Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta parziale del 12 febbraio 2018

a.a. 2017/2018

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("(0[bB][0-1]+)|(0|[1-9][0-9]*)|(\\s+)");
7        Matcher m = regEx.matcher("0b1 0");
8        m.lookingAt();
9        assert m.group(1).equals("0b1");
10       assert m.group(0) != null;
11       m.region(m.end(), m.regionEnd());
12       m.lookingAt();
13       assert !m.group(3).equals("0");
14       assert m.group(1) == null;
15       m.region(m.end(), m.regionEnd());
16       m.lookingAt();
17       assert m.group(2).equals("0");
18       assert m.group(0).length() == 1;
19    }
20 }
```

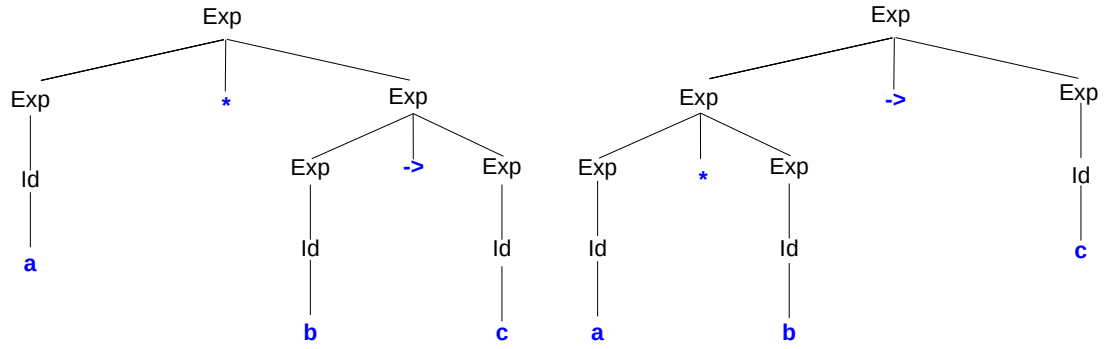
Soluzione:

- **assert** `m.group(1).equals("0b1");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `0b1 0` 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 `0b1` (stringa che inizia con `0b` o `0B` e continua con una o più cifre binarie, appartenente al gruppo di indice 1 e 0), quindi l'asserzione ha successo;
- **assert** `m.group(0) != null;` (linea 10): lo stato del matcher non è cambiato rispetto alla linea precedente, il gruppo 0 corrisponde a tutta l'espressione regolare, quindi per i motivi del punto precedente l'asserzione ha successo;
- **assert** `!m.group(3).equals("0");` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `0b1` (ossia uno spazio bianco) e l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene alla sotto-espressione regolare corrispondente al gruppo 3 e 0, quindi `m.group(3)` restituisce una stringa di spazi bianchi che è diversa dalla stringa `0` e l'asserzione ha successo;
- **assert** `m.group(1) == null;` (linea 14): lo stato del matcher non è cambiato rispetto alla linea precedente, il gruppo 1 corrisponde a una diversa sotto-espressione, quindi per i motivi del punto precedente `m.group(1)` restituisce **`null`** e l'asserzione ha successo;
- **assert** `m.group(2).equals("0");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `0`) e l'invocazione del metodo `lookingAt()` ha successo poiché la stringa `0` appartiene alla sotto-espressione regolare corrispondente al gruppo 2 (ma non a quella di gruppo 1); per tale motivo, `m.group(0)` restituisce tale stringa e l'asserzione ha successo;
- **assert** `m.group(0).length() == 1;` (linea 18): lo stato del matcher non è cambiato rispetto alla linea precedente, la stringa correntemente riconosciuta ha lunghezza 1, quindi l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Type  ::= Type * Type  | Type -> Type  | ( Type ) | Id
Id    ::= a | b | c
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `a * b -> c`



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Type` **resti invariato**.

Soluzione: Una possibile soluzione consiste nell'aggiunta dei non-terminali `Prod` e `Atom` per poter attribuire precedenza all'operatore `*`; inoltre, viene imposta l'associatività a sinistra per l'operatore `*` e a destra per `->`.

```
Type ::= Prod -> Type | Prod
Prod ::= Prod * Atom | Atom
Atom ::= ( Type ) | Id
Id ::= a | b | c
```

2. Sia `insert_after : ('a -> bool) -> 'a -> 'a list -> 'a list` la funzione così specificata: `insert_after` p e l aggiunge l'elemento e nella lista l immediatamente dopo ogni suo elemento che soddisfa il predicato p. Esempio:

```
# insert_after (fun x->x>3) 0 []
- : int list = []

# insert_after (fun x->x>3) 0 [1;4;2;5]
- : int list = [1;4;0;2;5;0]
```

   (a) Definire la funzione `insert_after` senza uso di parametri di accumulazione.

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

   (c) Definire la funzione `insert_after` come specializzazione della funzione `it_list` così definita:

```
let rec it_list f a = function x::l -> it_list f (f a x) l | _ -> a;;
val it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

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(float f) {
        return "P.m(float)";
    }
}
public class H extends P {
    String m(double d) {
        return super.m(d) + " H.m(double)";
    }

    String m(int i) {
        return super.m(i) + " 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(42.0)`

(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`, quindi entrambi i metodi di `P` sono accessibili e applicabili per sottotipo poiché **`int`** $\leq$ **`float, double`**, ma il metodo con segnatura `m(`**`float`**`)` è più specifico del metodo con segnatura `m(`**`double`**`)` poiché **`float`** $\leq$ **`double`**. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(`**`float`**`)` in `P`.

Viene stampata la stringa `"P.m(float)"`.

(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`, quindi viene eseguito il metodo con segnatura `m(`**`float`**`)` ereditato da `H`.

Viene stampata la stringa `"P.m(float)"`.

(c) L'espressione 42 ha tipo statico int, mentre il tipo statico di h è H; tutti e tre i metodi di segnatura m(double), m(float) e m(int) sono accessibili e applicabili per sottotipo, ma quello con segnatura `m(`**`int`**`)` è il più specifico poiché **`int`** $\leq$ **`float`** $\leq$ **`double`**. A runtime `h` contiene un'istanza di `H`, quindi viene eseguito il metodo di segnatura `m(`**`int`**`)` in `H`; l'invocazione **`super`**`.m(i)` viene risolta come al punto (a) poiché `i` ha tipo statico **`int`**, quindi viene stampata la stringa `"P.m(float) H.m(int)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`, quindi il solo metodo accessibile in `P` e applicabile per sottotipo ha segnatura `m(`**`double`**`)` pouché **`double`** $\not\leq$ **`float`**.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(`**`double`**`)` in `P`.

Viene stampata la stringa `"P.m(double)"`.

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo con segnatura `m(`**`double`**`)` ridefinito in `H`; l'invocazione **`super`**`.m(d)` viene risolta come al punto precedente poiché `d` ha tipo statico **`double`**, quindi viene stampata la stringa `"P.m(double) H.m(double)"`.

(f) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`, quindi solo il metodo di `H` con segnatura `m(`**`double`**`)` è accessibile e applicabile per sottotipo infatti **`double`** $\not\leq$ **`int,float`**. A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso di quello al punto precedente e viene stampata la stringa `"P.m(double) H.m(double)"`.