Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 10 settembre 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  Pattern regEx = Pattern.compile("(\\s+)|([a-z][_a-zA-Z]*)|(True|False)");
2  Matcher m = regEx.matcher("is_False True");
3  m.lookingAt();
4  assert m.group(2) == null;
5  assert m.group(0).equals("is");
6  m.region(m.end(), m.regionEnd());
7  m.lookingAt();
8  assert m.group(1) == null;
9  assert m.group(0).equals("");
10 m.region(m.end(), m.regionEnd());
11 m.lookingAt();
12 assert m.group(3) == null;
13 assert m.group(0).equals("False");
```

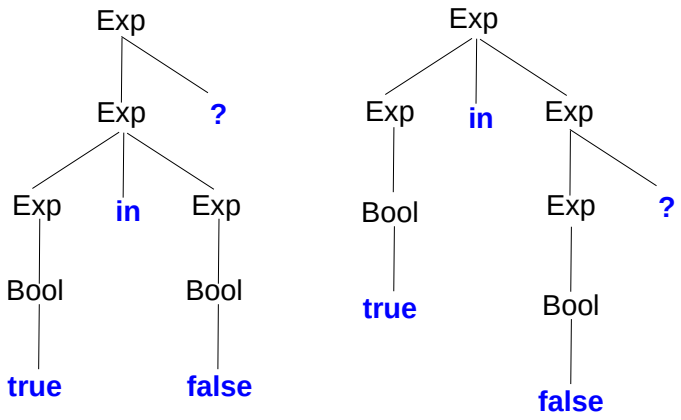
Soluzione:

- `assert m.group(2) == null;` (linea 4): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `is_False True` 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 `is_False` (appartenente ai soli gruppi 0 e 2) (qualsiasi stringa non vuota che inizia con una lettera minuscola seguita da zero o più lettere maiuscole, minuscole o _), quindi il metodo restituisce un oggetto non `null` e l'asserzione fallisce;
- `assert m.group(0).equals("is");` (linea 5): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(0)` restituisce un oggetto corrispondente alla stringa `is_False` e l'asserzione fallisce;
- `assert m.group(1) == null;` (linea 8): alla linea 6 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `is_False` (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 1, quindi `m.group(1)` restituisce un oggeto diverso da `null` e l'asserzione fallisce;
- `assert m.group(0).equals("");` (linea 9): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(0)` restituisce un oggetto corrispondente alla stringa `" "` e l'asserzione fallisce;
- `assert m.group(3) == null;` (linea 12): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `T`) e l'invocazione del metodo `lookingAt()` restituisce `true` poiché la stringa `True` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 3 (stringa `True` o `False`); per tale motivo, `m.group(0)` restituisce un oggetto diverso da `null` e l'asserzione fallisce;
- `assert m.group(0).equals("False");` (linea 13): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi precedenti la chiamata `m.group(0)` restituisce un oggetto corrispondente alla stringa `True`, quindi l'asserzione fallisce.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Exp ? | Exp in Exp | ( Exp ) | Bool
Bool ::= false | true
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `true in false?`



(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 `Quest` per poter attribuire la precedenza all'operatore unario `?` e forzare l'associatività (a sinistra) dell'operatore binario `in`.

```
Exp ::= Exp in Quest | Quest
Quest ::= Quest ? | ( Exp ) | Bool
Bool ::= false | true
```

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

   `filter_map` p f l restituisce la lista ottenuta da l eliminando gli elementi che non soddisfano il predicato p e applicando, nell'ordine, la funzione f ai restanti.

   Esempio:

```
# filter_map (fun x->x>=0.0) sqrt [-1.0;0.0;-4.0;4.0];;
- : float list = [0.0; 2.0]
```

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

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. (a) Completare le classi `IntLit` e `Add` che rappresentano i nodi di un albero della sintassi astratta corrispondenti, rispettivamente, a literal interi e all'operazione aritmetica di addizione.

```
public interface AST { <T> T accept(Visitor<T> v); }

public interface Visitor<T> {
   T visitIntLit(int i);
   T visitAdd(AST left, AST right);
}

public class IntLit implements AST {
   private final int value;
   public IntLit(int value) { /* completare */  }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

```
public class Add implements AST {
   private final AST left, right;
   public Add(AST left, AST right) { /* completare */ }
   public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

(b) Completare le classi `Eval` e `ToString` che implementano visitor su oggetti di tipo `AST`.

```
/* Un visitor Eval restituisce il valore dell'espressione visitata,
   calcolato secondo le regole convenzionali */
public class Eval implements Visitor<Integer> {
   public Integer visitIntLit(int i) { /* completare */}
   public Integer visitAdd(AST left, AST right) { /* completare */}
}

/* Un visitor ToString restituisce la stringa che rappresenta l'espressione visitata
   secondo la sintassi convenzionale senza parentesi */
public class ToString implements Visitor<String> {
   public String visitIntLit(int i) { /* completare */}
   public String visitAdd(AST left, AST right) { /* completare */}
}

// Classe di prova
public class Test {
   public static void main(String[] args) {
        AST i1 = new IntLit(1), i2 = new IntLit(2), i3 = new IntLit(3);
        AST i1_plus_i2_plus_i3 = new Add(new Add(i1, i2), i3);
        assert i1_plus_i2_plus_i3.accept(new Eval()) == 6;
        assert i1_plus_i2_plus_i3.accept(new ToString()).equals("1+2+3");
   }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(long l) { return "P.m(long)"; }
    String m(int i) { return "P.m(int)"; }
}
public class H extends P {
    String m(long l) { return super.m(l) + " H.m(long)"; }
    String m(Integer i) { return super.m(i) + " H.m(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(42L)`

(e) `p2.m(42L)`

(f) `h.m(42L)`

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, ma `m(int)` è più specifico poiché **`int`** $\leq$ **`long`**. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(int)` in `P`.

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

(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(int)` ereditato da `H` e viene stampata la stringa `"P.m(int)"`.

(c) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`, i metodi applicabili per sotto-tipo sono gli stessi dei due punti precedenti visto che **`int`**$\not\leq$`Integer`, quindi viene selezionato il metodo `m(int)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa `"P.m(int)"`.

(d) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `p` è `P`, quindi `m(long)` è l'unico metodo accessibile e applicabile per sotto-tipo (dato che **`long`** $\not\leq$ **`int`**).

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

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

(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`, quindi viene eseguito il metodo con segnatura `m(long)` ridefinito in `H`; l'invocazione **`super.`**`m(l)` viene risolta come al punto precedente poiché `l` ha tipo statico **`long`**; viene stampata la stringa `"P.m(long) H.m(long)"`.

(f) Il literal `42L` ha tipo statico **`long`** e il tipo statico di `h` è `H`, il metodo applicabile per sotto-tipo ha segnatura `m(long)` come per i due punti precedenti visto che **`long`**$\not\leq$`Integer,`**`int`**.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi il comportamento è lo stesso del punto precedente; viene stampata la stringa `"P.m(long) H.m(long)"`.