# Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Ernu Catalina

## Theory:

Lexical analysis, a fundamental stage in the compilation or interpretation of programming, markup, or other languages, involves breaking down a sequence of characters into meaningful units known as tokens or lexemes. Often referred to interchangeably as a lexer, tokenizer, or scanner, this component is tasked with parsing the input stream and generating a stream of tokens or lexemes that represent the building blocks of the language being processed.

At its core, a lexeme is the smallest meaningful unit identified during lexical analysis. It comprises a sequence of characters from the source code that matches the pattern for a token. These lexemes are typically obtained by segmenting the input stream based on delimiters such as spaces, tabs, or punctuation marks. However, they lack semantic categorization at this stage and represent raw fragments of the code.

On the other hand, tokens are the result of categorizing lexemes according to the language's rules and grammar. Each token represents a specific type of construct within the language, such as keywords, identifiers, operators, literals, or punctuation symbols. By associating lexemes with predefined categories or names, tokens provide semantic meaning to the components of the code.

For instance, consider the expression x = 5 + 3. Lexical analysis of this expression would yield lexemes such as x, =, 5, +, and 3. However, it is the subsequent tokenization process that assigns meaning to these lexemes, categorizing them as Identifier, Assignment operator, Integer literal, and Addition operator, respectively. This distinction between lexemes and tokens is crucial, as it allows for the differentiation between the raw character sequences and their categorized representations within the language's syntax.

In summary, the lexer's role extends beyond simple character parsing; it transforms a stream of characters into structured tokens that encapsulate the semantic essence of the code. By delineating between lexemes and tokens, lexical analysis lays the groundwork for subsequent stages of compilation or interpretation, enabling accurate processing and execution of the language's instructions.

# 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:

The Lexer class plays a vital role in language processing by facilitating the tokenization process, which involves breaking down input strings into meaningful units known as tokens.

**Initialization:**
Upon instantiation, the Lexer class is equipped with the input string provided by the user, along with an index to keep track of the current position within that string. This index serves as a crucial tool for navigating through the input string character by character, enabling systematic processing.

**Tokenization:**
The next_token() method lies at the heart of the tokenization process. With each invocation, it meticulously traverses the input string, character by character. At each step, the method scrutinizes the type of the current character, discerning whether it represents a digit, operator, parenthesis, or another entity. This determination is made possible through the aid of auxiliary methods like is_digit() and is_operator(), which provide invaluable assistance in identifying the nature of each character.

Upon recognizing a valid character, the next_token() method constructs a corresponding token, encapsulating both its type and value. However, the method is also mindful of whitespace characters, promptly bypassing them during the tokenization process to ensure they do not interfere with the formation of tokens.

**Code Snippet:**

```
def next_token(self):
   while self.index < len(self.input):
       current_char = self.input[self.index]
       if current_char.isspace():
           ...
       if self.is_digit(current_char):
          ...
       if self.is_operator(current_char):
           ...
       if current_char == '(':
           ...
       if current_char == ')':
           ...
       raise ValueError(f"Unknown token: {current_char}")
   return Token("EOF", "")
```

**Error Handling:**
If the lexer encounters a character that it cannot recognize or classify, it raises a ValueError, as demonstrated in the code snippet above. This deliberate handling of unknown characters ensures the robustness of the tokenization process by preventing the inclusion of ambiguous or undefined tokens in the stream. By signaling the presence of an unrecognized token, the lexer maintains the integrity of the token stream, thereby enhancing the reliability and accuracy of subsequent language processing tasks.

**Tokenizing Process:**
The tokenize() method acts as the conductor of the tokenization process. It oversees the sequential extraction of tokens by invoking the next_token() method until it reaches the end of the input string, denoted by EOF (End of File). During each iteration, the method generates a token representing the current character or sequence of characters, categorizing them based on their type and value. These tokens are then systematically accumulated into a list, forming the structured representation of the input string. Upon completion, this list of tokens is returned as the output of the tokenization process, serving as the foundation for further language analysis and compilation tasks. Through its orchestration of token extraction and aggregation, the tokenize() method plays a pivotal role in transforming raw input strings into structured token streams, facilitating comprehensive language processing and understanding.

**Code Snippet:**

```
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:

The Token class functions as a receptacle for individual tokens that have been parsed by the lexer, offering a structured representation characterized by a type and value pair.

**Initialization:**
Upon instantiation, each instance of the Token class is endowed with specific attributes denoting its type and value. These attributes are paramount in categorizing the token according to its function within the language syntax and preserving its associated content.

By meticulously defining the type and value of each token upon creation, the Token class facilitates clear categorization and storage of tokens parsed by the lexer. This structured representation ensures that each token is imbued with its distinct identity, enabling seamless integration into subsequent language analysis and compilation processes.

**Code Snippet:**

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

### The Logic Behind the Code:

**Tokenization Logic:**
The lexer systematically processes each character within the input string, determining its nature—whether it's a digit, operator, parenthesis, or another type—by leveraging auxiliary methods such as is_digit() and is_operator(). Upon recognition of a valid character, the lexer constructs a Token object tailored to its type and value. Throughout this process, whitespace characters are disregarded, ensuring they do not influence the tokenization. However, should the lexer stumble upon an unfamiliar character, it promptly raises a ValueError, thereby signaling the presence of an unrecognized token and ensuring the robustness of the tokenization process.

**Token Construction:**
Token objects serve as containers for tokens identified and parsed by the lexer. Each token is characterized by a type-value pair, encapsulating both its categorization and content. This structured representation enables straightforward categorization and subsequent processing in further stages of language analysis.

**Token Collection:**
Throughout the tokenization process, all tokens generated by the lexer are systematically accumulated into a list. This list acts as the organized representation of the input string, laying the groundwork for subsequent language analysis and compilation stages. By capturing each token within this list, the lexer ensures a structured foundation upon which further processing can be built, facilitating comprehensive language understanding and compilation tasks.

## 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