Laboratory Work Nr.3 Lexer Scanner

Cretu Cristian

March 18, 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 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

Token Class

A class that represents a token in the language I try to tokenize. I overload the built in functions so that I can easily manipulate tokens.

```
class Token:
    """
    Represents a token in a lexer.

    Attributes:
```

```
        - type (str): The type of the token.
        - value (str): The value of the token.
    """

    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __repr__(self):
        return f"{self.type}"

    def __str__(self):
        return f"{self.type}"

    def __eq__(self, other):
        return self.type == other.type and self.value ==
    other.value

    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        return hash((self.type, self.value))

    def __lt__(self, other):
        return self.type < other.type and self.value < other
    .value
```

Tokenizer Class

This class is responsible for tokenizing a line of code and matching it to tokens while preserving the identifiers in the code.

```
from Token import Token
import re


class Tokenizer:
    """
    A class that tokenizes a given line of code based on
    predefined tokens.

    Attributes:
        - tokens (list): A list of Token objects
    representing the predefined tokens.

    Methods:
        - tokenize(line): Tokenizes the given line of code
    and returns a list of tokens found.
```

```
        - get_tokens(): Returns the list of predefined tokens.
        - print_tokens(): Prints the list of predefined tokens.
    """

    def __init__(self):
        self.tokens = [
            Token("int", r"\bint\b"),
            Token("float", r"\bfloat\b"),
            Token("string", r"\bstring\b"),
            Token("bool", r"\bbool\b"),
            Token("true", r"\btrue\b"),
            Token("false", r"\bfalse\b"),
            Token("if", r"\bif\b"),
            Token("else", r"\belse\b"),
            Token("while", r"\bwhile\b"),
            Token("for", r"\bfor\b"),
            Token("return", r"\breturn\b"),
            Token("break", r"\bbreak\b"),
            Token("continue", r"\bcontinue\b"),
            Token("function", r"\bfun\b"),
            Token("print", r"\bprint\b"),
            Token("lparen", r"\("),
            Token("rparen", r"\)"),
            Token("lbrace", r"\{"),
            Token("rbrace", r"\}"),
            Token("lbracket", r"\["),
            Token("rbracket", r"\]"),
            Token("comma", r","),
            Token("semicolon", r";"),
            Token("colon", r":"),
            Token("dot", r"\."),
            Token("plus", r"\+"),
            Token("minus", r"-"),
            Token("multiply", r"\*"),
            Token("divide", r"/"),
            Token("modulus", r"%"),
            Token("assign", r"="),
            Token("equal", r"=="),
            Token("not_equal", r"!="),
            Token("greater", r">"),
            Token("less", r"<"),
            Token("greater_equal", r">="),
            Token("less_equal", r"<="),
            Token("quote", r'"'),
            Token("single_quote", r"'"),
            Token("and", r"&&"),
            Token("or", r"\|\|"),
            Token("not", r"!"),
```

```
            Token("identifier", r"[a-zA-Z_][a-zA-Z0-9_]*"),
            Token("int_literal", r"\d+"),
            Token("float_literal", r"\d+\.\d+"),
            Token("string_literal", r'".*"'),
        ]

    def tokenize(self, line):
        """
        Tokenizes the given line of code and returns a list of tokens found.

        Args:
            - line (str): The line of code to be tokenized.

        Returns:
            - list: A list of Token objects representing the tokens found in the line of code.
        """
        tokens_found = []
        index = 0
        while index < len(line):
            match_found = False
            for token in self.tokens:
                pattern = re.compile(token.value)
                match = pattern.match(line, index)
                if match and match.start() == index:
                    tokens_found.append((token, match.group()))
                    index = match.end()
                    match_found = True
                    break
            if not match_found:
                index += 1
        return tokens_found

    def get_tokens(self):
        """
        Returns the list of predefined tokens.

        Returns:
            - list: A list of Token objects representing the predefined tokens.
        """
        return self.tokens

    def print_tokens(self):
        """
        Prints the list of predefined tokens.
        """
        print(self.tokens)
```

Lexer Class

This class is a wrapper around the Tokenizer class and it encapsulates the Tokenizer class to be easier to use.

```
class Lexer:
    """
    The Lexer class is responsible for tokenizing input lines from a file.

    Args:
        - file_path (str): The path to the input file.

    Attributes:
        - lines (FileReader): An instance of the FileReader class to read lines from the file.
        - tokens (Tokenizer): An instance of the Tokenizer class to tokenize the lines.

    Methods:
        - tokenize(): Tokenizes each line from the file and prints the resulting tokens.
    """

    def __init__(self, file_path: str):
        self.lines = FileReader(file_path=file_path)
        self.tokens = Tokenizer()

    def tokenize(self):
        """
        Tokenizes each line from the file and prints the resulting tokens.
        """
        self.tokens = [self.tokens.tokenize(line) for line in self.lines.get()]

    def get_tokens(self):
        """
        Returns the list of predefined tokens.

        Returns:
            - list: A list of Token objects representing the predefined tokens.
        """
        self.tokenize()
        return self.tokens

    def print_tokens(self):
        """
        Prints the list of predefined tokens.
```

```
        """
        self.tokenize()
        for line in self.tokens:
            print(line)
```

Main File

```
from Lexer import Lexer

lxr = Lexer(file_path="/Users/cristiancretu/Documents/
    UniCode/LFA/Lexer/example.flexksx")
lxr.print_tokens()
```

0.1 Example Usage

I run the main file over this code example:

```
print("Hello");
print("Hi world");
for(i=0;i<5;i+=1) {print("Hello world")}
```

Which gives me the following output:

```
[(print, 'print'), (lparen, '('), (quote, '"'), (identifier,
    'Hello'), (quote, '"'), (rparen, ')'), (semicolon, ';')]
[(print, 'print'), (lparen, '('), (quote, '"'), (identifier,
    'Hi'), (identifier, 'world'), (quote, '"'), (rparen, ')'
    ), (semicolon, ';')]
[(for, 'for'), (lparen, '('), (identifier, 'i'), (assign, '=
    '), (int_literal, '0'), (semicolon, ';'), (identifier, 'i
    '), (less, '<'), (int_literal, '5'), (semicolon, ';'), (
    identifier, 'i'), (plus, '+'), (assign, '='), (
    int_literal, '1'), (rparen, ')'), (lbrace, '{'), (print,
    'print'), (lparen, '('), (quote, '"'), (identifier, '
    Hello'), (identifier, 'world'), (quote, '"'), (rparen, ')
    '), (rbrace, '}')]
```

Where each line is a list of token objects, that when printed give a representation of their type and their value.

Conclusions / Screenshots / Results

The program successfully converted my custom written code into tokens that can be further used for future implementations. It used Regular Expressions used from a Python library to implement the identification of certain symbol patterns that represent tokens in order to achieve the desired tokenization.