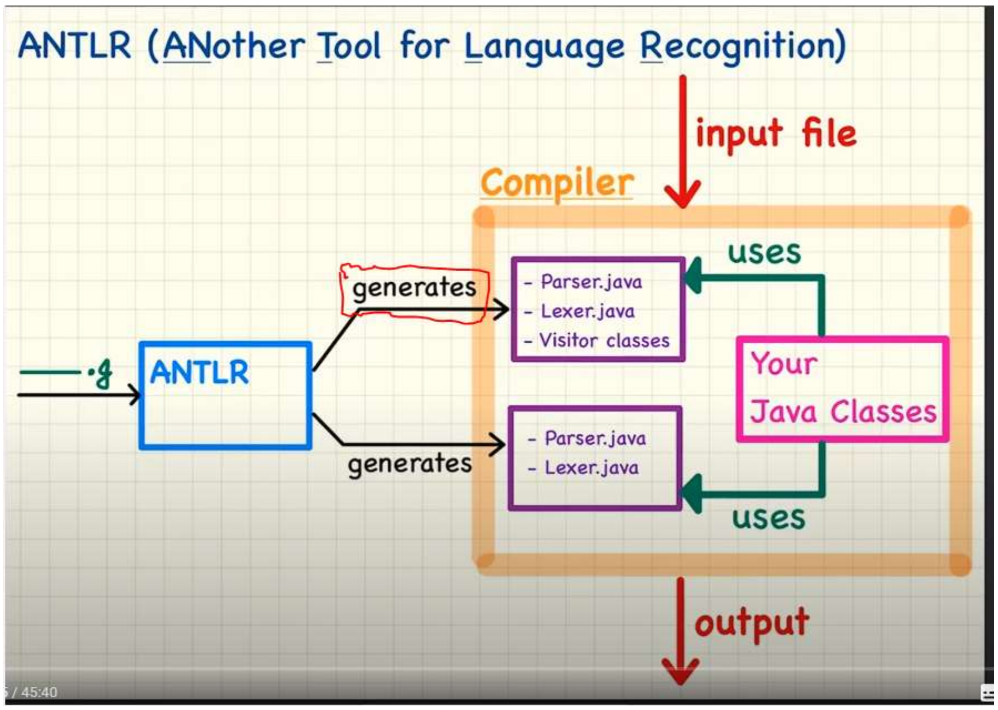
Antlr Tutorial

utorak, 19. januar 2021. 13:49

This tutorial will use an example to explain how to create antlr grammar file, how to use antlr tool to generate Parser and Lexer based on your grammar file and how to create Evaluator to complete compilation process.

High Level View

Antlr workflow:



This example will use upper method (with Visitor Classes).

Your Java Classes on the right of screenshot will contain you Visitor Classes and , your Model of DSL and your Evaluator.

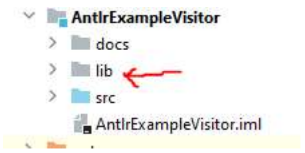
The example will use IntelIJ Idea IDE.

Installation of Antlr4 and Grun

Go to https://www.antlr.org/ and follow installation instructions for you platform (OSx, Win, Linux):

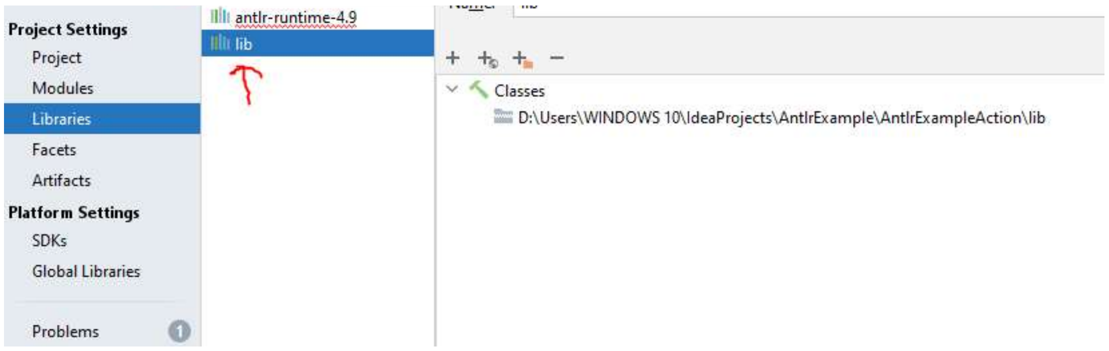
In IDE create new project e.g AntlrExampleVisitor (it will use Visitor design pattern).

In you project create new folder named **lib.**



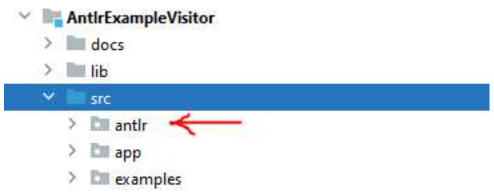
Download antlr4 Jar here https://www.antlr.org/download/antlr-4.9.1-complete.jar.
And copy it in **lib** folder.

Go to File -> Project Structure -> Libraries and add lib folder to as library:

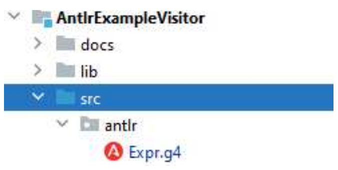


D:\Users\WINDOWS 10\IdeaProjects\AntlrExample\AntlrExampleAction\lib

In src folder create **antlr** package:



In **antlr** package create **Expr.g4** file which represents you grammar file:



Grammar file for this example:

```
grammar Expr;

/* the file name and grammar name must match */

@header {
    package antlr;
}

// Start Variable
prog: (decl | expr)+ EOF        # Program
    ;

decl: ID ':' INT_TYPE '=' NUM   # Declaration
    ;

/* Antlr resolves ambiguity in favor of alternative given first. */
expr: expr '*' expr             # Multiplication
    | expr '+' expr             # Addition
    | expr '-' expr             # Subtraction
    | ID                        # Variable
    | NUM                       # Number
    ;

/* Tokens */
ID : [a-z][a-zA-Z0-9_]*;  // identifiers
NUM: '0' | '-'?[1-9][0-9]*;
INT_TYPE: 'INT';
COMMENT: '--' ~[\r\n]* -> skip;
WS: [ \t\n\r]+ -> skip;
```

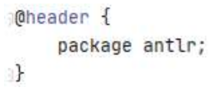
Grammar file explanation:

Make sure your grammar file starts with:

grammar name_of_grammar_file;

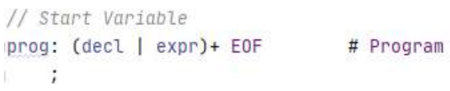
All Parser and Lexer Classes generated by Antlr tool will have name_of_grammar_file as prefix in their name.

