Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 12 luglio 2017

a.a. 2016/2017

1. (a) Indicare quali delle asserzioni contenute nel seguente codice Java hanno successo e quali falliscono, motivando la risposta.

```
1  Pattern regEx =
2      Pattern.compile("([a-zA-Z][a-zA-Z0-9]+(?:\\.[a-zA-Z][a-zA-Z0-9]+)*)|(\"[^\"\\\\]*\")|(\\s+)");
3  Matcher m = regEx.matcher("java.lang \"java.lang\"");
4  m.lookingAt();
5  assert m.group(1).equals("java.lang");
6  assert m.group(2) == null;
7  m.region(m.end(), m.regionEnd());
8  m.lookingAt();
9  assert m.group(3) != null;
10 assert m.group(0).equals("java.lang");
11 m.region(m.end(), m.regionEnd());
12 m.lookingAt();
13 assert !m.group(2).equals("\"java.lang\"");
14 assert m.group(3) != null;
```

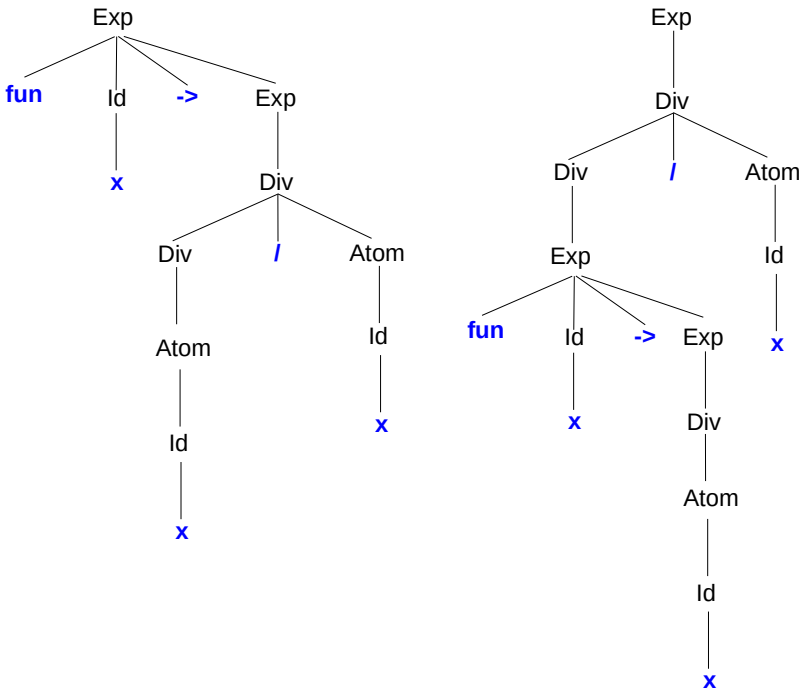
Soluzione:

- `assert m.group(1).equals("java.lang");` (linea 5): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `java.lang "java.lang"` 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 `java.lang` (stringa appartenente ai gruppi di indice 0 e 1), quindi l'asserzione ha successo;
- `assert m.group(2) == null;` (linea 6): lo stato del matcher non è cambiato rispetto alla linea precedente, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(3) != null;` (linea 9): alla linea 7 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `java.lang` (uno spazio bianco), l'invocazione del metodo `lookingAt()` ha successo poiché una successione non vuota di spazi bianchi appartiene all'espressione regolare (gruppi 0 e 3), quindi l'asserzione ha successo;
- `assert m.group(0).equals("java.lang");` (linea 10): l'asserzione fallisce per i motivi del punto precedente, dato che lo stato del matcher è rimasto invariato;
- `assert !m.group(2).equals("\"java.lang\"");` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo allo spazio bianco (carattere `"`) e l'invocazione di `lookingAt()` ha successo poiché la stringa `"java.lang"` è delimitata dai caratteri `"` e contiene al suo interno una successione di caratteri diversi da `"` e `\` e quindi appartiene ai gruppi 0 e 2. Per tali motivi l'asserzione fallisce;
- `assert m.group(3) != null;` (linea 14): l'asserzione fallisce per i motivi del punto precedente, dato che lo stato del matcher è rimasto invariato.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::= fun Id -> Exp | Div
Div  ::= Div / Atom | Atom | Exp
Atom ::= ( Exp ) | Id
Id   ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `fun x->x/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'eliminare la produzione `Div ::= Exp`.

```
Exp  ::= fun Id -> Exp | Div
Div  ::= Div / Atom | Atom
Atom ::= ( Exp ) | Id
Id   ::= x | y | z
```

