Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 12 luglio 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).

```
([0-9]+)|([A-Z][a-zA-Z0-9]*)|(move|store|load)|(\s+)
```

   i. `"Move"`
   ii. `"move42"`
   iii. `"store "`
   iv. `"000"`
   v. `"0a"`
   vi. `" "`

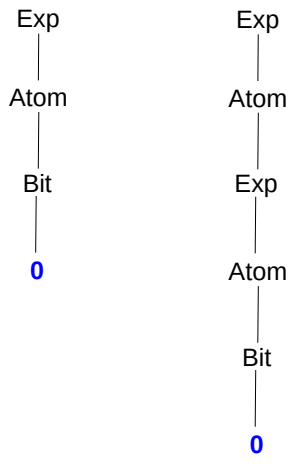
Soluzione:

   i. L' unico gruppo che corrisponde a stringhe che iniziano con una lettera maiuscola seguita da zero o più caratteri alfanumerici è il secondo, quindi la stringa appartiene a tale gruppo.
   ii. L'unico gruppo che corrisponde a stringhe che iniziano con una lettera minuscola è il terzo, ma in tal caso le stringhe non contengono cifre numeriche, quindi la stringa non appartiene all'espressione regolare.
   iii. L'unico gruppo che corrisponde a stringhe che iniziano con `"store"` è il terzo, ma in tal caso le stringhe non contengono spazi bianchi, quindi la stringa non appartiene all'espressione regolare.
   iv. L'unico gruppo che corrisponde a stringhe di una o più cifre numeriche è il primo, quindi la stringa appartiene a tale gruppo.
   v. L'unico gruppo che corrisponde a stringhe che iniziano con `"0"` è il primo, ma in tal caso le stringhe non contengono lettere, quindi la stringa non appartiene all'espressione regolare.
   vi. L'unico gruppo che corrisponde a stringhe di uno o più spazi bianchi è il quarto, quindi la stringa appartiene a tale gruppo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Atom + Exp | Atom - Exp | Atom
Atom ::=  Bit | ( Exp ) | Exp
Bit ::= 0 | 1
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa, per esempio `0`:



(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'eliminare la seguente produzione che risulta essere ridondante:

```
Atom ::= Exp
```

ottenendo così la grammatica equivalente (rispetto a `Exp`) definita qua sotto:

```
Exp ::= Atom + Exp | Atom - Exp | Atom
Atom ::=  Bit | ( Exp )
Bit ::= 0 | 1
```

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

   `swap [`(x_1, y_1)`; ... ;`(x_k, y_k)`]` $=$ `[`(y_1, x_1)`; ... ;`(y_k, x_k)`]`, con $k \geq 0$.

   Esempi:

   ```
   swap [(1,"one");(2,"two");(3,"three")] = [("one", 1); ("two", 2); ("three", 3)]
   swap []=[]
   ```

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

   (b) Definire `swap` usando `List.map:('a -> 'b) -> 'a list -> 'b list`.

   Soluzione: Vedere il file `soluzione.ml`.

3. Completare il seguente codice che implementa gli alberi di ricerca binari con nodi etichettati da numeri interi (con la relazione d'ordine usuale) e la ricerca di un elemento nell'albero tramite visitor pattern.

   Esempio:

   ```
   var t = new Node(10,new Leaf(5),new Node(42,new Leaf(31),null));
   assert t.accept(new Search(31)); // 31 è un'etichetta contenuta nell'albero t
   assert !t.accept(new Search(43)); // 43 non è un'etichetta contenuta nell'albero t
   assert !t.accept(new Search(30)); // 30 non è un'etichetta contenuta nell'albero t
   ```

   Definizione di albero binario di ricerca: Un albero binario di ricerca con nodi etichettati da numeri interi soddisfa le seguenti proprietà: ogni nodo v è etichettato da un numero intero i, le etichette dei nodi del sottoalbero sinistro di v sono $\leq i$, le etichette dei nodi del sottoalbero destro di v sono $\geq i$.

   Completare costruttori e metodi delle classi `Leaf`, `Node` e `Search`:

   ```
   public interface Visitor<T> {
      T visitLeaf(int value);
      T visitNode(int value, BSTree left, BSTree right);
   }

   public abstract class BSTree {
      protected final int value;
      protected BSTree(int value) { this.value = value; }
      public abstract <T> T accept(Visitor<T> v);
   }

   public class Leaf extends BSTree { // nodi foglia
      public Leaf(int value) { /* completare */ }
      @Override public <T> T accept(Visitor<T> v) { /* completare */ }
   }

   public class Node extends BSTree { // nodi interni
      private final BSTree left, right; // left, right possono contenere null, ma non entrambi
      public Node(int value, BSTree left, BSTree right) { /* completare */ }
      @Override public <T> T accept(Visitor<T> v) { /* completare */ }
   }

   public class Search implements Visitor<Boolean> { // ricerca l'elemento searchedValue nell'albero
      private final int searchedValue; // elemento da cercare
      public Search(int searchedValue) { /* completare */ }
      @Override public Boolean visitLeaf(int value) { /* completare */ }
      @Override public Boolean visitNode(int value, BSTree left, BSTree right) { /* completare */ }
   }
   ```

   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(long[] la) { return "P.m(long[])"; }
}
public class H extends P {
   String m(int i) { return "H.m(int)"; }
   String m(int[] ia) { return "H.m(int[])"; }
}
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(new long[]{42L, 24L})`

(e) `p2.m(new long[]{42L, 24L})`

(f) `h.m(new long[]{42L, 24L})`

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`** $\leq$ **`long`** e **`int`** $\not\leq$ **`long`**`[]`, viene scelto il metodo `m(`**`long`**`)` di `P` che è l'unico accessibile e applicabile per sottotipo.

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

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

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

- primo tentativo (solo sottotipo): poiché **`int`** $\leq$ **`int`**, **`int`** $\leq$ **`long`** e **`int`** $\not\leq$ **`int`**`[]`, **`int`** $\not\leq$ **`long`**`[]`, gli unici metodi accessibili e applicabili per sottotipo sono `m(`**`int`**`)` definito in `H` e `m(`**`long`**`)` ereditato da `P`; viene scelto il metodo più specifico `m(`**`int`**`)` poiché **`int`** $\leq$ **`long`**.

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

(d) Il literal `42L` ha tipo **`long`** e l'espressione **`new long`**`[]{42L, 24L}` ha tipo statico **`long`**`[]`, mentre il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché **`long`**`[]` $\leq$ **`long`**`[]` e **`long`**`[]` $\not\leq$ **`long`**, l'unico metodo di `P` applicabile per sottotipo è `m(`**`long`**`[])`;

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

(f) Il literal `42L` ha tipo **`long`** e l'espressione **`new long`**`[]{42L, 24L}` ha tipo statico **`long`**`[]`, mentre il tipo statico di `h` è `H`.

- primo tentativo (solo sottotipo): poiché **`long`**`[]` $\not\leq$ **`int`** e **`long`**`[]` $\not\leq$ **`int`**`[]`, l'unico metodo accessibile e applicabile per sottotipo è `m(`**`long`**`[])` ereditato da `P` come spiegato nei due punti precedenti.

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(long[])"`.