**Universitatea Tehnică a Moldovei**
**Facultatea Calculatoare, Informatică și Microelectronică**
**Departamentul Inginerie software și automatică**

# Laboratory work 3:
# Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-232 Istrati Ștefan

Verified:
asist. univ. Crețu Dumitru

Chişinău - 2025

**Theory:**

A **lexer**, also known as a **lexical analyzer**, is a fundamental component in the process of interpreting or compiling source code. It is responsible for breaking down a sequence of characters (source code) into a sequence of tokens, which are the smallest units of meaning in the language. Tokens are used by the parser to understand the structure of the code. The lexer's main function is to simplify the complexity of the raw source code by transforming it into a set of identifiable units that can be processed further.

**Key Concepts**

1. **Lexical Analysis**: This is the first phase of a compiler or interpreter, where the raw input is divided into tokens. The lexer reads the input character by character and groups them into meaningful sequences. These sequences, or tokens, represent keywords, operators, identifiers, literals, punctuation marks, etc.
2. **Tokens**: Tokens are the output of the lexer and are classified into categories, such as:
    - **Keywords**: Reserved words with special meaning (e.g., if, while, return).
    - **Identifiers**: Names of variables, functions, classes, etc.
    - **Literals**: Constants like numbers and strings (e.g., 42, "hello").
    - **Operators**: Symbols representing operations (e.g., +, -, =, ==).
    - **Punctuation**: Symbols that help in separating code components (e.g., parentheses (), semicolons ;).
3. **Regular Expressions**: Lexers often use regular expressions (regex) to identify patterns in the input text. A regular expression is a sequence of characters that defines a search pattern, which the lexer can use to match tokens in the source code.
4. **Finite Automata**: The lexer can be implemented using finite automata, which are mathematical models used to represent and control the transitions between states. A **deterministic finite automaton (DFA)** is used for token recognition, where each input character causes a transition to a new state, ultimately leading to a final state representing a valid token.
5. **Lexical Units**: The lexer typically processes the source code in chunks, ignoring irrelevant parts like whitespace and comments. The relevant units are passed as tokens for further analysis.

**Steps in Lexical Analysis**

1. **Input Reading**: The lexer reads the source code character by character.
2. **Pattern Matching**: It attempts to match parts of the input to predefined patterns (regular expressions) for valid tokens.
3. **Tokenization**: Once a match is found, the lexer groups the characters into a token and sends it to the parser.
4. **Error Handling**: If the lexer encounters an invalid sequence of characters that doesn't match any pattern, it generates an error or reports the failure.

**Types of Lexers**

- **Handwritten Lexer**: A lexer can be written manually by defining all the token patterns and the logic for transitioning between states.
- **Automated Lexer**: Tools like **Lex** or **Flex** can automatically generate lexers from regular expressions. These tools generate code that handles the process of reading the input and recognizing tokens.

**Efficiency Considerations**

- **Time Complexity**: The time complexity of a lexer is generally linear with respect to the size of the input, O(n), because each character is processed exactly once.
- **Memory Complexity**: The memory complexity depends on the number of tokens and their respective sizes. If the lexer is implemented using finite automata, memory usage is proportional to the number of states and transitions.

**Role of the Lexer in the Compiler**

The lexer plays a crucial role in the compiler or interpreter pipeline:

1. **Preprocessing**: It simplifies the input by removing unnecessary details like whitespace and comments, focusing only on the meaningful parts.
2. **Error Detection**: It can catch basic syntax errors related to token formation, such as invalid characters or incorrect token patterns.
3. **Tokenization**: It converts the source code into a more manageable form (tokens) that the parser can work with, facilitating the next phase of parsing the syntax.

**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.

**Note:** Just because too many students were showing me the same idea of lexer for a calculator, I've decided to specify requirements for such case. Try to make it at least a little more complex. Like, being able to pass integers and floats, also to be able to perform trigonometric operations (cos and sin).

