CE305 Assignment 1
Expression Analyser

Drew Berry

2000878

Specification

Tokens

VAR

- Consists of one or more letters.
- Used to denote variables.

NUM

- Consists either of one or more number, or one or more numbers followed by a period and one or more numbers.
- Used to denote an integer or a floating point number.

NEWLINE

- Consists of one or more \r or \n.
- Used to support multiple statements in the input file.

Operators ADD and MULT

- ADD refers to addition and subtraction, whereas MULT refers to multiplication and division.
- ADD consists of a plus (+) or a minus (-), whereas MULT consists of a star (*) or a slash (/).

Others

- Each opening and closing brackets are separate tokens, so “(“ is one token and “)” is another.
- The caret (^) symbol is used to denote indices.
- The equals (=) symbol is used to assign a value to a variable.

Syntax

First of all, any whitespace given in the input file is ignored.

Each line of the input file must be an assignment statement or an expression, where the assignment statement consist of a variable name followed by an equals symbol and an expression. Each of these expressions are one of five types, which are as follows:

- Two expressions separated by a star (*) or a slash (/).
- Two expressions separated by a plus (+) or a minus (-).
- A number or a number raised to a power.
- A variable or a variable raised to a power.
- An expression inside of brackets or the same but raised to a power.

Implementation

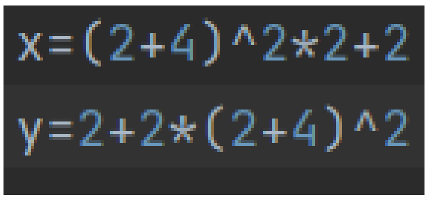
This program requires an input file called “test.cc” to run, this needs to be included in the same package as the other source code files.

Inside the grammar file (ExprLang.g4), I implemented several terminal rules to ensure my syntax worked as intended:

- “prog” is a rule that recursively calls itself to build up the whole input file, each recursion adds one or more assignments or expressions to itself, as each line of the program must be one of the two.
- “expression” is a rule that refers to the expr rule, it is used to find the final expression of a line.
- “assign” is a rule that denotes an assignment statement, which consists of a variable, an equals symbol and an expression.
- “expr” is a rule that consists of five sub rules, where the first two of these rules include “expr” themselves allowing the connection of two expressions with an operator. The five sub rules are as follows:
    - “EXPR1” which denotes the multiplication or division between two expressions, this comes before “EXPR2” as it allows the EvalVisitor class to evaluate expressions in the correct order of “BIDMAS”, *see below*.
    - “EXPR2” which denotes the addition or subtraction between two expressions.
    - “EXPR3”, “EXPR4” and “EXPR5” are the starting building blocks of an expression. “EXPR3” is for numbers, “EXPR4” is for variables and “EXPR5” is for brackets.
- “brackets” is a rule that encapsulates an expression in brackets.

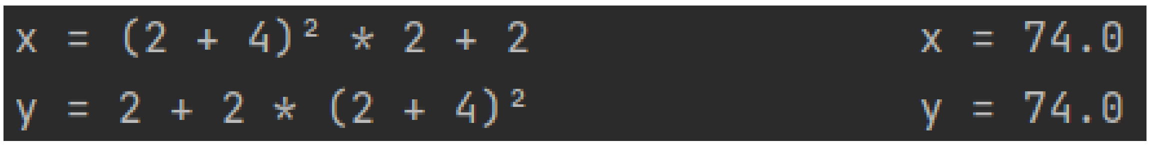
“BIDMAS”

Here is an input file:



```
x=(2+4)^2*2+2
y=2+2*(2+4)^2
```

This is the programs output:



```
x = (2 + 4)² * 2 + 2                    x = 74.0
y = 2 + 2 * (2 + 4)²                    y = 74.0
```

As you can see, both statements evaluate to the same value when write in opposite orientations, this demonstrates that the evaluations are correct for “BIDMAS”.

MainErrorListener

This is a class that extends BaseErrorListener. It is used to catch syntax errors in the input file and return error messages to the user. When an error occurs its details are saved to an ArrayList to be displayed later in MainParser.

PrettyVisitor

This is a class that extends ExprLangBaseVisitor with type string. It is used to convert the input file to a pretty print form. This is achieved by traversing the parse tree that is passed into the visitStart() function. Whenever the input file is empty or has an error unrelated to the syntax, the methods of this class return the string "ERROR" which is returned by visitStart() to the MainParser class and displays an error to the user.

The visit function for each terminal rule are overwritten in this class and flow as defined in the grammar file. The "expr" rule is separated into the five sub rules for these overwritten functions, so I implemented a function called visitExpr(). This relays the context that is passed into it to the respective sub rules, for example, if the context is an EXPR1Context it is passed into visitEXPR1(). Each visit function for a rule that appears in the grammar file below prog returns the pretty print of a statement as a string if there are no errors. When these returns reach back to visitProg(), the strings are added to the strings ArrayList to be displayed later in MainParser.

I also implemented a superscript() function, this converts a given string of numbers to the superscript version via the use of Unicode and the replaceAll() function, for example 123 becomes ^{123}. Refer to the references section for more info on this.

EvalVisitor

This is a class that extends ExprLangBaseVisitor with type double. It is used to evaluate each line of the input file. This is achieved by traversing the parse tree that is passed into the visitStart() function in the same way as I have previously stated for the PrettyVisitor class, however instead of returning the pretty print, this class's methods return the evaluation of each line, which are then stored in the strings ArrayList to be displayed later in MainParser. Whenever a variable is assigned a new value, it is stored in a HashMap called assignments, this happens in visitAssign() function. The visitEXPR4() function checks the assignments HashMap for a current variable, if the variable isn't in the HashMap, the input file is invalid as it includes a reference to an undefined variable. Whenever this occurs or when there is an unseen error, the methods of this class return a Double.NaN and sets a field called "valid" to false, this is then used by the MainParser class to display an error to the user.

MainParser

This class consists of only a main method. To begin, the input file is converted into a CharStream which is then used to create a lexer, which is then used to create a token stream, which is then used to create a parser. The lexer and parser have their error listeners removed and then each get a MainErrorListener added to them. Next, an instance of PrettyVisitor is created, and the visit() function is called. If there are any errors in the syntax, output these errors, otherwise continue. Then if the result of the PrettyVisitor.visit() function call isn't equal to null, output that the input file is invalid, otherwise continue. Next, reset the parser, create an instance of EvalVisitor and call its visit() function. If the EvalVisitor's valid field is false, output that the input file contains a reference to an undefined variable, otherwise continue. Now retrieve the strings ArrayLists from both Visitor classes and display them one line at a time.

Extended features

I implemented the handling of both integer and floating point numbers by including a decimal point in the NUM rule in the grammar file, and then taking the NUM tokens and parsing them as doubles.

I also implemented support for multiple statements by adding the prog rules to the grammar file, and recursing through each prog to find all statements.

References

A post by a user who goes by AvrDragon helped me write my superscript() function in the PrettyVisitor class. The post is at this link:

superscript in Java String - Stack Overflow