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:

**JANA WALEED ATTA**

ID:

**200045662**

**Under Supervision**

Dr.Nehal

Eng.Fares

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

**The lexical analyzer is the first phase of the process of analyzing the source code in a compiler. The primary goal is to convert the input stream (source code) into a sequence of meaningful tokens, such as identifiers, keywords, literals, operators, and others. The lexical analyzer acts as a bridge between the source code and later stages of the compiler, which is implemented in a programming language like C++, facilitating syntax analysis and semantic analysis. In this project, a lexical analyzer was developed using C++. It reads a text file (**[**front.in**](https://front.in/)**), recognizes tokens accurately, and prints the type and value of each token. This project serves as a practical application to understand the compiler phases and their mechanics.**

1. **Lexical Analyzer**

#include <iostream>

#include <fstream>

#include <cctype>

#include <string>

using namespace std;

// Define token and character classes

enum CharClass { LETTER, DIGIT, UNKNOWN, END\_OF\_FILE = -1 };

enum TokenType {

INT\_LIT = 10, IDENT = 11,

ASSIGN\_OP = 20, ADD\_OP = 21, SUB\_OP = 22,

MULT\_OP = 23, DIV\_OP = 24,

LEFT\_PAREN = 25, RIGHT\_PAREN = 26

};

// Global variables

CharClass charClass;

string lexeme;

char nextChar;

int nextToken;

ifstream inFile;

// Function to add a character to the current lexeme

void addChar() {

lexeme += nextChar;

}

// Function to get the next character from the file and classify it

void getChar() {

if (inFile.get(nextChar)) {

if (isalpha(nextChar))

charClass = LETTER;

else if (isdigit(nextChar))

charClass = DIGIT;

else

charClass = UNKNOWN;

} else {

charClass = END\_OF\_FILE;

}

}

// Function to skip whitespace

void getNonBlank() {

while (isspace(nextChar))

getChar();

}

// Function to determine the token of symbols/operators

int lookup(char ch) {

switch (ch) {

case '(': addChar(); return LEFT\_PAREN;

case ')': addChar(); return RIGHT\_PAREN;

case '+': addChar(); return ADD\_OP;

case '-': addChar(); return SUB\_OP;

case '\*': addChar(); return MULT\_OP;

case '/': addChar(); return DIV\_OP;

default: addChar(); return END\_OF\_FILE;

}

}

// The main lexical analyzer function

int lex() {

lexeme = "";

getNonBlank();

switch (charClass) {

case LETTER:

addChar();

getChar();

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

addChar();

getChar();

}

nextToken = IDENT;

break;

case DIGIT:

addChar();

getChar();

while (charClass == DIGIT) {

addChar();

getChar();

}

nextToken = INT\_LIT;

break;

case UNKNOWN:

nextToken = lookup(nextChar);

getChar();

break;

case END\_OF\_FILE:

lexeme = "EOF";

nextToken = END\_OF\_FILE;

break;

}

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

return nextToken;

}

// Main program

int main() {

inFile.open("front.in");

if (!inFile) {

cerr << "ERROR - cannot open front.in" << endl;

return 1;

}

getChar();

do {

lex();

} while (nextToken != END\_OF\_FILE);

inFile.close();

return 0;

}

**Software Tools**

* 1. **Computer Program**

A **C++ program** that reads code from front.in.

* 1. **Programming Language**

**C++ (POWERFUL, EFFICIENT, COMPLEXT).**

1. **Implementation of a Lexical Analyzer**

**#include <iostream>**

Includes standard input/output stream library for cout and cin.

**#include <fstream>**

Includes file stream library to handle file input.

**#include <cctype>**

Includes character classification functions like isalpha and isdigit.

**#include <string**>

Includes string library to use std::string.

**using namespace std;**

Allows use of standard library names without prefixing with std::.

**int charClass;**

Stores the class of the current character (LETTER, DIGIT, UNKNOWN, or EOF).

**string lexeme = "";**

Holds the current lexeme (token being built).

**char nextChar;**

Stores the next character read from the input file.

