Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 16 gennaio

a.a. 2014/2015

5 febbraio 2015

1. (a) Dato il seguente codice Java, indicare quali delle asserzioni contenute in esso 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(
7              "(null|(?<HEAD>[a-zA-Z]))(?<TAIL>[a-zA-Z0-9]*)|0[bB](?<NUM>[01]+)|(?<SKIP>\\s+)");
8          Matcher m = regEx.matcher("null3    420b11");
9          assert m.lookingAt();
10         assert m.group("TAIL").length() > 0;
11         assert m.group("HEAD") != null;
12         m.region(m.end(), m.regionEnd());
13         m.lookingAt();
14         assert m.group("SKIP") != null;
15         m.region(m.end(), m.regionEnd());
16         assert m.lookingAt();
17         m.find();
18         assert Integer.parseInt(m.group("NUM"), 2) == 3;
19     }
20 }
```

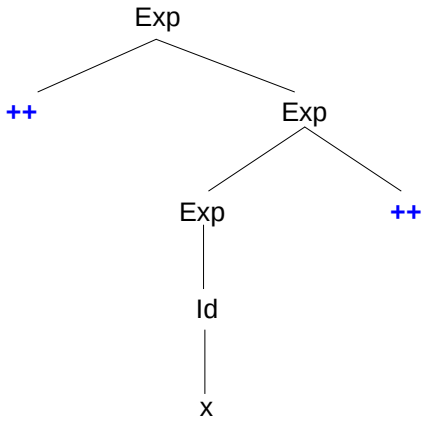
Soluzione:

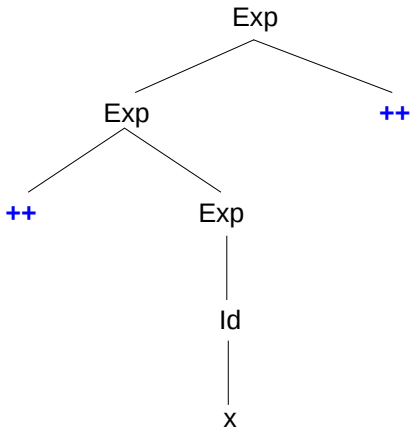
- `assert m.lookingAt();` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `"null3  420b11"` e `lookingAt()` controlla che a partire da tale indice esista una sotto-stringa che appartenga all'insieme definito dall'espressione regolare in `regEx`. Tale sotto-stringa esiste ed è `"null3"`, ottenuta come concatenazione di `"null"` (gruppo 1) e `3` (gruppo `TAIL`), quindi l'asserzione ha successo;
- `assert m.group("TAIL").length() > 0;` (linea 10): per le ragioni al punto precedente, ha successo `assert m.group("TAIL").equals("3")`, per cui l'asserzione ha successo;
- `assert m.group("HEAD") != null;` (linea 11): poiché `m.group(1).equals("null")`, necessariamente `assert m.group("HEAD") == null`, quindi l'asserzione fallisce;
- `assert m.group("SKIP") != null;` (linea 14): alla linea 12 l'inizio della regione viene spostato al carattere immediatamente successivo a `3` (il primo spazio bianco), e la successione di spazi bianchi appartiene al gruppo `SKIP`, quindi l'asserzione ha successo;
- `assert m.lookingAt();` (linea 16): alla linea 15 l'inizio della regione viene spostato al carattere immediatamente successivo all'ultimo spazio bianco (carattere `4`), ma nessuna stringa che inizia con `4` appartiene al linguaggio specificato dall'espressione regolare del matcher, quindi l'asserzione fallisce;
- `assert Integer.parseInt(m.group("NUM"), 2) == 3;` (linea 18): alla linea 17, tramite il metodo `find`, viene trovata la prima sotto-stringa che appartiene al linguaggio specificato dall'espressione regolare del matcher, senza che il primo carattere debba necessariamente coincidere con l'inizio della regione (il carattere `4`); tale stringa è `0b11` e la sua sotto-stringa `11` appartiene al gruppo `NUM`, per cui l'asserzione ha successo, visto che la decodifica di `11` in base 2 senza segno è 3.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp  ::=  Id | ++ Exp | Exp ++
Id ::= x | y | z
```

Basta mostrare due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `++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**.

Esistono due soluzioni diverse, a seconda delle precedenze; volendo dare maggiore precedenza all'operatore postfisso, la grammatica può essere così ridefinita:

```
Exp  ::= Post | ++ Exp
Post ::= Id |  Post ++
Id   ::= x | y | z
```

2. In OCaml la funzione predefinita `max : 'a -> 'a -> 'a` permette di calcolare il massimo tra due elementi dello stesso tipo; per esempio,

```
# max 42 2
- : int = 42

# max "hello" "world"
- : string = "world"
```

Considerare la funzione `max_list :'a -> 'a list -> 'a` tale che `max_list` n l restituisce il massimo tra n e tutti gli elementi della lista l.