The following block means that all files generated by antlr tool based on this grammar file will be stored in antlr package:



```
@header {
    package antlr;
}
```

First production rule in your grammar file will be Start variable:

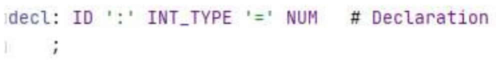


```
// Start Variable
prog: (decl | expr)+ EOF        # Program
    ;
```

Note: you can use Java style comments in grammar file.
Note: variables/production rules are written in lower case and tokens in upper case letters

This production rule says that it can contain one or more **decl** or **expr**(will be defined later) and must end with **EOF.** This means that at the top level sentences will consist of mixture of declarations and expressions and it will end with EOF.

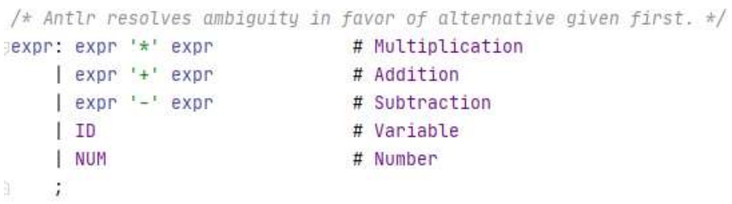
Second production rule is **decl** which represents declaration of a variable:



```
decl: ID ':' INT_TYPE '=' NUM   # Declaration
    ;
```

This means that declaration must start with ID followed by ':' then INT_TYPE .....
Where ID, INT_TYPE and NUM are tokens.

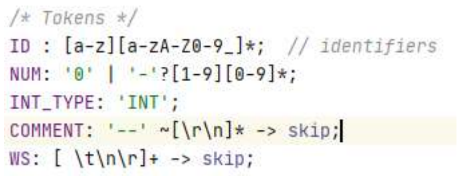
Last prod rule is **expr** which represents various expressions:



```
/* Antlr resolves ambiguity in favor of alternative given first. */
expr: expr '*' expr             # Multiplication
    | expr '+' expr             # Addition
    | expr '-' expr             # Subtraction
    | ID                        # Variable
    | NUM                       # Number
    ;
```

This means that expression can contain adding, subtracting, multiplying operations or just ID or NUM.

Next are Tokens defined using regular exp like syntax:



```
/* Tokens */
ID : [a-z][a-zA-Z0-9_]*;  // identifiers
NUM: '0' | '-'?[1-9][0-9]*;
INT_TYPE: 'INT';
COMMENT: '--' ~[\r\n]* -> skip;
WS: [ \t\n\r]+ -> skip;
```

This means that ID must start with lower letter alpha and can also have 0 or more aplhanumerics or lower dash symbol etc...

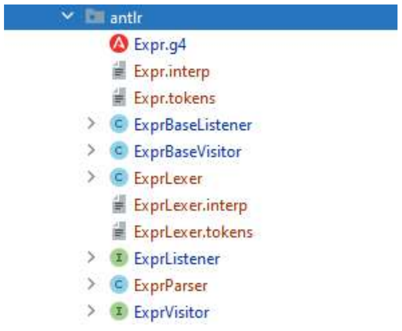
-> skip means that all expressions before -> skip will not be evaluated as Tokens.

Generate Lexer and Parser from your grammar file

In terminal/cmd go to your projects -> src -> antlr and run:

antlr4 –visitor Expr.g4

This command will generate some files :



Every time you change your grammar file, you must use command above (**antlr4 –visitor Expr.g4**) to regenerate these files.

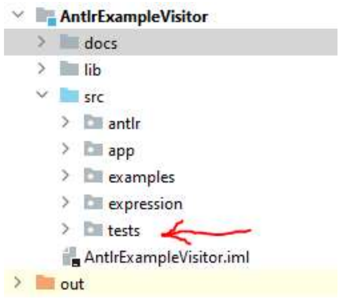
To be able to use **grun** tool you must compile java classes in antlr package. Got to src/antlr and run:

javac *.java

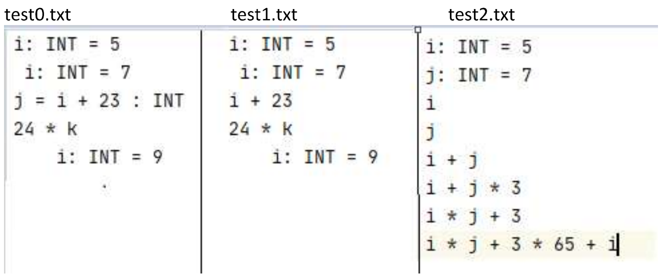
More Class files will appear in src/antlr depending on your grammar complexity

Test Source Files and Grun Tool

Create tests package in your src folder (in IDE):



In this package create 3 test files: test0.txt, test1.txt and test2.txt.



test0.txt
```
i: INT = 5
 i: INT = 7
j = i + 23 : INT
24 * k
    i: INT = 9
```

test1.txt
```
i: INT = 5
 i: INT = 7
i + 23
24 * k
    i: INT = 9
```

test2.txt
```
i: INT = 5
j: INT = 7
i
j
i + j
i + j * 3
i * j + 3
i * j + 3 * 65 + i
```

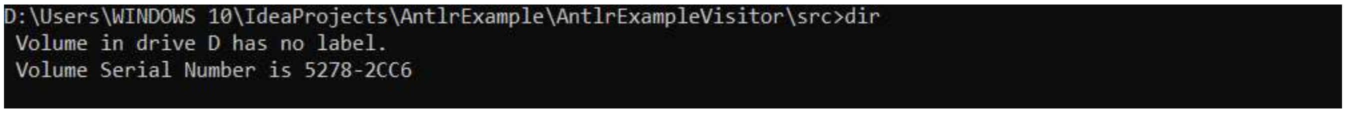
Test0.txt has syntax error (j = I + 23 : INT) because there was no assignment operator defined in the grammar file. Test0.txt also has semantic error (variable i declared twice).

Test1.txt has one semantic error (variable i declared twice) and no syntax errors (all expressions are written according to grammar file).

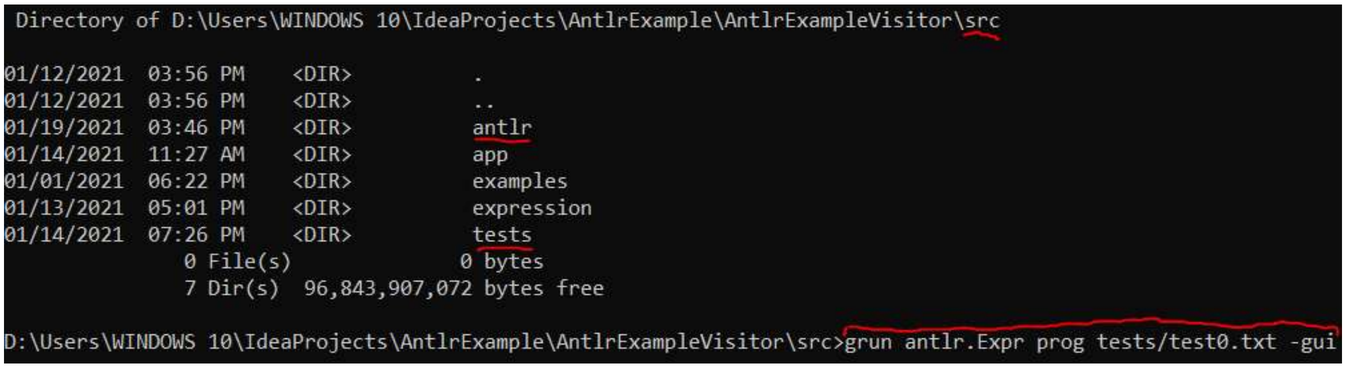
Test2.txt does not have syntax or semantic errors.

