Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 10 luglio 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("([0-9]+)|(\\s+)|([a-zA-Z][a-zA-Z]*(\\.[a-zA-Z][a-zA-Z]*)*)");
8          Matcher m = regEx.matcher("001 a.b");
9          m.lookingAt();
10         assert m.group(1).equals("001");
11         assert m.group(0).equals("001 a.b");
12         m.region(m.end(), m.regionEnd());
13         m.lookingAt();
14         assert m.group(2) != null;
15         assert m.group(0).equals(" a.b");
16         m.region(m.end(), m.regionEnd());
17         m.lookingAt();
18         assert m.group(3).equals("a.b");
19         assert m.group(0).equals("a.b");
20     }
21 }
```

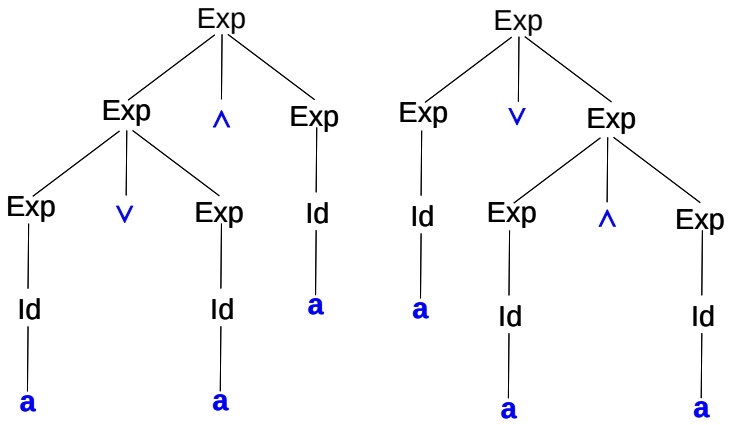
Soluzione:

- `assert m.group(1).equals("001");` (linea 10): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `001 a.b` 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 `001` (appartenente ai soli gruppi 0 e 1: qualsiasi sequenza di uno o più cifre decimali), quindi l'asserzione ha successo;
- `assert m.group(0).equals("001 a.b")` (linea 11): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione fallisce;
- `assert m.group(2) != null;` (linea 14): alla linea 12 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `001` (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 2, quindi l'asserzione ha successo;
- `assert m.group(0).equals(" a.b")` (linea 15): 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("a.b");` (linea 18): alla linea 16 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `a`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `a.b` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3 (sequenza di uno o più identificatori formati da lettere minuscole e maiuscole e separati dal punto); per tale motivo, l'asserzione ha successo;
- `assert m.group(0).equals("a.b");` (linea 19): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi precedenti l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp ∧ Exp | Exp ∨ Exp | { ExpSeq } | Id
ExpSeq ::= Exp | Exp ExpSeq
Id ::= a | b
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `a ∨ 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 `And` per poter attribuire la precedenza all'operatore ∧ e forzare l'associatività a sinistra degli operatori ∧ e ∨.

```
Exp ::= Exp ∨ And | And
And ::= And ∧ Atom | Atom
Atom ::= { ExpSeq } | Id
ExpSeq ::= Exp | Exp ExpSeq
Id ::= a | b
```

2. Sia `gen_cat : ('a -> string) -> 'a list -> string` la funzione così specificata, dove ^ è l'operatore di concatenazione di stringhe:

   `gen_cat` f `[`s_1`; `s_2`; ... ; `s_n`]` $= f(s_1)\hat{}f(s_2)\hat{}\ldots\hat{}f(s_n)$, con $n \geq 0$.

   Esempio:

```
# gen_cat (fun s -> s^"__") ["one";"two";"three"];;
- : string = "one__two__three__"
```

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

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. (a) Completare le seguenti classi che implementano i nodi di un AST per espressioni con literal interi positivi e operazione di elevamento a potenza.

