Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Popov Nichita

Theory

In the realm of computer science, lexical analysis, or tokenization, plays a pivotal role in the interpretation and compilation of programming languages. This process, facilitated by a lexer or scanner, involves dissecting a string of characters into tokens, which are categorized units of meaning that conform to the syntax of the language. Unlike lexemes, which are the raw textual fragments identified by the lexer, tokens are enriched with a layer of semantic meaning, categorizing each fragment under a specific type or class based on predefined language rules. This distinction is crucial for the subsequent phases of parsing and semantic analysis in compilers and interpreters, enabling a structured and efficient processing of source code.

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

The implementation focuses on constructing a simple yet functional lexer capable of tokenizing a basic arithmetic expression. The project is structured into several core components:

- `Lexer` Class: The heart of the lexer, responsible for navigating through the input text, character by character, and extracting tokens based on matching patterns and rules. The class identifies different token types such as integers, arithmetic operators, and parentheses, allowing for the tokenization of simple arithmetic expressions.

Method `get_next_token` used for tokenization:

```
def get_next_token(self):
    """Tokenize the input text by recognizing tokens one at a time."""
    while self.current_char is not None:
        if self.current_char.isspace():  # Skip whitespace
            self.advance()
            continue
```

```
            if self.current_char.isdigit():  # Handle integers
                return Token(TokenType.INTEGER, self.integer())

            if self.current_char == '+':  # Plus token
                self.advance()
                return Token(TokenType.PLUS)

            if self.current_char == '-':  # Minus token
                self.advance()
                return Token(TokenType.MINUS)

            if self.current_char == '(':  # Left parenthesis
                self.advance()
                return Token(TokenType.LPAREN)

            if self.current_char == ')':  # Right parenthesis
                self.advance()
                return Token(TokenType.RPAREN)

            self.error()  # If none match, raise an error

        # Once we're done with all characters, return an EOF token
        return Token(TokenType.EOF)

    def error(self):
        """Raise an exception if an invalid character is encountered."""
        raise Exception('Invalid character')
```

- `Token` and `TokenType` Class: These classes define the structure of tokens and the possible types of tokens the lexer can recognize. The `TokenType` class uses an enumeration to define token types, including `INTEGER`, `PLUS`, `MINUS`, `LPAREN`, and `RPAREN`, which represent numeric values, addition and subtraction operators, and parentheses, respectively.

`TokenType` Class:

```
from enum import Enum, auto

# Define the types of tokens that our lexer can recognize.
class TokenType(Enum):
    INTEGER = auto()  # Represents an integer
    PLUS = auto()     # Represents a plus sign '+'
    MINUS = auto()    # Represents a minus sign '-'
    LPAREN = auto()   # Represents a left parenthesis '('
    RPAREN = auto()   # Represents a right parenthesis ')'
    EOF = auto()      # Represents the end of the input
```

`Token` Class:

```
class Token:
    # A token consists of a type and an optional value.
    def __init__(self, type, value=None):
        self.type = type  # The type of the token (from TokenType)
        self.value = value  # The value of the token (relevant for integers)

    # Representation of the Token instance for debugging and testing.
    def __repr__(self):
        return f"Token({self.type}, {repr(self.value)})"
```

- `main.py` Entry Point: Demonstrates the lexer's capabilities by tokenizing a sample arithmetic expression. The program iterates through each token extracted by the lexer, printing it to the console, thereby showcasing the lexer's ability to break down and categorize parts of the expression.

Results

For example I took a simple expression as a string input:

```
3 + 5 - ( 7 - 2 )
```

And as the uotput I got:

```
Token(TokenType.INTEGER, 3)
Token(TokenType.PLUS, None)
Token(TokenType.INTEGER, 5)
Token(TokenType.MINUS, None)
Token(TokenType.LPAREN, None)
Token(TokenType.INTEGER, 7)
Token(TokenType.MINUS, None)
Token(TokenType.INTEGER, 2)
Token(TokenType.RPAREN, None)
```

Conclusion

This project underscores the fundamental role of lexical analysis in the processing of programming languages. Through the practical task of implementing a lexer, we gain a deeper understanding of how compilers and interpreters begin to interpret and compile code. While the scope of this lexer is limited to basic arithmetic expressions, the concepts and techniques explored are applicable to a wide range of programming languages and contexts. This endeavor not only achieves its educational objectives but also paves the way for further exploration into the intricacies of language processing and compiler construction.

References

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html
- https://en.wikipedia.org/wiki/Lexical_analysis

- https://www.youtube.com/watch?v=dQw4w9WgXcQ