

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică**

**Ciumacenco Pavel FAF-232**

**Report**

*Laboratory work nr.3*

***Course: Formal Languages & Finite Automata***

Checked by:

**Cretu Dumitru,** *university prof.*

FCIM, UTM

**Chișinău – 2025**

1. **Theory:**

### Introduction to Lexical Analysis

Lexical analysis is the first stage of a compiler or interpreter, responsible for breaking down a sequence of characters into meaningful components known as tokens. This process is fundamental in understanding programming, markup, and domain-specific languages, as it facilitates subsequent stages of parsing and syntax analysis.

The term "lexer" is derived from "lexical analysis" and is often interchangeably referred to as a tokenizer or scanner. The lexer operates by recognizing patterns in text and categorizing them into predefined classes, thus simplifying language processing.

### Alternative Names and Roles of a Lexer

A lexer is known by several names, including:

* **Tokenizer:** Focuses on splitting text into distinct tokens based on syntax rules.
* **Scanner:** Emphasizes the scanning of the character stream and producing recognized tokens.

Despite these different names, the fundamental role remains the same: converting a raw text stream into structured tokens that can be understood by a parser.

### The Relationship Between Tokens and Lexemes

One of the essential aspects of lexical analysis is distinguishing between **lexemes** and **tokens**:

* **Lexemes**: These are substrings extracted from the input that match a particular pattern. They are the direct output of splitting a string based on rules such as whitespace or punctuation.
* **Tokens**: These provide a category or type to a lexeme. Tokens do not necessarily retain the actual value of a lexeme but store its classification along with relevant metadata.

For example, in the statement:

int x = 10;

* The lexemes might be: "int", "x", "=", "10", ";"
* The corresponding tokens could be:
  + TOKEN\_INT\_DECLAR (for "int")
  + TOKEN\_IDENTIFIER (for "x")
  + TOKEN\_EQUAL (for "=")
  + TOKEN\_INT (for "10")
  + TOKEN\_SEMI (for ";")

Thus, while lexemes are raw substrings, tokens encapsulate their meaning in the language context.

### Inner Workings of a Lexer

A lexer follows a systematic approach to process an input string:

1. **Reading Input:** The lexer reads the raw input as a character stream.
2. **Skipping Unnecessary Elements:** Whitespace and comments are ignored unless they have significance in the language.
3. **Pattern Matching:** The lexer identifies patterns based on predefined rules (such as regular expressions or deterministic finite automata).
4. **Generating Tokens:** Recognized patterns are classified into tokens and passed to the next processing stage.
5. **Error Handling:** Unrecognized sequences result in lexical errors, prompting error messages or corrective mechanisms.

### Methods for Implementing a Lexer

There are various approaches to implementing a lexer, including:

* **Manual Lexers**: Handwritten state-based lexers that use loops and conditionals to categorize tokens.
* **Finite State Machines (FSMs)**: Formalized lexers that transition between predefined states based on input.
* **Regular Expressions (Regex-based Lexers)**: Pattern-matching lexers that use regular expressions to identify token types.
* **Lexer Generators**: Tools such as Lex, Flex, or ANTLR automate lexer creation based on formal grammar definitions.

Each method has its trade-offs in terms of efficiency, complexity, and maintainability.

### Applications of Lexical Analysis

Lexical analysis extends beyond compilers and interpreters. Some practical applications include:

* **Syntax highlighting in text editors**: Differentiating keywords, literals, and comments in code.
* **Static analysis tools**: Checking for errors in source code before execution.
* **Natural language processing (NLP)**: Tokenizing text for linguistic analysis.
* **Security and input validation**: Ensuring structured data input in systems.