```
public interface Visitor<T> {
    T visitNatLit(int val);
    T visitPow(Exp left, Exp right);
}
public interface Exp { <T> T accept(Visitor<T> visitor); }

// nodi AST per literal interi positivi
public class PosLit implements Exp {
    private final int val;

    // precondizione: val > 0
```

```
    public PosLit(int val) { /* completare */ }
    public <T> T accept(Visitor<T> visitor) { /* completare */ }
}
// nodi AST per elevamento a potenza
public class Pow implements Exp {
    private final Exp left; // base
    private final Exp right; // potenza

    // precondizione: left!=null, right!=null
    public Pow(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> visitor) { /* completare */ }
}
```

(b) Completare la classe `Eval` per la valutazione degli AST; la classe `Test` contiene una prova esplicativa.

```
public class Eval implements Visitor<Integer> {
    public Integer visitNatLit(int val) { /* completare */ }
    public Integer visitPow(Exp left, Exp right) { /* completare */ }
}
public class Test {
    public static void main(String[] args) {
        Exp e = new Pow(new PosLit(3), new PosLit(4)); // AST per 3 elevato 4
        assert e.accept(new Eval()) == 81;
    }
}
```

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(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Integer i1, Integer i2) { return super.m(i1, i2) + " H.m(Integer,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, 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, si hanno i seguenti casi:

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

- primo tentativo (solo sottotipo): poiché **`int`** $\not\leq$ `Number` e **`int`** $\not\leq$ `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 **int** a `Integer`, poiché `Integer` $\leq$ `Number` e `Integer` $\not\leq$ `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 **int** come ai punti precedenti.

- primo tentativo (solo sottotipo): nessun metodo accessibile definito in `H` o ereditato da `P` è applicabile per sottotipo (**int** $\not\leq$ `Number`, **int** $\not\leq$ `Number[]`, **int** $\not\leq$ `Integer`);
- secondo tentativo (boxing/unboxing e sottotipo): dopo aver applicato una conversione boxing da **int** a `Integer`, poiché `Integer` $\leq$ `Number`, `Integer` $\leq$ `Integer` e `Integer` $\not\leq$ `Number[]`, solo i metodi `m(Number)` e `m(Integer)` sono applicabili per boxing e reference widening (quest'ultimo richiesto solo nel caso `m(Number)`) e poiché `Integer` $\leq$ `Number`, 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`, la chiamata **super**`.m(i)` si comporta analogamente al caso illustrato al punto (a); viene quindi stampata la stringa `"P.m(Number) H.m(Integer)"`.

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

- primo tentativo (solo sottotipo): poiché nessun metodo ha due parametri (al primo e al secondo tentativo `m(Number... o)` viene trattato come metodo con un solo parametro), non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): continuano a non esistere metodi di `P` accessibili e applicabili, anche dopo una conversione boxing da **int** a `Integer`, poiché nessun metodo ha due parametri;
- terzo tentativo (metodi con numero variabile di parametri): `m(Number... o)` viene trattato come metodo con numero variabile di parametri di tipo `Number`; dopo una conversione boxing da **int** a `Integer`, poiché `Integer` $\leq$ `Number`, il metodo `m(Number... o)` è l'unico 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` 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 dato che l'unico metodo con due parametri (al primo e al secondo tentativo `m(Number... o)` viene trattato come metodo con un solo parametro) ha segnatura `m(Integer,Integer)` e **int** $\not\leq$ `Integer`;
- secondo tentativo (boxing/unboxing e sottotipo): dopo una conversione boxing da **int** a `Integer`, `m(Integer,Integer)` è il solo metodo applicabile per boxing e sottotipo.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(Integer,Integer)`; poiché i parametri `i1` e `i2` hanno tipo statico `Integer` e **super** si riferisce alla classe `P`, la chiamata **super**`.m(i1, i2)` si comporta analogamente al caso illustrato al punto (d); viene quindi stampata la stringa `"P.m(Number...) H.m(Integer,Integer)"`.