Laboratory Work Nr.3

Topic: Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Tofan Liviu

Group: FAF-223

Chişinău – 2024

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.

A tokenizer is in charge of preparing the inputs for a model. Tokenization is the process of creating a digital representation of a real thing. Tokenization can be used to protect sensitive data or to efficiently process large amounts of data. In general, tokenization is the process of issuing a digital, unique, and anonymous representation of a real thing. In Web3 applications, the token is used on a (typically private) blockchain, which allows the token to be used within specific protocols. Tokens can represent assets, including physical assets like real estate or art, financial assets like equities or bonds, intangible assets like intellectual property, or even identity and data. In AI applications, tokenization works in a different way, by enabling large language models (LLMs) that use deep learning techniques to process, categorize, and link pieces of information—from whole sentences down to individual characters. And payment tokenization protects sensitive data by generating a temporary code that's used in place of the original data. Tokenization can create several types of tokens. From the financial-services industry, one example would be stablecoins, a type of cryptocurrency pegged to real-world money designed to be fungible, or replicable. Another type of token is an NFT—a nonfungible token, meaning a token that can't be replicated—which is a digital proof of ownership people can buy and sell. Yet another example could simply be the word "cat"; an LLM would tokenize the word "cat" and use it to understand relationships between "cat" and other words.

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 tips:

Implementation

I defined a Tokenizer class that is responsible for tokenizing source code based on predefined token patterns. I used RegEx library to check if a string contains the specified search pattern. Tokenization is the process of breaking a sequence of characters into meaningful pieces called tokens. Tokens typically represent individual words, numbers, symbols, or other significant elements in the source code.

The re library in Python provides functions and methods for working with regular expressions. In this code, the re.compile() function is used to compile regex patterns into pattern objects, which are then used to match against the source code text. Additionally, the match() method of pattern objects is utilized to find matches at the beginning of the text.

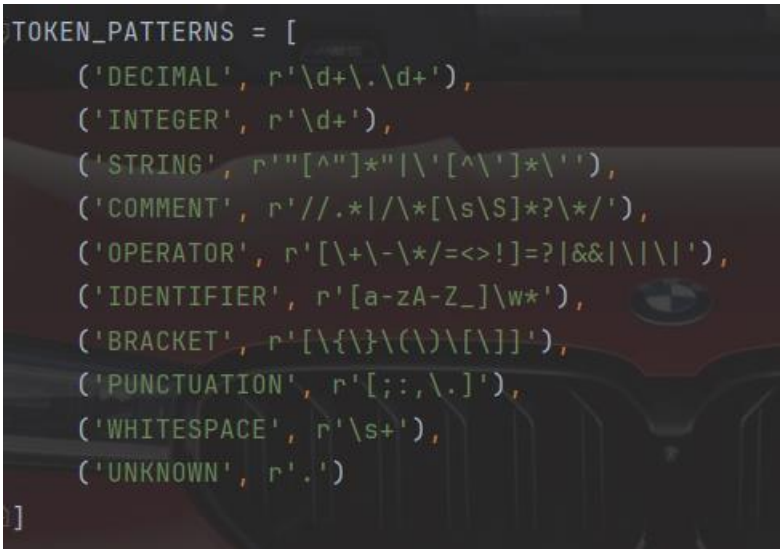

Figure 1. Token Patterns

Token patterns serve as templates that define the structure of different types of tokens within the source code. Each pattern consists of a regular expression, which is a sequence of characters that defines a search pattern. These regular expressions are matched against the source code text to identify and extract tokens of specific types. By associating each pattern with a token name, the tokenizer can categorize the extracted tokens based on their respective types. This categorization enables the tokenizer to differentiate between integers, decimals, strings, comments, operators, identifiers, brackets, punctuation marks, whitespace, and unknown characters present in the source code. Regular expressions provide a flexible and powerful mechanism for describing complex patterns within text data. They allow for the specification of various token formats and structures, accommodating the diverse syntaxes and conventions of programming languages. The token patterns defined in the tokenizer encompass a wide range of lexical elements commonly found in programming languages. This comprehensive coverage ensures that the tokenizer can effectively handle different types of source code, regardless of language or syntax, by accurately identifying and tokenizing each constituent element.

Token Patterns:

DECIMAL: Matches floating-point numbers.

INTEGER: Matches integers.

STRING: Matches strings enclosed in either single or double quotes.

COMMENT: Matches single-line and multi-line comments.

OPERATOR: Matches various operators.

IDENTIFIER: Matches identifiers (variable/function names) adhering to Python naming conventions.

BRACKET: Matches various types of brackets.

PUNCTUATION: Matches punctuation symbols.

WHITESPACE: Matches spaces, tabs, and newlines.

UNKNOWN: Matches any other characters not covered by the above patterns.

Figure 2. Tokenizer Class and tokenize() function.

Initialization: The ‘Tokenizer’ class initializes with the source code text and an empty list to store the tokens.

Tokenize Method: The tokenize() method is responsible for tokenizing the source code. It iterates over the source code text until there are no characters left. For each iteration, it matches the current text against each token pattern using regular expressions.

