Formal Languages & Finite Automata

Laboratory work 3:

Lexer & Scanner

Student: Grigoraș Dumitru
Verified: prof., Cojuhari Irina
univ. assist., Crețu Dumitru

Chișinău, 2024

Theory

Lexer and Scanner - In the realm of programming languages, lexers and scanners play vital roles in the process of transforming source code into a format that can be understood and processed by computers. These components are fundamental in the lexical analysis phase of compilation, where the input source code is broken down into meaningful tokens or lexemes.

Lexer:

- A lexer is often the first component in the compilation process. Its primary function is to scan the input source code character by character and identify tokens based on predefined patterns or rules.
- The lexer operates on the principle of regular expressions and finite automata. It uses regular expressions to define the patterns of tokens and finite automata to recognize and tokenize these patterns efficiently.
- The output of the lexer is a stream of tokens, each representing a specific type of element in the source code. These tokens serve as input for the parser in the subsequent parsing phase.

Scanner:

- The terms "lexer" and "scanner" are often used interchangeably, referring to the same component in the lexical analysis phase.
- However, in some contexts, the scanner may refer specifically to the part of the lexer that performs the actual scanning and tokenization of the input stream.
- The scanner typically utilizes techniques such as deterministic finite automata (DFA) or non-deterministic finite automata (NFA) to efficiently recognize and tokenize input based on predefined lexical rules.

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

TokenType

This enum represents different types of tokens that can be encountered during lexical analysis or parsing tasks. Here's an explanation of each token type:

```
package Lab3;

public enum TokenType {
    AGENT,
    TEXT,
    DESTINATION,
    END_OF_FILE,
    ERROR
}
```

TokenClass:

The Token class represents a lexical token in the context of lexical analysis and parsing. It encapsulates information about the type and value of a token, allowing for precise identification and handling of different elements within the source code.

Token Class Elements:

1. **type**: A member variable of type TokenType that represents the type of the token, such as identifier, keyword, literal, or punctuation.
2. **value**: A member variable of type String that stores the actual value of the token, providing additional context or information about its content.

Constructors:

- **Token**(TokenType type, String value): Constructs a Token object with the specified type and value. This constructor initializes the type and value member variables of the Token instance.

Getter Methods:

- getType(): Returns the TokenType of the Token object, indicating its classification or category.
- getValue(): Returns the value of the Token object, providing the actual content or representation.

toString() Method - overrides the default toString() method to provide a string representation of the Token

Lexer Class:

The Lexer class is responsible for tokenizing input strings based on predefined patterns and rules. It identifies and categorizes different elements within the input, such as agents, text, and destinations, and generates tokens representing these elements.

Lexer Class Elements:

1. AGENT_PATTERN: A regular expression pattern to match strings representing agents, such as "Agent1", "Agent2", or "Agent3".
2. TEXT_PATTERN: A regular expression pattern to match text enclosed within double quotes, such as "Hello, world!".
3. DESTINATION_PATTERN: A regular expression pattern to match strings representing destinations, similar to the agent pattern.

Methods:

- **tokenize**(String input): This method takes an input string and tokenizes it based on predefined patterns. It scans the input string, identifies tokens using regular expressions, and creates Token objects representing the identified elements.
    - It uses a Matcher to match patterns and extracts tokens accordingly.
    - It handles special cases such as identifying destinations following the "to" keyword and extracting text between "says" and "to" keywords.
    - Unrecognized tokens are tokenized as TokenType.ERROR to indicate errors in the input.

Results

1. **Input =** *Agent1 says \"Hello, world!\" to Agent2*

```
Tokens:
(AGENT, Agent1)
(TEXT, "Hello, world!")
(DESTINATION, Agent2)
```

2. **Input =** *GPT says \"Make a photo of a cat\" to DALLE*

```
Tokens:
(AGENT, GPT)
(TEXT, "Make a photo of a cat")
(DESTINATION, DALLE)
```

Conclusion

In this lab project focused on Lexer and Scanner implementation, I have successfully applied fundamental concepts from lexical analysis and parsing to develop a practical coding framework. By creating a Lexer class capable of tokenizing input strings based on predefined patterns and rules, I have delved into the intricacies of language processing and strengthened my understanding of compiler design principles.

The construction of the Lexer class involved the utilization of regular expressions and pattern matching techniques to identify and categorize different elements within the input. This process allowed for the creation of Token objects representing various token types, such as agents, text, destinations, and error tokens.

Furthermore, the implementation of the tokenize() method in the Lexer class showcased the practical application of lexical analysis techniques, including handling special cases and generating tokens according to specified rules. The Lexer class serves as a crucial component in the overall compilation process, contributing to the initial phase of source code analysis and interpretation.

Through this project, I have gained valuable hands-on experience in designing and implementing lexer components, which are fundamental in transforming source code into a format that can be processed by compilers. This practical application has deepened my understanding of formal languages, automata theory, and compiler construction, enhancing my problem-solving skills and reinforcing the importance of theoretical foundations in computer science.

Overall, this lab project has been a reflective and insightful journey that bridges theory with practice, demonstrating the transformation of complex theoretical constructs into functional and practical solutions. It has contributed significantly to my academic and personal development, solidifying my knowledge in language processing and compiler design concepts.

6

Bibliography

[1] Formal Languages and Compiler Design Accessed February 14, 2024. https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf.

[2] Regular Language. Finite Automata. Accessed February 15, 2024. https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view