Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta parziale dell' 11 febbraio 2019

a.a. 2018/2019

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

```
1  Pattern regEx = Pattern.compile("([A-Z][_a-zA-Z]*)|(int|bool|double)|(\\s+)");
2  Matcher m = regEx.matcher("Is_bool double");
3  m.lookingAt();
4  assert m.group(2) == null;
5  assert m.group(0).equals("Is_bool");
6  m.region(m.end(), m.regionEnd());
7  m.lookingAt();
8  assert m.group(2) == null;
9  assert m.group(3) != null;
10 m.region(m.end(), m.regionEnd());
11 m.lookingAt();
12 assert m.group(2) != null;
13 assert m.group(0).equals("double");
```

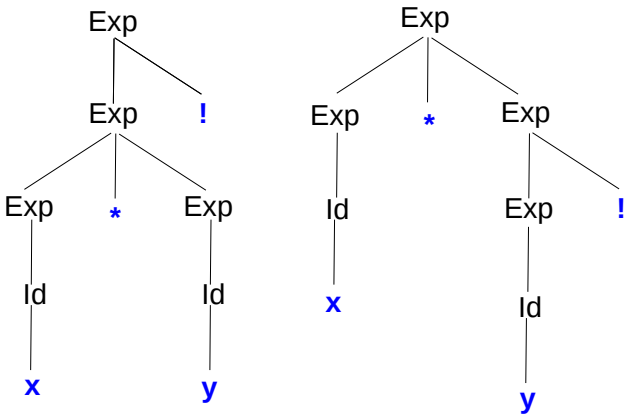
Soluzione:

- `assert m.group(2) == null;` (linea 4): la regione del matcher inizia dall'indice 0, corrispondente al primo carattere della stringa `Is_bool double` 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 `Is_bool` (appartenente ai soli gruppi 0 e 1: qualsiasi stringa non vuota che inizia con una lettera maiuscola seguita da zero o più lettere maiuscole, minuscole o _), quindi il metodo restituisce **`null`** e l'asserzione ha successo;
- `assert m.group(0).equals("Is_bool");` (linea 5): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(0)` restituisce un oggetto corrispondente alla stringa `Is_bool` e l'asserzione ha successo;
- `assert m.group(2) == null;` (linea 8): alla linea 6 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo a `Is_bool` (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 corrispondente ai soli gruppi 0 e 3, quindi `m.group(2)` restituisce **`null`** e l'asserzione ha successo;
- `assert m.group(3) != null;` (linea 9): lo stato del matcher non è cambiato rispetto alla linea sopra, quindi per i motivi del punto precedente `m.group(1)` restituisce un oggetto di tipo `String`, quindi diverso da **`null`**, e l'asserzione ha successo;
- `assert m.group(2) != null;` (linea 12): alla linea 10 l'inizio della regione viene spostata alla posizione corrispondente al carattere immediatamente successivo agli spazi bianchi (ossia `d`) e l'invocazione del metodo `lookingAt()` restituisce **`true`** poiché la stringa `double` appartiene alla sotto-espressione regolare corrispondente ai soli gruppi 0 e 2 (stringa `int`, `bool` o `double`); per tale motivo, `m.group(2)` restituisce un oggetto che rappresenta la stringa `double`, quindi diverso da **`null`**, e l'asserzione ha successo;
- `assert m.group(0).equals("double");` (linea 13): 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 ! | Exp * Exp | < Exp > | Id
Id ::= x | y
```

Soluzione: Basta esibire due diversi alberi di derivazione per una stessa stringa del linguaggio, per esempio `x * y !`



(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 `Bang` per poter attribuire la precedenza all'operatore unario `!` e forzare l'associatività (a sinistra) dell'operatore binario $\star$.

```
Exp  ::= Exp * Bang | Bang
Bang  ::= Bang ! |  < Exp > | Id
Id ::= x | y
```

2. Sia `cond_map : ('a -> 'b) -> ('a -> 'b) -> ('a -> bool) -> 'a list -> 'b list` la funzione così specificata:

   `cond_map` f g p l restituisce la lista ottenuta da l applicando, nell'ordine, la funzione f agli elementi di l che soddisfano il predicato p e la funzione g a quelli che non lo soddisfano.

   Esempio:

```
# cond_map  sqrt (fun x -> 0.) (fun x -> x>=0.0) [-1.0;9.0;-4.0;4.0]
- : float list = [0.; 3.; 0.; 2.]
```

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

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

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

Soluzione: Vedere il file `soluzione.ml`.

3. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
   String m(Object o) { return "P.m(Object)"; }
   String m(Number n) { return "P.m(Number)"; }
}
public class H extends P {
   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, 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, si hanno i seguenti casi:

(a) Il literal `42` ha tipo statico **`int`** e il tipo statico di `p` è `P`; poiché **`int`** $\not\leq$ `Object` e **`int`** $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da **`int`** a `Integer` e osservando che `Integer` $\leq$ `Object` e `Integer` $\leq$ `Number`, entrambi i metodi di `P` sono applicabili per boxing e reference widening. Dato che `Number` $\leq$ `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

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)"`.

(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(Number)` non è ridefinito in `H`, viene eseguito lo stesso metodo del punto precedente e, quindi, viene stampata la stringa `"P.m(Number)"`.

(c) Il literal `42` ha tipo statico **`int`** e il tipo statico di `h` è `H`, l'unico metodo accessibile e applicabile per sottotipo ha segnatura `m(`**`int`**`)`, viste le considerazioni sui metodi ereditati da `P` riportate al punto (a).

A runtime, il tipo dinamico dell'oggetto in `h` è `H`, quindi viene eseguito il metodo di `H` con segnatura `m(`**`int`**`)`; poiché il parametro `i` ha tipo statico **`int`** e **`super`** si riferisce alla classe `P`, la risoluzione e il comportamento della chiamata **`super`**`.m(i)` coincidono con il caso illustrato al punto (a); viene quindi stampata la stringa `"P.m(Number) H.m(int)"`.

(d) Il literal `42.0` ha tipo statico **`double`** e il tipo statico di `p` è `P`; poiché **`double`** $\not\leq$ `Object` e **`double`** $\not\leq$ `Number`, non esistono metodi di `P` accessibili e applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da **`double`** a `Double` e osservando che `Double` $\leq$ `Object` e `Double` $\leq$ `Number`, entrambi i metodi di `P` sono applicabili per boxing e reference widening. Dato che `Number` $\leq$ `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

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`; per le considerazioni riportate al punto (a) e poiché **`double`** $\not\leq$ **`int`**, né il metodo definito in `H`, né quelli ereditati da `P` sono applicabili per sottotipo; tuttavia, applicando una conversione di tipo boxing da **`double`** a `Double` e osservando che `Double` $\leq$ `Object` e `Double` $\leq$ `Number`, ma `Double` $\not\leq$ **`int`**, i soli metodi ereditati da `P` sono applicabili per boxing e reference widening. Dato che `Number` $\leq$ `Object`, la chiamata viene risolta con il metodo che ha la segnatura più specifica `m(Number)`.

A runtime, il tipo dinamico dell'oggetto in `h` 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)"`.