Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta, 20 gennaio 2022

a.a. 2020/2021

1. (a) Per ogni stringa elencata sotto stabilire, motivando la risposta, se appartiene alla seguente espressione regolare e, in caso affermativo, indicare il gruppo di appartenenza (escludendo il gruppo 0).
Nota bene: la notazione `(?: )` permette di usare le parentesi in un'espressione regolare **senza** definire un nuovo gruppo.

```
([a-z]+(?:\.[a-z]*)*)|(0[bB][0-1]+[lL]?)|(\s+)|(mv\.ACC|add\.SP|sub\.FM)
```

i. `"mv.ACC"`
ii. `"MV.acc"`
iii. `"0B"`
iv. `"b."`
v. `"0BL"`
vi. `"0B10L"`

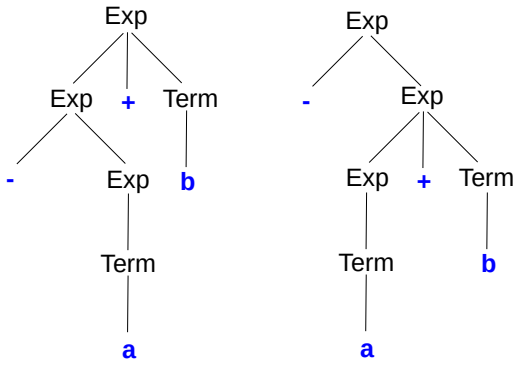
Soluzione:

i. Gli unici gruppi che corrispondono a stringhe che iniziano con `"mv."` sono il primo e il quarto, ma il carattere che segue il punto nel primo gruppo deve necessariamente essere una lettera minuscola, quindi la stringa appartiene al quarto gruppo.
ii. Nessun gruppo corrisponde a stringhe che possono iniziare con una lettera maiuscola, quindi la stringa non appartiene all'espressione regolare.
iii. L'unico gruppo che corrisponde a stringhe che iniziano con `"0B"` è il secondo, ma in tale gruppo la lettera `B` deve essere seguita da almeno una cifra binaria, quindi la stringa non appartiene all'espressione regolare.
iv. L'unico gruppo che corrisponde a stringhe che iniziano con la lettera `b` è il primo; la stringa appartiene a tale gruppo, dato che dopo il punto non devono necessariamente seguire altri caratteri.
v. La stringa non appartiene all'espressione regolare per gli stessi motivi del punto iii.
vi. A differenza dei punti iii e v, la stringa appartiene al secondo gruppo poiché la lettera `B` é seguita da due cifre binarie terminate dalla lettera opzionale `L`.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Term | Exp + Term | - Exp
Term ::= a | b | - Term | ( Exp )
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa, per esempio `-a+b`:



(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: La soluzione più semplice consiste nell'imporre la precedenza del `-` modificando la grammatica nel seguente modo:

```
Exp ::= Term | Exp + Term
Term ::= a | b | - Term | ( Exp )
```

2. Sia `sum_wise : int list -> int list -> int list` la funzione tale che

   `sum_wise` $[a_0;\ldots;a_n]$ $[b_0;\ldots;b_n]$ = $[a_0+b_0;\ldots;a_n+b_n]$

   e `sum_wise` l_1 l_2 solleva un'eccezione (usare `raise (Invalid_argument "sum_wise")`) se l_1 e l_2 non hanno la stessa lunghezza.

   Esempi:

   ```
   sum_wise [0;2;4] [1;3;5]=[1;5;9]
   sum_wise [] []=[]
   sum_wise [0] [1;3;5] solleva un'eccezione
   sum_wise [0;2;4] [1] solleva un'eccezione
   ```

   (a) Implementare `sum_wise` senza uso di parametri di accumulazione.

   (b) Implementare `sum_wise` usando un parametro di accumulazione affinché la ricorsione sia di coda.

   Soluzione: Vedere il file `soluzione.ml`.