### 7. Conclusion

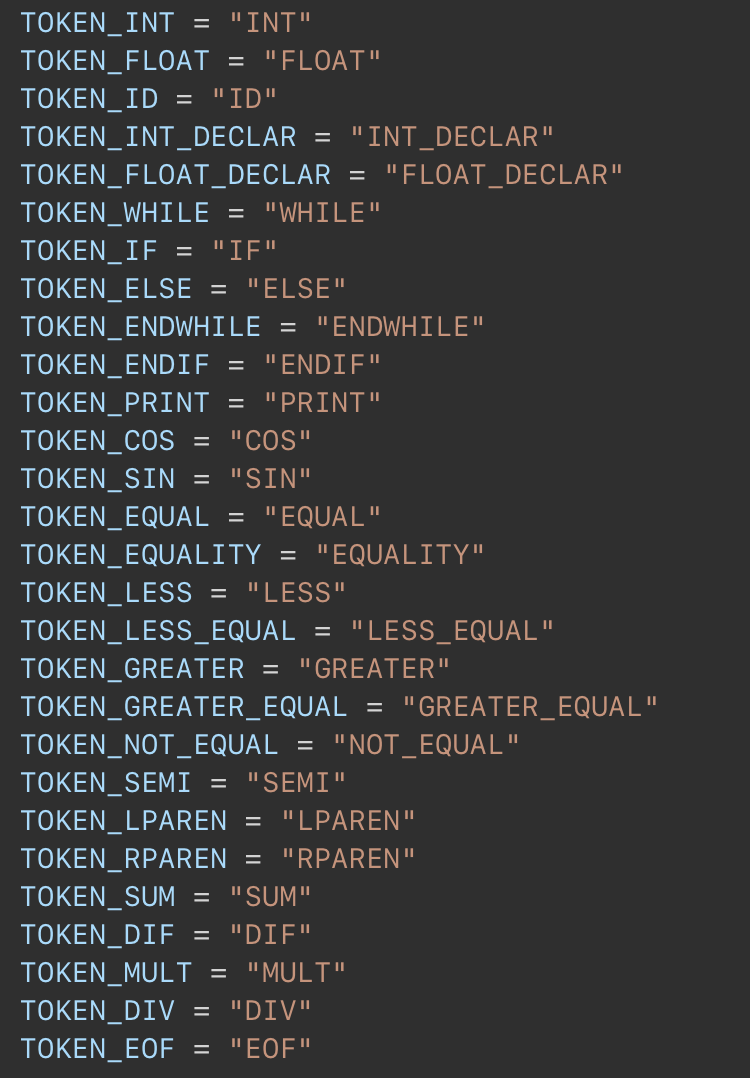
Lexical analysis serves as a bridge between raw text and structured data, facilitating language understanding in both programming and broader computational fields. By implementing lexers, developers can effectively process textual data, enabling further analysis and execution within various applications. Understanding how lexers work provides insights into compiler design, data processing, and even AI-driven text analysis.

1. **Objectives:**

-Understand what lexical analysis [1] is.

-Get familiar with the inner workings of a lexer/scanner/tokenizer.

-Implement a sample lexer and show how it works.

1. **Implementation Description:  
     
     
   **Figure 1: code snippet

**Token Definitions**:The code defines a set of constants representing different token types (e.g., TOKEN\_INT, TOKEN\_FLOAT, TOKEN\_IF, TOKEN\_EQUALITY). These tokens categorize different elements in the input string for the lexer.

