Lexer & Scanner

Author: Plămădeală Maxim

Group: FAF-222

Theory

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:

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 tips:

1. You can find implementation tips on the reference [2].
2. Please take into consideration the indicated structure of the project. Find a suitable place for the lexer implementation and for the new report of course.

Implementation

Main `lexer.py`

The `lexer.py` file contains the main methods that filter out or tokenize the given string using the regex or simply re (library in python).

Let's first look into one of the basic functions present in here:

```
def t_FLAG(data, pos):
    match = re.match(r'--\w+', data[pos:])
    if match:
        return match.group(), pos + len(match.group())
    return None, pos
```

The functions here as you can see are pretty simple. As an example the `t_FLAG` def, determines whether in the string a flag is present. In simple words a flag is a string that contains `--` before some letters.
As a parameter the function receives the data, the initial string and the position to check onto. If there are some matches, it groups them and returns the match group and the coordinates of the string, else returns none and the position where it started.

In general all the methods work like this. After creating all the methods, I created a def `tokenize()` to tokenize my string according to my rules that I have written:

```
def tokenize(data):
    tokens = []
    pos = 0
    while pos < len(data):
        found_token = False
        for token_type in [t_PLUS, t_MINUS, t_ASSIGN, t_SEMICOLON, t_INTEG
            if callable(token_type):
                token_value, new_pos = token_type(data, pos)
                if token_value is not None:
                    tokens.append((token_type.__name__[2:], token_value))
                    pos = new_pos
                    found_token = True
                    break
        if not found_token:
            # Handle exceptions for specific characters
            current_char = data[pos]
            if current_char in ['"', "'", ' ', '\t', '\n']:
                pos += 1
            else:
                unknown_token = data[pos]
                tokens.append(("UNKNOWN", unknown_token))
                pos += 1
    return tokens
```

In simple words it checks if every string separated by whitespaces is matching with any filters, and appends them into a dictionary of a specific structure `[(RULE TYPE, STRING)]`.
The most banal example could be:

```
data = 'sudo dnf'
tokenize(data)
```

```
[('COMMAND', 'sudo'), ('COMMAND', 'dnf')]
```

Tests

Paths and images

```
tokenize('imp --img="image.jpg" --x=1000 --y=100 --w=200 --h=200')
```

```
[('COMMAND', 'imp'),
 ('FLAG', '--img'),
 ('ASSIGN', '='),
 ('IMAGE_PATH', '"image.jpg"'),
 ('FLAG', '--x'),
 ('ASSIGN', '='),
 ('INTEGER', 1000),
 ('FLAG', '--y'),
 ('ASSIGN', '='),
 ('INTEGER', 100),
 ('FLAG', '--w'),
 ('ASSIGN', '='),
 ('INTEGER', 200),
 ('FLAG', '--h'),
 ('ASSIGN', '='),
 ('INTEGER', 200)]
```

Linux

```
tokenize('sudo dnf update')
```

```
[('COMMAND', 'sudo'), ('COMMAND', 'dnf'), ('COMMAND', 'update')]
```

Directory management

```
tokenize('cd "C:\\Users\\User\\Desktop\\folder"')
```

```
[('DIR_NAV_COMMANDS', 'cd'), ('PATH', '"C:\\Users\\User\\Desktop\\folder"'
tokenize('touch "/path/to/existing_image.png"')
```

```
[('DIR_NAV_COMMANDS', 'touch'),
 ('IMAGE_PATH', '"/path/to/existing_image.png"')]
```

Conclusion

In this project I built a tool that separates image processing instructions into meaningful parts. This "lexer" takes a series of characters and turns them into useful building blocks, like commands, options, numbers, and file types. It can handle different situations because it's built on a flexible system. Overall, this project not only teaches about formal languages but also shows how to create a lexer for real-world tasks.

The code for this project is in a file named `lexer.py` and there's a notebook named `test.ipynb` that explains everything in more detail. This notebook shows how the lexer works by giving examples of code and its output.

Bibliography

- no bibliography