Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2016/2017

11 settembre 2017

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("([$a-zA-Z][a-zA-Z0-9]+)|([0-9]+\\.?[0-9]*|\\.[0-9]+)|(\\s+)");
      Matcher m = regEx.matcher(".09 12.");
      m.lookingAt();
      assert m.group(2).equals(".09");
      assert m.group(3) == null;
      m.region(m.end(), m.regionEnd());
      m.lookingAt();
      assert m.group(0) != null;
      assert m.group(0).length() > 0;
      m.region(m.end(), m.regionEnd());
      m.lookingAt();
      assert m.group(2).equals("12.");
      assert m.group(0).length() == 3;
   }
}
```

   (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. Sia `odd : 'a list -> 'a list` la funzione così specificata:
   `odd` l restituisce la lista che contiene, nello stesso ordine, i soli elementi di l di posizione dispari (ossia, il primo, il terzo, ecc.).

   Esempio:

```
# odd []
- : 'a list = []

# odd [1]
- : int list = [1]

# odd [1;2]
- : int list = [1]

# odd [1;2;3;4;5]
- : int list = [1; 3; 5]
```

   (a) Definire la funzione `odd` senza uso di parametri di accumulazione.

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

3. La seguente classe `FilteredIterator` permette di iterare sugli elementi appartenenti a una lista di tipo `ArrayList<E>` che soddisfano un predicato di tipo `Predicate<E>`.

```
public interface Predicate<E> {
        boolean test(E t);
}
public class FilteredIterator<E> implements Iterator<E> {
   private final Predicate<E> pred;
   private final ArrayList<E> list;
   private int curr; // index of the current element

   public FilteredIterator(Predicate<E> pred, ArrayList<E> list) { /* da completare */ }
   public boolean hasNext() { /* da completare */  }
   public E next() { /* da completare */  }
}
```

Per esempio, le **`assert`** nel seguente frammento di codice sono sempre verificate:

```
Predicate<Integer> odd = ... // predicato che testa se un intero e' dispari
ArrayList<Integer> list = ... // lista  [1, 2, 4, 3, 5, 6]
FilteredIterator<Integer> fit = new FilteredIterator<>(odd, list);
int elem = 1;
int count = 0;
while (fit.hasNext()) {
   assert fit.next().equals(elem);
   elem += 2;
   count++;
}
assert count == 3;
```

(a) Completare le definizioni del costruttore della classe.

(b) Completare le definizioni dei metodi `hasNext()` e `next()`.

(c) Utilizzando la classe `FilteredIterator`, completare il metodo `find` che restituisce il primo elemento di una lista di tipo `ArrayList<E>` che verifica un predicato di tipo `Predicate<E>`, se tale elemento esiste, o solleva l'eccezione `NoSuchElementException` altrimenti.

```
public static <E> E find(Predicate<E> pred, ArrayList<E> list) { /* completare */ }
```

Per esempio, la seguente **`assert`** ha successo:

```
Predicate<Integer> positive = ... // predicato che testa se un numero intero e' positivo
ArrayList<Integer> list = ... // lista  [-1, -2, -3, -4, 5, 6]
assert find(positive, list).equals(5);
```

4. Assumere che le seguenti dichiarazioni di classi Java siano contenute nello stesso package:

```
public class P {
   String m(Double d) {
      return "P.m(Double)";
   }
   String m(Long l) {
      return "P.m(Long)";
   }
}
public class H extends P {
   String m(double d) {
      return super.m(d) + " H.m(double)";
   }
   String m(long l) {
      return super.m(l) + " H.m(long)";
   }
}
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(42)`

(b) `h.m(42)`

(c) `p.m(42L)`

(d) `h.m(42L)`

(e) `p.m(42.)`

(f) `p2.m(42.)`