Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 23 gennaio 2019

a.a. 2019/2020

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  public class MatcherTest {
4      public static void main(String[] args) {
5          Pattern regEx = Pattern.compile("(\\s+)|([*+/-])|(0[0-7]*)|([a-zA-Z][0-9]*)");
6          Matcher m = regEx.matcher("077+x42");
7          m.lookingAt();
8          assert "077".equals(m.group(3));
9          assert "".equals(m.group(4));
10         m.region(m.end(), m.regionEnd());
11         m.lookingAt();
12         assert null != m.group(2);
13         assert "+".equals(m.group(0));
14         m.region(m.end(), m.regionEnd());
15         m.lookingAt();
16         assert "x".equals(m.group(4));
17         assert "42".equals(m.group(3));
18     }
19 }
```

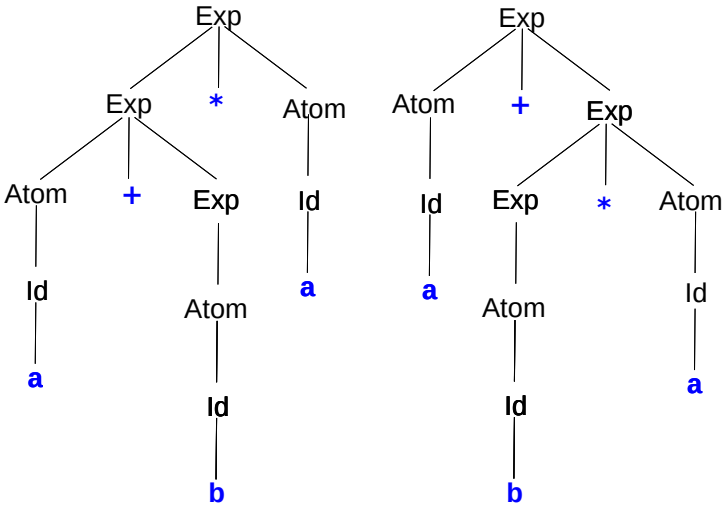
Soluzione:

- `assert "077".equals(m.group(3));` (linea 8): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `077+x42` e `lookingAt()` controlla che a partire da tale indice esista una sottostringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sottostringa esiste ed è `077` (appartenente ai soli gruppi 0 e 3, literal numerici in base 8), quindi l'asserzione ha successo;
- `assert "".equals(m.group(4));` (linea 9): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente il metodo `group` restituisce **`null`** e l'asserzione fallisce;
- `assert null != m.group(2);` (linea 12): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `077` (ossia il carattere `+`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `+` appartiene alla sottoespressione regolare `[*+/-]` corrispondente ai soli gruppi 0 e 2 (simboli di operazione), quindi il metodo `group` restituisce la stringa `+` e l'asserzione ha successo;
- `assert "+".equals(m.group(0));` (linea 13): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert "x".equals(m.group(4));` (linea 16): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo alla stringa `+` (ossia `x`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `x42` appartiene alla sottoespressione regolare `[a-zA-Z][0-9]*` corrispondente ai soli gruppi 0 e 4 (identificatori); per tale motivo, il metodo `group` restituisce la stringa `x42` e l'asserzione fallisce;
- `assert "42".equals(m.group(3));` (linea 17): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi precedenti il metodo `group` restituisce **`null`** e l'asserzione fallisce.

(b) Mostrare che nella seguente grammatica la stringa `a+b*a` è ambigua rispetto a `Exp`.

```
Exp ::= Exp * Atom | Atom + Exp | Atom
Atom ::=  ( Exp ) | Id
Id ::= a | b
```

Soluzione: Per la stringa `a+b*a` a partire da `Exp` esistono i seguenti due diversi alberi di derivazione:



(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 grammatica è ambigua poiché i due operatori hanno lo stesso livello di precedenza, ma associatività diversa, quindi una semplice soluzione consiste nell'imporre la stessa associatività (per esempio, quella a sinistra).