- If a match is found, it extracts the matched value and token name, then adds the token to the list of tokens, unless it's whitespace or a comment.
- If no match is found for the current text, it raises a TokenizerError indicating an unknown token.

Furthermore, the Tokenizer class and its tokenize() method serve as foundational components in the process of source code analysis and interpretation. They provide a crucial mechanism for breaking down complex code structures into manageable tokens, facilitating subsequent stages of code parsing, interpretation, compilation, or transformation.

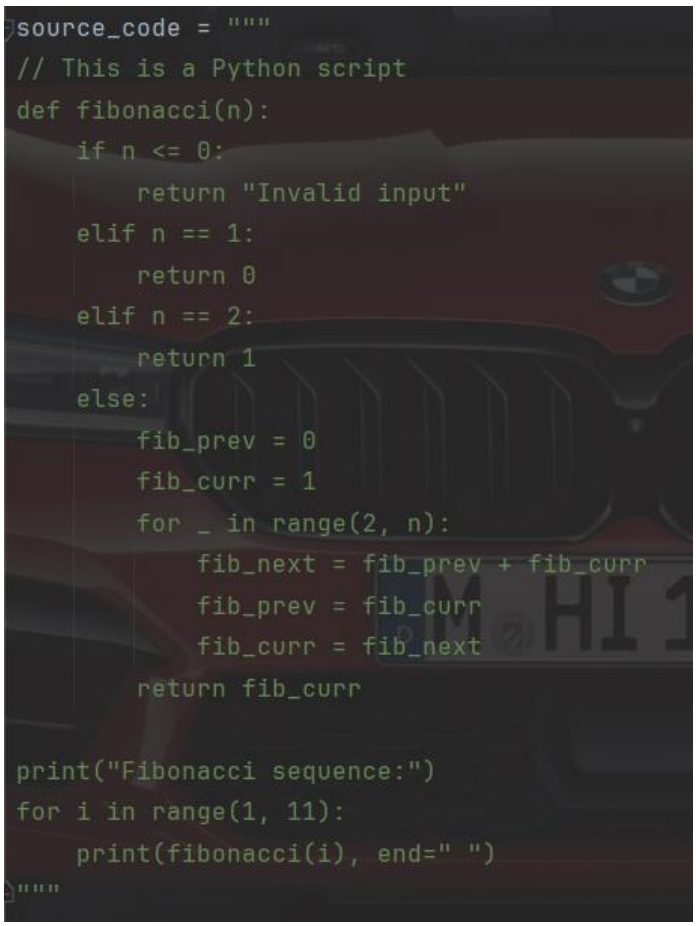

Figure 3. Source Code

The provided source code is an example input to be tokenized. It includes a Python script containing comments, function definitions, conditional statements, loops, and print statements.

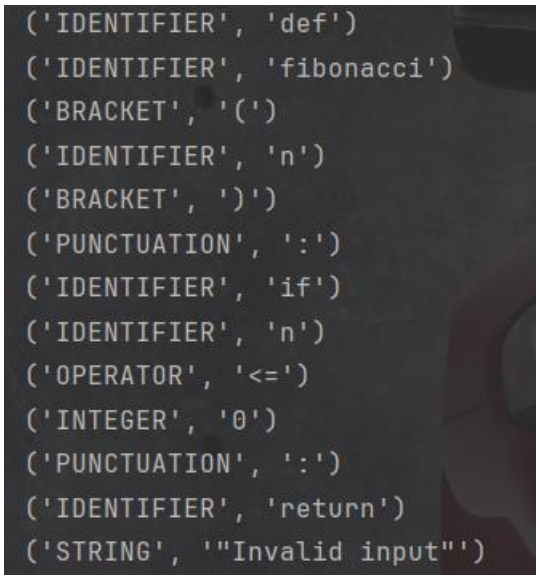

Figure 3. Part of Output

Conclusion

In conclusion, the presented code implements a versatile tokenizer using regular expressions, facilitating the extraction of tokens from source code. This tokenizer, encapsulated within the `Tokenizer` class, is adept at handling various token types such as decimals, integers, strings, comments, operators, identifiers, brackets, punctuation, whitespace, and unknown characters. The iterative tokenization process, driven by the `tokenize()` method, ensures comprehensive coverage of the source code, enabling precise identification and classification of tokens.

The significance of a lexer or tokenizer in programming cannot be overstated. These components serve as fundamental building blocks in software development, aiding in tasks ranging from syntax highlighting and code analysis to compiler construction and natural language processing. By breaking down complex source code into manageable tokens, lexers and tokenizers facilitate subsequent stages of code processing, interpretation, and transformation.

The implementation provided here offers a generic and adaptable solution for tokenizing source code across various programming languages and contexts. Its modular design allows for easy extension and customization to accommodate specific language features or tokenization requirements. Moreover, the capability to tokenize code snippets, as demonstrated with the given source code input, underscores the versatility and applicability of the tokenizer in real-world scenarios.

Moving forward, the code can be further refined and augmented with additional token patterns or enhancements to handle more complex language constructs. Additionally, it can serve as a foundational component in the development of sophisticated software tools, IDE plugins, or language processing frameworks aimed at improving developer productivity, code quality, and language understanding.