Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 24 gennaio 2019

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("(String|Float)|(\\s+)|([a-z][\\-a-zA-Z]*)");
2  Matcher m = regEx.matcher("is-Float String");
3  m.lookingAt();
4  assert m.group(3) != null;
5  assert m.group(0).equals("is-Float");
6  m.region(m.end(), m.regionEnd());
7  m.lookingAt();
8  assert m.group(2) != null;
9  assert m.group(1) == null;
10 m.region(m.end(), m.regionEnd());
11 m.lookingAt();
12 assert m.group(3) != null;
13 assert m.group(0).equals("Float");
```

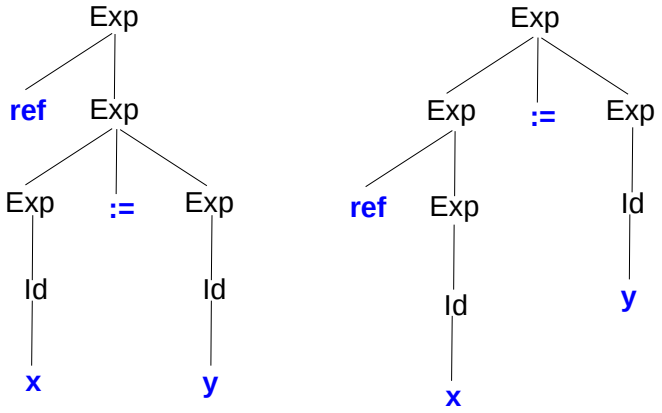
Soluzione:

- `assert m.group(3) != null;` (linea 4): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `is-Float String` 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-Float` (appartenente ai soli gruppi 0 e 3: 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 ha successo;
- `assert m.group(0).equals("is-Float");` (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-Float` e l'asserzione ha successo;
- `assert m.group(2) != null;` (linea 8): alla linea 6 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `is-Float` (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 2, quindi `m.group(2)` restituisce un oggeto diverso da **`null`** e l'asserzione ha successo;
- `assert m.group(1) == null;` (linea 9): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(1)` restituisce **`null`** e l'asserzione ha successo;
- `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 `S`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `String` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 1 (stringa `String` o `Float`); per tale motivo, `m.group(3)` restituisce **`null`** e l'asserzione fallisce;
- `assert m.group(0).equals("Float");` (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 `String`, quindi l'asserzione fallisce.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= ref Exp | Exp := Exp | ( Exp ) | Id
Id ::= x | y
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `ref x := y`



(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 `Ref` per poter attribuire la precedenza all'operatore unario `ref` e forzare l'associatività (a sinistra) dell'operatore binario `:=`.

```
Exp  ::= Exp := Ref | Ref
Ref ::= ref Ref | ( Exp ) | Id
Id ::= x | y
```

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

   `cond_map` f p l restituisce la lista ottenuta da l applicando, nell'ordine, la funzione f agli elementi di l che soddisfano il predicato p e lasciando invariati i restanti.

   Esempio:

```
# cond_map  sqrt (fun x->x>=0.0) [-1.0;9.0;-4.0;4.0]
- : float list = [-1.0; 3.0; -4.0; 2.0]
```

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

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

   (c) Definire `cond_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 `BoolLit` e `And` che rappresentano i nodi di un albero della sintassi astratta corrispondenti, rispettivamente, a literal booleani e all'and logico.

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

public interface Visitor<T> {
   T visitBoolLit(boolean b);
   T visitAnd(AST left, AST right);
}

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

```
public class And implements AST {
   private final AST left, right;
   public And(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,
   calcolato secondo le regole convenzionali;
   la valutazione dell'and logico e' con short-circuit */
public class Eval implements Visitor<Boolean> {
   public Boolean visitBoolLit(boolean b) { /* completare */  }
   public Boolean visitAnd(AST left, AST right) { /* completare */  }
}
/* Un visitor ToString restituisce la rappresentazione in
   notazione polacca postfissa dell'espressione;
   usare String.valueOf per la conversione da boolean a String */
public class ToString implements Visitor<String> {
   public String visitBoolLit(boolean b) { /* completare */  }
   public String visitAnd(AST left, AST right) { /* completare */  }
}
// Classe di prova
public class Test {
   public static void main(String[] args) {
      AST b1 = new BoolLit(true), b2 = new BoolLit(true), b3 = new BoolLit(false);
      AST b1_b2_and_b3_and = new And(new And(b1, b2), b3);
      assert !b1_b2_and_b3_and.accept(new Eval());
      assert b1_b2_and_b3_and.accept(new ToString()).equals("true true && false &&");
   }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Number n) { return "P.m(Number)"; }
   String m(String s) { return "P.m(String)"; }
}
public class H extends P {
   String m(Number n) { return super.m(n) + " H.m(Number)"; }
   String m(String s) { return super.m(s) + " H.m(String)"; }
}
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)`

(e) `p2.m(42)`

(f) `h.m(42)`

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

(a) Il literal `"42"` ha tipo statico `String` e il tipo statico di `p` è `P`, quindi solo il metodo di `P` con segnature `m(String)` è accessibile e applicabile per sottotipo, dato che `String` $\not\leq$ `Number`. A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(String)` in `P`.

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

(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(String)` ridefinito in `H`; poiché il parametro `s` ha tipo statico `String`, la chiamata **`super`**`.m(s)` viene risolta come al punto precedente e viene invocato il metodo della classe `P` con segnatura `m(String)`; viene stampata la stringa `"P.m(String) H.m(String)"`.

(c) Il literal `"42"` ha tipo statico `String` e il tipo statico di `h` è `H`, l'unico metodo accessibile e applicabile per sotto-tipo ha segnatura `m(String)` per gli stessi motivi dei punti precedenti.

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

(d) Il literal `42` ha tipo statico **`int`** e il tipo statico di `p` è `P`, quindi non esistono metodi accessibili e applicabili per sottotipo, dato che **`int`** $\not\leq$ `Number, String`; dato che **`int`** può essere implicitamente convertito a `Integer` per boxing e che `Integer` $\leq$ `Number`, `Integer` $\not\leq$ `String`, l'unico metodo applicabile ha segnatura `m(Number)`.

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

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`, quindi viene eseguito il metodo con segnatura `m(Number)` ridefinito in `H`; l'invocazione **`super`**`.m(n)` viene risolta con il metodo con segnatura `m(Number)` poiché il parametro `n` ha tipo statico `Number`, `Number` $\leq$ `Number` e `Number` $\not\leq$ `String`; viene stampata la stringa `"P.m(Number) H.m(Number)"`.

(f) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`; dato che i metodi accessibili di `H` hanno le stesse segnature di quelli di `P`, la chiamata viene risolta come al punto precedente.

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(Number) H.m(Number)"`.