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**

Ahmed Abdelaziz

200043891

**Under Supervision**

:Nihal

:Fares

1. **Introduction**

lexical analyzer that converts code into tokens, handling identifiers, numbers, and operators. It forms the essential first step in compiler design.

* 1. **Phases of Compiler**

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

Phase Description

Lexical Analysis Converts source code into tokens.

Syntax Analysis Checks grammatical correctness and builds a parse tree.

Semantic Analysis Ensures valid meaning and detects type errors.

Intermediate Code Generation Produces an intermediate representation.

Code Optimization Improves performance by reducing redundancies.

Code Generation Converts optimized code to machine code.

Error Handling Identifies and reports errors.

1. **Lexical Analyzer**

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

**Software Tools**

diverse set of software tools is used in the process of creating compilers..

* 1. **Computer Program**

A compiler is a program that translates source code into machine code. It

Make sure that syntax and semantics are true.

* 1. **Programming Language**

javascript.

1. **Implementation of a Lexical Analyzer**

const INT\_LIT = 10;

1. const IDENT = 11;
2. const ASSIGN\_OP = 20;
3. const ADD\_OP = 21;
4. const SUB\_OP = 22;
5. const MULT\_OP = 23;
6. const DIV\_OP = 24;
7. const LEFT\_PAREN = 25;
8. const RIGHT\_PAREN = 26;
9. const UNKNOWN = 99;
10. const EOF = -1;
11. const LETTER = 0;
12. const DIGIT = 1;
13. const UNKNOWN\_CHAR = 99;
14. let charClass;
15. let lexeme = [];
16. let nextChar;
17. let lexLen = 0;
18. let token;
19. let nextToken;
20. let inFp;
21. const tokenNameMap = new Map([
22. [INT\_LIT, "INT\_LIT"],
23. [IDENT, "IDENT"],
24. [ASSIGN\_OP, "ASSIGN\_OP"],
25. [ADD\_OP, "ADD\_OP"],
26. [SUB\_OP, "SUB\_OP"],
27. [MULT\_OP, "MULT\_OP"],
28. [DIV\_OP, "DIV\_OP"],
29. [LEFT\_PAREN, "LEFT\_PAREN"],
30. [RIGHT\_PAREN, "RIGHT\_PAREN"],
31. [UNKNOWN, "UNKNOWN"],
32. [EOF, "EOF"]
33. ]);
34. function getTokenName(code) {
35. return tokenNameMap.get(code) || code.toString();
36. }
37. function isAlpha(c) {
38. return /^[a-zA-Z]$/.test(c);
39. }
40. function isDigit(c) {
41. return /^[0-9]$/.test(c);
42. }
43. function isSpace(c) {
44. return /\s/.test(c);
45. }
46. function processChar() {
47. if (lexLen <= 98) {
48. lexeme[lexLen++] = nextChar;
49. lexeme[lexLen] = '\0';
50. } else {
51. console.error("Error - lexeme is too long");
52. }
53. }
54. function fetchChar() {
55. if (inFp && inFp.position < inFp.data.length) {
56. nextChar = inFp.data[inFp.position++];
57. if (isAlpha(nextChar)) {
58. charClass = LETTER;
59. } else if (isDigit(nextChar)) {
60. charClass = DIGIT;
61. } else {
62. charClass = UNKNOWN\_CHAR;
63. }
64. } else {
65. charClass = EOF;
66. nextChar = 'EOF';
67. }
68. }
69. function skipWhitespace() {
70. while (isSpace(nextChar)) {
71. fetchChar();
72. }
73. }
74. function identifySymbol(ch) {
75. processChar();
76. switch (ch) {
77. case '(':
78. nextToken = LEFT\_PAREN;
79. break;
80. case ')':
81. nextToken = RIGHT\_PAREN;
82. break;
83. case '+':
84. nextToken = ADD\_OP;
85. break;
86. case '-':
87. nextToken = SUB\_OP;
88. break;
89. case '\*':
90. nextToken = MULT\_OP;
91. break;
92. case '/':
93. nextToken = DIV\_OP;
94. break;
95. case '=':
96. nextToken = ASSIGN\_OP;
97. break;
98. default:
99. nextToken = UNKNOWN;
100. break;
101. }
102. return nextToken;
103. }
104. function analyzeLexeme() {
105. lexLen = 0;
106. skipWhitespace();
107. switch (charClass) {
108. case LETTER:
109. processChar();
110. fetchChar();
111. while (charClass === LETTER || charClass === DIGIT) {
112. processChar();
113. fetchChar();
114. }
115. nextToken = IDENT;
116. break;
117. case DIGIT:
118. processChar();
119. fetchChar();
120. while (charClass === DIGIT) {
121. processChar();
122. fetchChar();
123. }
124. nextToken = INT\_LIT;
125. break;
126. case UNKNOWN\_CHAR:
127. identifySymbol(nextChar);
128. fetchChar();
129. break;
130. case EOF:
131. nextToken = EOF;
132. lexeme[0] = 'E';
133. lexeme[1] = 'O';
134. lexeme[2] = 'F';
135. lexeme[3] = '\0';
136. lexLen = 3;
137. break;
138. }
139. const lexText = lexeme.slice(0, lexLen).join('').replace('\0', '');
140. console.log(`Next token is: ${nextToken} (${getTokenName(nextToken)}), Next lexeme is ${lexText}`);
141. return nextToken;
142. }
143. function executeAnalysis(inputText) {
144. lexeme = Array(100).fill('');
145. lexLen = 0;
146. nextChar = '';
147. inFp = { data: inputText, position: 0 };
148. fetchChar();
149. do {
150. nextToken = analyzeLexeme();
151. } while (nextToken !== EOF);
152. }
153. function runTestCases() {
154. console.log("LEXICAL ANALYZER TEST");
155. console.log("=====================");
157. console.log("\nTesting expression: \"(sum + 47) / total\"");
158. executeAnalysis("(sum + 47) / total");
160. console.log("\nTesting expression: \"x = y \* (z - 5)\"");
161. executeAnalysis("x = y \* (z - 5)");
163. console.log("\nAnalysis complete");
164. }
165. runTestCases();