3. Completare il seguente codice che implementa

   - gli alberi della sintassi astratta `AST` composti da nodi corrispondenti ai literal di tipo booleano (`BoolLit`) e all'operatore di congiunzione (`And`);
   - il visitor `Eval` che valuta l'espressione nel modo convenzionale restituendo un booleano come risultato.

   Esempio:

   ```
   var exp = new And(new BoolLit(true), new BoolLit(true));
   var v = new Eval();
   assert exp.accept(v); // true && true si valuta in true
   exp = new And(new BoolLit(true), new BoolLit(false));
   assert !exp.accept(v); // true && false si valuta in false
   ```

   Codice da completare:

   ```
   import static java.util.Objects.requireNonNull;
   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 b;
     public BoolLit(boolean b) {
       // completare
     }
     @Override
     public <T> T accept(Visitor<T> v) {
       // completare
     }
   }

   public class And implements AST {
     // invariant left != null && right !=null
     private final AST left;
     private final AST right;
     public And(AST left,AST right) {
       // completare
     }
     @Override
     public <T> T accept(Visitor<T> v) {
       // completare
     }
   }

   public class Eval implements Visitor<Boolean> {
     @Override
     public Boolean visitBoolLit(boolean b) {
   ```

```
    // completare
  }
  @Override
  public Boolean visitAnd(AST left, AST right) {
    // completare
  }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
  String m(float f1,float f2) { return "P.m(float,float)"; }
  String m(float[] f) { return "P.m(float[])"; }
}
public class H extends P {
  String m(double d1,double d2) { return "H.m(double,double)"; }
  String m(double[] d) { return "H.m(double[])"; }
}
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(42.0f,42.0f)`

(b) `p2.m(42.0f,42.0f)`

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

(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) I literal `42.0f` hanno tipo statico **`float`** e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): il metodo con solo un parametro non è applicabile, mentre quello con due è accessibile e applicabile per sottotipo, poiché **`float`** $\leq$ **`float`**; viene scelto l'unico metodo applicabile `m(`**`float,float`**`)` che è ovviamente anche il più specifico.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `m(`**`float,float`**`)` in `P` e viene stampata la stringa `"P.m(float,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`, ma poiché il metodo `m(`**`float,float`**`)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(float,float)"`.

(c) I literal `42.0f` hanno tipo statico **`float`** e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): i metodi `m` di `H` (sia quelli definiti nella classe, sia quelli ereditati da `P`) con un solo parametro non sono applicabili, mentre quelli con due sono entrambi accessibili e applicabili per sottotipo, dato che **`float`** $\leq$ **`float`** e **`float`** $\leq$ **`double`**; viene scelto il più specifico, ossia `m(`**`float,float`**`)` poiché **`float`** $\leq$ **`double`**.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(`**`float,float`**`)` ereditato da `P`; viene stampata la stringa `"P.m(float,float)"`.

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

- primo tentativo (solo sottotipo): il metodo con solo un parametro non è applicabile, mentre quello con due è accessibile e applicabile per sottotipo, poiché **`int`** $\leq$ **`float`**; viene scelto l'unico metodo applicabile `m(`**`float,float`**`)` che è ovviamente anche il più specifico.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `m(`**`float,float`**`)` in `P` e viene stampata la stringa `"P.m(float,float)"`.

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

(f) I literal `42` hanno tipo statico **`int`** e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): i metodi `m` di `H` (sia quelli definiti nella classe, sia quelli ereditati da `P`) con un solo parametro non sono applicabili, mentre quelli con due sono entrambi accessibili e applicabili per sottotipo, dato che **`int`** $\leq$ **`float`** e **`int`** $\leq$ **`double`**; viene scelto il più specifico, ossia `m(`**`float,float`**`)` poiché **`float`** $\leq$ **`double`**.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(`**`float,float`**`)` ereditato da `P`; viene stampata la stringa `"P.m(float,float)"`.