To test these files we will use **Grun tool:**

In your cmd/terminal go to one directory above antlr directory(in our case it is src directory), and run **grun antlr.Expr prog tests/test0.txt -gui &:**



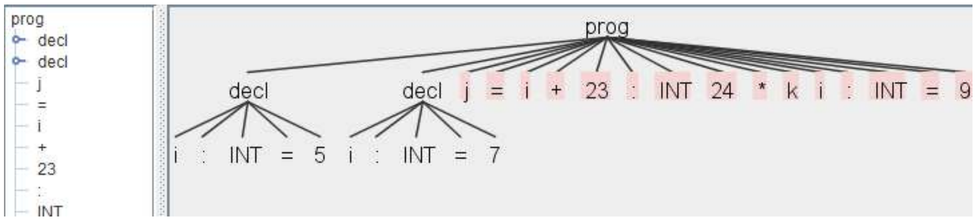
```
D:\Users\WINDOWS 10\IdeaProjects\AntlrExample\AntlrExampleVisitor\src>dir
 Volume in drive D has no label.
 Volume Serial Number is 5278-2CC6
```



```
 Directory of D:\Users\WINDOWS 10\IdeaProjects\AntlrExample\AntlrExampleVisitor\src

01/12/2021  03:56 PM    <DIR>          .
01/12/2021  03:56 PM    <DIR>          ..
01/19/2021  03:46 PM    <DIR>          antlr
01/14/2021  11:27 AM    <DIR>          app
01/01/2021  06:22 PM    <DIR>          examples
01/13/2021  05:01 PM    <DIR>          expression
01/14/2021  07:26 PM    <DIR>          tests
               0 File(s)              0 bytes
               7 Dir(s)  96,843,907,072 bytes free

D:\Users\WINDOWS 10\IdeaProjects\AntlrExample\AntlrExampleVisitor\src>grun antlr.Expr prog tests/test0.txt -gui
```

This is expected **result**:

line 3:2 no viable alternative at input 'j='



This AST shows that only first two lines were parsed and lines starting with line 3 are not parsed since there is syntax error in line 3.

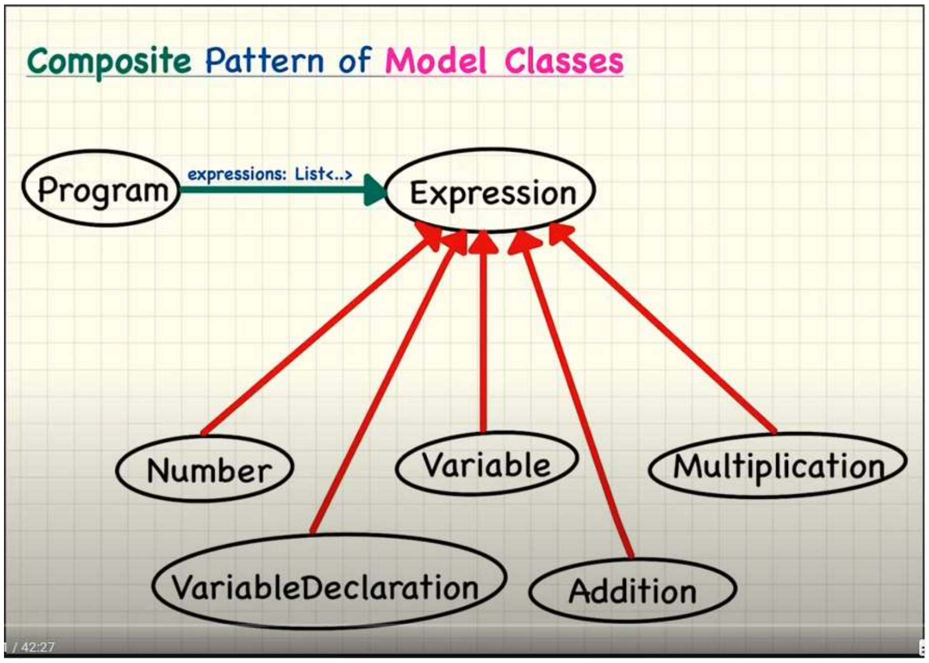
Similarly you can test files test1.txt and test2.txt.

The Grun tool is useful when creating Compiler and because you can compare expected AST(Abstract Parse Tree) with actual AST and correct your grammar file.

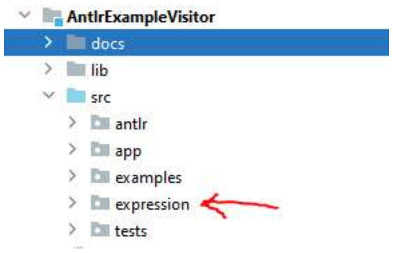
Antlr Tree Nodes to Custom Expression Objects

In order to have easier manipulation of AST, easier debugging and better implementation of Evaluator Classes we will create **Model Classes:**

This model uses **Composite design pattern** to allow expressions to be recursively visited. For example expression 25 * k + 4 has two operands 25 * 4 and 4 but 25 * 4 needs to be recursively visited so it can be evaluated and it's value returned to expression 25 * k + 4.



In IDE create package expression:



In expression package create Expression class:

```java
package expression;

public abstract class Expression {
}
```

In same package create Program class:

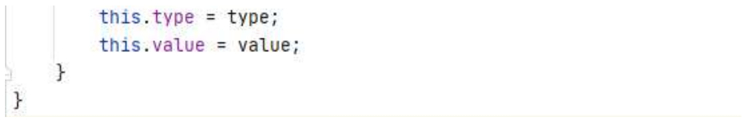
```java
package expression;

import java.util.ArrayList;
import java.util.List;

public class Program {
    public List<Expression> expressions;

    public Program() {
        this.expressions = new ArrayList<>();
    }

    public void addExpression(Expression e){
        expressions.add(e);
    }
}
```

In same package create Number class:

```java
package expression;

public class Number extends Expression{
    int num;

    public Number(int num) {
        this.num = num;
    }

    @Override
    public String toString() {
        return new ~~Integer~~(num).toString();
    }
}
```

