MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.3

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

1. Theory:

Lexical analysis - a cornerstone in the edifice of computational linguistics and programming language theory, epitomizes the intricate interplay between linguistic abstraction and algorithmic precision. It embodies a multifaceted process wherein raw textual inputs are subjected to meticulous scrutiny and deconstruction, with the overarching goal of delineating discrete units or tokens imbued with semantic significance and syntactic relevance. Nestled at the inception of the compilation or interpretation pipeline, lexical analysis serves as the foundational bedrock upon which subsequent stages of analysis and interpretation are erected. Through a symphony of algorithms and heuristics, lexical analysis navigates the labyrinthine expanse of textual data, deciphering lexical constructs, and attributing contextual attributes, thereby paving the way for the seamless traversal into the realms of syntactic and semantic analysis.

A *Lexer*, standing at the forefront of computational analysis, transcends the realm of mere string manipulation, embodying the essence of linguistic deconstruction and classification. It embodies a sophisticated computational apparatus, meticulously transforming raw textual inputs into a structured sequence of tokens, each imbued with semantic significance and syntactic context. Operating within the intricate landscape of programming languages, data processing, or lexical analysis, the lexer employs a repertoire of algorithms and heuristics to unravel the intricacies of the input, attributing categorical labels and contextual attributes to each token, thereby bestowing upon them a coherent representation within the domain of computational interpretation and manipulation.

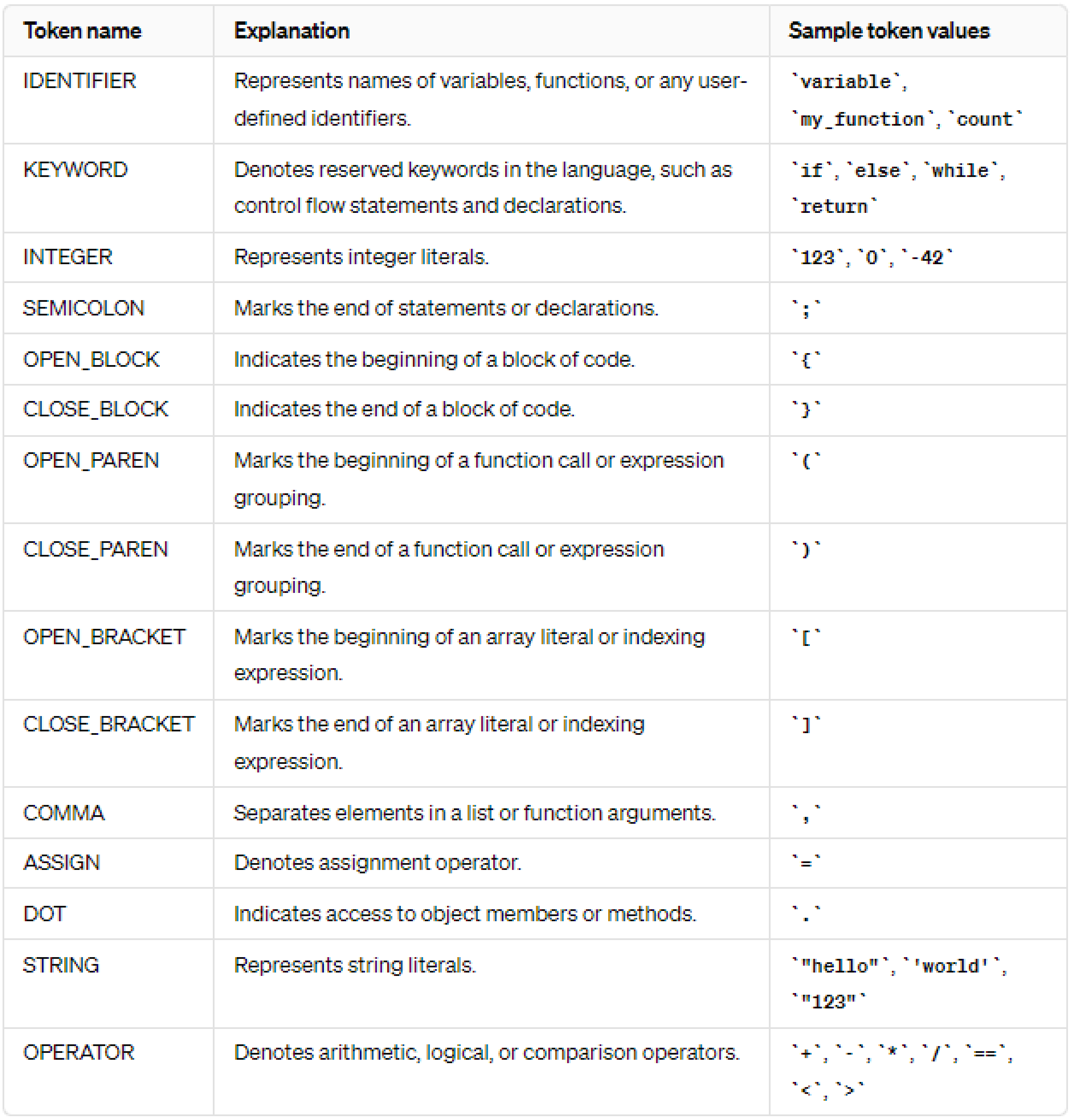
A *Scanner* is a computational entity tasked with the intricate process of parsing input streams or strings, discerning and isolating discrete units or tokens based on predetermined delimiters or patterns. Operating within the realm of computational linguistics or data processing, the scanner meticulously dissects the input, employing sophisticated algorithms to navigate through the intricate tapestry of characters, identifying and delineating substrings that represent meaningful units of information. It is typically employed in diverse domains ranging from compiler design to natural language processing, where the meticulous extraction of tokens serves as the foundational step towards subsequent analysis and interpretation.