Esempio:

```
# max_list 0 [1;2;3;4;4;3;2;1]
- : int = 4

# max_list 10 [1;2;3;4;4;3;2;1]
- : int = 10
```

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

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

(c) Definire la funzione `max_list` 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. (a) Completare la definizione del costruttore e dei metodi della classe `Pair<E1,E2>` che implementa coppie di oggetti rispettivamente di tipo `E1` ed `E2`.

```
public final class Pair<E1, E2> {
    private final E1 first;
    private final E2 second;

    public Pair(E1 first, E2 second) {
    // completare
    }
    public E1 getFirst() {
    // completare
    }
    public E2 getSecond() {
    // completare
    }
    @Override
    public int hashCode() {
    // completare
    }
    @Override
    public boolean equals(Object obj) {
    // completare
    }
    @Override
    public String toString() {
    // completare
    }
}
```

(b) Considerare la classe `ComposeIterator<E1, E2, T>` che permette di comporre, tramite un oggetto di tipo `Composer<E1, E2, T>`, due iteratori rispettivamente di tipo `Iterator<E1>` e `Iterator<E2>`, per ottenere un iteratore di tipo `Iterator<T>`. L'iteratore produce elementi ottenuti componendo, tramite l'oggetto `composer`, le coppie di elementi via via restituite da `firstIterator` e `secondIterator`. L'iterazione termina quando uno dei due iteratori termina.

Esempio di uso:

```
public class AddLengths implements Composer<String, String, Integer> {
    @Override
    public Integer compose(String e1, String e2) {
        return e1.length() + e2.length();
    }
}
...
Iterator<String> it1 = Arrays.asList("a", "ab").iterator();
Iterator<String> it2 = Arrays.asList("", "abc", "a").iterator();
Iterator<Integer> it3 = new ComposeIterator<>(it1, it2, new AddLengths());
while(it3.hasNext())
    // stampa "1 5" ossia la lunghezza di "a"+"" e di "ab"+"abc"
    System.out.print(it3.next()+" ");
```

Completare la definizione del costruttore e dei metodi della classe `ComposeIterator<E1, E2, T>`.

```
public interface Composer<E1, E2, T> {
    T compose(E1 e1, E2 e2);
}
public class ComposeIterator<E1, E2, T> implements Iterator<T> {
    private final Iterator<E1> firstIterator;
    private final Iterator<E2> secondIterator;
    private final Composer<E1, E2, T> composer;

    public ComposeIterator(Iterator<E1> firstIterator,
            Iterator<E2> secondIterator, Composer<E1, E2, T> composer) {
    // completare
    }
    @Override
    public boolean hasNext() {
    // completare
    }
    @Override
    public T next() {
    // completare
    }
}
```

(c) Considerare il metodo `buildPairs` che prende come argomenti due iteratori `it1` e `it2` e restituisce un iteratore che genera le coppie di elementi prodotti da `it1` e `it2`.

Esempio:

```
Iterator<Pair<Integer, String>> it = buildPairs(Arrays.asList(1, 2, 3).iterator(),
                                                Arrays.asList("one", "two", "three").iterator());
while (it.hasNext())
    // stampa "(1,one) (2,two) (3,three)"
    System.out.print(it.next() + " ");
```

Completare la definizione del metodo `buildPairs` **utilizzando la classe** `ComposeIterator` e definire la classe ausiliaria usata per creare il necessario oggetto di tipo `Composer`.

```
public static <E1, E2> Iterator<Pair<E1, E2>> buildPairs(Iterator<E1> it1,
                                                         Iterator<E2> it2) {
// completare
}
```

Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(int i) {
        return "P.m(int)";
    }
    String m(Object o) {
        return "P.m(Object)";
    }
}
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(int... is) {
        return "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((short)42)`

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

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

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

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

(f) `p2.m(4,2)`

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 cast è staticamente corretto (narrowing primitive conversion) e l'espressione `(short) 42` ha tipo **`short`**; in accordo con la sua dichiarazione, 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`. Durante la conversione da **`int`** a **`short`** non c'è perdita di informazione visto che 42 è rappresentabile in complemento a 2 su 16 bit e, ovviamente, non c'è perdita di informazione nella conversione inversa da **`short`** a **`int`** dovuta al passaggio del parametro.

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

(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 `String m(int i)` in `H`. Per le conversioni a cui è soggetto il literal `42` valgono le stesse considerazioni del punto precedente. Poiché `i` ha tipo statico **`int`**, per l'invocazione **`super`**`.m(i)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(int i)`.

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

(c) Il literal `42` ha tipo statico **`int`** e per l'invocazione `Integer.valueOf(42)` esiste un solo metodo statico nella classe `Integer` accessibile e applicabile, che restituisce un valore di tipo `Integer`. Il tipo statico di `p` è `P` ed esiste un solo metodo di `P` accessibile e applicabile per sotto-tipo, `String m(Object i)`.

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

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

(d) 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`, ma poiché `H` non definisce alcun metodo `"P.m(Object)"`, viene eseguito il metodo ereditato dalla superclasse `P`.

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

(e) Per gli stessi motivi dei punti (c) e (d), l'espressione `Integer.valueOf(42)` ha tipo statico `Integer`, mentre `h` ha tipo statico `H` ed esiste un solo metodo di `H` accessibile e applicabile per sotto-tipo, `String m(Integer i)`.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo `String m(Integer i)` in `H`. Poiché `i` ha tipo statico `Integer`, per l'invocazione **`super`**`.m(i)` esiste un solo metodo in `P` accessibile e applicabile per sotto-tipo, `String m(Object o)`.

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

(f) La variabile `p2` ha tipo statico `P` e poiché in `P` non esiste alcun metodo accessibile e applicabile a due argomenti, l'espressione non è staticamente corretta.