**int lexLen;**

Stores the length of the current lexeme.

**int token;**

Stores the current token type (not used directly here).

**int nextToken;**

Stores the next token value to be returned.

**ifstream in\_fp;**

Input file stream to read the source code.

**#define LETTER 0**

Constant to represent alphabetic characters.

**#define DIGIT 1**

Constant to represent numeric characters.

**#define UNKNOWN 99**

Constant to represent unrecognized characters or symbols.

**#define INT\_LIT 10**

Token code for integer literals.

**#define IDENT 11**

Token code for identifiers.

**#define ASSIGN\_OP 20**

Token code for assignment operator (not used here).

**#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 ')'.

**void addChar() {**

Adds the current character to the lexeme string.

**if (lexeme.size() <= 98) {**

**lexeme += nextChar;**

Append the character to lexeme.

**} else {**

**cout << "Error - lexeme is too long" << endl;**

Show error if lexeme is too long.

**}**

**}**

**void getChar() {**

Reads the next character from the file and classifies it.

**if (in\_fp.get(nextChar)) {**

Reads character and checks if read is successful.

**if (isalpha(nextChar))**

**charClass = LETTER;**

Set class to LETTER if alphabet.

**else if (isdigit(nextChar))**

**charClass = DIGIT;**

Set class to DIGIT if number.

**else**

**charClass = UNKNOWN;**

Set class to UNKNOWN for other characters.

**} else {**

**charClass = EOF;**

Set class to EOF if end of file reached.

**}**

**}**

**void getNonBlank() {**

Skips whitespace characters in input.

**while (isspace(nextChar)) {**

**getChar();**

Keep reading until non-space character is found.

**}**

**}**

**int lookup(char ch) {**

Determines the token type for a given symbol/operator.

**switch (ch) {**

**case '(': addChar(); nextToken = LEFT\_PAREN; break;**

Handle '('.

**case ')': addChar(); nextToken = RIGHT\_PAREN; break;**

Handle ')'.

**case '+': addChar(); nextToken = ADD\_OP; break;**

Handle '+'.

**case '-': addChar(); nextToken = SUB\_OP; break;**

Handle '-'.

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

// Handle '\*'.

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

Handle '/'.

**default: addChar(); nextToken = EOF; break;**

Handle unknown character.

**}**

**return nextToken;**

Return the determined token.

**}**

**int lex() {**

Performs lexical analysis and returns a token.

**lexeme = "";**

Reset the lexeme string.

**getNonBlank();**

Skip any white spaces.

**switch (charClass) {**

**case LETTER:**

If character is a letter, it's likely an identifier.

**addChar(); getChar();**

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

**addChar(); getChar();**

Keep adding letters and digits.

**}**

**nextToken = IDENT;**

Set token type to identifier.

**break;**

**case DIGIT:**

If character is digit, it’s an integer literal.

**addChar(); getChar();**

**while (charClass == DIGIT) {**

**addChar(); getChar();**

Keep adding digits.

**}**

**nextToken = INT\_LIT;**

Set token type to integer literal.

**break;**

**case UNKNOWN:**

If character is a symbol/operator.

**lookup(nextChar); getChar();**

Look it up and read next

**character.**

**break;**

**case EOF:**

If end of file is reached.

**nextToken = EOF;**

Set token to EOF.

**lexeme = "EOF";**

Set lexeme to string "EOF".

**break;**

**}**

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

Print the result.

**return nextToken;**

Return the token.

**}**

**int main() {**

Entry point of the program.

**in\_fp.open("front.in");**

Open the input file "front.in".

**if (!in\_fp) {**

**cout << "ERROR - cannot open front.in" << endl;**

Print error if file cannot be opened.

**return 1;**

Exit program with error code.

**}**

**getChar();**

Read the first character from the file.

**do {**

**lex();**

Call lexical analyzer until end of file.

**} while (nextToken != EOF);**

Repeat until EOF token is returned.

**return 0;**

Exit program.

1. **References**

**Concepts of Programming Languages - Sebesta - E12**