```
Exp ::= Exp * Atom | Exp + Atom | Atom
Atom ::=  ( Exp ) | Id
Id ::= a | b
```

2. Sia `zip : 'a list -> 'b list -> ('a * 'b) list` la funzione così specificata:

   - `zip` $[x_1;\ldots;x_{n+k}]$ $[y_1;\ldots;y_n]$ restituisce la lista di coppie $[(x_1,y_1);\ldots;(x_n,y_n)]$;
   - `zip` $[x_1;\ldots;x_n]$ $[y_1;\ldots;y_{n+k}]$ restituisce la lista di coppie $[(x_1,y_1);\ldots;(x_n,y_n)]$.

   Esempi:

```
zip [1;2;3] ["one";"two";"three"]=[(1, "one"); (2, "two"); (3, "three")];;
zip [1;2] ["one";"two";"three"]=[(1, "one"); (2, "two")];;
zip [1;2;3] ["one";"two"]=[(1, "one"); (2, "two")];;
```

   (a) Completare la seguente definizione di `zip` senza uso di parametri di accumulazione.

```
let rec zip l1 l2 = match l1,l2 with (* completare *)
```

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

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
  String m(long i) {return "P.m(long)";}
  String m(long... i) {return "P.m(long...)";}
}
public class H extends P {
  String m(int i) {return super.m(i) + " H.m(int)";}
```

```
  String m(int... i) {return super.m(i[0],i[1]) + " H.m(int...)";}
}
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(42,42)` (e) `p2.m(42,42)` (f) `h.m(42,42)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, tutti i metodi sono accessibili.

(a) Il tipo statico di `p` è `P`, il literal `42` ha tipo statico **`int`**.

- prima fase (applicabilità per sottotipo): poiché **`int`** $\leq$ **`long`** e **`int`** $\not\leq$ **`long`**`[]` (nella prima e seconda fase **`long...`** viene considerato uguale al tipo array **`long`**`[]`), solo il metodo `m(`**`long`**`)` di `P` è applicabile.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `m(`**`long`**`)` in `P` e viene stampata la stringa `"P.m(long)"`.

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto (a), visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo `m(`**`long`**`)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto (a) e, quindi, viene stampata la stringa `"P.m(long)"`.

(c) Il tipo statico di `h` è `H` mentre l'argomento ha sempre tipo statico **`int`**.

- prima fase (applicabilità per sottotipo): oltre al metodo `m(`**`long`**`)` ereditato da `P` e applicabile per le stesse ragioni dei punti (a) e (b), risulta anche applicabile il metodo `m(`**`int`**`)` di `H` (dato che ovviamente **`int`**$\leq$**`int`**), mentre non lo è `m(`**`int...`**`)` poiché **`int`** $\not\leq$ **`int`**`[]` (nella prima e seconda fase **`int...`** viene considerato uguale al tipo array **`int`**`[]`); poiché **`int`**$\leq$**`long`** viene selezionato il metodo più specifico `m(`**`int`**`)`.
  A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(`**`int`**`)` di `H`; poiché il parametro `i` ha tipo statico **`int`** e **`super`** si riferisce alla classe `P`, la chiamata **`super`**`.m(i)` si comporta come al punto (a); viene quindi stampata la stringa `"P.m(long) H.m(int)"`.

(d) Il literal `42` ha tipo statico **`int`** e il tipo statico di `p` è `P`.

- prima fase (applicabilità per sottotipo): poiché entrambi i metodi `m` di `P` hanno un solo parametro (il metodo `m(`**`long...`**`)` è considerato con numero variabile di parametri solo nella terza fase) mentre gli argomenti sono due, nessun metodo è applicabile;
- seconda fase (applicabilità per boxing/unboxing e sottotipo): nessun metodo `m` di `P` è applicabile per le stesse ragioni del punto precedente;
- terza fase (applicabilità per arietà variabile, boxing/unboxing e sottotipo): il metodo `m(`**`long...`**`)` viene considerato con un numero variabile di parametri di tipo **`long`** ed è quindi applicabile dato che **`int`**$\leq$**`long`**.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `m(`**`long...`**`)` in `P` e viene stampata la stringa `"P.m(long...)"`.

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto (d), visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo `m(`**`long...`**`)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto (d) e viene stampata la stringa `"P.m(long...)"`.

(f) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`.

- prima fase (applicabilità per sottotipo): per le stesse ragioni dei punti (d) ed (e) non sono applicabili né i metodi di `P`, né quelli di `H`;
- seconda fase (applicabilità per boxing/unboxing e sottotipo): nessun metodo `m` di `P` e `H` è applicabile per le stesse ragioni del punto precedente;

- terza fase (applicabilità per arietà variabile, boxing/unboxing e sottotipo): oltre al metodo `m(long...)` ereditato da `P` e applicabile per le stesse ragioni dei punti (d) ed (e), il metodo `m(int...)` in `H` viene considerato con un numero variabile di parametri di tipo **`int`** ed è quindi applicabile (dato che ovviamente **`int`**$\leq$**`int`**); poiché **`int`**$\leq$**`long`** viene selezionato il metodo più specifico `m(int...)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(int...)` di `H`; poiché il parametro `i` ha tipo statico **`int`**`[]`, **`super`** si riferisce alla classe `P` e gli argomenti `i[0]` e `i[1]` hanno tipo **`int`**, la chiamata **`super`**`.m(i[0],i[1])` si comporta come al punto (d); viene quindi stampata la stringa `"P.m(long...) H.m(int...)"`.

4. (a) Completare la classe `PairImp` che implementa coppie di valori diversi da **`null`**.

