Esercizio 1

Hierarchy of classes representing the DSL forms:

```
public class Parser {

public class Compiler {

    private Dictionary<String, int> keyword = new Dictionary<string, int>(){

public class Token {

public class Form {

public class Block {

public class StatementList {

public class Statement {

public class Control {

public class StatementListHead {

public class StatementListTail {

public class Type {

public class TypeTail {

public class Guard {

public class ConditionList {

public class ConditionListTail {

public class Condition {

public class ConditionHead {

public class ConditionTail {

public class Expr {

public class ExprTail {

public class Term {

public class TermTail {

public class Factor {

public class Num {

public class Base {

public class NumTail {

public class Exponent {

public class ExponentTail {

public class Atomic<T> {
```

Esercizio 2

The grammar from wich I have made the parser:

⟨*Form*⟩ ::= 'form' Id ⟨*Block*⟩

⟨*Block*⟩ ::= '`{`' ⟨*StatementList*⟩ '`}`'

⟨*StatementList*⟩ ::= ⟨*Statement*⟩ ⟨*StatementListHead*⟩
| ⟨*Control*⟩ ⟨*StatementList*⟩
| ⟨*empty*⟩

⟨*Statement*⟩ ::= Id '`:`' String ⟨*Type*⟩

⟨*Control*⟩ ::= '`if`' '`(`' ⟨*Guard*⟩ '`)`' ⟨*Block*⟩

⟨*StatementListHead*⟩ ::= '`,`' ⟨*StatementListTail*⟩
| ⟨*empty*⟩

⟨*StatementListTail*⟩ ::= ⟨*Statement*⟩ ⟨*StatementListHead*⟩
| ⟨*Control*⟩ ⟨*StatementList*⟩

⟨*Type*⟩ ::= '`integer`' ⟨*TypeTail*⟩
| '`real`' ⟨*TypeTail*⟩
| '`boolean`' ⟨*TypeTail*⟩
| '`money`' ⟨*TypeTail*⟩
| '`date`' ⟨*Typetail*⟩
| '`string`' ⟨*TypeTail*⟩

⟨*TypeTail*⟩ ::= '`(`' ⟨*Expr*⟩ '`)`'
| ⟨*empty*⟩

⟨*Guard*⟩ ::= ⟨*ConditionList*⟩
| ⟨*empty*⟩

⟨*ConditionList*⟩ ::= ⟨*Condition*⟩ ⟨*ConditionListTail*⟩
| ⟨*empty*⟩

⟨*ConditionListTail*⟩ ::= '`&&`' ⟨*Condition*⟩ ⟨*ConditionList*⟩
| '`||`' ⟨*Condition*⟩ ⟨*ConditionList*⟩
| ⟨*empty*⟩

⟨*Condition*⟩ ::= ⟨*ConditionHead*⟩ ⟨*ConditionTail*⟩

⟨*ConditionHead*⟩ ::= ⟨*Expr*⟩
| '`!`' ⟨*Expr*⟩

⟨*ConditionTail*⟩ ::= '`<`' ⟨*ConditionHead*⟩
| '`>`' ⟨*ConditionHead*⟩
| '`<=`' ⟨*ConditionHead*⟩
| '`>=`' ⟨*ConditionHead*⟩
| '`==`' ⟨*ConditionHead*⟩
| '`!=`' ⟨*ConditionHead*⟩
| ⟨*empty*⟩

⟨*Expr*⟩ ::= ⟨*Term*⟩ ⟨*ExprTail*⟩

⟨*ExprTail*⟩ ::= '`+`' ⟨*Term*⟩ ⟨*ExprTail*⟩
| '`-`' ⟨*Term*⟩ ⟨*ExprTail*⟩
| ⟨*empty*⟩

⟨*Term*⟩ ::= ⟨*Factor*⟩ ⟨*TermTail*⟩

⟨*TermTail*⟩ ::= '`*`' ⟨*Factor*⟩ ⟨*TermTail*⟩
| '`/`' ⟨*Factor*⟩ ⟨*TermTail*⟩
| ⟨*empty*⟩

