Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work *No. 3*

Formal Languages & Finite Automata

Elaborated by:
Gorcea Alexandrina, FAF-223
Verified by:
Dumitru Crețu, univ. assist.

Chișinău, 2024

Theoretical considerations:

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages.

The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

Objectives:

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Implementation description:

The Python lexer provided here is tailored specifically for the Python programming language. It comprises a single class named "Lexer," featuring a constructor and a method named "get_tokens()". The lexer identifies various tokens present in Python code, including:

- Comment: Lines prefixed with '#' denoting comments.
- Keyword: Reserved keywords specified in the main file.
- Identifier: Names assigned to functions, variables, etc.
- Number: Integers and floating-point numbers.
- String: Text enclosed within double quotes ("") or single quotes (').
- Comparison: Comparison operators such as '>', '<', etc.
- Assignment: Assignment operators like '=', '+=', etc.
- Arithmetic: Arithmetic operators like '+', '-'.
- Separator: Symbols used to delimit the aforementioned tokens.
- ASCII value: For unidentified symbols, the token represents their ASCII value.

The constructor helps to build the "Lexer" object by receiving the list of keywords in Python programming language and the Python program to tokenize. After that, we create a "while" loop that would analyze each character from the given program and identify and skip the whitespaces in order to get to the lexemes.

```
while c < len(self.program):
    # Skip whitespaces.
    if self.program[c].isspace():
        c += 1
```

Further, we check if there is a comment token. Thus, we look if the current char is '#'. If it is, we find the index of the newline symbol '/n' at the end of the comment, and take the whole comment as the value of the token. However, the newline symbol might not be there if it is the end of the file, thus, we could just take the whole comment from the '#' char to the end of the file.

```
elif self.program[c] == '#':
    new_line = self.program[c:].find('\n')
    if new_line != -1:
        tokens.append(('COMMENT', self.program[c:c + new_line]))
        c += new_line + 1
    elif new_line == -1:
        tokens.append(('COMMENT', self.program[c:len(self.program)]))
        c = len(self.program)
```

Next, we seek the keyword or the identifier tokens. We do this if the current char of the loop is a letter or the underscore symbol. Taking into account the rules of Python to create an identifier, we build a string of the digits, letters, or underscores, with the first char a letter or an underscore. Then, we check if the built string is in the list of keywords, which would suggest that it belongs to the keyword token, otherwise to the identifier token.

```
elif self.program[c].isalpha() or self.program[c] == '_':
    string = ''
    while c < len(self.program) and (
            self.program[c].isdigit() or self.program[c].isalpha() or self.program[c] == '_'):
        string += self.program[c]
        c += 1
    if string in keywords:
        tokens.append(('KEYWORD', string))
    else:
        tokens.append(('IDENTIFIER', string))
```

If the current char is a digit or a dot and the next char is also a digit then, we identify a number token from the next chars. We also create a boolean variable to not allow more than one dot in a number. If there are more dots combined with digits, then unnecessary dots should be taken as separator tokens.

```
elif self.program[c].isdigit() or (self.program[c] == '.' and self.program[c + 1].isdigit()):
    number, dot = '', False
    while c < len(self.program) and (self.program[c].isdigit() or (self.program[c] == '.' and not dot)):
        if self.program[c] == '.':
            dot = True
        number += self.program[c]
        c += 1
    tokens.append(('NUMBER', number))
```

Then we identify string tokens. If the current char is a quote of either type, we take the whole text between the first quote and the next one of the same type. This string is added with its corresponding quotes in the token value cell of the tuple.

```
    elif self.program[c] == "'" or self.program[c] == '"':
        quote, string = self.program[c], ''
        c += 1
        while c < len(self.program) and self.program[c] != quote:
            string += self.program[c]
            c += 1
        tokens.append(('STRING', quote + string + quote))
        c += 1
```

Next, we extract comparison operator tokens. If the current char and the next char belong to the below first list, or if the current char belongs to the below second list then we found a comparison operator token, which we add to the 'tokens' list accordingly with a tuple.

```
    elif self.program[c:c + 2] in ['==', '!=', '<=', '>=']:
        tokens.append(('COMPARISON', self.program[c:c + 2]))
        c += 2
    elif self.program[c] in ['<', '>']:
        tokens.append(('COMPARISON', self.program[c]))
        c += 1
```

