

Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta dell'8 settembre

a.a. 2015/2016

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali 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 =
7              Pattern.compile("([A-Za-z][A-Za-z0-9$_]*)|(0[bB]([01]|[01][01_]*[01])[1L]?)|->|-|\\s+");
8          Matcher m = regex.matcher("b0_0->0B1L");
9          mLookingAt();
10         assert m.group(1).equals("b0_0");
11         m.region(m.end(), m.regionEnd());
12         assert mLookingAt();
13         assert m.group(0).equals("->");
14         m.region(m.end(), m.regionEnd());
15         assert mLookingAt();
16         assert m.group(2).equals("0B1");
17         assert m.group(3).equals("0");
18     }
19 }
```

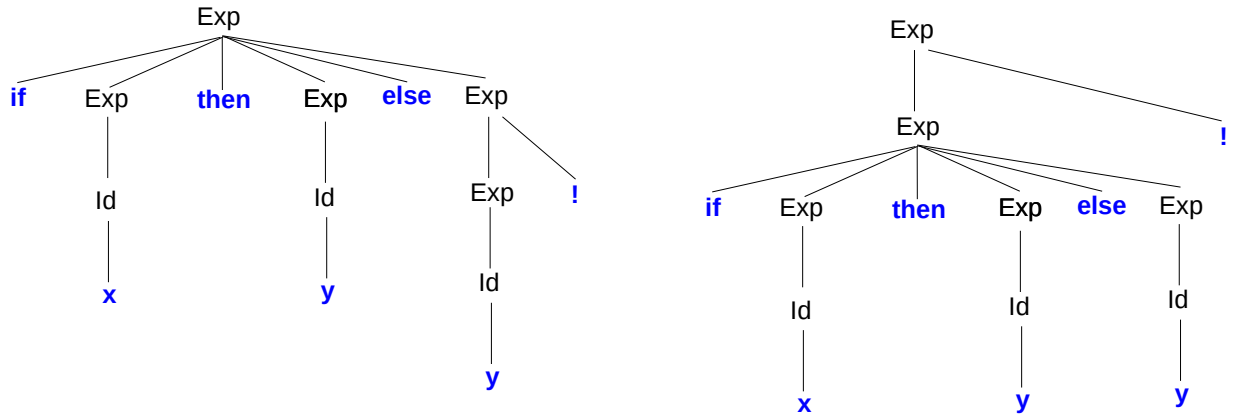
Soluzione:

- i. **assert** m.group(1).equals("b0_0"); (linea 10): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa **b0_0->0B1L** 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 **b0_0** (stringa appartenente al gruppo di indice 1), quindi l'asserzione ha successo;
- ii. **assert** mLookingAt(); (linea 12): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a **b0_0** (ossia -); l'invocazione di **lookingAt()** restituisce **true** poiché **->** appartiene all'espressione regolare (solo gruppo 0), quindi l'asserzione ha successo;
- iii. **assert** m.group(0).equals("->"); (linea 13): per le motivazioni del punto precedente l'asserzione ha successo;
- iv. **assert** mLookingAt(); (linea 15): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a **->** (ossia 0) e l'invocazione del metodo **lookingAt()** ha successo poiché **0B1L** appartiene all'espressione regolare (gruppo 2, mentre la sottostringa **1** appartiene anche al gruppo 3), quindi l'asserzione ha successo;
- v. **assert** m.group(2).equals("0B1"); (linea 16): per i motivi del punto precedente l'asserzione fallisce poiché le due stringhe **0B1L** e **0B1** sono diverse;
- vi. **assert** m.group(3).equals("0"); (linea 17): per i motivi del punto iv l'asserzione fallisce poiché le due stringhe **1** e **0** sono diverse.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= if Exp then Exp else Exp | Exp ! | ( Exp ) | Id
Id  ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `if x then y else y !`



(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**.

Soluzione: Una possibile soluzione consiste nell'aggiunta del non-terminale `Bang` per poter attribuire precedenza all'operatore unario `!`.

```
Exp ::= if Exp then Exp else Exp | Bang
Bang ::= Bang ! | ( Exp ) | Id
Id  ::= x | y | z
```

2. Considerare la funzione `values : ('a -> bool) -> ('a * 'b) list -> 'b list` che, preso un predicato `p` e una lista di coppie (*chiave, valore*), restituisce la lista dei valori associati alle chiavi che soddisfano il predicato `p`, mantenendo l'ordine iniziale e le possibili ripetizioni.

Esempi:

```
# values (fun k -> k>0) [(1,"one");(0,"zero");(2,"two")]
```

```
- : string list = ["one"; "two"]
```

```
# values (fun k -> k>0) [(1,"b");(2,"b");(0,"a")]
```

```
- : string list = ["b"; "b"]
```

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

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

(c) Definire la funzione `values` 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. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni che denotano insiemi di interi e sono formate dall'operatore binario di unione, dall'operatore unario di complementazione, dai literal di tipo insieme e dagli identificatori di variabile.