**But it does not mean that you need to do the calculator, you can pick anything interesting you want**

**Implementation**:

1. Token Types and Keywords:

TOKEN_TYPES dictionary: This dictionary maps symbolic names (like 'NUMBER', 'PLUS', etc.) to their corresponding token representations. These types correspond to different kinds of syntactic units found in the code (e.g., operators, parentheses, keywords).

```
TOKEN_TYPES = {
    'NUMBER': 'NUMBER',
    'IDENTIFIER': 'IDENTIFIER',
    'PLUS': '+',
    'MINUS': '-',
    'MULTIPLY': '*',
    'DIVIDE': '/',
    'LPAREN': '(',
    'RPAREN': ')',
    'EQUAL': '=',
    'LESS': '<',
    'LESSEQUAL': '<=',
    'GREATER': '>',
    'GREATEREQUAL': '>=',
    'FOR': 'FOR',
    'WHILE': 'WHILE',
    'SIN': 'SIN',
    'COS': 'COS',
    'TAN': 'TAN',
    'COT': 'COT',
    'LBRACE': '{',
    'RBRACE': '}',
    'SEMICOLON': ';'
}
```

KEYWORDS dictionary: This dictionary defines the reserved keywords (like "for", "while", "sin", etc.) in the language being lexically analyzed. Keywords are mapped to specific token types.

```
KEYWORDS = {
    "for": "FOR",
    "while": "WHILE",
    "sin": "SIN",
    "cos": "COS",
    "tan": "TAN",
    "cot": "COT"
}
```

2. The Token Class:

Token class: This represents a token with a type_ (the token's category, like NUMBER, PLUS, etc.) and an optional value (the specific value of the token, e.g., the actual number or the identifier's name).

The __repr__ method ensures that tokens are printed in a readable format, like Token(NUMBER, 3).

```
class Token:
    def __init__(self, type_, value=None):
        self.type = type_
        self.value = value

    def __repr__(self):
        return f"Token({self.type}, {repr(self.value)})"
```

3. The Lexer Class:

The Lexer class is the core of the lexing process, responsible for scanning the input text and generating tokens.

Initialization (__init__): The Lexer is initialized with a string of code (text). It starts at the first character (position = 0) and keeps track of the current character (current_char).

```
def __init__(self, text):
    self.text = text
    self.position = 0
    self.current_char = self.text[self.position] if self.text else None
```

- advance method: Moves to the next character in the input text and updates the current_char.

```
def advance(self):
    """Move to the next character."""
    self.position += 1
    self.current_char = self.text[self.position] if self.position < len(self.text) else None
```

- peek method: Returns the character following the current one without advancing the cursor. This is useful for lookahead in the lexing process.

```
def peek(self):
    """Look at the next character without advancing."""
    next_pos = self.position + 1
    return self.text[next_pos] if next_pos < len(self.text) else None
```

- skip_whitespace method: Skips over any spaces or tabs, as whitespace is generally not part of the syntax of the language.

```
def skip_whitespace(self):
    """Skip spaces and tabs."""
    while self.current_char is not None and self.current_char in ' \t':
        self.advance()
```

- get_number method: This method extracts numbers from the input string. It handles both integers and floating-point numbers. If a period (.) is encountered, it treats the number as a float.

```
def get_number(self):
    """Extracts a number (integer or float)."""
    num_str = ''
    while self.current_char is not None and self.current_char.isdigit():
        num_str += self.current_char
        self.advance()

    if self.current_char == '.':
        num_str += self.current_char
        self.advance()
        while self.current_char is not None and self.current_char.isdigit():
            num_str += self.current_char
            self.advance()

    return Token(TOKEN_TYPES['NUMBER'], float(num_str) if '.' in num_str else int(num_str))
```

