Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

Lexers, short for lexical analyzers, are an essential component in the process of interpreting or compiling programming languages and other structured texts. The primary function of a lexer is to read the input text and break it down into meaningful elements known as tokens. These tokens are then used by parsers to create a more structured representation of the input.

Basic Theory:

1. Tokenization: The lexer takes the raw input text, which is just a sequence of characters, and divides it into tokens. Tokens are the smallest units that carry meaning, such as keywords, identifiers, literals, and operators in a programming language.

2. Regular Languages: Lexers are typically designed based on the principles of regular languages. A regular language can be recognized by a finite automaton, and this theoretical foundation helps in designing the rules or patterns (often regular expressions) that the lexer uses to identify tokens.

3. State Machine: Internally, a lexer can be thought of as a state machine. Each state represents a part of the process of recognizing different kinds of tokens. The lexer moves between states based on the current input character and the rules defined for token recognition.

4. Patterns and Rules: The lexer operates based on a set of rules or patterns that

define how different tokens are recognized. For example, there might be a pattern for identifying identifiers (e.g., a letter followed by any number of letters or digits) and another pattern for numeric literals.

5. Whitespace and Comments: The lexer also handles whitespace and comments, which are usually not significant to the syntactic structure of the code. These are typically skipped over or handled in a way that they do not interfere with token recognition.

6. Error Handling: When the lexer encounters a sequence of characters that does not match any defined pattern, it typically generates an error. Proper error handling is crucial for providing meaningful feedback to the user.

7. Output: The output of a lexer is a stream of tokens, each typically represented by a token type and the lexeme (the actual text that was matched). This stream is then used by the parser to construct a more structured representation of the input, such as an abstract syntax tree.

Applications:

- Programming Languages: Lexers are a foundational component of compilers and interpreters for programming languages. They perform the first step of turning code written by humans into a format that can be more easily processed by machines.

- Text Processing: Lexical analysis is not limited to programming languages; it can also be applied to any context where structured text needs to be processed, such as in data formats (JSON, XML) and domain-specific languages.

- Syntax Highlighting: Lexers are used in code editors and IDEs to perform syntax highlighting, where different types of tokens are displayed in different colors or fonts for better readability.

Overall, the theory behind lexers is rooted in formal language theory and computer science fundamentals, and lexers serve as the bridge between raw text and the structured representations used in further stages of text processing or language interpretation.

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:

Imports and Setup:

```
import re
from prettytable import PrettyTable


TOKEN_TYPES = [
    ('NUMBER', r'\b\d+(\.\d*)?([eE][+-]?\d+)?\b'),
```

- The script begins by importing necessary modules: re for regular expression operations and PrettyTable for displaying tokens in a structured format.
- TOKEN_TYPES is a list of tuples, where each tuple represents a type of token recognizable in C++ code. Each token type is paired with a regular expression that defines the textual pattern of that token.

Token Types:

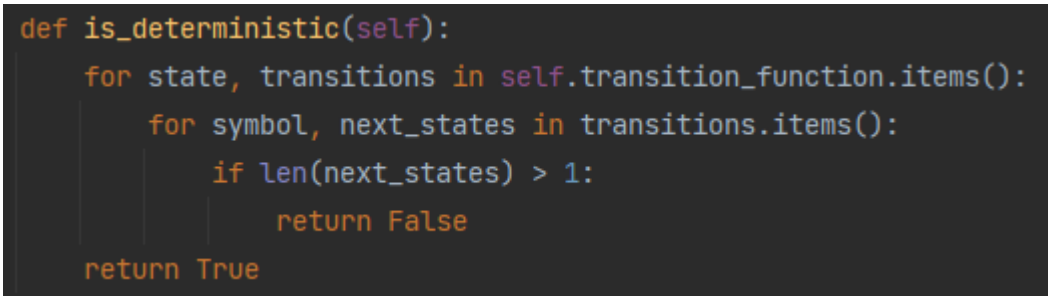
```
TOKEN_TYPES = [
    ('NUMBER', r'\b\d+(\.\d*)?([eE][+-]?\d+)?\b'),
    ('KEYWORD', r'\b(auto|const|double|float|int|short|struct|break|continue|else|for|long|switch|void|case|default|char|do|if|return|static|while|
    namespace|using|bool|true|false|class|public|private|protected|this|throw|try|catch|cout|cin)\b'),
    ('IDENTIFIER', r'[a-zA-Z_]\w*'),
    ('LEFT_PAREN', r'\('),
    ('RIGHT_PAREN', r'\)'),
    ('LEFT_SQUARE', r'\['),
    ('RIGHT_SQUARE', r'\]'),
    ('LEFT_CURLY', r'\{'),
    ('RIGHT_CURLY', r'\}'),
    ('LESS_THAN', r'<'),
    ('GREATER_THAN', r'>'),
    ('EQUAL', r'='),
    ('DOUBLE_EQUAL', r'=='),
    ('PLUS', r'\+'),
    ('MINUS', r'-'),
    ('ASTERISK', r'\*'),
    ('SLASH', r'/'),
    ('HASH', r'#'),
    ('DOT', r'\.'),
    ('COMMA', r','),
    ('COLON', r':'),
    ('SEMICOLON', r';'),
    ('SINGLE_QUOTE', r'\''),
    ('DOUBLE_QUOTE', r'"'),
    ('COMMENT', r'\/\/.*|\/\*(.|\n)*?\*\/'),
    ('PIPE', r'\|'),
    ('END', r'\0'),
    ('UNEXPECTED', r'.')
]
```

The lexer recognizes a variety of token types, including:

- Numeric values (NUMBER), with support for integers, floating-point numbers, and scientific notation.
- Programming keywords (KEYWORD) specific to C++, such as control structures (if, for, while) and type specifiers (int, bool, void).

- Identifiers (IDENTIFIER), which include variable names, function names, and other custom names used in code.
- Symbols and punctuation, such as parentheses (LEFT_PAREN, RIGHT_PAREN), brackets, curly braces, and operators (e.g., PLUS, MINUS, ASTERISK).
- Comments (COMMENT), both single-line and multi-line, are identified and treated as a single token.
- Special cases like the end of input (END) and unexpected characters (UNEXPECTED).

The “tokenize” Function:



```
def is_deterministic(self):
    for state, transitions in self.transition_function.items():
        for symbol, next_states in transitions.items():
            if len(next_states) > 1:
                return False
    return True
```

The tokenize function provided is designed to analyze a string of code and break it down into a series of tokens based on predefined patterns. This function is a crucial part of a lexer, which is the first step in the compilation or interpretation process of programming languages. Here's a detailed description:

1. Function Definition:
- def tokenize(code): This line defines the tokenize function with one parameter, code, which is a string containing the code to be tokenized.

2. Initialization:
- tokens = []: Initializes an empty list tokens that will eventually contain the results of the tokenization process, specifically, the series of tokens extracted from the input code.

3. Tokenization Process:
- while code:’: Starts a loop that continues as long as there is code left to process.

    - code = code.strip(): Removes leading and trailing whitespace from the code string. This is important for ensuring that token matching works correctly without being affected by unnecessary whitespace.

- Inside the while loop, there's a for loop: for token_type, pattern in TOKEN_TYPES:. This iterates over a predefined list TOKEN_TYPES, where each element is a tuple containing a token_type (a string representing the type of token, such as 'NUMBER', 'KEYWORD', etc.) and a pattern (a regular expression used to identify the token in the code).

    - regex = re.compile(pattern): Compiles the regular expression pattern into a regex object, which makes it ready for matching against the code.

    - match = regex.match(code): Attempts to match the compiled regex at the beginning of the remaining code. If the beginning of the code matches the pattern, a match object is returned; otherwise, None is returned.