⟨*Factor*⟩ ::= '`(`' ⟨*Expr*⟩ '`)`'
| ⟨*Num*⟩
| String
| Bool

⟨*Num*⟩ ::= ⟨*Base*⟩ ⟨*NumTail*⟩
| '`-`' ⟨*Base*⟩ ⟨*NumTail*⟩

⟨*Base*⟩ ::= Integer
| Real
| Id

⟨*NumTail*⟩ ::= '`^`' ⟨*Exponent*⟩
| ⟨*empty*⟩

⟨*Exponent*⟩ ::= '`-`' ⟨*ExponentTail*⟩
| ⟨*ExponentTail*⟩

⟨*ExponentTail*⟩ ::= Integer
| Id

```
public class Parser {
    protected Tokenizer t;
    protected Token lookahead;
    public Parser() { }
    public T parse<T>(String s) {
        t = new Tokenizer(s);
        lookahead = t.nextToken();
        T p = (T)(Object)Form();
        Match(type.EOF);
        return p;
    }
    protected Form Form() {
        Match(type.FORM);
        string id = lookahead.value;
        Match(type.ID);
        return new Form(id, Block());
    }
    protected Block Block() {
        Match(type.OPEN_CURLY);
        StatementList sl = StatementList();
        Match(type.CLOSE_CURLY);
        return new Block(sl);
    }
    protected StatementList StatementList() {
        if (lookahead.type == (int)type.ID) {
            return new StatementList(Statement(), StatementListHead());
        } else if (lookahead.type == (int)type.IF) {
            return new StatementList(Control(), StatementList());
        } else return null;
    }
    protected StatementListHead StatementListHead() {
        if (lookahead.type == (int)type.COMMA) {
            Match(type.COMMA);
            return new StatementListHead(StatementListTail());
        } else return null;
    }
    protected StatementListTail StatementListTail() {
        if (lookahead.type == (int)type.ID) {
            return new StatementListTail(Statement(), StatementListHead());
        } else {
            return new StatementListTail(Control(), StatementList());
        }
    }
    protected Control Control() {
        Atomic<string> ifCode = new Atomic<string>(lookahead.value);
        Match(type.IF);
        Match(type.OPEN_PAR);
        Guard g = Guard();
        Match(type.CLOSE_PAR);
        return new Control(ifCode, g, Block());
    }
    protected Statement Statement() {
        string id = lookahead.value;
        Match(type.ID);
        Match(type.COLON);
        string stringValue = lookahead.value;
        Match(type.STRING);
        return new Statement(id, stringValue, Type());
    }
    protected Type Type() {
        int found = lookahead.type;
        Atomic<string> typeString = new Atomic<string>(lookahead.value);
        Match((type)found);
        return new Type(typeString, TypeTail());
    }
    protected virtual TypeTail TypeTail() {
        if (lookahead.type == (int)type.OPEN_PAR) {
            Match(type.OPEN_PAR);
            Expr e = Expr();
            Match(type.CLOSE_PAR);
            return new TypeTail(e);
        } else return null;
    }
    protected Guard Guard() {
        if (lookahead.type == (int)type.INTEGER || lookahead.type == (int)type.REAL ||
            lookahead.type == (int)type.ID || lookahead.type == (int)type.MINUS ||
            lookahead.type == (int)type.STRING || lookahead.type == (int)type.BOOL ||
```

```
            lookahead.type == (int)type.OPEN_PAR || lookahead.type == (int)type.NOT) {
            return new Guard(CondictionList());
        } else return null;
    }
    protected ConditionList CondictionList() {
        if (lookahead.type == (int)type.INTEGER || lookahead.type == (int)type.REAL ||
            lookahead.type == (int)type.ID || lookahead.type == (int)type.MINUS ||
            lookahead.type == (int)type.STRING || lookahead.type == (int)type.BOOL ||
            lookahead.type == (int)type.OPEN_PAR || lookahead.type == (int)type.NOT) {
            return new ConditionList(Condiction(), CondictionListTail());
        } else return null;
    }
    protected ConditionListTail CondictionListTail() {
        if (lookahead.type == (int)type.AND) {
            Atomic<string> op = new Atomic<string>(lookahead.value);
            Match(type.AND);
            return new ConditionListTail(op, Condiction(), CondictionList());
        } else if (lookahead.type == (int)type.OR) {
            Atomic<string> op = new Atomic<string>(lookahead.value);
            Match(type.OR);
            return new ConditionListTail(op, Condiction(), CondictionList());
        } else return null;
    }
    protected Condition Condiction() {
        return new Condition(CondictionHead(), CondictionTail());
    }
    protected ConditionHead CondictionHead() {
        Atomic<char> not = null;
        if (lookahead.type == (int)type.NOT) {
            Match(type.NOT);
            not = new Atomic<char>('!');
        }
        return new ConditionHead(not, Expr());
    }
    protected CondictionTail CondictionTail() {
        if (lookahead.type == (int)type.LT || lookahead.type == (int)type.GT ||
            lookahead.type == (int)type.LE || lookahead.type == (int)type.GE ||
            lookahead.type == (int)type.EQUAL || lookahead.type == (int)type.DISEQUAL) {
            int found = lookahead.type;
            Atomic<string> op = new Atomic<string>(lookahead.value);
            Match((type)found);
            return new CondictionTail(op, CondictionHead());
        } else return null;
    }
    protected Expr Expr() {
        return new Expr(Term(), ExprTail());
    }
    protected ExprTail ExprTail() {
        if (lookahead.type == (int)type.PLUS) {
            Match(type.PLUS);
            return new ExprTail(new Atomic<char>('+'), Term(), ExprTail());
        } else if (lookahead.type == (int)type.MINUS) {
            Match(type.MINUS);
            return new ExprTail(new Atomic<char>('-'), Term(), ExprTail());
        } else return null;
    }
    protected Term Term() {
        return new Term(Factor(), TermTail());
    }
