Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta dell'11 febbraio

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 = Pattern.compile("([A-Z][A-Z_]*)|(([0-9]+[\\.0-9]*)[fF])|<=|<|(\\s+)");
7          Matcher m = regEx.matcher("A_B< =3.14F");
8          m.lookingAt();
9          assert m.group(1).equals("A_B");
10         m.region(m.end(), m.regionEnd());
11         assert m.lookingAt();
12         assert m.group(2) == null;
13         m.region(m.end(), m.regionEnd());
14         m.lookingAt();
15         m.region(m.end(), m.regionEnd());
16         assert m.lookingAt();
17         m.find();
18         assert m.group(2).equals("3.14");
19         assert Float.parseFloat(m.group(3)) == Float.parseFloat("3.14");
20     }
21 }
```

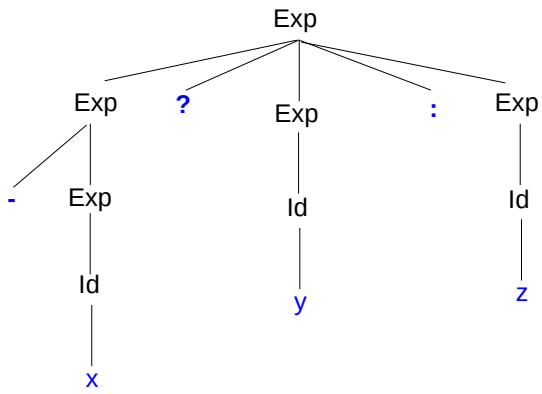
Soluzione:

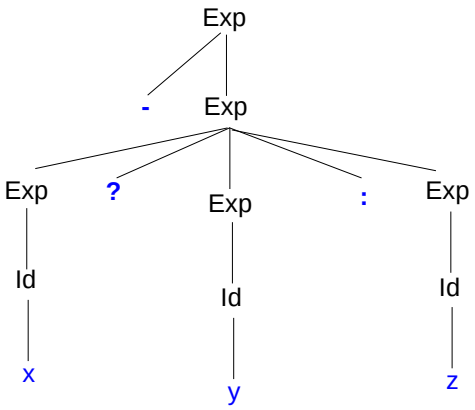
- `assert m.group(1).equals("A_B");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `A_B< =3.14F` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `A_B` (stringa appartenente al gruppo di indice 1), quindi l'asserzione ha successo;
- `assert m.lookingAt();` (linea 11): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo ad `A_B` (`<`); 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 12): per i motivi del punto precedente l'asserzione ha successo;
- `assert m.lookingAt();` (linea 16): alla linea 13 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `<` (spazio bianco) e l'invocazione del metodo `lookingAt()` alla linea 14 ha successo, per cui alla linea successiva la posizione dell'inizio della regione viene spostata in corrispondenza del carattere `=`, quindi l'asserzione fallisce poichè non esistono stringhe che appartengono all'espressione regolare che inizino con `=`;
- `assert m.group(2).equals("3.14");` (linea 18): l'invocazione del metodo `find()` alla linea precedente ha successo poiché dopo il carattere `=` segue la stringa `3.14F` che appartiene interamente al gruppo 2, quindi l'asserzione fallisce;
- `assert Float.parseFloat(m.group(3)) == Float.parseFloat("3.14");` (linea 19): il gruppo 3 contribuisce al successo del metodo `find()` invocato alla linea 17 per quanto riguarda la sotto-stringa `3.14`, quindi l'asserzione ha successo visto che entrambe le invocazioni di `parseFloat` hanno lo stesso argomento e che per tale argomento la conversione è definita.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Exp ? Exp : Exp | - Exp | ( Exp ) | Id
Id   ::= x | y | z
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `-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**.

Soluzione: Una possibile soluzione consiste nell'aggiunta del non-terminale `Minus` per poter attribuire precedenza all'operatore unario **-** e imporre che l'operatore condizionale associ a sinistra.

```
Exp   ::=  Exp ? Exp : Minus | Minus
Minus ::= - Minus | ( Exp ) | Id
Id    ::= x | y | z
```

2. Considerare la funzione `limit : int -> 'a list -> 'a list` così definita:
`limit` n `[`e_1`;`e_2`;...;`e_k`]` restituisce

   - `[`e_1`;`e_2`;...;`e_k`]` se $n \geq k$;
   - `[`e_1`;`e_2`;...;`e_n`]` se $0 < n < k$;
   - `[]` se $n \leq 0$.

   Esempi:

```
# limit (-1) [1;2];;
- : int list = []
# limit 0 [1;2];;
- : int list = []
# limit 1 [1;2];;
- : int list = [1]
# limit 2 [1;2];;
- : int list = [1; 2]
# limit 3 [1;2];;
- : int list = [1; 2]
```

   (a) Definire la funzione `limit` direttamente, senza uso di parametri di accumulazione.

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

   Soluzione: Vedere il file `soluzione.ml`.

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio che include il test di uguaglianza, le espressioni condizionali e gli identificatori di variabile.

