Antlr Report

Expr.g4

```
grammar Expr;

// parser rules
prog : (expr ';' NEWLINE?)*;

expr : expr ('*'|'/') expr  # infixExpr
     | expr ('+'|'-') expr  # infixExpr
     | '-'?num              # numberExpr
     | '(' expr ')'         # parensExpr
     | VAR '=' '-'?num      # assingExpr
     | VAR                  # variableExpr
     ;

num  : INT
     | REAL
     ;

// lexer rules
NEWLINE: [\r\n]+ ;
INT: [0-9]+ ;          // should handle negatives
REAL: [0-9]+'.'[0-9]* ; // should handle signs(+/-)
WS: [ \t\r\n]+ -> skip ;
VAR: [a-zA-Z_]+ ;
```

numberExpr 앞에 '-'?를 붙여 음수인 경우도 처리할 수 있도록 하였다.

assign과 variable Expr도 추가하여 조건을 충족했다.

lexer rule에 VAR이라는 키워드를 추가하여 변수 스펙인 알파벳과 _ (언더바)를 포함하도록 하였다.

AstCall.java

```java
public class AstCall {

    public void Call(AstNodes astNodes, int tabs) {

        // astNodes instance : Variable Number InfixNode VariableDeclaration

        String indent = "";

        //
        for (int i = 0; i < tabs; i++) {
            indent += "\t";
        }

        if (astNodes instanceof Variable) {

            Variable var = (Variable) astNodes;

            String varName = var.variable;

            System.out.println(indent + varName);

        } else if (astNodes instanceof Number) {

            Number number = (Number) astNodes;

            System.out.println(indent + number.num);

        } else if (astNodes instanceof InfixNode) {
            InfixNode infixNode = (InfixNode) astNodes;

            AstNodes left = infixNode.left;
            AstNodes right = infixNode.right;

            String operator = infixNode.operator; // ADD SUB MUL DIV

            System.out.println(indent + operator);
            this.Call(left, tabs + 1);
            this.Call(right, tabs + 1);

        } else if (astNodes instanceof VariableDeclaration) {

            VariableDeclaration decl = (VariableDeclaration) astNodes;

            String varName = decl.variable;
            double value = decl.value;

            System.out.println(x:"ASSIGN");
            System.out.println("\t" + varName);
            System.out.println("\t" + value);

        }

    }
}
```

AST를 출력할 때 해당 동작을 먼저 출력하고 그 아래로 탭을 하여 출력을 해야한다.

그렇기에 Call이라는 함수에 tabs라는 변수를 추가하여 몇 계단 아래로 내려왔는지를 담

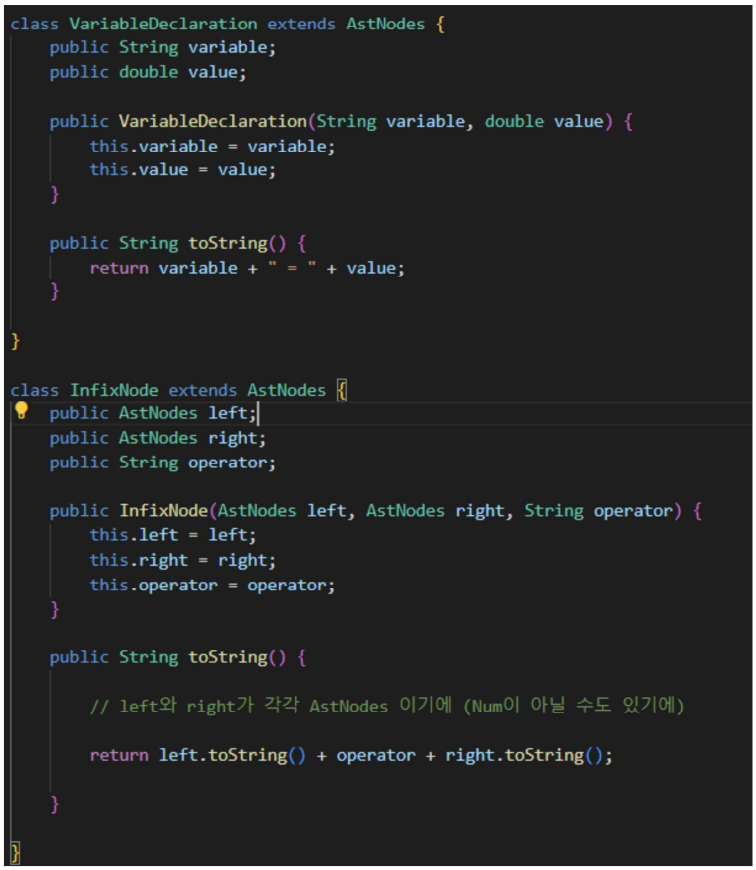
는 변수를 추가하여 인자로 넘겨주었다.

