Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 19 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 =
7            Pattern.compile("(\\s+|//.*)|([a-zA-Z][a-zA-Z0-9]*(\\.[a-zA-Z][a-zA-Z0-9]*)*)");
8          Matcher m = regEx.matcher("java.util // example");
9          m.lookingAt();
10         assert m.group(2).equals("java.util");
11         assert m.group(0) != null;
12         m.region(m.end(), m.regionEnd());
13         m.lookingAt();
14         assert m.group(0) != null;
15         assert m.group(1) != null;
16         m.region(m.end(), m.regionEnd());
17         m.lookingAt();
18         assert m.group(1).equals("// example");
19         assert m.group(2) != null;
20     }
21 }
```

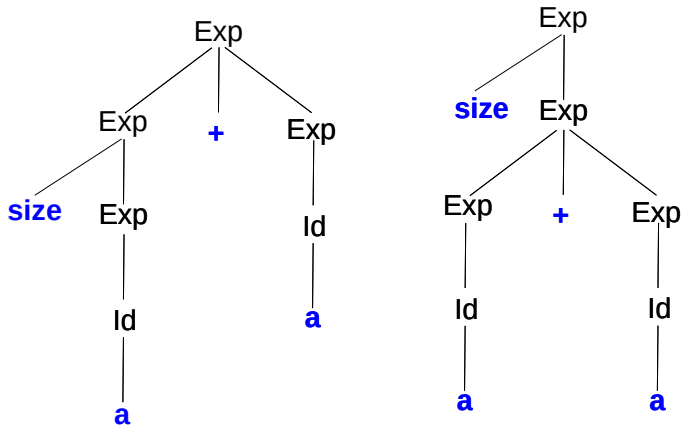
Soluzione:

- `assert m.group(2).equals("java.util");` (linea 10): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `java.util // example` 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 `java.util` (appartenente ai soli gruppi 0 e 2: qualsiasi sequenza di uno o più identificatori (stringhe non vuote di caratteri alfanumerici che iniziano con una lettera) separati dal punto; quindi, l'asserzione ha successo;
- `assert m.group(0) != null;` (linea 11): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(0) != null;` (linea 14): alla linea 12 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `java.util` (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 `\\s+` corrispondente ai soli gruppi 0 e 1, quindi l'asserzione ha successo;
- `assert m.group(1) != null;` (linea 15): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(1).equals("// example");` (linea 18): alla linea 16 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 `// example` appartiene alla sotto-espressione regolare `//.*` corrispondente ai soli gruppi 0 e 1 (commento monolinea delimitato da `//`); per tale motivo, l'asserzione ha successo;
- `assert m.group(2) != null;` (linea 19): 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 ::= size Exp | Exp + Exp | [ Exps ] | Id
Exps ::= Exp | Exp , Exps
Id ::= a | b
```

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



(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 `Size` per poter attribuire la precedenza all'operatore unario `size` e forzare l'associatività a sinistra dell'operatore binario `+`.

```
Exp ::= Exp + Size | Size
Size ::= size Size | [ Exps ] | Id
Exps ::= Exp | Exp , Exps
Id ::= a | b
```

2. Sia `gen_prod : ('a -> int) -> 'a list -> int` la funzione così specificata:
   `gen_prod` f `[`x_1`; `x_2`; ... ; `x_n`]` $= f(x_1) \cdot f(x_2) \cdot \ldots \cdot f(x_n)$, con $n \geq 0$.
   Esempi:

```
# gen_prod (fun x->x+2) []
- : int = 1
# gen_prod (fun x->x+2) [1]
- : int = 3
# gen_prod (fun x->x+2) [1;2]
- : int = 12
# gen_prod (fun x->x+2) [1;2;3]
- : int = 60
```

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

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

   (c) Definire `gen_prod` 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 `Powers` per generare la sequenza di potenze di un numero intero in ordine crescente di esponente a partire da 1. Per esempio, il seguente codice

```
// genera la sequenza 3^1, 3^2, 3^3, 3^4
for (int n : new Powers(3, 4))
    System.out.println(n);
```

   stampa la sequenza

```
3
9
27
81
```

```
import java.util.Iterator;

public class Powers implements Iterator<Integer>, Iterable<Integer> {

    private final int base; // base dell'esponente
    private int items; // numero di elementi ancora da generare
    private int next; // prossimo elemento da restituire

    // precondizione: items >= 0
    public Powers(int base, int items) {
        // completare
    }
    public boolean hasNext() {
        // completare
    }
    public Integer next() {
        // completare
    }
    // restituisce se stesso
    public Iterator<Integer> iterator() {
        // completare
    }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number... o) {
        return "P.m(Number...)";
    }
    String m(Number o) {
        return "P.m(Number)";
    }
}
public class H extends P {
    String m(Short s) {
        return super.m(s) + " H.m(Short)";
    }
    String m(Float f) {
        return super.m(f) + " H.m(Float)";
    }
}
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((short) 42)`

(b) `p2.m((short) 42)`

(c) `h.m((short) 42)`

(d) `p.m(42.0f)`

(e) `p2.m(42.0f)`

(f) `h.m(42.0f)`

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

(a) Il tipo statico di `p` è `P`, il literal `42` ha tipo statico **`int`**, il cast a **`short`** è staticamente corretto (narrowing primitive conversion) e l'argomento `(short) 42` ha tipo statico **`short`**.

- primo tentativo (solo sottotipo): poiché **`short`** $\not\leq$ `Number` e **`short`** $\not\leq$ `Number[]` (al primo e al secondo tentativo `Number...` viene trattato come `Number[]`), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da **`short`** a `Short`, poiché `Short` $\leq$ `Number` e `Short` $\not\leq$ `Number[]`, solo il metodo `m(Number)` 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(Number)` in `P` e viene stampata la stringa `"P.m(Number)"`.

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

(c) Il tipo statico di `h` è `H` e l'argomento ha tipo statico **`short`** come ai punti precedenti.

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

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Short)`; poiché il parametro `s` ha tipo statico `Short` e **`super`** si riferisce alla classe `P`, la chiamata **`super`**`.m(s)` si comporta analogamente al caso illustrato al punto (a); viene quindi stampata la stringa `"P.m(Number) H.m(Short)"`.

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

- primo tentativo (solo sottotipo): poiché **`float`** $\not\leq$ `Object` e **`float`** $\not\leq$ `Number[]`, non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da **`float`** a `Float`, poiché `Float` $\leq$ `Number` e `Float` $\not\leq$ `Number[]`, solo il metodo `m(Number)` 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(Number)` in `P` e viene stampata la stringa `"P.m(Number)"`.

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

(f) Il literal `42.0f` ha tipo statico **`float`** 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 (**`float`** $\not\leq$ `Number`, **`float`** $\not\leq$ `Number[]`, **`float`** $\not\leq$ `Short`, **`float`** $\not\leq$ `Float`);
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da **`float`** a `Float`, poiché `Float` $\leq$ `Number`, `Float` $\leq$ `Float` e `Float` $\not\leq$ `Number[]`, `Float` $\not\leq$ `Short`, solo i metodi `m(Number)` e `m(Float)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Number)`) e poiché `Float` $\leq$ `Number`, l'overloading viene risolto con la segnatura più specifica `m(Float)`.

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