2. Sia `update : ('a -> bool) -> ('b -> 'b) -> ('a * 'b) list -> ('a * 'b) list` la funzione così specificata:
`update` p f l sostituisce nella lista l ogni coppia (chiave, valore) tale che il predicato p è vero su chiave, con la coppia (chiave, f valore), mentre lascia invariate tutte le altre coppie.

Esempio:

```
# update (fun k -> k>9) String.uppercase
    [(8,"eight");(10,"ten");(4,"four");(11,"eleven")]
- : (int * string) list =
    [(8, "eight"); (10, "TEN"); (4, "four"); (11, "ELEVEN")]
```

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

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

(c) Definire la funzione `update` 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 degli alberi della sintassi astratta (AST) di un semplice linguaggio di espressioni formate a partire da literal di tipo stringa e dagli operatori di addizione intera e di calcolo della lunghezza di una stringa:

```
public interface Visitor<T> {
        T visitStringLit(String value);
        T visitLength(Exp exp);
        T visitAdd(Exp left, Exp right);
}
public class StringLitExp implements Exp {
        private final String value;
        public StringLitExp(String value) { /* da completare */ }
        public <T> T accept(Visitor<T> v) { /* da completare */ }
}
public class LengthExp implements Exp {
   private final Exp exp;
   public LengthExp(Exp exp) { /* da completare */ }
   public <T> T accept(Visitor<T> v) { /* da completare */ }
}
public class AddExp implements Exp {
   private final Exp left, right;
   public AddExp(Exp left, Exp right) { /* da completare */ }
   public <T> T accept(Visitor<T> v) { /* da completare */ }
}
```

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

   (b) Completare le definizioni dei metodi `accept` delle classi `StringLitExp`, `LengthExp` e `AddExp`.

   (c) Completare la classe `Typecheck`, i cui oggetti permettono di effettuare il typechecking (secondo la semantica statica convenzionale) dell'espressione rappresentata dall'AST visitato.

   Esempio:

```
Exp exp = new AddExp(new LengthExp(new StringLitExp("abc")), new LengthExp(new StringLitExp("de")));
assert exp.accept(new Typecheck()) == INT;
```

   Definizioni:

```
public enum Type { INT, STRING }

public class Typecheck implements Visitor<Type> {
   private static Type check(Type expected, Type found) {
      if (expected != found)
         throw new RuntimeException("Expected " + expected + ", found " + found);
      return expected;
   }
   public Type visitStringLit(String value) { /* da completare */ }
   public Type visitLength(Exp exp) { /* da completare */ }
   public Type visitAdd(Exp left, Exp right) { /* da completare */ }
}
```

   (d) Completare la classe `Eval`, i cui oggetti permettono di valutare l'espressione rappresentata dall'AST visitato secondo la semantica dinamica convenzionale.

