MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

Laboratory work 3:
Lexer & Scanner.

Course: Formal Languages and Finite Automata

Author: Cucoș Maria

Chișinău, 2024

TABLE OF CONTENTS

Theory

The lexer, also known as the lexical analyzer, plays a crucial role in the compilation or interpretation process of a programming, markup, or domain-specific language (DSL). It serves as the initial stage of language processing, responsible for breaking down the input source code into meaningful units called *lexemes*. These *lexemes* represent the fundamental building blocks of the language syntax and are subsequently used by the parser for further analysis.

Lexical Analysis Process

The lexical analysis process involves scanning the input source code character by character and identifying lexical *tokens* based on predefined rules or patterns specified by the language grammar. *Tokens* represent meaningful units of the language, such as keywords, identifiers, literals, operators, and punctuation symbols.

Lexical Components

Lexers typically use regular expressions to define patterns for recognizing different types of tokens. Each regular expression corresponds to a specific token type. The lexer tokenizes the input source code by applying the defined regular expressions iteratively. As it scans through the input, it matches the characters against the patterns to identify tokens. Once a token is recognized, the lexer emits it along with any associated metadata, such as the token type and its value.

Difference Between Lexemes and Tokens

Lexemes are the atomic units produced by splitting the input source code based on delimiters or separators, such as whitespace or punctuation characters. Tokens are categorized representations of lexemes that carry semantic meaning in the language. They provide names or types to each lexeme, enabling subsequent stages of language processing to understand the structure and semantics of the code.

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

In the context of designing a Domain-Specific Language (DSL) for AI projects architecture with my team for another course, I decided to develop a lexer for it. The language is designed to address the specific needs and requirements of AI project architecture. The language features a set of symbols, variables, and strings that encapsulate common actions and parameters encountered in AI project workflows.

The symbols represent high-level actions or commands that the DSL can interpret and execute. These symbols are designed to cover a broad range of AI-related tasks, including image processing, text recognition, model training, data loading, and audio processing. Each symbol corresponds to a specific operation or task in an AI project workflow, such as detecting objects, recognizing text, or generating text.

The DSL includes support for variables, which allow users to define and manipulate parameters or input/output paths within their AI projects. Variables provide a flexible mechanism for configuring various aspects of AI workflows, such as specifying file paths, model parameters, or input data sources.

The `main` function ensures that when the script is executed directly, it reads the contents of the ”test.txt” file, parses it using the `parse()` function, and prints the resulting lexemes to the console.

```
if __name__ == "__main__":
    argv.append("test.txt")
    print(parse(argv[1]))
```

The `interpreter` file contains two functions. The `lexer(contents)` function transforms a text file into a structured representation where each line is broken down into a list of lexemes, shown in Fig.1, with each lexeme classified based on its type (string, symbol, variable). This structured representation is then used by the `parse` function for further processing. Then, the function `lines = contents.split("\n")` takes the contents of the file as input and splits it into individual lines using the split() method with the new-line character.

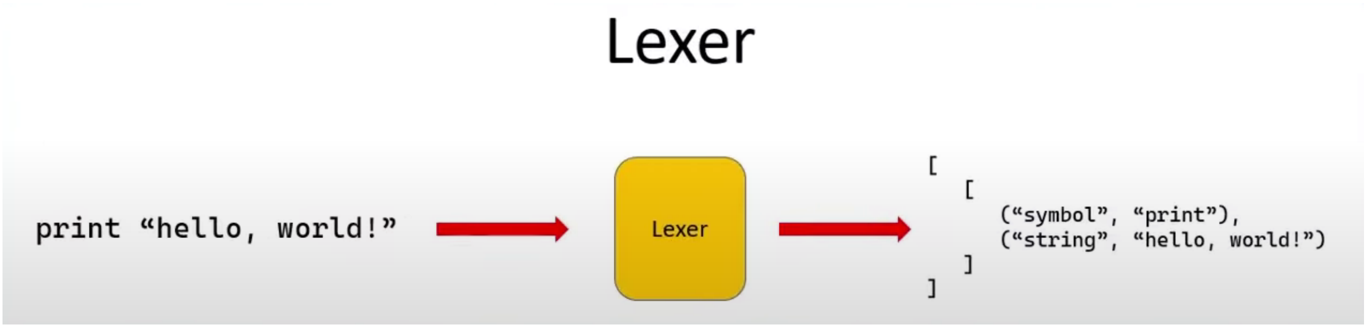

Fig.1 Lexer Logic

For each line, the function iterates over the characters. It maintains a token variable to accumulate characters until a delimiter is encountered. Delimiters are defined as spaces when not within quotes (indicating the end of a token). If a character is within quotes, the `in_quotes` flag is toggled, indicating that it’s part of a string literal.

```
for char in line:
    if char in ['"', "'"]:
        in_quotes = not in_quotes
        token += char
    elif char == " " and not in_quotes:
        if token:
            lexemes.append(token)
            token = ""
    else:
        token += char
```

Once a token is complete, it is appended to a list of `lexemes` for the current line. If a token starts with a quote character, it is classified as a string literal. If it matches any symbol in the Symbols set, it is classified as a symbol. If it contains an equals sign, it is split into a variable name and its corresponding value, and both are classified as such.

```
    for token in lexemes:
        if token.startswith('"') or token.startswith("'"):
            line_lexemes.append(("String", token))
        elif token in Symbols:
            line_lexemes.append(("Symbol", token))
        elif "=" in token:
            parts = token.split("=")
            variable = parts[0]
            value = parts[1]
            line_lexemes.append(("Variable", variable))
            line_lexemes.append(("String", value))
```

After processing all tokens in a line, the list of `lexemes` for that line is appended to the `lexemes_list`. Finally, the function returns the list of `lexemes` for each line in the file.

