laboratory_3.md

Laboratory Work 3

Course: Formal Languages & Finite Automata

Author: Felicia Novac

Theory

Lexical Analysis which is also known as tokenization is the process of breaking down a complex string of characters from source code into manageable pieces called tokens (from here comes the term tokenization). These so called tokens are small units that carry meaning and are essential for the programming language's understanding and further processing of the input. The tokens are categorized in types `regex`, each representing a different element of the syntax of the programming language. Such examples of tokens might include:

- *keywords* - crop, resize
- *identifiers* - foo, var
- *numbers* - 7, 23
- ...

Objectives:

- Understand what lexical analysis is.
- Get familiar with the inner workings of a lexer/scanner/tokenizer.
- Implement a sample lexer and show how it works.

Implementation description

I've decided to create a sample Tokenizer for the ELSD project, which is a DSL for image processing. Despite the fact that it was implemented within the project using ANTLR, I tried to do it on my own in the way I see it, with the use of example from [1].

The code within the lab is an implementation of a tokenizer class in Java. This class has a method called tokenize(String input). The method uses regular expressions to identify tokens in the input string, and returns a list of Token objects.

The Tokenizer class contains a constant String `TOKEN_REGEX`, which represents all possible token names that can be identified by the tokenizer, splitted with the use of OR operator. It also contains several constant regular expressions, one for each type of token that can be identified. These regular expressions are used to match tokens in the input string and were constructed based on the Grammar definition within the DSL.

The tokenize method creates a new ArrayList of Token objects to hold the tokens found in the input string. It then creates a Matcher object using the Pattern constant created from the regular expressions. The method then iterates over the input string, using the Matcher object to find matches for the regular expressions. For each match found, the method identifies Token Type by using a helper method `determineTokenType(String value)`, and creates a new Token object with its corresponding type and value, and adds it to the list of tokens. It is worthy to mention that I've additionally taken into account the index of the tokens being processed while tokenizing, in order to meaningfully handle errors regarding the unmatched patterns. So, I've basically compared the position where a token match is found against the position it last stopped at. If there's a gap, meaning some input hasn't matched any token pattern, I report the error using an additional helper method `reportError(String value)`. Also, as the quotations are defined within our language as limits of file/folder paths, I manually extract the quotations from those tokens and place them as separate ones, and also place the paths without the quotations (beginning and end of the substring) for convenience in future work.

Finally, the tokenize method returns the list of Token objects.

Conclusions / Screenshots / Results

In order to "simulate" the DSL in function, I take the input commands from the console, and I'll provide some screenshots with valid & invalid input prompts and the corresponding results.

As it can be seen from Table 1 and compared to the Grammar from [2], the only *parts* of the input that are being added to the List of Token objects, are those defined within the *grammar* and *regex*.

Valid Input Prompts	Invalid Input Prompts

`convert --img="image.png" --format="jpg"`
`Token{type='COMMAND', value='convert'}`
`Token{type='IMAGE_IDENTIFIER', value='--img'}`
`Token{type='EQUALS', value='='}`
`Token{type='QUOTE', value='"'}`
`Token{type='FILE_PATH', value='image.png'}`
`Token{type='QUOTE', value='"'}`
`Token{type='PARAMETER', value='--format'}`
`Token{type='EQUALS', value='='}`
`Token{type='IMAGE_TYPE', value='"jpg"'}`
	`rotate --img="image.png" --degrees=90`
	`Unrecognized input: '--degrees'`
	`Token{type='COMMAND', value='rotate'}`
	`Token{type='IMAGE_IDENTIFIER', value='--img'}`
	`Token{type='EQUALS', value='='}`
	`Token{type='QUOTE', value='"'}`
	`Token{type='FILE_PATH', value='image.png'}`
	`Token{type='QUOTE', value='"'}`
	`Token{type='EQUALS', value='='}`
	`Token{type='NUMBER', value='90'}`

`rotate --img="folder\path" --deg=90`
`Token{type='COMMAND', value='rotate'}`
`Token{type='IMAGE_IDENTIFIER', value='--img'}`
`Token{type='EQUALS', value='='}`
`Token{type='QUOTE', value='"'}`
`Token{type='FOLDER_PATH', value='folder\path'}`
`Token{type='QUOTE', value='"'}`
`Token{type='PARAMETER', value='--deg'}`
`Token{type='EQUALS', value='='}`
`Token{type='NUMBER', value='90'}`
	`please resize my --img="image.png" ^-^`
	`Unrecognized input: 'please'`
	`Unrecognized input: 'my'`
	`Unrecognized input: '^-^'`
	`Token{type='COMMAND', value='resize'}`
	`Token{type='IMAGE_IDENTIFIER', value='--img'}`
	`Token{type='EQUALS', value='='}`
	`Token{type='QUOTE', value='"'}`
	`Token{type='FILE_PATH', value='image.png'}`
	`Token{type='QUOTE', value='"'}`

Table 1. Tokenizer in Action

In conclusion, I've managed to understand the essence of Lexical Analysis, how to implement it and perform an accurate tokenization with corresponding error validation. It took relatively much time when reviewing all possible input prompts and trying to overcome some possible issues that might occur (hopefully I covered all of them). However, it was interesting to code and see how the Lexer does its job.

P.S. I've implemented lexer the way I saw it the most convenient to work with when parsing, so please do not judge :)

References

[1] **A Sample of a Lexer Implementation** - Accessed March 13, 2024. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html.

[2] **DSL Grammar for image processing** - Accessed March 13, 2024. [https://1drv.ms/b/c/06b88d8283f1f472/EcWOdkET6aVOtnGd6-9MG3QB221Cpzyqvsfwg0QWueeGLg?e=m6SUlf].

[3] **Introduction to Lexical Analysis** - Accessed March 12, 2024. [https://www.geeksforgeeks.org/introduction-of-lexical-analysis/].