CE305 Assignment 2

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.

BOOL

- Can be either ‘True’ or ‘False’.
- Used to denote boolean values.

STRING

- Consists of two speech marks either side of a set of characters.
- Used to denote string values.

MOD

- Can only be a % sign.
- Used to denote the modulo calculation.

ASSIGN_OP

- Consist of either a (=), (+=) or (-=).
- Used to denote an assignment operation.
- (=) sets a variable to the given value.
- (+=) adds a value to the value of the variable.
- (-=) subtracts a value from the value of the variable.

COND_OP

- Consists of either a (===), (>=), (<=), (>), (<), (!==).
- Used to denote conditional operations.
- Used in if, elif, while, and for statements/loops.
- (===) and (!==) have an extra = so that the syntax works, they mean equal to and not equal to.

RETURN_TYPE

- Can be the words string, bool, float or void.
- Used to specify the return type of a function.

FUNCT

- Consists of a set of characters followed by an open bracket.
- Used to denote the name of a function.
- Used in the definition and calling of functions.

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.
- Each opening and closing curly brackets are separate tokens, so “{“ is one token and “}” is another.
- Words for certain commands:
    - def
    - return
    - if/elif/else
    - for
    - in range
    - print
- 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 one of the following:

- Assignment statement
    - Consists of a variable name, an assignment operator and an expression, function call or value.
- Print statement
    - Consists of the word print followed by an expression or string that is encapsulated by brackets.
- Function call
    - Consist of a function name and a set of parameters that are encapsulated by brackets.

- if/elif/else statement
    - Requires a conditional statement in brackets.
    - Holds a body of lines which are encapsulated by curly brackets.
- while loop
    - Requires a conditional statement in brackets.
    - Holds a body of lines which are encapsulated by curly brackets.
- for loop
    - Requires a conditional statement in brackets.
    - Holds a body of lines which are encapsulated by curly brackets.
- function definition
    - Starts with the key word “def”.
    - Can have a set of parameters but must have a return type.
    - Holds a body of lines which are encapsulated by curly brackets.
- return statement
    - Starts with the keyword “return”.
    - Followed by an expression or variable.
    - Must be inside a function definition.

Additionally, each expression is split into one of six EXPRs which each have a different purpose. EXPR1-3 are for multiplication, division, addition, subtraction and modulo operations. EXPR4-5 are for number and variable creation. Finally, EXPR6 is for brackets.

Conditional statements can consist of one of four types. The first condition is an expression compared to another expression. Second is a variable compared to a string. Third is a boolean variable and fourth is the opposite, so **not** the boolean variable.

The syntax trees are made from recursive “prog”s, which then search their children.#

This grammar is a combination of java and python.

This grammar requires the input files to end in a new line.

Implementation

This program gives five input file to choose from at the top of MainParser. To test a certain input file, you must uncomment it’s filename string and comment out the others.

Inside the grammar file (Compiler.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 lines to itself.

- “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.
- “function_def” which allows the definition of functions.
- “function_body” which holds all lines of a functions body
- “line” which can be any line in a given file (assignment statements, while loops etc.)
- “ifelse” which holds all types of if else statements, so just an if, an if and else, or an if, elif and an else.
- “if”, “elif” and “else” which are used by the previous rule.
- “while” which allows while loops.
- “for” which allows for loops.
- “condition” which holds the conditions of a loop.
- “print” for the use of print statements.
- “function_call” which denotes a call to a defined function.

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.

PythonVisitor

This is a class that extends ExprLangBaseVisitor with type string. It is used to convert the input file to a set of strings that are in the format of Python code. 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(). This also happens in the “line” rule and the overwritten visitLine(), the context is passed to the correct place. Each visit function for a rule that appears in the grammar file below prog returns the Python version 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 written to a Python source code file later in MainParser. The strings that are return are indented correctly using a variable called “indent”, the “\t” string is repeated based on the number that “indent” currently equals.

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 PythonVisitor 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. Now retrieve the strings ArrayLists from the PythonVisitor class and write them to a Python source code file.

Variable

This class holds the name, datatype and scope of a variable. It is used to hold variables in an ArrayList in PythonVisitor, to check that a variable hasn’t been referenced before assignment.

Function

Similarly, this class holds the name, return type and parameters of a function. It is used to hold functions in an ArrayList in PythonVisitor, to check that a function hasn’t been referenced before definition.

Extended features

I have implemented error handling in PythonVisitor by passing an error message back through the traversal, that interrupts the program and reports the error to the user.

I have implemented type checking in PythonVisitor with my Variable and Function classes, these allow the checking of undefined variables and incorrect datatypes in certain scenarios. Also, the Variable class allows the declaration of local variables which cannot be accessed outside their scope and are destroyed when no longer needed. Furthermore, the grammar allows the use of floating point numbers, with the NUM token and the translation to Python code.

I have implemented functions with rules such as “function_def”, “function_body” and “function_call”, as well as tokens such as FUNCT. Furthermore, my class Function allows functions to be stored in an arraylist for usage checking.