Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 9 giugno

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-z$][a-z_]*)|(((?:\\.[0-9]|[0-9]+\\.?)[0-9]*)[dD])|--|-|(\\s+)");
8        Matcher m = regEx.matcher("$var--.42D.");
9        m.lookingAt();
10       assert m.group(1).equals("var");
11       m.region(m.end(), m.regionEnd());
12       assert m.lookingAt();
13       assert m.group(2) != null;
14       m.region(m.end(), m.regionEnd());
15       assert m.lookingAt();
16       assert m.group(3).equals(".42");
17       m.region(m.end(), m.regionEnd());
18       assert m.lookingAt();
19    }
20 }
```

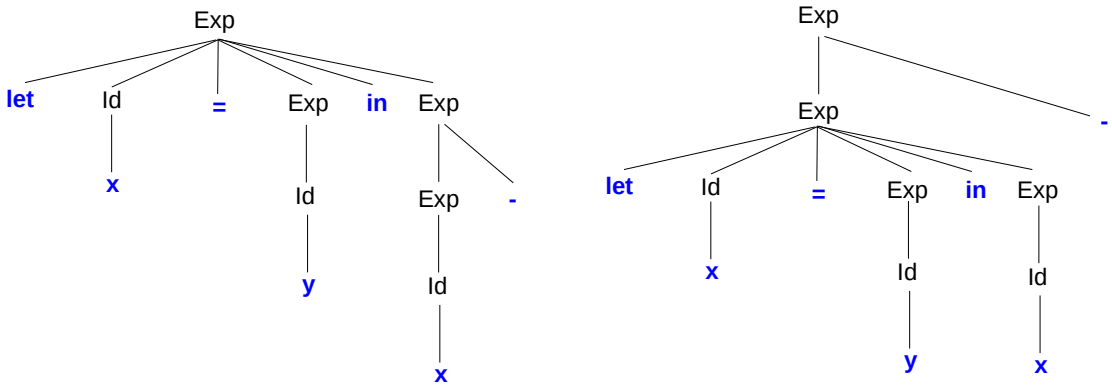
Soluzione:

- `assert m.group(1).equals("var");` (linea 10): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `$var--.42D.` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `$var` (stringa appartenente al gruppo di indice 1), quindi l'asserzione fallisce;
- `assert m.lookingAt();` (linea 12): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `$var` (ossia `-`); l'invocazione di `lookingAt()` restituisce `true` poiché `--` appartiene all'espressione regolare (solo gruppo 0), quindi l'asserzione ha successo;
- `assert m.group(2) != null;` (linea 13): per i motivi del punto precedente l'asserzione fallisce;
- `assert m.lookingAt();` (linea 15): alla linea 14 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `--` (ossia `.`) e l'invocazione del metodo `lookingAt()` ha successo poiché `.42D` appartiene all'espressione regolare (gruppi 0 e 2, mentre la sottostringa `.42` appartiene al gruppo 3), quindi l'asserzione ha successo;
- `assert m.group(3).equals(".42");` (linea 16): per i motivi del punto precedente l'asserzione ha successo;
- `assert m.lookingAt();` (linea 18): alla linea 17 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `--` (ossia `.`) e l'invocazione del metodo `lookingAt()` fallisce poiché `.` non appartiene all'espressione regolare: gli identificatori non contengono il carattere `.` e i numeri possono contenere `.` solo se preceduto o seguito almeno da una cifra numerica.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= let Id = Exp in Exp | Exp ! | ( Exp ) | Id
Id   ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `let x=y in 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**.

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

```
Exp  ::= let Id = Exp in Exp | Bang
Bang ::= Bang ! | ( Exp ) | Id
Id   ::= x | y | z
```

2. Considerare la funzione `range : int -> int -> int list` tale che `range` a b `=` $[a; a+1; a+2; \ldots; b]$; ossia, `range` a b restituisce la lista in ordine strettamente crescente degli elementi compresi nell'intervallo $[a, b]$ (estremi inclusi). Esempi:

```
# range 1 0
- : int list = []
# range 0 0
- : int list = [0]
# range 1 5
- : int list = [1; 2; 3; 4; 5]
```

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

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

(c) Definire senza uso di parametri di accumulazione `step_range : int -> int -> int -> int list` tale che `step_range` a b c `=` $[a; a+c; a+2c; \ldots; b]$, dove c è un intero positivo. Esempi:

```
# step_range 1 0 2
- : int list = []
# step_range 0 0 3
- : int list = [0]
# step_range 1 8 3
- : int list = [1; 4; 7]
```

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni formate dagli operatori binari di congiunzione ($\wedge$) e disgiunzione ($\vee$) logica, dai literal di tipo booleano e dagli identificatori di variabile.

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

public interface Visitor<T> {
    T visitOr(Exp left, Exp right);
    T visitAnd(Exp left, Exp right);
    T visitVarId(String name);
    T visitBoolLit(boolean value);
}
public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) { /* completare */ }
    public Exp getLeft() { return left; }
    public Exp getRight() { return right; }
}
public class Or extends BinaryOp {
    public Or(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 31 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Or))
            return false;
        Or other = (Or) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}
public class And extends BinaryOp {
    public And(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 37 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof And))
            return false;
        And other = (And) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}
public class BoolLit implements Exp {
    protected final boolean value;
    protected BoolLit(boolean value) { /* completare */ }
    public int getValue() { return value; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return Boolean.hashCode(value); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof BoolLit))
            return false;
        return value == ((BoolLit) obj).value;
    }
}
public class VarId implements Exp {
    private final String name;
    public VarId(String name) { /* completare */ }
    public String getName() { return name; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return name.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarId))
            return false;
        return name.equals(((VarId) obj).name);
    }
}
```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi `accept` delle classi `Or`, `And`, `BoolLit`, e `VarId`.

