Linguaggi e Programmazione Orientata agli Oggetti

Prova scritta

a.a. 2013/2014

9 Giugno 2014

1. Dato il seguente frammento di codice Java, indicare quali delle asserzioni contenute in esso falliscono, motivando la risposta.

```
import java.util.regex.*;

public class PatternTest {

    private static String NUMBER = "(?<NUMBER>0[xX][0-9a-fA-F]([0-9a-fA-F_]*[0-9a-fA-F])?[lL]?)";
    private static String IDENT = "(?<IDENT>[a-zA-Z_][a-zA-Z0-9_]*)";

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("\\s+|").append(NUMBER)
                .append("|").append(IDENT);
        Pattern p = Pattern.compile(sb.toString());
        Matcher m = p.matcher("x00xA 0XA__45__FLL00");
        m.find();
        assert m.group("IDENT").equals("x00xA");
        m.find();
        assert m.group("NUMBER") != null;
        assert m.group("IDENT") == null;
        assert m.group().equals("\\s");
        m.find();
        assert m.group("NUMBER").equals("0XA__45__FL");
        m.find();
        assert m.group("IDENT") == null;
    }
}
```

2. (a) Mostrare che la seguente grammatica è ambigua.

```
Type  ::=  Prim | Type -> Type | ( Type )
Prim ::= int | bool
```

   (b) Modificare la grammatica definita al punto precedente in modo che **non sia ambigua** e che il linguaggio generato a partire dal non terminale `Type` **resti invariato**.

3. Considerare la funzione `map_pairs : ('a * 'b -> 'c) -> ('a * 'b) list -> 'c list` che, presa una funzione f e una lista di coppie $[(e'_1, e''_1); \ldots; (e'_n, e''_n)]$, restituisce la lista $[f(e'_1, e''_1); \ldots; f(e'_n, e''_n)]$.

   Esempio:

```
map_pairs (function x, y -> x+y) [];;
- : int list = []
map_pairs (function x, y -> x+y) [0, 1];;
- : int list = [1]
map_pairs (function x, y -> x+y) [1, 2; 3, 4; 5, 6];;
- : int list = [3; 7; 11]
```

   Senza utilizzare la funzione `map` di sistema:

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

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

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

   Se volete, potete utilizzare la funzione standard `List.rev`, che "rovescia" una lista; per esempio:

```
# List.rev [1;2;3]
- : int list = [3; 2; 1]
```

4. Considerare le seguenti dichiarazioni:

   - la gerarchia di classi che ha radice `MovieFilter` rappresenta dei filtri di ricerca per dei film (classe `Movie`)
   - l'interfaccia generica `MovieFilterVisitor<T>` rappresenta un *visitor* che effettua un'operazione, su un filtro, e restituisce un oggetto di tipo `T`.

   Più in dettaglio, la classe `TitleContains` rappresenta il filtro che controlla se una certa stringa `aString` è contenuta nel titolo del film; un oggetto di tipo `TitleContains` deve poter essere costruito a partire da una `String`a.

   La classe `HasGenre` rappresenta il filtro che controlla se un film ha un certo `genre` (per esempio, *commedia* o *fantascienza*), codificato come una stringa Java; un oggetto di tipo `HasGenre` deve poter essere costruito a partire da una `String`a. Notate che ogni film, `Movie`, può avere un numero arbitrario di generi.

   Infine, le classi `And` e `Or` rappresentano dei filtri costruiti facendo l'*And*-logico (risp., *Or*-logico) di una sequenza, non vuota, di filtri.

