MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory work 3: Implementing a simple lexer

Course: Formal Languages & Finite Automata

Author: Bostan Victor, FAF-222

Chișinău, 2024

THEORY

In the field of computer science, lexical analysis is a crucial phase in the process of interpreting or compiling programming languages. This process involves analyzing the input strings of characters, such as source code, and converting them into a sequence of tokens. Tokens are the fundamental building blocks of a language's syntax, representing logically grouped character sequences, such as keywords, identifiers, literals, and operators. Lexical analysis serves as the bridge between raw text and its syntactical structure, laying the groundwork for subsequent parsing stages.

A lexer, short for lexical analyzer or tokenizer, is a software component designed to perform lexical analysis. Its primary function is to scan the input text, character by character, identifying and categorizing substrings according to the rules of the language's syntax. These substrings are then output as tokens, each labeled with a token type that describes its role within the language. For example, in CSS, token types might include selectors, properties, and values. The lexer simplifies the input text, stripping away irrelevant details such as whitespace and comments, which are not essential for understanding the structure of the code.

The process of lexical analysis typically follows a well-defined set of steps. The lexer starts at the beginning of the input text and reads characters sequentially, using pattern-matching techniques, often implemented through regular expressions, to recognize different elements of the language. Once a pattern is identified, the corresponding substring is converted into a token. The lexer then continues this process, moving through the input text until all characters have been examined and categorized. Error handling is an integral part of this stage; if the lexer encounters a sequence of characters that does not match any known pattern, it must decide how to proceed, which often involves generating an error message or skipping the problematic characters.

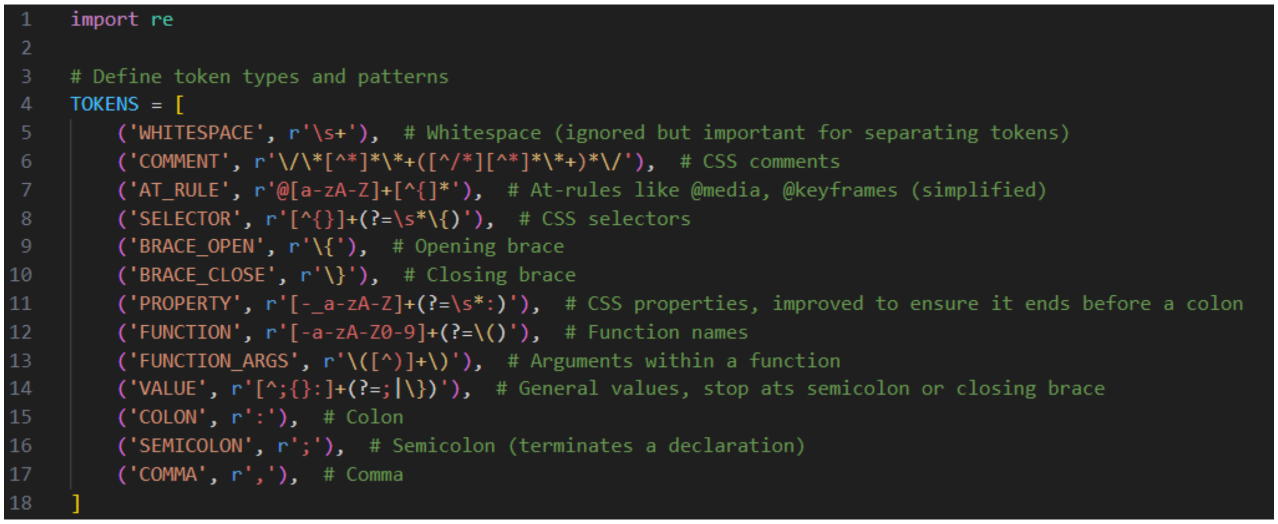
The efficiency and accuracy of a lexer are vital for the overall performance and reliability of a compiler or interpreter. By transforming a stream of characters into a stream of tokens, the lexer abstracts away the details of the text, providing a clean, simplified interface for the parser to work with. This separation of concerns allows each component of the compiler or interpreter to focus on its specific task, contributing to the modularity and maintainability of the software.

