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**

Name Student:shahd omar

ID:200044386.

**Under Supervision**

Name of Doctor:Nehal

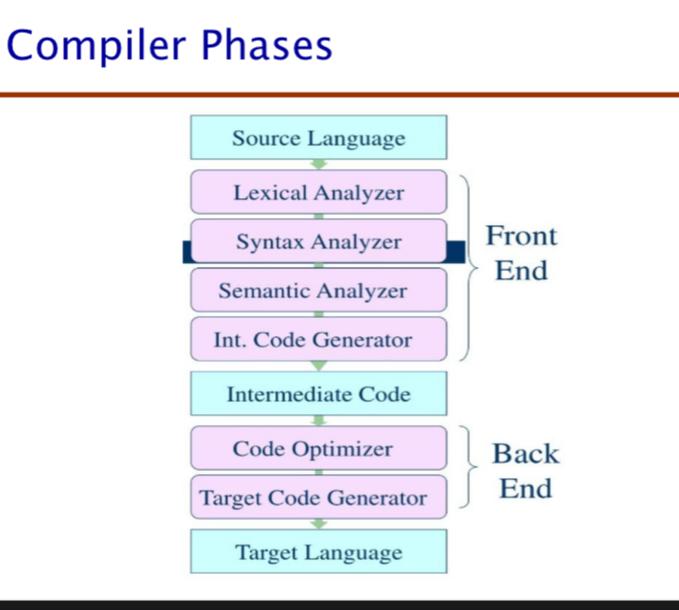
Name of T.A:eslam

1. **Introduction).**

The lexical analyzer is an important part of the compilation. Its job is to accept the source code and divide it into tokens, which include keywords, variables, and operators. The tokens simplify the task of comprehending the code for the compiler.

In this project, I have used a C program which reads simple arithmetic expressions from a file, parses them, and translates them to tokens in order to make the rest of the compilation steps easier (like parsing or code generation).

* 1. **Phases of Compiler**



1.phases: A compiler operates in multiple phases, each responsible for a different aspect of translation.

1. Phase Description
2. Lexical Analysis
3. Syntax Analysis
4. Semantic Analysis
5. Intermediate Code Generation Produces an intermediate representation. Code Optimization Improves performance by reducing redundancies.
6. Converts source code into tokens.
7. Checks grammatical correctness and builds a parse tree. Ensures valid meaning and detects type errors.
8. Code Generation Converts optimized code to machine code. Error Handling Identifies and reports errors.
9. Lexical Analyzer
10. Phase Description
11. **Lexical Analyzer**

A Lexical Analyzer: reads the source code character by character and groups them into meaningful tokens.

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

Visual studio

* 1. **Programming Language:**

C++

1. **Implementation of a Lexical Analyzer:**

#include <iostream> // Includes the input/output stream library (used for cout, etc.)

#include <cctype> // Includes character handling functions (isalpha, isdigit, isspace, etc.)

#include <string> // Includes the string class for handling strings

using namespace std; // Allows using standard names (like cout, string) without prefixing with std::

#define LETTER 0 // Constant representing a letter character

#define DIGIT 1 // Constant representing a digit character

#define UNKNOWN 99 // Constant for unknown (symbols/operators)

#define END\_OF\_FILE -1 // Constant for end of file/input

// Token type definitions (representing different types of tokens)

#define INT\_LIT 10 // Integer literal (e.g., 5, 123)

#define IDENT 11 // Identifier (e.g., variable names)

#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 ')'

// Global variables used in the lexical analyzer

int charClass; // Class of the current character (LETTER, DIGIT, etc.)

