Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2014/2015

5 febbraio 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("(int|bool|(?<HEAD>[a-zA-Z]))(?<TAIL>[a-zA-Z0-9]*)|(?<NUM>0[0-7]*)|\\s+");
        Matcher m = regEx.matcher("int 017");
        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("NUM") != null;
        m.region(m.end(), m.regionEnd());
        m.lookingAt();
        assert m.group("NUM") != null;
        assert Integer.parseInt(m.group("NUM"), 8) == 15;
    }
}
```

   (b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Not | ! Exp | Exp && Not
Not  ::= Id | ! Not
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 in l che soddisfano il predicato p.

   Esempio:

```
# count (fun x -> x > 0) [-1;2;3;-4;1]
- : int = 3
# count (fun x -> x="red") ["black";"white";"red";"green";"red"]
- : int = 2
```

   (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 definizione della classe `KeepPositive` con il metodo `keep` in modo che restituisca **`true`** se e solo se il suo argomento è positivo.

```
public interface Filter<T> {
    boolean keep(T t);
}
public class KeepPositive implements Filter<Integer> {
    // completare
}
```

(b) Completare il costruttore e il metodo `hasCurrent` della classe `EnhancedIteratorClass<T>` che permette di estendere le funzionalità di un oggetto di tipo `Iterator<T>`.

```
import java.util.Iterator;

public interface EnhancedIterator<T> extends Iterator<T> {
    boolean hasCurrent();
    T getCurrent();
    boolean moveNext();
}
```

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class EnhancedIteratorClass<T> implements EnhancedIterator<T> {
    private final Iterator<T> iterator;
    private T current;
    private boolean hasCurrent;

    public EnhancedIteratorClass(Iterator<T> iterator) {
        // completare
    }
    @Override
    public boolean hasCurrent() {
        // completare
    }
    @Override
    public T getCurrent() {
        if (!hasCurrent())
            throw new NoSuchElementException();
        return current;
    }
    @Override
    public boolean hasNext() {
        return iterator.hasNext();
    }
    @Override
    public T next() {
        if (moveNext())
            return current;
        else
            throw new NoSuchElementException();
    }
    @Override
    public boolean moveNext() {
        if (hasCurrent = hasNext())
            current = iterator.next();
        return hasCurrent;
    }
}
```

(c) Utilizzando la classe `EnhancedIteratorClass<T>` completare il costruttore e i metodi `hasNext` e `next` della classe `FilteredIterator<T>` che permette di costruire un nuovo iteratore i', a partire da un altro iteratore i e da un filtro f; i' restituisce nello stesso ordine tutti e soli gli elementi restituiti da i che sono filtrati da f, ossia, gli elementi e tali che f`.keep(`e`)` si valuta in **`true`**.

Esempio:

```
public class KeepNotEmpty implements Filter<String> {
    @Override
    public boolean keep(String st) {
        return st != null && st.length() > 0;
    }
}
...
java.util.List<String> l = java.util.Arrays.asList(null, "a", "", "ab", "");
FilteredIterator<String> it = new FilteredIterator<>(l.iterator(), new KeepNotEmpty());
while(it.hasNext())
    System.out.print(it.next().length()+" "); // stampa 1 2
}
```

Definizione della classe `FilteredIterator<T>`:

```
import java.util.Iterator;

public class FilteredIterator<T> implements Iterator<T> {

    private final EnhancedIterator<T> iterator;
    private final Filter<T> filter;

    public FilteredIterator(Iterator<T> iterator, Filter<T> filter) {
        // completare
    }

    @Override
    public boolean hasNext() {
        // completare
    }
    @Override
    public T next() {
        // completare
    }
}
```

(d) Utilizzando le classi `FilteredIterator<T>` e `KeepPositive`, completare la definizione del metodo `getAllPositive` che presa una collezione di interi, restituisce un iteratore che genera tutti e solo gli elementi positivi della collezione.

```
import java.util.Collection;
public class Util {
    public static FilteredIterator<Integer> getAllPositive(
            Collection<Integer> col) {
        // completare
    }
}
```

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
        String m(double d) {
                return "P.m(double)";
        }
        String m(Object o) {
                return "P.m(Object)";
        }
}
public class H extends P {
        String m(double d) {
                return super.m(d) + " H.m(double)";
        }
        String m(Double d) {
                return super.m(d) + " H.m(Double)";
        }
        String m(double... ds) {
                return "H.m(double...)";
        }
}
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(42.42)`
(b) `p2.m(42.42)`
(c) `p.m(Double.valueOf(42))`
(d) `p2.m(Float.valueOf(42))`
(e) `h.m(Double.valueOf(42))`
(f) `h.m(4,2)`