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

REPORT

Laboratory work No. 3
Discipline: Formal Languages & Finite Automata
Topic: Lexer & Scanner.

Done by: Barbarov Nadejda st.gr., FAF-221

Checked by:
asist.univ. Crețu Dumitru

Chișinău 2024

Theory

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

First of all, I defined the constants representing different types of tokens such as integers, operators, and parentheses. These constants serve as identifiers throughout the lexer to categorize the tokens. Each constant corresponds to a specific token type, such as INT for integers, PLUS for the addition operator, and LPAREN for the left parenthesis.

```
# Define token types
TOKEN_INT = 'INT'
TOKEN_PLUS = 'PLUS'
TOKEN_MINUS = 'MINUS'
TOKEN_MULTIPLY = 'MULTIPLY'
TOKEN_DIVIDE = 'DIVIDE'
TOKEN_LPAREN = 'LPAREN'
TOKEN_RPAREN = 'RPAREN'
```

Figure 1. Token definitions

Following that, I defined a utility function named **is_digit** to determine whether a given character is a digit. It checks if the character falls within the range of '0' to '9', indicating that it is a digit.

```
def is_digit(char):
    return '0' <= char <= '9'
```

Figure 2. Function for checking integers

Next, I defined the **Token** class, which represents individual tokens in the input expression. Each token has two attributes: a type and a value. The type indicates the category of the token (e.g., INT, PLUS, MINUS), while the value holds the actual content of the token (e.g., the integer value, operator symbol). Additionally, the **__str__** method is implemented to provide a human-readable representation of the token.

```
class Token:
    Nadejda Barbarov
    def __init__(self, type, value):
        self.type = type
        self.value = value

    Nadejda Barbarov
    def __str__(self):
        return f'Token({self.type}, {self.value})'
```

Figure 3. Token class

Moving on to the **Lexer** class, which is responsible for tokenizing the input text. Upon initialization, the **Lexer** class receives the input text and initializes the current position within the text. The core functionality lies in the **get_next_token** method, which iterates through the input text character by character.

```
class Lexer:
    Nadejda Barbarov
    def __init__(self, text):
        self.text = text
        self.pos = 0

    1 usage  Nadejda Barbarov
    def error(self):
        raise Exception('Invalid character')

    1 usage  Nadejda Barbarov
    def get_next_token(self):
        while self.pos < len(self.text):
            current_char = self.text[self.pos]
```

Figure 4. Lexer class

Within the **get_next_token** method, several logical steps are executed. Firstly, the method skips any whitespace characters encountered, ensuring that they are not considered as tokens. It then proceeds to examine the current character to determine its token type.

```
while self.pos < len(self.text):
    current_char = self.text[self.pos]

    if current_char.isspace():
        self.pos += 1
        continue
```

Figure 5. Iterating characters

If the current character is a digit, the **lexer** identifies and extracts the entire integer token by advancing through consecutive digits. The integer token is then returned with its appropriate type (i.e., TOKEN_INT).

```
if is_digit(current_char):
    start_pos = self.pos
    while self.pos < len(self.text) and is_digit(self.text[self.pos]):
        self.pos += 1
    return Token(TOKEN_INT, self.text[start_pos:self.pos])
```

Figure 6. Checks for digit

Similarly, if the current character is an operator or a parenthesis, such as +, -, *, /, (, or), the **lexer** generates a token corresponding to that operator or parenthesis and advances its position accordingly.

In the case where the current character does not match any recognized token type, an error is raised to indicate an invalid character in the input expression.

```
    else:
        self.error()

return Token( type: None,  value: None)
```

Figure 7. Error raising

Finally, the **main** function orchestrates the process by prompting the user to enter an expression. It then creates an instance of the **Lexer** class with the input text and iterates through the tokens generated by calling the **get_next_token** method. The tokens are stored in a list for further processing, and ultimately, the list of tokens is printed to the console, providing a lexical analysis of the input expression.

```python
def main():
    text = input("Enter an expression: ")
    lexer = Lexer(text)

    tokens = []
    while True:
        token = lexer.get_next_token()
        if token.type is None:
            break
        tokens.append(token)

    print("Tokens:")
    for token in tokens:
        print(token)


if __name__ == "__main__":
    main()
```

Figure 8. Main function

Conclusions / Screenshots / Results

In conclusion, this laboratory work has allowed me to delve into the design and implementation of a lexer for tokenizing arithmetic expressions without relying on external libraries. By defining token types, utility functions, and classes such as Token and Lexer, I was able to effectively parse input expressions into individual tokens representing integers, operators, and parentheses. Through manual iteration and logical decision-making within the lexer, I achieved efficient and accurate tokenization of input text. This exercise provided me with valuable insights into lexical analysis techniques and allowed me to grasp the fundamental steps involved in building a lexer from scratch. Overall, this laboratory work has enhanced my understanding of the underlying principles of lexical analysis and laid a solid foundation for further exploration into compiler construction and language processing tasks.

```
Enter an expression: 1+3*7-(6-2/2)
Tokens:
Token(INT, 1)
Token(PLUS, +)
Token(INT, 3)
Token(MULTIPLY, *)
Token(INT, 7)
Token(MINUS, -)
Token(LPAREN, ()
Token(INT, 6)
Token(MINUS, -)
Token(INT, 2)
Token(DIVIDE, /)
Token(INT, 2)
Token(RPAREN, ))
```

Figure 9. End results

References

1. **Llvm.** https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html
2. **Github.**https://github.com/filpatterson/DSL_laboratory_works/blob/master/2_FiniteAutomata/task.md
3. **Github.** https://github.com/nadea-b/LFA