    protected TermTail TermTail() {
        if (lookahead.type == (int)type.TIMES) {
            Match(type.TIMES);
            return new TermTail(new Atomic<char>('*'), Factor(), TermTail());
        } else if (lookahead.type == (int)type.SLASH) {
            Match(type.SLASH);
            return new TermTail(new Atomic<char>('/'), Factor(), TermTail());
        } else return null;
    }
    protected Factor Factor() {
        if (lookahead.type == (int)type.OPEN_PAR) {
            Match(type.OPEN_PAR);
            Expr e = Expr();
            Match(type.CLOSE_PAR);
            return new Factor(e);
        } else if (lookahead.type == (int)type.STRING) {
            string id = lookahead.value;
```

```
            Match(type.STRING);
            return new Factor(id, "STRING");
        } else if (lookahead.type == (int)type.BOOL) {
            string boolValue = lookahead.value;
            Match(type.BOOL);
            return new Factor(boolValue.Equals("true"));
        } else return new Factor(Num());
    }
    protected Num Num() {
        Atomic<char> minus = null;
        if (lookahead.type == (int)type.MINUS) {
            Match(type.MINUS);
            minus = new Atomic<char>('-');
        }
        return new Num(minus, Base(), NumTail());
    }
    protected Base Base() {
        if (lookahead.type == (int)type.INTEGER) {
            int intValue = Convert.ToInt32(lookahead.value);
            Match(type.INTEGER);
            return new Base(intValue);
        } else if (lookahead.type == (int)type.REAL) {
            double doubleValue = Convert.ToDouble(lookahead.value);
            Match(type.REAL);
            return new Base(doubleValue);
        } else {
            string id = lookahead.value;
            Match(type.ID);
            return new Base(id);
        }
    }
    protected NumTail NumTail() {
        if (lookahead.type == (int)type.POWER) {
            Match(type.POWER);
            return new NumTail(Exponent());
        } else return null;
    }
    protected Exponent Exponent() {
        Atomic<char> minus = null;
        if (lookahead.type == (int)type.MINUS) {
            Match(type.MINUS);
            minus = new Atomic<char>('-');
        }
        return new Exponent(minus, ExponentTail());
    }
    protected ExponentTail ExponentTail() {
        if (lookahead.type == (int)type.INTEGER) {
            int intValue = Convert.ToInt32(lookahead.value);
            Match(type.INTEGER);
            return new ExponentTail(intValue);
        } else {
            string id = lookahead.value;
            Match(type.ID);
            return new ExponentTail(id);
        }
    }
    protected void Match(type t) {
        Debug.Assert(lookahead.type == (int)t, "Syntax error expected ");
        lookahead = this.t.nextToken();
    }
}

    private Dictionary<String, int> keyword = new Dictionary<string, int>(){
        {"true",(int)type.BOOL},{"false",(int)type.BOOL},{"if",(int)type.IF},{"form",(int)type.FOR
        {"integer",(int)type.TYPE_INT},{"real",(int)type.TYPE_REAL},{"boolean",(int)type.TYPE_BOO
        {"date",(int)type.TYPE_DATE},{"money",(int)type.TYPE_MONEY},{"string",(int)type.TYPE_STRIN
    public Tokenizer(String expr) {
        this.s = expr;
        this.idx = 0;
        this.tokenList = new LinkedList<Token>();
    }
    public Token nextToken() {
        String lexeme;
        Token t;
        if (idx >= s.Length) return new Token("EOF".ToString(), (int)type.EOF);
        if (s[idx] == ',') return new Token(s[idx++].ToString(), (int)type.COMMA);
        if (s[idx] == '^') return new Token(s[idx++].ToString(), (int)type.POWER);
```

```
if (s[idx] == '+') return new Token(s[idx++].ToString(), (int)type.PLUS);
if (s[idx] == '-') return new Token(s[idx++].ToString(), (int)type.MINUS);
if (s[idx] == '*') return new Token(s[idx++].ToString(), (int)type.TIMES);
if (s[idx] == '/') return new Token(s[idx++].ToString(), (int)type.SLASH);
if (s[idx] == '(') return new Token(s[idx++].ToString(), (int)type.OPEN_PAR);
if (s[idx] == ')') return new Token(s[idx++].ToString(), (int)type.CLOSE_PAR);
if (s[idx] == '[') return new Token(s[idx++].ToString(), (int)type.OPEN_SQUARE);
if (s[idx] == ']') return new Token(s[idx++].ToString(), (int)type.CLOSE_SQUARE);
if (s[idx] == ':') return new Token(s[idx++].ToString(), (int)type.COLON);
if (s[idx] == '}') return new Token(s[idx++].ToString(), (int)type.CLOSE_CURLY);
if (s[idx] == '{') return new Token(s[idx++].ToString(), (int)type.OPEN_CURLY);
if (s[idx] == '&') {
    if (s[idx + 1] == '&') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.AND);
        idx += 2;
        ignoreBlanks = true;
    } else {
        ignoreBlanks = false;
        t = new Token(s[idx++].ToString(), (int)type.INVALID_TOKEN);
    }
} else if (s[idx] == '|') {
    if (s[idx + 1] == '|') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.OR);
        idx += 2;
        ignoreBlanks = true;
    } else {
        ignoreBlanks = false;
        t = new Token(s[idx++].ToString(), (int)type.INVALID_TOKEN);
    }
} else if (s[idx] == '<') {
    if (s[idx + 1] == '=') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.LE);
        idx += 2;
        ignoreBlanks = true;
    } else {
        t = new Token(s[idx++].ToString(), (int)type.LT);
    }
} else if (s[idx] == '>') {
    if (s[idx + 1] == '=') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.GE);
        idx += 2;
        ignoreBlanks = true;
    } else {
        t = new Token(s[idx++].ToString(), (int)type.GT);
    }
} else if (s[idx] == '=') {
    if (s[idx + 1] == '=') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.EQUAL);
        idx += 2;
        ignoreBlanks = true;
    } else {
        ignoreBlanks = false;
