Lexer Scanner

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

A lexer, also known as a scanner or tokenizer, plays a critical role in the field of computer programming, particularly in the development of compilers and interpreters. Its primary responsibility is to simplify the complexities of the source code by breaking down the raw text into manageable pieces known as tokens. These tokens serve as the foundational building blocks for the parsing process, enabling a more structured and understandable interpretation of the code.

The process begins when the lexer takes in a sequence of characters from the source code. This stream of characters, which can be anything from letters and digits to punctuation marks and whitespace, is not yet meaningful for the purposes of syntax analysis. The lexer's job is to transform this stream into a sequence of lexemes, which are strings of characters that logically belong together. Each lexeme is then classified into a token, based on predefined rules of the programming language. For instance, a sequence of characters like 'while' might be classified as a keyword token, a numeric sequence like '123' as a literal token, and a pattern like '+=' as an operator token.

The implementation of a lexer can vary greatly depending on the specific needs of the programming language and the design choices of the compiler. In some cases, lexers are handcrafted by programmers, tailored to the unique syntax and semantics of the language. In others, they are generated automatically by tools such as Lex or Flex, which use regular expressions to define the patterns that make up different tokens. These tools provide a more streamlined approach, allowing for rapid development and modification.

However, lexical analysis is not without its challenges. The process must be meticulously designed to handle ambiguities, where a sequence of characters could potentially match more than one token type. The lexer must decide which token type is the correct one, often relying on additional information or lookahead techniques. Furthermore, it must handle whitespace and comments, which can vary widely in their significance from one language to another. In languages like Python, for example, whitespace is crucial to the structure of the code, whereas in languages like C, it is largely insignificant.

Another important aspect of lexical analysis is error handling. The lexer must be capable of recognizing and reporting illegal characters or sequences that do not correspond to any known token pattern. This requires a balance between strictness and flexibility, ensuring that developers receive meaningful feedback without being overwhelmed by trivial errors.

The separation of lexical analysis from parsing offers several advantages. It simplifies the overall design of the compiler, allowing developers to focus on one aspect of the language at a time. This separation also enhances performance, as the lexical analysis can be optimized independently of the parsing process.

Objectives:

- Understand what lexical analysis is.
- Get familiar with the inner workings of a lexer/scanner/tokenizer.
- Implement a sample lexer and show how it works.

Implementation description

```
def lexer(input_string):
    tokens = []
    while input_string:
        match = None
        for token_type, pattern in TOKENS:
            regex = re.compile(pattern)
            match = regex.match(input_string)
            if match:
                value = match.group(0)
                if token_type != 'WHITESPACE':
                    tokens.append((token_type, value))
                input_string = input_string[match.end():]
                break
        if not match:
            print("Invalid character:", input_string[0])
            input_string = input_string[1:]

    return tokens
```

This code defines a function named lexer that takes an input_string as its parameter and uses the token patterns defined in the TOKENS list to tokenize the input. The function iterates over the input string, applying each token's regex pattern to find matches. Non-whitespace tokens are added to a list named tokens, along with their type and value. If no token matches the beginning of the input_string, an error message is printed, indicating an invalid character, and the offending character is removed from the input. The process repeats until the entire input string has been tokenized or an error is encountered. The function ultimately returns the list of tokens.

```
import re

TOKENS = [
    ('NUMBER', r'\d+'),
    ('PLUS', r'\+'),
    ('MINUS', r'-'),
    ('TIMES', r'\*'),
    ('DIVIDE', r'/'),
    ('LPAREN', r'\('),
    ('RPAREN', r'\)'),
    ('WHITESPACE', r'\s+'),
    ('ERROR', r'.'),
]
```

The TOKENS list is defined with pairs of token names and their corresponding regex patterns. These token types include 'NUMBER' for numerical literals, 'PLUS', 'MINUS', 'TIMES', and 'DIVIDE' for arithmetic operators, 'LPAREN' and 'RPAREN' for left and right parentheses, 'WHITESPACE' for space characters, and 'ERROR' for unrecognized characters.

```
input_string = "3 + 4 * 2 / (1 - 5)"
tokens = lexer(input_string)
print(tokens)
```

The expression is passed to the lexer function, which utilizes the previously defined TOKENS to identify and categorize each part of the string as a token. The result, which is a list of tokens representing numbers, operators, and parentheses, is then assigned to the variable tokens.

The output :

[('NUMBER', '3'), ('PLUS', '+'), ('NUMBER', '4'), ('TIMES', '*'), ('NUMBER', '2'), ('DIVIDE', '/'), ('LPAREN', '('), ('NUMBER', '1'), ('MINUS', '-'), ('NUMBER', '5'), ('RPAREN', ')')]

Conclucions :

In conclusion, this laboratory work was designed to deepen our understanding of lexical analysis and to introduce us to the practical workings of a lexer, scanner, or tokenizer. The objectives were clear: grasp the essence of lexical analysis, familiarize ourselves with the inner mechanics of a lexer, and implement a sample lexer to see it in action.

Throughout this lab, we achieved these objectives by dissecting and understanding each component of a lexer. Starting from defining a TOKENS list that represents the vocabulary of our simple language, we moved on to crafting a lexer function in Python that systematically breaks down an arithmetic expression into identifiable tokens based on regular expressions.

By applying our lexer to a sample expression, we not only observed lexical analysis in action but also appreciated the lexer's crucial role in the compilation process: transforming a string of characters into a meaningful sequence of tokens that a parser can understand. This hands-on experience reinforced the theoretical concepts and highlighted the significance of each step in the lexical analysis.

Overall, the lab was not just a coding exercise; it was a comprehensive learning experience that bridged theoretical knowledge with practical application, enhancing our understanding of the initial stages of the compilation process and preparing us for more advanced topics in computer science.

References

1. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
2. https://medium.com/@pythonmembers.club/building-a-lexer-in-python-a-tutorial-3b6de161fe84
3. https://www.youtube.com/watch?v=BI3K-ME3L74
4. Theory lessons from else