2. Purpose of the task work:

- 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.

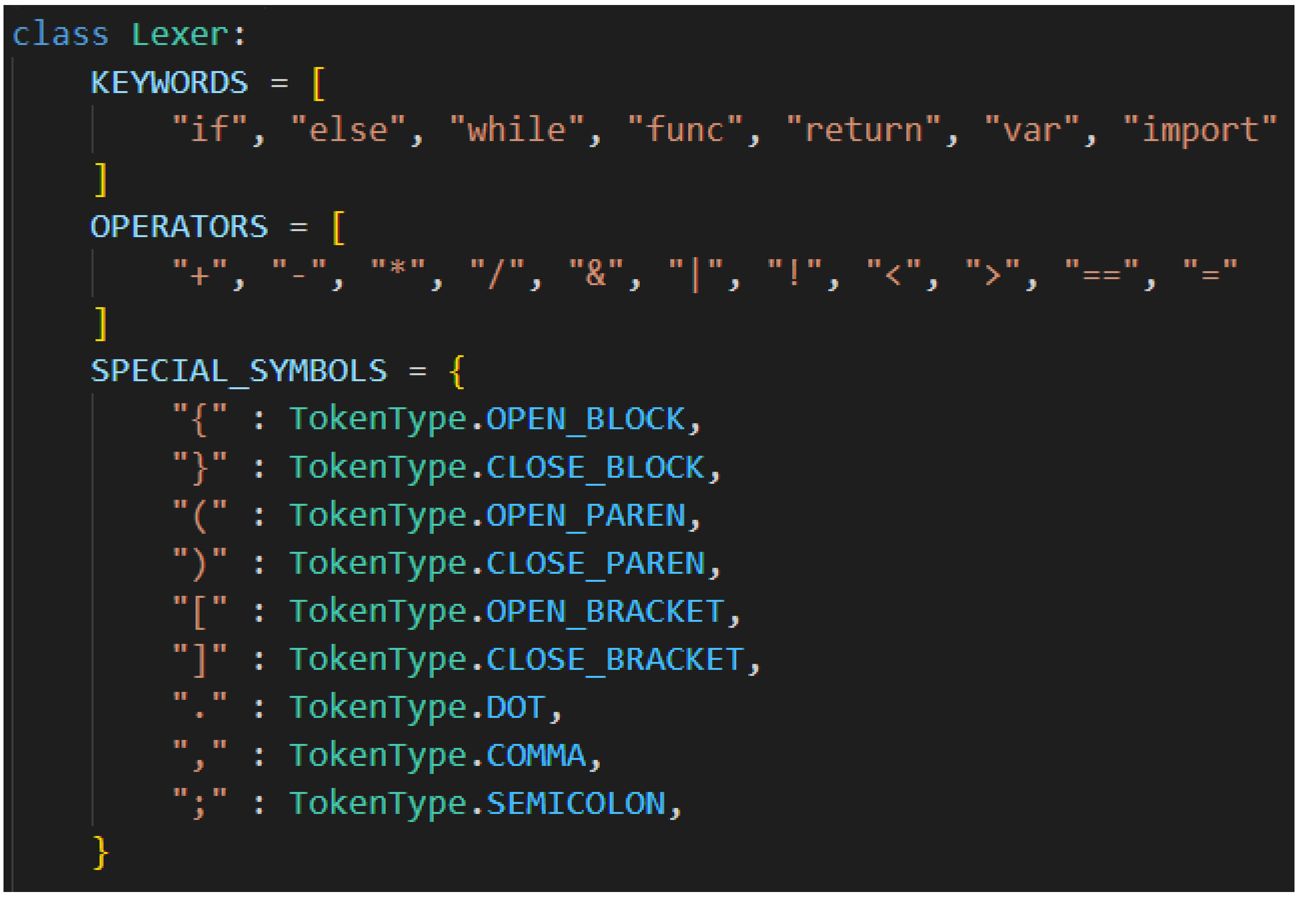
3. Implementation description:

