A Report on

LEXER IMPLEMENTATION USING ANTLR4 (SCANNER PART)

Submitted By:

Tanisha Majumder
Vaishak Balachandra

Under the Guidance of

Hairong Zhao, PhD
Professor of Computer Science; Graduate Advisor

Purdue University Northwest
Department of Computer Science
2025-26

LEXER IMPLEMENTATION
(GROUP – 4)

Input: Lexer Grammar Specification

Our lexer is defined using ***ANTLR4***'s grammar syntax. The complete grammar file `MyLexer.g4` is presented below:

```
lexer grammar MyLexer;

// Keywords
KEYWORD_DOUBLE : 'double' ;
KEYWORD_INT : 'int' ;
KEYWORD_LONG : 'long' ;
KEYWORD_CHAR : 'char' ;
KEYWORD_BOOL : 'bool' ;
KEYWORD_FUN : 'fun' ;
KEYWORD_IF : 'if' ;
KEYWORD_THEN : 'then' ;
KEYWORD_ELSE : 'else' ;
KEYWORD_TRUE : 'true' ;
KEYWORD_FALSE : 'false' ;
KEYWORD_ORELSE : 'orelse' ;
KEYWORD_ANDALSO : 'andalso' ;

// Operators
ASSIGN : '=' ;
PLUS_ASSIGN : '+=' ;
MINUS_ASSIGN : '-=' ;
MULT_ASSIGN : '*=' ;
DIV_ASSIGN : '/=' ;
PLUS : '+' ;
MINUS : '-' ;
MULT : '*' ;
DIV : '/' ;
INT_DIV : '//' ;
GT : '>' ;
LT : '<' ;
EQ : '==' ;
NEQ : '!=' ;
NOT : '!' ;

// Separators
SEMICOLON : ';' ;
COMMA : ',' ;
LPAREN : '(' ;
RPAREN : ')' ;
LBRACE : '{' ;
RBRACE : '}' ;
```

```
// Valid Integer Literals
INTEGER_LITERAL : '0'
                | [1-9] ('_'* [0-9])*
                | [0-9] ('_'* [0-9])* [lL]
                ;

// Valid Double Literals
DOUBLE_LITERAL : [0-9]+ '.' [0-9]+
               | '.' [0-9]+
               ;

// Invalid Integer Literals
INVALID_INTEGER_LITERAL : '0' [0-9_]+ // Leading zero
                        | '_' [0-9_]+ // Starting with underscore
                        | [0-9] ('_'* [0-9])* '_' // Ending with underscore
                        ;

// Invalid Double Literals
INVALID_DOUBLE_LITERAL : '0' [0-9]* '.' [0-9]* // Leading zero for non-zero
                       | [0-9]+ '.' // No digits after decimal
                       ;

// Char Literal
CHAR_LITERAL : '"' . '"' ;

// Identifier
IDENTIFIER : [a-zA-Z] [a-zA-Z0-9]* ;

// Invalid Identifier starting with a number
INVALID_IDENTIFIER : [0-9]+ [a-zA-Z] [a-zA-Z0-9]* ;

// Comments - skip
COMMENT : '(*' .*? '*)' -> skip ;

// Whitespace - skip
WS : [ \t\r\n]+ -> skip ;

// Error catch-all
ERROR : . ;
```

Output: Generated Lexer

The ANTLR4 tool processes our grammar file to generate lexer code. Below is a simplified explanation of the key components in the generated lexer (the actual output would be much more extensive and language-specific):

Main Components of the Generated Lexer

1. **Token Definition**: The generated lexer includes constants for each token type defined in the grammar.
2. **Lexical States**: ANTLR4 generates a state machine that recognizes tokens based on the patterns specified in the grammar.
3. **Character Classification**: The lexer includes methods to classify input characters and determine which token patterns they could potentially match.
4. **Token Recognition**: Methods that implement the transition logic for each token type, following the automata defined in our transition diagrams.
5. **Error Handling**: Code to handle invalid input and report lexical errors.

Python Implementation Example (test_lexer.py)

Here's a simplified example of how the generated lexer might be used in a Python application:

```
import sys
from antlr4 import FileStream, CommonTokenStream
from MyLexer import MyLexer

# Ensure a filename is provided
if len(sys.argv) < 2:
    print("Usage: python test_lexer.py <input_file>")
    sys.exit(1)

# Read the specified input file
input_filename = sys.argv[1]
input_stream = FileStream(input_filename, encoding="utf-8")

# Initialize lexer and token stream
lexer = MyLexer(input_stream)
token_stream = CommonTokenStream(lexer)
token_stream.fill()

# Define invalid token categories based on lexer rules
INVALID_TOKENS = {
    "INVALID_INTEGER_LITERAL",
    "INVALID_DOUBLE_LITERAL",
    "INVALID_IDENTIFIER",
    "ERROR"
}

# Open output file for writing
output_filename = input_filename.replace('.txt', '_output.txt')
with open(output_filename, 'w') as output_file:
    # Iterate through tokens and print in the format <token, token_category> --- VALID/INVALID
    for token in token_stream.tokens:
        if token.type == -1:  # EOF token
            break

        token_text = token.text
        token_category = lexer.symbolicNames[token.type] if token.type>0 else "UNKNOWN"
```

```
        # Mark as INVALID if it belongs to the invalid category
        validity = "INVALID" if token_category in INVALID_TOKENS else "VALID"

        # Format the output
        output_line = f"<{token_text}, {token_category}> --- {validity}"

        # Print to console and write to file
        print(output_line)
        # output_file.write(output_line + '\n')

# print(f"\nTokenization complete. Results saved to {output_filename}")
```

Implementation Details

Token Recognition Process

The generated lexer follows these steps to recognize tokens:

1. **Input Reading**: The lexer reads input characters one by one from the input stream.
2. **State Transitions**: For each character, the lexer transitions between states according to the DFA (Deterministic Finite Automaton) generated from the grammar.
3. **Token Identification**: When the lexer reaches an accepting state, it identifies the corresponding token type.
4. **Token Creation**: The lexer creates a token object with information about its type, text, position, and channel.
5. **Error Handling**: If the lexer encounters invalid input, it generates an ERROR token and continues processing.

Performance Considerations

The generated lexer is optimized for performance with several key features:

1. **Deterministic Automaton**: ANTLR4 generates a deterministic finite automaton, ensuring linear-time complexity for lexical analysis.
2. **Lookahead**: The lexer uses lookahead when necessary to disambiguate between potential token matches.
3. **Character Classification**: The lexer efficiently classifies characters to quickly determine potential matching patterns.
4. **Token Buffering**: The token stream implementation buffers tokens, reducing the overhead of repeated lexical analysis.

Integration with Parser

While our current focus is on the lexer component, it's worth noting how the lexer integrates with a parser in a complete compiler pipeline:

1. The lexer generates a stream of tokens from the input text.
2. The parser consumes these tokens to build a parse tree according to the language grammar.
3. Semantic analysis then processes the parse tree to extract meaning and validate the program.

The clean separation between lexical and syntactic analysis in ANTLR4 allows for modular development and testing of each component.