```
public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
    T visitComplement(Exp exp);
    T visitUnion(Exp left, Exp right);
    T visitVarId(String name);
    T visitSetLit(java.util.Set<Integer> value);
}

public abstract class UnaryOp implements Exp {
    protected final Exp exp;
    protected UnaryOp(Exp exp) { /* completare */ }
}

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) { /* completare */ }
}

public abstract class AbstractLit<V> implements Exp {
    protected final V value;
    protected AbstractLit(V value) { /* completare */ }
    public int hashCode() { return value.hashCode(); }
}

public class Union extends BinaryOp {
    public Union(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Complement extends UnaryOp {
    public Complement(Exp exp) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class SetLit extends AbstractLit<java.util.Set<Integer>> {
    public SetLit(java.util.Set<Integer> value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!obj instanceof SetLit)
            return false;
        return value.equals(((SetLit) obj).value);
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

- Completare le definizioni dei costruttori di tutte le classi.
- Completare le definizioni dei metodi `accept` delle classi `Union`, `Complement`, `SetLit`, e `VarId`.
- Completare la classe `Eval` che valuta un'espressione restituendo un valore booleano secondo le seguenti regole:
 - l'operatore di complementazione corrisponde alla negazione logica;
 - l'operatore di unione corrisponde alla disgiunzione logica;
 - i literal vengono valutati in `true` se e solo se l'insieme rappresentato non è vuoto;
 - le variabili si valutano sempre in `true`.

Per esempio, la seguente asserzione ha successo:

```
exp = new Complement(new Union(new SetLit(new java.util.HashSet<>()), new VarId("x")));
assert ! exp.accept(new Eval());

public class Eval implements Visitor<Boolean> {
    public Boolean visitComplement(Exp exp) { /* completare */ }
    public Boolean visitUnion(Exp left, Exp right) { /* completare */ }
    public Boolean visitVarId(String name) { /* completare */ }
    public Boolean visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}
```

- Completare la classe `ReplaceVar` che costruisce una nuova espressione ottenuta da quella visitata rimpiazzando le variabili con il literal che rappresenta l'insieme vuoto. Per esempio, la seguente asserzione ha successo:

```

Exp exp = new Complement(new Union(new SetLit(new java.util.HashSet<>()), new VarId("x")));
assert exp.accept(new ReplaceVar()).accept(new Eval());

public class ReplaceVar implements Visitor<Exp> {
    private static final Exp emptyLit = new SetLit(new java.util.HashSet<>());
    public Exp visitComplement(Exp exp) { /* completare */ }
    public Exp visitUnion(Exp left, Exp right) { /* completare */ }
    public Exp visitVarId(String name) { /* completare */ }
    public Exp visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}

```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```

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(Integer i) {
        return super.m(i) + " H.m(Integer) ";
    }
    String m(int i) {
        return super.m(i) + " H.m(int) ";
    }
    String m(Number n) {
        return super.m(n) + " H.m(Number) ";
    }
}

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(42)`
- (b) `p2.m(42)`
- (c) `h.m(42)`
- (d) `p.m(4.2)`
- (e) `p2.m(4.2)`
- (f) `h.m(4, 0)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, tutti i metodi di `P` e `H` sono accessibili da `Test`; si hanno i seguenti casi:

- (a) Il literal `42` ha tipo statico `int` e il tipo statico di `p` è `P`; nessun metodo di `P` è applicabile per sottotipo, ma il metodo con segnatura `m(Number)` è l'unico applicabile per boxing e widening reference conversion poiché $\text{Integer} \leq \text{Number}$ (mentre $\text{Integer} \not\leq \text{Double}$).
A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Number)` in `P`. Viene stampata la stringa `"P.m(Number) "`.
- (b) L'espressione è staticamente corretta e l'overloading viene risolto come al 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 con segnatura `m(Number)` ridefinito in `H`; per la chiamata `super.m(n)`, poiché `n` ha tipo statico `Number`, l'unico metodo della classe `P` applicabile per sottotipo ha segnatura `m(Number)`, quindi viene eseguito lo stesso metodo del punto precedente e viene stampata la stringa `"P.m(Number) H.m(Number) "`.

- (c) Il literal `42` ha tipo statico `int` e il tipo statico di `h` è `H`; l'unico metodo di `H` applicabile per sottotipo ha segnatura `m(int)`.
- A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo dichiarato in `H` con segnatura `m(int)`; per la chiamata `super.m(i)`, poiché `i` ha tipo statico `int`, l'invocazione viene risolta come al punto (a), quindi viene stampata la stringa `"P.m(Number) H.m(int)"`.
- (d) Il literal `4.2` ha tipo statico `double` e il tipo statico di `p` è `P`; nessun metodo di `P` è applicabile per sottotipo, mentre i metodi con segnatura `m(Number)` e `m(Double)` sono entrambi applicabili per boxing e widening reference conversion; poiché `Double ≤ Number`, il metodo più specifico è quello con segnatura `m(Double)`.
- A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Double)` in `P` e viene stampata la stringa `"P.m(Double)"`.
- (e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che `p2` ha lo stesso tipo statico di `p`.
- A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Double)` non viene ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e viene stampata la stringa `"P.m(Double)"`.
- (f) I literal `4` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; dato che `H` non ha metodi con due parametri, né con arità variabile, l'invocazione non è staticamente corretta.