Programming Language Assignment 1

2020314193 HWNAG SU YEONG

Expr.g4

토큰

```
// lexer rules
NEWLINE: [\r\n]+ ;
INT: '0' | '-'?[1-9][0-9]* ;          // should handle negatives
REAL: ('0' | '-'?[1-9][0-9]*) '.'[0-9]+ ; // should handle signs(+/-)
ID: [a-zA-Z_][a-zA-Z0-9_]* ;
WS: [ \t\r\n]+ -> skip ;
```

INT

- 음수인 경우 마이너스 부호인 `-`가 맨 앞에 오고 양수인 경우 그렇지 않다
- `-0`은 토큰으로 인정되지 않는다
- `032`와 같이 0이 아닌 수는 맨 앞에 0이 오지 못한다

REAL

- 정수부와 소수부는 `.`으로 구분된다
- 정수부의 규칙은 INT와 동일하다
- 소수부의 마지막 수는 0으로 끝나는것을 허용한다
  가령 `1.0`과 같은수는 토큰으로 허용되며 INT가 아닌 REAL 토큰으로 인식된다

ID

- 숫자가 맨 앞에 오는 경우 토큰으로 허용되지 않는다
  일반적인 변수 규칙은 맨 앞에 숫자가 오는걸 허용하지 않는다

파서

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

statement : (expr | decl)     # Instruction;

