MISR UNIVERSITY
FOR SCIENCE & TECHNOLOGY
College of Information Technology

جامعة مصر
للعلوم والتكنولوجيا
كلية تكنولوجيا المعلومات

LEXICAL ANALYZER

Build Scanner

Prepared By
Noor Abdelraouf Abdalla
200039544

Under Supervision
Nehal Abdelsalam
Name of T. A.

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.

1. Phases of Compiler

1. Input String Initialization

The program begins by setting a hardcoded string,"int x = 8 - 4 (y*z) ; ", as the input to analyze. This is the raw source code that will be processed by the lexer.

2. Character Classification

The program processes the input one character at a time.

- **LETTER**: If the character is a letter (A-Z, a-z).
- **DIGIT**: If the character is a number (0-9).

3. Token Recognition

The program processes each character in the string and forms tokens. Depending on the character type

4. Identifying Operators and Symbols

This phase handles special characters such as operators (+, -, *, /), punctuation (;), and brackets ((,)).

5. Token Type Output

this function prints the **token type** and its **lexeme**.

2. Lexical Analyzer

```
#include <stdio.h>     // standard input/output library
#include <ctype.h>     // character handling library

/* Character classes */
#define LETTER 0       // alphabetic characters
#define DIGIT 1        // numeric characters
#define UNKNOWN 99     // for operators and punctuations

/* Token codes */
#define INT_LITERAL 10     // integer literal
#define IDENTIFIER 11      // variable names
#define ASSIGN_OP 20
#define PLUS_OP 21
#define MINUS_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_BRACKET 25
#define RIGHT_BRACKET 26
#define SEMICOLON 27
#define END_OF_STRING -1

/* Global variables */
int characterType;              // type of current character
char currentWord[100];          // current lexeme
char currentChar;               // current character
int wordLength;                 // length of the lexeme
int nextToken;                  // token type of current word
const char* inputString;        // pointer to input string
int currentPos = 0;             // index for input string

/* Function Prototypes */
void appendChar();
void readChar();
void skipWhitespace();
int getToken();
```

```
int identifyChar(char symbol);
void printTokenType(int token);

/* Main Program */
int main() {
    // Input string to analyze
    inputString = "int x = 8 - 4 ( y * z ) ;";

    readChar(); // Initialize the first character

    do {
        getToken();
    } while (nextToken != END_OF_STRING);

    return 0;
}

/* Append character to current word */
void appendChar() {
    if (wordLength <= 98) {
        currentWord[wordLength++] = currentChar;
        currentWord[wordLength] = '\0';
    } else {
        printf("Error - word is too long!\n");
    }
}

/* Read next character and classify it */
void readChar() {
    if (inputString[currentPos] != '\0') {
        currentChar = inputString[currentPos++];
        if (isalpha(currentChar))
            characterType = LETTER;
        else if (isdigit(currentChar))
            characterType = DIGIT;
        else
            characterType = UNKNOWN;
    } else {
```

```
        characterType = END_OF_STRING;
    }
}

/* Skip spaces and tabs */
void skipWhitespace() {
    while (isspace(currentChar)) {
        readChar();
    }
}

/* Identify symbols like operators/brackets */
int identifyChar(char symbol) {
    switch (symbol) {
        case '(':
            appendChar();
            nextToken = LEFT_BRACKET;
            break;
        case ')':
            appendChar();
            nextToken = RIGHT_BRACKET;
            break;
        case '+':
            appendChar();
            nextToken = PLUS_OP;
            break;
        case '-':
            appendChar();
            nextToken = MINUS_OP;
            break;
        case '*':
            appendChar();
            nextToken = MULT_OP;
            break;
        case '/':
            appendChar();
            nextToken = DIV_OP;
            break;
```

```
        case '=':
            appendChar();
            nextToken = ASSIGN_OP;
            break;
        case ';':
            appendChar();
            nextToken = SEMICOLON;
            break;
        default:
            appendChar();
            nextToken = END_OF_STRING;
            break;
    }
    return nextToken;
}

/* Main lexical analyzer function */
int getToken() {
    wordLength = 0;
    skipWhitespace();

    switch (characterType) {
        case LETTER:
            appendChar();
            readChar();
            while (characterType == LETTER || characterType == DIGIT)
{
                appendChar();
                readChar();
            }
            nextToken = IDENTIFIER;
            break;

        case DIGIT:
            appendChar();
            readChar();
            while (characterType == DIGIT) {
                appendChar();
```

```
            readChar();
        }
        nextToken = INT_LITERAL;
        break;

    case UNKNOWN:
        identifyChar(currentChar);
        readChar();
        break;

    case END_OF_STRING:
        nextToken = END_OF_STRING;
        currentWord[0] = 'E';
        currentWord[1] = 'O';
        currentWord[2] = 'S';
        currentWord[3] = '\0';
        break;
  }

  printTokenType(nextToken);
  printf("Token: %d | Lexeme: %s\n", nextToken, currentWord);

  return nextToken;
}

/* Print token type */
void printTokenType(int token) {
  switch (token) {
    case IDENTIFIER:
        printf("[IDENTIFIER] ");
        break;
    case INT_LITERAL:
        printf("[INT_LITERAL] ");
        break;
    case ASSIGN_OP:
        printf("[ASSIGN_OP] ");
        break;
    case PLUS_OP:
```

```
            printf("[PLUS_OP] ");
            break;
        case MINUS_OP:
            printf("[MINUS_OP] ");
            break;
        case MULT_OP:
            printf("[MULT_OP] ");
            break;
        case DIV_OP:
            printf("[DIV_OP] ");
            break;
        case LEFT_BRACKET:
            printf("[LEFT_BRACKET] ");
            break;
        case RIGHT_BRACKET:
            printf("[RIGHT_BRACKET] ");
            break;
        case SEMICOLON:
            printf("[SEMICOLON] ");
            break;
        case END_OF_STRING:
            printf("[END_OF_STRING] ");
            break;
        default:
            printf("[UNKNOWN] ");
            break;
    }
}
```

OUTPUT:

Output Clear

```
[IDENTIFIER] Token: 11 | Lexeme: int
[IDENTIFIER] Token: 11 | Lexeme: x
[ASSIGN_OP] Token: 20 | Lexeme: =
[INT_LITERAL] Token: 10 | Lexeme: 8
[MINUS_OP] Token: 22 | Lexeme: -
[INT_LITERAL] Token: 10 | Lexeme: 4
[LEFT_BRACKET] Token: 25 | Lexeme: (
[IDENTIFIER] Token: 11 | Lexeme: y
[MULT_OP] Token: 23 | Lexeme: *
[IDENTIFIER] Token: 11 | Lexeme: z
[RIGHT_BRACKET] Token: 26 | Lexeme: )
[SEMICOLON] Token: 27 | Lexeme: ;
[END_OF_STRING] Token: -1 | Lexeme: EOS
```

3. Computer Program

Online C compiler

4. Programming Language

C

5. Implementation of a Lexical Analyzer

STATEMENT : int x = 8 - 4 (y * z) ;

int	Reserved word
x	Identifier
=	Assign_op
8	Int_literal
-	minus_op
4	Int_literal
(	L_BRACE
y	Identifier
*	Mult_op
z	identifier
)	R_BRACE
;	semicolon

6. References

Book of Concepts of Programming Languages – Sebesta -E12