ONJECTIVES

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 DESCRIPTION

For this laboratory, as for the previous ones, I decided to work in python. Now, I had to decide what I would implement the lexer for and I chose *CSS*. The reason behind this is that it is a well-structured style sheet language and also it provides a more complex task for implementation. Now, in order to implement the lexer in python i decided to use the RegEx library, which stands for "regular expressions". Regular expressions are used for searching, matching, and manipulating strings based on specific patterns. In the context of this lexer, they are employed to identify the different types of tokens present in CSS code.

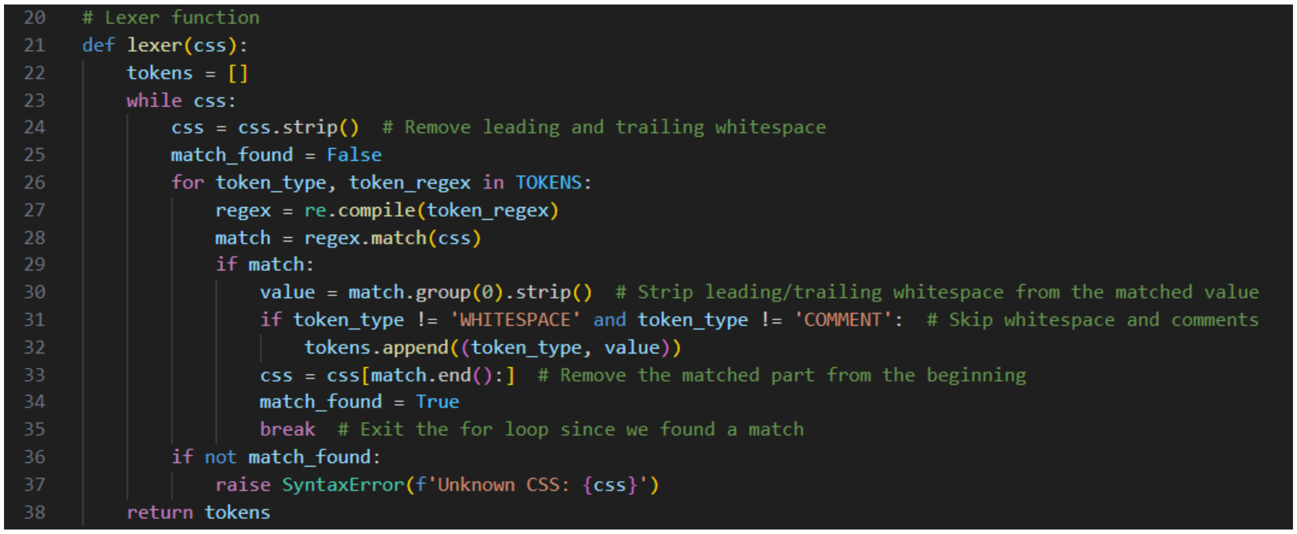


```python
import re

# Define token types and patterns
TOKENS = [
    ('WHITESPACE', r'\s+'),  # Whitespace (ignored but important for separating tokens)
    ('COMMENT', r'\/\*[^*]*\*+([^/*][^*]*\*+)*\/'),  # CSS comments
    ('AT_RULE', r'@[a-zA-Z]+[^{]*'),  # At-rules like @media, @keyframes (simplified)
    ('SELECTOR', r'[^{}]+(?=\s*\{)'),  # CSS selectors
    ('BRACE_OPEN', r'\{'),  # Opening brace
    ('BRACE_CLOSE', r'\}'),  # Closing brace
    ('PROPERTY', r'[-_a-zA-Z]+(?=\s*:)'),  # CSS properties, improved to ensure it ends before a colon
    ('FUNCTION', r'[-a-zA-Z0-9]+(?=\()'),  # Function names
    ('FUNCTION_ARGS', r'\([^)]+\)'),  # Arguments within a function
    ('VALUE', r'[^;{}:]+(?=;|\})'),  # General values, stop ats semicolon or closing brace
    ('COLON', r':'),  # Colon
    ('SEMICOLON', r';'),  # Semicolon (terminates a declaration)
    ('COMMA', r','),  # Comma
]
```