expr : '(' expr ')'           # Parenthesis
     | expr ('*'|'/') expr    # Operation
     | expr ('+'|'-') expr    # Operation
     | num                    # Number
     | ID                     # Variable
     | '(' expr ')'           # Parenthesis
     ;
```

prog

- start rule에 해당된다
- statement 뒤에는 항상 `;`이 와야한다

statement

- expr 혹은 decl을 statement로 인식한다

expr

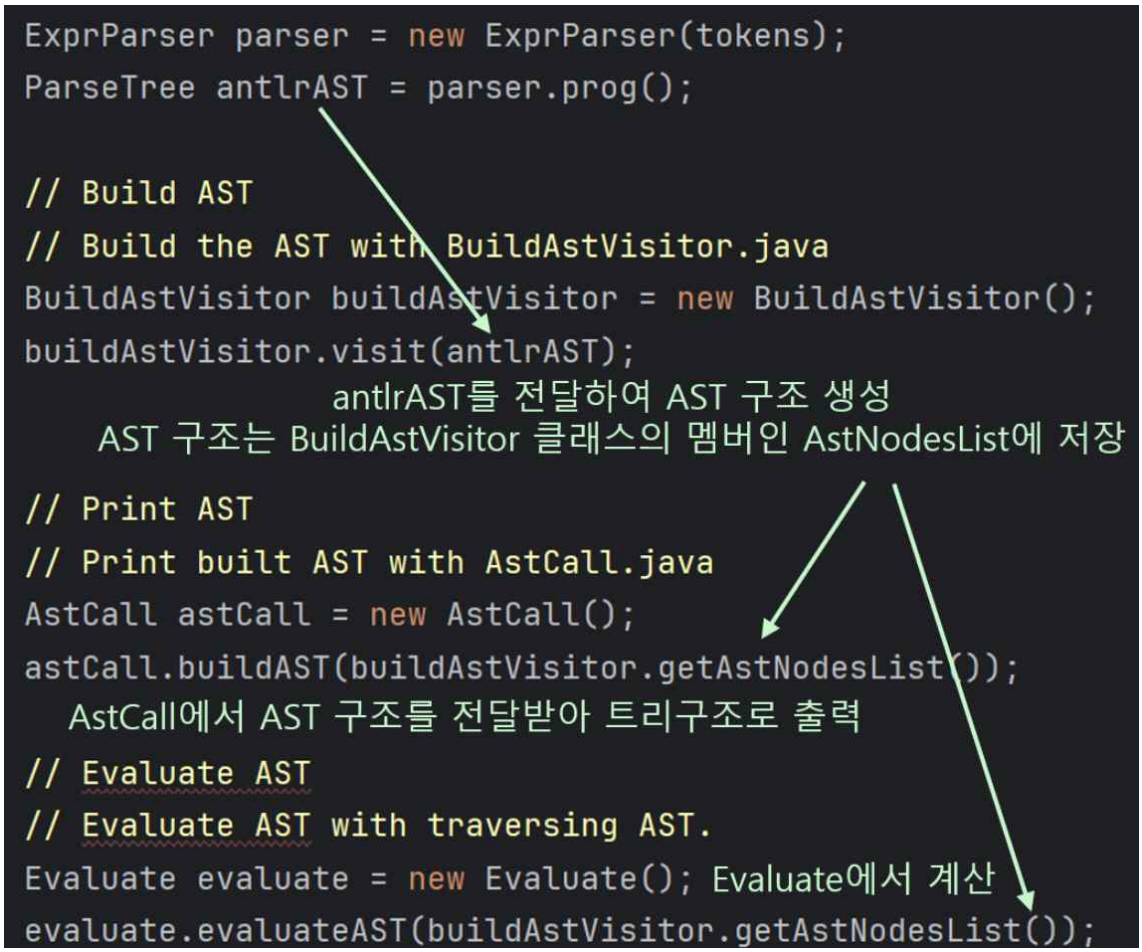
- .g4파일의 경우 상단에 위치한 production rule일수록 우선순위가 높다
  이를 고려하여 production rule을 배치하였다

→ 괄호친 expr의 우선순위가 가장 높으며, 곱하기와 나누기는 더하기 빼기보다 높은 우선순위를 갖으며 동일한 우선순위를 갖는다

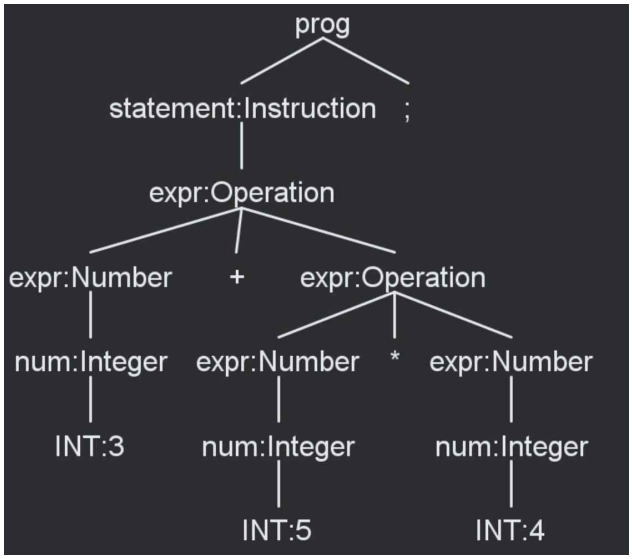
java 파일

program.java

- main 메소드를 갖고있어 프로그램의 entry point를 담당하는 클래스이다



parser.prog()가 생성하는 AST는 전체 Parse Tree이다



왜냐하면 위 parse tree에서 확인할 수 있듯이, parser가 작성하는 parse tree의 루트 노드는 항상 prog이므로 전체 Parse Tree가 antlrAST에 있다.

AstNodes.java

```
public class AstNodes {
    6개 사용 위치
    AstNodes left;
    6개 사용 위치
    AstNodes right;
    7개 사용 위치
    double value;
    11개 사용 위치
    String type;

    4개 사용 위치
    String id;
```

- 트리구조를 이루게되는 핵심적인 부분은 사칙연산이다

→ 사칙연산의 각 연산자는 피연산자가 두 개이므로 AstNodes의 내부 멤버로 left와 right의 AstNodes를 갖게 한다

- value에 값을 저장하고, type에 AstNode가 어떤 타입인지 기록한다
- 변수를 사용하는 경우 id를 통해 변수를 인식할 수 있도록 하였다

BuildAstVisitor.java

```
private Map<String, Double> vars;
3개 사용 위치
private List<AstNodes> astNodesList;
```

- vars : Map 자료구조를 통해 id(혹은 코드의 varName)을 키값으로 변수의 값을 저장
- astNodesList : AstNodes이 저장되는 곳
  astNodeList는 AstCall이나 Evaluate클래스에 전달되어 트리구조를 출력하고 결과값을 계산하는데 사용된다

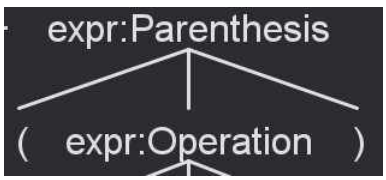
- visitInstruction 메소드

