1. **Objective**: The task is to make a simple calculator using ANTLR.
2. **Submission Requirements**:
    - Submit source codes (with *.ml file extension) and a report.
    - If the source code does not compile correctly, no points will be awarded.
    - The zip file for submission should contain the following files: AstCall.java, AstNodes.java, BuildAstVisitor.java, Evaluate.java, Expr.g4, program.java, and a Report.

Sure, let's break down the tasks and approach them one at a time.

Step 1: Modify the Grammar

- Given grammar file

```
grammar Expr;

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

expr : expr ('*'|'/') expr  # infixExpr
     | expr ('+'|'-') expr  # infixExpr
     | num                  # numberExpr
     | '(' expr ')'         # parensExpr
     ;

num  : INT
     | REAL
     ;

// lexer rules
NEWLINE: [\r\n]+ ;
INT: [0-9]+ ;          // should handle negatives
```

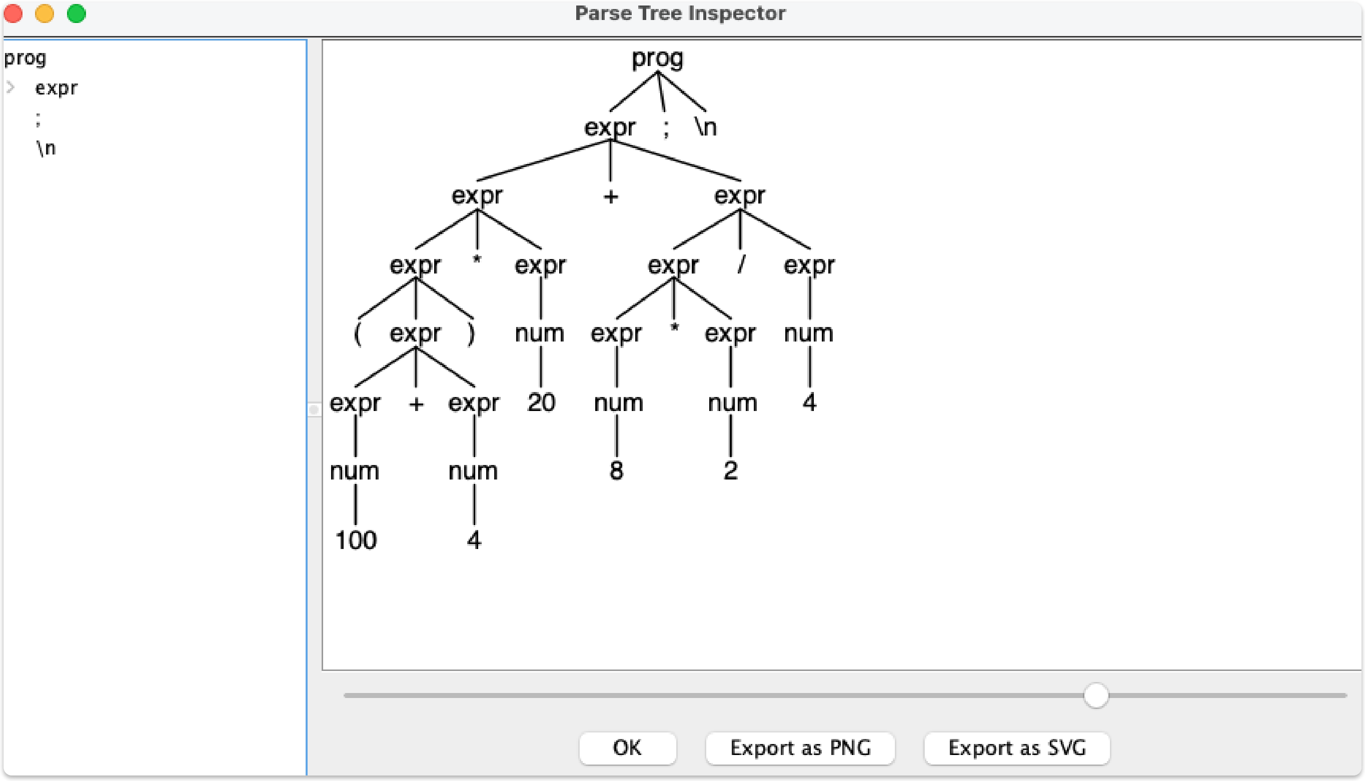
```
REAL: [0-9]+'.'[0-9]* ; // should handle signs(+/-)
WS: [ \t\r\n]+ -> skip ;
```