- get_identifier method: This method processes identifiers (variable names and keywords). It recognizes both alphanumeric characters and underscores as valid parts of an identifier. If the identifier matches a keyword, it returns the corresponding keyword token; otherwise, it returns a general identifier token.

```
def get_identifier(self):
    """Extracts an identifier (variable name or keyword)."""
    ident_str = ''
    while self.current_char is not None and (self.current_char.isalnum() or self.current_char == '_'):
        ident_str += self.current_char
        self.advance()

    token_type = KEYWORDS.get(ident_str, 'IDENTIFIER')
    return Token(token_type, ident_str)
```

- get_next_token method: This method is responsible for extracting the next token from the input string. It checks the current character and determines whether it is a digit, an alphabetic character, or a known operator or symbol (like +, =, (, etc.). Depending on the character, it delegates the work to the appropriate method (e.g., get_number, get_identifier, or simply creating a token for operators and symbols).

- tokenize method: This method repeatedly calls get_next_token to extract all tokens from the input text until there are no more tokens. The resulting list of tokens is returned.

```
def tokenize(self):
    """Tokenizes the entire input string."""
    tokens = []
    while (token := self.get_next_token()) is not None:
        tokens.append(token)
    return tokens
```

4. Sample Code:

The sample code provided simulates a small piece of code that will be lexically analyzed. The code includes common language constructs like:x

A for loop

Mathematical function calls (sin, cos, tan, cot)

An if statement with comparison operators

```
code = "for (i = 0; i <= 10; i = i + 1) { sin(i); cos(i); tan(i); cot(i); if (x > y) { x = y; } }"
lexer = Lexer(code)
tokens = lexer.tokenize()
```

The output of the execution of this code :

```
Token(FOR, 'for')
Token(LPAREN, '(')
Token(IDENTIFIER, 'i')
Token(EQUAL, '=')
Token(NUMBER, 0)
Token(SEMICOLON, ';')
Token(IDENTIFIER, 'i')
Token(LESSEQUAL, '<=')
Token(NUMBER, 10)
Token(SEMICOLON, ';')
Token(IDENTIFIER, 'i')
Token(EQUAL, '=')
Token(IDENTIFIER, 'i')
Token(PLUS, '+')
Token(NUMBER, 1)
Token(RPAREN, ')')
Token(LBRACE, '{')
Token(SIN, 'sin')
Token(LPAREN, '(')
Token(IDENTIFIER, 'i')
Token(RPAREN, ')')
Token(SEMICOLON, ';')
Token(COS, 'cos')
Token(LPAREN, '(')
Token(IDENTIFIER, 'i')
```

```
Token(RPAREN, ')')
Token(SEMICOLON, ';')
Token(TAN, 'tan')
Token(LPAREN, '(')
Token(IDENTIFIER, 'i')
Token(RPAREN, ')')
Token(SEMICOLON, ';')
Token(COT, 'cot')
Token(LPAREN, '(')
Token(IDENTIFIER, 'i')
Token(RPAREN, ')')
Token(SEMICOLON, ';')
Token(IDENTIFIER, 'if')
Token(LPAREN, '(')
Token(IDENTIFIER, 'x')
Token(GREATER, '>')
Token(IDENTIFIER, 'y')
Token(RPAREN, ')')
Token(LBRACE, '{')
Token(IDENTIFIER, 'x')
Token(EQUAL, '=')
Token(IDENTIFIER, 'y')
Token(SEMICOLON, ';')
Token(RBRACE, '}')
Token(RBRACE, '}')
```

**Conclusion:**

In conclusion, this lexer efficiently breaks down a string of source code into meaningful tokens by identifying various language constructs such as keywords, operators, numbers, and identifiers. It uses a series of methods to process characters, skip whitespace, and handle different token types, including numbers and keywords. The lexer also includes error handling to catch unrecognized characters, ensuring robust analysis. This tokenization process is essential for further parsing or execution in compilers or interpreters. Overall, the lexer serves as a crucial first step in transforming raw code into a structured format that can be understood and processed by the rest of the compiler or interpreter.