Simple Language Grammar and Interpreter

The grammar defines a simple expression language which is able to handle variables, arithmetic operations, logical expressions, relational comparisons and logical negation.

**Grammar Rules**

**Program Rule –** The program consists of a list of statements. Each statement can be a declaration, assignment or standalone expression.

**Statement Rule –** Each statement can either be a declaration, assignment, or an expression.

**Declaration Rule –** initializes a variable. This rule only handles the declaration and not assignment.

**Assignment Rule –** updates the value of a variable.

**Expression Rules –** in order of precedence from low to high as defined in the order in the grammar, the expression rules are as follows:

* **Logical OR Expression –** handles the Boolean logic for the OR (||) operator.
* **Logical AND Expression –** handles the Boolean logic for the AND (&&) operator.
* **Equality Expression –** handles the == and != operators. Two relational expressions are compared with the two (EQ and NEQ) operators.
* **Relational Expressions –** handles the >, <, >=,<= operators (greater than, less than, greater than or equals to, less than or equals to respectively). Expression made up of additive expressions.
* **Additive Expressions –** handles the arithmetic operations of + (addition) and – (subtraction). Made up of multiplicative expressions.
* **Multiplicative Expression –** handles the\* (multiplication) and / (division) operations.
* **Unary Expression –** handles the ! (NOT) operator of a primary expression.
* **Primary Expressions –** handles the parenthesised expressions, variables, constants and Boolean literals (TRUE, FALSE).

**Visitor Implementation**

The PrettyPrintVisitor class is designed to traverse the parse tree generated by the ANTLR parser and print a human-readable output that shows the steps of the computation. The visitor class only displays the useful rules for the nodes that are visited. Without this, the output is hard to make sense of as the tree is traversed. It is a custom implementation of the base visitor, overriding the visit rule method to maintain a symbol table and print an output in the appropriate format.

Indentation provides a visual cue for when the visitor is moving up and down the parse tree. An indent() method is created which uses the variable indentLevel which is changed accordingly when the visitor visits and leaves a node. Each indentation represents to a new level of expression or operation being evaluated.

My symbol table is represented as a hash map which stores String and Integer value pairs, the string representing the variable name and the integer value the value of the variable. This implementation ensures the variables, and their values/states are tracked and can be evaluated. The visitor is responsible for evaluating each type of expression and printing the appropriate details about it for pretty printing as mentioned previously and updates the symbol table.

The printing is initiated in visitProgram(). Here, the statements are iterated through and are recursively visited using the visit() method. After all statements are visited, the symbol table is printed to show the final values of the variables.

**Visit Methods –**

* The visitStatement() method is used to throw an exception if the statement given is not a declaration, assignment or expression, meaning it’s received an invalid input.
* The visitDeclaration() method handles the variable declarations. When a declaration is visit, the variable name and a default value (0) is entered into the symbol table and an appropriate output is given for pretty printing.
* The visitAssignment() method handles the variable assignment. This method will throw a runtime exception if a variable is being assigned a value before it has been declared. If the variable has been declared, it will be in the symbol table already. The value is put into the table and an appropriate output is made.
* The visitExpression ()method is a helper that begins the evaluation process by visiting the logicalOrExpression().
* The visitLogicalOrExpression() handles the logical OR (||) expressions. If there are no OR operators, the visitor moves on to logical AND expressions. If OR operators are found, the method recursively evaluates left and right operands, performing OR operations on them. An output is made here for pretty printing.
* The visitLogicalAndExpression() method is responsible for processing logical AND (&&) expressions. If no AND operators are present, the program will visit the equality expressions method. If AND operators are present, left and right operands are evaluated using a logical AND operation.
* The visitEqualityExpressions() method handles the equality and inequality (== and !=) operations. If no equality operators are present, the visitor visits relational expression method. If they are present, left and right operands are evaluated using an equality or inequality operation.
* The visitRelationalExpression() method evaluates relational expressions (<,>,<=,>=). If none of the previous operators are present, the visitor moves on to visit the additive expression method. Relational operations are performed left and right operands and are evaluated as either 1 or 0 (true or false).
* The visitAdditiveExpression() method evaluates addition and subtraction operators based on the presence of either a + or – operator. If none are present, the visitor moves onto the visit multiplicative expression method. Else, addition or subtraction are performed on left and right operands.
* The visitMultiplicativeExpression() method evaluates multiplication and division operations. If no multiplicative methods are present, the visit moves to the visit unary expression method. If \* or / operators are found, multiplication or division operations are performed on left and right operands respectively.
* The visitUnaryExpression() method evaluates NOT operations on expressions. If no unary operators are found, the program moves to the visit primary expression method.
* Finally, the visitPrimaryExpression() method evaluates primary expressions based on their type. A primary expression can either be a integer, identifier, Boolean true or false, or a parenthesised expression. The correct corresponding value is returned for each type, and if a left parenthesis is present, program visits the parenthesised expression.

**Challenges and Limitations**

Due to the symbol table implementation and the visit method return types, I was unable to return Boolean expressions are “true” or “false”, instead opting for the integers 1, 0 for true and false respectively. Although in simple application this can make sense, later comparing Boolean values will lead to issues such as where a true statement could be checked as true equals true, 1 doesn’t equal true.

Initially, I had tried to implement the pretty printer as an listener, however, I struggled with stack management and retrieving values from the context and eventually opted for my current approach.

A screenshot of a computer

AI-generated content may be incorrect.**Tests and Examples**

Figure - full parse tree for example input.

**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure - example input with successful parse tree pretty print output

A screenshot of a computer

AI-generated content may be incorrect.**A screenshot of a computer program

AI-generated content may be incorrect.**

Figure - successful error catch, undeclared variable.

Figure - successful multiplicative test