        t = new Token(s[idx++].ToString(), (int)type.INVALID_TOKEN);
    }
} else if (s[idx] == '!') {
    if (s[idx + 1] == '=') {
        string str = s[idx].ToString() + s[idx + 1].ToString();
        t = new Token(str, (int)type.DISEQUAL);
        idx += 2;
        ignoreBlanks = true;
    } else {
        t = new Token(s[idx++].ToString(), (int)type.NOT);
    }
} else if (s[idx] == '"') {
    lexeme = s[idx++].ToString();
    while (idx < s.Length && (isChar(s[idx]) || isDigit(s[idx]) || isBlank(s[idx])) && s[
        lexeme += s[idx++];
    }
    lexeme += s[idx++];
    t = new Token(lexeme.ToString(), (int)type.STRING);
} else if (isDigit(s[idx])) {
    int dot = 0;
```

```
            lexeme = s[idx++].ToString();
            while (idx < s.Length && isDigit(s[idx])) {
                lexeme += s[idx++];
            }
            if (idx < s.Length - 1 && s[idx] == '.') {
                dot++;
                lexeme += s[idx++];
            }
            while (idx < s.Length && isDigit(s[idx])) {
                lexeme += s[idx++];
            }
            if (dot == 0) {
                t = new Token(lexeme.ToString(), (int)type.INTEGER);
            } else if (dot == 1) {
                t = new Token(lexeme.ToString(), (int)type.REAL);
            } else {
                t = new Token(lexeme.ToString(), (int)type.INVALID_TOKEN);
            }
        } else if (isChar(s[idx])) {
            lexeme = s[idx++].ToString();
            while (idx < s.Length && isChar(s[idx])) {
                lexeme += s[idx++];
            }
            if (keyword.ContainsKey(lexeme)) {
                t = new Token(lexeme.ToString(), keyword[lexeme]);
            } else {
                t = new Token(lexeme.ToString(), (int)type.ID);
            }
        } else {
            if (isBlank(s[idx]) && ignoreBlanks) {
                idx++;
                t = nextToken();
            } else {
                t = new Token(s[idx++].ToString(), (int)type.INVALID_TOKEN);
            }
        }
        return t;
    }
    private bool isDigit(char c) {return (c >= '0' && c <= '9');}
    private bool isChar(char c) {return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');}
    private bool isBlank(char c) {return (c != ' ') || (c != '\r') || (c != '\n');}
}

public class Token {
    public string value { get; private set; }
    public int type { get; private set; }
    public Token(String value, int type) {
        this.value = value;
        this.type = type;
    }
}

public class Form {
    public Atomic<string> formCode { get; private set; }
    public string id { get; private set; }
    public Block b { get; private set; }
    public Form(string id, Block b) {
        this.formCode = new Atomic<string>("form");
        this.id = id;
        this.b = b;
    }
}

public class Block {
    public Atomic<char> openCurly { get; private set; }
    public StatementList sl { get; private set; }
    public Atomic<char> closeCurly { get; private set; }
    public Block(StatementList sl) {
        this.openCurly = new Atomic<char>('{');
        this.sl = sl;
        this.closeCurly = new Atomic<char>('}');
    }
}

public class StatementList {
    public Statement s { get; private set; }
    public StatementListHead slh { get; private set; }
```

```
    public Control c { get; private set; }
    public StatementList sl { get; private set; }
    public StatementList(Statement s, StatementListHead slh) {
        this.s = s;
        this.slh = slh;
    }
    public StatementList(Control c, StatementList sl) {
        this.c = c;
        this.sl = sl;
    }
}

public class Statement {
    public string id { get; private set; }
    public Atomic<char> colon { get; private set; }
    public string stringValue { get; private set; }
    public Type t { get; private set; }
    public Statement(string id, string stringValue, Type t) {
        this.id = id;
        this.colon = new Atomic<char>(':');
        this.stringValue = stringValue;
        this.t = t;
    }
}

public class Control {
    public Atomic<string> ifCode { get; private set; }
    public Atomic<char> openPar { get; private set; }
    public Guard g { get; private set; }
    public Atomic<char> closePar { get; private set; }
    public Block b { get; private set; }
public Control(Atomic<string> ifCode, Guard g, Block b) {
        this.ifCode = new Atomic<string>("if");
        this.openPar = new Atomic<char>('(');
        this.g = g;
        this.closePar = new Atomic<char>(')');
        this.b = b;
    }
}

public class StatementListHead {
    public Atomic<char> comma { get; private set; }
    public StatementListTail slt { get; private set; }
    public StatementListHead(StatementListTail slt) {
        this.comma = new Atomic<char>(',');
        this.slt = slt;
    }
}

public class StatementListTail {
    public Atomic<char> comma { get; private set; }
    public Statement s { get; private set; }
    public Control c { get; private set; }
    public StatementListHead slh { get; private set; }
    public StatementList sl { get; private set; }
    public StatementListTail(Statement s, StatementListHead slh) {
        this.comma = new Atomic<char>(',');
        this.s = s;
        this.slh = slh;
    }
    public StatementListTail(Control c, StatementList sl) {
        this.comma = new Atomic<char>(',');
        this.c = c;
        this.sl = sl;
    }
}

public class Type {
    public Atomic<string> typeString { get; private set; }
    public TypeTail tt { get; private set; }
    public Type(Atomic<string> typeString, TypeTail tt) {
        this.typeString = typeString;
        this.tt = tt;
    }
}

public class TypeTail {
    public Atomic<char> openPar { get; private set; }
