A black background with grey leaves

AI-generated content may be incorrect.

Lexical Analyzer

Build Scanner

A grey logo on a black background

AI-generated content may be incorrect.

**Prepared By**

Student Name

Youssef Ashraf Halim

Student ID

**200039009**

**Under Supervision**

Name of Doctor

DR/Nihal

Name of T. A.

ENG/Fares

1. **Introduction**

**Project Overview: Lexical Analyzer Implementation**

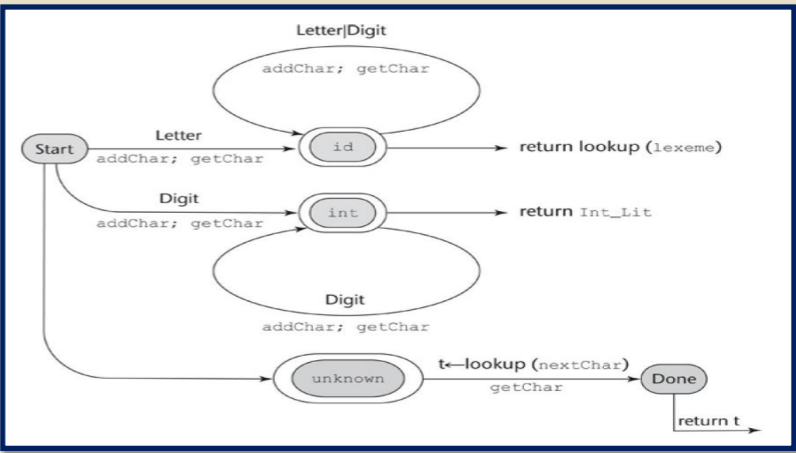
This document provides a comprehensive overview of the implementation of a **Lexical Analyzer**, a crucial component in the compilation process and a fundamental phase in compiler design. The lexical analyzer, often referred to as the **scanner**, is responsible for reading the source code as a sequence of characters and converting it into a sequence of **tokens**. These tokens represent logical units such as keywords, identifiers, literals, operators, and punctuation symbols, which are later used by the parser for syntax analysis.

The documentation begins by outlining the **phases of a compiler**, emphasizing the role and importance of lexical analysis in the overall compilation pipeline. Specifically, it discusses how the lexical analyzer interacts with both the source code and the subsequent syntactic and semantic analysis phases, acting as a bridge between raw code and structured input for the parser.

The document also describes the **software tools** and technologies used in the implementation, such as the C programming language, standard input/output libraries, and character classification utilities like ctype.h. These tools enable efficient character processing and token recognition, which are critical to the analyzer's performance.

In the **implementation section**, the document dives into the design of the lexical analyzer, explaining how input expressions are read from the user, processed character-by-character, and classified into meaningful tokens. The program uses functions like getChar(), addChar(), lookup(), and lex() to manage character reading, lexeme construction, token classification, and whitespace handling. Each function plays a specific role in simulating the behavior of a real-world compiler’s lexical phase.

By the end of this documentation, readers will gain a clear understanding of how lexical analyzers work, how to implement one using C, and why this phase is essential for the accurate and efficient translation of programming languages.

* 1. **Phases of Compiler**
  2. ****

A **compiler** is a complex software system that translates high-level programming code into machine-readable instructions. It operates through several well-defined phases, each with a specific role in the compilation process:

1. **Lexical Analysis** – The source code is scanned and broken down into a sequence of **tokens** (such as keywords, identifiers, operators, and literals). This is the first phase of the compiler and is handled by the **lexical analyzer**.
2. **Syntax Analysis** – Also known as **parsing**, this phase checks the arrangement of tokens to ensure they follow the grammatical rules of the programming language. It typically constructs a **parse tree** or **syntax tree**.
3. **Semantic Analysis** – This phase validates the meaning of the code by ensuring that operations are semantically correct (e.g., variables are declared before use, type compatibility is maintained).
4. **Intermediate Code Generation** – The compiler converts the parsed code into an **intermediate representation** that is easier to analyze and optimize. This form is independent of both the source and target languages.
5. **Code Optimization** – The intermediate code is analyzed and optimized to enhance **performance**, **reduce resource usage**, and **improve execution efficiency** without altering its functionality.
6. **Code Generation** – The final phase involves translating the optimized intermediate code into **target machine code** or assembly language, suitable for execution on a specific hardware platform.
7. **Lexical Analyzer**

The Lexical Analyzer scans the source code and transforms it into a sequence of tokens. It recognizes and categorizes various elements such as keywords, operators, identifiers, and other components essential for further processing.

