Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 5 febbraio

a.a. 2014/2015

13 febbraio 2015

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

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class MatcherTest {
5      public static void main(String[] args) {
6          Pattern regEx = Pattern
7              .compile("(int|bool|(?<HEAD>[a-zA-Z]))(?<TAIL>[a-zA-Z0-9]*)|(?<NUM>0[0-7]*)|\\s+");
8          Matcher m = regEx.matcher("int 017");
9          assert m.lookingAt();
10         assert m.group("TAIL").length() > 0;
11         assert m.group("HEAD") == null;
12         m.region(m.end(), m.regionEnd());
13         m.lookingAt();
14         assert m.group("NUM") != null;
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group("NUM") != null;
18         assert Integer.parseInt(m.group("NUM"), 8) == 15;
19     }
20 }
```

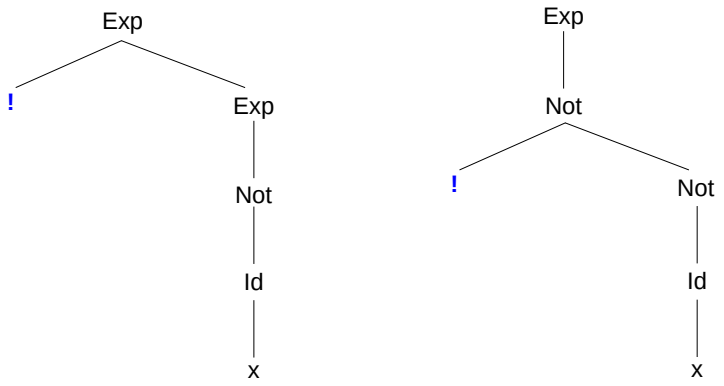
Soluzione:

- `assert m.lookingAt();` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `"int 017"` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `"int"`, ottenuta come concatenazione di `"int"` (gruppo 1) e `""` (gruppo `TAIL`), quindi l'asserzione ha successo;
- `assert m.group("TAIL").length() > 0;` (linea 10): `m.group("TAIL").length()` si valuta in 0, per i motivi indicati al punto precedente, per cui l'asserzione fallisce;
- `assert m.group("HEAD") == null;` (linea 11): poiché `m.group(1).equals("int")`, e `"int"` corrisponde alla prima sotto-espressione in `"int|bool|(?<HEAD>[a-zA-Z])"`, necessariamente l'asserzione ha successo;
- `assert m.group("NUM") != null;` (linea 14): alla linea 12 l'inizio della regione viene spostato al carattere immediatamente successivo a `"int"` che è uno spazio bianco e appartiene al gruppo `SKIP`, quindi l'asserzione fallisce;
- `assert m.group("NUM") != null;` (linea 17): alla linea 15 l'inizio della regione viene spostato al carattere immediatamente successivo allo spazio bianco (carattere `0`), l'invocazione `m.lookingAt()` alla linea 16 ha successo poiché la stringa `"017"` appartiene al gruppo `NUM`, quindi l'asserzione ha successo;
- `assert Integer.parseInt(m.group("NUM"), 8) == 15;` (linea 18): `m.group("NUM")` restituisce la stringa `"017"` appena riconosciuta, per cui l'asserzione ha successo, visto che la decodifica di `017` in base 8 senza segno è 15.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Not | ! Exp | Exp && Not
Not  ::= Id | ! Not
Id   ::= x | y | z
```

Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `!x`



(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**.

La soluzione più semplice consiste nell'eliminare la produzione `Exp ::= !Exp`. La grammatica risultante è non ambigua (l'operatore `!` ha precedenza su `&&` che associa da sinistra) e il linguaggio generato è lo stesso grazie alla produzione `Not ::= !Not`.

```
Exp  ::=  Not | Exp && Not
Not  ::= Id | ! Not
Id   ::= x | y | z
```

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>
```

Soluzione: Vedere il file `soluzione.ml`.

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
    }
}
```

Soluzione: Vedere il file `soluzione.jar`.

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)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, si hanno i seguenti casi:

(a) Il literal `42.42` ha tipo statico **`double`**; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(double d)` in `P`.

Viene stampata la stringa `"P.m(double)"`.

(b) L'espressione è staticamente corretta per esattamente lo stesso motivo del punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(double d)` in `H`. Poiché `d` ha tipo statico **`double`**, per l'invocazione **`super`**`.m(d)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(double d)`.

Viene stampata la stringa `"P.m(double) H.m(double)"`.

(c) Il literal `42` ha tipo statico **`int`** e per l'invocazione `Double.valueOf(42)` esiste un solo metodo statico nella classe `Double` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Double`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo `String m(Object o)` in `P`. La conversione di 42 da **`int`** a **`double`** non comporta perdita di informazione.

Viene stampata la stringa `"P.m(Object)"`.

(d) Il literal `42` ha tipo statico **`int`** e per l'invocazione `Float.valueOf(42)` esiste un solo metodo statico nella classe `Float` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Float`. Il tipo statico di `p2` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi viene eseguito il metodo `String m(Object o)` in `P`. La conversione di 42 da **`int`** a **`double`** non comporta perdita di informazione.

Viene stampata la stringa `"P.m(Object)"`.

(e) Come al punto (c), l'epressione `Double.valueOf(42)` ha tipo statico `Double`. Il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(Double d)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Double d)` in `H`. Poiché `d` ha tipo statico `Double`, per l'invocazione **`super`**`.m(d)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(Object o)`.

Viene stampata la stringa `"P.m(Object) H.m(Double)"`.

(f) La variabile `h` ha tipo statico `H`, mentre i literal 4 e 2 hanno entrambi tipo **`int`**. e poiché in `P` non esiste alcun metodo con due argomenti, l'unico metodo accessibile e applicabile è `String m(double... ds)`, visto che **`int`** è sotto-tipo di **`double`**.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(double... ds)` in `H`. La conversione da **`int`** a **`double`** è senza perdita di informazione.

Viene stampata la stringa `"H,m(double...)"`.