```

```
    public Expr e { get; private set; }
    public Atomic<char> closePar { get; private set; }
    public Atomic<char> openSquare { get; private set; }
    public SelectList sl { get; private set; }
    public Atomic<char> closeSquare { get; private set; }
    public string id { get; private set; }
    public TypeTail(Expr e) {
        this.openPar = new Atomic<char>('(');
        this.e = e;
        this.closePar = new Atomic<char>(')');
    }
    public TypeTail(SelectList sl, string id) {
        this.openSquare = new Atomic<char>('[');
        this.sl = sl;
        this.closeSquare = new Atomic<char>(']');
        this.id = id;
    }
}

public class Guard {
    public ConditionList cl { get; private set; }
    public Guard(ConditionList cl) {
        this.cl = cl;
    }
}

public class ConditionList {
    public Condition c { get; private set; }
    public ConditionListTail clt { get; private set; }
    public ConditionList(Condition c, ConditionListTail clt) {
        this.c = c;
        this.clt = clt;
    }
}

public class ConditionListTail {
    public Atomic<string> op { get; private set; }
    public Condition c { get; private set; }
    public ConditionList cl { get; private set; }
    public ConditionListTail(Atomic<string> op, Condition c, ConditionList cl) {
        this.op = op;
        this.c = c;
        this.cl = cl;
    }
}

public class Condition {
    public ConditionHead ch { get; private set; }
    public CondictionTail ct { get; private set; }
    public Condition(ConditionHead ch, CondictionTail ct) {
        this.ch = ch;
        this.ct = ct;
    }
}

public class ConditionHead {
    public Atomic<char> not { get; private set; }
    public Expr e { get; private set; }
    public ConditionHead(Atomic<char> not, Expr e) {
        this.not = not;
        this.e = e;
    }
}

public class CondictionTail {
    public Atomic<string> op { get; private set; }
    public ConditionHead ch { get; private set; }
    public CondictionTail(Atomic<string> op, ConditionHead ch) {
        this.op = op;
        this.ch = ch;
    }
}

public class Expr {
    public Term t { get; private set; }
    public ExprTail et { get; private set; }
    public Expr(Term t, ExprTail et) {
```

```
        this.t = t;
        this.et = et;
    }
}

public class ExprTail {
    public Atomic<char> op { get; private set; }
    public Term t { get; private set; }
    public ExprTail et { get; private set; }
    public ExprTail(Atomic<char> op, Term t, ExprTail et) {
        this.op = op;
        this.t = t;
        this.et = et;
    }
}

public class Term {
    public Factor f { get; private set; }
    public TermTail tt { get; private set; }
    public Term(Factor f, TermTail tt) {
        this.f = f;
        this.tt = tt;
    }
}

public class TermTail {
    public Atomic<char> op { get; private set; }
    public Factor f { get; private set; }
    public TermTail tt { get; private set; }
    public TermTail(Atomic<char> op, Factor f, TermTail tt) {
        this.op = op;
        this.f = f;
        this.tt = tt;
    }
}

public class Factor {
    public Atomic<char> openPar { get; private set; }
    public Expr e { get; private set; }
    public Atomic<char> closePar { get; private set; }
    public string id { get; private set; }
    public Num n { get; private set; }
    public string stringValue { get; private set; }
    public bool boolValue { get; private set; }
    public string type { get; private set; }
    public Factor(Expr e) {
        this.openPar = new Atomic<char>('(');
        this.e = e;
        this.closePar = new Atomic<char>(')');
    }
    public Factor(Num n) {
        this.n = n;
    }
    public Factor(string value, string type) {
        if (type == "STRING") this.stringValue = value;
        else this.id = value;
        this.type = type;
    }
    public Factor(bool boolValue) {
        this.boolValue = boolValue;
    }
}

public class Num {
    public Atomic<char> minus { get; private set; }
    public Base b { get; private set; }
    public NumTail nt { get; private set; }
    public Num(Atomic<char> minus, Base b, NumTail nt) {
        this.minus = minus;
        this.b = b;
        this.nt = nt;
    }
}

public class Base {
    public string type { get; private set; }
    public int intValue { get; private set; }
    public double doubleValue { get; private set; }
```

```
    public string id { get; private set; }
    public Base(int intValue) {
        this.intValue = intValue;
        this.type = "INTEGER";
    }
    public Base(double doubleValue) {
        this.doubleValue = doubleValue;
        this.type = "REAL";
    }
    public Base(string id) {
        this.id = id;
        this.type = "ID";
    }
}

public class NumTail {
    public Atomic<char> power { get; private set; }
    public Exponent e { get; private set; }
    public NumTail(Exponent e) {
        this.power = new Atomic<char>('^');
        this.e = e;
    }
}

public class Exponent {
    public Atomic<char> minus { get; private set; }
    public ExponentTail et { get; private set; }
    public Exponent(Atomic<char> minus, ExponentTail et) {
        this.minus = minus;
        this.et = et;
    }
}

public class ExponentTail {
    public string type { get; private set; }
    public string id { get; private set; }
    public int intValue { get; private set; }
    public ExponentTail(int intValue) {
        this.intValue = intValue;
        this.type = "INTEGER";
    }
    public ExponentTail(string id) {
        this.id = id;
        this.type = "ID";
    }
}

public class Atomic<T> {
    public T value { get; private set; }
    public Atomic(T value) {
        this.value = value;
    }
}
```

