Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 21 luglio

a.a. 2014/2015

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  public class MatcherTest {
4      public static void main(String[] args) {
5          Pattern regEx = Pattern
6                  .compile("(foreach|for|(?<FIRST>[a-zA-Z]))(?<REST>[a-zA-Z0-9_]*)|(?<NUM>0[0-7]*)|
7                          (?<SKIP>\\s+)");
8          Matcher m = regEx.matcher("for 0718");
9          assert m.lookingAt();
10         assert m.group("FIRST") != null;
11         assert m.group("REST").equals("for");
12         m.region(m.end(), m.regionEnd());
13         m.lookingAt();
14         assert m.group("SKIP") != null;
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group("NUM").equals("071");
18         assert Integer.parseInt(m.group("NUM"),8) == 57;
19     }
20 }
```

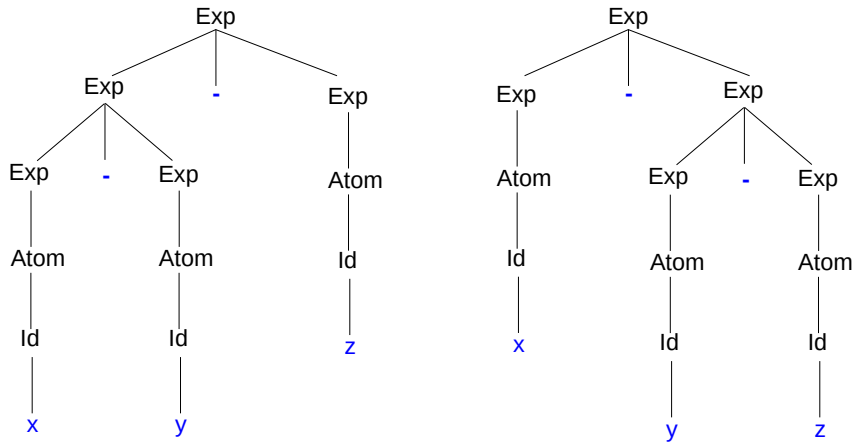
Soluzione:

- `assert m.lookingAt();` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `"for 0718"` 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 `"for"`, ottenuta come concatenazione di `"for"` (nessun gruppo con nome) e `""` (gruppo `REST`), quindi l'asserzione ha successo;
- `assert m.group("FIRST") != null;` (linea 10): `m.group("FIRST")` restituisce il valore `null` poiché il gruppo non contribuisce al ricosnoscimento della stringa `"for"`, per cui l'asserzione fallisce;
- `assert m.group("REST").equals("for");` (linea 11): `m.group("REST")` restituisce la stringa `""` per i motivi indicati al punto 1, per cui l'asserzione fallisce;
- `assert m.group("SKIP") != null;` (linea 14): alla linea 12 l'inizio della regione punta al carattere immediatamente successivo a `"for"` (uno spazio bianco in questo caso) che appartiene al gruppo `SKIP`, quindi `m.group("SKIP")` restituisce la stringa `" "` e l'asserzione ha successo;
- `assert m.group("NUM").equals("071");` (linea 17): alla linea 15 l'inizio della regione punta al carattere immediatamente successivo allo spazio bianco (carattere `0`), l'invocazione `m.lookingAt()` alla linea 16 ha successo poiché la stringa `"071"` appartiene al gruppo `NUM`, quindi `m.group("NUM")` restituisce la stringa `"071"` e l'asserzione ha successo;
- `assert Integer.parseInt(m.group("NUM"),8) == 57;` (linea 18): `m.group("NUM")` restituisce la stringa `"071"` appena riconosciuta, per cui l'asserzione ha successo, visto che in base 8 la stringa rappresenta il valore di tipo `int` `57`.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Exp - Exp | Atom
Atom  ::= Id | - Atom | ( Exp )
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: La soluzione più semplice consiste nel modificare la produzione `Exp ::= Exp - Exp` forzando così l'operatore binario **-** ad associare da sinistra.

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

2. Considerare la funzione `decode : int list -> int` che decodifica un numero binario senza segno rappresentato dalla lista dei suoi bit in ordine inverso (ossia, il bit più significativo è quello più a destra). Convenzionalmente, la lista vuota corrisponde al numero 0.

   Esempio:

```
# decode [0;0;0;1];;
- : int = 8
# decode [1;1;1;1];;
- : int = 15
```

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

   (b) Definire la funzione `decode` direttamente, usando un parametro di accumulazione affinché la ricorsione sia di coda. Per rovesciare la lista utilizzare la funzione di libreria `List.rev`.

   (c) Definire la funzione `decode` 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>
```

   Per rovesciare la lista utilizzare la funzione di libreria `List.rev`.

Soluzione: Vedere il file `soluzione.ml`.