**code description: -**

the tokens

const INT\_LIT = 10; // Integer literals

const IDENT = 11; // Identifiers (variables)

const ASSIGN\_OP = 20; // Assignment operator '='

const ADD\_OP = 21; // Addition operator '+'

const SUB\_OP = 22; // Subtraction operator '-'

const MULT\_OP = 23; // Multiplication operator '\*'

const DIV\_OP = 24; // Division operator '/'

const LEFT\_PAREN = 25; // Left parenthesis '('

const RIGHT\_PAREN = 26; // Right parenthesis ')'

const UNKNOWN = 99; // Unknown tokens

const EOF = -1; // End of file marker

## Character

const LETTER = 0; // Alphabetic characters

const DIGIT = 1; // Numeric digits

const UNKNOWN\_CHAR = 99; // Unrecognized characters

## Global Variables

* charClass: Current character classification
* lexeme: Array storing current lexeme
* nextChar: Next character to process
* lexLen: Current lexeme length
* token: Current token
* nextToken: Next token to process
* inFp: Input data pointer

## Functions

### getTokenName(code)

Returns the string name for a token code.

**Parameters:**

* code: Token code to lookup

**Returns:**

* String representation of token name

### Character Classification Functions

isAlpha(c) // Returns true if character is alphabetic

isDigit(c) // Returns true if character is numeric

isSpace(c) // Returns true for whitespace characters

### Lexer Operations

processChar() // Adds current char to lexeme

fetchChar() // Gets next char and classifies it

skipWhitespace() // Advances past whitespace

identifySymbol(ch) // Determines token type for symbol

### Main Analysis Function

analyzeLexeme() // Identifies and returns next token

### Execution Functions

executeAnalysis(inputText) // Runs lexer on input string

runTestCases() // Executes predefined test cases

## Usage Example

runTestCases(); // Executes analysis on test expressions

## Output

For input "(sum + 47) / total":

Next token is: 25 (LEFT\_PAREN), Next lexeme is (

Next token is: 11 (IDENT), Next lexeme is sum

Next token is: 21 (ADD\_OP), Next lexeme is +

Next token is: 10 (INT\_LIT), Next lexeme is 47

Next token is: 26 (RIGHT\_PAREN), Next lexeme is )

Next token is: 24 (DIV\_OP), Next lexeme is /

Next token is: 11 (IDENT), Next lexeme is total

**References:**

1.

ONCEPTS OF PROGRAMMING

LANGUAGES

TWELFTH EDITI

2

MDN Web Docs

3

ECMAScript 2022 Spec

.