Furthermore, we can identify assignment operator tokens. If the current char and the next char, as an entity, belong to the below first list, or if the current char is '=' then we found an assignment operator token, which we add to the 'tokens' list.

```
elif self.program[c:c + 2] in ['+=', '-=', '*=', '/=']:
    tokens.append(('ASSIGNMENT', self.program[c:c + 2]))
    c += 2
elif self.program[c] in '=':
    tokens.append(('ASSIGNMENT', self.program[c]))
    c += 1
```

Next, we can determine arithmetic operator tokens. If the current char and the next char, as an entity, belong to ['//', '**'], or if the current char is in '+-*/%' then we found an arithmetic operator token, which we add to the 'tokens' list.

```
    elif self.program[c:c + 2] in ['//', '**']:
        tokens.append(('ARITHMETIC', self.program[c:c + 2]))
        c += 2
    elif self.program[c] in '+-*/%':
        tokens.append(('ARITHMETIC', self.program[c]))
        c += 1
```

Also, we can identify separator tokens. If the current char is in '()[]{},.;:', then it is a separator char. Thus, we add it to the 'tokens' list.

```
    elif self.program[c] in '()[]{},.;:':
        tokens.append(('SEPARATOR', self.program[c]))
        c += 1
```

In the end, if the current char is unknown by the lexer, then the token name in the token tuple is going to be the ASCII value, and the lexeme is going to be the unknown char alone.

```
    else:
        tokens.append((ord(self.program[c]), self.program[c]))
        c += 1
```

Also, I use a main file which instantiates the class and tests the methods.

Results:

Input:

```
# A program to calculate the factorial of a number.
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print("Factorial of 5:", result)
```

Output:

```
('COMMENT', '# A program to calculate the factorial of a number.')
('KEYWORD', 'def')
('IDENTIFIER', 'factorial')
('SEPARATOR', '(')
('IDENTIFIER', 'n')
('SEPARATOR', ')')
('SEPARATOR', ':')
('KEYWORD', 'if')
('IDENTIFIER', 'n')
('COMPARISON', '==')
('NUMBER', '0')
('SEPARATOR', ':')
('KEYWORD', 'return')
('NUMBER', '1')
('KEYWORD', 'else')
('SEPARATOR', ':')
('KEYWORD', 'return')
('IDENTIFIER', 'n')
```

```
('ARITHMETIC', '*')
('IDENTIFIER', 'factorial')
('SEPARATOR', '(')
('IDENTIFIER', 'n')
('ARITHMETIC', '-')
('NUMBER', '1')
('SEPARATOR', ')')
('IDENTIFIER', 'result')
('ASSIGNMENT', '=')
('IDENTIFIER', 'factorial')
('SEPARATOR', '(')
('NUMBER', '5')
('SEPARATOR', ')')
('IDENTIFIER', 'print')
('SEPARATOR', '(')
('STRING', '"Factorial of 5:"')
('SEPARATOR', ',')
('IDENTIFIER', 'result')
('SEPARATOR', ')')
```

Conclusion:

In conclusion, the Lexer plays a crucial role in the compilation or interpretation process, acting as the initial step in converting source code into meaningful units for further processing. It extracts tokens from the input sequence and categorizes them based on predefined rules. These tokens are then passed to the parser, which constructs a syntax tree representing the structure of the code.

To develop a Lexer, one must first define the tokens or categories of symbols present in the language. This involves identifying keywords, operators, literals, and other elements specific to the language's syntax. The implementation of a Lexer typically involves iterating through the input sequence character by character and categorizing each character or sequence of characters into the appropriate token type. This process may involve creating cases or conditions for each token category, ensuring accurate tokenization.

Ultimately, the Lexer serves as a fundamental component in the compilation or interpretation pipeline, bridging the gap between raw source code and structured representations that can be further analyzed and executed by the compiler or interpreter.