Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2012/2013

13 febbraio 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.

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

ii. `assert p.matcher("0Xabc_L").matches();`

iii. `assert p.matcher("0101").matches();`

iv. `assert p.matcher("0B_0_1").matches();`

v. `assert p.matcher("0b12").matches();`

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

(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
```

(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**.

2. (a) Considerare la funzione `apply_all : 'a -> ('a -> 'b) list -> 'b list` tale che

`apply_all` x `[`f_1`; ...` f_n`] = [`f_1 x`; ...` f_n x`]`.

Esempio:

```
# apply_all 3 [(fun x->x);(fun x->x+1);(fun x->x-1);(fun x->x*x)];;
- : int list = [3; 4; 2; 9]
```

Definire la funzione `apply_all` direttamente, senza uso di parametri di accumulazione.

(b) Definire la funzione `apply_all` direttamente, usando un parametro di accumulazione affinché la ricorsione sia di coda.

(c) Considerare la funzione `comp_all : 'a -> ('a -> 'a) list -> 'a` tale che

`comp_all` x `[`f_1`; ...` f_n`] = (`f_n `...` `(`f_2 `(`f_1 x`))` `...` `)`.

Esempio:

```
# comp_all 3 [(fun x->x+1);(fun x->2*x);(fun x->x-1);(fun x->x*x)];;
- : int = 49
```

Definire la funzione `comp_all` come specializzazione della funzione `it_list` così definita:

```
let rec it_list f a = function x::l -> it_list f (f a x) l | _ -> a;;
val it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

3. Considerare le seguenti dichiarazioni di classi Java:

```
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(...);
    }
}
```

Dire, per ognuno dei casi elencati sotto, che cosa succede sostituendo al posto dei puntini nella classe `Test` il codice indicato.

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) `p1.m(1)`

(b) `h.m(1)`

(c) `h.m(new Integer(1))`

(d) `h.m(3.2)`

(e) `p2.m(new Integer(1))`

(f) `((H) p2).m((short) 1)`

4. Considerare la seguente classe `ConcatLang` che implementa l'interfaccia `Language` e che rappresenta la concatenazione di due linguaggi finiti (ossia, di due insiemi finiti di stringhe).

```
import java.util.Set;

public interface Language extends Iterable<String> {
    boolean contains(String str);

    Set<String> getSet();
}
```

```
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Set;

public class ConcatLang implements Language {

    final private Set<String> first;
    final private Set<String> second;

    public ConcatLang(Set<String> first, Set<String> second) {
        // completare
    }
    public boolean contains(String str) {
        // completare
    }
    public Set<String> getSet() {
        // completare
    }
    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private Iterator<String> it1 = first.iterator();
            private Iterator<String> it2 = Collections.<String> emptyIterator();
            private String firstString; // stores the last string returned by it1.next()

            public boolean hasNext() {
                // completare
            }
            public String next() {
                // completare
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Il costruttore e i metodi implementati nella classe `ConcatLang` sono i seguenti:

- **public** `ConcatLang(Set<String> first, Set<String> second)`: costruisce un nuovo oggetto che rappresenta la concatenazione dei due insiemi contenuti rispettivamente in `first` e `second`; solleva l'eccezione `IllegalArgumentException` se `first` o `second` contiene **null**.
- **public boolean** `contains(String str)`: restituisce **true** se e solo se la stringa in `str` appartiene alla concatenazione dei due linguaggi; solleva l'eccezione `IllegalArgumentException` se `str` contiene **null**.
- **public** `Set<String> getSet()`: restituisce un nuovo insieme corrispondente alla concatenazione dei due linguaggi.
- **public** `Iterator<String> iterator()`: restituisce un iteratore in grado di iterare su tutte le stringhe che appartengono alla concatenazione dei due linguaggi.

A titolo di esempio, il seguente test deve avere successo.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class Test {

    public static void main(String[] args) {
        Language lan = new ConcatLang(new HashSet<String>(
                Arrays.asList("a", "")), new HashSet<String>(Arrays.asList("b", "c")));
        assert lan.contains("ac");
        assert !lan.contains("a");
        Set<String> set = lan.getSet();
        for(String st:lan)
            assert set.contains(st);
    }

}
```

(a) Completare il costruttore della classe `ConcatLang`.

(b) Completare i metodi `contains(String str)` e `getSet()` della classe `ConcatLang` (suggerimento: utilizzare un iteratore).

(c) Completare il metodo `hasNext()` della classe anonima definita nel metodo `iterator()` (ricordare che la concatenazione di due insiemi è vuota nel caso in cui uno dei due insiemi sia vuoto).

(d) Completare il metodo `next()` della classe anonima definita nel metodo `iterator()`. Assumere che i due insiemi in `first` e `second` non contengano **`null`**.