**Ministerul Educaţiei și Cercetării al Republicii Moldova**
**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

# Course: Formal Languages & Finite Automata

# Laboratory work 3: Lexer & Scanner.

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

# Theory

## Lexical Analysis

Lexical tokenization is conversion of a text into (semantically or syntactically) meaningful lexical tokens belonging to categories defined by a "lexer" program. In case of a natural language, those categories include nouns, verbs, adjectives, punctuations etc. In case of a programming language, the categories include identifiers, operators, grouping symbols and data types.

Lexical Analysis is the first phase of the compiler, also known as a scanner. It converts the High-level input program into a sequence of Tokens:

1. Lexical Analysis can be implemented with the Deterministic finite Automata.
2. The output is a sequence of tokens that is sent to the parser for syntax analysis.

## Tokens

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

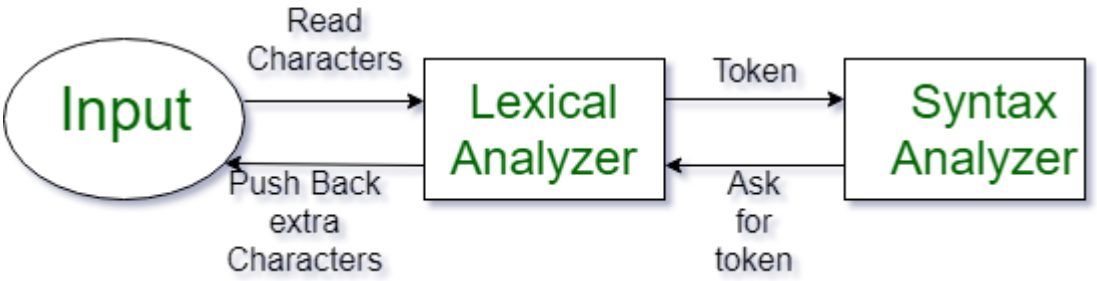
- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Example of non-tokens:

- Comments
- Preprocessor directive
- Macros
- Blanks, tabs, newline, etc.

**Lexical Analysis Steps:**

1. Input preprocessing – This stage involves cleaning up the input text and preparing it for lexical analysis (i.e., comments, whitespaces, etc.).
2. Tokenization - This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
3. Token classification - In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
4. Token validation - In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
5. Output generation - In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.




## Objectives

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

# Implementation

Like the previous laboratory works, I used Python for implementing the lab. In order to make a lexer, I follow a tutorial on YouTube. To start off, we define a set of tokens, that are used in order to analyze the input text for patterns.

```
#########################
#        Tokens         #
#########################


TT_INT = 'INT'
TT_FLOAT = 'FLOAT'
TT_PLUS = 'PLUS'
TT_MINUS = 'MINUS'
TT_MULT = 'MULT'
TT_DIV = 'DIV'
TT_LPAREN = 'LPAREN'
TT_RPAREN = 'RPAREN'
TT_EOF = 'EOF'
```

After that, we defined the Lexer, where we tokenize our input text based on our tokens, finding the patterns in our code. To do this, we create a class, where we track the current character and ensure we don't pass the length of the input text. We also define positions in order to track the current sequences of characters (such as numbers that have a length of 2+).

```
class Lexer:
    def __init__(self, fn, text):
        self.fn = fn
        self.text = text
        self.pos = Position(-1, 0, -1, fn, text)
        self.current_char = None
        self.advance()

    def advance(self):
        self.pos.advance(self.current_char)
        self.current_char = self.text[self.pos.idx] if self.pos.idx < len(self.text) else None
```

Afterwards, we check for tokens in the form of numbers (for which we defined a function to create a number) and the operations defined as numbers, which we use to append to the tokens array, which will be displayed at the end of the execution as the output.

```
elif self.current_char in DIGITS:
    tokens.append(self.make_number())
elif self.current_char == '+':
    tokens.append(Token(TT_PLUS, pos_start=self.pos))
    self.advance()
```

To ensure that the input is correct/valid for our program and grammar, we additionally create a parser class, where we check for any sorts of syntax errors. If any such error is encountered, we output the error to the user's screen. For example, you cannot have 2 '+' signs following one another, meaning this would represent an invalid expression and call the error function.

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.tok_idx = -1
        self.advance()

    def advance(self):
        self.tok_idx += 1
        if self.tok_idx < len(self.tokens):
            self.current_tok = self.tokens[self.tok_idx]
        return self.current_tok
```

# Results

Input:

```
lexer = run("test", "5 + 3 * 20 + 4 + 3 * 3 * 2 * 1")
print(lexer)
```

Output function:

```
def run(fn, text):
    lexer = Lexer(fn, text)
    tokens, error = lexer.make_tokens()

    if error:
        return None, error.as_string()

    parser = Parser(tokens)
    ast = parser.parse()

    return ast.node, ast.error
```

Output:

```
((((INT:5, PLUS, (INT:3, MULT, INT:20)), PLUS, INT:4), PLUS, (((INT:3, MULT, INT:3), MULT, INT:2), MULT, INT:1)), None)
```

# Conclusions

This laboratory work has taught me a lot about how lexers and parsers work. Although I've recently studied about them for my PBL/ELSD project, I haven't fully understood what their purpose was until I made one of my own from scratch. Lexers, serving as tokenizers, parse text in order to generate a list of tokens, comparing them to predefined tokens to ensure that there are no invalid characters. Parsers, on the other hand, ensure that all of the tokens respect the syntax rules of the grammar, such that no tokens would follow an invalid pattern.

I've learnt a lot more about how abstract languages are defined, and how programming languages operate on a lower level. Additionally, following a tutorial, along with my own research gave me some very interesting info about how parsers interact with the tokens, as well as how errors are generated whenever a syntax error is encountered. Lastly, I've also developed my Python skills, learning how to bundle functions together to write more efficient code, saving time and making it look cleaner.

To conclude, this laboratory work has taught me a lot about lexers and parsers, and about how similar finite automatas are to lexers and parsers.

# References:


1. Introduction to lexical analysis. [online]. [accessed 17.03.2024]. Available: https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
2. Lexical Analysis. [online]. [accessed 17.03.2024]. Available: https://en.wikipedia.org/wiki/Lexical_analysis
3. Making your own Programming Language. [online]. [accessed 17.03.2024]. Available: https://www.youtube.com/watch?v=Eythq9848Fg&list=PLZQftyCk7_SdoVexSmwy_tBgs7P0b97yD&index=1