327106 - Complier Construction

Assignment No. 02

Compilation Phases

Name: Hassan Ahmed Khan (200901086)

Dept/Batch: BSCS - 01 Section: B

Lecturer: Mam Reeda Saeed

Date: 30.12.2022

Phases of Compiler

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', ')']*).

Explanation

- **Step 1: Tokenization of Expression**

  The first step in implementing a lexical analyzer is to tokenize the expression. This involves breaking down the string into individual elements such as numbers, symbols, and words.

  For example, the expression *" **3 + (5 * 2)** "* can be broken down into the following tokens:

  [*'**3**', '**+**', '**(**', '**5**', '*****', '**2**', '**)**'*]

- **Step 2: Building Regex for the Expression**

  The next step is to build a regular expression (regex) for the expression. This allows the lexer to identify and match patterns within the string.

  For example, the regex for the expression *" **3 + (5 * 2)** "* might look like this:

  r ' **\ *d+*** | **[*a-z A-Z*] *+*** | **\+** | ***** | **\ (**| **\)** '

  or

  r ' ***d +*** | ***w +*** |***S*** '

  This regex will match any alphabetic character (***a-z*** or ***A-Z***) or any special character (***+*** , ***-*** , ******* , ***/*** , ***(*** or ***)***).

- **Step 3: Tokens of the Expression**

  The final step is to display the output for tokens of the expression. This is done by looping through the expression and matching each element against the regex. If a match is found, the corresponding tag is displayed.

  For example, in the expression *"**3 + (5 * 2)**"* the following tags would be displayed:

  ['*NUMBER*', '*OPERATOR*', '*PAREN_OPEN*', '*NUMBER*', '*OPERATOR*', '*NUMBER*', *'PAREN_CLOSE'*]

Python Source Code

```python
# Importing the re module for regular expressions.
import re


# Function for Tokenization of the expression.
def tokenize(expression):
    # Building the RegEx pattern.
    pattern = r'\d+|[a-zA-Z]+|\+|\*|\(|\)'
    # OR
    # pattern = r'\d+|\w+|\S'
    tokens = re.findall(pattern, expression)
    return tokens


expression = "3 + (5 * 2)"
tokens = tokenize(expression)
print("\nTokens of the expression are: ", tokens)
```

Output

```
Tokens of the expression are:  ['3', '+', '(', '5', '*', '2', ')']

Process finished with exit code 0
```

MODULE 02

Implementation of syntax tree using *AST* library of python.

Explanation

The ***AST*** library of python provides the means to construct an abstract syntax tree of a Python code. This is a useful tool for analyzing, understanding and representing the structure of the code

An *AST* is a tree structure that represents the syntactic structure of a program. It consists of nodes, which can be either terminals or non-terminals. Terminals are symbols that cannot be further expanded, such as identifiers, literals, operators, and keywords. Whereas non-terminals are symbols that can be further expanded, such as functions, classes, and variables. Moreover, it provides a set of *APIs* to construct an abstract syntax tree. The *APIs* can be used to construct a node and then connect it to other nodes in the tree by also providing methods to traverse the tree and perform various operations on it.

The AST library of python is used in many applications such as code analysis, code refactoring, code generation, and static analysis. It is also used to generate code from a given AST.

Python Source Code

```python
# Importing the AST library for syntax tree.
import ast

# Creating a syntax tree.
syntax_tree = ast.parse("print('AST created!')")
print(syntax_tree)

print("The syntax tree is: ")
print(ast.dump(syntax_tree))

# Executing the abstract syntax tree.
exec(compile(syntax_tree, filename="", mode="exec"))
```

Output

```
<ast.Module object at 0x000002439DF18A00>
The syntax tree is:
Module(body=[Expr(value=Call(func=Name(id='print', ctx=Load()), args=[Constant(value='AST created!')], keywords=[]))], type_ignores=[])
AST created!

Process finished with exit code 0
```