Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 3
Course: Formal languages and finite automata
Topic: Lexer & Scanner

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages. The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

Objectives:

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 implementation I chose to use Python, because it is a familiar language.

1. First, I declare some token types. Some of them are:

```
IDENTIFIER = 'IDENTIFIER'
INTEGER = 'INTEGER'
CHAR = 'CHAR'
STRING = 'STRING'
PLUS = 'PLUS'
MINUS = 'MINUS'
MULTIPLY = 'MULTIPLY'
DIVIDE = 'DIVIDE'
ASSIGN = 'ASSIGN'...
```

2. Next, I will define regular expressions for each token using regex:

```
TOKEN_REGEX = [
    (r'[a-zA-Z][a-zA-Z0-9_]*', IDENTIFIER),
    (r'\d+', INTEGER),
    (r'\'[a-zA-Z\s][a-zA-Z\s]+\'', STRING),
    (r'\'[a-zA-Z]\'', CHAR),
    (r'\+=', INCREMENT),
    (r'\+', PLUS),
    (r'-', MINUS),
    (r'\*', MULTIPLY),
    (r'/', DIVIDE),...]
KEY_WORDS = ['for', 'in', 'range', 'if', 'else', 'print', 'while' ]
```

Here is a list of tuples for each token. For example, the IDENTIFIER token in as regular expression is a combination of a character in first place, going on with combinations of chars, digits and underscores. Also, I define a list of predefined key words, which are not formed with regex.

3. Next, I defined a class Token, which will take the type of token and the token itself.

The function of this class is to return the token in form (token_type, token). After this is defined the Lexer class, with the method of getting the next token from the input string.

```
def get_next_token(self):
        while self.pos < len(self.text):
            for pattern, token_type in TOKEN_REGEX:
                regex = re.compile(pattern)
                match = regex.match(self.text, self.pos)
                if match:
                    value = match.group(0)
                    self.pos = match.end()
                    if token_type == SPACE or token_type == NEWLINE:
                        break  # Skip spaces
                    elif value in KEY_WORDS:
                        return Token(value.upper(), value)
                    else:
                        return Token(token_type, value)
            else:
                self.error()

        return Token(EOF, None)
```

Here, with a while loop, it is traversed the input string. Inside, there is a loop which takes the patterns from the initialized REGEX and compile them. After that it searches for a match in the input string. After finding a match, it checks either it is a space or newline and breaks, or if it is a keyword and returns the token in the specified form. If there is not match in the loop, it raises an error as the character is invalid. If the while loop ends, it is returned the token of end of file.

4. In the main function, there are just test strings and tokens are printed in console

Screenshots

```
Token(IDENTIFIER, i)
Token(ASSIGN, =)
Token(INTEGER, 5)
Token(FOR, for)
Token(IDENTIFIER, i)
Token(IN, in)
Token(RANGE, range)
Token(LPAREN, ()
Token(INTEGER, 10)
Token(RPAREN, ))
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, i)
Token(RPAREN, ))
Token(IDENTIFIER, i)
Token(INCREMENT, +=)
Token(INTEGER, 1)
Token(IF, if)
Token(IDENTIFIER, i)
Token(EQUAL, ==)
Token(INTEGER, 10)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, Hello)
Token(RPAREN, ))
Token(ELSE, else)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, World)
Token(RPAREN, ))
```

Figure 1: First input

```
Token(IDENTIFIER, i)
Token(ASSIGN, =)
Token(INTEGER, 0)
Token(WHILE, while)
Token(IDENTIFIER, i)
Token(LESS, <)
Token(INTEGER, 10)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, i)
Token(RPAREN, ))
Token(IDENTIFIER, i)
Token(INCREMENT, +=)
Token(INTEGER, 1)
Token(IF, if)
Token(IDENTIFIER, i)
Token(EQUAL, ==)
Token(INTEGER, 10)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, Hello)
Token(RPAREN, ))
Token(ELSE, else)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, World)
Token(RPAREN, ))
```

Figure 2: Second input

```
Token(IDENTIFIER, age)
Token(ASSIGN, =)
Token(INTEGER, 18)
Token(IF, if)
Token(IDENTIFIER, age)
Token(GREATER_EQUAL, >=)
Token(INTEGER, 18)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(STRING, 'You are an adult')
Token(RPAREN, ))
Token(ELSE, else)
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(STRING, 'You are a child')
Token(RPAREN, ))
```

Figure 3: Third input

Conclusions

In conclusion, this exercise has led to a comprehensive exploration of lexical analysis and the implementation of a lexer for tokenizing source code. By defining token types and corresponding regular expressions, we constructed a lexer capable of identifying various elements in a program, including integers, strings, arithmetic operators, keywords, and punctuation symbols. Through this process, we gained a deeper understanding of the role of lexical analysis in the compilation process and its significance in parsing and interpreting programming languages.

Furthermore, by examining and adjusting the lexer's behavior to correctly identify specific phrases like "You are a child" as a string token, we underscored the importance of accuracy and precision in lexical analysis. This exercise also highlighted the flexibility of lexer implementations in accommodating different language constructs and token types.

Overall, this exercise has provided valuable hands-on experience in building a fundamental component of a compiler or interpreter. The concepts explored here are foundational in understanding language processing systems and their applications in software development, ranging from code analysis and transformation to text processing and data extraction.