Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta parziale del 18 febbraio 2020

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+)|([01]+)|([a-z][a-zA-Z0-9]*)|(ADD|SUB|MUL|DIV)");
6        Matcher m = regEx.matcher("ADD1010x01");
7        m.lookingAt();
8        assert "".equals(m.group(1));
9        assert "ADD".equals(m.group(4));
10       m.region(m.end(), m.regionEnd());
11       m.lookingAt();
12       assert "1010".equals(m.group(0));
13       assert null == m.group(2);
14       m.region(m.end(), m.regionEnd());
15       m.lookingAt();
16       assert "x01".equals(m.group(3));
17       assert null == m.group(2);
18    }
19 }
```

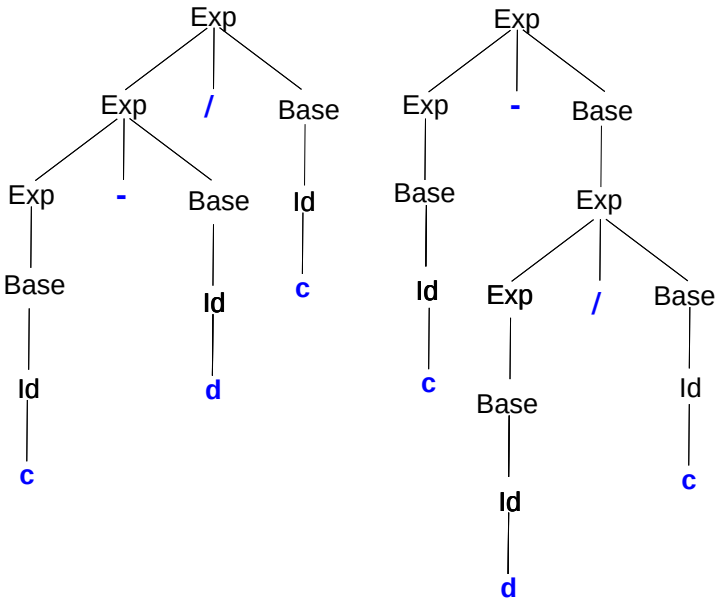
Soluzione:

- `assert "".equals(m.group(1));` (linea 8): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `ADD1010x01` 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 `ADD` (appartenente ai soli gruppi 0 e 4, sottoespressione `ADD|SUB|MUL|DIV`); poiché il gruppo 1 non contribuisce al riconoscimento della stringa, il metodo `group` restituisce **`null`** e l'asserzione fallisce;
- `assert "ADD".equals(m.group(4));` (linea 9): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione ha successo;
- `"1010".equals(m.group(0))` (linea 12): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo alla stringa `ADD` (ossia il carattere `1`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `1010` appartiene alla sottoespressione regolare `[01]+` corrispondente ai soli gruppi 0 e 2 (numeri binari), quindi il metodo `group` restituisce la stringa `1010` e l'asserzione ha successo;
- `assert null == m.group(2);` (linea 13): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione fallisce;
- `assert "x01".equals(m.group(3));` (linea 16): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo alla stringa `1010` (ossia `x`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `x01` appartiene alla sottoespressione regolare `[a-z][a-zA-Z0-9]*` corrispondente ai soli gruppi 0 e 3 (identificatori); per tale motivo, il metodo `group` restituisce la stringa `x01` e l'asserzione ha successo;
- `assert null == m.group(2);` (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 ha successo.

(b) Mostrare che nella seguente grammatica la stringa `c-d/c` è ambigua rispetto a `Exp`.

```
Exp ::= Exp / Base | Exp - Base | Base
Base ::=  Id | { Exp } | Exp
Id ::= c | d
```

Soluzione: Per la stringa `c-d/c` 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: L'ambiguità è causata dalla sola produzione `Base ::= Exp`; la grammatica ottenuta rimuovendo tale produzione non è ambigua poiché impone lo stesso livello di precedenza per i due operatori e associatività a sinistra; inoltre, il linguaggio generato a partire da `Exp` rimane lo stesso.

```
Exp ::= Exp / Base | Exp - Base | Base
Base ::=  Id | { Exp }
Id ::= c | d
```

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

   `unzip` $[(x_1, y_1); \ldots; (x_n, y_n)]$ restituisce la lista $[x_1; \ldots; x_n]$.

   Esempi:

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

   (a) Definire `unzip` senza uso di parametri di accumulazione.

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

   (c) Definire `unzip` come specializzazione della funzione `List.map : ('a -> 'b) -> 'a list -> 'b list`.

   Soluzione: Vedere il file `soluzione.ml`.

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(double d) {
      return "P.m(double)";
   }
   String m(double[] da) {
      return "P.m(double[])";
   }
}
public class H extends P {
   String m(float f) {
      return super.m(f) + " H.m(float)";
   }
   String m(float[] fa) {
      return super.m(new double[]{ fa[0], fa[1] }) + " H.m(float[])";
   }
}
public class Test {
   public static void main(String[] args) {
      P p = new P();
      H h = new H();
      P p2 = h;
      float f = 4.2f;
      double[] da = { 1.2, 2.3 };
      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(f)` (b) `p2.m(f)` (c) `h.m(f)` (d) `p.m(da)` (e) `p2.m(da)` (f) `h.m(da)`

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

(a) Il tipo statico di `p` è `P` e quello di `f` è **`float`**.

- prima fase (applicabilità per sottotipo): poiché **`float`** $\leq$ **`double`** e **`float`** $\not\leq$ **`double`**`[]`, solo il metodo `m(`**`double`**`)` di `P` è applicabile.

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

(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(`**`double`**`)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto (a) e, quindi, viene stampata la stringa `"P.m(double)"`.

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

- prima fase (applicabilità per sottotipo): oltre al metodo `m(`**`double`**`)` ereditato da `P` e applicabile per le stesse ragioni dei punti (a) e (b), risulta anche applicabile il metodo `m(`**`float`**`)` di `H` (dato che ovviamente **`float`**$\leq$**`float`**), mentre non lo è `m(`**`float`**`[])` poiché **`float`** $\not\leq$ **`float`**`[]`; poiché **`float`** $\leq$ **`double`** viene selezionato il metodo più specifico `m(`**`float`**`)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(`**`float`**`)` di `H`; poiché il parametro `f` ha tipo statico **`float`** e **`super`** si riferisce alla classe `P`, la chiamata **`super`**`.m(f)` si comporta come al punto (a) e quindi viene stampata la stringa `"P.m(double) H.m(float)"`.

(d) Il tipo statico di `p` è `P` e quello di `da` è **`double`**`[]`.

- prima fase (applicabilità per sottotipo): poiché **`double`**`[]` $\leq$ **`double`**`[]` e **`double`**`[]` $\not\leq$ **`double`**, solo il metodo `m(`**`double`**`[])` di `P` è applicabile.

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

(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(`**`double`**`[])` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto (d) e viene stampata la stringa `"P.m(double[])"`.

(f) Il tipo statico di `h` è `H` mentre l'argomento `da` ha sempre tipo statico **`double`**`[]`.

- prima fase (applicabilità per sottotipo): oltre al metodo `m(`**`double`**`[])` ereditato da `P` e applicabile per le stesse ragioni dei punti (d) ed (e), non sono applicabili altri metodi poiché **`double`**`[]` $\not\leq$ **`float`** e **`double`**`[]` $\not\leq$ **`float`**`[]`.

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