The first part of the implementation involves defining the types of tokens that can be encountered in CSS files and their corresponding regular expression patterns. These tokens include whitespace, comments, at-rules, selectors, braces, properties, function names, function arguments, values, colons, semicolons, and commas. This definition is crucial as it allows the lexer to recognize different elements within the CSS code. It is also important to note that for at_rule I used a simplified version, since it only matches any string that starts with '@' followed by alphabetic characters, but does not account for all the nuances and specific formats of various at-rules like @media or @keyframes in full CSS syntax. I chose to do this since the point of this laboratory is not to create the actual lexer for CSS, but rather understand how a lexer works in general.

Next, I create a function, named lexer that takes a string containing CSS code as input. It starts by initializing an empty list called tokens, which will store the recognized tokens.

The main loop of the lexer continues as long as there is CSS code to analyze. Inside the loop, the CSS code is first stripped of leading and trailing whitespace. This preprocessing step simplifies pattern matching since we don't have to deal with extraneous whitespace.



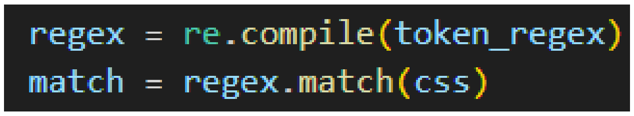
```python
# Lexer function
def lexer(css):
    tokens = []
    while css:
        css = css.strip()  # Remove leading and trailing whitespace
        match_found = False
        for token_type, token_regex in TOKENS:
            regex = re.compile(token_regex)
            match = regex.match(css)
            if match:
                value = match.group(0).strip()  # Strip leading/trailing whitespace from the matched value
                if token_type != 'WHITESPACE' and token_type != 'COMMENT':  # Skip whitespace and comments
                    tokens.append((token_type, value))
                css = css[match.end():]  # Remove the matched part from the beginning
                match_found = True
                break  # Exit the for loop since we found a match
        if not match_found:
            raise SyntaxError(f'Unknown CSS: {css}')
    return tokens
```

For each iteration of the loop, the lexer iterates over the predefined list of token types and their corresponding regular expression patterns. For each token type:

1. A regular expression object is compiled from the token's pattern.
2. The lexer tries to match this regular expression at the beginning of the remaining CSS code.
3. If a match is found:
    a) The matched string is extracted, and leading/trailing whitespace is removed from it.
    b) If the token type is not 'WHITESPACE' or 'COMMENT' (since these are typically not needed for further analysis), the token type and the trimmed matched string are appended as a tuple to the tokens list.
    c) The matched portion is removed from the beginning of the CSS code, and the lexer moves on to find the next token.
    d) A flag match_found is set to True, indicating that a token has been successfully matched and processed.

If no match is found for any token type, this indicates that the CSS code contains syntax that is not recognized by the lexer. In this case, a SyntaxError is raised, pointing out the problematic portion of the CSS code.

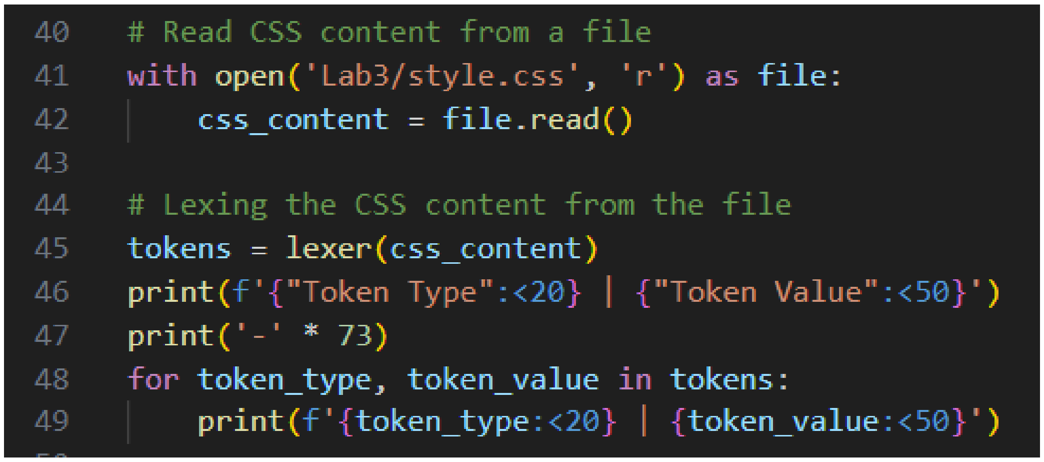
After the entire CSS content has been processed, the list of tokens is returned. This list represents the lexical structure of the CSS code and can be used for further analysis or processing, such as parsing the CSS into a more structured format.



```python
regex = re.compile(token_regex)
match = regex.match(css)
```