To construct a lexer, we devised a language from scratch. While this language lacks a formally defined grammar, it draws inspiration from low-level languages such as C, C++, and even Turbo Pascal. Here, we present a comprehensive table delineating the tokens recognized within this language. These tokens encapsulate a wide array of syntactic elements, including identifiers, keywords, operators, and literals.



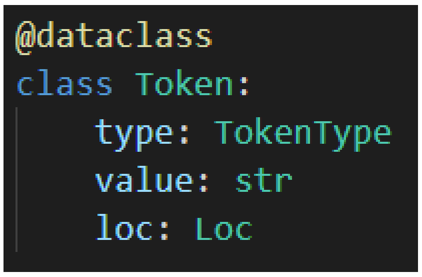
Token name	Explanation	Sample token values
IDENTIFIER	Represents names of variables, functions, or any user-defined identifiers.	`variable`, `my_function`, `count`
KEYWORD	Denotes reserved keywords in the language, such as control flow statements and declarations.	`if`, `else`, `while`, `return`
INTEGER	Represents integer literals.	`123`, `0`, `-42`
SEMICOLON	Marks the end of statements or declarations.	`;`
OPEN_BLOCK	Indicates the beginning of a block of code.	`{`
CLOSE_BLOCK	Indicates the end of a block of code.	`}`
OPEN_PAREN	Marks the beginning of a function call or expression grouping.	`(`
CLOSE_PAREN	Marks the end of a function call or expression grouping.	`)`
OPEN_BRACKET	Marks the beginning of an array literal or indexing expression.	`[`
CLOSE_BRACKET	Marks the end of an array literal or indexing expression.	`]`
COMMA	Separates elements in a list or function arguments.	`,`
ASSIGN	Denotes assignment operator.	`=`
DOT	Indicates access to object members or methods.	`.`
STRING	Represents string literals.	`"hello"`, `'world'`, `"123"`
OPERATOR	Denotes arithmetic, logical, or comparison operators.	`+`, `-`, `*`, `/`, `==`, `<`, `>`

Lexer:
Based on the table above, the main class of the lexer was created. The Lexer class helps us recognize these tokens in a given piece of code. For example, if we give it the word "if", it knows it's a keyword. Or if we give it the symbol "{", it knows it's an opening curly brace. This understanding of tokens is crucial for further processing code, like compiling or interpreting it. We have KEYWORDS which are special words in the programming language, like "if" or "return". OPERATORS, that are symbols used for mathematical or logical operations, such as "+", "-", or "==", and also SPECIAL_SYMBOLS. that are characters like parentheses, brackets, commas, and semicolons that have special meanings in code.