string lexeme; // Current lexeme (token string)

char nextChar; // Next character from the input

int lexLen; // Length of the current lexeme

int token; // Current token type

int nextToken; // Next token type

string inputLine; // Input line to be analyzed

int lineIndex = 0; // Current index in the input line

// Function declarations

void addChar(); // Adds the current character to the lexeme

void getChar(); // Gets the next character from input

void getNonBlank(); // Skips whitespace characters

int lookup(char ch); // Determines token type for single-character operators/symbols

int lex(); // Lexical analyzer function

/\* Adds the current character to the lexeme \*/

void addChar() {

if (lexLen <= 98) { // Check if lexeme length is within the limit

lexeme += nextChar; // Append current character to the lexeme

lexLen++; // Increase the lexeme length counter

} else {

cout << "Error - lexeme is too long" << endl; // Print error if too long

}

}

/\* Gets the next character from the input line \*/

void getChar() {

if (lineIndex < inputLine.length()) { // Check if more characters are available

nextChar = inputLine[lineIndex++]; // Get next character and advance index

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

charClass = LETTER; // Assign class LETTER

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

charClass = DIGIT; // Assign class DIGIT

else

charClass = UNKNOWN; // Otherwise, assign class UNKNOWN

} else {

charClass = END\_OF\_FILE; // No more characters; mark end of input

nextChar = '\0'; // Set nextChar to null character

}

}

/\* Skips blank spaces \*/

void getNonBlank() {

while (isspace(nextChar)) // While current character is a space or tab

getChar(); // Get the next character

}

/\* Looks up and returns the token code for a given symbol character \*/

int lookup(char ch) {

switch (ch) {

case '(': addChar(); nextToken = LEFT\_PAREN; break; // Left parenthesis

case ')': addChar(); nextToken = RIGHT\_PAREN; break; // Right parenthesis

case '+': addChar(); nextToken = ADD\_OP; break; // Plus operator

case '-': addChar(); nextToken = SUB\_OP; break; // Minus operator

case '\*': addChar(); nextToken = MULT\_OP; break; // Multiplication

case '/': addChar(); nextToken = DIV\_OP; break; // Division

case '=': addChar(); nextToken = ASSIGN\_OP; break; // Assignment operator

default: addChar(); nextToken = UNKNOWN; break; // Unknown character

}

return nextToken; // Return the token type

}

/\* The lexical analyzer function that identifies the next token \*/

int lex() {

lexLen = 0; // Reset lexeme length

lexeme = ""; // Clear the lexeme string

getNonBlank(); // Skip any whitespace

switch (charClass) {

case LETTER: // If character is a letter

addChar(); // Add to lexeme

getChar(); // Move to next character

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

addChar(); // Keep adding letters/digits

getChar();

}

nextToken = IDENT; // This is an identifier

break;

case DIGIT: // If character is a digit

addChar(); // Add to lexeme

getChar(); // Move to next character

while (charClass == DIGIT) {

addChar(); // Keep adding digits

getChar();

}

nextToken = INT\_LIT; // This is an integer literal

break;

case UNKNOWN: // If character is unknown (probably an operator)

lookup(nextChar); // Try to match it to a known token

getChar(); // Move to the next character

break;

case END\_OF\_FILE: // If we reached the end of input

nextToken = END\_OF\_FILE; // Set token as END\_OF\_FILE

lexeme = "EOF"; // Set lexeme text to "EOF"

break;

}

// Output the token and lexeme found

cout << "Next token is: " << nextToken << ", Next lexeme is: " << lexeme << endl;

