COMPILER CONSTRUCTION #03

Particulars:

Name:	Asad Abbas
Reg No:	200901024
Date:	28-12-2022

Code:

```python
import re    # Regex library
import ast   # Abstract Syntax Tree library

# defines regular expressions for different types of tokens
number_regex = r'\d+(\.\d+)?'  # to match integers and floating point
numbers
operator_regex = r'[+-/*()]'   # to match basic arithmetic operators and
parentheses
```

```
identifier_regex = r'[a-zA-Z_][a-zA-Z0-9_]*'  # for matching identifiers

def tokenize(expression):
  # extract the tokens from the input string using regular expressions
  numbers = re.findall(number_regex, expression)
  operators = re.findall(operator_regex, expression)
  identifiers = re.findall(identifier_regex, expression)

  # combine the numbers, operators, and identifiers into a single list of
tokens
  tokens = []
  for number in numbers:
    tokens.append(number)
  for operator in operators:
    tokens.append(operator)
  for identifier in identifiers:
    tokens.append(identifier)

  print(tokens)
  return tokens

# run the tokenizer by entering expression
expression = 'a + (b * c)'
tokens = tokenize(expression)
print(tokens)

tree = ast.parse(expression)   # Parse the input expression by takeing a
string containing a Python expression, and returns an AST object
representing the parsed expression.
print(ast.dump(tree))    # Use this function to print the abstract syntax
tree in a human understandable form

# Using a node visiter class (optional)
class TreeVisitor(ast.NodeVisitor):
  def visit_Name(self, node):
    print(f'Node type: {type(node).__name__}, Node value: {node.id}')
  def visit_BinOp(self, node):
    print(f'Node type: {type(node).__name__}, Node value:
{node.op.__class__.__name__}')

def build_syntax_tree(expression):
  tree = ast.parse(expression)
  visitor = TreeVisitor()
  visitor.visit(tree)

build_syntax_tree(expression)  # Generates Syntax tree of given expression
```

Working of this code:

This is a program that tokenizes an input string and builds a syntax tree for it. It does this in the following steps:

1) In the beginning, we import two libraries**: re** and **ast**. re is the Python library for working with regular expressions, which are a way of specifying patterns in text. ast is the Python library for working with abstract syntax trees, which are a way of representing the structure of a piece of code as a tree.
2) Then we define three regular expressions: **number_regex, operator_regex,** and **identifier_regex.** These regular expressions are used to identify numbers, operators, and identifiers in the input string.
3) Next, a function called tokenize is made that takes an expression as input and returns a list of tokens. The function first uses the re library to extract numbers, operators, and identifiers from the input string using the regular expressions defined earlier. It then combines these three lists of tokens into a single list and returns it.
4) Then we defines a string called 'expression' and call the tokenize function on it, assigning the result to the tokens variable.
5) The code uses the **ast** library to parse the expression string and build an abstract syntax tree for it and then prints this syntax tree using the **ast.dump** function.
6) Additionally, a class called TreeVisitor inherits from ast.NodeVisitor. which has two methods: visit_Name and visit_BinOp. These methods are called when the visitor encounters a Name or BinOp node in the

abstract syntax tree, respectively. Each method prints information about the node it is visiting.

7) Lastly, build_syntax_tree takes an expression as input and builds a syntax tree for it. It then creates an instance of the TreeVisitor class and calls its visit method on the syntax tree, which causes the visitor to traverse the tree and print information about each node it encounters.
8) Calling the build_syntax_tree function on the expression string will cause the syntax tree to be built and printed.

GitHub Link:

https://github.com/asadnaqviii/Semester-5--OS---CC