`; la classe `Double` ha un costruttore pubblico con parametro di tipo **`double`** che è l'unico applicabile per sottotipo, quindi l'argomento dell'invocazione ha tipo statico `Double`. Poiché `Double`≤`Number`, `Double`≰**`double`** l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number)"`.

(e) Per analogia con il caso precedente i tipi statici coinvolti sono identici, quindi anche in questo caso viene selezionato il metodo con segnatura `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi la ricerca del metodo inizia dalla classe `H`; non essendo ridefinito in `H`, viene eseguito come nel caso precedente il metodo con segnatura `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number)"`.

(f) Come nei due casi precedenti, l'argomento dell'invocazione ha tipo statico `Double`, mentre `h` ha tipo statico `H`. Poiché `Double`≤`Double`, `Double`≤`Number`, `Double`≰`Integer`, `Double`≰**`double`**, esistono due metodi di `H` accessibili e applicabili, aventi segnatura `m(Double)` e `m(Number)`; viene selezionato il più specifico `m(Double)`, dato che `Double`≤`Number`.

A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo `m(Double)` di `H`. La chiamata **`super`**`.m(d)` viene risolta come ai punti precedenti visto che il tipo statico dell'oggetto receiver è `P` e il tipo statico dell'argomento `d` è `Double`, quindi viene invocato il metodo `m(Number)` di `P`. Viene stampata la stringa `"P.m(Number) H.m(Double)"`.