Programming Languages

2021315385

이건

Programming Assignment 2 Report

This report will discuss the detailed explanation of the code design of a parse tree based arithmetic expression calculator. The program is mainly divided into 4 parts, the tokenizer, the parser with validation, the calculator, and file I/O handler.

First, the tokenizer splits the input string into numbers, operators and parentheses and saved into a list. So, this function starts with an empty list called ‘tokens’ and an empty string called ‘number’. The reason for the empty string is to consider multi digit inputs such as 145. Once the variables are declared, a for loop will be used to iterate through each character in the expression. The first if will check for a numerical value and continuously add to the ‘number’ string. Once a non-numerical value is read, the tokenizer will check if the ‘number’ string is empty. If ‘number’ string is not empty, the string will be appended to the list. The append() function is used instead of the add() function to consider possible repetitions, which should all be added to the list, and not ignored. For other non-numerical values, the tokenizer will append all operators and parentheses and ignore whitespace. Any other characters will also be appended separately in the ‘tokens’ list as the validation will occur in the parser. Before the tokenizer returns the list, the tokenizer will check the ‘number’ string one last time. If this string is not empty, the string will be appended to the list, then the tokenizer will return the ‘tokens’ list.

The parser analyzes the tokens list and validates the expression using a recursive descent parsing implementation, checking for grammatical accuracy and raising errors in case of invalid expressions, then saves the parsing tree in a tuple. The parser is divided into 5 functions, the main parser function and 4 functions that each define an EBNF expression. The parser() function receives the tokens list, calls for the parse_expr() function that begins the recursive descent parsing and returns the parse tree. The four functions follow the EBNF grammar in the following table.

EBNF grammar
parse_expr → parse_term { (+ | -) parse_term }
parse_term → parse_power { (* | /) parse_power }
parse_power → parse_factor { (^) parse_power }

parse_factor → - parse_factor | (parse_expr) | integer

The parse_expr() function is the lowest precedence with addition and subtraction. This function will call the parse_term() function onto the current node, then it seeks for a '+' or '-' operator. When the corresponding operators are found, the right node will also call the parse_term() function. Then this operation will be stored as a subtree in the form of a tuple, with the order of operator, left subtree, and right subtree. A while loop is used to consider left associativity within the same precedence. Once the left and right subtrees are built, these subtrees will build the final parse tree.

The parse_term() function is the second lowest precedence with multiplication and division. Parse_term() follows a similar design as parse_expr(), calling the parse_power() function instead for recursion and looking for a '*' or '/' operator instead. Parse_term() also considers left associativity and uses a while loop to tackle this concept. This operation will also be stored as a subtree in the form of a tuple, with the order of operator, left subtree, and right subtree.

The parse_power() function is the second highest precedence with exponents. This function will call the parse_factor() function on the left token and stores the '^' operator if found. Since the exponents are right associative, instead of using a while loop, the parse_power() function will recursively call itself. This operation will also be stored as a subtree in the form of a tuple, with the order of operator, left subtree, and right subtree.

The parse_factor() function is the highest precedence, considering parentheses, unary minus operator, and integer values and checking for the validity of the expression. As this is the highest precedence, the function will first check if the tokens list still has values, raising an error if the index value exceeds the tokens list size. Now the function will get the current token and identify this token. If a unary minus is received, the function will add the 'neg' value as the operator and recursively call itself for the subtree value. If an integer value is received, this value will directly be returned to be used as a leaf node for the parse tree. If a parenthesis is received, the function will call parse_expr() function to analyze the expression inside the parentheses. During this process, the function will check if a closing parenthesis exists, also checking if the index is within the tokens list length to avoid index errors. A value error will also be raised for invalid characters. This operation will also be stored as a subtree in the form of a tuple, with the order of operator, left subtree, and right subtree.

To manage the position within the tokens list, the current index is passed as an argument to each recursive call, and an updated index is returned along with the generated

subtree. The main parser() function initiates the process and performs a final validation check to ensure all tokens were consumed.

The calculator evaluates the parse tree recursively. The calculator() function receives the parse tree as an argument and evaluates its value. The function first checks if the current value is an integer. If it is, the parse tree will be returned for further calculations. If the current value is not an integer, then it is an operator, which will be declared using the ' op' variable. If the op variable is the string 'neg', then it is a unary minus. So, the function will recursively call itself and return the negative value of it. If the op variable is any other operator, then the function will recursively call itself on the left and right subtrees, then perform the operation using the resulting values. During this stage, if the expression notices a division by zero, the function will raise a ZeroDivisionError.

The final file I/O function receives the `expressions. txt` file to read and write the answers on the `result.txt` file. The I/O function, named run_from_file(), will read each line of the expressions. txt and sequentially call the tokenizer(), parser(), and calculator() functions using a for loop. Before that, the strip() function will remove the leading and trailing whitespaces. The expression itself and the results of all the functions will be written onto the 'result.txt' file using the given template. For the validation, since all the validation occurred in the parser, a try...except block will be used on the parser. The corresponding error message will be written onto file using the except block. The only additional error that can be raised is the ZeroDivisionError in the calculator () function, which will write an error message after the parser.

To finalize, the main.py will call the run_from_file() function to read the expressions. txt file, calculate all the expressions, and print the results onto the result.txt file.