CE305 Language And Compiler (Assignment 1)

Student no: 2004595

Ogundiran Jamie

# ABSTRACT

In this paper, I will be building an expression analyser that takes arithmetic expressions. The arithmetic expression handles integer, floating-point number, and variables. The main features for the expression analyser include evaluator that evaluates and calculates the arithmetic expression, pretty-print function that prints the parsed tree in a readable form, a tree graph that displays the parse tree and a support for multiples statements and control flow for multiple statements with Boolean expressions.

# INTRODUCTION

Antlr also known as (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.[[1]](#endnote-1) It's widely used to build languages, tools, and frameworks, some software built using Antlr are Groovy, Jython, twitter’s search query language, etc. On GitHub there are over 200 grammars implemented using ANTLR 4, the implemented grammars include grammars for URL to grammars for entire languages like C, Java and Go. From a grammar, ANTLR generates a parser that can build and walk parse trees. [[2]](#endnote-2) In this paper, with the use of Antlr and Java we will investigate the ways of implementing Antlr, creating a grammar file for a simple arithmetic expression and creating pretty print function and evaluator with its own features.

# SPECIFICATION

Grammar file is used to implement the parser and lexer rules to the compiler, in this project I’m planning to create an arithmetic expression that accepts both integer and variable and supports assignment.

Here is the grammar file that does the given requirement above:

grammar Arithmetic;  
  
start : statementList;  
  
statementList: statement+;  
  
statement: assign | expr;  
  
assign: VARIABLE '=' expr;  
  
expr: expr op=('\*' | '/') expr #Mult  
 | expr op=('+' | '-') expr #Add  
 | '(' expr ')' #Paren  
 | NUMBER #NUMBER  
 | VARIABLE #VARIABLE  
 ;  
  
NUMBER: '-'? DIGIT+ ('.' DIGIT+)?;  
VARIABLE: [a-zA-Z]+;  
  
fragment DIGIT: [0-9];  
fragment NEWLINE: [\r\n]+;  
IGNORED: NEWLINE -> skip;  
WS: [ \t\r\n]+ -> skip;

The lexer rules are described at the bottom side of the code, it defines what rules/regular expression are allowed to be used on a variable set on the parser rules. First, we define the fragment which is a function that makes the lexer rules easier to read. We defined two fragments, DIGIT and NEWLINE, DIGIT consists of a regular expression that matches any integer and NEWLINE consists of a regular expression that matches newline. We then define NUMBER, VARIABLE, IGNORED and ws. NUMBER consists of a regular expression with fragment DIGIT that matches integer with float number or a dash in between, VARIABLE consists of a regular expression that matches any alphabet character or word, IGNORED consists of a regular expression that skips the newline defined in fragment NEWLINE and ws skips any whitespace.

The parser rules consist of start which holds statementlist, statementlist which consists of more than one statements, statement which consists of assign and expression, assign which consists of VARIABLE from lexer rule and “=” and expr and expr which consists of parser rules for arithmetic expression.

# IMPLEMENTATION

## 3.2Evaluator

The evaluator extends arithmeticBaseListener generated from the Antlr recognizer. The class have functions exitAssign, exitNumber, exitVariable, exitStart, exitMult and exitAdd which was overridden from ArithmeticBaseListener.

private Stack<Double> stack = new Stack<Double>(); //stores variable, number and the result  
Map<String, Double> variables = new HashMap<>(); // stores the assigned variable

We create a stack named stack for handling calculation between the expression and a hash map named variables to store the variable and the integer of the assigned variables. The variables for the map and stack are defined in double to allow manipulation of float numbers during the calculation.

@Override  
public void exitAssign(ArithmeticParser.AssignContext context)//Assigning variables to the hash map variables  
{  
 String variableName = context.VARIABLE().getText();  
 double Number = Double.*parseDouble*(context.expr().getText());  
 variables.put(variableName, Number);  
}

Function exitAssign gets the variable and the number values from the ArithmeticParser class and stores it in a hash map variables for later use.

@Override  
public void exitNUMBER(ArithmeticParser.NUMBERContext context)//storing the number in a stack  
{  
 Double i = Double.*valueOf*(context.NUMBER().getText());  
 stack.push(i);  
}  
@Override  
public void exitVARIABLE(ArithmeticParser.VARIABLEContext context)//storing the value of the variables in map to a stack  
{  
 String i = context.getText();  
 double a = variables.getOrDefault(i,0.0);  
  
 stack.push(a);  
}

ExitNUMBER function obtains the number from Arithmetic Parser class and turn it into a double so that it could be stored in stack. ExitVARIABLE function gets the character/word of the assigned value, get the number of the assigned variable using “.getOrDefualt(i,0)” which gets the value corresponding to the key and push it to the stack for later use.

