

Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2014/2015

16 gennaio 2015

1. (a) Dato il seguente codice Java, indicare quali delle asserzioni contenute in esso falliscono, motivando la risposta.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MatcherTest {
    public static void main(String[] args) {
        Pattern regex = Pattern.compile(
            "(null|(?<HEAD>[a-zA-Z]))(?<TAIL>[a-zA-Z0-9]*)|0[bB](?<NUM>[01]+)|(?<SKIP>\\s+)");
        Matcher m = regex.matcher("null13      420b11");
        assert m.lookingAt();
        assert m.group("TAIL").length() > 0;
        assert m.group("HEAD") != null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group("SKIP") != null;
        m.region(m.end(), m.regionEnd());
        assert m.lookingAt();
        m.find();
        assert Integer.parseInt(m.group("NUM"), 2) == 3;
    }
}
```

- (b) Mostrare che la seguente grammatica è ambigua.

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

- (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 resti invariato**.

2. In OCaml la funzione predefinita `max : 'a -> 'a -> 'a` permette di calcolare il massimo tra due elementi dello stesso tipo; per esempio,

```
# max 42 2
- : int = 42
```

```
# max "hello" "world"
- : string = "world"
```

Considerare la funzione `max_list : 'a -> 'a list -> 'a` tale che `max_list n l` restituisce il massimo tra `n` e tutti gli elementi della lista `l`.

Esempio:

```
# max_list 0 [1;2;3;4;4;3;2;1]
- : int = 4
```

```
# max_list 10 [1;2;3;4;4;3;2;1]
- : int = 10
```

- (a) Definire la funzione `max_list` direttamente, senza uso di parametri di accumulazione.
(b) Definire la funzione `max_list` direttamente, usando un parametro di accumulazione affinché la ricorsione sia di coda.
(c) Definire la funzione `max_list` 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. (a) Completare la definizione del costruttore e dei metodi della classe `Pair<E1,E2>` che implementa coppie di oggetti rispettivamente di tipo `E1` ed `E2`.

```
public final class Pair<E1, E2> {
    private final E1 first;
    private final E2 second;

    public Pair(E1 first, E2 second) {
        // completare
    }
    public E1 getFirst() {
        // completare
    }
    public E2 getSecond() {
        // completare
    }
    @Override
    public int hashCode() {
        // completare
    }
    @Override
    public boolean equals(Object obj) {
        // completare
    }
    @Override
    public String toString() {
        // completare
    }
}
```

- (b) Considerare la classe `ComposeIterator<E1, E2, T>` che permette di comporre, tramite un oggetto di tipo `Composer<E1, E2, T>`, due iteratori rispettivamente di tipo `Iterator<E1>` e `Iterator<E2>`, per ottenere un iteratore di tipo `Iterator<T>`. L'iteratore produce elementi ottenuti componendo, tramite l'oggetto `composer`, le coppie di elementi via via restituite da `firstIterator` e `secondIterator`. L'iterazione termina quando uno dei due iteratori termina.

Esempio di uso:

```
public class AddLengths implements Composer<String, String, Integer> {
    @Override
    public Integer compose(String e1, String e2) {
        return e1.length() + e2.length();
    }
}

...
Iterator<String> it1 = Arrays.asList("a", "ab").iterator();
Iterator<String> it2 = Arrays.asList("", "abc", "a").iterator();
Iterator<Integer> it3 = new ComposeIterator<>(it1, it2, new AddLengths());
while(it3.hasNext())
    // stampa "1 5" ossia la lunghezza di "a"+" " e di "ab"+"abc"
    System.out.print(it3.next()+" ");
```

Completare la definizione del costruttore e dei metodi della classe `ComposeIterator<E1, E2, T>`.

```
public interface Composer<E1, E2, T> {
    T compose(E1 e1, E2 e2);
}

public class ComposeIterator<E1, E2, T> implements Iterator<T> {
    private final Iterator<E1> firstIterator;
    private final Iterator<E2> secondIterator;
    private final Composer<E1, E2, T> composer;

    public ComposeIterator(Iterator<E1> firstIterator,
        Iterator<E2> secondIterator, Composer<E1, E2, T> composer) {
        // completare
    }
    @Override
    public boolean hasNext() {
        // completare
    }
    @Override
    public T next() {
        // completare
    }
}
```

- (c) Considerare il metodo `buildPairs` che prende come argomenti due iteratori `it1` e `it2` e restituisce un iteratore che genera le coppie di elementi prodotti da `it1` e `it2`.

Esempio:

```

Iterator<Pair<Integer, String>> it = buildPairs(Arrays.asList(1, 2, 3).iterator(),
                                                Arrays.asList("one", "two", "three").iterator());
while (it.hasNext())
    // stampa "(1,one) (2,two) (3,three)"
    System.out.print(it.next() + " ");

```

Completare la definizione del metodo `buildPairs` **utilizzando la classe** `ComposeIterator` e definire la classe ausiliaria usata per creare il necessario oggetto di tipo `Composer`.

```

public static <E1, E2> Iterator<Pair<E1, E2>> buildPairs(Iterator<E1> it1,
                                                        Iterator<E2> it2) {
    // completare
}

```

4. Considerare le seguenti dichiarazioni di classi Java:

```

public class P {
    String m(int i) {
        return "P.m(int)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
public class H extends P {
    String m(int i) {
        return super.m(i) + " H.m(int)";
    }
    String m(Integer i) {
        return super.m(i) + " H.m(Integer)";
    }
    String m(int... is) {
        return "H.m(int...)";
    }
}
public class Test {
    public static void main(String[] args) {
        P p = 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) `p.m((short)42)`
- (b) `p2.m((short)42)`
- (c) `p.m(Integer.valueOf(42))`
- (d) `p2.m(Integer.valueOf(42))`
- (e) `h.m(Integer.valueOf(42))`
- (f) `p2.m(4,2)`