The reason why the parenthesis is near the end

When we test the parse tree using `grun` with given test case,

```
a=(100+4)*20+8*2/4;
```

It will show the below image. It didn't process assignment well.



We need to enhance the grammar `Expr.g4` to support assignment operations. Here's the modified grammar:

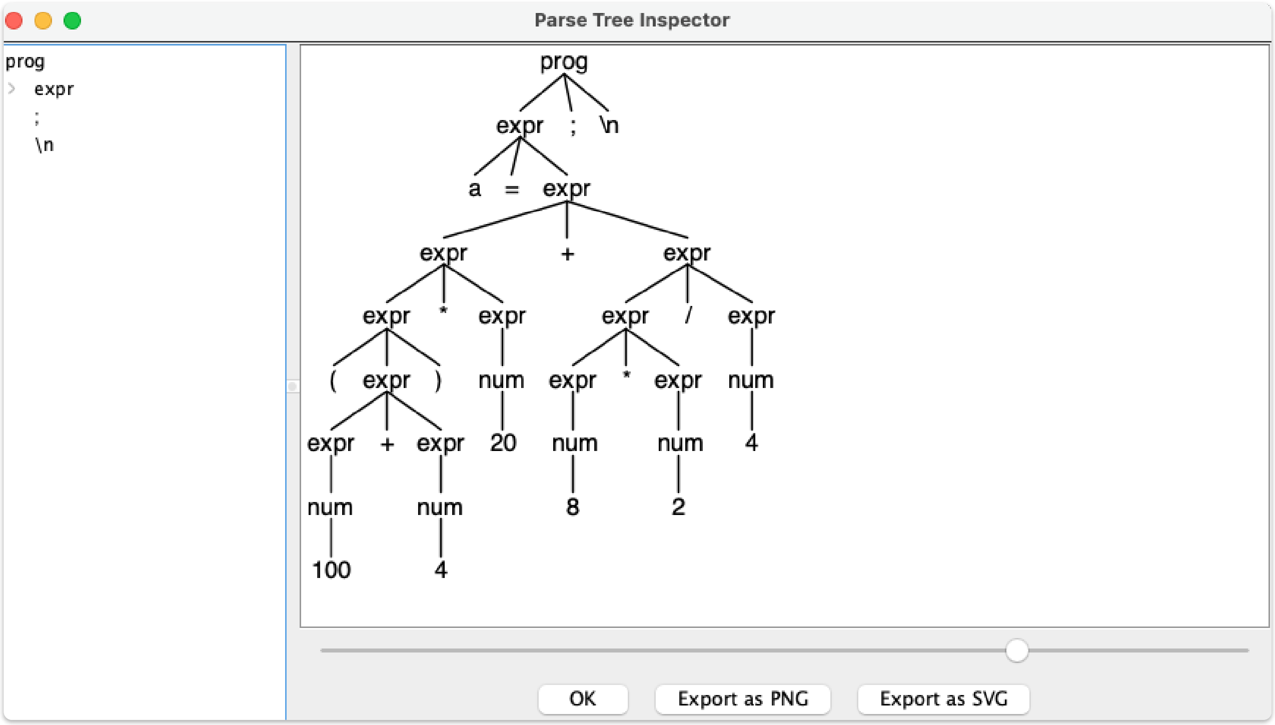
```
 grammar Expr;

 @header {
       package antlr;  // import antlr package (every
generated java file)
 }
```

