Institute of Space Technology, Islamabad

BS CS-01

Compiler Construction

Assignment 02

Due Date: Saturday, 2022

Submitted by: Faraz Ahmad Qureshi

Regno: (200901045)

Section: B

Instructor: Reeda Saeed

Module 1:

Lexical Analyzer:

- Tokenization of expression (expression can be i.e., a + (b*c) or 3+ (5*2) digits, alphabets, characters)
- Building regex for the expression
- Output tags/ tokens of the expression (i.e. ['a', '+', '(', 'b', '*', 'c', ')'])

We have implemented the lexical analyzer in Python using the Regular Expression (re) module/library

Regular Expressions allow us to perform operations on python strings quickly and efficiently. There are many types of regular expressions that we can use in python and manipulate strings anyway we want.

The Lexical Analyzer functions as a tokenizer for our input expressions. We can assign a token to each character that our analyzer reads. This way, we can assign each character a specific meaning or functionality. On the basis of this fundamental design concept, we have assigned the following tokens to our Lexical Analyzer:

Identifier: Any character that serves as a variable which can be any of the 26 English alphabets (uppercase and lowercase both)

Digit or Constant: This is the constants or numbers between 0 and 9 that we can enter and our tokenizer will recognize them and assign a specific token to each number.

Operators: We can recognize the four operations: (+, -, *, /)

Example:

We will take an example expression:

“ a + (b*c) “

Lets analyze this expression:

The first character is “a”. The lexical analyzer will read this character and compare it with the available regular expression. As it matches “Identifiers” token, it will be given the Identifier tag.

Then we will ignore the “ “ whitespace character.

Afterwards, we find a ‘+’ sign. This is the Addition Operator. Our operator token will be assigned to this character.

Moving onwards, the rest of the Expression will be assigned proper tokens based on regular expressions. This is the first step of a compiler.

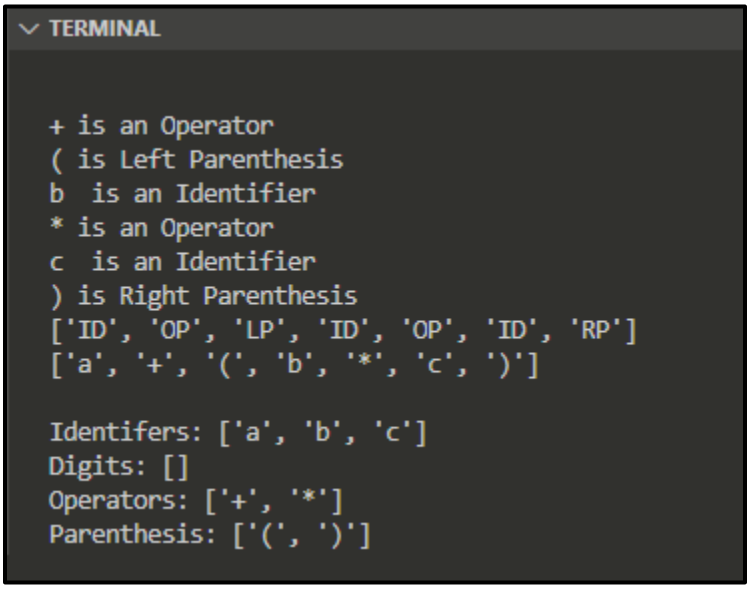
We can better see this process through the code:

Code:

```
# Compiler Construction Assignment
# Faraz Ahmad Qureshi [reg no: 200901045]
# BSCS - 01
import re
import ast
#regular expressions that we will use
token_spec = [r'\d+(\.\d+)?', # identify any no. of digits
              r'[a-zA-z]+',   # identify any identifer(character)
              r'[+\-*/]',     # identify any operator
              r'[()]',        # identify parenthesis
              r'[ \t]+']      # whitespaces,tabs
Identifiers = []
Constants = []
Operators = []
Brackets = []
def lexical_analyzer(text):
    tokens = []

    token_list = ('ID','CONST','OP','LP','RP')
    print("\n[|Lexical Analyzer|]\n")
    for x in text:
        if re.findall(token_spec[0],x):
            print(x," is a Digit ")
            tokens.append(token_list[1])
            Constants.append(x)
        elif re.findall(token_spec[1],x):
            print(x," is an Identifier")
            tokens.append(token_list[0])
            Identifiers.append(x)
        elif re.findall(token_spec[2],x):
            print(x,"is an Operator")
            tokens.append(token_list[2])
            Operators.append(x)
        elif re.findall(token_spec[3],x):
            if x == '(':
                print(x,"is Left Parenthesis")
                tokens.append(token_list[3])
                Brackets.append(x)
            elif x == ')':
                print(x,"is Right Parenthesis")
                tokens.append(token_list[4])
                Brackets.append(x)
        elif re.findall(token_spec[4],x):
            print(x,"Whitespace")
        else:
            print("Erorr! ", x + " is not part of language")
    print(tokens)
input_choice = input("Press:\n[A] to auto-enter expression\n[B] to type expression yourself: ")
if input_choice == 'A' or input_choice == 'a':
    txt = 'a + (b*c)'
elif input_choice == 'B' or input_choice =='b':
    txt = input("\nPlease enter an expression to be tokenized and evaluated: ")
else: print("\nwrong choice entered...\n")
#txt = "a+(b*c)"
txt = re.sub(r'\s+', '', txt) #simple regex to remove whitespaces
lexical_analyzer(txt)
print(list(txt))
print(f"\nIdentifers: {Identifiers}\nDigits: {Constants}\nOperators: {Operators}\nParenthesis:
{Brackets}\n")
code = ast.parse(txt, mode='eval')   # parse the expression
print("\n[|Expression Tree|]\n")
print(ast.dump(code))                # print the AST
```

Output:



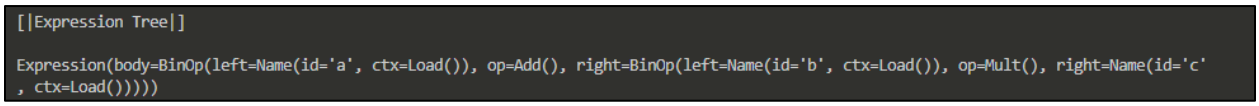
Module 2:

Implementation of syntax tree using AST library of Python

Abstract Syntax Tree (AST) is a powerful module which showcases the exact Abstract Grammar / Syntax that is recognized as our code is run. We can use it to check if our expressions will be understood by Python or not.

Code of AST is same as printed above in the previous module. However, the output will be different this time:

Output:



As we can see, the AST prints a tree like structure where it reads the characters of our expression one node at the time starting from the left and moving towards the right. We can also see how the characters are automatically tokenized and printed accordingly.

This is the second phase / module of compiler construction.