Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta parziale

16 febbraio 2022, a.a. 2021/2022

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.

   ```
   (0[bB](?:[0-1]*|[0-1]+[lL]?))|(\s+)|([a-z]+(?:\.[a-zA-Z\d]*)*)
   ```

   i. `"0B"`
   ii. `"0bL"`
   iii. `"a.b"`
   iv. `"a."`
   v. `"a.A3.."`
   vi. `"aA."`

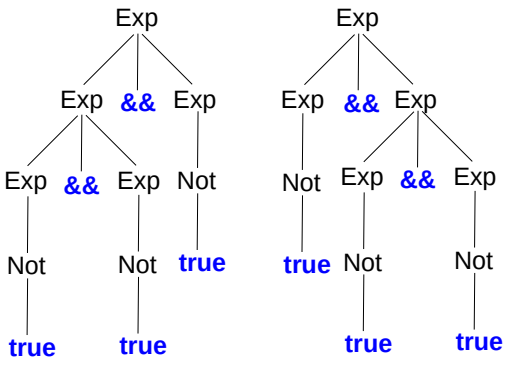
   Soluzione:

   i. L'unico gruppo che corrisponde a stringhe che iniziano con `"0"` seguito opzionalmente da `"B"` è il primo; poiché dopo `"B"` può seguire una sequenza vuota di cifre binarie, la stringa appartiene a tale gruppo.
   ii. L'unico gruppo che corrisponde a stringhe che iniziano con `"0"` seguito opzionalmente da `"b"` è il primo; tuttavia, la stringa `"L"` può comparire opzionalmente solo dopo una sequenza non vuota di cifre binarie, quindi la stringa non appartiene all'espressione regolare.
   iii. L'unico gruppo che corrisponde a stringhe che iniziano con `"a"` è il terzo; poiché possono seguire zero o più stringhe che iniziano con `"."` seguito da una sequenza anche vuota di caratteri alfanumerici, la stringa appartiene a tale gruppo.
   iv. La stringa appartiene al terzo gruppo per lo stesso motivo del punto precedente.
   v. La stringa appartiene al terzo gruppo per lo stesso motivo dei punti precedenti.
   vi. L'unico gruppo che corrisponde a stringhe che iniziano con `"a"` è il terzo; tuttavia, prima della stringa `"."` possono comparire solo lettere minuscole, quindi la stringa non appartiene all'espressione regolare.

   (b) Mostrare che la seguente grammatica è ambigua.

   ```
   Exp ::= Exp && Exp | Not
   Not ::= false | true | ! Not | ( Exp )
   ```

   Soluzione: Esistono due diversi alberi di derivazione per la stringa `true && true && true`:



   (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: È sufficiente modificare la produzione `Exp ::= Exp && Exp` per forzare l'associatività sintattica dell'operatore `&&`, per esempio a sinistra:

```
Exp ::= Exp && Not | Not
Not ::= false | true | ! Not | ( Exp )
```

2. Sia `select : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list` la funzione tale che

   `select` p `[`a_1`;...;`a_n`]` `[`b_1`;...;`b_n`]` `=` `[`c_1`;...;`c_n`]`

   dove per ogni $i = 1..n$, $c_i = a_i$ se $p\ a_i\ b_i$ è `true`, $c_i = b_i$ altrimenti; se l_1 e l_2 non hanno la stessa lunghezza, allora `select` $p\ l_1\ l_2$ solleva un'eccezione (usare `raise (Invalid_argument "select")`).

   Esempi:

   ```
   select ( < ) [0;2;5] [1;3;4]=[0;2;4];;
   select ( < ) [] []=[];;
   select ( < ) [0] [1;3;5] solleva un'eccezione
   select ( < ) [0;2;4] [1] solleva un'eccezione
   ```

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

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

   Soluzione: Vedere il file `soluzione.ml`.

3. Considerare le seguenti dichiarazioni di classi Java:

   ```
   public class P {
     String m(float f1, float f2) {return "P.m(float,float)";}
     String m(double[] d) {return "P.m(double[])";}
   }
   public class H extends P {
     String m(float f1, float f2) {return "H.m(float,float)";}
     String m(double d1, double d2) {return "H.m(double,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(42L,42L)`

   (b) `p2.m(42L,42L)`

   (c) `h.m(42L,42L)`

   (d) `p.m(42.0,42.0)`

   (e) `p2.m(42.0,42.0)`

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

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

(a) I literal `42L` hanno tipo statico `long` 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 `long` $\leq$ `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` e poiché il metodo `m(float,float)` è ridefinito in `H`, viene eseguito quest'ultimo e, quindi, viene stampata la stringa `"H.m(float,float)"`.

(c) I literal `42L` hanno tipo statico `long` e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): `H` eredita da `P` solamente il metodo `m` che non è comunque applicabile dato che ha solo un parametro; i metodi `m` di `H` hanno due parametri e sono entrambi accessibili e applicabili per sottotipo, dato che `long` $\leq$ `float` e `long` $\leq$ `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)` di `H`; viene stampata la stringa `"H.m(float,float)"`.

(d) I literal `42.0` hanno tipo statico `double` e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): il metodo con solo un parametro non è applicabile, ma neanche quello con due, poiché `double` $\not\leq$ `float`; viene scelto l'unico metodo applicabile `m(float,float)` che è ovviamente anche il più specifico.
- secondo e terzo tentativo: anche tali tentativi falliscono poiché non sono applicabili conversioni di tipo boxing/unboxing e non ci sono metodi con arità variabile, quindi ci sarà un errore di compilazione.

(e) Visto che i tipi statici sono gli stessi, ci sarà un errore di compilazione come nel caso precedente.

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

- primo tentativo (solo sottotipo): `H` eredita da `P` solamente il metodo `m` che non è comunque applicabile dato che ha solo un parametro; i metodi `m` di `H` hanno due parametri, ma l'unico che è sia accessibile sia applicabile per sottotipo è `m(double,double)` dato che `double` $\not\leq$ `float` e `double` $\leq$ `double`; viene scelto l'unico metodo applicabile `m(double,double)` che è ovviamente anche il più specifico.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `m(double,double)` di `H`; viene stampata la stringa `"H.m(double,double)"`.