The `Symbols` set contains the symbols recognized by the interpreter. These symbols represent actions or commands that the interpreter can execute. They are the following:

```
Symbols = {
"detect",
"recognize",
"read_text",
"read_text_lang",
"train",
"load",
"generate_text",
"retrieve_params",
"recognize",
"process",
"transcribe",
"convert"
}
```

The `Vars = {}` dictionary stores variables and their values encountered during parsing. It starts as an empty dictionary and gets populated as the parser encounters variable assignments in the input file.

The `parse(filename)` function takes a filename as input and reads the contents of the file. It then calls the `lexer()` function to tokenize the contents of the file into a list of lexemes for each line. It iterates over each line of lexemes returned by the `lexer()` function. For each line, it constructs an instruction line `instr_line` by concatenating symbols encountered in the line. If a token is a symbol recognized by the interpreter, it adds it to the instruction line. If a token is classified as a variable, it checks if the next token is a string (indicating a variable assignment) and adds the variable-value pair to the Vars dictionary. After processing each line, if there are no errors and all tokens are properly classified, it prints the instruction line.

```
for line in lines:
    instr_line = ""
    for token in line:
        if token[0] == "Symbol":
            if token[1] in Symbols:
                instr_line += token[1] + " "
        elif token[0] == "Variable":
            if len(line) > 1 and line[line.index(token) + 1][0] == "String":
                Vars[token[1]] = line[line.index(token) + 1][1]
        else:
            break
    else:
        print(instr_line)
```

Additionally, it prints each line of lexemes separately.

The contents of the `test.txt` file that are being examined are the following:

```
detect imagePath="path/to/image1.png"
recognize imagePath="path/to/image2.png"
read_text imagePath="path/to/image3.png"
read_text_lang imagePath="path/to/image4.png"
train trainingData="path/to/training_data.txt"
load modelPath="path/to/model"
generate_text prompt="Hello, World!"
```

```
retrieve_params modelName="my_model"
recognize audioPath="path/to/audio1.wav"
process audioPath="path/to/audio2.wav"
transcribe audioPath="path/to/audio3.wav"
convert audioPath="path/to/audio4.wav"
```

The output is presented bellow.

```
[('Symbol', 'detect'), ('Variable', 'imagePath'), ('String', '"path/to/image1.png"')]
[('Symbol', 'recognize'), ('Variable', 'imagePath'), ('String', '"path/to/image2.png"')]
[('Symbol', 'read_text'), ('Variable', 'imagePath'), ('String', '"path/to/image3.png"')]
[('Symbol', 'read_text_lang'), ('Variable', 'imagePath'), ('String', '"path/to/image4.png"')]
[('Symbol', 'train'), ('Variable', 'trainingData'), ('String', '"path/to/training_data.txt"')]
[('Symbol', 'load'), ('Variable', 'modelPath'), ('String', '"path/to/model"')]
[('Symbol', 'generate_text'), ('Variable', 'prompt'), ('String', '"Hello, World!"')]
[('Symbol', 'retrieve_params'), ('Variable', 'modelName'), ('String', '"my_model"')]
[('Symbol', 'recognize'), ('Variable', 'audioPath'), ('String', '"path/to/audio1.wav"')]
[('Symbol', 'process'), ('Variable', 'audioPath'), ('String', '"path/to/audio2.wav"')]
[('Symbol', 'transcribe'), ('Variable', 'audioPath'), ('String', '"path/to/audio3.wav"')]
[('Symbol', 'convert'), ('Variable', 'audioPath'), ('String', '"path/to/audio4.wav"')]
```

Conclusion

In this report, I explored the particularities of lexical analysis and its application in designing a Domain-Specific Language for AI projects architecture. My primary objectives were to understand lexical analysis, familiarize myself with the inner workings of a lexer, and implement a sample lexer to demonstrate its functionality.

Through comprehensive research and study, I gained insights into the role of lexical analysis in parsing and interpreting source code or textual data. By dissecting the process of lexical analysis, I identified key concepts such as tokens, lexemes, and delimiters, which form the building blocks of a lexer. Drawing inspiration from established techniques in compiler theory, I designed and coded a lexer capable of breaking down input text into meaningful tokens. To showcase the efficacy of the lexer, I conducted experiments with sample input data, illustrating how the lexer effectively tokenizes text according to predefined rules and patterns.

In conclusion, I have learned a lot of lexers and how they work. It's a powerful tool for designing and implementing DSLs tailored to specific domains, such as AI project architecture in this report.