Compiler Construction

Assignment 2

Roll no:	200901018
Name:	Muhammad Tayyab
Batch & Section:	BS-CS-01 (B)
Date of Submission:	30/12/2022

Phases of Compiler

Module 1: Implementation of lexical analyzer

A lexical analyser, also known as a lexer or tokenizer, is a program or function that performs lexical analysis. Lexical analysis is the process of breaking down a stream of text into smaller units called tokens, which can be more easily processed by a computer.

The main purpose of a lexical analyser is to recognize and classify the tokens in given input stream. For example, in a programming language, the lexer might recognize keywords, identifiers, operators, and punctuation as different types of tokens.

In Python, you can use the re module to work with regular expressions. This module provides functions for searching and manipulating strings using regular expressions.

In this program, we are using the `re.findall` function, which searches for all characters of a string to locate certain expressions or tokens. Here it’s being used to find all types of characters, constants, operators, parenthesis, or special characters in the given input string.

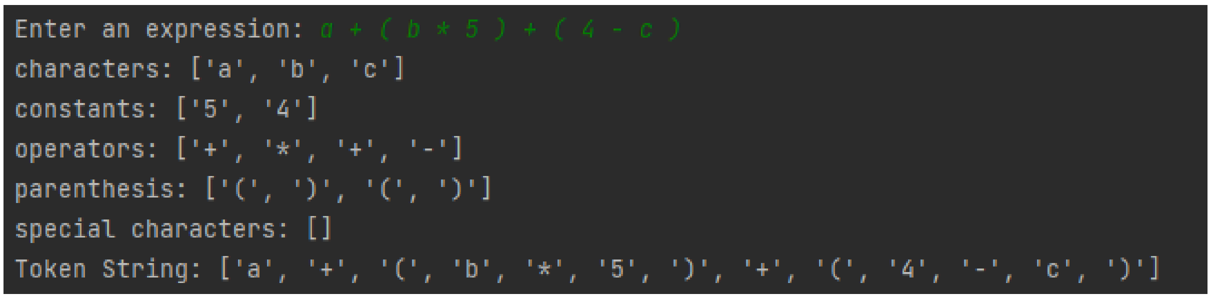
```
def LexicalAnalyser(exp):
    token_list = []

    # show sets of all included characters (excluding whitespaces and
carriage return)
    print(f"characters: {re.findall(r'[a-zA-Z]', exp)}")
    print(f"constants: {re.findall(r'[0-9]', exp)}")
    print(f"operators: {re.findall(r'[=|&^%+*/-]', exp)}")
    print(f"parenthesis: {re.findall(r'[(){}]', exp)}")
    print(f"special characters: {re.findall(r'[,.:;#@$!]', exp)}")

    # Remove Whitespaces and Insert into Token List
    token_list += re.sub('       ', '', exp)
    return token_list
```

This is our lexical Analyser function which inputs a string and generates tokens of the expression along with tokenization of the expression and puts each character in a list.

This is how It works:

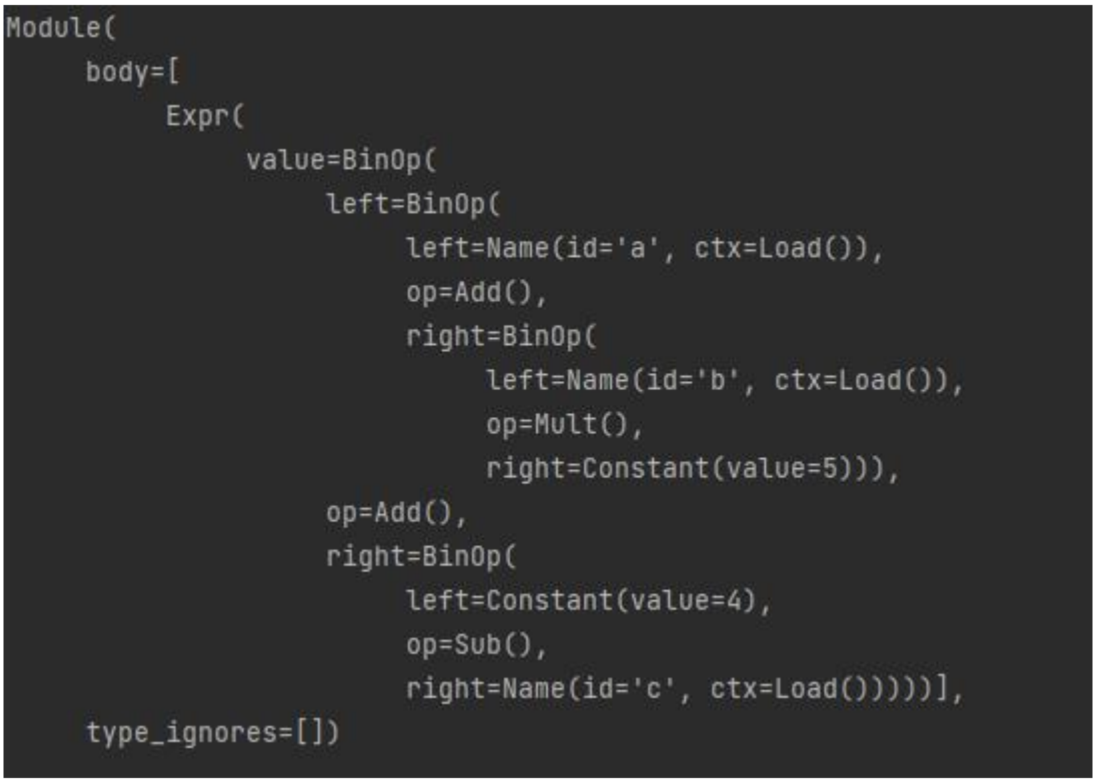


```
Enter an expression: a + ( b * 5 ) + ( 4 - c )
characters: ['a', 'b', 'c']
constants: ['5', '4']
operators: ['+', '*', '+', '-']
parenthesis: ['(', ')', '(', ')']
special characters: []
Token String: ['a', '+', '(', 'b', '*', '5', ')', '+', '(', '4', '-', 'c', ')']
```

Module 2: Implementation of syntax tree

For Syntax tree we will be using the `ast` library in python, which is extremely useful to interpret a regular expression into a tree. Which is how our interpreter reads the code.

We import the `ast` library in our program and use it to parse and display the syntax tree. This is the parse tree for the same example: `a + ( b * 5 ) + ( 4 - c )`



```
Module(
     body=[
          Expr(
               value=BinOp(
                    left=BinOp(
                         left=Name(id='a', ctx=Load()),
                         op=Add(),
                         right=BinOp(
                              left=Name(id='b', ctx=Load()),
                              op=Mult(),
                              right=Constant(value=5))),
                    op=Add(),
                    right=BinOp(
                         left=Constant(value=4),
                         op=Sub(),
                         right=Name(id='c', ctx=Load()))))],
     type_ignores=[])
```

Code File:

The code file is attached alongside the pdf, it has also been uploaded to github

https://github.com/ProgradeX/Semester-Project-Archive/tree/master/Compiler%20Construction