```
public interface AST { <T> T accept(Visitor<T> visitor); }
public interface Variable extends AST {
    boolean equals(Object obj);
    int hashCode();
}
public interface Visitor<T> {
    T visitEq(AST left, AST right);
    T visitSimpleVar(SimpleVar var);
    T visitIfThenElse(AST exp, AST thenStmt, AST elseStmt);
```

```
}
public class Eq implements AST {
    private final AST left; // obbligatorio
    private final AST right; // obbligatorio
    public Eq(AST left, AST right) { /* da completare */}
    public <T> T accept(Visitor<T> v) { return v.visitEq(left, right); }
}
public class IfThenElse implements AST {
    private final AST exp; // obbligatorio
    private final AST thenStmt; // obbligatorio
    private final AST elseStmt; // opzionale
    public IfThenElse(AST exp, AST thenStmt, AST elseStmt) { /* da completare */ }
    public IfThenElse(AST exp, AST thenStmt) { /* da completare */ }
    public <T> T accept(Visitor<T> visitor) { return visitor.visitIfThenElse(exp, thenStmt, elseStmt); }
}
public class SimpleVar implements Variable {
    private final String name; // obbligatorio, non vuoto
    public SimpleVar(String name) { /* da completare */ }
    public <T> T accept(Visitor<T> visitor) { return visitor.visitSimpleVar(this); }
    public final boolean equals(Object obj) { /* da completare */ }
    public int hashCode() { /* da completare */ }
}
```

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

(b) Completare le definizioni dei metodi `equals(Object obj)` e `hashCode()` della classe `SimpleVar`.

(c) Completare la classe `GetFreeVars` che permette di restituire l'insieme degli identificatori di variabile contenuti in un AST. Per esempio, per l'AST che rappresenta l'espressione **`if`** `x==y then z` **`else`** `y`, il risultato della visita è l'insieme {`x`,`y`,`z`}.

```
public class GetFreeVars implements Visitor<Set<Variable>> {
    private final Set<Variable> vars = new HashSet<>();
    public Set<Variable> visitEq(AST left, AST right) { /* da completare */ }
    public Set<Variable> visitSimpleVar(SimpleVar var) { /* da completare */ }
    public Set<Variable> visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) {
        /* da completare */
    }
}
```

(d) Completare la classe `ApplySubs` che permette di applicare una sostituzione a un AST. Una sostituzione è una mappa finita da variabili ad AST; per esempio, la sostituzione σ = {`x`↦`w`, `z`↦`z==y`, `u`↦`s`}, sostituisce simultaneamente ogni occorrenza di `x` con `w`, di `z` con `z==y` e di `u` con `s`, mentre lascia invariate le occorrenze di tutte le variabili diverse da `x`, `z` e `u`. Quindi, l'applicazione di σ all'AST dell'espressione **`if`** `x==y then z` **`else`** `y` restituisce l'AST dell'espressione **`if`** `w==y then z==y` **`else`** `y`.

```
public interface Subst {
    /** restituisce la sostituzione per var; restituisce var se non esiste sostituzione per var */
    AST apply(Variable var);
    /** aggiorna la sostituzione */
    void update(Variable var, AST value);
}
public class ApplySubs implements Visitor<AST> {
    private final Subst subst; // obbligatorio
    public ApplySubs(Subst subst) {
        this.subst = requireNonNull(subst);
    }
    public AST visitEq(AST left, AST right) { /* da completare */ }
    public AST visitSimpleVar(SimpleVar var) { /* da completare */ }
    public AST visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) {
        /* da 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(int i) { return super.m(i) + " H.m(int)"; }
    String m(Integer i) { return super.m(i) + " H.m(Integer)"; }
    String m(Object o) { return super.m((Integer) o) + " H.m(Object)"; }
}
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) `p.m(new Long(42))`

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

(e) `h.m((byte)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`**; il tipo statico di `p` è `P` quindi non esistono metodi accessibili in `P` applicabili per sotto-tipo, ma il metodo con segnatura `m(Integer)` è accessibile e applicabile per boxing, mentre l'altro no, dato che `Integer` $\not\leq$ `Long`.

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)"`.

(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 con segnatura `m(Integer)` ridefinito in `H`. Poiché `i` ha tipo statico `Integer`, l'overloading per l'invocazione **`super`**`.m(i)` viene risolta con il metodo con segnatura `m(Integer)` che è l'unico accessibile e applicabile per sotto-tipo.

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

(c) L'espressione **`new`** `Long(42)` ha tipo statico `Long` poiché `42` ha tipo **`int`** che è sotto-tipo del tipo **`long`** del parametro dell'unico costruttore applicabile per sotto-tipo di `Long` (l'altro ha tipo `String`); il tipo statico di `p` è `P`. L'unico metodo della classe `P` applicabile per sotto-tipo ha segnatura `m(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)"`.

(d) L'espressione `42L` ha tipo statico **`long`** e il tipo statico di `h` è `H`; nessun metodo in `H` è applicabile per sotto-tipo, mentre esistono due metodi accessibili e applicabili per boxing (e successivo reference widening in uno dei due casi) con segnature `m(Long)` e `m(Object)`; poiché `Long`$\leq$`Object`, il metodo con segnatura `m(Long)` è il più specifico.

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

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

(e) Il literal `42` ha tipo statico **`int`**, il cast è corretto staticamente (la conversione in questo caso è senza perdita di informazione), l'argomento `(`**`byte`**`)42` ha tipo statico **`byte`**; il tipo statico di `h` è `H` e l'unico metodo accessibile e applicabile per sotto-tipo ha segnatura `m(`**`int`**`)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(`**`int`**`)`. Poiché `i` ha tipo statico **`int`**, l'overloading per l'invocazione **`super`**`.m(i)` viene risolta come al punto (a) e, quindi, viene chiamato il metodo della classe `P` con segnatura `m(Integer)`.

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

(f) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`; nessun metodo di `H` (inclusi queli ereditati da `P`) è applicabile per sotto-tipo, mentre l'unico metodo accessibile e applicabile per boxing e reference widening ha segnatura `m(Object)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo in `H` con segnatura `m(Object)`. In seguito alla conversione per boxing, l'argomento del metodo ha tipo dinamico `Double` che non è sottotipo di `Integer`, quindi il cast `(Integer) o` solleva l'eccezione `ClassCastException`.