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

expr: expr op=('*'|'/') expr  # infixExpr
     | expr op=('+'|'-') expr  # infixExpr
     | ID op='=' expr           # assignExpr
     | num                  # numberExpr
     | '(' expr ')'         # parensExpr
     | ID                   # idExpr
     ;

num  : '-'? INT
     | '-'? REAL
     ;

// lexer rules
ID: [a-zA-Z]+ ;             // variable name
NEWLINE: [\r\n]+ ;
INT: [0-9]+ ;
REAL: [0-9]+'.'[0-9]* ;
WS: [ \t\r\n]+ -> skip ;
```

when we test the pares tree with this command,

```
 grun antlr.Expr prog -gui
a=(100+4)*20+8*2/4;
^D
```

- Modified Parse tree image



Step 2: Examination of what should we do next

- program.java

```
import java.io.IOException;
import org.antlr.v4.runtime.*;

public class program {

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

        // 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
        ExprParser.ProgContext ctx = parser.prog();
        ProgNode AST = (ProgNode)new
BuildAstVisitor().visitProg(ctx);
        AST.children.forEach(node -> new
AstCall().Call(node, 0));

        // Evaluate AST result
        Evaluate Evaluator = new Evaluate();
        AST.children.forEach(node ->
System.out.println(Evaluator.evaluate(node)));
    }
}
```

Explanation of program.java

In this program file, what we should implement to do is below parts.

```
ProgNode AST = (ProgNode)new
BuildAstVisitor().visitProg(ctx);
        AST.children.forEach(node -> new
AstCall().Call(node, 0));

        // Evaluate AST result
        Evaluate Evaluator = new Evaluate();
        AST.children.forEach(node ->
System.out.println(Evaluator.evaluate(node)));
```

We will call the `visitProg()` which is in the `BuildAstVisitor()` module. The return type will be `ProgNode` type. In this `ProgNode`, We will iterate its children and call each child by `AstCall()` to print the AST nodes. When we finish printing all AST node, finally, we will calculate each AST node to print final output.

Step 3: Define AST Nodes

In `AstNodes.java`, we will define the nodes required for our AST:

```java
package expression;

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

class AstNodes {
}

class ProgNode extends AstNodes {
    public List<AstNodes> expressions;

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

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

class InfixNode extends AstNodes {
    String op;  // e.g. "+", "-", "*", "/"
    AstNodes left, right;
}

class NumberNode extends AstNodes {
    double value;
}

class IdNode extends AstNodes {
    String IdName;
}
```

```
class AssignNode extends AstNodes {
    IdNode id;
    String op;
    AstNodes right;
}
```

Step 5: Build AST using ANTLR Visitor class

In `BuildAstVisitor.java`, we will override the visitor methods to construct the AST:

```
package expression;

import antlr.ExprBaseVisitor;
import antlr.ExprParser;
import antlr.ExprParser.AssignExprContext;
import antlr.ExprParser.IdExprContext;
import antlr.ExprParser.InfixExprContext;
import antlr.ExprParser.NumberExprContext;
import antlr.ExprParser.ParensExprContext;
import antlr.ExprParser.ProgContext;

public class BuildAstVisitor extends
ExprBaseVisitor<AstNodes> {

    @Override
    public AstNodes visitProg(ProgContext ctx) {
        ProgNode progNode = new ProgNode();

        for (int i = 0; i < ctx.getChildCount(); i++) {
        /*last child of the start symbol(prog) is EOF */
                //Do not visit this child and attempt to
convert it to an Expression object.
            if (i != ctx.getChildCount() - 1) {

progNode.addExpression(visit(ctx.getChild(i)));
```

```
                //visit method is in Antlr library and it
will convert parse tree into expression and recursively do
the visit
            }
        }
        return progNode;
    }