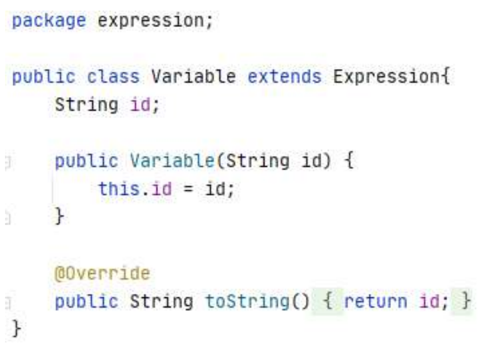
In same package create VariableDeclaration class:

```java
package expression;

public class VariableDeclaration extends Expression {
    public String id;
    public String type;
    public int value;

    public VariableDeclaration(String id, String type, int value) {
        this.id = id;
```



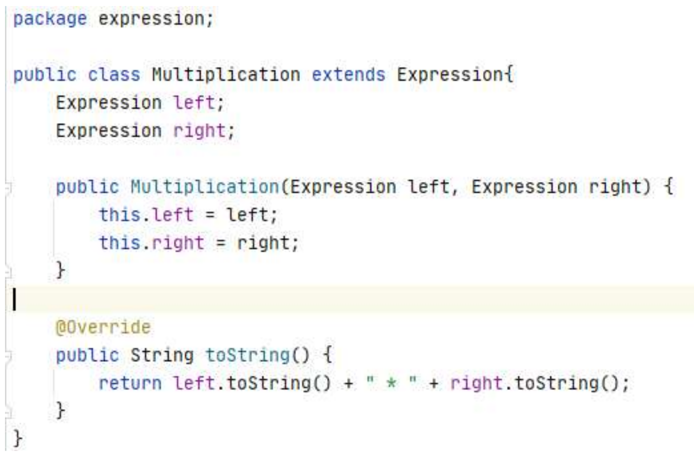
```java
        this.type = type;
        this.value = value;
    }
}
```

In same package create Variable class:



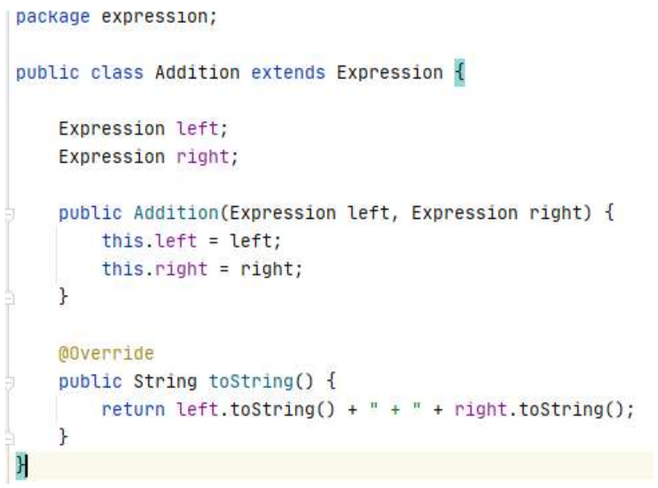
```java
package expression;

public class Variable extends Expression{
    String id;

    public Variable(String id) {
        this.id = id;
    }

    @Override
    public String toString() { return id; }
}
```

In same package create Multiplication class:



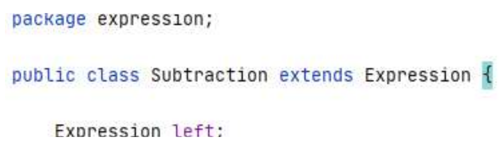
```java
package expression;

public class Multiplication extends Expression{
    Expression left;
    Expression right;

    public Multiplication(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public String toString() {
        return left.toString() + " * " + right.toString();
    }
}
```

In same package create Addition class:

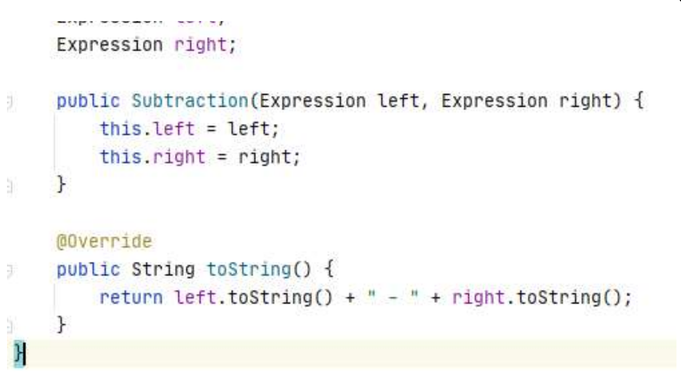


```java
package expression;

public class Addition extends Expression {

    Expression left;
    Expression right;

    public Addition(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public String toString() {
        return left.toString() + " + " + right.toString();
    }
}
```

In same package create Subtraction class:



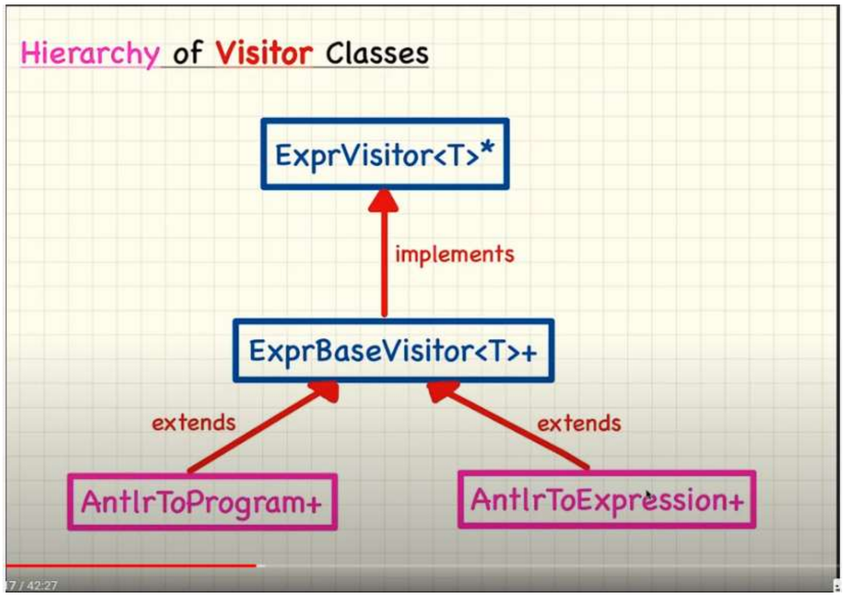
```java
package expression;

public class Subtraction extends Expression {

    Expression left;
```



```
    Expression left;
    Expression right;

    public Subtraction(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public String toString() {
        return left.toString() + " - " + right.toString();
    }
}
```

