Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Ernu Catalina

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:

- Understand what lexical analysis [1] is.
- Get familiar with the inner workings of a lexer/scanner/tokenizer.
- Implement a sample lexer and show how it works.

Lexer Class:

Purpose: The purpose of the Lexer class is to tokenize input strings, breaking them down into individual elements called tokens.

Functionality: The Lexer class facilitates the segmentation of input strings into individual tokens, a pivotal task in language processing and compiler development. Initialized with an input string, it iterates through each character, identifying its type (digit, operator, or parenthesis) and constructing corresponding tokens. By skipping whitespace and raising errors for unrecognized characters, the lexer ensures robust tokenization. The resulting tokens provide a structured representation of the input, serving as the foundation for subsequent language analysis and compilation processes.

Code Snippet:

```
class Lexer:
    def __init__(self, input):
        self.input = input
        self.index = 0
```

```
    def is_digit(self, char):
        return char.isdigit()

    def is_operator(self, char):
        return char in ['+', '-', '*', '=', '/']

    def next_token(self):
        while self.index < len(self.input):
            current_char = self.input[self.index]

            if current_char.isspace():
                self.index += 1
                continue

            if self.is_digit(current_char):
                num = ""
                while self.index < len(self.input) and
self.is_digit(self.input[self.index]):
                    num += self.input[self.index]
                    self.index += 1
                return Token("INTEGER", int(num))

            if self.is_operator(current_char):
                self.index += 1
                return Token("OPERATOR", current_char)

            if current_char == '(':
                self.index += 1
                return Token("L_PAREN", current_char)

            if current_char == ')':
                self.index += 1
                return Token("R_PAREN", current_char)

            raise ValueError(f"Unknown token: {current_char}")

        return Token("EOF", "")

def tokenize(self):
        tokens = []
        next_token = self.next_token()
        while next_token.type != "EOF":
            tokens.append(next_token)
            next_token = self.next_token()
        return tokens
```

Token Class:

Purpose: The purpose of the Token class is to encapsulate individual elements parsed by the lexer, providing a structured representation with a type and value pair.

Functionality: The Token class encapsulates individual elements parsed by the lexer, comprising a type and value pair. With its constructor method, it initializes tokens with specific types and corresponding values. This class acts as the fundamental building block for representing different components of the input string as tokens, enabling clear categorization and processing in subsequent stages of language analysis and compilation.

Code Snippet:

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value
```

Conclusions

In conclusion, this laboratory exercise has been instrumental in illuminating the essential concepts of lexical analysis through the creation of a lexer in Python. By meticulously crafting a Token class to encapsulate token types and values and implementing a robust Lexer class capable of parsing input strings, we've gained invaluable insights into the foundational components of language processing. Through methodical identification of digits, operators, and parentheses, our lexer showcases the intricacies of tokenization, laying the groundwork for understanding more complex parsing techniques.

Furthermore, the practical application of our lexer in tokenizing user-provided arithmetic expressions underscores its significance in real-world scenarios. By seamlessly transforming raw input into structured tokens, we've witnessed the critical role of lexical analysis in the compilation process. This laboratory experience not only equips us with practical skills in building lexers but also fosters a deeper appreciation for the underlying principles of language processing and compiler construction, paving the way for continued exploration and experimentation in this dynamic field.

References

[1] https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html