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**

Abdallah Mojibo Qaid

200036055

**Under Supervision**

Name of Doctor

Name of T. A.

1. **Introduction**
   1. **Phases of Compiler**

**Lexical Analysis** – Breaks source code into tokens (like words), removes whitespace/comments.

**Syntax Analysis** – Checks code structure using grammar, builds a syntax tree.

**Semantic Analysis** – Validates logic (e.g., type checking, undeclared variables), updates symbol table.

**Intermediate Code Generation** – Converts code into a platform-independent intermediate form.

**Code Optimization** – Improves performance by removing unnecessary or redundant code.

**Code Generation** – Generates low-level machine or assembly code for the target system.

**Linking and Assembly** – Links compiled code with libraries, produces the final executable.

1. **Lexical Analyzer**

A lexical analyzer is the first phase of a compiler or interpreter. Its main job is to scan the source code, character by character, and group them into tokens.

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

Visual Studio Code

* 1. **Programming Language**

C++

1. **Implementation of a Lexical Analyzer**

// lexical\_analyzer.cpp - A simple lexical analyzer using a state transition diagram in C++

// This program reads from a hardcoded input string and uses a state transition diagram

// to recognize tokens like identifiers, integer literals, arithmetic operators, and parentheses.

#include <iostream> // For standard input/output (std::cout)

#include <cctype> // For character classification functions (std::isalpha, std::isdigit, std::isspace)

#include <string> // For using the std::string class

// Global Declarations

int charClass; // Variable to store the character class (LETTER, DIGIT, UNKNOWN)

std::string lexeme; // String to accumulate characters into a token (lexeme)

char nextChar; // Variable to hold the current character being processed

int lexLen; // Length of the current lexeme

int nextToken; // Token code of the token just recognized

// Hardcoded input string and index to traverse it

std::string inputString = "A+B\*(C-1)"; // The input expression to be analyzed

size\_t inputIndex = 0; // Current index into the input string

// Character Classes (constants)

#define LETTER 0 // Represents alphabetic characters

#define DIGIT 1 // Represents numeric characters

#define UNKNOWN 99 // Represents any character that is neither a letter nor a digit

// Token Codes (constants)

#define INT\_LIT 10 // Token code for integer literals

#define IDENT 11 // Token code for identifiers

#define ASSIGN\_OP 20 // (Not used in this example) Token code for assignment operator

#define ADD\_OP 21 // Token code for '+' operator

#define SUB\_OP 22 // Token code for '-' operator

#define MULT\_OP 23 // Token code for '\*' operator

#define DIV\_OP 24 // Token code for '/' operator

#define LEFT\_PAREN 25 // Token code for '('

#define RIGHT\_PAREN 26 // Token code for ')'

// Function declarations

void addChar(); // Adds nextChar to the lexeme

void getChar(); // Gets the next character from inputString and sets its class

void getNonBlank(); // Skips over whitespace characters

int lookup(char ch); // Determines the token code for single-character tokens

int lex(); // Main lexical analyzer function

// Main function: drives the lexical analyzer

int main() {

getChar(); // Initialize by reading the first character from inputString

// Loop until end of input (EOF represented by -1) is reached

do {

lex(); // Process the next token

} while (nextToken != -1);

return 0; // End of program

}

// lookup() function: returns the token code for operators and parentheses

int lookup(char ch) {

switch(ch) { // Check the character against known single-character tokens

case '(':

addChar(); // Add '(' to the lexeme

nextToken = LEFT\_PAREN; // Set token code for left parenthesis

break;

case ')':

addChar(); // Add ')' to the lexeme

nextToken = RIGHT\_PAREN; // Set token code for right parenthesis

break;

case '+':

addChar(); // Add '+' to the lexeme

nextToken = ADD\_OP; // Set token code for addition operator

break;

case '-':

addChar(); // Add '-' to the lexeme

nextToken = SUB\_OP; // Set token code for subtraction operator

break;

case '\*':

addChar(); // Add '\*' to the lexeme

nextToken = MULT\_OP; // Set token code for multiplication operator

break;

case '/':

addChar(); // Add '/' to the lexeme

nextToken = DIV\_OP; // Set token code for division operator

break;

default:

addChar(); // Add the unrecognized character to the lexeme

nextToken = -1; // Set token code to -1 (treated as EOF for simplicity)

break;

}

return nextToken; // Return the token code determined by the lookup

}

// addChar() function: adds the current character (nextChar) to the lexeme string

void addChar() {

lexeme.push\_back(nextChar); // Append nextChar to lexeme

lexLen++; // Increment the length of the lexeme

}

// getChar() function: fetches the next character from inputString and categorizes it

void getChar() {

// Check if there are still characters left in inputString

if (inputIndex < inputString.size()) {

nextChar = inputString[inputIndex++]; // Get the next character and increment inputIndex

// Determine the character class using cctype functions

if (std::isalpha(nextChar))

charClass = LETTER; // Character is a letter

else if (std::isdigit(nextChar))

charClass = DIGIT; // Character is a digit

else

charClass = UNKNOWN; // Character is neither a letter nor a digit

} else {

// No more characters: set charClass to -1 to represent EOF

charClass = -1;

}

}

// getNonBlank() function: skips over any whitespace characters in the input

void getNonBlank() {

// While nextChar is a whitespace character, call getChar() to skip it

while (std::isspace(nextChar))

getChar();

}

// lex() function: the core lexical analyzer that identifies tokens based on character classes

int lex() {

// Clear the current lexeme and reset its length

lexeme.clear();

lexLen = 0;

getNonBlank(); // Skip any whitespace

// Determine what type of token to build based on the current character's class

switch(charClass) {

// If the character is a letter, start an identifier token

case LETTER:

addChar(); // Add the initial letter to lexeme

getChar(); // Get the next character

// Continue adding letters or digits (allowed in identifiers)

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

addChar(); // Append the character to lexeme

getChar(); // Move to the next character

}

nextToken = IDENT; // Set token code for identifier

break;

// If the character is a digit, start an integer literal token

case DIGIT:

addChar(); // Add the initial digit to lexeme

getChar(); // Get the next character

// Continue adding digits to form the complete number

while (charClass == DIGIT) {

addChar(); // Append the digit to lexeme

getChar(); // Move to the next character

}

nextToken = INT\_LIT; // Set token code for integer literal

break;

// For non-alphanumeric characters (operators, parentheses)

case UNKNOWN:

lookup(nextChar); // Use lookup() to determine the token for the operator or parenthesis

getChar(); // Get the next character after processing the unknown token

break;

// End-of-file: when there are no more characters to process

case -1:

nextToken = -1; // Set token code to -1 to indicate EOF

lexeme = "EOF"; // For clarity, set the lexeme to the string "EOF"

break;

}

// Output the token code and the lexeme for debugging purposes

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

return nextToken; // Return the token code for potential further processing

}

1. **References**

Concepts of Programming Languages - Sebesta - E12 (P 298 – P 302)

**Important Note: -**

Technical reports include a mixture of text, tables, and figures. Consider how you can present the information best for your reader. Would a table or figure help to convey your ideas more effectively than a paragraph describing the same data?

Figures and tables should: -

* Be numbered
* Be referred to in-text, e.g. *In Table 1*…, and
* Include a simple descriptive label - above a table and below a figure.