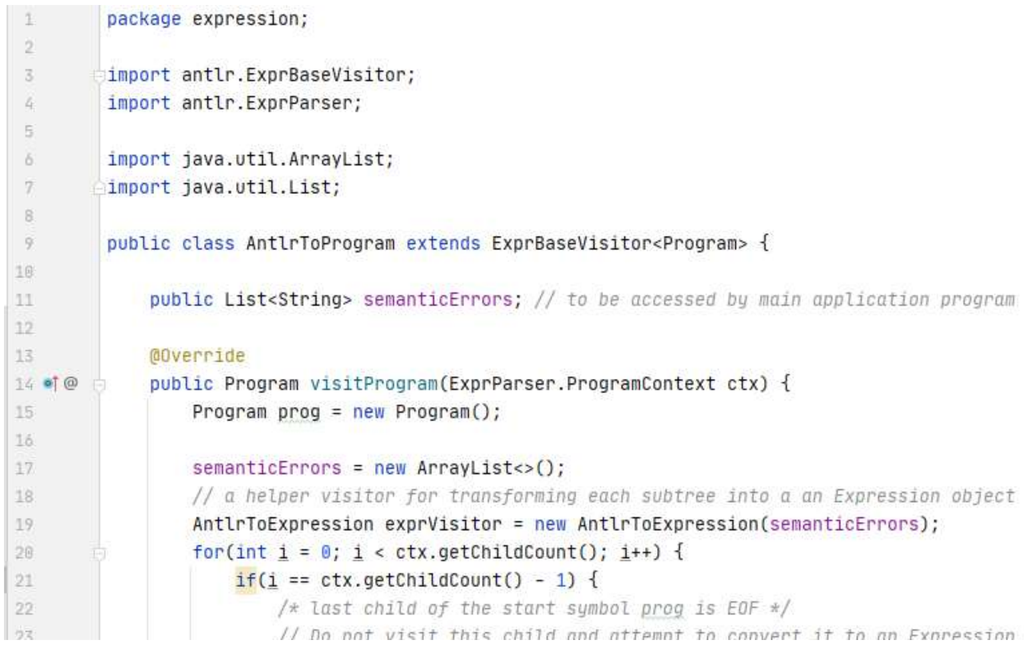
Using Visitor to convert Antlr generated Tree into your model Objects

More about Visitor design pattern here and here.

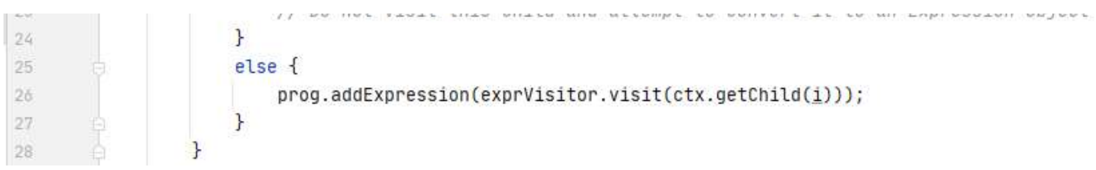


In expression package create two Visitor class:

First create AntlrToProgram Class which will be used to convert Program Tree Node to Program Object of your model.



```
package expression;

import antlr.ExprBaseVisitor;
import antlr.ExprParser;

import java.util.ArrayList;
import java.util.List;

public class AntlrToProgram extends ExprBaseVisitor<Program> {

    public List<String> semanticErrors; // to be accessed by main application program

    @Override
    public Program visitProgram(ExprParser.ProgramContext ctx) {
        Program prog = new Program();

        semanticErrors = new ArrayList<>();
        // a helper visitor for transforming each subtree into a an Expression object
        AntlrToExpression exprVisitor = new AntlrToExpression(semanticErrors);
        for(int i = 0; i < ctx.getChildCount(); i++) {
            if(i == ctx.getChildCount() - 1) {
                /* last child of the start symbol prog is EOF */
                // Do not visit this child and attempt to convert it to an Expression object
```

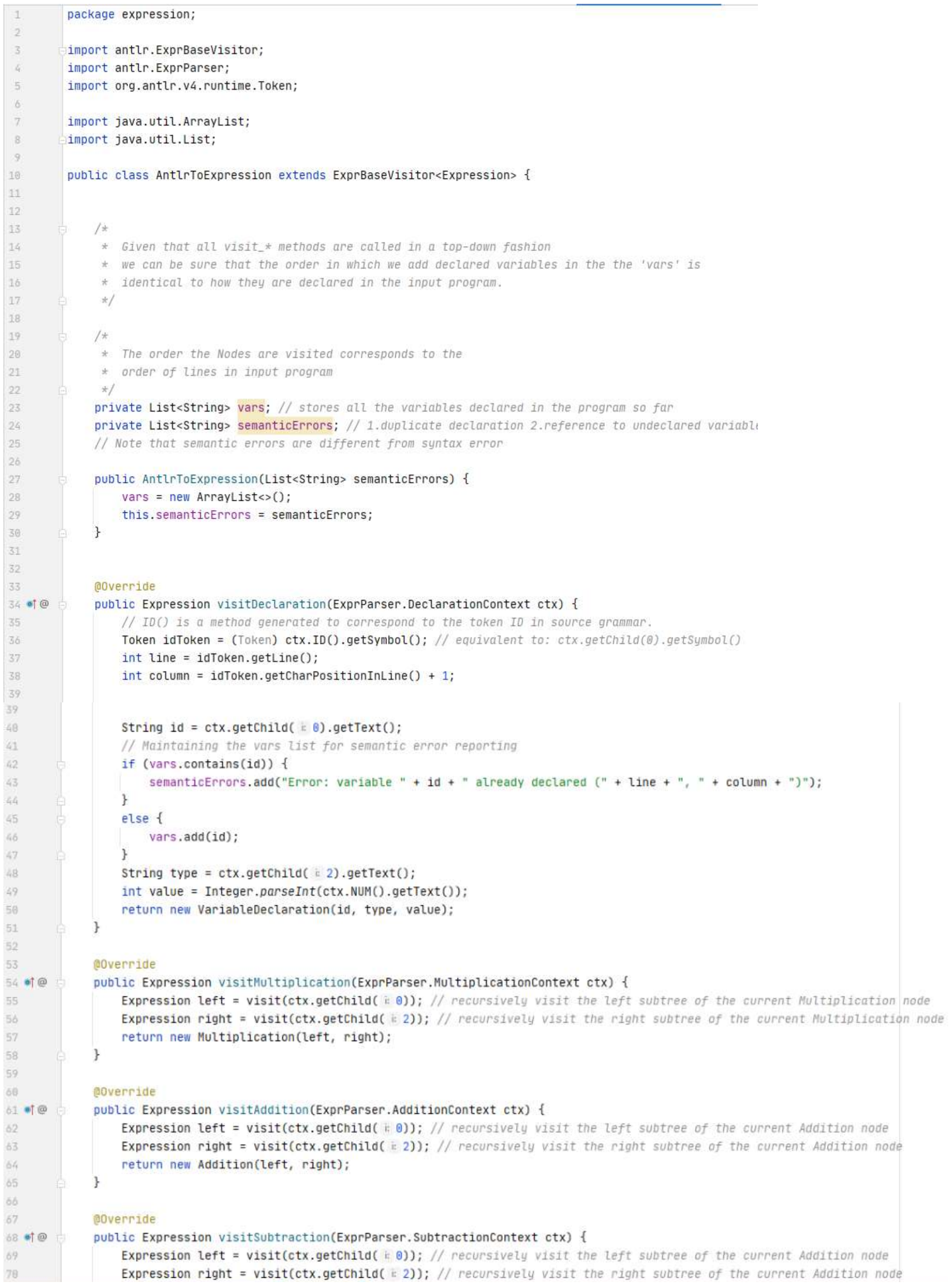


```
            // Do not visit this child and attempt to convert it to an Expression object
        }
        else {
            prog.addExpression(exprVisitor.visit(ctx.getChild(i)));
        }
    }
```

