Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 3
Course: Formal languages and finite automata
Topic: Lexer and Scanner

Elaborated:
st. gr. FAF-221 Călugăreanu Ana

Verified:
asist. univ. Cretu Dumitru

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.

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 Description

In this laboratory exercise, you created a Lexer, also known as a lexical analyzer, for mathematical expressions. The Lexer is designed to read through a string of characters (source code) and convert it into a series of tokens represented by the `Symbol` class, each classified according to predefined types in the `SymbolType` enumeration. This process involves identifying numbers, arithmetic operators, parentheses, and handling unknown or end-of-input scenarios, facilitating the parsing and interpretation of mathematical expressions. By methodically advancing through the input and categorizing each character or sequence of characters, your Lexer lays the foundational groundwork for any subsequent parsing or evaluation stages, such as syntax analysis or expression evaluation. This exercise demonstrates the initial step in compiling or interpreting code, focusing on breaking down and understanding the structure of the input.

SymbolType Enumeration:

```python
class SymbolType(str, Enum):
    NUMBER = "NUMBER"
    ADD = "ADD"
    SUBTRACT = "SUBTRACT"
    MULTIPLY = "MULTIPLY"
    DIVIDE = "DIVIDE"
    OPEN_BRACE = "OPEN_BRACE"
    CLOSE_BRACE = "CLOSE_BRACE"
    END = "END"
    UNKNOWN = "UNKNOWN"
```

The SymbolType class defines different types of symbols that can be encountered in mathematical expressions. It inherits from Python's Enum class for distinct symbol representation and from str for easy string comparison and readability. The symbols include types for numeric values, arithmetic operators, parenthesis for grouping, a special end-of-input marker, and a type for any unrecognized characters. This enumeration ensures type safety and clarity over using plain strings throughout the analysis process. By defining explicit symbol types, the code helps prevent bugs related to typo errors in string literals and improves the maintainability of the lexer.

Symbol Class:

```python
class Symbol:
    Ana Calugareanu
    def __init__(self, kind, value=None):
        self.kind = kind
        self.value = value

    Ana Calugareanu
    def __str__(self):
        return f"Symbol({self.kind}, {self.value})"

    Ana Calugareanu
    def __repr__(self):
        return self.__str__()

    Ana Calugareanu
    def equals(self, other):
        return self.kind == other.kind and self.value == other.value
```

This section introduces the Symbol class, which represents individual elements (tokens) of the source code being analyzed. The __init__ method initializes a Symbol with a type (kind) and an optional value, which is particularly useful for numeric symbols. The __str__ and __repr__ methods ensure that when a Symbol instance is printed or logged, it presents a clear and unambiguous representation of itself, making debugging easier. The equals method provides a way to compare two Symbol instances, checking if both their kind and value match, essential for syntax analysis and symbol matching.

Analyser Class Initialization:

```
class Analyser:
    Ana Calugareanu
    def __init__(self, source_code):
        self.source = source_code
        self.pointer = 0
        self.current = self.update_current()
```

The Analyser class is designed to parse and tokenize a string of source code. Its constructor takes the source code as input and initializes the pointer to the start of this source, setting the current character to be the first character in the source code. This setup is crucial for the lexical analysis process, as it sets the stage for sequential character examination and symbol extraction.

update_current method:

```
def update_current(self):
    if self.pointer >= len(self.source):
        return None
    return self.source[self.pointer]
```

Updates the current character that the analyzer is examining. If the pointer is beyond the end of the source code, it returns None; otherwise, it returns the character at the current pointer position.

move_forward method:

```
def move_forward(self):
    self.pointer += 1
    self.current = self.update_current()
```

Advances the pointer by one position and updates the current character accordingly. This method is essential for progressing through the source code character by character.

ignore_space method:

```
def ignore_space(self):
    while self.current and self.current.isspace():
        self.move_forward()
```