@Override  
public void exitMult(ArithmeticParser.MultContext context) //calculation for the operator multiply or divide  
{ //using the values stored in the stack  
 double left = stack.pop(); //values.get(context.expr(0));  
 double right = stack.pop(); //values.get(context.expr(1));  
 String operator = context.op.getText();  
 double result;  
 if (operator.equals("\*")) {  
 result = left \* right;  
 } else {  
 result = left / right;  
 }  
 stack.push(result);  
}  
  
@Override  
public void exitAdd(ArithmeticParser.AddContext context)  
 //calculation for the operator addition or  
{ //subtraction using the values stored in the stack  
 double left = stack.pop(); // values.get(context.expr(0));  
 double right = stack.pop(); //values.get(context.expr(1));  
 String operator = context.op.getText();  
  
 double result;  
 if (operator.equals("+")) {  
 result = left + right;  
 } else {  
 result = left - right;  
 }  
 stack.push(result);  
 }  
}

exitMult function and exitAdd function does the calculation of the expression. It uses stack.pop() to get the value on the left of the expression and the right of the expression, for example if the expression is 1 + 10 \* 2, the left would be 1 and the right would be 10 for exitAdd. The values obtained are now then calculated depending on what the operator is and added to the stack as a result. Similarly, for exitMult it calculates the left and the right value with the appropriate operator and stores the result in a stack.

@Override  
public void exitStart(ArithmeticParser.StartContext context)//printing the result  
{  
 double result;  
 result = stack.pop();  
  
 System.*out*.println("result: " + result);  
}

exitStart prints the accumulated result of the expression by doing stack.pop().

## 3.3PrettyPrint

similarly, to the evaluator, class pretty print extends the arithmeticBaseListener. Pretty print consists of functions enterAsign, exitAsign, enterNUMBER, enterVARIABLE, enterMult, enterAdd, exitAdd, enterParen and getResult.

private StringBuilder sb = new StringBuilder();

We initialise StringBuilder named sb which hold strings of characters.

@Override  
public void enterAssign(ArithmeticParser.AssignContext context) {  
 sb.append("(= ").append(context.VARIABLE().getText()).append(" ");  
}  
  
@Override  
public void exitAssign(ArithmeticParser.AssignContext context) {  
 sb.append(" )");  
}

The enterAssign function and exitAssign function concatenate (= AssignedValue) into sb. For example if 10 is assigned to a enterAssign and exitAssign will concatenate (= a (10) ) into sb.

@Override  
public void enterNUMBER(ArithmeticParser.NUMBERContext context) {  
 sb.append("( ").append(context.getText()).append(" )");  
}  
//creates ( number ) and append to sb

The enterNumber function appends ( NUMBER ) into sb.

@Override  
public void enterVARIABLE(ArithmeticParser.VARIABLEContext context) {  
 sb.append("( ").append(context.getText()).append(" )");  
}  
//create ( variable ) and append to sb

The enterVARIABLE function appends ( VARIABLE ) into sb.

@Override  
public void enterMult(ArithmeticParser.MultContext context) {  
 sb.append("(\* ");  
}  
  
@Override  
public void exitMult(ArithmeticParser.MultContext context) {  
 sb.append(")");  
}

The enterMult and exitMult function appends (\*) into sb.

@Override  
public void enterAdd(ArithmeticParser.AddContext context) {  
 sb.append("(+ ");  
}  
  
@Override  
public void exitAdd(ArithmeticParser.AddContext context) {  
 sb.append(")");  
}  
//create (+ ) and append to sb

The enterAdd and exitAdd function appends (+) into sb.

@Override  
public void enterParen(ArithmeticParser.ParenContext context) {  
 sb.append("(");  
}  
  
@Override  
public void exitParen(ArithmeticParser.ParenContext context) {  
 sb.append(")");  
}

enterParen and exitParen function appends () into sb. The brackets will surround the expressions.

//appends () around the expression  
public String getResult() {  
 return sb.toString();  
}

Returns the result.

# TESTING

## 4.1 Arithmetic Expression

The text file we will be using to test the functionality and the feature of this program is:

x = 5  
y = 6  
c = 20  
x \* y + c + c

The expected result is 70 for the evaluator and the expected pretty print in LISP-like text form is

(= x ( 5 ) )(= y ( 6 ) )(= c ( 20 ) )(+ (+ (\* ( x )( y ))( c ))( c )).