Esercizio 3

```
var GUIRenderer = function(obj)
{
    var form = $('form');
    var div = $(document.createElement('div'));
    var tag = $(document.createElement(obj.tag)).attr({'id' : obj.id});
    var label = $(document.createElement('label')).attr('for', obj.id).text(obj.label);
    div.attr('hidden', obj.hidden ? 'hidden' : false);
    tag.attr('disabled', obj.readOnly ? 'disabled' : false);
    switch(obj.type)
    {
        case 'integer':
        case 'real':
        case 'money':
            tag.attr('type', 'number');
            if(obj.expr)
            {
                var evalExpr = function() {tag[0].value = eval(obj.expr);};
                setInterval(evalExpr, 1000);
            }
            var updateNumber = function() { window[obj.id] = parseFloat(tag[0].value); };
```

```
            updateNumber();
            tag.on('change', updateNumber);
            tag.on('input', updateNumber);
        break;
        case 'boolean':
            tag.attr('type', 'checkbox');
            if(obj.expr)
            {
                var evalExpr = function() {tag[0].checked = eval(obj.expr);};
                setInterval(evalExpr, 100);
            }
            var updateBoolean = function() { window[obj.id] = tag[0].checked; };
            updateBoolean();
            tag.on('change', updateBoolean);
        break;
        case 'date':
            tag.attr('type', 'date');
        break;
        case 'string':
            tag.attr('type', 'text');
            if(obj.expr)
            {
                var evalExpr = function() {tag[0].value = eval(obj.expr);};
                setInterval(evalExpr, 100);
            }
            var updateText = function() { window[obj.id] = tag[0].value; };
            updateText();
            tag.on('change', updateText);
            break;
    }
    if(obj.list)
    {
        for (var key in obj.list)
        {
            var option = $(document.createElement('option')).text(key).attr('value', obj.list[key
            tag.append(option);
        }
        tag.on('change', function() {
            var target = $('#'.concat(obj.target));
            target[0].value = this.value;
        target.trigger('change')});
    }
    if(obj.cond)
    {
        setInterval(function()
        {
            try {if(eval(obj.cond))
                div.prop('hidden', false);
            else div.prop('hidden', 'hidden');}
            catch(e){}
        }, 100);
    }
    form.append(div.append(label).append(tag));
};
```