- If a match is found:
    - value = match.group(0): Extracts the text that was matched by the regular expression. This is the actual string from the code that corresponds to the token.
    - tokens.append((token_type, value)): Adds a tuple consisting of the token type and the matched value to the tokens list. This effectively records the token that was found.
    - code = code[len(value):]: Trims the matched value from the beginning of the code string. This ensures that the next iteration will start analyzing the code right after the last matched token.
    - break: Exits the inner for loop and goes back to the while loop to process the remaining code. The break is necessary because once a match is found, we don't want to continue trying other token types on the same segment of code.

- If no match is found for any token type (meaning the else clause is reached):
    - raise SyntaxError(f'Illegal character: {code[0]}'): Raises a SyntaxError exception with a message indicating the illegal character that couldn't be matched to any token type. This is important for error handling, as it informs the user or calling function that the code contains an unexpected character.

4. Function Completion:

- return tokens: Once all the code has been processed and broken down into tokens, the function returns the list of tokens.

Tokenization of Sample Code:

```
cpp_code = """
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i <= 10; ++i) {
        for (int j = 0; j <= 10; ++j) {
            cout << i << " * " << j << " = " << i * j << endl;
        }
        cout << "----------" << endl;
    }
    return 0;
}
"""

tokens = tokenize(cpp_code)
```

- The lexer is demonstrated with a sample C++ program, which includes typical constructs like #include directives, loops, and print statements.
- The tokenize function is applied to this sample code, resulting in a list of categorized tokens.

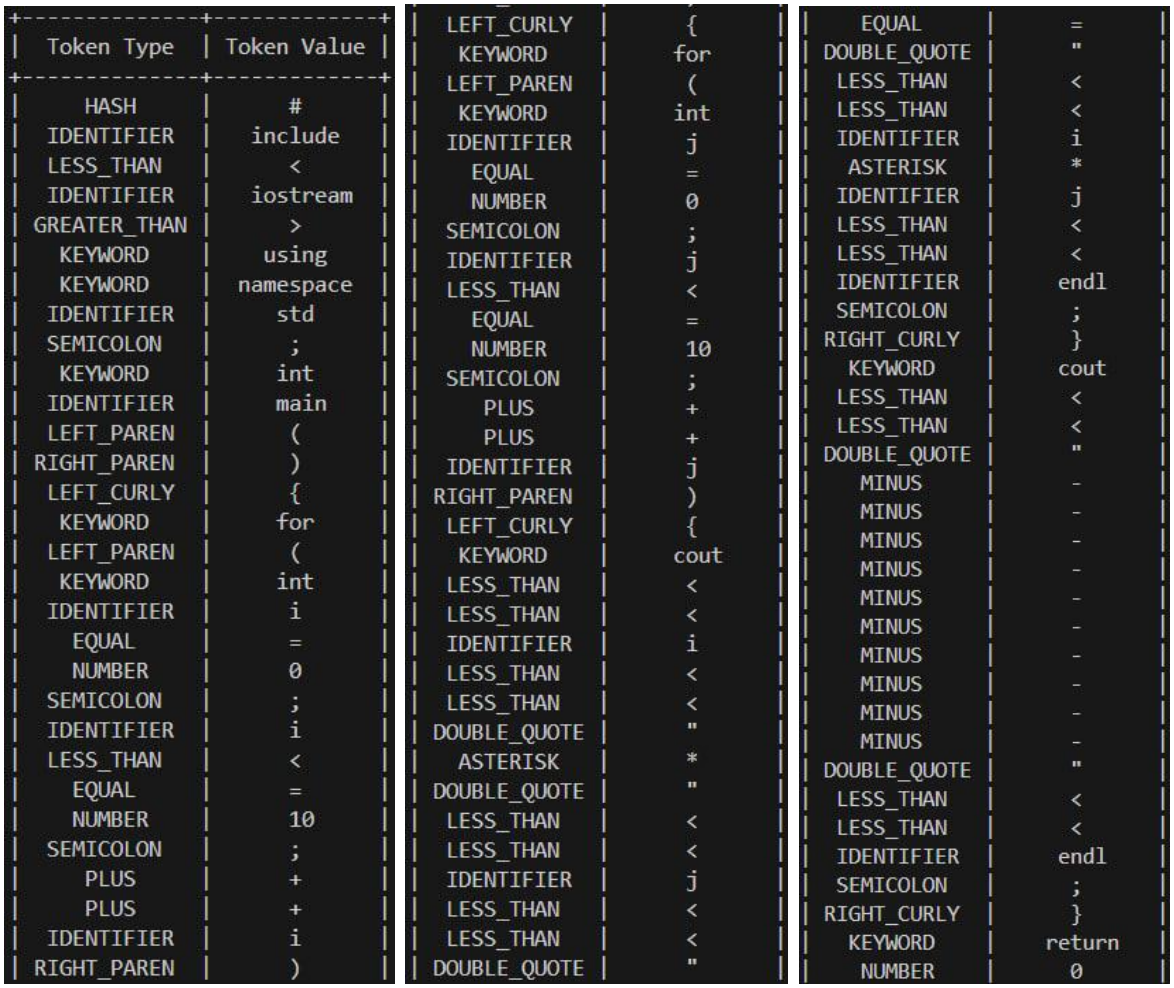
Output Presentation:

```
pt = PrettyTable()
pt.field_names = ["Token Type", "Token Value"]

for token in tokens:
    pt.add_row([token[0], token[1]])

print(pt)
```