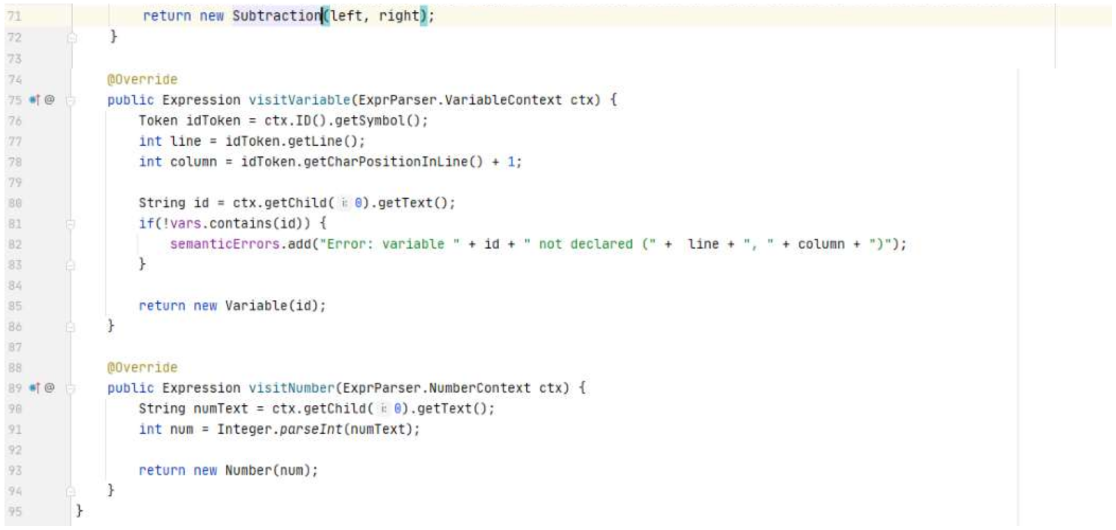
VisitProgram method takes ExprParser.ProgramContext as a parameter since ExprParser.ProgramContext is type of Program Node in AST.

ExprParser.ProgramContext needs to be converted to Program object.
In for loop all child nodes from program node in AST will be visited(converted) and added to Program object's list of expressions.

Create AntlrToExpression class:



```
package expression;

import antlr.ExprBaseVisitor;
import antlr.ExprParser;
import org.antlr.v4.runtime.Token;

import java.util.ArrayList;
import java.util.List;

public class AntlrToExpression extends ExprBaseVisitor<Expression> {


    /*
     *  Given that all visit_* methods are called in a top-down fashion
     *  we can be sure that the order in which we add declared variables in the the 'vars' is
     *  identical to how they are declared in the input program.
     */

    /*
     *  The order the Nodes are visited corresponds to the
     *  order of lines in input program
     */
    private List<String> vars; // stores all the variables declared in the program so far
    private List<String> semanticErrors; // 1.duplicate declaration 2.reference to undeclared variabl
    // Note that semantic errors are different from syntax error

    public AntlrToExpression(List<String> semanticErrors) {
        vars = new ArrayList<>();
        this.semanticErrors = semanticErrors;
    }


    @Override
    public Expression visitDeclaration(ExprParser.DeclarationContext ctx) {
        // ID() is a method generated to correspond to the token ID in source grammar.
        Token idToken = (Token) ctx.ID().getSymbol(); // equivalent to: ctx.getChild(0).getSymbol()
        int line = idToken.getLine();
        int column = idToken.getCharPositionInLine() + 1;


        String id = ctx.getChild( i: 0).getText();
        // Maintaining the vars list for semantic error reporting
        if (vars.contains(id)) {
            semanticErrors.add("Error: variable " + id + " already declared (" + line + ", " + column + ")");
        }
        else {
            vars.add(id);
        }
        String type = ctx.getChild( i: 2).getText();
        int value = Integer.parseInt(ctx.NUM().getText());
        return new VariableDeclaration(id, type, value);
    }

    @Override
    public Expression visitMultiplication(ExprParser.MultiplicationContext ctx) {
        Expression left = visit(ctx.getChild( i: 0)); // recursively visit the left subtree of the current Multiplication node
        Expression right = visit(ctx.getChild( i: 2)); // recursively visit the right subtree of the current Multiplication node
        return new Multiplication(left, right);
    }

    @Override
    public Expression visitAddition(ExprParser.AdditionContext ctx) {
        Expression left = visit(ctx.getChild( i: 0)); // recursively visit the left subtree of the current Addition node
        Expression right = visit(ctx.getChild( i: 2)); // recursively visit the right subtree of the current Addition node
        return new Addition(left, right);
    }

    @Override
    public Expression visitSubtraction(ExprParser.SubtractionContext ctx) {
        Expression left = visit(ctx.getChild( i: 0)); // recursively visit the left subtree of the current Addition node
        Expression right = visit(ctx.getChild( i: 2)); // recursively visit the right subtree of the current Addition node
```

