Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2016/2017

25 gennaio 2018

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("(0[0-7]*)|([1-9][0-9]*)|(\\s+)");
      Matcher m = regEx.matcher("01 7");
      m.lookingAt();
      assert m.group(1).equals("01");
      assert m.group(2) == null;
      m.region(m.end(), m.regionEnd());
      m.lookingAt();
      assert !m.group(3).equals(null);
      assert m.group(3).length() >= 1;
      m.region(m.end(), m.regionEnd());
      m.lookingAt();
      assert m.group(2).equals("7");
      assert !m.group(2).equals(7);
   }
}
```

   (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 `list_gen : ('a -> 'a) -> 'a -> int -> 'a list` la funzione così specificata: `list_gen` f i n restituisce la lista di lunghezza n dove il primo elemento è i, il secondo è $f(i)$, il terzo $f(f(i))$ e così via. Esempi:

```
# list_gen (fun x->x+1) 0 3;;
- : int list = [0; 1; 2]

# list_gen (fun x->x*2) 1 4;;
- : int list = [1; 2; 4; 8]

# list_gen (fun x->"a"^x) "" 5;;
- : string list = [""; "a"; "aa"; "aaa"; "aaaa"]
```

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

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

3. Completare il costruttore e i metodi della classe `FunIterator<T>` che permette di creare iteratori infiniti a partire da un oggetto di tipo `Function<T,T>` e un valore iniziale di tipo `T`; i valori di tipo `Function<T,T>` rappresentano funzioni da valori di tipo `T` a valori di tipo `T`.

```
// functions from T to R
public interface Function<T, R> {
        R apply(T t); // applies the function to t
}

public class FunIterator<T> implements Iterator<T> {
   private final Function<T, T> fun;
   private T first; // allowed to be null

   public FunIterator(Function<T, T> fun, T first) {
      // completare
   }

   public boolean hasNext() {
      // completare
   }

   public T next() {
      // completare
   }
}
```

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

```
Function<String, String> append = // oggetto che rappresenta la funzione OCaml fun x -> "a"^x
Iterator<String> it = new FunIterator<String>(append, "");
assert it.next().equals("");
assert it.next().equals("a");
assert it.next().equals("aa");
```

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

```
public class P {
   String m(Number n) {
      return "P.m(Number)";
   }
   String m(double d) {
      return "P.m(double)";
   }
}
public class H extends P {
   String m(Double d) {
      return super.m(d) + " H.m(Double)";
   }
   String m(Integer i) {
      return super.m(i) + " H.m(Integer)";
   }
}
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(new Integer(42))`

(b) `p2.m(new Integer(42))`

(c) `h.m(new Integer(42))`

(d) `p.m(new Double(42))`

(e) `p2.m(new Double(42))`

(f) `h.m(new Double(42))`