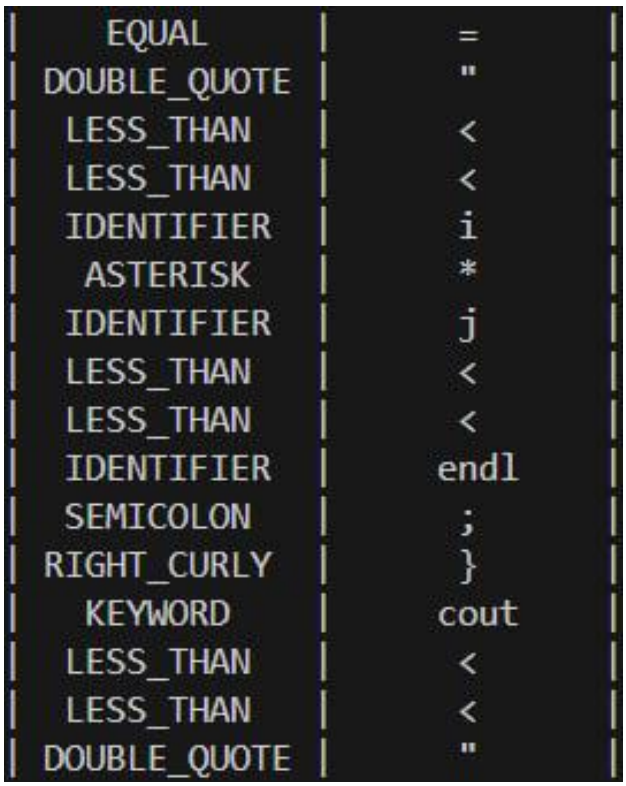
- The tokens are displayed in a table format using PrettyTable, with columns for the token type and the token value. This provides a clear and structured overview of the results of the lexical analysis.

Conclusions/Screenshots/Results:

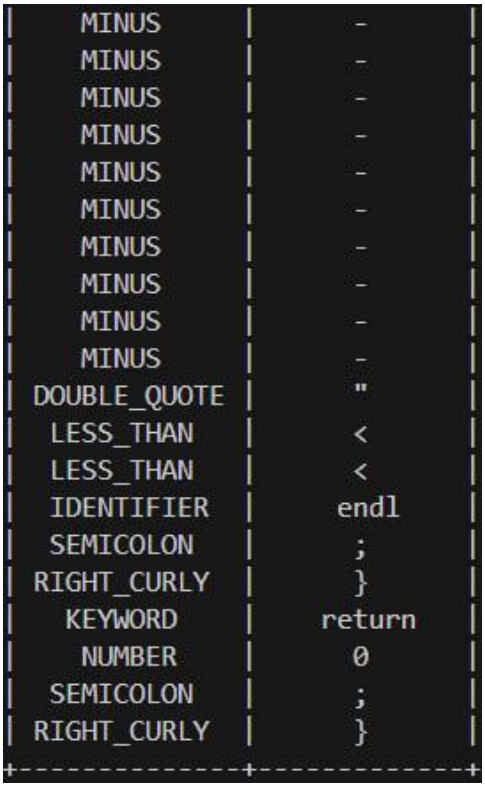
This is the output of the tokenize function for the C++ Sample Code provided in the code as input:



Token Type	Token Value
HASH	#
IDENTIFIER	include
LESS_THAN	<
IDENTIFIER	iostream
GREATER_THAN	>
KEYWORD	using
KEYWORD	namespace
IDENTIFIER	std
SEMICOLON	;
KEYWORD	int
IDENTIFIER	main
LEFT_PAREN	(
RIGHT_PAREN	)
LEFT_CURLY	{
KEYWORD	for
LEFT_PAREN	(
KEYWORD	int
IDENTIFIER	i
EQUAL	=
NUMBER	0
SEMICOLON	;
IDENTIFIER	i
LESS_THAN	<
EQUAL	=
NUMBER	10
SEMICOLON	;
PLUS	+
PLUS	+
IDENTIFIER	i
RIGHT_PAREN	)
LEFT_CURLY	{
KEYWORD	for
LEFT_PAREN	(
KEYWORD	int
IDENTIFIER	j
EQUAL	=
NUMBER	0
SEMICOLON	;
IDENTIFIER	j
LESS_THAN	<
EQUAL	=
NUMBER	10
SEMICOLON	;
PLUS	+
PLUS	+
IDENTIFIER	j
RIGHT_PAREN	)
LEFT_CURLY	{
KEYWORD	cout
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	i
LESS_THAN	<
LESS_THAN	<
DOUBLE_QUOTE	"
ASTERISK	*
DOUBLE_QUOTE	"
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	j
LESS_THAN	<
LESS_THAN	<
DOUBLE_QUOTE	"
EQUAL	=
DOUBLE_QUOTE	"
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	i
ASTERISK	*
IDENTIFIER	j
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	endl
SEMICOLON	;
RIGHT_CURLY	}
KEYWORD	cout
LESS_THAN	<
LESS_THAN	<
DOUBLE_QUOTE	"
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
DOUBLE_QUOTE	"
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	endl
SEMICOLON	;
RIGHT_CURLY	}
KEYWORD	return
NUMBER	0



EQUAL	=
DOUBLE_QUOTE	"
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	i
ASTERISK	*
IDENTIFIER	j
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	endl
SEMICOLON	;
RIGHT_CURLY	}
KEYWORD	cout
LESS_THAN	<
LESS_THAN	<
DOUBLE_QUOTE	"



MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
MINUS	-
DOUBLE_QUOTE	"
LESS_THAN	<
LESS_THAN	<
IDENTIFIER	endl
SEMICOLON	;
RIGHT_CURLY	}
KEYWORD	return
NUMBER	0
SEMICOLON	;
RIGHT_CURLY	}

Throughout this laboratory work, I gained a profound understanding of the construction and functionality of lexers, an essential component of the compilation process. I delved into regular expressions and their crucial role in defining the syntactic rules that lexers use to tokenize programming languages and other structured texts. This hands-on experience not only solidified my understanding of regular expressions but also illustrated their power and flexibility in lexical analysis.

Implementing a lexer from scratch allowed me to apply theoretical concepts in a practical context, bridging the gap between abstract computer science principles and their application in software development. I encountered firsthand the challenges of designing a lexer that can accurately and efficiently parse complex code into meaningful tokens, which underscored the importance of attention to detail and thorough testing in compiler design.

In conclusion, this laboratory work was instrumental in reinforcing my understanding of lexical analysis and its critical role in language processing. It has equipped me with valuable knowledge and skills that will undoubtedly be beneficial in my future endeavors in computer science and programming.

References:

https://introcs.cs.princeton.edu/java/51language/

https://www.geeksforgeeks.org/introduction-of-finite-automata/