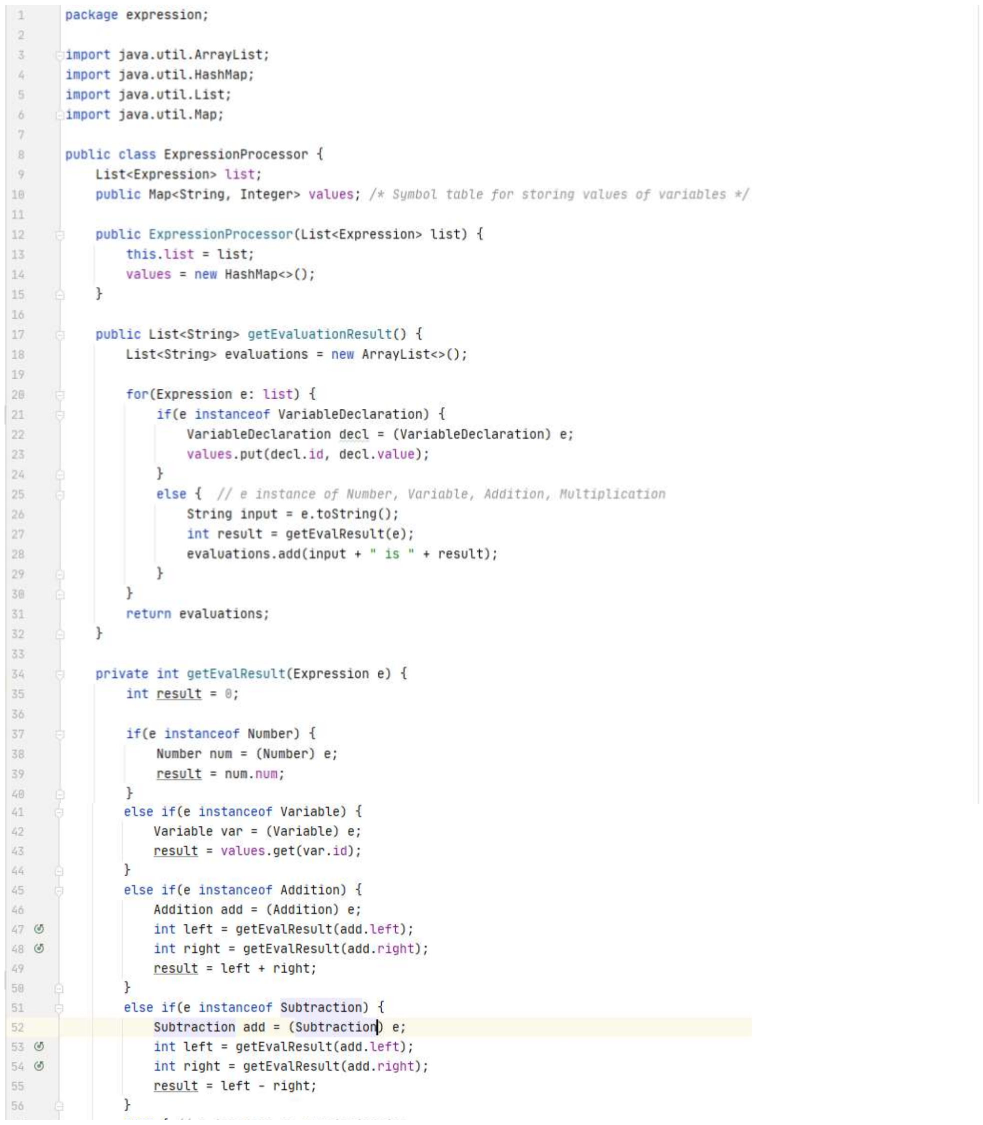


```java
        return new Subtraction(left, right);
    }

    @Override
    public Expression visitVariable(ExprParser.VariableContext ctx) {
        Token idToken = ctx.ID().getSymbol();
        int line = idToken.getLine();
        int column = idToken.getCharPositionInLine() + 1;

        String id = ctx.getChild( i: 0).getText();
        if(!vars.contains(id)) {
            semanticErrors.add("Error: variable " + id + " not declared (" +  line + ", " + column + ")");
        }

        return new Variable(id);
    }

    @Override
    public Expression visitNumber(ExprParser.NumberContext ctx) {
        String numText = ctx.getChild( i: 0).getText();
        int num = Integer.parseInt(numText);

        return new Number(num);
    }
}
```

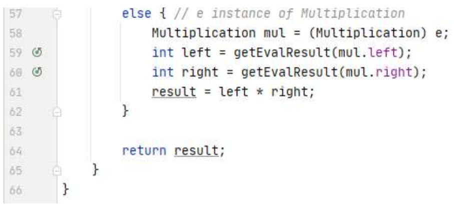
This Visitor class is used to visit all nodes except Program node. Visit methods (visitVariable, visitAddition ...) will be called in AntlrToProgram in line 26. Each of these methods takes ExpressionContext as parameter, takes needed information from it and stores it in Model Object (Variable, Addition ....) and returns that Object.

Evaluation process

Create ExpressionProcessor in expression package



```java
package expression;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ExpressionProcessor {
    List<Expression> list;
    public Map<String, Integer> values; /* Symbol table for storing values of variables */

    public ExpressionProcessor(List<Expression> list) {
        this.list = list;
        values = new HashMap<>();
    }

    public List<String> getEvaluationResult() {
        List<String> evaluations = new ArrayList<>();

        for(Expression e: list) {
            if(e instanceof VariableDeclaration) {
                VariableDeclaration decl = (VariableDeclaration) e;
                values.put(decl.id, decl.value);
            }
            else {  // e instance of Number, Variable, Addition, Multiplication
                String input = e.toString();
                int result = getEvalResult(e);
                evaluations.add(input + " is " + result);
            }
        }
        return evaluations;
    }

    private int getEvalResult(Expression e) {
        int result = 0;

        if(e instanceof Number) {
            Number num = (Number) e;
            result = num.num;
        }
        else if(e instanceof Variable) {
            Variable var = (Variable) e;
            result = values.get(var.id);
        }
        else if(e instanceof Addition) {
            Addition add = (Addition) e;
            int left = getEvalResult(add.left);
            int right = getEvalResult(add.right);
            result = left + right;
        }
        else if(e instanceof Subtraction) {
            Subtraction add = (Subtraction) e;
            int left = getEvalResult(add.left);
            int right = getEvalResult(add.right);
            result = left - right;
        }
```



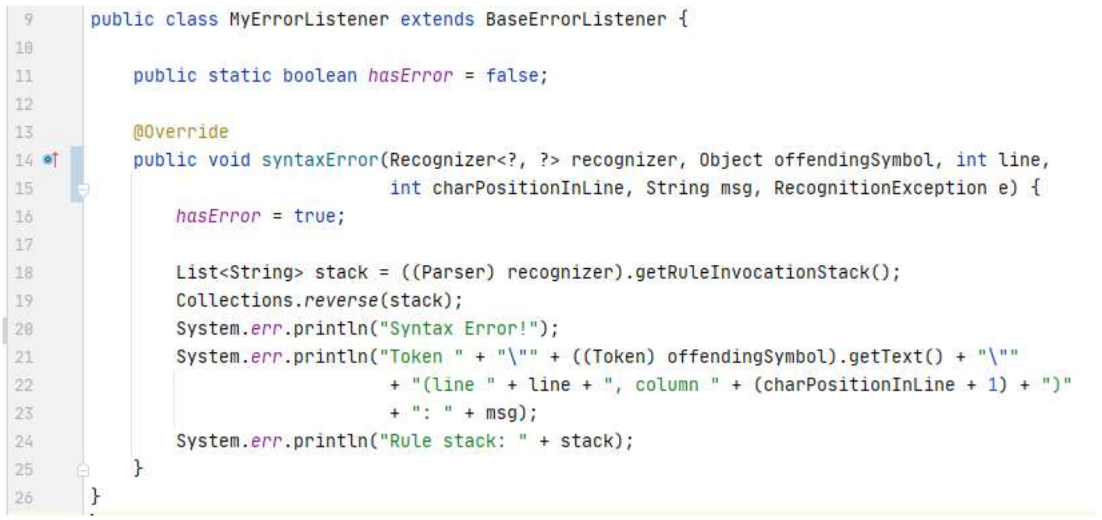
```java
        else { // e instance of Multiplication
            Multiplication mul = (Multiplication) e;
            int left = getEvalResult(mul.left);
            int right = getEvalResult(mul.right);
            result = left * right;
        }

        return result;
    }
}
```

This Class has public method getEvaluationResult that goes through list of all expressions and calls helper method getEvalResult to evaluate every individual expression. At the end all evaluated expressions are added into evaluations list which is returned by getEvaluationResult.

