**COMPILER CONSTRUCTION**

**A picture containing graphical user interface

Description automatically generated**

**ASSIGNMENT: 02**

|  |  |
| --- | --- |
| **Name** | M. ZAIN PARACHA |
| **Registration Number** | 200901076 |
| **Batch & Section** | CS (01)-A |
| **Assignment** | COMPILER CONSTRUCTION |
| **Submitted to** | MA’AM TAYYABA ARSHAD |
| **Date** | 25th Dec, 2022 |

**MODULE: 01**

Implementation of 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', ')'] )

**Tokenization:**

The process of tokenizing data involves splitting the text's body. After converting the strings to objects, the data is then stored in a stream of tokens. Each token has its own unique word, number, punctuation sign, and URL.

**Code:**

|  |
| --- |
| import re  string = input("Enter string:")  print("Original string : " + str(string)) #original string  res = [sub.split() for sub in string] #tokenize string  print("After spliting string: " + str(res)) |

**Explanation:**

In the above code its importing through re (regular expression ) taking string from user and then dividing it in tokens splitting the expression in inverted commas ‘’ which shows the expression in tokens.

**Regex:**

Regex means regular expression. It is an efficient tool for matching text based on a pre-defined structure. It can also detect the presence of a particular pattern or its absence, and it can split a pattern into multiple sub-patterns. The standard library of Python provides a re-module that can be used to search for regular expressions. Its main function is to provide a search, which takes a string and a regular expression.

**MODULE: 02**  
Implementation of syntax tree using AST library of python.

**AST:**

It is an abstract syntax tree. It is a tree representation of the abstract syntactic structure of source code written in a programming language.

**CODE:**

|  |
| --- |
| import ast  print ("Abstract syntax tree:")  code=input("Enter expression for AST:")  print("Expression:" +str(code))  code = ast.parse("print ( a + ( b \* c ) )") #create ast  print(ast.dump(code)) #print ast |

**Explanation:**

In this its creates an abstract tree and matches with the expression and shows in which nodes it saves the expression. It takes expression from user then parse and makes its abstract tree.

**FULL CODE:**

|  |
| --- |
| import ast  import re  string = input("Enter string:")  print("Original string : " + str(string)) #original string  res = [sub.split() for sub in string] #tokenize string  print("After spliting string: " + str(res))  print ("Abstract syntax tree:")  code=input("Enter expression for AST:")  print("Expression:" +str(code))  code = ast.parse("print ( a + ( b \* c ) )") #create ast  print(ast.dump(code)) #print ast |