이후 출력할 때는 indent라는 String 변수를 활용하여 tab을 추가하여 출력하였다.

InfixNode를 출력할 때 left와 right가 각각 AstNode이기에 다시 재귀적으로 Call함수를 호출하여 AST를 출력하였다. 이때 left와 right는 한 계단 아래로 내려가는 것이기에 this.Call(left, tab +1)로 호출을 진행하였다.

AstNodes.java

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

public class AstNodes {

}

class ExprNode extends AstNodes {

    public List<AstNodes> expressions;

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

    public void addExpressions(AstNodes astNode) {
        expressions.add(astNode);
    }

}

class Number extends AstNodes {
    public double num;

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

    public String toString() {
        return new Double(num).toString();
    }
}

class Variable extends AstNodes {
    public String variable;

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

    public String toString() {
        return variable;
    }
}
```



```
class VariableDeclaration extends AstNodes {
    public String variable;
    public double value;

    public VariableDeclaration(String variable, double value) {
        this.variable = variable;
        this.value = value;
    }

    public String toString() {
        return variable + " = " + value;
    }

}

class InfixNode extends AstNodes {
    public AstNodes left;
    public AstNodes right;
    public String operator;

    public InfixNode(AstNodes left, AstNodes right, String operator) {
        this.left = left;
        this.right = right;
        this.operator = operator;
    }

    public String toString() {

        // left와 right가 각각 AstNodes 이기에 (Num이 아닐 수도 있기에)

        return left.toString() + operator + right.toString();

    }

}
```

print할 AST들을 정의하였다. InfixtNode, VariableDeclaration, ExprNode, Number, Variable 5가지의 AstNodes를 상속한 class를 만들어 각각 필요한 멤버 변수들을 저장하였다.

ExprNode는 나중에 문법에서 정의한 prog로 변환된 애들을 각각의 Expr들로 바꿔주는데 사용하였다.

BuildAstVisitor.java

```java
public class BuildAstVisitor extends ExprBaseVisitor<AstNodes> {

    @Override
    public AstNodes visitVariableExpr(ExprParser.VariableExprContext ctx) {
        String variable = ctx.getChild(0).getText();

        return new Variable(variable);

    }

    @Override
    public AstNodes visitInfixExpr(ExprParser.InfixExprContext ctx) {

        // expr ('*'|'/') expr or expr ('+'|'-') expr 이므로
        // 0 => left, 2 => right, 1 => operator

        // left, right에는 num이 되는 expr이 아닌 다른 애들이 들어올 수 있기에
        // visit으로 처리해준다.

        AstNodes left = this.visit(ctx.getChild(0));
        AstNodes right = this.visit(ctx.getChild(2));
        String operator = ctx.getChild(1).getText();

        switch (operator) {
            case "+":
                operator = "ADD";
                break;
            case "-":
                operator = "SUB";
                break;
            case "*":
                operator = "MUL";
                break;
            case "/":
                operator = "DIV";
                break;
        }

        return new InfixNode(left, right, operator);

    }
```

visit ~ 메소드들은 visit을 했을 때 return 해줄 AstNode 객체들을 만드는 메소드이다.

Variable은 문법에서 정의했듯이 VAR 하나이기에 0번째 child의 text를 받아오면 그것이 변수 명이기에 새로운 변수를 추가해준다.

infix노드는 left와 right가 각각 AstNodes이기에 text를 가져오지 않고 visit하여 아래의 AstNode들을 담게 한다.

BuildAstVisitor.java

```java
public AstNodes visitNumberExpr(ExprParser.NumberExprContext ctx) {
    // '-' 'num' or 'num'
    //
    double num;
    String minusCheck = ctx.getChild(0).getText();

