LEXICAL ANALYZER

Build Scanner

Prepared By

Student Name: Shahd Sayed Abdelbaky
Student ID: 200044963

Under Supervision

Name of Doctor: Nehal Abd El-salam Mohamed
Name of T. A: Fares Emad Eldin

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.

Token Value	Token Name	Description
10	INT_LIT	Integer literal
11	IDENT	Identifier (variable name)
20	ASSIGN_OP	Assignment operator (=)
21	ADD_OP	Addition operator (+)
22	SUB_OP	Subtraction operator (-)
23	MULT_OP	Multiplication operator (*)
24	DIV_OP	Division operator (/)
25	LEFT_PAREN	Left parenthesis '('
26	RIGHT_PAREN	Right parenthesis ')'

The code and explanation of the lexical analyzer:

```
#include <stdio.h>       // For printf function

#include <ctype.h>       // For isalpha() and isdigit() functions

#include <string.h>      // For string functions



#define LETTER 0         // Identifier character

#define DIGIT 1          // Digit character

#define UNKNOWN 99       // Unknown character

#define EOF_TOKEN -1     // End of 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 ')'
```

```
int charClass;           // Stores type of current character
char lexeme[100];        // Array to hold the lexeme
char nextChar;           // The next character read
int lexLen;              // Current length of lexeme
int token;               // Token value
int nextToken;           // Next token to return
int index = 0;           // Index to track position in input

char input[] = "(total / sum + 62)";  // The input expression to analyze

// Function declarations
void addChar();
void getChar();
void getNonBlank();
int lex();
int lookup(char ch);

// Adds nextChar to lexeme
void addChar() {
    if (lexLen <= 98) {
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = '\0';
    }
    else {
        printf("Error: lexeme is too long\n");
    }
}

// Gets the next character and classifies it
void getChar() {
    if (input[index] != '\0') {
        nextChar = input[index++];
        if (isalpha(nextChar))
            charClass = LETTER;
        else if (isdigit(nextChar))
            charClass = DIGIT;
        else
            charClass = UNKNOWN;
    }
    else {
        charClass = EOF_TOKEN;
    }
}

// Skips whitespace
void getNonBlank() {
    while (isspace(nextChar))
        getChar();
}

// Returns token code for operators and parentheses
int lookup(char ch) {
    switch (ch) {
    case '(':
        addChar();
        nextToken = LEFT_PAREN;
        break;
    case ')':
        addChar();
        nextToken = RIGHT_PAREN;

        break;

    case '+':

        addChar();

        nextToken = ADD_OP;

        break;

    case '-':

        addChar();

        nextToken = SUB_OP;

        break;

    case '*':

        addChar();

        nextToken = MULT_OP;

        break;

    case '/':

        addChar();

        nextToken = DIV_OP;

        break;

    default:

        addChar();
        nextToken = EOF_TOKEN;
        break;
    }
    return nextToken;
}


// Main lexical analyzer function
int lex() {
    lexLen = 0;
    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:
    lookup(nextChar);
    getChar();
    break;
case EOF_TOKEN:
    nextToken = EOF_TOKEN;
    strcpy(lexeme, "EOF");
    break;
}
printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);
return nextToken;
```

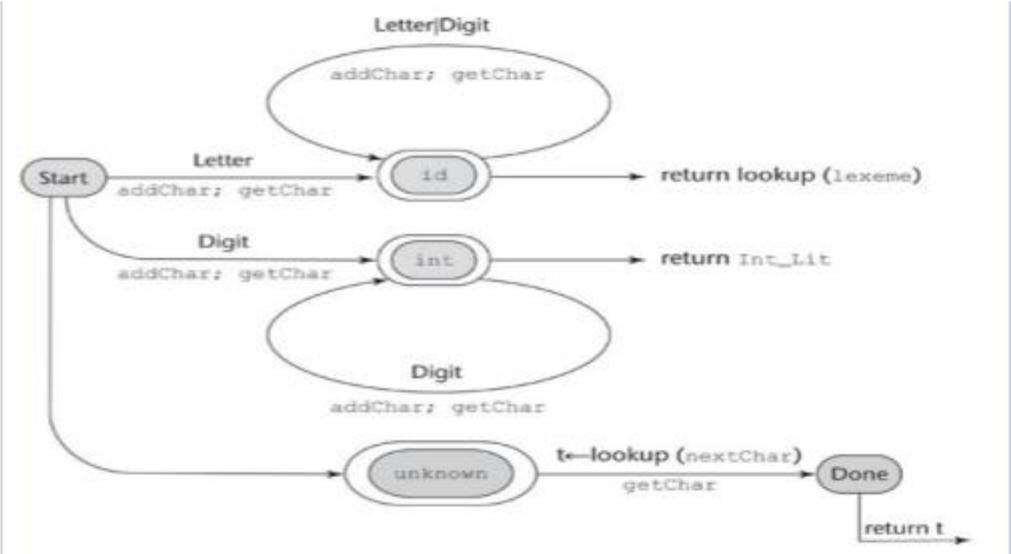
```
}


// Main function to drive the lexer
int main() {
    getChar();
    do {
        lex();
    } while (nextToken != EOF_TOKEN);
    return 0;
}
```

The code is written in C language and helps in **simplifying and analyzing** mathematical expressions in an organized way.

This lexical analyzer can be used as the first stage in the compilation process, where it creates a precise representation of what will happen in the later stages of the compiler.

So, the code breaks down the code into small, clear parts that are easier to handle in future stages of programming.

Diagram of Lexical Analysis Process
[Source Code] → [Lexical Analyzer] → [Tokens]


Letter|Digit
addChar; getChar
Start
Letter
addChar; getChar
id
return lookup (lexeme)
Digit
addChar; getChar
int
return Int_Lit
Digit
addChar; getChar
unknown
t←lookup (nextChar)
getChar
Done
return t