Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 5 giugno 2019

a.a. 2018/2019

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("('[^']*')|([$%][a-zA-Z0-9]*)|(\\s+)");
7          Matcher m = regEx.matcher("$S3 'hello world'");
8          m.lookingAt();
9          assert m.group(2).equals("$");
10         assert m.group(0).equals("%S3");
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(3).equals("'");
14         assert m.group(0).equals("'hello world'");
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group(1).equals("'");
18         assert m.group(0).equals("'hello");
19     }
20 }
```

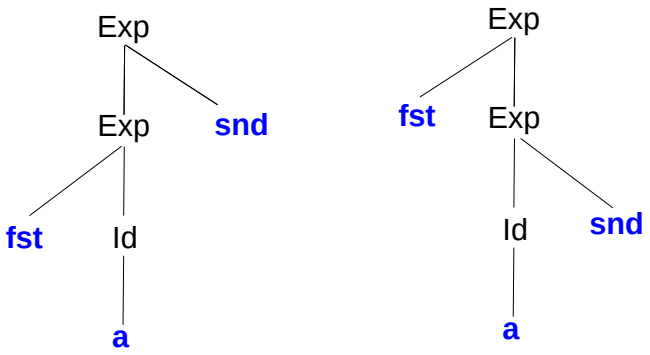
Soluzione:

- **assert** `m.group(2).equals("$");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `$S3 'hello world'` 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 `$S3` (appartenente ai soli gruppi 0 e 2: qualsiasi stringa non vuota che inizia con `$` o `%` seguito da zero o più lettere maiuscole, minuscole o cifre decimali, quindi il metodo restituisce un oggetto che rappresenta la stringa `$S3` e l'asserzione fallisce;
- **assert** `m.group(0).equals("%S3");` (linea 10): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione fallisce;
- **assert** `m.group(3).equals("'");` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `$S3` (ossia uno spazio bianco) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché una successione non vuota di spazi bianchi appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3, quindi l'asserzione fallisce;
- **assert** `m.group(0).equals("'hello world'");` (linea 14): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione fallisce;
- **assert** `m.group(1).equals("'");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `'`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `'hello world'` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 1 (successione possibilmente vuota di caratteri diversi da `'` e delimitata da `'` da entrambe le parti); per tale motivo, `m.group(2)` restituisce un oggetto che rappresenta la stringa `'hello world'`, quindi l'asserzione fallisce;
- **assert** `m.group(0).equals("'hello");` (linea 18): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi precedenti l'asserzione fallisce.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= fst Exp | Exp snd | ( Exp , Exp ) | Id
Id ::= a | b
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `fst a snd`



(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 `Snd` per poter attribuire la precedenza all'operatore unario `snd`.

```
Exp ::= fst Exp | Snd
Snd ::= Snd snd | ( Exp , Exp ) | Id
Id ::= a | b
```

2. Sia `gen_sum : ('a -> int) -> 'a list -> int` la funzione così specificata:

   `gen_sum` f `[`x_1`; `x_2`; ... ; `x_n`]` $= f(x_1) + f(x_2) + \ldots + f(x_n)$, con $n \geq 0$.

   Esempi:

```
# gen_sum (fun x->x*x) []
- : int = 0
# gen_sum (fun x->x*x) [1]
- : int = 1
# gen_sum (fun x->x*x) [1;2]
- : int = 5
# gen_sum (fun x->x*x) [1;2;3]
- : int = 14
```

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

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

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

   Soluzione: Vedere il file `soluzione.ml`.

3. Completare la seguente classe di iteratori `Multiples` per generare in ordine crescente la sequenza dei multipli di un numero. Per esempio, il seguente codice

```
for (int n : new Multiples(3, 4)) // genera i primi 4 multipli di 3
    System.out.println(n);
```

   stampa la sequenza

```
3
6
9
12
```

```
import java.util.Iterator;

public class Multiples implements Iterator<Integer>, Iterable<Integer> {
        private final int step; // passo tra un elemento della sequenza e il suo seguente
        private final int max; // ultimo numero della sequenza
        private int next; // prossimo numero della sequenza

        /* parametro step: passo tra un elemento della sequenza e il suo seguente
           parametro items: numero totale di elementi della sequenza
           pre-condizione: step > 0 && items >= 0 */
        public Multiples(int step, int items) {
                // completare
        }
        public boolean hasNext() {
                // completare
        }
        public Integer next() {
                // completare
        }
        public Iterator<Integer> iterator() { // restituisce se stesso
                // completare
        }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Object... o) {
        return "P.m(Object...)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
public class H extends P {
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(Double d) {
        return super.m(d) + " 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(42)`

(b) `p2.m(42)`

(c) `h.m(42)`

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`

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

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

- primo tentativo (solo sottotipo): poiché **`int`** $\not\leq$ `Object` e **`int`** $\not\leq$ `Object[]` (al primo e al secondo tentativo `Object...` viene trattato come `Object[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da **`int`** a `Integer` e poiché `Integer` $\leq$ `Object` e `Integer` $\not\leq$ `Object[]`, solo il metodo `m(Object)` di `P` è applicabile per boxing e reference widening.

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

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

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

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

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (**`int`** $\not\leq$ `Object`, **`int`** $\not\leq$ `Object[]`, **`int`** $\not\leq$ `Integer`, **`int`** $\not\leq$ `Double`);
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da **`int`** a `Integer` e poiché `Integer` $\leq$ `Object`, `Integer` $\leq$ `Integer` e `Integer` $\not\leq$ `Object[]`, `Integer` $\not\leq$ `Double`, solo i metodi `m(Object)` e `m(Integer)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Object)`) e poiché `Integer` $\leq$ `Object`, l'overloading viene risolto con la segnatura più specifica `m(Integer)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Integer)`; poiché il parametro `i` ha tipo statico `Integer` e **`super`** si riferisce alla classe `P`, semantica statica e dinamica della chiamata **`super`**`.m(i)` coincidono con il caso illustrato al punto (a); viene quindi stampata la stringa `"P.m(Object) H.m(Integer)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché **`double`** $\not\leq$ `Object` e **`double`** $\not\leq$ `Object[]` (al primo e al secondo tentativo `Object...` viene trattato come `Object[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da **`double`** a `Double` e poiché `Double` $\leq$ `Object` e `Double` $\not\leq$ `Object[]`, solo il metodo `m(Object)` di `P` è applicabile per boxing e reference widening.

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

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

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

(f) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (**`double`** $\not\leq$ `Object`, **`double`** $\not\leq$ `Object[]`, **`double`** $\not\leq$ `Integer`, **`double`** $\not\leq$ `Double`);
- secondo tentativo (boxing/unboxing e sottotipo): applicando una conversione boxing da **`double`** a `Double` e poiché `Double` $\leq$ `Object`, `Double` $\leq$ `Double` e `Double` $\not\leq$ `Object[]`, `Double` $\not\leq$ `Integer`, solo i metodi `m(Object)` e `m(Double)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Object)`) e poiché `Double` $\leq$ `Object`, l'overloading viene risolto con la segnatura più specifica `m(Double)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Double)`; poiché il parametro `d` ha tipo statico `Double` e **`super`** si riferisce alla classe `P`, semantica statica e dinamica della chiamata **`super`**`.m(d)` coincidono con il caso illustrato al punto (d); viene quindi stampata la stringa `"P.m(Object) H.m(Double)"`.