   Tutte queste classi devono fornire degli opportuni *getter*.

```
interface MovieFilterVisitor<T> {
        T visit(TitleContains titleContainsFilter);
        T visit(HasGenre hasGenreFilter);
        T visit(Or orFilter);
        T visit(And andFilter);
}

class Movie {
        private String _title;
        private String [] _genres;
        public Movie(String _title, String... _genres) {
                this._title = _title;
                this._genres = _genres;
        }
        public String getTitle() { return this._title; }
        public String[] getGenres() { return this._genres; }
        @Override public String toString() { return this._title; }
        // ...
}

abstract class MovieFilter {
        abstract <T> T accept(MovieFilterVisitor<T> visitor);
}

class TitleContains extends MovieFilter {
    /* DA COMPLETARE (1) */
}

class HasGenre extends MovieFilter {
    /* DA COMPLETARE (2) */
}

abstract class CombinedFilter extends MovieFilter {
        final private MovieFilter [] _filters;
        protected CombinedFilter(MovieFilter... filters) {
                if (filters.length==0)
                        throw new IllegalArgumentException("filters");
                this._filters = filters;
        }
        public MovieFilter [] getSubFilters() { return _filters; }
}

class And extends CombinedFilter {
        public And(MovieFilter... _filters) { super(_filters); }
        @Override public <T> T accept(MovieFilterVisitor<T> visitor) { /* DA COMPLETARE (3) */ }
}

class Or extends CombinedFilter {
        public Or(MovieFilter... _filters) { super(_filters); }
        @Override public <T> T accept(MovieFilterVisitor<T> visitor) { /* DA COMPLETARE (4) */ }
}
```

   (a) Completare i quattro frammenti "DA COMPLETARE (...)" nelle definizioni delle classi.

   (b) Completare la seguente definizione del *visitor* "valutazione" `Eval`. Un oggetto di tipo `Eval`, costruito a partire da un certo `Movie movie`, deve implementare la visita che restituisce un valore di verità che corrisponde al fatto che il film `movie` soddisfi il filtro.

```
class Eval implements MovieFilterVisitor<Boolean> {
    private final Movie _movie;
    public Eval(Movie movie) { this._movie = movie; }
    @Override public Boolean visit(TitleContains titleContainsFilter) {
        /* DA COMPLETARE (6) */
    }
    @Override public Boolean visit(HasGenre hasGenreFilter) {
        /* DA COMPLETARE (7) */
    }
```

```
        @Override public Boolean visit(And andFilter) {
            /* DA COMPLETARE (8) */
        }
        @Override public Boolean visit(Or orFilter) {
            /* DA COMPLETARE (9) */
        }
    }
```

(c) Implementare la definizione del *visitor* `ToString`, calcola una rappresentazione testuale di un filtro. La rappresentazione, dei vari tipi di filtro, deve essere la seguente:

- `TitleContains`: stringa `"title contains <...>"`, dove i puntini vanno sostituiti con la stringa cercata
- `HasGenre`: stringa `"has genre <...>"`, dove i puntini vanno sostituiti con la stringa cercata
- `And`: stringa `"( ... )"`, dove i puntini vanno sostituiti dalle rappresentazioni stringa dei sotto-filtri, separati dalla stringa `" and "`
- `Or`: stringa `"( ... )"`, dove i puntini vanno sostituiti dalle rappresentazioni stringa dei sotto-filtri, separati dalla stringa `" or "`

5. Considerare le seguenti dichiarazioni di classi Java:

```
public class P {
    Object m(Object... os) {
        return "P.m(Object...)";
    }

    Object m(Object o) {
        return "P.m(Object)";
    }

}
public class H extends P {
    String m(Number... ns) {
        return super.m(ns) + " H.m(Number...)";
    }

    String m(Number n) {
        return (super.m(n)) + " H.m(Number)";
    }

    String m(long i) {
        return (super.m(i)) + " H.m(long)";
    }

}
import static java.lang.System.out;

public class Test {

    public static void main(String[] args) {
        P p1 = new P();
        H h = new H();
        P p2 = h;
        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) `p1.m(42l)`

(b) `h.m(42l)`

(c) `p2.m(42l)`

(d) `p1.m(42l, 42.0)`

(e) `h.m(42l, 42.0)`

(f) `p2.m(42l, 42.0)`