Custom Error Listener

In order to separate syntax errors from semantic errors, we must implement our custom Error Listener. This error listener will process syntax errors.

In expressions package create MyErrorListener Class.

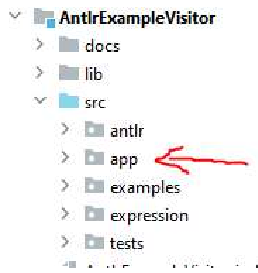


```java
public class MyErrorListener extends BaseErrorListener {

    public static boolean hasError = false;

    @Override
    public void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol, int line,
                            int charPositionInLine, String msg, RecognitionException e) {
        hasError = true;

        List<String> stack = ((Parser) recognizer).getRuleInvocationStack();
        Collections.reverse(stack);
        System.err.println("Syntax Error!");
        System.err.println("Token " + "\"" + ((Token) offendingSymbol).getText() + "\""
                            + "(line " + line + ", column " + (charPositionInLine + 1) + ")"
                            + ": " + msg);
        System.err.println("Rule stack: " + stack);
    }
}
```

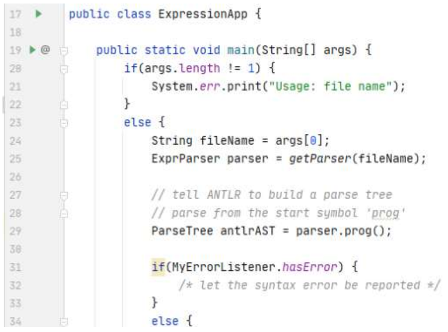
This class has one overriden method – syntaxError which will be called internally by parser if there are syntax errors.

Putting all together

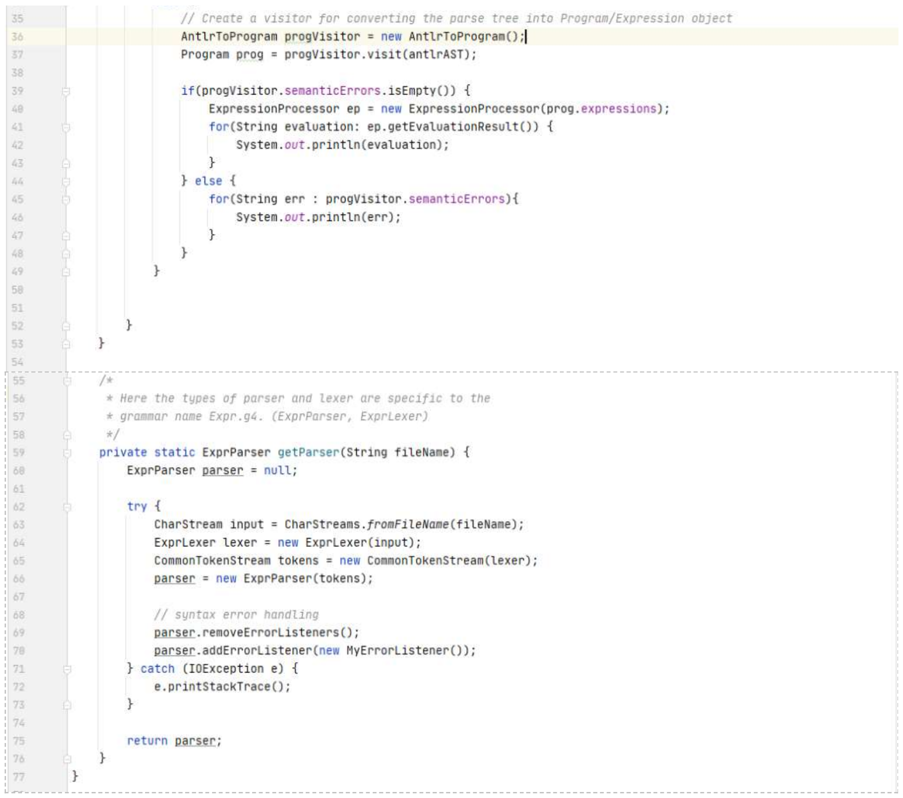
In src create **app** package:



In app package create ExpressionApp class:



```java
public class ExpressionApp {

    public static void main(String[] args) {
        if(args.length != 1) {
            System.err.print("Usage: file name");
        }
        else {
            String fileName = args[0];
            ExprParser parser = getParser(fileName);

            // tell ANTLR to build a parse tree
            // parse from the start symbol 'prog'
            ParseTree antlrAST = parser.prog();

            if(MyErrorListener.hasError) {
                /* let the syntax error be reported */
            }
            else {
```



```
35                  // Create a visitor for converting the parse tree into Program/Expression object
36                  AntlrToProgram progVisitor = new AntlrToProgram();
37                  Program prog = progVisitor.visit(antlrAST);
38
39                  if(progVisitor.semanticErrors.isEmpty()) {
40                      ExpressionProcessor ep = new ExpressionProcessor(prog.expressions);
41                      for(String evaluation: ep.getEvaluationResult()) {
42                          System.out.println(evaluation);
43                      }
44                  } else {
45                      for(String err : progVisitor.semanticErrors){
46                          System.out.println(err);
47                      }
48                  }
49              }
50
51
52          }
53      }
54
55      /*
56       * Here the types of parser and lexer are specific to the
57       * grammar name Expr.g4. (ExprParser, ExprLexer)
58       */
59      private static ExprParser getParser(String fileName) {
60          ExprParser parser = null;
61
62          try {
63              CharStream input = CharStreams.fromFileName(fileName);
64              ExprLexer lexer = new ExprLexer(input);
65              CommonTokenStream tokens = new CommonTokenStream(lexer);
66              parser = new ExprParser(tokens);
67
68              // syntax error handling
69              parser.removeErrorListeners();
70              parser.addErrorListener(new MyErrorListener());
71          } catch (IOException e) {
72              e.printStackTrace();
73          }
74
75          return parser;
76      }
77  }
```

This class is program entry point. Starting in line 59 there is **getParser** method which is helper method to get Parser in main method. It takes filename(e.g test0.txt) creates lexer from it, uses that lexer to create tokens and then uses tokens to create parser which is returned by this method.

In line 69 default error listener is removed and in line 70 custom error listener is added to parser.

Parser object is created in line 25 and then in line 29 it is used to create ParseTree. In line 36 AntlrToProgram(Program Visitor) is created and used in line 37 to visit prog node of ParseTree.

In line 40 ExpressionProcessor object is created and it is used in line 41 in for loop to get eval result for every expression(which is printed in line 42).