Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2014/2015

17 settembre 2015

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali falliscono, motivando la risposta.

```
public class MatcherTest {
    public static void main(String[] args) {
        Pattern regEx = Pattern.compile("(?<KEY>const|with)|(?<ID>[a-zA-Z][0-9]*)|
                                                0[xX](?<NUM>[a-fA-F0-9]+)|(?<SKIP>\\s+)");
        Matcher m = regEx.matcher("const0xAf a1");
        assert m.lookingAt();
        assert m.group("KEY").equals("const");
        assert m.group("ID") != null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert Integer.parseInt(m.group("NUM"), 16) == 175;
        m.region(m.end(), m.regionEnd());
        assert m.lookingAt();
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group("ID") == null;
    }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Exp + Exp | + Exp | ( Exp ) | Id
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. Considerare la funzione `count : ('a -> bool) -> 'a list -> int` tale che `count` p l restituisce il numero di elementi della lista l che soddisfano il predicato p.

   Esempi:

```
# count (fun x -> x > 0) [-1; 2; 0; 3; -1]
- : int = 2
# count (fun x -> x > 0) [-1; -2; 0; -3; -1]
- : int = 0
# count (fun x -> x > 0) [1; 2; 3; 4; 5]
- : int = 5
```

   (a) Definire la funzione `count` direttamente, senza uso di parametri di accumulazione.

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

   (c) Definire la funzione `count` 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 classe `CatIterator<E>` che implementa la concatenazione di due iteratori `it1` e `it2` su elementi di tipo `E`. L'iteratore ottenuto dalla concatenazione di `it1` e `it2` restituisce nell'ordine prima tutti gli elementi di `it1` e poi quelli di `it2`.

```
import java.util.Iterator;
public class CatIterator<E> implements Iterator<E> {
    private final Iterator<E> it1;
    private final Iterator<E> it2;
    public CatIterator(Iterator<E> it1, Iterator<E> it2) { /* da completare */ }
    @Override public boolean hasNext() { /* da completare */ }
    @Override public E next() { /* da completare */ }
}
```

(b) Completare la classe `CombIterator` che permette di combinare due iteratori `it1` e `it2`, rispettivamente di tipo `Iterator<T1>` e `Iterator<T2>`, tramite il metodo `R apply(T1 t1,T2 t2)` di un oggetto `comb` di tipo `BiFunction<T1,T2,R>`.

A ogni iterazione l'iteratore ottenuto combinando `it1` e `it2` si comporta nel seguente modo:

- se entrambi gli iteratori hanno, rispettivamente, un prossimo elemento e_1 ed e_2, allora viene restituito `comb.app(`e_1`,`e_2`)` come prossimo elemento e l'iterazione avanza per entrambi gli iteratori;
- se solo `it1` ha un prossimo elemento e_1, allora viene restituito `comb.app(`e_1`,null)` come prossimo elemento e l'iterazione avanza solo per `it1`;
- se solo `it2` ha un prossimo elemento e_2, allora viene restituito `comb.app(null,`e_2`)` come prossimo elemento e l'iterazione avanza solo per `it2`;
- se nessuno dei due iteratori ha un prossimo elemento, allora l'iterazione termina.

```
import java.util.Iterator;
public interface BiFunction<T,U,R> { R apply(T t, U u); }
public class CombIterator<T1, T2, R> implements Iterator<R> {
    private final Iterator<T1> it1;
    private final Iterator<T2> it2;
    private final BiFunction<T1, T2, R> comb;
    public CombIterator(Iterator<T1> it1, Iterator<T2> it2,
            BiFunction<T1, T2, R> comb) { /* da completare */ }
    @Override public boolean hasNext() { /* da completare */ }
    @Override public R next() { /* da completare */ }
}
```

(c) Completare la classe `MergeIterator` che permette di combinare due iteratori `it1` e `it2` di tipo `Iterator<Integer>` per ottenere un nuovo iteratore di tipo `Iterator<Integer>`, assumendo che entrambi gli iteratori non restituiscano mai il valore `null`.

A ogni iterazione l'iteratore ottenuto combinando `it1` e `it2` si comporta nel seguente modo:

- se entrambi gli iteratori hanno, rispettivamente, un prossimo elemento i_1 e i_2, allora se $i_1 \leq i_2$ viene restituito come prossimo elemento i_1 e l'iterazione avanza solo per `it1`, altrimenti (ossia, se $i_1 > i_2$) viene restituito come prossimo elemento i_2 e l'iterazione avanza solo per `it2`;
- se solo `it1` ha un prossimo elemento i_1, allora viene restituito i_1 come prossimo elemento e l'iterazione avanza solo per `it1`;
- se solo `it2` ha un prossimo elemento i_2, allora viene restituito i_2 come prossimo elemento e l'iterazione avanza solo per `it2`;
- se nessuno dei due iteratori ha un prossimo elemento, allora l'iterazione termina.

```
import java.util.Iterator;
public class MergeIterator implements Iterator<Integer> {
    private final Iterator<Integer> it0;
    private final Iterator<Integer> it1;
    // curr[0] current element of it0, curr[1] current element of it1
    private Integer[] curr = new Integer[2];
    private Integer tryNext(Iterator<Integer> it) {
        if (it.hasNext())
            return it.next();
        return null;
    }
    private Integer advance(int i, Iterator<Integer> it) {
        Integer res = curr[i];
        curr[i] = tryNext(it);
        return res;
    }
    public MergeIterator(Iterator<Integer> it0, Iterator<Integer> it1) { /* da completare */ }
    @Override public boolean hasNext() { /* da completare */ }
    @Override public Integer next() {
        if (curr[0] != null)
            if (curr[1] == null || curr[0] <= curr[1])
                return advance(0, it0);
            else
                return advance(1, it1);
        // da completare
    }
}
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(char c) { return "P.m(char)"; }
    String m(String s) { return "P.m(String)"; }
}
public class H extends P {
    String m(char c) { return super.m(c) + " H.m(char)"; }
    String m(Character c) { return super.m(c) + " H.m(Character)"; }
    String m(Character... cs) {
        StringBuilder sb = new StringBuilder();
        for (Character c : cs)
            sb.append(c).append(" ");
        return sb.append("H.m(Character...)").toString();
    }
}
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, assumendo che tutte le classi siano dichiarate nello stesso package.

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('a')`

(b) `p2.m('a')`

(c) `h.m(Character.valueOf('a'))`

(d) `p.m("a")`

(e) `p2.m(new char[] { '4','2' })`

(f) `h.m(new Character[] { '4','2' })`