Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 25 gennaio

a.a. 2012/2013

26 gennaio 2013

1. (a) Data la seguente linea di codice Java

```
Pattern p = Pattern.compile("([1-9][0-9]*|0)(\\.([0-9]*[1-9])?)?");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

Le stringhe definite dall'espressione sono quelle generate da `([1-9][0-9]*|0)`, ossia tutte le sequenze non vuote di cifre numeriche che non iniziano per 0 se la loro lunghezza è maggiore di 1, opzionalmente seguite da una stringa generata da `(\\.([0-9]*[1-9])?)?`, ossia una stringa che inizia con `.` opzionalmente seguita da una sequenza non vuota di cifre numeriche che non terminano per 0.

i. `assert p.matcher("13.0.9").matches();` fallisce, solo un'occorrenza di `.` è ammessa

ii. `assert p.matcher("03.").matches();` fallisce, la parte intera non può iniziare con 0 se ha più di una cifra

iii. `assert p.matcher("150.0").matches();` fallisce, la parte frazionari non può terminare con 0

iv. `assert p.matcher("00").matches();` fallisce, la parte intera non può iniziare con 0 se ha più di una cifra

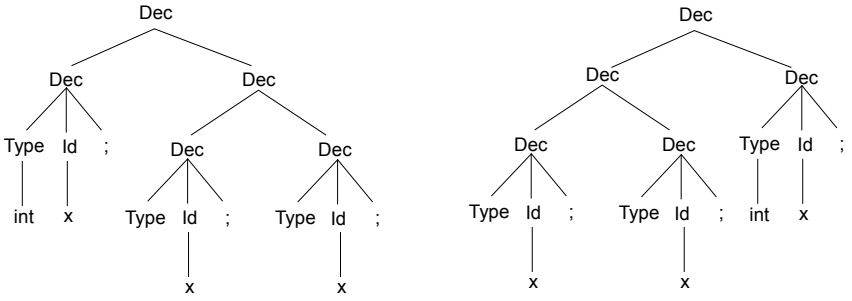
v. `assert p.matcher("3.").matches();` ha successo

vi. `assert p.matcher(".30").matches();` fallisce, la parte intera non può essere vuota

(b) Mostrare che la seguente grammatica è ambigua.

```
Dec  ::= Type Id ; | Dec Dec
Id ::= x | y | z
Type ::= int | bool
```

Basta mostrare due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `int x; int x; int x;`



(c) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Dec` **rimanga lo stesso**.

```
Dec  ::= Type Id ; | Type Id ; Dec
Id ::= x | y | z
Type ::= int | bool
```

2. Vedere il file `soluzione.ml`

3.
```
public class P {
    public String m(Object o) {
        return "P.m(Object)";
    }
    public String m(int i) {
        return "P.m(int)";
    }
}
public class H extends P {
    public String m(Object o) {
        return "H.m(Object)";
    }
    public String m(int i) {
        return super.m(i) + '\n' + "H.m(int)";
    }
    public String m(Number... n) {
        return super.m(n) + '\n' + "H.m(Number...)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p1 = new P();
        H h = new H();
        P p2 = h;
        System.out.println(...);
    }
}
```

(a) `p1.m(1)`: `p1` ha tipo statico `P`. Tutti i metodi `m` di `P` sono **`public`** e quindi accessibili. L'invocazione è corretta staticamente, infatti ha un argomento di tipo statico **`int`**, quindi c'è un'unica versione applicabile per sottotipo, quella con segnatura `m(int)`, che è anche appropriata.

A run-time `p1` contiene un oggetto della classe `P`, quindi il metodo invocato è quello in `P` con segnatura `m(int)`. Viene stampata la stringa

```
P.m(int)
```

(b) `h.m(1)`: `h` ha tipo statico `H`. Tutti i metodi `m` di `H` sono **`public`** e quindi accessibili. L'invocazione è corretta staticamente, infatti ha un argomento di tipo statico **`int`**, quindi c'è un'unica versione applicabile per sottotipo, quella con segnatura `m(int)`, che è anche appropriata.

A run-time `h` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` con segnatura `m(int)`. Nel body del metodo l'invocazione **`super`**`.m(i)` è staticamente corretta: **`super`** corrisponde alla classe `P`, tutti i metodi `m` di `P` sono **`public`** e quindi accessibili; l'argomento `i` ha tipo statico **`int`** quindi l'unica versione applicabile è quella con segnatura `m(int)` che è anche appropriata. Viene eseguito il metodo di `P` con segnatura `m(int)`, la stringa restituita viene concatenata con `'\n'` e `"H.m(int)"`, quindi viene stampato

```
P.m(int)
H.m(int)
```

(c) `h.m(new Integer(1))`: `h` ha tipo statico `H`. Tutti i metodi `m` di `H` sono **`public`** e quindi accessibili. L'invocazione è corretta staticamente, infatti ha un argomento di tipo statico `Integer`, quindi c'è una sola versione applicabile per sottotipo, quella con segnatura `m(Object)` che è anche appropriata.

A run-time `h` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` con segnatura `m(Object)`. Viene stampata la stringa

```
H.m(Object)
```

(d) `h.m(3.2)`: `h` ha tipo statico `H`. Tutti i metodi `m` di `H` sono **`public`** e quindi accessibili. L'invocazione è corretta staticamente, infatti ha un argomento di tipo statico **`double`**, non ci sono versioni applicabili per sottotipo, ma nella seconda fase è possibile applicare boxing conversion e poi reference widening (`Double` $\leq$ `Object`), quindi c'è un'unica versione applicabile, quella con segnatura `m(Object)`.

A run-time il comportamento è analogo al caso precedente, quindi viene stampata la stringa

```
H.m(Object)
```

(e) `h.m(1,2)`: `h` ha tipo statico `H`. Tutti i metodi `m` di `H` sono **`public`** e quindi accessibili. L'invocazione è corretta staticamente, infatti ha due argomenti di tipo statico **`int`**, nella prima e seconda fase non esistono metodi applicabili, mentre nella terza fase la versione `m(Number...)` è considerata con numero variabile di parametri, quindi è l'unica applicabile.

A run-time `h` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` con segnatura `m(Number...)`. Nel body del metodo l'invocazione **`super`**`.m(n)` è staticamente corretta: **`super`** corrisponde alla classe `P`, tutti i metodi `m` di `P` sono **`public`** e quindi accessibili; l'argomento n ha tipo statico `Number[]` quindi l'unica versione applicabile per sottotipo è quella con segnatura `m(Object)` (`Number[]` $\leq$ `Object`) che è anche appropriata. Viene eseguito il metodo di `P` con segnatura `m(Object)`, la stringa restituita viene concatenata con `'\n'` e `"H.m(Number...)"`, quindi viene stampata la stringa

```
P.m(Object)
H.m(Number...)
```

(f) `((H) p1).m(1,2)`: `p1` ha tipo statico `P`, quindi il cast è corretto visto che `P` ed `H` sono in relazione di inheritance. L'espressione `((H) p1)` ha tipo statico `H`. Tutti i metodi `m` di `H` sono **`public`** e quindi accessibili. Analogamente al caso precedente l'invocazione è corretta staticamente e la versione selezionata è quella con segnatura `m(Number...)`.

A run-time `p1` contiene un oggetto della classe `P`, quindi il controllo dinamico di tipo del cast fallisce (visto che `P` $\not\leq$ `H`) per cui l'esecuzione viene interrotta dal lancio dell'eccezione `ClassCastException`.

4. Vedere le soluzioni nel file `soluzione.jar`.