Esercizio 4

```
public class Compiler {
    protected bool hidden = false;
    protected string cond;
    public Compiler() { }
    public void compile(Form f) {
        string path = @"C:\Users\gianlu\Desktop\PA\output\" + f.id + ".html";
        StringBuilder output = new StringBuilder();
        output.Append("<!DOCTYPE html><head><script src=\"jquery-1.11.1.min.js\"></script>\n<scrip
        output.Append(compileBlock(f.b));
        output.Append("}</script></body></html>");
        Console.WriteLine(output.ToString());
        System.IO.File.WriteAllText(path, output.ToString());
    }
    protected string compileBlock(Block b) {
        StringBuilder output = new StringBuilder();
        try { output.Append(compileStatementList(b.sl)); }
        catch (ArgumentNullException) { };
        return output.ToString();
    }
    protected string compileStatementList(StatementList sl) {
```

```
    if (sl == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    if (sl.c != null) {
        output.Append(compileControl(sl.c));
        try { output.Append(compileStatementList(sl.sl)); }
        catch (ArgumentNullException) { }
    } else {
        output.Append(compileStatement(sl.s));
        try { output.Append(compileStatementListHead(sl.slh)); }
        catch (ArgumentNullException) { }
    }
    return output.ToString();
}
protected string compileStatementListHead(StatementListHead slh) {
    if (slh == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(compileStatementListTail(slh.slt));
    return output.ToString();
}
protected string compileStatementListTail(StatementListTail slt) {
    StringBuilder output = new StringBuilder();
    if (slt.c != null) {
        output.Append(compileControl(slt.c));
        try { output.Append(compileStatementList(slt.sl)); }
        catch (ArgumentNullException) { }
    } else {
        output.Append(compileStatement(slt.s));
        try { output.Append(compileStatementListHead(slt.slh)); }
        catch (ArgumentNullException) { }
    }
    return output.ToString();
}
protected string compileStatement(Statement s) {
    StringBuilder output = new StringBuilder();
    if (hidden)
        output.Append(string.Format("GUIRenderer({{\"hidden\" : {0} , \"id\" : \"{1}\" , \"lal
            hidden.ToString().ToLower(), s.id, s.stringValue, cond));
    else
        output.Append(string.Format("GUIRenderer({{\"hidden\" : {0} , \"id\" : \"{1}\" , \"lal
            hidden.ToString().ToLower(), s.id, s.stringValue));
    output.Append(compileType(s.t));
    output.AppendLine("});");
    return output.ToString();
}
protected string compileType(Type t) {
    StringBuilder output = new StringBuilder();
    output.Append(string.Format(",\"type\" : \"{0}\"", t.typeString.value));
    try { output.Append(compileTypeTail(t.tt)); }
    catch (ArgumentNullException) {
        output.Append(string.Format(", \"tag\" : \"input\", \"readOnly\" : false"));
    }
    return output.ToString();
}
protected virtual string compileTypeTail(TypeTail tt) {
    if (tt == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(string.Format(", \"tag\" : \"input\" , \"readOnly\" : true , \"expr\" : \"{
    return output.ToString();
}
protected string compileControl(Control c) {
    StringBuilder output = new StringBuilder();
    try { cond = compileGuard(c.g).ToString(); }
    catch (ArgumentNullException) { }
    hidden = true;
    output.Append(compileBlock(c.b));
    hidden = false;
    cond = "";
    return output.ToString();
}
protected string compileGuard(Guard g) {
    StringBuilder output = new StringBuilder();
    if (g == null) throw new ArgumentNullException();
    try { output.Append(compileCondictionList(g.cl)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
```

```
protected string compileCondictionList(ConditionList cl) {
    if (cl == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    try { output.Append(compileCondiction(cl.c)); }
    catch (ArgumentNullException) { }
    try { output.Append(compileCondictionListTail(cl.clt)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileCondictionListTail(ConditionListTail clt) {
    if (clt == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(clt.op.value);
    output.Append(compileCondiction(clt.c));
    try { output.Append(compileCondictionList(clt.cl)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileCondiction(Condition c) {
    StringBuilder output = new StringBuilder();
    output.Append(compileCondictionHead(c.ch));
    try { output.Append(compileCondictionTail(c.ct)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileCondictionHead(ConditionHead ch) {
    StringBuilder output = new StringBuilder();
    if (ch.not != null) output.Append(ch.not.value);
    output.Append(compileExpr(ch.e));
    return output.ToString();
}
protected string compileCondictionTail(CondictionTail ct) {
    if (ct == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(ct.op.value);
    output.Append(compileCondictionHead(ct.ch));
    return output.ToString();
}
protected string compileExpr(Expr e) {
    StringBuilder output = new StringBuilder();
    output.Append(compileTerm(e.t));
    try { output.Append(compileExprTail(e.et)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileExprTail(ExprTail et) {
    if (et == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(et.op.value);
    output.Append(compileTerm(et.t));
    try { output.Append(compileExprTail(et.et)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileTerm(Term t) {
    StringBuilder output = new StringBuilder();
    output.Append(compileFactor(t.f));
    try { output.Append(compileTermTail(t.tt)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileTermTail(TermTail tt) {
    if (tt == null) throw new ArgumentNullException();
    StringBuilder output = new StringBuilder();
    output.Append(tt.op.value);
    output.Append(compileFactor(tt.f));
    try { output.Append(compileTermTail(tt.tt)); }
    catch (ArgumentNullException) { }
    return output.ToString();
}
protected string compileFactor(Factor f) {
    StringBuilder output = new StringBuilder();
    if (f.e != null) {
        output.Append(f.openPar.value);
        output.Append(compileExpr(f.e));
        output.Append(f.closePar.value);
```

```
        } else if (f.n != null) {
            output.Append(compileNum(f.n));
        } else {
            output.Append(f.stringValue);
            output.Append(f.boolValue.ToString().ToLower());
        }
        return output.ToString();
    }
    protected string compileNum(Num n) {
        StringBuilder output = new StringBuilder();
        try { output.Append(string.Format("Math.pow({0},{1})", compileBase(n.b), compileNumTail(n
        catch (ArgumentNullException) { output.Append(compileBase(n.b)); }
        return output.ToString();
    }
    protected string compileNumTail(NumTail nt) {
        StringBuilder output = new StringBuilder();
        if (nt == null) throw new ArgumentNullException();
        output.Append(compileExponent(nt.e));
        return output.ToString();
    }
    protected string compileBase(Base b) {
        StringBuilder output = new StringBuilder();
        if (b.type == "ID") {
            output.Append(b.id);
        } else if (b.type == "INTEGER") {
            output.Append(b.intValue);
        } else {
            output.Append(b.doubleValue);
        }
        return output.ToString();
    }
    protected string compileExponent(Exponent e) {
        StringBuilder output = new StringBuilder();
        if (e.minus != null) output.Append(e.minus.value);
        output.Append(compileExponentTail(e.et));
        return output.ToString();
    }
    protected string compileExponentTail(ExponentTail et) {
        StringBuilder output = new StringBuilder();
        if (et.type == "ID") output.Append(et.id);
        if (et.type == "INTEGER") output.Append(et.intValue);
        return output.ToString();
    }
}
```

