MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

Laboratory work 3:
Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-223
Vieru Mihai

Verified:
asist. univ.
Cretu Dumitru

Chișinău, 2024

Content

Theory

Lexers, also known as lexical analyzers, are fundamental components in the field of computer science and compiler design. They play a crucial role in the process of converting source code into tokens that can be further processed by parsers.

At its core, a lexer serves as the initial stage of a compiler or interpreter. Its primary task is to break down the input source code into meaningful units called tokens. These tokens represent the smallest identifiable elements of the language syntax, such as keywords, identifiers, operators, and literals. By tokenizing the input, lexers lay the groundwork for subsequent phases of compilation or interpretation, facilitating syntactic analysis and semantic processing.

The process of lexing involves scanning the input character stream and categorizing characters or sequences of characters into distinct token types based on predefined rules. These rules are typically specified using formal languages such as regular expressions or finite automata. Each rule corresponds to a particular token type, defining the patterns that constitute valid tokens in the language grammar.

Lexers employ various techniques to efficiently tokenize the input source code. One common approach is the use of deterministic finite automata (DFA), which provide a systematic method for recognizing token patterns. DFA-based lexers traverse the input character by character, transitioning between states according to the defined rules until a token boundary is reached. This deterministic process ensures linear time complexity, making lexing computationally efficient even for large input files.

Another key concept in lexer theory is the notion of lexical analysis phases. Lexing is often divided into multiple phases, each responsible for handling different aspects of tokenization. These phases may include preprocessing, where whitespace and comments are discarded, and lexical scanning, where tokens are identified and classified. By breaking down the lexing process into distinct stages, lexers can be designed and optimized more effectively, enhancing modularity and maintainability.

Lexers also play a crucial role in error handling and reporting within compilers and interpreters. During lexing, lexical errors such as invalid characters or unrecognized tokens may be encountered. Lexers are responsible for detecting and diagnosing these errors, providing meaningful feedback to the user to aid in debugging. Error recovery mechanisms may also be incorporated into lexers to gracefully handle syntax errors and resume tokenization without disrupting the compilation or interpretation process.

In addition to traditional compilers, lexers find application in various other domains, including text processing, parsing, and language recognition. They serve as building blocks for tools such as syntax highlighters, code formatters, and static analyzers, enhancing the development experience and improving code quality.

In conclusion, lexers are indispensable components in the realm of compiler design and software

engineering. By transforming raw source code into structured tokens, lexers enable the systematic analysis and interpretation of programming languages, laying the groundwork for the implementation of compilers, interpreters, and related tools. Through their meticulous tokenization process and error-handling capabilities, lexers contribute significantly to the reliability, efficiency, and usability of software systems.

Lexer

Before using a lexer, we have to define the tokens that the lexer uses. Here are the names of the tokens and the delimiters which define these tokens. I chose a list of lists, where the delimiters are the regex which follows the pattern of the token, and the names of those tokens.

Tokens:

```
Tokens = [
    [r"\A\s+", "WHITESPACE"],
    [r"\A;" , ";"],
    [r"\A[(]", "("],
    [r"\A[)]", ")"],
    [r"\A[{]", "{"],
    [r"\A[}]", "}"],
    [r'\A"""([\s\S]*?)"""', "BCOMMENT"],
    [r"\A\#.*\$", "COMMENT"],
    [r"\A\bif\b", "IF"],
    [r"\A\belse\b", "ELSE"],
    [r"\A\blet\b", "DECLARATOR"],
    [r"\A[\^\s\W\d]+", "VARIABLE"],
    [r'\A=(?!=)', "DECLARATOR_OPERATOR"],
    [r'\A==(?!=)', "EQUAL_OPERATOR"],
    [r'\A[+\-]', "ADDITIVE_OPERATOR"],
    [r'\A[*\/]', "MULTIPLICATIVE_OPERATOR"],
    [r"\A\d+", "NUMBER"],
    [r'\A"[^"]*"', "STRING"],
    [r"\A'[^']*'", "STRING"],
]
```

Then, I initialize a class called Tokenizer (Lexer/Scanner), which initializes with 2 variables: the

string on which we will do tokenization and the cursor, which traverses the string so we don't tokenize a token that has already been met.

Tokenizer:

```
class Tokenizer:
    def __init__(self, string):
        self._string = string
        self._coursor = 0
```

Now, I can start with the actual tokenization. In order to do that, I have created a recursive method which utilizes the cursor and regex to match all of the tokens inside the given string. The way I do this is simple. I iterate through all of the tokens and see if the current string starts the same way as any of the tokens do. If it does, but it is a whitespace or comment, then that token is skipped by advancing the cursor and calling the method recursively again. Thus, as the cursor is shifted and that token is jumped over, we can start to search for the next token. But only after we have checked if there can be a next token by using the getNextToken method, which checks if the cursor did not exceed the length of the string. If yes, then break out of the recursive loop.

```
def hasMoreTokens(self):
        return self._coursor < len(self._string)
```

Then again, we can iterate through all of the tokens, and if the next token is found but it is not a whitespace or comment, then the cursor is advanced, and the lexeme is returned in the form of a dictionary made out of the type (name) of the lexeme (e.g., STRING, NUMBER) and its value (e.g., 1, "Hello!"), which is returned.

GetToken:

```
def getNextToken(self):
        if not self.hasMoreTokens():
            return None

        curr_string = self._string[self._coursor:]
```

```
            for regex, literal_type in Tokens:
                match = re.findall(regex, curr_string, flags=re.MULTILINE)

                if len(match) == 0:
                    continue

                self._coursor += len(match[0])

                if literal_type in ["WHITESPACE", "BCOMMENT","COMMENT"]:
                    if literal_type == "BCOMMENT":
                        self._coursor += 6

                    return self.getNextToken()

                return {
                    "type": literal_type,
                    "value": match[0]
                }
```

Conclusion:

In this lab, I developed a lexer to tokenize a given string into defined tokens based on specified delimiters. The process involved defining tokens and their corresponding regex delimiters, initializing a Tokenizer class, and implementing tokenization logic.

Firstly, I defined the tokens and delimiters using a list of lists structure, facilitating the association between token names and their regex patterns.

Next, I instantiated a Tokenizer class, equipped with variables for the input string and a cursor to track traversal progress.

The tokenization process involved recursively matching tokens within the string by iterating through each token and comparing its regex pattern with the current substring. Whitespace and comments were skipped by advancing the cursor, ensuring only valid tokens were identified.

Upon encountering a valid token, the cursor was shifted, and the lexeme was returned as a dictionary containing the token type and its corresponding value.

Through this lab, I gained hands-on experience with lexer implementation, understanding the importance of token definition, cursor management, and recursive tokenization methods. This knowledge lays a foundation for further exploration into language processing and compiler design.