Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta, 8 settembre 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).

```
([a-z][a-z0-9]*(?:\.[a-z][a-z0-9]*)*)|([0-9]+(?:\.[0-9]*)?)|(\s+)
```

Nota bene: la notazione `(?: )` permette di usare le parentesi in un'espressione regolare **senza** definire un nuovo gruppo.

i. `"a.0"`
ii. `".4"`
iii. `"42."`
iv. `"a2.b3"`
v. `"xy."`
vi. `" "`

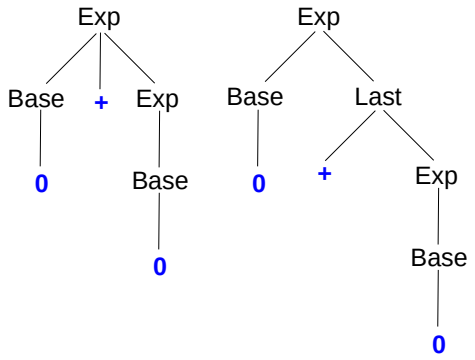
Soluzione:

i. L'unico gruppo che corrisponde a stringhe che iniziano con una o più lettere minuscole è il primo, ma dopo il punto deve necessariamente seguire una lettera minuscola, quindi la stringa non appartiene all'espressione regolare.
ii. Nessun gruppo corrisponde a stringhe che possono iniziare con il punto, quindi la stringa non appartiene all'espressione regolare.
iii. L'unico gruppo che corrisponde a stringhe che iniziano con una o più cifre numeriche è il secondo; in tale gruppo le cifre numeriche possono essere opzionalmente seguite dal punto e da una stringa numerica di lunghezza arbitraria, quindi la stringa appartiene a esso.
iv. L'unico gruppo che corrisponde a stringhe che iniziano con una lettera minuscola seguita da una cifra numerica è il primo; in tale gruppo i caratteri alfa-numerici possono essere seguiti opzionalmente da un punto, seguito da una lettera minuscola e da una stringa di lunghezza arbitraria composta da lettere minuscole e cifre numeriche, quindi la stringa appartiene a esso.
v. La stringa non appartiene all'espressione regolare per gli stessi motivi del punto i.
vi. L'unico gruppo che corrisponde a stringhe contenenti soli spazi bianchi è il terzo, quindi la stringa appartiene a tale gruppo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Base + Exp | Base Last | Base
Last ::= + Exp
Base ::= 0 | 1 | ( Exp )
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa, per esempio `0+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 le seguenti produzioni che risultano essere ridondanti:

```
Exp ::= Base Last
Last ::= + Exp
```

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

```
Exp ::= Base + Exp | Base
Base ::= 0 | 1 | ( Exp )
```

2. Sia `find : 'a -> 'a list -> int` la funzione tale che `find` e l restituisce il numero di volte con cui si ripete l'elemento e nella lista l.

   Esempi:

```
find "a" ["a";"c";"c"] = 1
find "c" ["a";"c";"c"] = 2
find "c" ["a";"b";"c"] = 1
find "d" ["a";"b";"c"] = 0
```

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

   (b) Implementare `find` 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 `FSTree` che rappresentano la struttura gerarchica di un file system con nodi di tipo `File` e attributo `size` (la dimensione del file) e nodi di tipo `Folder` e attributo `children` (la lista dei file e dei sotto-folder contenuti nel folder);
   - il visitor `FilesGreater` che conta i file di dimensione maggiore dell'attributo `minSize` che corrispondono al nodo visitato (se di tipo `File`) o a tutti i suoi discendenti (se di tipo `Folder`).

   Esempio:

```
var dir = new Folder(new File(10), new Folder(new File(2), new File(21)), new File(5), new File(42));
assert dir.accept(new FilesGreater(0)) == 5; // dir e il suo sotto-folder contengono 5 file con size>0
assert dir.accept(new FilesGreater(20)) == 2; // dir e il suo sotto-folder contengono 2 file con size>20
assert dir.accept(new FilesGreater(42)) == 0; // dir e il suo sotto-folder contengono 0 file con size>42
var f = new File(35);
assert f.accept(new FilesGreater(30)) == 1; // f ha size>30
assert f.accept(new FilesGreater(40)) == 0; // f non ha size>40
```

   Completare costruttori e metodi delle classi `File`, `Folder` e `FilesGreater`:

```
import java.util.List;

public interface Visitor<T> {
  T visitFile(int size);
  T visitFolder(List<FSTree> children);
}

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

public class File implements FSTree {
  private int size; // invariant size >= 0
  public File(int size) { /* completare */ }
  @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}

import java.util.List;
import java.util.LinkedList;

public class Folder implements FSTree {
  private final List<FSTree> children = new LinkedList<>();
  public Folder(FSTree... children) { /* completare */ }
  @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

```
import java.util.List;

public class FilesGreater implements Visitor<Integer> {
  // conta tutti i file con size > minSize
  private final int minSize; // puo' essere negativo
  public FilesGreater(int minSize) { /* completare */ }
  @Override public Integer visitFile(int size) { /* completare */ }
  @Override public Integer visitFolder(List<FSTree> children) { /* completare */ }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
  String m(double d1, double d2) { return "P.m(double,double)"; }
  String m(float f1, float f2) { return "P.m(float,float)"; }
}
public class H extends P {
  String m(long l1, long l2) { return "H.m(long,long)"; }
  String m(int i1, int i2) { return "H.m(int,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, 42)`

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

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

(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) Il literal `42` ha tipo statico **`int`** e il tipo statico di `p` è `P`.

- primo tentativo (solo sottotipo): poiché entrambi i metodi `m` di `P` hanno due parametri e **`int`** $\leq$ **`float`** $\leq$ **`double`** tutti e due sono accessibili e applicabili per sottotipo; viene scelto il più specifico, ossia `m(`**`float,float`**`)` poiché **`float`** $\leq$ **`double`**.

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

- primo tentativo (solo sottotipo): poiché tutti i metodi `m` di `H` (sia quelli definiti nella classe, sia quelli ereditati da `P`) hanno due parametri e **`int`** $\leq$ **`int`** $\leq$ **`long`** $\leq$ **`float`** $\leq$ **`double`** tutti sono accessibili e applicabili per sottotipo; viene scelto il più specifico, ossia `m(`**`int,int`**`)` poiché **`int`** $\leq$ **`long`** $\leq$ **`float`** $\leq$ **`double`**.

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

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

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

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

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

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

- primo tentativo (solo sottotipo): tra i quattro metodi `m` di `H` (sia quelli definiti nella classe, sia quelli ereditati da `P`) l'unico accessibile e applicabile per sottotipo è `m(double,double)`, dato che **`double`** $\leq$ **`double`**, **`double`** $\not\leq$ **`int`**, **`double`** $\not\leq$ **`long`** e **`double`** $\not\leq$ **`float`**.

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(double,double)"`.