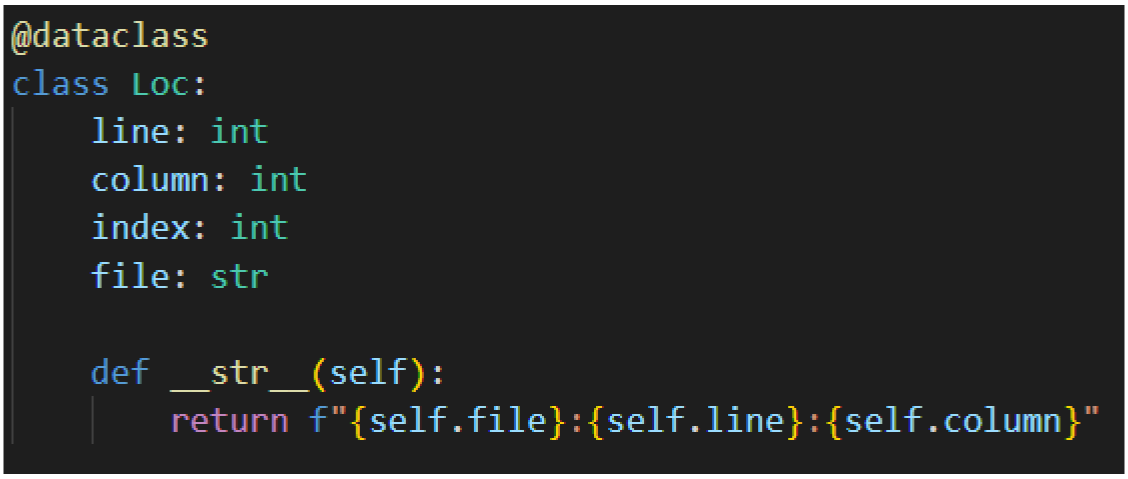
```
class Lexer:
    KEYWORDS = [
        "if", "else", "while", "func", "return", "var", "import"
    ]
    OPERATORS = [
        "+", "-", "*", "/", "&", "|", "!", "<", ">", "==", "="
    ]
    SPECIAL_SYMBOLS = {
        "{" : TokenType.OPEN_BLOCK,
        "}" : TokenType.CLOSE_BLOCK,
        "(" : TokenType.OPEN_PAREN,
        ")" : TokenType.CLOSE_PAREN,
        "[" : TokenType.OPEN_BRACKET,
        "]" : TokenType.CLOSE_BRACKET,
        "." : TokenType.DOT,
        "," : TokenType.COMMA,
        ";" : TokenType.SEMICOLON,
    }
```

Token class:
For easier and more organized work with the lexer, and also to be able to track the location of each token (column and line), we organized everything in different classes, this will also help with the printing of the tokens later. Token brings everything together. It combines a TokenType (like "keyword" or "operator") with a value (the actual word or symbol found in the code) and its location (where it was found in the code).



```
@dataclass
class Token:
    type: TokenType
    value: str
    loc: Loc
```