Skips over any whitespace characters in the source code. As long as the current character is a space, the method continues to advance the pointer, effectively ignoring spaces, tabs, and newline characters.

collect_number method:

```
def collect_number(self):
    number = ''
    while self.current and self.current.isdigit():
        number += self.current
        self.move_forward()
    return int(number)
```

Aggregates consecutive digit characters into a single integer value. This method is crucial for parsing numeric values from the source code, ensuring that numbers are correctly identified and extracted as whole entities.

next_symbol method:

```
def next_symbol(self):
    while self.current:
        if self.current.isspace():
            self.ignore_space()
            continue

        if self.current.isdigit():
            return Symbol(SymbolType.NUMBER, self.collect_number())

        if self.current == "+":
            self.move_forward()
            return Symbol(SymbolType.ADD)

        if self.current == "-":
            self.move_forward()
            return Symbol(SymbolType.SUBTRACT)

        if self.current == "*":
            self.move_forward()
            return Symbol(SymbolType.MULTIPLY)
```

```
        if self.current == "/":
            self.move_forward()
            return Symbol(SymbolType.DIVIDE)

        if self.current == "(":
            self.move_forward()
            return Symbol(SymbolType.OPEN_BRACE)

        if self.current == ")":
            self.move_forward()
            return Symbol(SymbolType.CLOSE_BRACE)

        # For unknown characters
        self.move_forward()
        return Symbol(SymbolType.UNKNOWN, self.current)

    return Symbol(SymbolType.END)
```

Determines and returns the next symbolic representation from the source code, based on the current character. It handles different types of symbols, including numbers, arithmetic operators, and parentheses, by recognizing each character or sequence of characters and converting them into a corresponding Symbol.

extract_symbols method :

```
def extract_symbols(self):
    symbols = []
    while (symbol := self.next_symbol()).kind != SymbolType.END:
        symbols.append(symbol)
    symbols.append(symbol)
    return symbols
```

Uses the next_symbol method to tokenize the entire source code into a sequence of Symbol objects until an 'END' symbol is encountered. This method effectively converts the entire source code into a list of lexical tokens, which can be used for further analysis or interpretation.

Results

In this laboratory exercise, the Lexer successfully transformed a given string of mathematical expression into a sequence of lexical tokens, each represented as instances of the `Symbol` class. The tokens include various types such as numbers, arithmetic operators, and parentheses, accurately reflecting the structure and components of the original expression. This tokenization process is a crucial first step in understanding and interpreting the expression, setting the stage for further syntactic and semantic analysis. The results demonstrate the Lexer's ability to methodically analyze and classify each part of the input, highlighting its effectiveness in parsing and preparing mathematical expressions for computation or evaluation.

```
# Example
code_to_analyze = "3 + 4 * (2 - 1)"
analyzer = Analyser(code_to_analyze)
extracted_symbols = analyzer.extract_symbols()
for sym in extracted_symbols:
    print(sym)
```

The output :

```
Symbol(NUMBER, 3)
Symbol(ADD, None)
Symbol(NUMBER, 4)
Symbol(MULTIPLY, None)
Symbol(OPEN_BRACE, None)
Symbol(NUMBER, 2)
Symbol(SUBTRACT, None)
Symbol(NUMBER, 1)
Symbol(CLOSE_BRACE, None)
Symbol(END, None)
```

Conclusions

In conclusion, this laboratory session was instrumental in understanding the principles and mechanisms of lexical analysis, particularly through the implementation and examination of a Lexer for mathematical expressions. The exercise provided hands-on experience in designing and coding a Lexer that systematically converts a string of characters into a sequence of tokens, which are essential for further stages of compilation or interpretation, such as parsing. The challenges encountered, such as ensuring correct tokenization and handling various types of symbols, underscored the importance of meticulous design and testing in the development of language processing tools. The successful completion of the lab reinforces the foundational role of lexical analysis in the broader context of compiler construction and language processing, laying the groundwork for deeper exploration into syntactic and semantic analysis in future sessions.