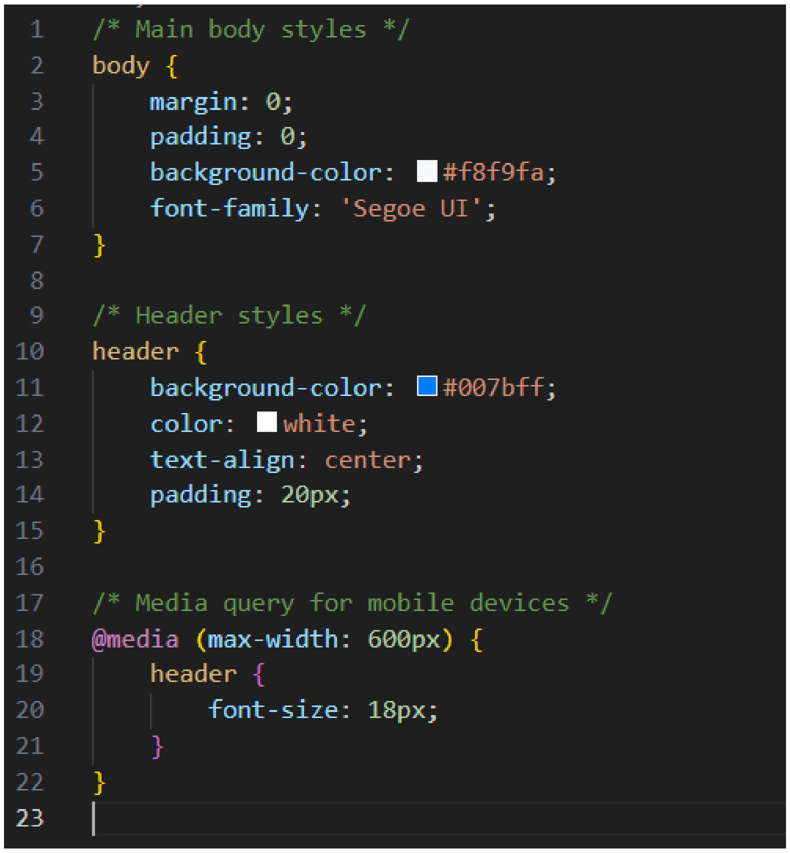
I also feel like it is necessary to further explain how these RegEx methods work behind the scenes, as they play a vital role in the lexer.

These lines of code compile the regular expression defined by token_regex and then use this compiled pattern to search for a match at the beginning of the css string. The re.compile function compiles the regular expression pattern into a regex object, which makes subsequent matching operations more efficient, especially when the same pattern is used multiple times. The match method of the compiled regex object then attempts to match the pattern from the start of the css string. If a match is found, the method returns a match object containing information about the part of the string that was matched; otherwise, it returns None.