1. **Software Tools**
   1. **Computer Program**

A compiler is a specialized program designed to convert source code into machine code, verifying the accuracy of both syntax and semantics in the process.

* 1. **Programming Language**

Lexical analyzers are commonly developed using programming languages such as Python, C, or Java. The specific implementation outlined in this document is coded in C.

1. **Implementation of a Lexical Analyzer**

Here is my c code with explanation written as comment

/\*

 \* Lexical Analyzer Project

 \* ------------------------

 \* This program is a simple lexical analyzer for arithmetic expressions.

 \* It reads input from the user, tokenizes the expression into meaningful units

 \* like identifiers, integer literals, operators, and parentheses, and prints

 \* each token along with its lexeme.

 \* The goal is to simulate part of a compiler's front-end (the scanner).

 \*/

#include <stdio.h>          // Standard input/output functions

#include <ctype.h>          // Functions for character classification (e.g., isalpha, isdigit)

#include <string.h>         // String manipulation functions (e.g., strcpy)

/\* Global declarations \*/

/\* Variables \*/

int charClass;              // Stores the character class (LETTER, DIGIT, UNKNOWN, or EOF)

char lexeme[100];           // Array to hold the current lexeme (token text)

char nextChar;              // The current character being analyzed

int lexLen;                 // The length of the current lexeme

int token;                  // The current token

int nextToken;              // The next token to be processed

char inputBuffer[100];      // Buffer to hold the input line

int inputPos = 0;           // Current position in the input buffer

/\* Function declarations \*/

void addChar();             // Adds nextChar to lexeme

void getChar();             // Gets the next character and classifies it

void getNonBlank();         // Skips whitespace characters

int lex();                  // Main lexical analyzer function

/\* Character classes \*/

#define LETTER 0            // Letter character

#define DIGIT 1             // Digit character

#define UNKNOWN 99          // Unknown character type (operators or others)

#define EOF -1              // End of file/input

/\* Token codes \*/

#define INT\_LIT 10          // Integer literal

#define IDENT 11            // Identifier

#define ASSIGN\_OP 20        // Assignment operator '='

#define ADD\_OP 21           // Addition operator '+'

#define SUB\_OP 22           // Subtraction operator '-'

#define MULT\_OP 23          // Multiplication operator '\*'

#define DIV\_OP 24           // Division operator '/'

#define LEFT\_PAREN 25       // Left parenthesis '('

#define RIGHT\_PAREN 26      // Right parenthesis ')'

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* main driver \*/

int main() {

    printf("Enter input: "); // Prompt user to enter an expression

    while (fgets(inputBuffer, sizeof(inputBuffer), stdin)) { // Read input line from user

        inputPos = 0;        // Reset input position to start

        getChar();           // Initialize nextChar and charClass

        do {

            lex();           // Perform lexical analysis on the input

        } while (nextToken != EOF); // Repeat until end of input

        printf("Enter input: "); // Prompt again after processing line

    }

    return 0;

}

/\* Main Function Block:

   This loop reads input lines one by one, analyzes them using lex(),

   and keeps going until EOF is encountered. It tokenizes each part of the line.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* lookup - a function to lookup operators and parentheses and return the token \*/

int lookup(char ch) {

    switch (ch) {

        case '(':               // If character is '('

            addChar();          // Add it to the lexeme

            nextToken = LEFT\_PAREN;

            break;

        case ')':               // If character is ')'

            addChar();

            nextToken = RIGHT\_PAREN;

            break;

        case '+':               // If character is '+'

            addChar();

            nextToken = ADD\_OP;

            break;

        case '-':               // If character is '-'

            addChar();

            nextToken = SUB\_OP;

            break;

        case '\*':               // If character is '\*'

            addChar();

            nextToken = MULT\_OP;

            break;

        case '/':               // If character is '/'

            addChar();

            nextToken = DIV\_OP;

            break;

        case '=':               // If character is '='

            addChar();

            nextToken = ASSIGN\_OP;

            break;

        default:                // Any unknown character

            addChar();

            nextToken = UNKNOWN; // Mark as unknown token

            break;

    }

    return nextToken;

}

/\* lookup():

   This function identifies single-character operators and parentheses.

   It maps each symbol to its respective token code.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* addChar - a function to add nextChar to lexeme \*/

void addChar() {

    if (lexLen <= 98) {                 // Check to prevent buffer overflow

        lexeme[lexLen++] = nextChar;   // Add character to lexeme

        lexeme[lexLen] = '\0';         // Null-terminate lexeme string

    } else {

        printf("Error - lexeme is too long \n"); // Error if lexeme is too long

    }

}

/\* addChar():

   Adds the current character to the lexeme and ensures string is null-terminated.

   Also avoids exceeding the array limit.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* getChar - a function to get the next character of input and determine its character class \*/

void getChar() {

    if (inputBuffer[inputPos] != '\0' && inputBuffer[inputPos] != '\n') { // Check if end of buffer or newline

        nextChar = inputBuffer[inputPos++];  // Get the next character

        if (isalpha(nextChar))               // Check if it's a letter

            charClass = LETTER;

        else if (isdigit(nextChar))          // Check if it's a digit

            charClass = DIGIT;

        else

            charClass = UNKNOWN;             // Otherwise, it's unknown

    } else {

        charClass = EOF;                     // No more input

    }

}

/\* getChar():

   Reads the next character from the input buffer and determines its class

   (letter, digit, or unknown). Marks EOF if the end of input is reached.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* getNonBlank - a function to call getChar until it returns a non-whitespace character \*/

void getNonBlank() {

    while (isspace(nextChar))  // Keep skipping whitespace

        getChar();

}

/\* getNonBlank():

   Skips over all whitespace characters in the input (spaces, tabs, etc.).

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* lex - a simple lexical analyzer for arithmetic expressions \*/

int lex() {

    lexLen = 0;                // Reset lexeme length

    getNonBlank();            // Skip any whitespace

    switch (charClass) {

        /\* Parse identifiers \*/

        case LETTER:

            addChar();        // Add first letter to lexeme

            getChar();        // Get next character

            while (charClass == LETTER || charClass == DIGIT) {

                addChar();    // Keep adding letters/digits to identifier

                getChar();

            }

            nextToken = IDENT; // Token is an identifier

            break;

        /\* Parse integer literals \*/

        case DIGIT:

            addChar();        // Add first digit to lexeme

            getChar();

            while (charClass == DIGIT) {

                addChar();    // Continue if next chars are digits

                getChar();

            }

            nextToken = INT\_LIT; // Token is an integer

            break;

        /\* Parentheses and operators \*/

        case UNKNOWN:

            lookup(nextChar); // Look up token type for operator/paren

            getChar();

            break;

        /\* EOF \*/

        case EOF:

            nextToken = EOF;         // Set token as EOF

            strcpy(lexeme, "EOF");   // Set lexeme to string "EOF"

            break;

    }

    printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme); // Print token info

    return nextToken;                // Return token to caller

}

/\* lex():

   The lexical analyzer function breaks the input string into tokens.