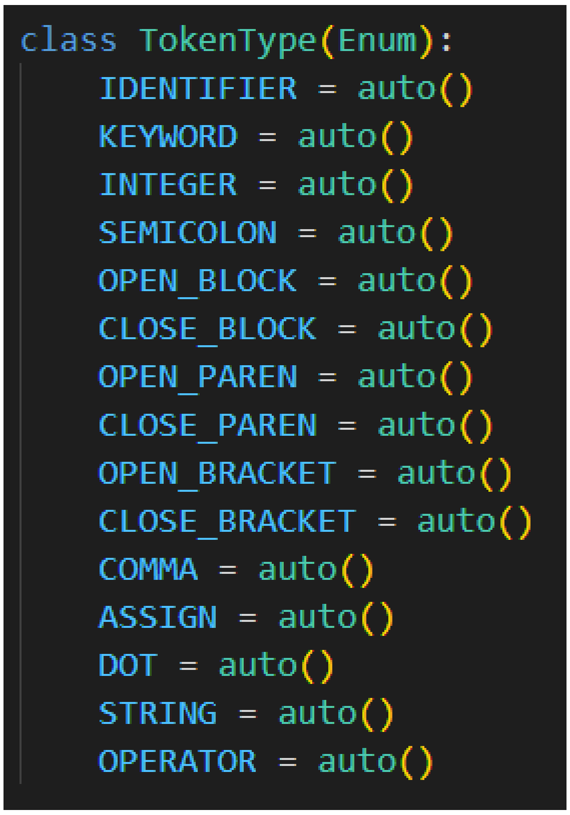
Loc class:
Loc stands for location, which tells us where in the code a certain token was found. It includes details like the line number, column number, index (position in the code), and the file name where the code is located.



```python
@dataclass
class Loc:
    line: int
    column: int
    index: int
    file: str

    def __str__(self):
        return f"{self.file}:{self.line}:{self.column}"
```

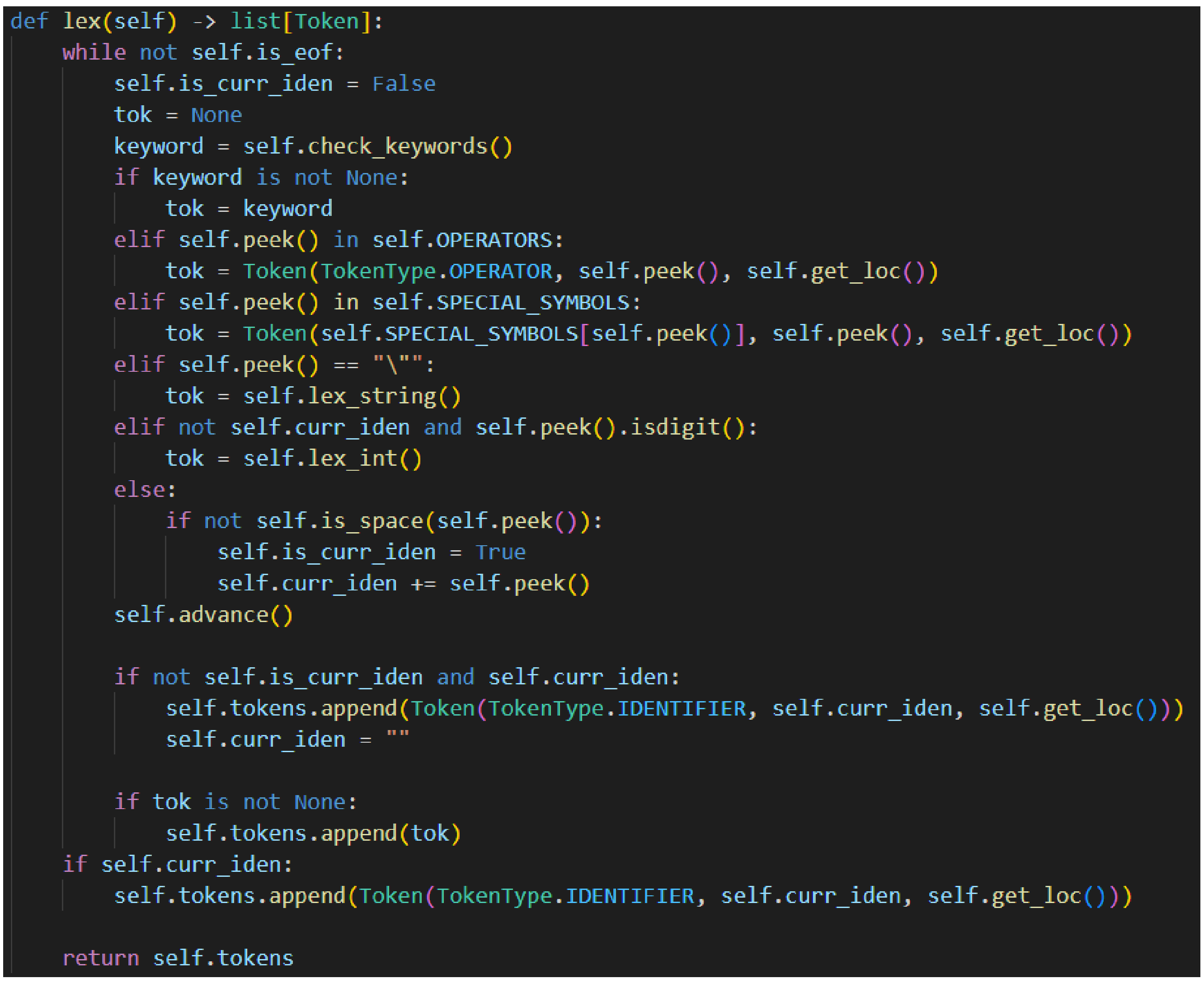
TokenType class:
The TokenType class serves as a blueprint for defining the various types of tokens that can be identified within a programming language.
The auto() function in the Enum class, when used within the TokenType class, automatically assigns unique values to each member of the enumeration. Here, each token type (IDENTIFIER, KEYWORD, INTEGER, etc.) will be assigned unique integer values starting from 1 incrementally. So IDENTIFIER will have the value 1, KEYWORD will have the value 2, INTEGER will have the value 3, and so on.



```python
class TokenType(Enum):
    IDENTIFIER = auto()
    KEYWORD = auto()
    INTEGER = auto()
    SEMICOLON = auto()
    OPEN_BLOCK = auto()
    CLOSE_BLOCK = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    OPEN_BRACKET = auto()
    CLOSE_BRACKET = auto()
    COMMA = auto()
    ASSIGN = auto()
    DOT = auto()
    STRING = auto()
    OPERATOR = auto()
```