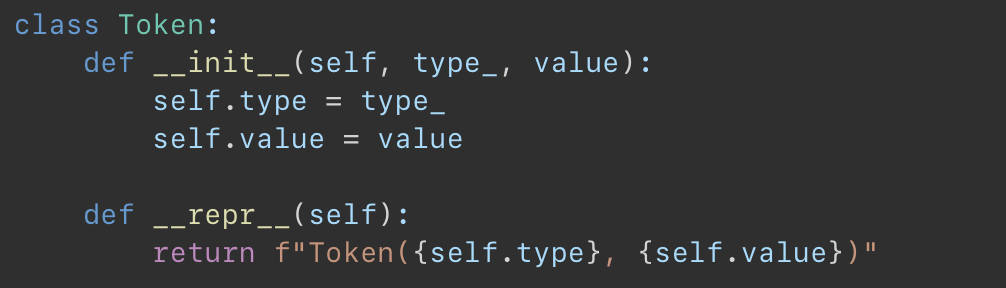
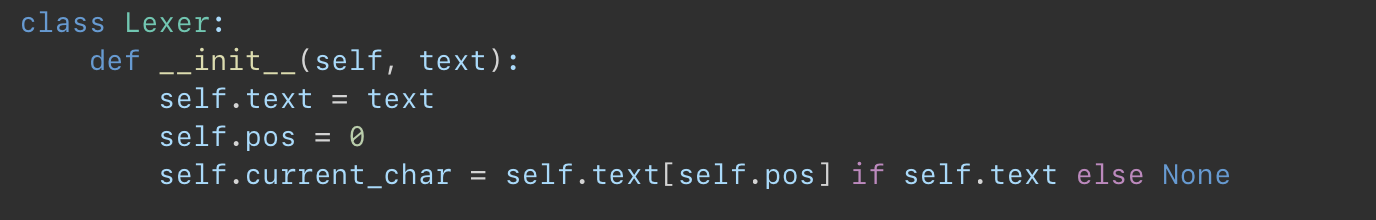
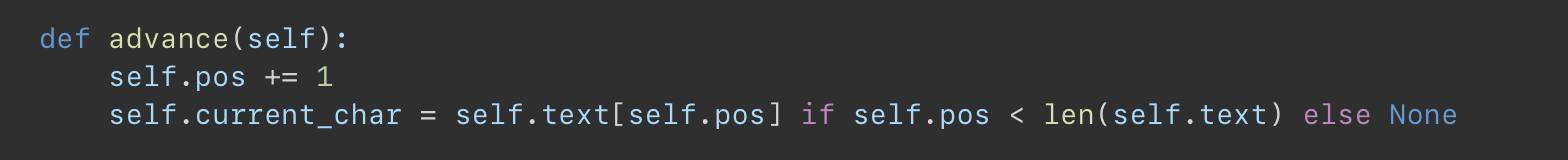
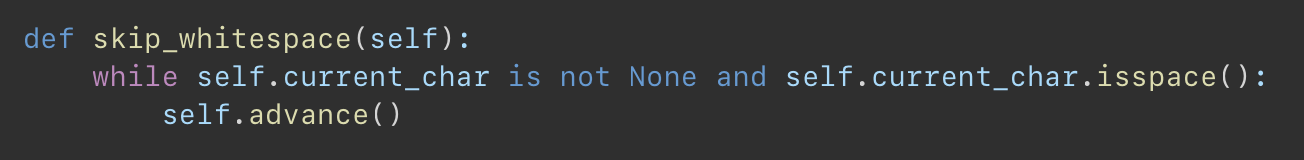
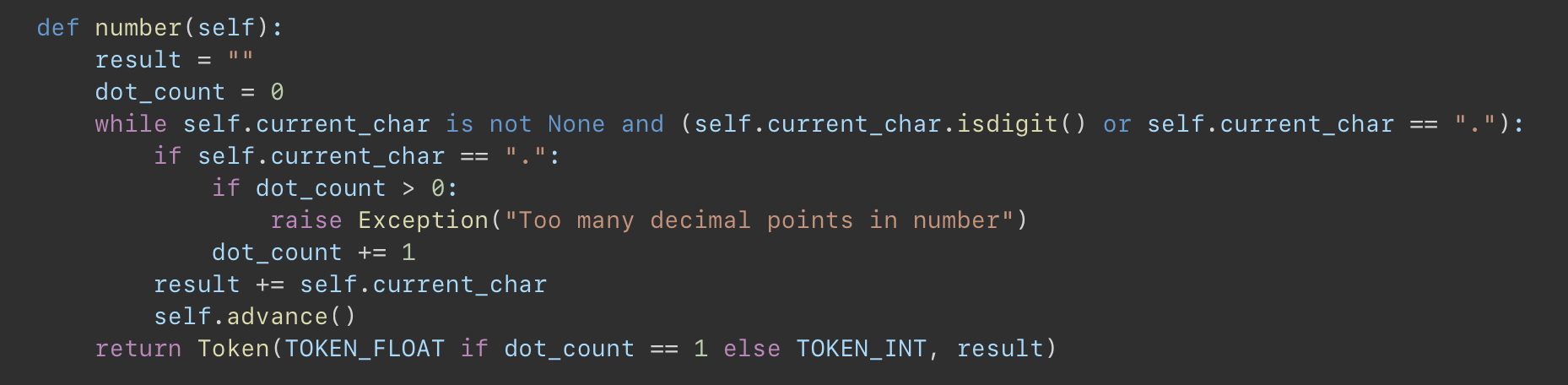


