Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 19 giugno

a.a. 2012/2013

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

```
Pattern p = Pattern.compile("\"(\\\\[btfnr]|[^\\\\\"])*\"");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.
L'espressione definisce un sottoinsieme dei literal di tipo `String` per il linguaggio Java.

i. **assert** `p.matcher("").matches();` fallisce, ogni literal deve iniziare e finire con `"`

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

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

iv. **assert** `p.matcher("\"01\"\"").matches();` fallisce, dopo la seconda occorrenza del carattere `"` non possono seguire altri caratteri

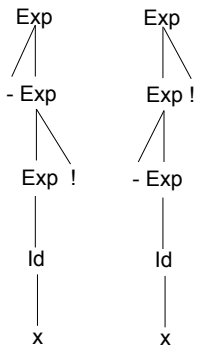
v. **assert** `p.matcher("\"10\\h\"").matches();` fallisce, il carattere `h` non può essere successore immediato di `\`

vi. **assert** `p.matcher("\"10\\n\"").matches();` ha successo

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= Id | Exp ! | - Exp
Id ::= x | y | z
```

Esistono due diversi alberi di derivazione per la stringa `-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  ::= Exp ! | Term
Term ::= - Term | Id
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(double d) {
                return "P.m(double)";
        }
}
public class H extends P {
        public String m(Number n) {
                return "H.m(Number) " + super.m(n);
        }
        public String m(Integer i) {
```

```
                return "H.m(Integer) " + super.m(i);
        }
        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(`**double**`)`.

A run-time `p` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `P` visto che `H` non ridefinisce il metodo `m(`**double**`)`. Viene stampata la stringa

```
P.m(double)
```

(b) `h.m(1)`: `h` e `1` hanno rispettivamente tipo statico `H` e **int**. Due metodi sono applicabili per sottotipo: `m(`**double**`)` e `m(`**int**`)`, ma `m(`**int**`)` è più specifico poiché **int** $\leq$ **double** 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(`**int**`)`. 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(`**double**`)` (vedi il punto precedente). Viene stampata la stringa

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

(c) `h.m((Integer) (1))`: il cast è staticamente e dinamicamente corretto per boxing; ricevitore e argomento hanno rispettivamente tipo statico `H` e `Integer`. Due soli metodi sono applicabili per sottotipo: `m(Number)` e `m(Integer)` (ricordare che `Integer` $\not\leq$ `Double`). Poiché `Integer` $\leq$ `Number`, `m(Integer)` è il metodo più specifico (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(Integer)`. Nel body del metodo l'invocazione **super**`.m(n)` è staticamente corretta: **super** corrisponde alla classe `P` ed `n` ha tipo statico `Integer`. Nessun metodo in `P` è applicabile per sottotipo, ma tramite unboxing e widening `Integer` può essere convertito a **double**, per cui l'unico metodo applicabile è `m(`**double**`)` che è anche appropriato.

Viene stampata la stringa

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

(d) `h.m((Number) 3.2)`: il cast è staticamente e dinamicamente corretto per boxing e widening; ricevitore e argomento hanno rispettivamente tipo statico `H` e `Number`. Il metodo `m(Number)` 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(Integer)`. Nel body del metodo l'invocazione **super**`.m(n)` è staticamente corretta: **super** corrisponde alla classe `P` ed `n` ha tipo statico `Number`. Nessun metodo in `P` è applicabile per sottotipo o per boxing/unboxing, ma il metodo `m(Number... n)` è applicabile (e appropriato).

Viene stampata la stringa

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

(e) `h.m((Double) 3)`: il cast non è staticamente corretto poiché `Integer` $\not\leq$ `Double`.

(f) `h.m(1,2)`: il tipo statico del ricevitore è `H`, mentre i due argomenti hanno tipo statico **int**. Tutti i metodi hanno un solo parametro, quindi non sono applicabili per sottotipo o boxing/unboxing, mentre `m(Number... n)` è applicabile per boxing e widening e perché ammette numero variabile di argomenti.

A run-time `h` contiene un oggetto della classe `H` che non ridefinisce il metodo `m(Number... n)`, quindi viene invocato il metodo della classe `P`.

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

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