```
@Override
public AstNodes visitInstruction(ExprParser.InstructionContext ctx) {
    AstNodes astNodes = visit(ctx.getChild( i: 0));
    astNodesList.add(astNodes);
    return super.visitInstruction(ctx);
}
```

과제에서 요구하는 출력값은 작성한 `Expr.g4`기준으로 Instruction(statment)단위로 출력하면 된다
따라서 `astNodesList`에 추가되는 `AstNodes`는 모두 ParsTree의 `Instruction`노드 방문시 저장하게 된다

- visitParenthesis 메소드

```
@Override
public AstNodes visitParenthesis(ExprParser.ParenthesisContext ctx) {
    return visit(ctx.getChild( i: 1));
}
```



- 1번째 자식이 리턴하는 `AstNodes`를 그대로 리턴

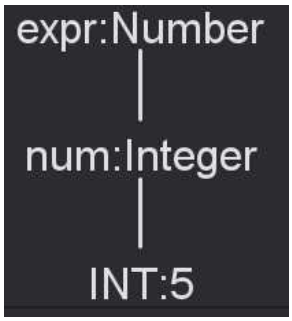
- visitVariable 메소드

```
@Override
public AstNodes visitVariable(ExprParser.VariableContext ctx) {
    String varName = ctx.getChild( i: 0).getText();
    double num = Integer.MIN_VALUE;
    if(vars.containsKey(varName)) num = vars.get(varName);
    return new AstNodes(num, varName);
}
```

- 현재 테스트케이스에서 참조되는 변수는 언제나 선언이 된 후에 참조되고 있다
- 그러나 선언되지 않은 변수가 참조되는 경우에는 별도 처리를 해주어야 한다
  이를 위해 else문을 추가하여 선언되지 않는 변수가 참조되는 경우 핸들링을 할 수 있다
- AstNode에 변수명과 값을 담고 리턴한다

- visitNumber 메소드

```java
@Override
public AstNodes visitNumber(ExprParser.NumberContext ctx) {
    return visit(ctx.getChild( i: 0));
}
```



- expr의 production rule 중 하나에 해당하는 Number는 언제나 자식으로 Integer 혹은 Real을 가진다

  → 자식이 리턴하는 AstNodes를 그대로 반환

- visitOperation 메소드

```java
@Override
public AstNodes visitOperation(ExprParser.OperationContext ctx) {
    AstNodes left = visit(ctx.getChild( i: 0));
    AstNodes right = visit(ctx.getChild( i: 2));
    String operator = ctx.getChild( i: 1).getText();
    switch(operator) {
        case "*": return new AstNodes(left, right, type: "MUL");
        case "/": return new AstNodes(left, right, type: "DIV");
        case "+": return new AstNodes(left, right, type: "ADD");
        case "-": return new AstNodes(left, right, type: "SUB");
        default: return super.visitOperation(ctx);
    }
}
```

- Operation의 경우에 두 개의 피연산자를 갖으므로 0번째, 2번째 자식을 각각 left, right에 할당한다

- Operation의 1번째 자식의 값을 확인하여 해당하는 연산자에 맞는 type과 left, right를 인자로 할당하여 AstNodes를 생성하여 리턴한다

- visitIntDeclaration, visitDoubleDeclaration 메소드

```java
@Override
public AstNodes visitIntDeclaration(ExprParser.IntDeclarationContext ctx) {
    String varName = ctx.getChild( i: 0).getText();
    String numText = ctx.getChild( i: 2).getText();
    Double num = Double.parseDouble(numText);

    if(vars.containsKey(varName)) vars.replace(varName, num);
    else vars.put(varName, num);

    return new AstNodes( type: "ASSIGN", varName, num);
}
```

```java
@Override
public AstNodes visitRealDeclaration(ExprParser.RealDeclarationContext ctx) {
    String varName = ctx.getChild( i: 0).getText();
    String numText = ctx.getChild( i: 2).getText();
    Double num = Double.parseDouble(numText);

    if(vars.containsKey(varName)) vars.replace(varName, num);
    else vars.put(varName, num);

    return new AstNodes( type: "ASSIGN", varName, num);
}
```

- 이 과제에서는 이미 선언한 변수를 재 선언하는것이 가능하기 때문에 if문을 통해 이미 존재하는 변수의 값을 바꿔주도록 하였다
- AstNodes에 `ASSIGN`타입을 적어주고 변수명과 값을 넣어서 리턴한다

- visitInteger, visitReal 메소드