The main lexer function.:
The lex method provided appears to be a part of a lexer implementation.
The method iterates over the input code until the end of the file (EOF) is reached. Inside the loop, it resets the flag is_curr_iden to False, indicating that the current token being processed is not an identifier.
It initializes a variable tok to None, which will be used to store the token being processed. It checks if the current character (peeked character) is a keyword using the check_keywords method. If it's a keyword, it assigns the corresponding token to tok.
If the current character is not a keyword, it checks if it's an operator. If so, it creates a token of type

TokenType.OPERATOR with the current character.
If the current character is not an operator, it checks if it's a special symbol defined in SPECIAL_SYMBOLS. If so, it creates a token using the corresponding TokenType from the dictionary. If the current character is a double quote ("), it calls the lex_string method to handle string literals.
If the current character is not a keyword, operator, special symbol, or double quote, and it's a digit, it calls the lex_int method to handle integer literals.
If none of the above conditions match, it checks if the current character is not a space and appends it to the curr_iden string if it's not already part of an identifier.
After processing the current character, it advances to the next character in the input code.
If the current character is not part of an identifier (is_curr_iden is False) and curr_iden is not empty, it appends the identifier token to the list of tokens and resets curr_iden to an empty string.
If tok is not None, indicating that a token was found, it appends the token to the list of tokens.
After processing all characters, if there is still content in curr_iden, it appends an identifier token to the list of tokens.
Finally, it returns the list of tokens.
This method essentially scans through the input code, identifies tokens (keywords, operators, special symbols, strings, integers, and identifiers), and creates corresponding Token objects, which are then added to the list of tokens.



```python
def lex(self) -> list[Token]:
    while not self.is_eof:
        self.is_curr_iden = False
        tok = None
        keyword = self.check_keywords()
        if keyword is not None:
            tok = keyword
        elif self.peek() in self.OPERATORS:
            tok = Token(TokenType.OPERATOR, self.peek(), self.get_loc())
        elif self.peek() in self.SPECIAL_SYMBOLS:
            tok = Token(self.SPECIAL_SYMBOLS[self.peek()], self.peek(), self.get_loc())
        elif self.peek() == "\"":
            tok = self.lex_string()
        elif not self.curr_iden and self.peek().isdigit():
            tok = self.lex_int()
        else:
            if not self.is_space(self.peek()):
                self.is_curr_iden = True
                self.curr_iden += self.peek()
        self.advance()

        if not self.is_curr_iden and self.curr_iden:
            self.tokens.append(Token(TokenType.IDENTIFIER, self.curr_iden, self.get_loc()))
            self.curr_iden = ""

        if tok is not None:
            self.tokens.append(tok)
    if self.curr_iden:
        self.tokens.append(Token(TokenType.IDENTIFIER, self.curr_iden, self.get_loc()))

    return self.tokens
```