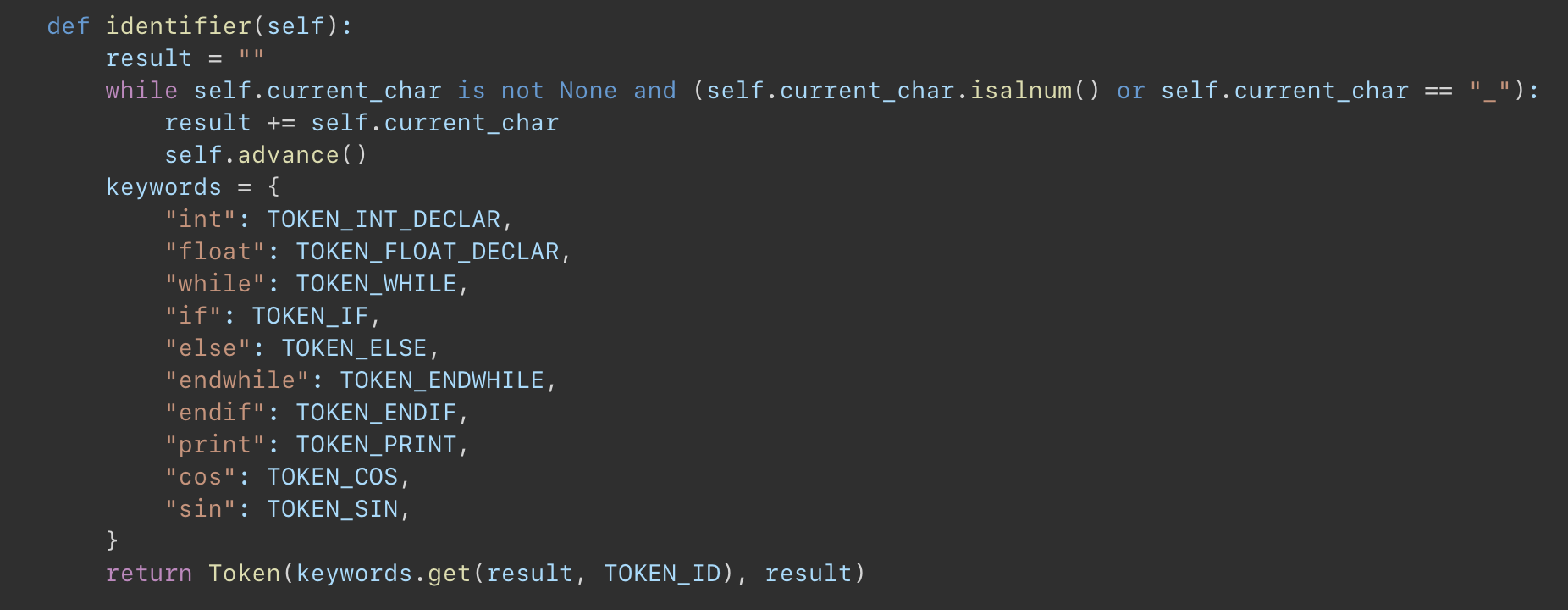
Figure 2: code snippet  
 **Token Class**: The Token class represents individual tokens. Each token has a type (like INT, FLOAT, or ID) and a value (like 10, 3.14, or x). The \_\_repr\_\_ method ensures tokens are printed in a readable format.

Figure 3: code snippet  
 **Lexer Class (Initialization)**: The Lexer class takes a string as input (text) and initializes pos (position in the string) and current\_char (the character currently being processed).

Figure 4: code snippet  
 **Character Advancement (advance method)**: The advance method moves the pos pointer forward and updates current\_char. If the end of the string is reached, current\_char becomes None.

Figure 5: code snippet  
 **Whitespace Handling (skip\_whitespace method)**: This method skips spaces, tabs, and newlines in the input to ensure they don’t interfere with token recognition.

Figure 6: code snippet  
 **Number Tokenization (number method)**: This method constructs integer or floating-point numbers. If more than one decimal point is found, it raises an error.

