|  |  |
| --- | --- |
|  | Compiler construction  Assignment # 02 |

****

|  |  |
| --- | --- |
|  | Implementation Of Compiler Phases |
|  | Made by: Instructor: Ma’am Reeda |
|  | MUHAMMAD QALB E ALI  200901105 CS-01 B  12/31/22 |

**Module 01:**

Implementation of lexical analyzer in python using ‘re’ library

**Code:**

import re

# List of tokens

tokens = []

# Regular expression patterns for each token

token\_patterns = [

(r'\d+', 'INTEGER'),

(r'\+', 'OPERATOR'),

(r'-', 'OPERATOR'),

(r'\\*', 'OPERATOR'),

(r'/', 'OPERATOR'),

(r'\(', 'PUNCTUATOR'),

(r'\)', 'PUNCTUATOR'),

]

# Tokenize the input code

def tokenize(code):

# Split the code into a list of lines

lines = code.split('\n')

# Iterate over each line

for line in lines:

# Match the line against the token patterns

for pattern, token\_type in token\_patterns:

# Find all occurrences of the pattern in the line

for match in re.finditer(pattern, line):

# Extract the matched string and add it to the list of tokens

tokens.append((token\_type, match.group()))

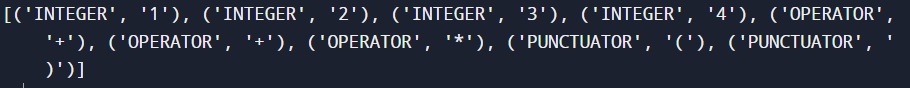
# Test the tokenize function with some sample code

code = "1 + 2 \* (3 + 4 )"

tokenize(code)

print(tokens)

**Output:**



**Code Explanation:**

The code defines a function called **tokenize** that takes in a string of code, **code**, as input and tokenizes it into a list of tokens. A token is a basic unit of information in the source code that can be recognized by the lexical analyzer.

The **tokenize** function uses the **re** (regular expression) library to match the input code against a list of regular expression patterns defined in **token\_patterns**. Each pattern represents a specific type of token, and the function adds a tuple containing the token type and the matched string to the **tokens** list for each pattern that matches a part of the code.

The **token\_patterns** list is a list of tuples, where each tuple consists of a regular expression pattern and a token type. The regular expression patterns are used to match different types of tokens in the code, such as integers, operators, and punctuators. The token type is a string that represents the category or type of the token, such as **INTEGER**, **OPERATOR**, or **PUNCTUATOR**.

The **tokenize** function first splits the input code into a list of lines and iterates over each line. It then matches each line against the regular expression patterns defined in **token\_patterns** and adds a tuple containing the token type and the matched string to the **tokens** list for each pattern that matches a part of the line. The **re.finditer** function is used to find all occurrences of a pattern in a string, and the **group** method of the **match** object returns the matched string.

Finally, the **tokenize** function is tested with the sample code **"1 + 2 \* (3 + 4 )"**, and the resulting list of tokens is printed to the console using the **print(tokens)** statement.

**Module 02:**

Implementation of syntax tree using AST library of python

**Code:**

import ast

def create\_syntax\_tree(code):

tree = ast.parse(code)

return tree

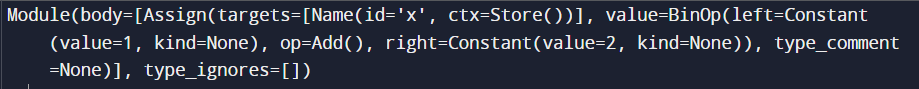
# Test the function with some sample code

code = "x = 1 + 2"

tree = create\_syntax\_tree(code)

print(ast.dump(tree))

**Output:**



**Code Explanation:**

The code defines a function called **create\_syntax\_tree** that takes in a string of code, **code**, as input. The function uses the **ast** (abstract syntax tree) module to parse the **code** and create a tree-like structure that represents the syntax of the code. The tree is then returned by the function.

The **ast.parse** function is used to parse the **code** and create an abstract syntax tree. An abstract syntax tree is a tree-like representation of the source code that abstracts away the details of the code and focuses on the structure and relationships between different elements in the code.

The **ast.dump** function is then used to print a representation of the syntax tree object. This can be useful for debugging or for understanding the structure of the code.

The function is then tested with a sample code **"x = 1 + 2"**, which is a simple assignment statement that assigns the result of the expression **1 + 2** to the variable **x**. The syntax tree for this code would represent the structure of the code and show that it consists of an assignment statement with a left-hand side (**x**) and a right-hand side (**1 + 2**).

The tree is then printed using **ast.dump**. The output of the code should be the syntax tree for the expression **x = 1 + 2**.

My GitHub: <https://github.com/Qalb-E-Ali>