3. (a) Considerare la classe `ArrayBinNum` che implementa i numeri binari memorizzando in un array i loro bit nell'ordine usuale (ossia, a partire dal bit più significativo). Per esempo, le asserzioni nel seguente frammento di codice hanno tutte successo:

```
assert new ArrayBinNum(0,0,0,1).decode() == 1;
assert new ArrayBinNum(0,0,0,1).length() == 4;
assert new ArrayBinNum(1,1,1).decode() == 7;
assert new ArrayBinNum(1,1,1).length() == 3;
```

Completare la seguente definizione della classe `ArrayBinNum`:

```
import java.util.Iterator;
public interface BinNum extends Iterable<Byte> {
    Iterator<Byte> revIterator();
    long decode();
    BinNum add(BinNum other);
    int length();
}
import java.util.Iterator;

public abstract class AbstractBinNum implements BinNum {
    @Override
    public long decode() {
        // da completare
    }
    @Override
    public BinNum add(BinNum other) {
        // da completare
    }
}
import java.util.Iterator;
public class ArrayBinNum extends AbstractBinNum {
    private final byte[] digits;
    /** new ArrayBinNum() equivale a new ArrayBinNum(0) */
    public ArrayBinNum(int... digits) {
        // da completare
    }
    /** itera sulle cifre binarie a partire da quella piu' significativa */
    public Iterator<Byte> iterator() {
        return new BinNumIterator(digits);
    }

    /** itera sulle cifre binarie a partire da quella meno significativa */
    public Iterator<Byte> revIterator() {
        return new RevBinNumIterator(digits);
    }
    /** restituisce il numero di cifre binarie */
    @Override
    public int length() {
        // da completare
    }
}
```

(b) Completare i metodi `decode` e `add` della classe astratta `AbstractBinNum`. Il metodo `decode` decodifica il numero binario.

Per esempio, `assert new ArrayBinNum(1, 1, 1).decode() == 7;` ha successo.

Il metodo `add` restituisce il numero binario ottenuto sommando il numero binario `this` con il numero binario `other`. Per esempio, `new ArrayBinNum(1,1,1).add(new ArrayBinNum(1))` deve restituire un'instanza di `ArrayBinNum` che rappresenta il numero binario `1000`.

(c) Completare la classe `BinNumIterator` che itera sulle cifre binarie.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
class BinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    BinNumIterator(byte[] digits) {
        if (digits == null)
            throw new NullPointerException();
        this.digits = digits;
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Byte next() {
        // da completare
    }
```

```
}
```

(d) Completare la classe `RevBinNumIterator` che itera sulle cifre binarie in ordine inverso.

```
import java.util.Iterator;
import java.util.NoSuchElementException;
class RevBinNumIterator implements Iterator<Byte> {
    private int index;
    private final byte[] digits;
    RevBinNumIterator(byte[] digits) {
        // da completare
    }
    @Override
    public boolean hasNext() {
        // da completare
    }
    @Override
    public Byte next() {
        // da completare
    }
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(byte b) { return "P.m(byte)"; }
    String m(int i) { return "P.m(int)"; }
    String m(Number... n) { return "P.m(Number...)"; }
}
public class H extends P {
    String m(float f) { return super.m(f) + " H.m(float)"; }
    String m(short s) { return super.m(s) + " H.m(short)"; }
    String m(Double... ds) { return super.m(ds) + "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, 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((byte) 42)`

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

(d) `p.m(Short.valueOf((byte) 42))`

(e) `p2.m(new Double[] { 4.2 })`

(f) `h.m(new double[]{4.2})`

Soluzione:

(a) Il literal `42` ha tipo statico `int`; il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.

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

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

(b) Il literal `42` ha tipo statico `int`, il cast a `byte` risulta staticamente corretto e il tipo statico dell'argomento è `byte`; il tipo statico di `p2` è `P` ed esistono due metodi di `P` accessibili e applicabili per sotto-tipo: `String m(byte b)` e `String m(int i)`. Viene selezionato il primo metodo, poiché è il più specifico.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il metodo `String m(byte b)` viene cercato a partire da `H`, ma poiché il metodo non viene ridefinito viene eseguito quello di `P`. La conversione di tipo dovuta al cast

non comporta in questo caso perdita di informazione, dato che la costante 42 è correttamente rappresentabile con un valore di tipo **`byte`**.

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

(c) Il literal `42.` ha tipo statico **`double`**, il cast a **`float`** risulta staticamente corretto e il tipo statico dell'argomento è **`float`**; il tipo statico di `h` è `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(float f)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(float f)` in `H`. La conversione di tipo dovuta al cast non comporta in questo caso perdita di informazione, dato che la costante 42. è correttamente rappresentabile con un valore di tipo **`float`**.

Poiché il parametro `f` ha tipo statico **`float`**, per la chiamata `super.m(f)` non esiste alcun metodo in `P` di arità costante accessibile e applicabile per sotto-tipo o per conversione di tipo boxing/unboxing, mentre il metodo di arità variabile è accessibile e applicabile per boxing da **`float`** a `Float` e reference widening da `Float` a `Number`.

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

(d) L'argomento di `Short.valueOf` ha tipo statico **`byte`** come al punto (b) ed esiste un solo metodo statico nella classe `Short` accessibile e applicabile per sotto-tipo, che restituisce un valore di tipo `Short`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.

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

(e) Il literal `42.` ha tipo statico **`double`**, l'argomento **`new`** `Double[]{4.2}` ha tipo statico `Double[]` (4.2 viene implicitamente convertito tramite boxing) e il tipo statico di `p2` è `P`; esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Number... n)` (durante il controllo di applicabilità per sotto-tipo il parametro `n` viene considerato di tipo `Number[]` e `Double[]` $\leq$ `Number[]` poiché `Double` $\leq$ `Number`.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, quindi il metodo `String m(byte b)` viene cercato a partire da `H`, ma poiché il metodo non viene ridefinito viene eseguito quello di `P`.

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

(f) Il literal `42.` ha tipo statico **`double`**, l'argomento **`new double`**`[]{4.2}` ha tipo statico **`double`**`[]` e il tipo statico di `h` è `H`; poiché **`double`**`[]` $\not\leq$ `Number[]` non esistono metodi accessibili e applicabili, quindi la chiamata non è staticamente corretta.