Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta del 9 settembre 2019

a.a. 2019/2020

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 =
6              Pattern.compile("(\\s+)|([0-3]?[0-9]/[0-1]?[0-9]/[0-9][0-9])|(0[xX]([A-Fa-f0-9]+))");
7          Matcher m = regEx.matcher("9/9/19 0XFF");
8          m.lookingAt();
9          assert m.group(2).equals("09/09/19");
10         assert m.group(0).equals("9/9/19");
11         m.region(m.end(), m.regionEnd());
12         m.lookingAt();
13         assert m.group(2) != null;
14         assert m.group(0).equals("0XFF");
15         m.region(m.end(), m.regionEnd());
16         m.lookingAt();
17         assert m.group(3).equals("0XFF");
18         assert m.group(4).equals("FF");
19     }
20 }
```

Soluzione:

- `assert m.group(2).equals("09/09/19");` (linea 9): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `9/9/19 0XFF` 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 `9/9/19` (appartenente ai soli gruppi 0 e 2, super-insieme delle date valide), quindi l'asserzione fallisce poiché le stringhe `9/9/19` e `09/09/19` sono diverse;
- `assert m.group(0).equals("9/9/19");` (linea 10): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione ha successo;
- `assert m.group(2) != null;` (linea 13): alla linea 11 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `9/9/19` (ossia uno spazio bianco) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché una successione non vuota di spazi bianchi appartiene alla sotto-espressione regolare `\\s+` corrispondente ai soli gruppi 0 e 1, quindi l'asserzione fallisce;
- `assert m.group(0).equals("0XFF");` (linea 14): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente l'asserzione fallisce;
- `assert m.group(3).equals("0XFF");` (linea 17): alla linea 15 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `0`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `0XFF` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0, 3 e 4 (gruppo 3: literal esadecimali, gruppo 4: literal esadecimali senza i primi due caratteri); per tale motivo, l'asserzione ha successo;
- `assert m.group(4).equals("FF");` (linea 18): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi precedenti l'asserzione ha successo.

(b) Mostrare che la seguente grammatica è ambigua.

```
Exp ::= Exp * Atom | Exp + Atom | Atom
Atom ::=  [ ExpSeq ] | Id
ExpSeq ::= ExpSeq Exp | Exp
Id ::= a | b
```

Soluzione: La grammatica **non** è ambigua e stabilisce che $*$ e $+$ hanno stessa precedenza e associano a sinistra; allo stesso modo, le sequenze di espressioni associano a sinistra.

2. Sia `count_zeros : ('a -> int) -> 'a list -> int` la funzione così specificata:

   `count_zeros` f l restituisce il numero di elementi e della lista l per i quali vale l'uguaglianza $f(e) = 0$.

   Esempi:

```
# count_zeros (fun x->(x-1)*(x-2)*(x+3)) [-3;1;2;0;4]
- : int = 3
# count_zeros (fun x->(x-1)*(x-2)*(x+3)) [-1;0;4]
- : int = 0
```

   (a) Definire `count_zeros` senza uso di parametri di accumulazione.

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

   (c) Definire `count_zeros` come specializzazione della funzione `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

   Soluzione: Vedere il file `soluzione.ml`.

3. (a) Completare le seguenti classi che implementano i valori di tipo intero e stringa.

```
public interface Value {
    public default String asString() {throw new RuntimeException("Expecting a string");}
    public default int asInt() {throw new RuntimeException("Expecting an integer");}
}
// valori primitivi generici
public abstract class PrimVal<T> implements Value {
    protected T val;
    // invariante di classe: val!=null
    protected PrimVal(T val) { /* completare */ }
}
public class IntVal extends PrimVal<Integer> {
    protected IntVal(Integer val) { /* completare */ }
    @Override public int asInt() { /* completare */ }
}
public class StringVal extends PrimVal<String> {
    protected StringVal(String val) { /* completare */ }
    @Override public String asString() { /* completare */ }
}
```

   (b) Completare la classe `Eval` per la valutazione delle espressioni. Nel metodo `visitTimes(Exp left, Exp right)` (prodotto tra stringhe e interi) l'operando `left` si deve valutare in una stringa s e `right` in un intero i; il risultato calcolato è la concatenazione di s ripetuta i volte. Viene sollevata `RuntimeException` se i valori degli operandi non sono del tipo giusto e `IllegalArgumentException` se $i < 0$.

