Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 3 giugno 2021

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).

```
(0[0-7]+[lL]?)|([A-Za-z]+(?:\.[A-Za-z]+)*)|(\s+)|(\.mv|\.add|\.sub)
```

   i. `"a.b7"`
   ii. `"a."`
   iii. `"mv.sub"`
   iv. `".sub"`
   v. `"0L"`
   vi. `"00L"`

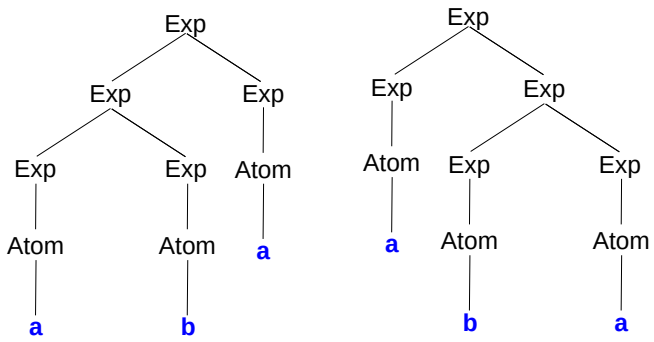
Soluzione:

   i. L'unico gruppo che definisce stringhe che iniziano con una lettera è il due, ma tali stringhe non contengono cifre decimali, quindi la stringa non appartiene all'espressione regolare.
   ii. L'unico gruppo che definisce stringhe che iniziano con una lettera seguita dal punto è il due, ma in tali stringhe il punto deve essere sempre seguito da almeno una lettera, quindi la stringa non appartiene all'espressione regolare.
   iii. Per motivazioni analoghe a quelle dei punti precedenti la stringa appartiene al gruppo due.
   iv. L'unico gruppo che definisce stringhe che iniziano con il punto è il quattro e, tra le varie opzioni della corrispondente espressione regolare c'è proprio la stringa in questione, che, quindi, appartiene al gruppo quattro.
   v. L'unico gruppo che definisce stringhe che iniziano con lo zero è l'uno, ma tali stringhe devono avere almeno un'altra cifra ottale, quindi la stringa non appartiene all'espressione regolare.
   vi. Per ragioni analoghe a quelle del punto precedente, la stringa appartiene al gruppo quattro.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp Exp | Atom
Atom ::=  a | b | ( Exp )
```

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



(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 nel forzare l'associatività sintattica dell'operatore binario sostituendo uno dei due `Exp` con `Atom` nella parte destra della produzione corrispondente. Per esempio, la seguente grammatica è equivalente a quella data, ma non è ambigua poiché per ogni espressione corretta esiste solo l'albero di derivazione corrispondente all'associatività da sinistra.

```
Exp ::= Exp Atom | Atom
Atom ::=  a | b | ( Exp )
```

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

   `fuse` f `[`x_1`; ... ;`x_k`] [`y_1`; ... ;`y_n`]` $=$ `[`$f\ x_1\ y_1$`; ... ;`$f\ x_m\ y_m$`]`, con $k, n \geq 0$ e $m = \min(n, k)$.

   Esempi:

   ```
   fuse max [1;2;3] [3;1;4] = [3;2;4]
   fuse max [1;2;3] [3] = [3]
   fuse max [4] [3;5;7] = [4]
   fuse (+) [1;2;3] [3;1;4] = [4;3;7]
   fuse (+) [1;2;3] [3] = [4]
   fuse (+) [4] [3;5;7] = [7]
   ```

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

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

   Soluzione: Vedere il file `soluzione.ml`.

3. Completare la seguente classe di iteratori `RangeIterator` per generare sequenze di interi da un estremo (`start`) incluso fino a un estremo (`end`) escluso con passo (`step`) diverso da zero.

   Esempio:

   ```
   for (var i : new RangeIterator(3)) // start=0 end=3 step=1
      System.out.println(i); // prints 0 1 2
   for (var i : new RangeIterator(3, -1)) // start=3 end=-1 step=1
      System.out.println(i); // no printed values
   for (var i : new RangeIterator(3, -1, -3)) // start=3 end=-1 step=-3
      System.out.println(i); // prints 3 0
   ```

   Codice da completare:

   ```
   import java.util.Iterator;
   import java.util.NoSuchElementException;

   class RangeIterator implements Iterator<Integer>, Iterable<Integer> {

      // dichiarare i campi mancanti
      // invariant step!=0

      // ranges from start (inclusive) to end (exclusive) with step!=0
      public RangeIterator(int start, int end, int step) {
         // completare
      }

      // ranges from start (inclusive) to end (exclusive) with step 1
      public RangeIterator(int start, int end) { this(start, end, 1); }

      // ranges from 0 (inclusive) to end (exclusive) with step 1
      public RangeIterator(int end) { this(0, end); }

      @Override
      public boolean hasNext() {
         // completare
      }

      @Override
      public Integer next() {
         // completare
      }

      @Override
      public Iterator<Integer> iterator() { return this; }
   }
   ```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Short s) { return "P.m(Short)"; }
   String m(Number n) { return "P.m(Number)"; }
}
public class H extends P {
   String m(short s) { return "H.m(short)"; }
   String m(float f) { return "H.m(float)"; }
}
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`.

- primo tentativo (solo sottotipo): poiché **`int`** $\not\leq$ `Short` e **`int`** $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): **`int`** può essere convertito a `Integer` mediante boxing; poiché `Integer` $\not\leq$ `Short` e `Integer` $\leq$ `Number`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

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

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

(c) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): tra i metodi ereditati da `P` e quelli definiti in `H`, solo `m(`**`float`**`)` è applicabile per sottotipo dato che **`int`** $\leq$ **`float`**, ma **`int`** $\not\leq$ **`short`**, **`int`** $\not\leq$ `Short`, **`int`** $\not\leq$ `Number`.

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

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché **`double`** $\not\leq$ `Short` e **`double`** $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): **`double`** può essere convertito a `Double` mediante boxing; poiché `Double` $\not\leq$ `Short` e `Double` $\leq$ `Number`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

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

(f) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): poiché **`double`** $\not\leq$ `Short`, **`double`** $\not\leq$ `Number`, **`double`** $\not\leq$ **`short`**, **`double`** $\not\leq$ **`float`** non esistono metodi ereditati da `P` o definiti in `H` accessibili e applicabili per sottotipo;
- secondo tentativo (boxing/unboxing e sottotipo): **`double`** può essere convertito a `Double` mediante boxing e `Double` $\leq$ `Number`, ma `Double` $\not\leq$ `Short`, `Double` $\not\leq$ **`short`**, `Double` $\not\leq$ **`float`**; quindi, tra i metodi ereditati da `P` e quelli definiti in `H`, solo il metodo `m(Number)` di `P` è applicabile per boxing e reference widening.

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