(c) Completare la classe `DisplayPrefix` che permette di visualizzare l'espressione in forma polacca pre-fissa. Per esempio, la seguente asserzione ha successo:

```
assert new And(new BoolLit(true), new Or(new VarId("x"), new VarId("y"))).accept(new DisplayPrefix())
   .equals("/\\ true \\/ x y");

public class DisplayPrefix implements Visitor<String> {
        public String visitOr(Exp left, Exp right) { /* completare */ }
        public String visitAnd(Exp left, Exp right) { /* completare */ }
        public String visitVarId(String name) { /* completare */ }
        public String visitBoolLit(boolean value) { /* completare */ }
}
```

(d) Completare la classe `Simplify` che permette di semplificare un'espressione applicando le seguenti identità: $e \vee \mathit{false} = \mathit{false} \vee e = e$, $e \wedge \mathit{true} = \mathit{true} \wedge e = e$. Per esempio, la seguente asserzione ha successo:

```
Exp exp1 = new And(new Or(new VarId("x"), new BoolLit(true)), new Or(new BoolLit(true), new BoolLit(false)));
Exp exp2 = new Or(new VarId("x"), new BoolLit(true));
assert exp1.accept(new Simplify()).equals(exp2);

public class Simplify implements Visitor<Exp> {
        public Exp visitOr(Exp left, Exp right) { /* completare */ }
        public Exp visitAnd(Exp left, Exp right) { /* completare */ }
        public Exp visitVarId(String name) { /* completare */ }
        public Exp visitBoolLit(boolean value) { /* completare */ }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Integer i) { return "P.m(Integer)"; }
   String m(long l) { return "P.m(long)"; }
}
public class H extends P {
   String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
   String m(Long l) { return super.m(l) + " H.m(Long)";   }
   String m(int... ia) { return super.m(ia[0]) + " H.m(int[])"; }
}
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(new Long(42))`

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

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

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

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

(f) `h.m(42,0)`

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

(a) Il literal `42` ha tipo statico `int`, gli unici due costruttori pubblici della classe `Long` hanno segnatura `Long(long)` e `Long(String)` e il primo è applicabile per sottotipo, quindi l'espressione `new Long(42)` ha tipo statico `Long`. Il tipo statico di `p` è `P` quindi non esistono metodi accessibili in `P` applicabili per sottotipo, ma il metodo con segnatura `m(long)` è accessibile e applicabile per unboxing.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(long)` in `P`. Viene stampata la stringa `"P.m(long)"`.

(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(long)` in `P`, visto che la sottoclasse `H` non ridefinisce il metodo, ma lo eredita da `P`.

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

(c) L'espressione `new Long(42)` ha tipo statico `Long` come già spiegato ai punti precedenti, mentre il tipo statico di `h` è `H`; tra i metodi definiti in `H` e quelli ereditati da `P` solo `m(Long)` è accessibile e applicabile per sottotipo.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Long)` in `H`. L'invocazione `super.m(l)` è staticamente corretta e l'overloading viene risolto come ai punti precedenti e, quindi, viene invocato il metodo `m(long)` in `P`.

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

(d) Il literal `42` ha tipo statico `int`, gli unici due costruttori pubblici della classe `Integer` hanno segnatura `Integer(int)` e `Integer(String)` e il primo è applicabile per sottotipo, quindi l'espressione `new Integer(42)` ha tipo statico `Integer`. Il tipo statico di `p` è `P` e l'unico metodo accessibile in `P` e applicabile per sottotipo ha segnatura `m(Integer)`.

A runtime, il tipo dinamico dell'oggetto in `p` è `P`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `P`.

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

(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`, quindi viene eseguito il metodo con segnatura `m(Integer)` ridefinito in `H`. L'invocazione `super.m(i)` è staticamente corretta e l'overloading viene risolto allo stesso modo, quindi, viene invocato il metodo `m(Integer)` in `P`.

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

(f) I literal `42` e `0` hanno tipo statico `int` e il tipo statico di `h` è `H`; nessun metodo di `H` (inclusi queli ereditati da `P`) è applicabile per sottotipo o per boxing/unboxing, mentre l'unico metodo ad arità variabile `m(int...)` (definito nella classe `H`) è accessibile e applicabile.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(int...)`. L'invocazione `super.m(ia[0])` è staticamente corretta, il tipo statico di `ia[0]` è `int` e l'unico metodo accessibile e applicabile ha segnatura `m(long)`.

Viene stampata la stringa `"P.m(long) H.m(int[])"`.