   It handles:

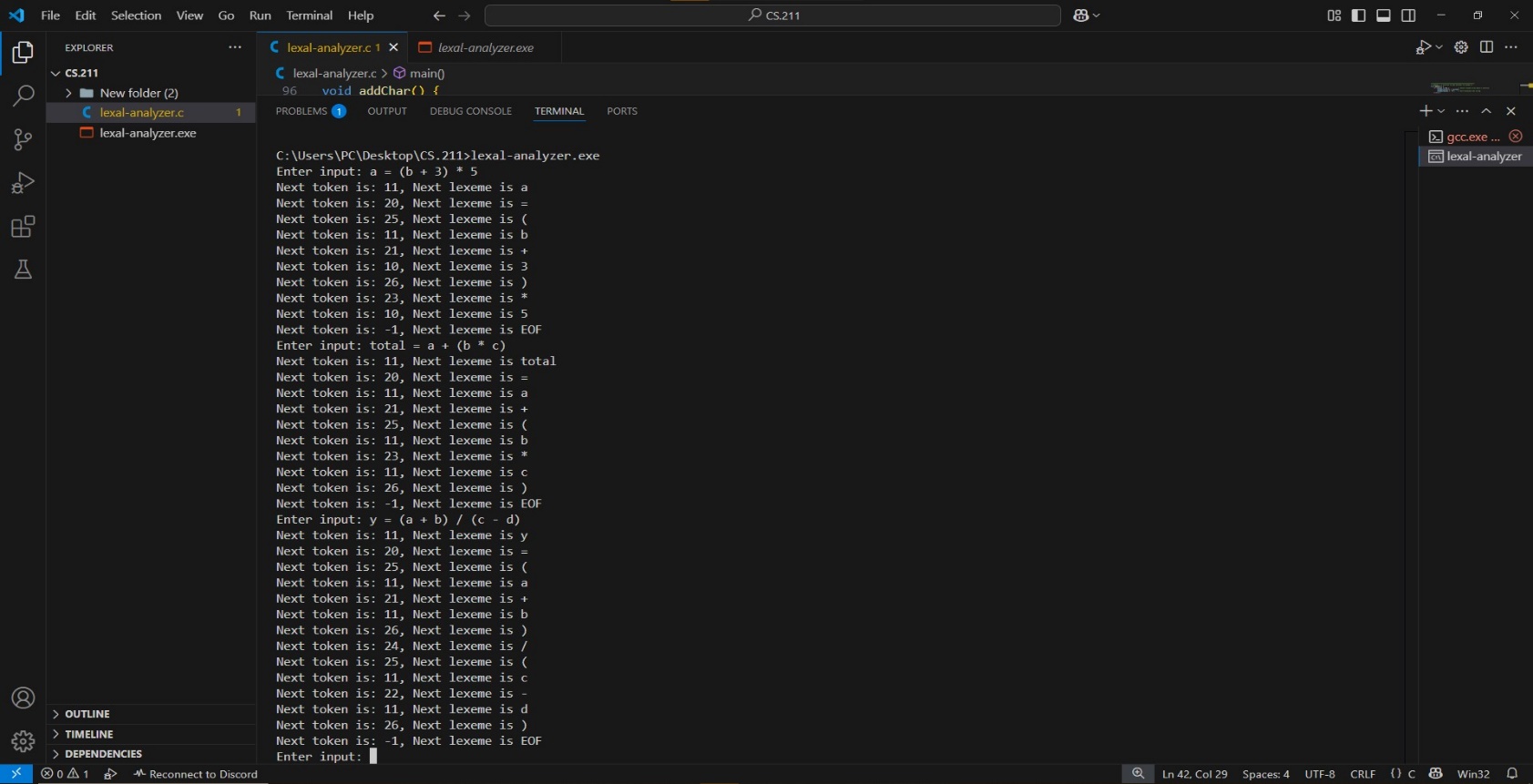
   - Identifiers (starting with a letter, followed by letters/digits)

   - Integer literals (continuous digits)

   - Operators and parentheses via lookup

   - EOF detection

\*/

**The output of the code:**

1. **References**

 Parr, T. (2022). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages with Python*.

 Parsons, D. (2021). *Introduction to Compiler Design*.

[Lexical analysis - Wikipedia](https://en.wikipedia.org/wiki/Lexical_analysis)

[Lexical Analysis in Compiler Design](https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm)

[Introduction of Lexical Analysis | GeeksforGeeks](https://www.geeksforgeeks.org/introduction-of-lexical-analysis/)