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**

Student Name :

Belal Salah

200047818

**Under Supervision**

DR/Nihal A.

Eng /Fares.

1. **Introduction**

This document provides an overview of the implementation of a Lexical Analyzer, which is a

fundamental phase in compiler design. It covers the phases of a compiler, the role of a lexical

analyzer, software tools used, and the implementation details.

* 1. **Phases of Compiler**

A compiler consists of several phases, including:

1. **Lexical Analysis**: Tokenizing the input code.

2. **Syntax Analysis**: Checking grammatical structure.

3. **Semantic Analysis**: Ensuring meaningful statements.

4. **Intermediate Code Generation**: Creating an intermediate representation.

5. **Optimization**: Improving performance and efficiency.

6. **Code Generation**: Producing machine code.

1. **Lexical Analyzer**

A **Lexical Analyzer** is responsible for scanning the source code and converting it into tokens.

It identifies keywords, operators, identifiers, and other elements

1. **Software Tools**

Various software tools are used in compiler construction.

* 1. **Computer Program**

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

ensures the correctness of syntax and semantics.

* 1. **Programming Language**

Lexical analyzers are often implemented using programming languages like Python, C,

or Java. The implementation in this document is in javascript.

1. **Implementation of a Lexical Analyzer**

Below is the Javascript implementation of a lexical analyzer:

1. const INT\_LIT = 10;
2. const IDENT = 11;
3. const ASSIGN\_OP = 20;
4. const ADD\_OP = 21;
5. const SUB\_OP = 22;
6. const MULT\_OP = 23;
7. const DIV\_OP = 24;
8. const LEFT\_PAREN = 25;
9. const RIGHT\_PAREN = 26;
10. const UNKNOWN = 99;
11. const EOF = -1;
12. const LETTER = 0;
13. const DIGIT = 1;
14. const UNKNOWN\_CHAR = 99;
15. let charClass;
16. let lexeme = Array(100).fill('');
17. let nextChar = '';
18. let lexLen = 0;
19. let token;
20. let nextToken;
21. let inFp = null;
22. function getTokenName(tokenCode) {
23. const tokenNames = {
24. 10: "INT\_LIT",
25. 11: "IDENT",
26. 20: "ASSIGN\_OP",
27. 21: "ADD\_OP",
28. 22: "SUB\_OP",
29. 23: "MULT\_OP",
30. 24: "DIV\_OP",
31. 25: "LEFT\_PAREN",
32. 26: "RIGHT\_PAREN",
33. 99: "UNKNOWN",
34. [-1]: "EOF"
35. };
36. return tokenNames[tokenCode] || tokenCode.toString();
37. }
38. function lookup(ch) {
39. switch (ch) {
40. case '(':
41. addChar();
42. nextToken = LEFT\_PAREN;
43. break;
44. case ')':
45. addChar();
46. nextToken = RIGHT\_PAREN;
47. break;
48. case '+':
49. addChar();
50. nextToken = ADD\_OP;
51. break;
52. case '-':
53. addChar();
54. nextToken = SUB\_OP;
55. break;
56. case '\*':
57. addChar();
58. nextToken = MULT\_OP;
59. break;
60. case '/':
61. addChar();
62. nextToken = DIV\_OP;
63. break;
64. case '=':
65. addChar();
66. nextToken = ASSIGN\_OP;
67. break;
68. default:
69. addChar();
70. nextToken = UNKNOWN;
71. break;
72. }
73. return nextToken;
74. }
75. function addChar() {
76. if (lexLen <= 98) {
77. lexeme[lexLen++] = nextChar;
78. lexeme[lexLen] = '\0';
79. } else {
80. console.error("Error - lexeme is too long");
81. }
82. }
83. function getChar() {
84. if (inFp && inFp.position < inFp.data.length) {
85. nextChar = inFp.data[inFp.position++];
86. if (isAlpha(nextChar)) {
87. charClass = LETTER;
88. } else if (isDigit(nextChar)) {
89. charClass = DIGIT;
90. } else {
91. charClass = UNKNOWN\_CHAR;
92. }
93. } else {
94. charClass = EOF;
95. nextChar = 'EOF';
96. }
97. }
98. function getNonBlank() {
99. while (isSpace(nextChar)) {
100. getChar();
101. }
102. }
103. function lex() {
104. lexLen = 0;
105. getNonBlank();
106. switch (charClass) {
107. case LETTER:
108. addChar();
109. getChar();
110. while (charClass === LETTER || charClass === DIGIT) {
111. addChar();
112. getChar();
113. }
114. nextToken = IDENT;
115. break;
116. case DIGIT:
117. addChar();
118. getChar();
119. while (charClass === DIGIT) {
120. addChar();
121. getChar();
122. }
123. nextToken = INT\_LIT;
124. break;
125. case UNKNOWN\_CHAR:
126. lookup(nextChar);
127. getChar();
128. break;
129. case EOF:
130. nextToken = EOF;
131. lexeme[0] = 'E';
132. lexeme[1] = 'O';
133. lexeme[2] = 'F';
134. lexeme[3] = '\0';
135. lexLen = 3;
136. break;
137. }
138. const lexemeStr = lexeme.slice(0, lexLen).join('').replace('\0', '');
139. console.log(`Next token is: ${nextToken} (${getTokenName(nextToken)}), Next lexeme is ${lexemeStr}`);
140. return nextToken;
141. }
142. function isAlpha(c) {
143. return typeof c === 'string' && /^[a-zA-Z]$/.test(c);
144. }
145. function isDigit(c) {
146. return typeof c === 'string' && /^[0-9]$/.test(c);
147. }
148. function isSpace(c) {
149. return typeof c === 'string' && /\s/.test(c);
150. }
151. function main(inputText) {
152. console.log(`Analyzing: "${inputText}"\n`);
153. lexeme = Array(100).fill('');
154. lexLen = 0;
155. nextChar = '';
156. inFp = { data: inputText, position: 0 };
157. getChar();
158. do {
159. nextToken = lex();
160. } while (nextToken !== EOF);
161. console.log("\nAnalysis complete\n");
162. }
163. function testWithExpression(expression) {
164. console.log(`Testing expression: "${expression}"\n`);
165. lexeme = Array(100).fill('');
166. lexLen = 0;
167. nextChar = '';
168. inFp = { data: expression, position: 0 };
169. getChar();
170. let results = [];
171. do {
172. nextToken = lex();
173. results.push({
174. token: nextToken,
175. tokenName: getTokenName(nextToken),
176. lexeme: lexeme.slice(0, lexLen).join('').replace('\0', '')
177. });
178. } while (nextToken !== EOF);
179. console.log("\nSummary of tokens:");
180. results.forEach((result, index) => {
181. if (result.token !== EOF) {
182. console.log(`${index + 1}. Token: ${result.tokenName} (${result.token}), Lexeme: ${result.lexeme}`);
183. }
184. });
185. console.log("\nEnd of analysis\n");
186. }
187. console.log("LEXICAL ANALYZER TEST");
188. console.log("=====================");
189. testWithExpression("(sum + 47) / total");
190. testWithExpression("x = y \* (z - 5)");

