Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta dell 13 febbraio

a.a. 2012/2013

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

```
Pattern p = Pattern.compile("(0[xX][0-9a-fA-F][0-9a-fA-F_]*[0-9a-fA-F]|0[bB][01][01_]*[01])L?");
```

Indicare quali delle seguenti asserzioni falliscono, motivando la risposta.

L'espressione definisce literal di tipo intero e long in base 16 o 2: in base 16 il literal deve iniziare con `0x` o `0X`, mentre in base 2 deve iniziare con `0b` o `0B`. Dopo di che devono esserci almeno due cifre esadecimali o binarie, a seconda del caso, una all'inizio e una alla fine; in mezzo può essere usato il carattere underscore, oltre alle normali cifre. I literal possono opzionalmente terminare con la lettera `L`.

i. **assert** `p.matcher("0xff__00L").matches();` ha successo.

ii. **assert** `p.matcher("0Xabc_L").matches();` fallisce: l'ultima cifra esadecimale prima di `L` non può essere il carattere underscore.

iii. **assert** `p.matcher("0101").matches();` fallisce: il literal deve iniziare con `0b` o `0B`.

iv. **assert** `p.matcher("0B_0_1").matches();` fallisce: subito dopo `B` non può esserci il carattere underscore.

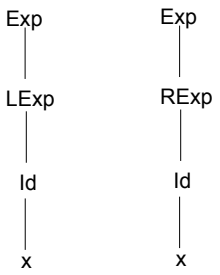
v. **assert** `p.matcher("0b12").matches();` fallisce: 2 non è una cifra binaria.

vi. **assert** `p.matcher("0XfF__Caffe__BEL").matches();` ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= LExp | RExp
RExp ::= Id | Bit | Id + RExp | Bit + RExp
LExp ::= Id
Id ::= x | y | z
Bit ::= 0 | 1
```

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  ::= RExp
RExp ::= Id | Bit | Id + RExp | Bit + RExp
Id ::= x | y | z
Bit ::= 0 | 1
```

2. Vedere il file `soluzione.ml`

3.
```
package pck;
public class P {
    public String m(Object o) {
        return "P.m(Object)";
    }

    protected String m(Number n) {
        return "P.m(Number)";
    }
```

```
    protected String m(int i) {
        return "P.m(int)";
    }

}

package pck;
public class H extends P {
    public String m(Object o) {
        return super.m(o) + '\n' + "H.m(Object)";
    }

    public String m(Number n) {
        return super.m(n) + '\n' + "H.m(Number)";
    }

    public String m(int i) {
        return super.m(i) + '\n' + "H.m(int)";
    }

}

package main;
import pck.*;

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`. Poichè la classe `Test` non è sottoclasse di `P` e si trova in un package diverso, l'unica versione accessibile è quella con segnatura `m(Object o)`; l'argomento ha tipo statico **`int`** perciò il metodo non è applicabile per sottotipo (prima fase), ma lo è per boxing e reference widening (`Integer` $\leq$ `Object`). La versione è anche appropriata.

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

```
P.m(Object)
```

(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 accessibili poiché `H` è sottotipo di `P`; l'argomento `i` ha tipo statico **`int`** quindi l'unica versione applicabile per sottotipo 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`, ci sono due versioni applicabili per sottotipo `m(Object)` e `m(Number)`, ma la seconda è più specifica visto che `Number` $\leq$ `Object` ed è anche appropriata.

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 accessibili poiché `H` è sottotipo di `P`; l'argomento `n` ha tipo statico `Number` quindi due versioni sono applicabili per sottotipo, ma quella con segnatura `m(Number)` è più specifica ed è anche appropriata. Viene eseguito il metodo di `P` con segnatura `m(Number)`, la stringa restituita viene concatenata con `'\n'` e `"H.m(int)"`, quindi viene stampato

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

(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` $\leq$ `Number` $\leq$ `Object`), quindi ci sono due versioni applicabili, ma quella con segnatura `m(Number)` è più specifica.

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

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

(e) `p2.m(new Integer(1))`: `p2` ha tipo statico `P`. Poichè la classe `Test` non è sottoclasse di `P` e si trova in un package diverso, l'unica versione accessibile è quella con segnatura `m(Object o)`; l'argomento ha tipo statico `Integer` perciò il metodo è applicabile per sottotipo ed è anche appropriato.

A run-time `p2` contiene un oggetto della classe `H`, quindi il metodo invocato è quello in `H` con segnatura `m(Object)`. Nel body del metodo l'invocazione `super.m(o)` è staticamente corretta: `super` corrisponde alla classe `P`, tutti i metodi `m` di `P` sono accessibili poiché `H` è sottotipo di `P`; l'argomento `o` ha tipo statico `Object` l'unica versione applicabile è quella con segnatura `m(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(int)"`, quindi viene stampato

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

(f) `((H) p2).m((short) 1)`: il tipo statico di `p2` è `P`, l'espressione `(H) p2` è corretta staticamente, poiché `P` e `H` sono in relazione di sottotipo, e ha tipo `H`. Il cast tra tipi numerici è sempre ammesso e il tipo statico dell'argomento è `short`. Tutti i metodi `m` di `H` sono `public` e quindi accessibili. Solo la versione con segnatura `m(int)` è applicabile per sottotipo ed è anche appropriata.

A run-time `p2` contiene un oggetto della classe `H`, quindi il controllo di tipo imposto dal narrowing cast ha successo e viene invocato i metodo di `H` con segnatura `m(int)`. Il comportamento è lo stesso del punto 2, quindi viene stampato

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

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