   Esempio:

```
Exp exp = new AddExp(new LengthExp(new StringLitExp("abc")), new LengthExp(new StringLitExp("de")));
assert exp.accept(new Eval()).equals(new IntValue(5));
```

   Definizioni:

```
public interface Value {
   default int asInt() { throw new ClassCastException(); }
   default String asString() { throw new ClassCastException(); }
}

import static java.util.Objects.requireNonNull;
public abstract class PrimValue<T> implements Value {
   final protected T value;
   protected PrimValue(T value) { this.value = requireNonNull(value); }
   public int asInt() { return (int) value; }
   public String asString() { return (String) value; }
   public int hashCode() { return value.hashCode(); }
}
public class IntValue extends PrimValue<Integer> {
   protected IntValue(int value) { super(value); }
   public boolean equals(Object obj) {
      return this == obj || obj instanceof IntValue && value.equals(((IntValue) obj).value);
   }
}
public class StringValue extends PrimValue<String> {
   protected StringValue(String value) { super(value); }
```

```
        public boolean equals(Object obj) {
            return this == obj || obj instanceof StringValue && value.equals(((StringValue) obj).value);
        }
    }
    public class Eval implements Visitor<Value> {
        public Value visitStringLit(String value) { /* da completare */ }
        public Value visitLength(Exp exp) { /* da completare */ }
        public Value visitAdd(Exp left, Exp right) { /* da completare */ }
    }
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java contenute nello stesso package:

```
public class P {
    String m(Long i) {
        return "P.m(Long)";
    }
    String m(long i) {
        return "P.m(long)";
    }
}
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)";
    }
}
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(42L)`

(b) `p2.m(42L)`

(c) `h.m(42L)`

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

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

(f) `h.m(Integer.valueOf(42))`

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

(a) Il literal `42L` ha tipo statico **`long`**, mentre `p` ha tipo statico `P`; l'unico metodo di `P` accessibile e applicabile per sottotipo ha segnatura `m(`**`long`**`)` poiché **`long`** $\not\leq$ `Long`.

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`**`)` ereditato da `P` e viene stampata la stringa `"P.m(long)"`.

(c) Il literal `42L` ha tipo statico **`long`**, mentre `h` ha tipo statico `H`; l'unico metodo di `H` accessibile e applicabile per sottotipo è quello ereditato da `P` con segnatura `m(`**`long`**`)`, poiché **`long`** $\not\leq$ `Long`, **`long`** $\not\leq$ `Integer` e **`long`** $\not\leq$ **`int`**.

Visto che `h` contiene un'istanza di `H`, il comportamento a runtime del metodo è lo stesso del punto precedente e quindi viene stampata la stringa `"P.m(long)"`.

(d) Il literal `42` ha tipo statico **`int`** e l'unico metodo statico `valueOf` di `Integer` accessibile e applicabile per sottotipo ha segnatura `valueOf(`**`int`**`)` e restituisce un valore di tipo `Integer` (le altre due versioni di `valueOf` hanno segnatura `valueOf(String)` e `valueOf(String,`**`int`**`)` e quindi non sono applicabili per sottotipo), perciò `Integer.valueOf(42)` ha tipo statico `Integer`.

Il tipo statico di `p` è `P` e non esistono metodi di `P` accessibili e applicabili per sottotipo poiché `Integer` $\not\leq$ `Long` e `Integer` $\not\leq$ **`long`**; tramite unboxing l'argomento può essere convertito al tipo **`int`** e dato che **`int`** $\leq$ **`long`**, ma **`int`** $\not\leq$ `Long`, il metodo con segnatura `m(`**`long`**`)` è l'unico applicabile per unboxing e widening primitive conversion.

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)"`.

(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 il comportamento è lo stesso di quello al punto (b).

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

(f) Come a due punti precedenti, il literal `Integer.valueOf(42)` ha tipo statico `Integer`, mentre il tipo statico di `h` è `H`. Poiché `Integer` $\not\leq$ `Long`, `Integer` $\not\leq$ **`long`** e `Integer` $\not\leq$ **`int`**, l'unico metodo della classe `H` accessibile e applicabile per sottotipo ha segnatura `m(Integer)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo con segnatura `m(Integer)` in `H`. L'argomento `i` ha tipo statico `Integer`, quindi come al punto (d) la chiamata **`super`**`.m(i)` viene staticamente risolta con il metodo con segnatura `m(`**`long`**`)` e viene eseguito il metodo di `P` (diretta superclasse di `H`) con tale segnatura.

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