Laboratory Work No. 3

Course: Formal Languages & Finite Automata

Student: Iordan Liviu, Group: FAF-223

Topic: Lexer and Scanner

Overview

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.

Lexical Considerations

The design of the lexer involved careful consideration of the following aspects:

- **Language Constructs:** The lexer was designed to recognize the specific keywords, operators, delimiters, and identifiers pertinent to the target language (in your case, Python). This involved defining regular expressions and patterns to accurately identify these elements.
- **Whitespace Handling:** A decision was made to treat whitespace as a delimiter, but not create tokens for it. This is common, as whitespace primarily serves to separate tokens rather than holding semantic meaning within the code.
- **Number Representation:** The lexer accommodates both integer and floating-point numbers, using regular expressions to distinguish and categorize them appropriately.
- **String Handling:** String literals are enclosed within quotation marks, and the lexer includes mechanisms to handle potential escape sequences within strings.
- **Comments:** The lexer recognizes the '#' character as the start of a single-line comment and ignores content until the end of the line. This prevents comments from interfering with the tokenization process.
- **Error Handling:** The lexer incorporates error handling mechanisms to raise informative exceptions when encountering characters or patterns that do not conform to the language's specifications. This helps maintain the integrity of the tokenization process during unexpected input.

Description Implementation:

This file (**main.py**) serves as the entry point of the program and the place where all the tests are located.

- The init function for the Lexer class:

```
def __init__(self, input_text):
    self.input_text = input_text
    self.tokens = []  # Tokens will be stored here
```

```
    self.current_pos = 0


    # Define keywords, operators,
    # identifiers, etc.
    self.keywords = {'if', 'else', 'while', 'for', 'int', 'float'}
    self.operators = {'+', '-', '*', '/', '=', '<', '>', '==', ':'}
    self.delimiters = {';', '(', ')', '{', '}', ',', "'"}
```

The __init__ function serves as the constructor for the Lexer class. Its primary responsibilities include:

- **Initialization:** It sets up the essential structures for the lexer:
    - self.input_text: Stores the code input that needs to be analyzed.
    - self.tokens: A list to hold the tokens that will be generated during analysis.
    - self.current_pos: A pointer to keep track of the lexer's position within the input code.
- **Defining Lexical Categories:** It establishes sets of characters that represent language elements:
    - self.keywords: A set of reserved words in the programming language.
    - self.operators: A set of mathematical and logical operators.
    - self.delimiters: A set of punctuation and structural symbols.

In Summary: The __init__ function prepares the lexer to break down the input code into meaningful units (tokens) by initializing variables and defining the sets of characters that have significance within the language.

- The tokenize function which sets the conditions food the character

```
def tokenize(self):
```

```
  while self.current_pos < len(self.input_text):
      char = self.input_text[self.current_pos]

      if char.isspace():
          self.current_pos += 1  # Skip whitespace

      elif re.match(r'[0-9]', char):  # Numeric literal
          number = self._read_number()
          self.tokens.append(('NUMBER', number))

      elif re.match(r'[a-zA-Z]', char):  # Identifier or keyword
          word = self._read_word()
          if word in self.keywords:
              self.tokens.append(('KEYWORD', word))
          else:
              self.tokens.append(('IDENTIFIER', word))

      elif char in self.operators:
          self.tokens.append(('OPERATOR', char))
          self.current_pos += 1

      elif char in self.delimiters:
          self.tokens.append(('DELIMITER', char))
          self.current_pos += 1

      elif char == '"':
          self.current_pos += 1
          string_value = self._read_string()
          self.tokens.append(('STRING', string_value))
      elif char == '#':  # Comment start
          comment = self._read_comment()
```

```
            self.tokens.append(('COMMENT', comment))
        else:
            raise Exception(f"Unknown character: '{char}' at position
{self.current_pos}")
```

The tokenize function: Breaking Down Code into Meaningful Units

The tokenize function within the Lexer class is the heart of lexical analysis. Here's what it does:

- **Iterates through the code:** The function uses a while loop to process each character (char) in the source code (self.input_text).
- **Handles whitespace:** If a whitespace character is found, it's skipped (self.current_pos += 1).

  Identifies numbers: It detects numeric literals (e.g., 123, 3.14) using a regular expression (re.match) and stores them as ('NUMBER', value) tokens.
- **Identifies keywords and identifiers:** Words are recognized, and keywords (like if, else) are distinguished from general identifiers (variable names) by checking against the self.keywords list. They are stored as ('KEYWORD', word) or ('IDENTIFIER', word) tokens, respectively.
- **Recognizes operators and delimiters:** It finds operators (+, -, /, etc.) and delimiters (;, (,), etc.), storing them as tokens with their corresponding types.
- **Recognizes strings:** Detects strings enclosed in double quotes (") and stores them as ('STRING', value) tokens.
- **Recognizes comments:** Finds comments starting with # and stores them as ('COMMENT', comment) tokens.
- **Error handling:** For unrecognized characters, an Exception is raised, providing context for debugging.

Purpose: The tokenize function transforms raw code into a list of tokens (self.tokens). These tokens carry semantic meaning that a parser can later use to understand the code's structure and logic.

- Number reading function which handles integer and float numbers

```
def _read_number(self):
   number = ''
   while self.current_pos < len(self.input_text) and \
            (self.input_text[self.current_pos].isdigit() or
self.input_text[self.current_pos] == '.'):
       number += self.input_text[self.current_pos]
       self.current_pos += 1
   return number
```

Purpose: This function is a core part of the Lexer class. Its job is to identify and extract numerical values (numbers) from the input source code.