    @Override
    public AstNodes visitInfixExpr(InfixExprContext ctx) {
        InfixNode infixNode = new InfixNode();
        infixNode.left = visit(ctx.expr(0));
        infixNode.right = visit(ctx.expr(1));
        infixNode.op = ctx.getChild(1).getText();  // the
operator is in the middle
        return infixNode;
    }

    @Override
    public AstNodes visitNumberExpr(NumberExprContext ctx)
{
        NumberNode numberNode = new NumberNode();
        numberNode.value =
Double.parseDouble(ctx.getText());
        return numberNode;
    }

    @Override
    public AstNodes visitParensExpr(ParensExprContext ctx)
{
        return visit(ctx.expr());
    }

    @Override
    public AstNodes visitAssignExpr(AssignExprContext ctx)
{
        AssignNode assignNode = new AssignNode();
        assignNode.id = visit(ctx.ID());
        assignNode.op = ctx.getChild(1).getText();
        assignNode.right = visit(ctx.expr());
        return assignNode;
    }

    @Override
    public AstNodes visitIdExpr(IdExprContext ctx) {
        IdNode idNode = new IdNode();
        idNode.IdName = ctx.ID().getText();
        return idNode;
    }
}
```

Step 4: Printing the AST

```
In `AstCall.java`, we will create methods to print the AST
nodes:
```

```
package expression;

class AstCall {

    public void Call(AstNodes node, int depth) {
        if (node == null) {
            return;
        }
        for (int i = 0; i < depth; i++) {
            System.out.print("    ");
        }

        if (node instanceof InfixNode) {
            InfixNode infixNode = (InfixNode) node;
            System.out.println(infixNode.op);
            Call(infixNode.left, depth + 1);
            Call(infixNode.right, depth + 1);

        } else if (node instanceof NumberNode) {
            NumberNode numberNode = (NumberNode) node;
            System.out.println(numberNode.value);

        } else if (node instanceof IdNode) {
            IdNode idNode = (IdNode) node;
            System.out.println(idNode.IdName);

        } else if (node instanceof AssignNode) {
            AssignNode assignNode = (AssignNode) node;
            System.out.println(assignNode.op);
            Call(assignNode.id, depth + 1);
            Call(assignNode.right, depth + 1);

        } else if (node instanceof ProgNode) {
            ProgNode progNode = (ProgNode) node;
            for (AstNodes n : progNode.expressions) {
                Call(n, 0);
            }
        }
    }
}
```

Step 5: Evaluating the AST

In `Evaluate.java`, implement the method to evaluate the AST:

```
package expression;

import java.util.HashMap;
import java.util.Map;

class Evaluate {
    private Map<String, Double> variables = new HashMap<>();

    public double evaluate(AstNodes node) {
        if (node instanceof InfixNode) {
            InfixNode infixNode = (InfixNode) node;
            double left = evaluate(infixNode.left);
            double right = evaluate(infixNode.right);
            switch (infixNode.op) {
                case "ADD":
                    return left + right;
                case "SUB":
                    return left - right;
                case "MUL":
                    return left * right;
                case "DIV":
                    return left / right;   // you don't have to worry about divison error.
                default:
                    //Just a place holder
                    return 0;
            }

        } else if (node instanceof NumberNode) {
            NumberNode numberNode = (NumberNode) node;
            return numberNode.value;

        } else if (node instanceof IdNode) {
            IdNode idNode = (IdNode) node;
            if (variables.containsKey(idNode.IdName)) {
                return variables.get(idNode.IdName);
            } else {
                System.err.println("Undefined variable: " + idNode.IdName);
                return 0.0;
            }
        } else if (node instanceof AssignNode) {
            AssignNode assignNode = (AssignNode) node;
            double value = evaluate(assignNode.right);
            String id = ((IdNode)assignNode.id).IdName;
            variables.put(id, value);
            return value;

        } else {
            return 0; // Default value for any other node
        }
    }
}
```

To run the program

```
java expression.program
```