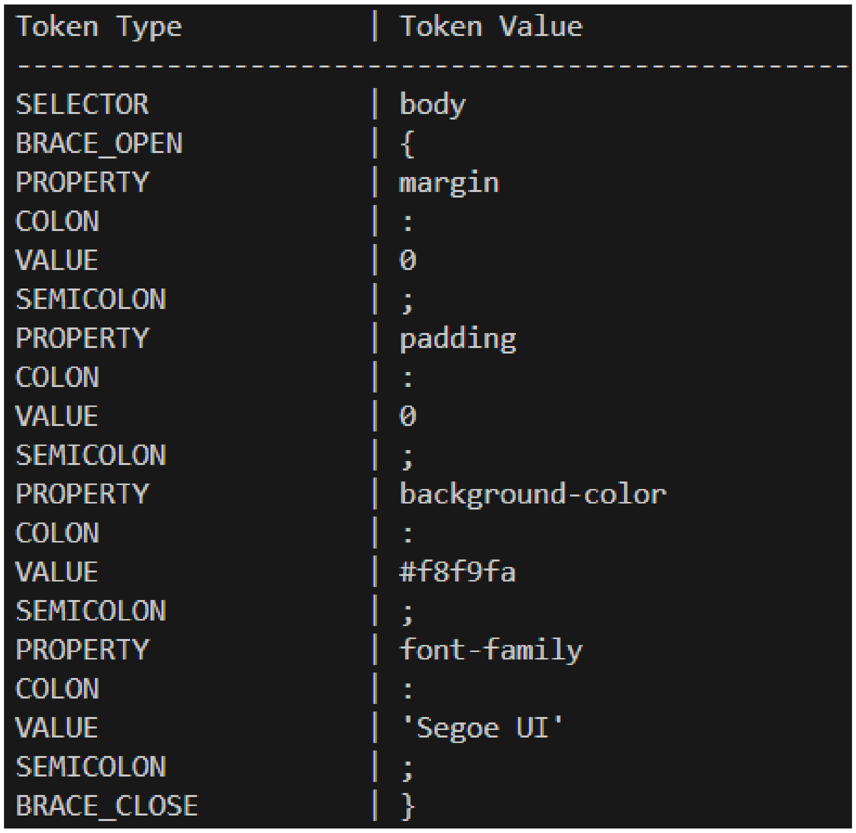
```python
# Read CSS content from a file
with open('Lab3/style.css', 'r') as file:
    css_content = file.read()

# Lexing the CSS content from the file
tokens = lexer(css_content)
print(f'{"Token Type":<20} | {"Token Value":<50}')
print('-' * 73)
for token_type, token_value in tokens:
    print(f'{token_type:<20} | {token_value:<50}')
```

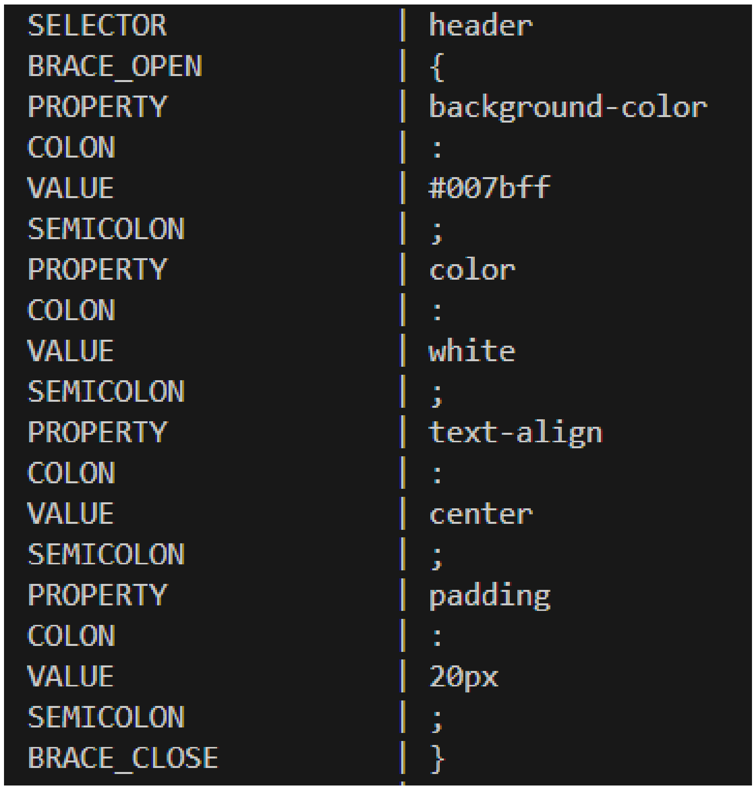
Finally, it's time to test the lexer. I did this by reading some sample css code from a file and then passing it to my lexer. Here is an example of css code:

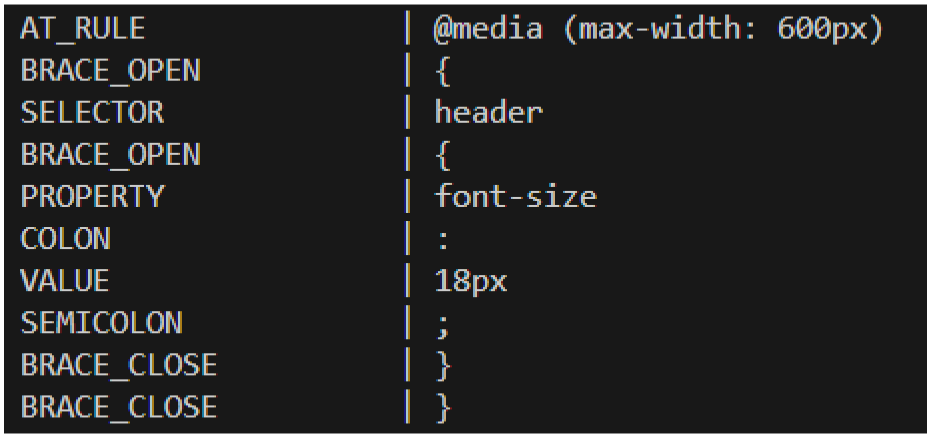


```css
/* Main body styles */
body {
    margin: 0;
    padding: 0;
    background-color: #f8f9fa;
    font-family: 'Segoe UI';
}

/* Header styles */
header {
    background-color: #007bff;
    color: white;
    text-align: center;
    padding: 20px;
}

/* Media query for mobile devices */
@media (max-width: 600px) {
    header {
        font-size: 18px;
    }
}

```

And here is the output for this code (I formmated the output so it's more readable):


```
Token Type             | Token Value
---------------------------------------------------
SELECTOR               | body
BRACE_OPEN             | {
PROPERTY               | margin
COLON                  | :
VALUE                  | 0
SEMICOLON              | ;
PROPERTY               | padding
COLON                  | :
VALUE                  | 0
SEMICOLON              | ;
PROPERTY               | background-color
COLON                  | :
VALUE                  | #f8f9fa
SEMICOLON              | ;
PROPERTY               | font-family
COLON                  | :
VALUE                  | 'Segoe UI'
SEMICOLON              | ;
BRACE_CLOSE            | }
```


```
SELECTOR               | header
BRACE_OPEN             | {
PROPERTY               | background-color
COLON                  | :
VALUE                  | #007bff
SEMICOLON              | ;
PROPERTY               | color
COLON                  | :
VALUE                  | white
SEMICOLON              | ;
PROPERTY               | text-align
COLON                  | :
VALUE                  | center
SEMICOLON              | ;
PROPERTY               | padding
COLON                  | :
VALUE                  | 20px
SEMICOLON              | ;
BRACE_CLOSE            | }
```



```
AT_RULE                 | @media (max-width: 600px)
BRACE_OPEN              | {
SELECTOR                | header
BRACE_OPEN              | {
PROPERTY                | font-size
COLON                   | :
VALUE                   | 18px
SEMICOLON               | ;
BRACE_CLOSE             | }
BRACE_CLOSE             | }
```

CONCLUSION

In conclusion, this lab exercise delved into the practical aspects of lexical analysis, particularly focusing on the construction and application of a lexer for CSS content. By developing and refining a lexer using regular expressions, I deepened my understanding of how lexical analyzers dissect and categorize source code into meaningful tokens, a critical step in the compilation or interpretation process. The employment of Python's RegEx module facilitated a hands-on approach to pattern matching and tokenization. Overall, this laboratory has solidified my comprehension of the role and mechanics of lexers in programming language processing, laying a robust groundwork for future studies in compiler construction and language design.