**code description: -**

The following constants represent the token codes used by the lexer:

const INT\_LIT = 10; // Token code for integer literals

const IDENT = 11; // Token code for identifiers

const ASSIGN\_OP = 20; // Token code for assignment operator '='

const ADD\_OP = 21; // Token code for addition operator '+'

const SUB\_OP = 22; // Token code for subtraction operator '-'

const MULT\_OP = 23; // Token code for multiplication operator '\*'

const DIV\_OP = 24; // Token code for division operator '/'

const LEFT\_PAREN = 25; // Token code for left parenthesis '('

const RIGHT\_PAREN = 26; // Token code for right parenthesis ')'

const UNKNOWN = 99; // Token code for unknown characters

const EOF = -1; // Token code for end of file

## Character Classes

The lexer classifies characters into the following categories:

const LETTER = 0; // Character class for letters

const DIGIT = 1; // Character class for digits

const UNKNOWN\_CHAR = 99; // Character class for unknown characters

## Global Variables

* charClass: Current character class.
* lexeme: Array to hold the current lexeme.
* nextChar: Next character to be processed.
* lexLen: Length of the current lexeme.
* token: Current token.
* nextToken: Next token to be processed.
* inFp: Input file pointer.

## Functions

### getTokenName(tokenCode)

Retrieves the name of the token based on its code.

**Parameters:**

* tokenCode: The code of the token.

**Returns:**

* A string representing the name of the token.

### lookup(ch)

Identifies the token type based on the character provided.

**Parameters:**

* ch: The character to be analyzed.

**Returns:**

* The token code of the identified token.

### addChar()

Adds the current character to the lexeme.

**Returns:**

* None. It modifies the lexeme array and updates lexLen.

### getChar()

Reads the next character from the input and determines its class.

**Returns:**

* None. It updates nextChar and charClass.

### getNonBlank()

Skips over any whitespace characters in the input.

**Returns:**

* None. It modifies the state of the lexer.

### lex()

Performs lexical analysis to identify the next token.

**Returns:**

* The token code of the next token.

### isAlpha(c)

Checks if the character is an alphabetic letter.

**Parameters:**

* c: The character to check.

**Returns:**

* true if c is a letter, otherwise false.

### isDigit(c)

Checks if the character is a digit.

**Parameters:**

* c: The character to check.

**Returns:**

* true if c is a digit, otherwise false.

### isSpace(c)

Checks if the character is a whitespace character.

**Parameters:**

* c: The character to check.

**Returns:**

* true if c is a whitespace character, otherwise false.

### main(inputText)

Main function to analyze the input text.

**Parameters:**

* inputText: The text to be analyzed.

**Returns:**

* None. It processes the input and outputs the results.

### testWithExpression(expression)

Tests the lexer with a specific expression and outputs the results.

**Parameters:**

* expression: The expression to be tested.

**Returns:**

* None. It outputs a summary of the tokens identified.

## Usage

To run the lexer, call the testWithExpression function with a string expression. For example:

testWithExpression("(sum + 47) / total");

This will analyze the expression and print the identified tokens along with their lexemes.

## Example Output

When testing the expression (sum + 47) / total, the output will include the identified tokens and their corresponding lexemes:

1. Token: IDENT (11), Lexeme: sum

2. Token: ADD\_OP (21), Lexeme: +

3. Token: INT\_LIT (10), Lexeme: 47

4. Token: DIV\_OP (24), Lexeme: /

5. Token: IDENT (11), Lexeme: total

**5. References:**

1.Parr, T. (2022). *Language Implementation Patterns: Create Your Own Domain-Specific and*

*General Programming Languages with Python*.

2.Parsons, D. (2021). *Introduction to Compiler Design*.