```
public class Eval implements Visitor<Value> {
    @Override public Value visitIntLit(int val) { /* completare */ }
    @Override public Value visitStringLit(String val) { /* completare */ }
    @Override public Value visitTimes(Exp left, Exp right) { /* completare */ }
}
```

   Suggerimento: per il calcolo della concatenazione, utilizzare il seguente metodo della classe predefinita `String`:

```
// Returns a string whose value is the concatenation of this string repeated count times.
public String repeat(int count)
```

   Soluzione: Vedere il file `soluzione.jar`.

4. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    String m(Number... o) { return "P.m(Number...)"; }
    String m(Object... o) { return "P.m(Object...)"; }
}
public class H extends P {
    String m(String s) { return super.m(s) + " H.m(String)"; }
    String m(Double d1, Double d2) { return super.m(d1, d2) + " H.m(Double,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("42")`

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

(d) `p.m(42.0)`

(e) `p2.m(42.0)`

(f) `h.m(42.0)`

Soluzione: assumendo che tutte le classi siano dichiarate nello stesso package, tutti i metodi sono accessibili.

(a) Il tipo statico di `p` è `P`, il literal `"42"` ha tipo statico `String`.

- prima fase (solo sottotipo): poiché `String` $\not\leq$ `Number[]` e `String` $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi array `Number[]` e `Object[]`), non esistono metodi di `P` applicabili per sottotipo;
- seconda fase (boxing/unboxing e sottotipo): nessuna conversione boxing/unboxing è applicabile al tipo `String`, quindi anche in questo caso non esistono metodi applicabili;
- terza fase (arità variabile, boxing/unboxing e sottotipo): è corretto considerare che i due metodi abbiano un solo parametro di tipo `Number` e `Object`; poiché `String` $\not\leq$ `Number`, ma `String` $\leq$ `Object`, solo il metodo `m(Object...)` di `P` è applicabile per boxing e reference widening.

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

(b) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Object...)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(Object...)"`.

(c) Il tipo statico di `h` è `H` mentre l'argomento ha sempre tipo statico `String`.

- prima fase (solo sottotipo): il metodo con due parametri non è applicabile a un solo argomento; inoltre, poiché `String` $\not\leq$ `Number[]`, `String` $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi array `Number[]` e `Object[]`) e `String` $\leq$ `String`, l'unico metodo di `H` applicabile per sottotipo ha segnatura `m(String)`;

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(String)`; poiché il parametro `s` ha tipo statico `String` e **`super`** si riferisce alla classe `P`, la chiamata **`super`**`.m(s)` si comporta come al punto (a); viene quindi stampata la stringa `"P.m(Object...) H.m(String)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`.

- prima fase (solo sottotipo): poiché **`double`** $\not\leq$ `Number[]` e **`double`** $\not\leq$ `Object[]` (nella prima e seconda fase `Number...` e `Object...` vengono considerati come i tipi array `Number[]` e `Object[]`), non esistono metodi di `P` applicabili per sottotipo;
- seconda fase (boxing/unboxing e sottotipo): il tipo **`double`** è convertibile a `Double` per boxing, ma poiché `Double` $\not\leq$ `Number[]` e `Double` $\not\leq$ `Object[]`, non esistono metodi di `P` applicabili;
- terza fase (arità variabile, boxing/unboxing e sottotipo): è corretto considerare che i due metodi abbiano un solo parametro di tipo `Number` e `Object`; poiché `Double` $\leq$ `Number` e `Double` $\leq$ `Object`, entrambi i metodi sono applicabili, ma dato che `Number` $\leq$ `Object`, il metodo con segnatura `m(Number...)` è più specifico.

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

(e) L'espressione è staticamente corretta e l'overloading viene risolto come al punto precedente, visto che i tipi statici sono gli stessi.

A runtime, il tipo dinamico dell'oggetto in `p2` è `H`, ma poiché il metodo con segnatura `m(Number...)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(Number...)"`.

(f) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `h` è `H`. I due metodi in `H` non sono comunque applicabili, quindi la risoluzione della chiamata procede come nei due punti precedenti; infatti, il metodo con segnatura `m(String)` non è applicabile perché non è possibile convertire un argomento da **`double`** a `String`, mentre quello con segnatura `m(Double,Double)` non è applicabile a un solo argomento perché ha due parametri.

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi come al punto precedente viene eseguito il metodo con segnatura `m(Number...)` in `P` e viene stampata la stringa `"P.m(Number...)"`.