To test it we first create a main function which implements the lexer and the parser. Therefore, we created a main function that reads the text file to be tested on, a lexer object, a parser object. We initialised the evaluate expression in line 4.

private final static String *DIR* = "./src/Assignment1/";

CharStream cs = *fromFileName*(*DIR* + "text.cc"); //read the input file  
ArithmeticLexer lexer = new ArithmeticLexer(cs); //create a lexer object  
ArithmeticParser parser = new ArithmeticParser(new CommonTokenStream(lexer));  
parser.addParseListener(new evaluateExpression());  
ParseTree parseTree = parser.start();

parser.start();

For pretty print we initialised it and used the built in Antlr function ParseTreeWalker to parse through each node and printed the pretty print output by calling getResult() function.

Pprint printer = new Pprint();  
ParseTreeWalker walker = new ParseTreeWalker();  
walker.walk(printer, parseTree);  
  
System.*out*.println("LISP-like text form: " + printer.getResult());

With that when the program is run, we get:

result: 70.0

LISP-like text form: (= x ( 5 ) )(= y ( 6 ) )(= c ( 20 ) )(+ (+ (\* ( x )( y ))( c ))( c ))

It’s passed the initial test where it outputted the right calculation for the expression in the text file and the right format for pretty print. I decided to do another test to see if the program handles decimal numbers.

Here is the text file that was tested:

x = 13.2  
y = 6.6  
c = 20.5  
x \* y + c

The result outputted were:

result: 107.61999999999999

LISP-like text form: (= x ( 13.2 ) )(= y ( 6.6 ) )(= c ( 20.5 ) )(+ (\* ( x )( y ))( c ))

Which is the right calculation for both evaluator and pretty print.

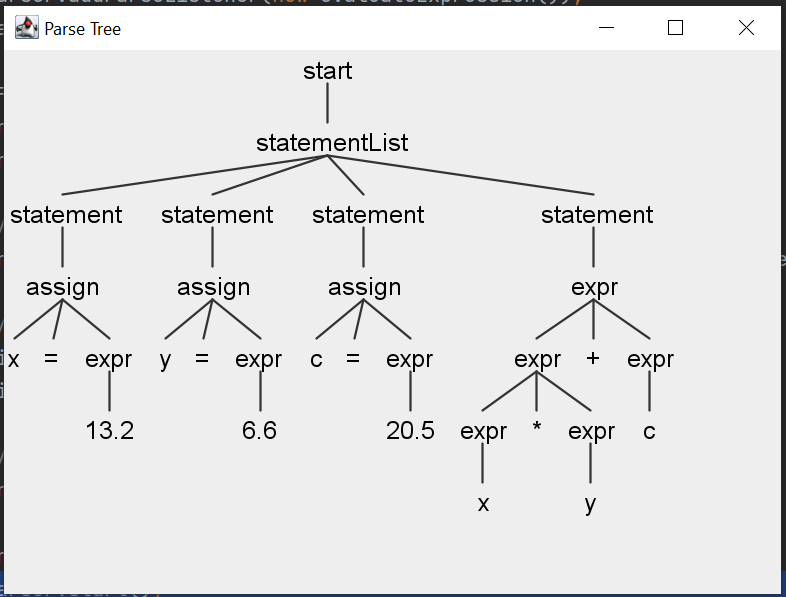
## 4.2Graph

To check if the nodes of the parse trees are placed in the right position, I used a graph to display it. The tool that I used to display the parse tree graph are Java swing and Tree viewer from Antlr inline function.

JFrame frame = new JFrame("Parse Tree");  
frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
frame.setSize(400, 400);

TreeViewer viewer = new TreeViewer(Arrays.*asList*(parser.getRuleNames()), parseTree);  
  
// Set some options for the TreeViewer  
viewer.setScale(1.5);  
viewer.setAutoscrolls(true);  
  
// Add the TreeViewer to the frame and show the frame  
frame.add(viewer);  
  
frame.setVisible(true);

Tree viewer display the graph by inputting its parse Tree which was initiated earlier with its Names of the rules. After adding viewer to Jframe and setting it to visible we get:



Which is the parseTree of the text file used earlier when testing for decimal numbers.

# 5.Conclusion

In conclusion, compilers can be built in Antlr using a grammar file to specify its rules, implementing evaluator and pretty print will let you add features to the grammar file such as calculating an expression, assigning values and printing the parse Tree into a easier readable text form like LISP-text form. There are a lot of possibilities with you can build using Antlr.

1. (Parr, 1989) [↑](#endnote-ref-1)
2. (Wikipedia, 2023) [↑](#endnote-ref-2)