Figure 7: code snippet  
 This prints whether the given FA is deterministic by calling the is\_deterministic() method. It's a basic diagnostic check to determine if conversion to DFA is needed.

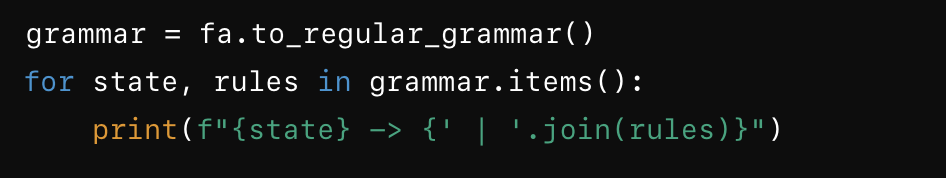
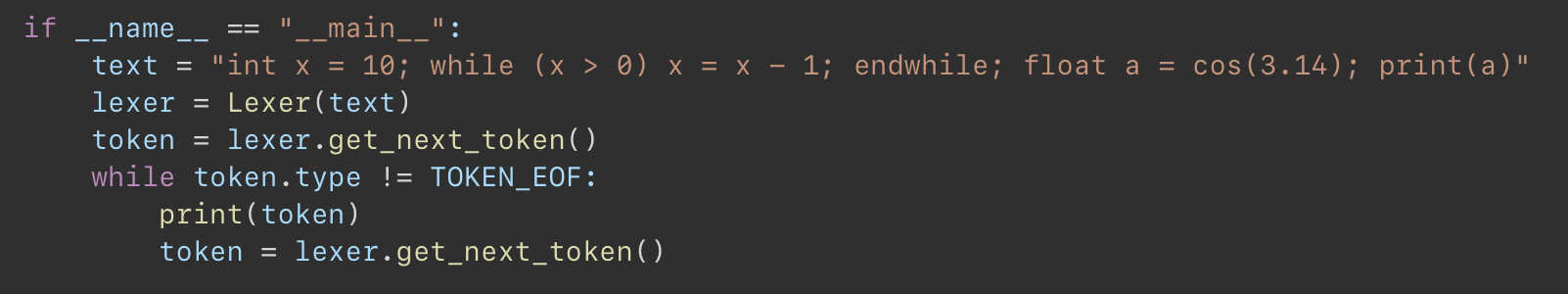
Figure 8: code snippet  
 **Identifier and Keyword Handling (identifier method)**: This method constructs variable names and recognizes keywords like int, float, while, if, and print. If a match is found in the keywords dictionary, it returns the corresponding token type.

Figure 9: code snippet  
 I**Main Tokenization (get\_next\_token method)**:

* It checks for different token types in a loop.
* If the character is a digit, it calls number().
* If it's a letter, it calls identifier().
* It also processes operators (=, ==, <, <=, etc.) and symbols (+, -, \*, /).
* If it finds an unrecognized character, it raises an exception.
* If all input is processed, it returns an EOF token.

Figure 9: code snippet  
**Main Execution (\_\_main\_\_ block)**: A sample string is passed to the lexer. Tokens are extracted one by one using get\_next\_token() and printed until the EOF token is reached. **4.Conclusions.Screenshots.Results.**

The lexer implemented in this code serves as a fundamental component of a language-processing system. By systematically scanning an input string, it categorizes sequences of characters into meaningful tokens such as keywords, identifiers, numbers, operators, and delimiters. This process is crucial in transforming raw text into a structured format that a parser or interpreter can further analyze.

Through methods like number() and identifier(), the lexer efficiently differentiates between numeric values, reserved keywords, and user-defined variables. Additionally, it correctly identifies comparison and arithmetic operators, ensuring that expressions are tokenized accurately. By employing structured error handling, the lexer also prevents malformed tokens from being processed, reinforcing the reliability of the lexical analysis stage.

Ultimately, this lexer lays the groundwork for building a robust compiler or interpreter. It highlights the importance of lexical analysis in language processing and demonstrates how a well-structured tokenization mechanism can facilitate the subsequent phases of syntax analysis and execution.



Figure 10: Results Picture