4. Program execution:

Testing our lexer with a code snippet based on our language rules:

```
func factorial(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

var num = 5;
var result = factorial(num);
print("Factorial of", num, "is", result);
```

The out provided by the lexer was the following:

```
test.lang:1:1:  KEYWORD func
test.lang:1:16:  IDENTIFIER factorial
test.lang:1:15:  OPEN_PAREN (
test.lang:1:18:  IDENTIFIER n
test.lang:1:17:  CLOSE_PAREN )
test.lang:1:19:  OPEN_BLOCK {
test.lang:2:6:  KEYWORD if
test.lang:2:9:  OPEN_PAREN (
test.lang:2:12:  IDENTIFIER n
test.lang:2:12:  OPERATOR <
test.lang:2:13:  OPERATOR =
test.lang:2:15:  INTEGER 1
test.lang:2:16:  CLOSE_PAREN )
test.lang:2:18:  OPEN_BLOCK {
test.lang:3:10:  KEYWORD return
test.lang:3:17:  INTEGER 1
test.lang:3:18:  SEMICOLON ;
test.lang:4:6:  CLOSE_BLOCK }
test.lang:4:8:  KEYWORD else
test.lang:4:13:  OPEN_BLOCK {
test.lang:5:10:  KEYWORD return
test.lang:5:19:  IDENTIFIER n
test.lang:5:19:  OPERATOR *
test.lang:5:31:  IDENTIFIER factorial
test.lang:5:30:  OPEN_PAREN (
test.lang:5:33:  IDENTIFIER n
test.lang:5:33:  OPERATOR -
test.lang:5:35:  INTEGER 1
test.lang:5:36:  CLOSE_PAREN )
test.lang:5:37:  SEMICOLON ;
test.lang:6:6:  CLOSE_BLOCK }
test.lang:7:2:  CLOSE_BLOCK }
test.lang:9:2:  KEYWORD var
test.lang:9:10:  IDENTIFIER num
test.lang:9:10:  OPERATOR =
test.lang:9:12:  INTEGER 5
test.lang:9:13:  SEMICOLON ;
test.lang:10:2:  KEYWORD var
test.lang:10:13:  IDENTIFIER result
test.lang:10:13:  OPERATOR =
test.lang:10:25:  IDENTIFIER factorial
test.lang:10:24:  OPEN_PAREN (
test.lang:10:29:  IDENTIFIER num
test.lang:10:28:  CLOSE_PAREN )
test.lang:10:29:  SEMICOLON ;
test.lang:11:8:  IDENTIFIER print
test.lang:11:7:  OPEN_PAREN (
test.lang:11:8:  STRING Factorial of
test.lang:11:22:  COMMA ,
test.lang:11:28:  IDENTIFIER num
test.lang:11:27:  COMMA ,
test.lang:11:29:  STRING is
test.lang:11:33:  COMMA ,
test.lang:11:42:  IDENTIFIER result
test.lang:11:41:  CLOSE_PAREN )
test.lang:11:42:  SEMICOLON ;
```

Given sufficient spare time, the outcomes can be cross-verified for accuracy using alternative software tools or traditional methods such as manual proofreading with pen and paper.

5. Conclusion:

In conclusion, the successful implementation of the lexer in this university laboratory work marks a significant achievement in our understanding and application of fundamental concepts in programming language theory and lexical analysis. Through meticulous design, diligent coding, and rigorous testing, we have demonstrated our proficiency in constructing a robust and efficient lexer capable of accurately tokenizing source code inputs.