Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Gîncu Olivia

Git Repository: https://github.com/livia994/LFALab

Overview

A lexer (also known as a tokenizer or scanner) **and a scanner** are both components of a compiler or interpreter responsible for the initial phase of processing source code. They play crucial roles in converting raw source code into meaningful tokens that can be further analyzed by the parser.

Lexer/Tokenizer:

A lexer, often referred to as a tokenizer, is the first stage of a compiler or interpreter. Its primary function is to break down the source code into smaller meaningful units called tokens. These tokens represent the smallest syntactic elements of the programming language, such as keywords, identifiers, literals, operators, and punctuation.

Here's how a lexer typically works:

1. **Scanning:** The lexer scans the input source code character by character, recognizing patterns and identifying tokens.

2. **Tokenization:** As it scans the input, the lexer groups characters into tokens based on predefined rules and regular expressions. For example, it identifies keywords like `if`, `while`, and `for`, literals like integers or strings, and identifiers like variable names.

3. **Token Output:** After identifying a token, the lexer outputs it along with its corresponding token type and possibly other metadata. These tokens are passed on to the parser for further analysis.

4. **Error Handling:** The lexer may also handle lexical errors, such as encountering characters that do not fit any defined token pattern, by producing error messages.

Scanner:

The term "scanner" is often used interchangeably with lexer or tokenizer, referring to the same concept of breaking down source code into tokens. In some contexts, however, "scanner" may refer specifically to the part of the lexer responsible for reading characters from the source file or input stream and passing them to the tokenization component.

Key Differences:

While lexer and scanner are often used synonymously, some distinctions might be made in specific contexts:

1. Scope: The term "lexer" is more commonly used in the context of programming language processing, while "scanner" may be used more broadly in various contexts where input needs to be tokenized.

2. Functionality: Conceptually, they perform the same task of tokenization, but in some cases, "scanner" might be used to emphasize the role of reading characters from input, whereas "lexer" might be used to emphasize the tokenization process itself.

To sum up, both a lexer and a scanner are essential components of a compiler or interpreter, responsible for breaking down the source code into tokens, which are then used by subsequent stages of the compilation process for further analysis and processing.

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 tips:

I have implemented a Lexer class, that is responsible for tokenizing the input text by breaking it down into meaningful tokens. It utilizes methods such as advance(), skip_whitespace(), and integer() to iterate through the input text and identify tokens based on predefined rules. The advance() method is used to progress through the input text by moving to the next character. It updates the current character being processed by the lexer.

```
def advance(self):
    self.pos += 1
    if self.pos < len(self.text):
        self.current_char = self.text[self.pos]
    else:
        self.current_char = None
```

The skip_whitespace() method ensures that whitespace characters in the input text are ignored during tokenization. By iterating through the input text, this method moves past whitespace characters until a non-whitespace character is encountered.

```
def skip_whitespace(self):
    while self.current_char is not None and self.current_char.isspace():
        self.advance()
```

The integer() method is responsible for identifying and extracting integer tokens from the input text. It iterates through consecutive digits in the input text, collecting them to form an integer token. These methods collectively enable the lexer to iterate through the input text, identify tokens based on predefined rules, and construct meaningful tokens for further processing or analysis.

```
def integer(self):
    result = ""
    while self.current_char is not None and self.current_char.isdigit():
        result += self.current_char
        self.advance()
    return int(result)
```

Additionally, the Lexer class contains a get_next_token() method. This method iterates through the input text, examining each character to identify and construct tokens based on predefined rules. It recognizes tokens such as integers and specific characters representing operators like plus and minus. If the current character is a whitespace, the method utilizes skip_whitespace() to ignore it and move to the next non-whitespace character. When the end of the input text is reached, the method returns an EOF (End of File) token to signify the completion of tokenization.

```
def get_next_token(self):
    while self.current_char is not None:
        if self.current_char.isspace():
            self.skip_whitespace()
            continue
        if self.current_char.isdigit():
            return Token( type: "INTEGER", self.integer())
        if self.current_char == "+":
            self.advance()
            return Token( type: "PLUS",  value: "+")
        if self.current_char == "-":
            self.advance()
            return Token( type: "MINUS",  value: "-")
        self.error()
    return Token( type: "EOF",  value: None)
```

Output/Conclusion:

For the input: *text = "8 + 22 + 0"*

```
Token(INTEGER, 8)
Token(PLUS, '+')
Token(INTEGER, 22)
Token(PLUS, '+')
Token(INTEGER, 0)
```

For the input: *text = "122 - 10 + 3"*

```
Token(INTEGER, 122)
Token(MINUS, '-')
Token(INTEGER, 10)
Token(PLUS, '+')
Token(INTEGER, 3)
```

Throughout this laboratory work, we explored the fundamental concept of lexical analysis and its practical implementation through the development of a lexer/scanner in Python. Lexical analysis, a pivotal stage in the compilation process of programming languages, involves the breakdown of source code into meaningful tokens. By dissecting the input text, lexers facilitate subsequent stages of language processing, including syntax and semantic analysis.

Lexers and scanners serve as indispensable components in the compilation process, enabling the transformation of raw source code into structured tokens. These tokens serve as the foundation for syntactic and semantic analysis, aiding in the detection of errors and the verification of program correctness. Furthermore, efficient lexers contribute to the optimization of compilers and interpreters, enhancing their performance and efficiency.

Beyond compilation, lexers and scanners find applications in various language processing tools and systems. They form the backbone of syntax highlighters, code editors, static analyzers, and other tools used in software development. By providing accurate and efficient tokenization, lexers enable the development of robust and effective language processing solutions.

References:

[1] [A sample of a lexer implementation](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html)

[2] [Lexical analysis](https://en.wikipedia.org/wiki/Lexical_analysis)