```java
@Override
public AstNodes visitInteger(ExprParser.IntegerContext ctx) {
    String numText = ctx.getChild( i: 0).getText();
    Double num = Double.parseDouble(numText);
    return new AstNodes(num);
}
```

```java
@Override
public AstNodes visitReal(ExprParser.RealContext ctx) {
    String numText = ctx.getChild( i: 0).getText();
    Double num = Double.parseDouble(numText);
    return new AstNodes(num);
}
```

- 각각 값을 파싱하여 AstNodes의 생성자에 전달하고 그것을 리턴한다

정리하자면 `BuildAstVisitor`의 `visitor*`메소드는 각 노드에 방문할때마다 AstNodes를 리턴한다. AstNodes는 left, right 멤버를 통해 트리구조를 가지게 되고, 마지막 visitInstruction 메소드에서 이를 `BuildAstVisitor` 클래스의 멤버 astNodesList에 저장한다.

AstCall.java

- buildAST 메소드

```
public void buildAST(List<AstNodes> astNodesList) {
    for (AstNodes astNodes : astNodesList) {
        String type = astNodes.type;
        if (astNodes.type.equals("ASSIGN")) {
            System.out.println("ASSIGN");
            System.out.println("\t" + astNodes.id);
            System.out.println("\t" + astNodes.value);
        }
        else traverseAST(astNodes, depts: 0);
    }
}
```

`BuildAstVisitor` 클래스의 멤버 astNodesList를 넘겨받아서
각 astNodes를 확인하여 type이 ASSIGN인 경우, 굳이 트리구조를 통해 출력할 필요 없으므로 바로 출력한다
type이 ASSIGN이 아닌 경우 traverseAST함수를 통해 출력한다

- traverseAST

```
private void traverseAST(AstNodes astNodes, int depts) {
    String type = astNodes.type;
    for (int i = 0; i < depts; i++) System.out.print("\t");
    if(type.equals("terminal")) {
        System.out.println(astNodes.value);
        return;
    }
    if(type.equals("variable")) {
        System.out.println(astNodes.id);
        return;
    }

    System.out.println(astNodes.type);
    traverseAST(astNodes.left, depts: depts + 1);
    traverseAST(astNodes.right, depts: depts + 1);
}
```

- 재귀적으로 AstNodes를 순회하여 출력한다

Evaluate.java

- evaluateAST

```java
public void evaluateAST(List<AstNodes> astNodesList) {
    for (AstNodes astNodes : astNodesList) {
        String type = astNodes.type;
        if (astNodes.type.equals("ASSIGN")) {
            System.out.println("0.0");
        }
        else System.out.println(traverseAST(astNodes));
    }
}
```

type이 "ASSIGN"인 경우 표현식을 계산할 필요 없이 바로 0.0을 출력한다
type이 "ASSIGN"이 아닌 경우 계산값을 traverseAST를 통해 얻어 출력한다

```java
private double traverseAST(AstNodes astNodes) {
    String type = astNodes.type;
    if(type.equals("terminal")) {
        return astNodes.value;
    }
    if(type.equals("variable")) {
        return astNodes.value;
    }

    switch (type) {
        case "ADD": return traverseAST(astNodes.left) + traverseAST(astNodes.right);
        case "SUB": return traverseAST(astNodes.left) - traverseAST(astNodes.right);
        case "MUL": return traverseAST(astNodes.left) * traverseAST(astNodes.right);
        case "DIV": return traverseAST(astNodes.left) / traverseAST(astNodes.right);
        default: return  0;
    }
}
```

- 재귀적으로 AstNodes를 순회하여 표현식을 계산한다