the code produced after compilation phase:

```
<body onload="init();">
<form></form>
<script>
var init = function() {
GUIRenderer({"hidden" : false , "id" : "price" , "label" : "Vehicle price:" ,"type" : "money", "ta
GUIRenderer({"hidden" : false , "id" : "wantLoan" , "label" : "Do you request a loan?" ,"type" :
GUIRenderer({"hidden" : true , "id" : "interest" , "label" : "Interest rate:" , "cond" : "wantLoa
GUIRenderer({"hidden" : true , "id" : "months" , "label" : "Months" , "cond" : "wantLoan" ,"type"
GUIRenderer({"hidden" : true , "id" : "payment" , "label" : "Monthly payment:" , "cond" : "wantLo
}
</script>
</body>
</html>
```

Esercizio 5

⟨*TypeTail*⟩	::=	'(' ⟨*Expr*⟩ ')'
	|	'[' ⟨*SelectList*⟩ ']' Id
	|	⟨*empty*⟩
⟨*SelectList*⟩	::=	⟨*SelectListHead*⟩ ⟨*SelectListTail*⟩
	|	⟨*empty*⟩
⟨*SelectListHead*⟩	::=	⟨*SelectType*⟩ ':' ⟨*SelectType*⟩
⟨*SelectListTail*⟩	::=	',' ⟨*SelectListHead*⟩ ⟨*SelectListTail*⟩
	|	⟨*empty*⟩
⟨*SelectType*⟩	::=	Id
	|	Real
	|	Integer

```
public class ExtendedParser : Parser {
    public ExtendedParser() : base() { }
    protected override TypeTail TypeTail() {
        if (lookahead.type == (int)type.OPEN_PAR) {
            Match(type.OPEN_PAR);
            Expr e = Expr();
            Match(type.CLOSE_PAR);
            return new TypeTail(e);
        } else if (lookahead.type == (int)type.OPEN_SQUARE) {
            Match(type.OPEN_SQUARE);
            SelectList sl = SelectList();
            Match(type.CLOSE_SQUARE);
            string id = lookahead.value;
            Match(type.ID);
            return new TypeTail(sl, id);
        } else return null;
    }
    protected SelectList SelectList() {
        if (lookahead.type == (int)type.ID || lookahead.type == (int)type.REAL ||
            lookahead.type == (int)type.INTEGER) {
            return new SelectList(SelectListHead(), SelectListTail());
        } else return null;
    }
    protected SelectListHead SelectListHead() {
        SelectType st = SelectType();
        Match(type.COLON);
        return new SelectListHead(st, SelectType());
    }
    protected SelectListTail SelectListTail() {
        if (lookahead.type == (int)type.COMMA) {
            Match(type.COMMA);
            return new SelectListTail(SelectListHead(), SelectListTail());
        } else return null;
    }
    protected SelectType SelectType() {
        if (lookahead.type == (int)type.ID) {
            string id = lookahead.value;
            Match(type.ID);
            return new SelectType(id, "ID");
        } else if (lookahead.type == (int)type.REAL) {
            double doubleValue = Convert.ToDouble(lookahead.value);
            Match(type.REAL);
            return new SelectType(doubleValue);
        } else {
            int intValue = Convert.ToInt32(lookahead.value);
            Match(type.INTEGER);
            return new SelectType(intValue);
        }
    }
}

public class ExtendedCompiler : Compiler {
    public ExtendedCompiler() : base() { }
    protected override string compileTypeTail(TypeTail tt) {
        if (tt == null) throw new ArgumentNullException();
        StringBuilder output = new StringBuilder();
        if (tt.e != null) {
            output.Append(string.Format(", \"tag\" : \"input\" , \"readOnly\" : true , \"expr\" :
                compileExpr(tt.e)));
        } else {
            try { output.Append(string.Format(", \"tag\" : \"select\" , \"list\" : {{{0}}} , \"ta
                compileSelectList(tt.sl), tt.id)); }
            catch (ArgumentNullException) { output.Append(string.Format(", \"tag\" : \"select\" ,
                tt.id)); }
        }
        return output.ToString();
    }
    protected string compileSelectList(SelectList sl) {
        if (sl == null) throw new ArgumentNullException();
        StringBuilder output = new StringBuilder();
        output.Append(compileSelectListHead(sl.slh));
        try { output.Append(compileSelectListTail(sl.slt)); }
        catch (ArgumentNullException) { }
        return output.ToString();
    }
    protected string compileSelectListHead(SelectListHead slh) {
```

```
        StringBuilder output = new StringBuilder();
        output.Append(compileSelectType(slh.st1));
        output.Append(slh.colon.value);
        output.Append(compileSelectType(slh.st2));
        return output.ToString();
    }
    protected string compileSelectListTail(SelectListTail slt) {
        if (slt == null) throw new ArgumentNullException();
        StringBuilder output = new StringBuilder();
        output.Append(slt.comma.value);
        output.Append(compileSelectListHead(slt.slh));
        try { output.Append(compileSelectListTail(slt.slt)); }
        catch (ArgumentNullException) { }
        return output.ToString();
    }
    protected string compileSelectType(SelectType st) {
        StringBuilder output = new StringBuilder();
        if (st.type == "ID") {
            output.Append(st.id);
        } else if (st.type == "INTEGER") {
            output.Append(st.intValue);
        } else {
            output.Append(st.doubleValue);
        }
        return output.ToString();
    }
}
```