How it Works:

1. **Initialization:** An empty string number is created to store the numerical value.
2. **Loop:** The function iterates as long as the current_pos (current position within the code) is valid and the characters encountered are either digits or a decimal point (.).
3. **Character Check:** Each character is checked to see if it represents a digit or a decimal point.
4. **Concatenation:** If the character is valid, it's appended to the number string.
5. **Position Update:** The current_pos is incremented to move to the next character.
6. **Return:** When an invalid character (not a number) is found, the loop ends, and the complete number string is returned.

- String reading function which handles Strings which start with ‘“’

```
def _read_string(self):
   string_value = ''
```

```
   while self.current_pos < len(self.input_text) and
self.input_text[self.current_pos] != '"':
       string_value += self.input_text[self.current_pos]
       self.current_pos += 1
   if self.current_pos == len(self.input_text):
       raise Exception("Unterminated string!")
   self.current_pos += 1
   return string_value
```

Function Purpose:

The `_read_string` function is a component of the `Lexer` class and is responsible for extracting string literals from the source code being analyzed. String literals are sequences of characters enclosed within double quotes (e.g., "This is a string").

How it Works:

1. **Initialization:** The `string_value` variable starts empty to store the accumulated string.
2. **Loop:** It iterates as long as the current position is within the source code and the next character is not a closing double quote.
3. **Character Accumulation:** Inside the loop, each character (except the opening quote) is added to the `string_value`.
4. **Position Update:** The `current_pos` is incremented to move to the next character.
5. **Error Handling:** If the end of the source code is reached without finding a closing quote, an "Unterminated string!" exception is raised to signal an error.
6. **Return:** After the loop (and handling a potential error), the function advances `current_pos` to skip the closing quote and returns the extracted `string_value`.

Example:

If the input text is message = "Hello, world!", the _read_string function would return the string Hello, world!.

- Comment reading function which handles the comments which start with '#'

```
def _read_comment(self):
    comment = ''
    while self.current_pos < len(self.input_text) and
self.input_text[self.current_pos] != '\n':
        comment += self.input_text[self.current_pos]
        self.current_pos += 1
    return comment
```

Function Purpose:

The _read_comment function is part of the Lexer class. Its purpose is to identify and extract comments within the analyzed code. It specifically handles single-line comments that begin with the "#" character.

Explanation

- **Initialization:** The function begins by creating an empty string called comment. This will store the extracted comment text.
- **Loop:** A while loop iterates as long as the current character position (self.current_pos) is within the code's length and the current character is not a newline (\n).
- **Character Accumulation:** Inside the loop, each character of the comment is appended to the comment string.
- **Position Update:** self.current_pos is incremented, moving the lexer's position forward within the code.
- **Returning the Comment** After the loop (when the end of the comment line is reached), the accumulated comment string is returned.

- Source code used to verify the functionality of the code, by calling the tokenize function from the lexer class.

```
source_code = """
# A simple calculator program


def add(num1, num2):
   # Adds two numbers and returns the result.
   return num1 + num2


def subtract(num1, num2):
   # Subtracts two numbers and returns the result.
   return num1 - num2


while True:
   print("Select an operation:")
   print("1. Add")
   print("2. Subtract")
   print("3. Exit")


   choice = input("Enter your choice (1/2/3): ")


   if choice in ('1', '2'):
       try:
           num1 = float(input("Enter the first number: "))
           num2 = float(input("Enter the second number: "))
       except ValueError:
           print("Invalid number format. Please try again.")
           continue


       if choice == '1':
           print(num1, "+", num2, "=", add(num1, num2))
```

```
        else:
            print(num1, "-", num2, "=", subtract(num1, num2))
    elif choice == '3':
        break
    else:
        print("Invalid choice. Please try again.")
"""
lexer = Lexer(source_code)
lexer.tokenize()
print(lexer.tokens)
```

Conclusion

This laboratory exercise deepened my understanding of lexical analysis as a fundamental stage in the compilation and interpretation of programming languages. Here's a breakdown of what I learned, how I applied it, and its potential future benefits:

- **Key Takeaways:**
    - **Lexical Analysis:** I gained a strong grasp of how lexers break down source code into meaningful tokens.
    - **Tokenization Rules:** I learned to define rules based on keywords, operators, delimiters, and other language elements to guide the tokenization process.
    - **Implementation:** By implementing the Lexer class, I solidified my ability to translate the theoretical concepts of lexical analysis into practical Python code.
- **Applications:**
    - **Compiler/Interpreter Development:** This lab equipped me with the foundational knowledge necessary to contribute to the creation of compilers and interpreters.
    - **Code Understanding:** The experience of building a lexer enhanced my ability to analyze and understand the structure of code in various programming languages.

- **Future Benefits**
    - **Advanced Compiler Concepts:** My understanding of lexical analysis lays the groundwork for delving into further compiler phases, such as parsing, syntax analysis, and code generation.
    - **Natural Language Processing (NLP):** The principles of tokenization have applicability in NLP tasks, where text needs to be broken down into units for analysis.

This lab has been an invaluable step in my journey with formal languages and finite automata. The insights gained will undoubtedly play a vital role in my future endeavors in computer science, enabling me to tackle problems in language design, compiler construction, and text processing.