return nextToken; // Return the token type

}

/\* Main function: entry point of the program \*/

int main() {

inputLine = "Y=5+Z/4"; // The input line to analyze

getChar(); // Start by reading the first character

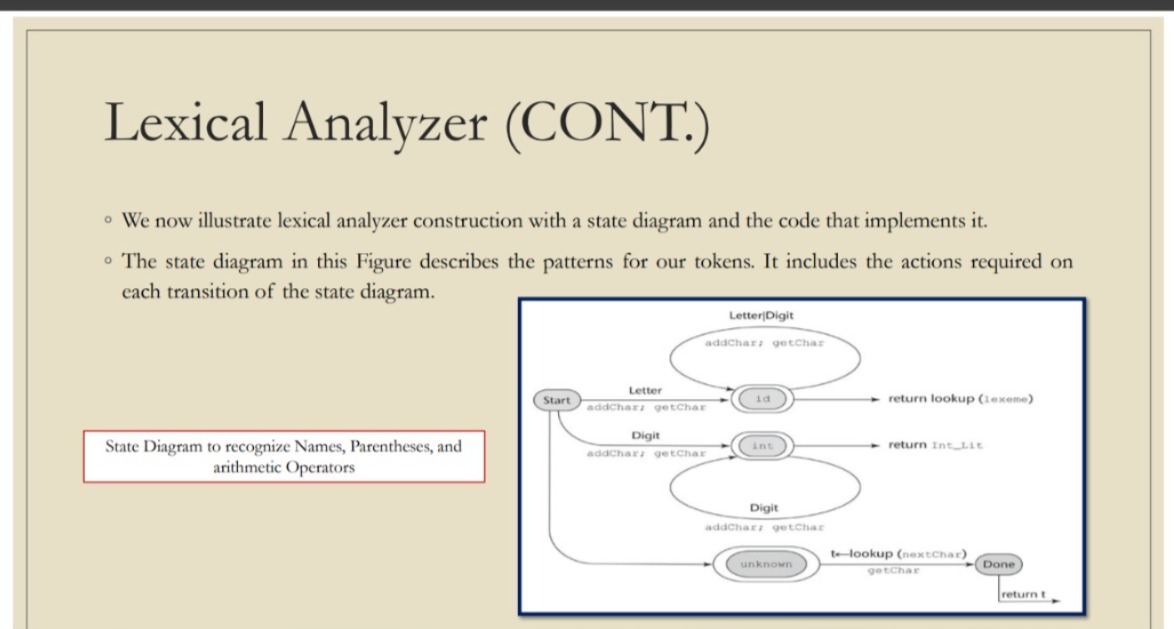
while (nextToken != END\_OF\_FILE) { // Continue until end of input

lex(); // Analyze the next token

}

return 0; // Program ends successfully

}



this state diagram is basically a map for the lexical analyzer—it’s like a guide that tells the lexer how to break down an input string (like “Y=5+Z/4”) into smaller pieces called tokens, such as variable names, numbers, or operators. Think of it as a little machine that moves through different “stages” (or states) to figure out what each piece of the input is. Let me walk you through it:

Start State: This is where the lexer kicks off. It’s like, “Alright, let’s start reading the input!”

addChar\_getChar State: Once the lexer finds something interesting, it moves here. In this state, it’s like, “Oh, I found a character! Let me add it to my token (which we call a lexeme) and grab the next character to see what’s up.” It keeps doing this to build the token.

Done State: This is the finish line! When the lexer is like, “I’m done building this token, here it is!” it moves to this state and hands over the token (like “Y” or “5” or “+”).

Now, the arrows between these states show how the lexer decides where to go next, depending on what character it sees:

If it sees a Letter (like “Y” or “Z”): From the Start state, it jumps to addChar\_getChar. It starts building an identifier (a fancy word for a variable name, like “Y”). If the next character is also a letter or a digit, it loops back to keep adding to the identifier. Once it hits something else (like “=”), it’s done and moves to the Done state, saying, “Here’s an identifier, it’s called IDENT!”

If it sees a Digit (like “5” or “4”): From Start, it also goes to addChar\_getChar. It starts building a number (like “5”). If the next character is another digit, it keeps adding to the number. When it’s done (like when it sees a “+”), it moves to Done and says, “Here’s a number, it’s called INT\_LIT!”

If it sees something else (like “+”, “-”, “/”, “=”, “(“, or “)”): From Start, it doesn’t need to build anything long—it just calls a function called lookup(nextChar) to figure out what this symbol is (like “+” means ADD\_OP). Then it goes straight to Done and returns that token.

This diagram is basically a visual version of the lex() function in your code. It’s showing how the lexer steps through the input, one character at a time, and decides what each piece means—like breaking a sentence into words and figuring out if each word is a noun, verb, or something else! Does that make sense?

**5.References**

Geekforgeeks,Wikipedia-lexical Analysis