Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 12 luglio

a.a. 2012/2013

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

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

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

L'espressione definisce un sottoinsieme dei literal di tipo **`double`** per il linguaggio Java.

i. **`assert assert`** `p.matcher("42").matches();` ha successo

ii. **`assert assert`** `p.matcher("42.").matches();` ha successo

iii. **`assert assert`** `p.matcher("42.42").matches();` ha successo

iv. **`assert assert`** `!p.matcher("42eE42").matches();` ha successo, poiché il literal non è corretto; infatti l'esponente deve essere un'unica lettera (`e` o `E`)

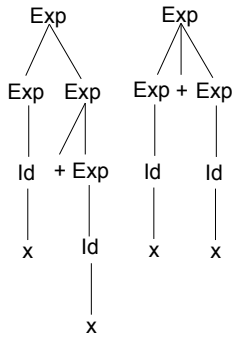
v. **`assert assert`** `!p.matcher("42E.-42").matches();` ha successo, poiché il literal non è corretto; infatti la virgola non può seguire l'esponente

vi. **`assert assert`** `p.matcher("42E2").matches();` ha successo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Id | Exp Exp | Exp + Exp | + Exp
Id ::= x | y | z
```

Esistono due diversi alberi di derivazione per la stringa `x+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 `Exp` **rimanga lo stesso**.

```
Exp  ::= App | Exp + App
App  ::= Term | App Id
Term ::= Id | + Term
Id ::= x | y | z
```

2. Vedere il file `soluzione.ml`

3.
```
public class P {
        public String m(Number n) {
                return "P.m(Number)";
        }
        public String m(Double d) {
                return "P.m(Double)";
        }
        public String m(int i) {
                return "P.m(int)";
        }
}
public class H extends P {
        public String m(Number n) {
                return "H.m(Number) " + super.m(n);
        }
```

```
        public String m(double d) {
                return "H.m(double) " + super.m(d);
        }
        public String m(int i) {
                return "H.m(int) " + super.m(i);
        }
}

public class Test {
        public static void main(String[] args) {
                H h = new H();
                P p = h;
                System.out.println(...);
        }

}
```

I metodi sono tutti **public**, quindi accessibili in `Test`.

(a) `p.m(1)`: `p` e `1` hanno rispettivamente tipo statico `P` e **int**. L'unica versione applicabile per sottotipo (e anche appropriata) è quella con segnatura `m(int)`.

A run-time `p` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` che ridefinisce quello in `P`. Nel body del metodo l'invocazione **super**`.m(i)` è staticamente corretta: **super** corrisponde alla classe `P`, l'argomento `i` ha tipo statico **int**, quindi l'unica versione applicabile per sottotipo è quella con segnatura `m(int)`. Viene stampata la stringa

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

(b) `p.m((Integer) (1))`: il cast è staticamente e dinamicamente corretto per boxing; ricevitore e argomento hanno rispettivamente tipo statico `P` e `Integer`. Solamente il metodo `m(Number)` è applicabile per sottotipo (ed è anche appropriato).

A run-time `p` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` che ridefinisce quello in `P`. Nel body del metodo l'invocazione **super**`.m(n)` è staticamente corretta: **super** corrisponde alla classe `P`, l'argomento `n` ha tipo statico `Number`, quindi l'unica versione applicabile per sottotipo è quella con segnatura `m(Number)`. Viene stampata la stringa

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

(c) `h.m((Integer) (1))`: il cast è staticamente e dinamicamente corretto per boxing; ricevitore e argomento hanno rispettivamente tipo statico `H` e `Integer`. Solamente il metodo `m(Number)` è applicabile per sottotipo (ed è anche appropriato).

A run-time `h` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` e, come nel punto precedente, viene stampata la stringa

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

(d) `h.m(4.2)`: ricevitore e argomento hanno rispettivamente tipo statico `H` e **double**. Il metodo `m(`**double**`)` è l'unico applicabile per sottotipo (ed è anche appropriato).

A run-time `h` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` con segnatura `m(`**double**`)`. Nel body del metodo l'invocazione **super**`.m(d)` è staticamente corretta: **super** corrisponde alla classe `P` ed `d` ha tipo statico **double**. Nessun metodo in `P` è applicabile per sottotipo, ma `m(Double)` e `m(Number)` sono entrambi applicabili per boxing (e widening per `m(Number)`); poiché `Double` $\leq$ `Number`, `m(Double)` è più specifico (e anche appropriato). Viene stampata la stringa

```
H.m(double) P.m(Double)
```

(e) `h.m((Double) 4.2)`: il cast è staticamente e dinamicamente corretto per boxing; ricevitore e argomento hanno rispettivamente tipo statico `H` e `Double`. I metodi `m(Double)` e `m(Number)` sono entrambi applicabili per subtyping, ma poiché `Double` $\leq$ `Number`, `m(Double)` è più specifico (e anche appropriato).

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

```
P.m(Double)
```

(f) `h.m(1,2)`: l'invocazione non è staticamente corretta poiché non esistono metodi con due argomenti o con numero di argomenti variabile.

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