    // minusCheck가 '-'이면 => '-' num 이므로 음수 처리

    if ("-".equals(minusCheck)) {

        String temp = ctx.getChild(1).getText();
        num = (-1) * Double.parseDouble(temp);

    } else {
        // 아니면 그냥 num이므로 double로 변환
        num = Double.parseDouble(minusCheck);

    }

    return new Number(num);

}

@Override
public AstNodes visitParensExpr(ExprParser.ParensExprContext ctx) {

    // '(' 'expr' ')' => expr이므로 1
    // visitInfixExpr과 동일한 원리로 visit으로 처리

    return this.visit(ctx.getChild(1));
}

@Override
public AstNodes visitAssingExpr(ExprParser.AssingExprContext ctx) {
    // VAR '=' '-'num
    // VAR '=' num

    String variable = ctx.getChild(0).getText();
    double value;

    String minusCheck = ctx.getChild(2).getText();

    if ("-".equals(minusCheck)) {

        String temp = ctx.getChild(3).getText();
        value = (-1) * Double.parseDouble(temp);

    } else {
        // 아니면 그냥 num이므로 double로 변환
        value = Double.parseDouble(minusCheck);

    }

    return new VariableDeclaration(variable, value);
```

위와 동일하게 처리하여주고 음수인 경우만 추가하여 처리한다.

visitNumberExpr을 보면 만약 -가 붙어있는 경우에는 getChild(0).getText()한 값이 '-'일 것이다. 그러므로 해당되는 경우와 안되는 경우를 구분하여 음수를 처리했다. (Assign)도

동일하게 진행하였다.

AntlrToProgram.java

```java
public class AntlrToProgram extends ExprBaseVisitor<ExprNode> {

    @Override
    public ExprNode visitProg(ExprParser.ProgContext ctx) {

        ExprNode program = new ExprNode();

        BuildAstVisitor visitor = new BuildAstVisitor();

        for (int i = 0; i < ctx.getChildCount(); i++) {
            if (i == ctx.getChildCount() - 1) {
                // last child 이므로 visit하면 안됨

            } else {

                // ExprNode 안에 <AstNode>를 담는 List가 있는데 그 List에 <AstNode>들을 추가해주는 것
                // prog -> 아래에 있는 child들을 돌면서
                program.addExpressions(visitor.visit(ctx.getChild(i)));
            }
        }

        return program;
    }
}
```

youtube 영상을 참고하여 코드를 작성하였고, prog가 들어오면 해당 prog의 자식 노드들을 탐색하여 list로 반환해주는 visit 메소드를 작성하였다. (last child는 visit할 필요가 없기에 제외하였다.)

program.java

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTree;

public class program {

    Run | Debug
    public static void main(String[] args) throws IOException {

        List<Double> result = new ArrayList<>();

        // Get Lexer
        ExprLexer lexer = new ExprLexer(CharStreams.fromStream(System.in));

        // Get a list of matched tokens
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Pass tokens to parser
        ExprParser parser = new ExprParser(tokens);

        // Make AST from prog and print the tree
        ParseTree antlrAst = parser.prog();

        AntlrToProgram progVisitor = new AntlrToProgram();

        ExprNode prog = progVisitor.visit(antlrAst);

        Evaluate Evaluator = new Evaluate();

        AstCall astCall = new AstCall();

        for (AstNodes node : prog.expressions) {

            // 마지막 node는 출력되면 안되므로
            if (node != null) {

                astCall.Call(node, tabs:0);
                // Evaluate AST result
                result.add(Evaluator.evaluate(node));
                // Sysout으로 출력하려고 하였지만 여러 input이 들어올 시
                // AST를 다 그리고 마지막에 출력을 해야하므로 아래로 빼야함.
                // System.out.println(Evaluator.evaluate(node));
            }

        }

        for (double res : result) {
            System.out.println(res);
        }

    }
}
```

prog.expressions는 AstNode들을 담고있는 List이기에 해당 리스트에서 각각 AstNode를 가져와서 AST를 그리고 값을 계산한다. 계산한 값은 바로 출력하지 않고 list에 저장한 뒤 AST를 다 그린 후에 출력한다.