```
public interface Pair<T1, T2> {
  T1 getFirst();
  T2 getSecond();
}
public class PairImp<T1, T2> implements Pair<T1, T2> {
  public final T1 first; /* invariante: first != null */
  public final T2 second; /* invariante: second != null */

  public PairImp(T1 first, T2 second) { /* completare */ }
  public T1 getFirst() { /* completare */ }
  public T2 getSecond() { /* completare */ }
  public String toString() { /* completare */ } // restituisce "("first","second")"
}
```

(b) Completare la classe `Zipper` che definisce iteratori zipper a partire da due iteratori `iterator1` e `iterator2` inizialmente ottenuti da due oggetti `iterable1` e `iterable2` di tipo `Iterable`. A ogni iterazione il metodo `next()` dello zipper restituisce la coppia di elementi rispettivamente ottenuti dai due iteratori `iterator1` e `iterator2` tramite il metodo `next()`; l'iterazione termina quando uno dei due iteratori `iterator1` e `iterator2` non ha più elementi da restituire.

```
public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {
  private final Iterator<T1> iterator1;
  private final Iterator<T2> iterator2;

  public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) { /* completare */ }
  public boolean hasNext() { /* completare */ }
  public Pair<T1, T2> next() { /* completare */ }
}
```

Il seguente codice mostra un semplice esempio di uso della classe `Zipper`. Il metodo statico `java.util.Arrays.asList` crea una nuova lista a partire dagli argomenti passati; per esempio, `asList(1, 2, 3)` restituisce la lista `[1,2,3]`.

```
import static java.util.Arrays.asList;

public class Test {
  public static void main(String[] args) {
    Zipper<Integer, String> zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two", "three"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two) (3,three)
    zipper = new Zipper<>(asList(1, 2), asList("one", "two", "three"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two)
    zipper = new Zipper<>(asList(1, 2, 3), asList("one", "two"));
    while (zipper.hasNext())
      System.out.println(zipper.next()); // stampa (1,one) (2,two)
  }
}
```

Soluzione: Vedere il file `soluzione.jar`.