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**

Malak Hanafy Mahmoud

200044093

**Under Supervision**

DR: Nehal A.Mohammed

Eng: Fares emedeldin Youssef

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

### 

### 1. **Lexical Analysis**

* **Input:** Source code
* **Output:** Tokens
* This is the **first phase** of the compiler.
* It scans the source code character by character and groups sequences of characters (lexemes) into **tokens**.
* It removes whitespaces and comments.

### 2. **Syntax Analysis (Parsing)**

* **Input:** Stream of tokens
* **Output:** Parse tree or syntax tree
* Checks whether the tokens form a valid syntactic structure according to grammar rules.
* Detects syntax errors and organizes tokens into a tree structure that shows hierarchy and order.

### 3. **Semantic Analysis**

* **Input:** Syntax tree
* **Output:** Annotated syntax tree
* Ensures the program is **semantically correct**.
* Checks for type compatibility, undeclared variables, function call correctness, etc.
* Builds and updates the **symbol table** (a database of variable names, types, scopes).

### 4. **Intermediate Code Generation**

* **Input:** Annotated syntax tree
* **Output:** Intermediate code (e.g., 3-address code)
* Converts the high-level code into a simplified, lower-level intermediate representation.
* It's easier to optimize and translate to machine code.

### 5. **Code Optimization**

* **Input:** Intermediate code
* **Output:** Optimized intermediate code
* Improves performance of the code (speed or memory usage).
* Removes unnecessary code, simplifies expressions, and reuses computations.

### 6. **Code Generation**

* **Input:** Optimized intermediate code
* **Output:** Machine code (assembly or binary)
* Translates the intermediate code to target **machine-level instructions**.
* Handles register allocation and memory addressing.

### 7. **Symbol Table**

* A global structure used throughout compilation.
* Stores information about **identifiers** (name, type, scope, memory location).
* Referenced during semantic analysis, code generation, and error checking.

### 8. **Error Handling**

* Occurs at all phases.
* Detects and reports errors such as:
  + **Lexical errors:** invalid characters
  + **Syntax errors:** missing semicolon
  + **Semantic errors:** type mismatches
* It helps provide informative messages and recover from minor errors to continue compiling.

1. **Lexical Analyzer**

A lexical analyzer serves as the front end of a syntax analyzer.

A lexical analyzer performs syntax analysis at the lowest level of program structure.

An input program appears to a compiler as a single string of characters.

The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure.

these logical groupings are named lexemes, and the internal codes for categories of these groupings are named tokens.

Lexemes: are recognized by matching the input character string against character string patterns.

Tokens: are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants.

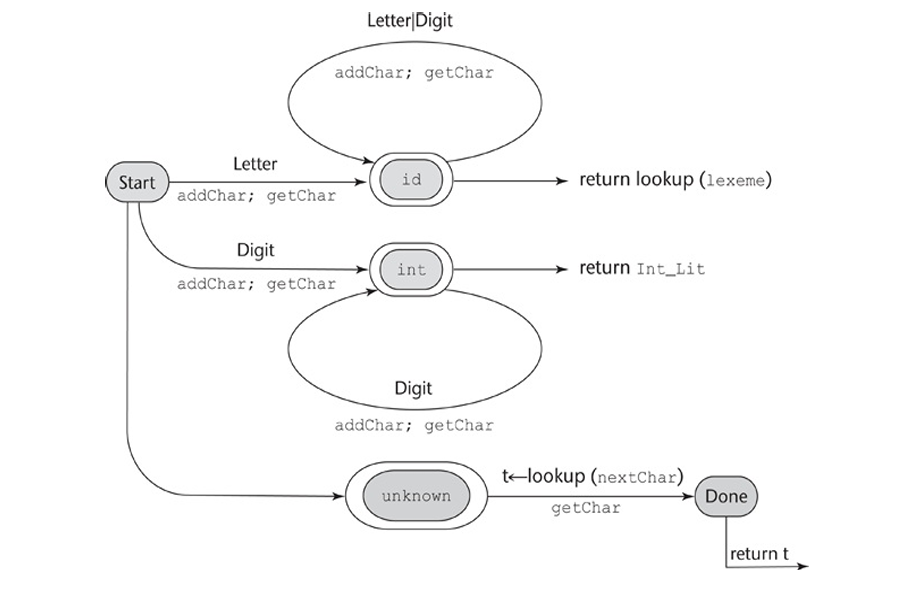
Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens.

In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes.

Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer.

So, each call to the lexical analyzer returns a single lexeme and its token.

The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

****

**The statement:**

M=(Z+F)/(10\*S);

|  |  |  |
| --- | --- | --- |
| **lexeme** | **Token code** | **Token type** |
| **M** | **11** | **ID** |
| **=** | **20** | **ASSIGN\_OP** |
| **(** | **25** | **LEFT\_PAREN** |
| **Z** | **11** | **ID** |
| **+** | **21** | **ADD\_OP** |
| **F** | **11** | **ID** |
| **)** | **26** | **RIGHT\_PAREN** |
| **/** | **24** | **DIV\_OP** |
| **(** | **25** | **LEFT\_PAREN** |
| **10** | **10** | **INT\_LIT** |
| **\*** | **23** | **MULT\_OP** |
| **S** | **11** | **ID** |
| **)** | **26** | **RIGHT\_PAREN** |
| **;** | **-1** | **SEMICOLON** |

* **Identifiers** like M, Z, F, and S are assigned the token code 11 and type ID, indicating they are variable names.
* The **assignment operator** = is assigned token code 20 and labeled as ASSIGN\_OP.
* **Parentheses** ( and ) are assigned codes 25 and 26 respectively, representing LEFT\_PAREN and RIGHT\_PAREN.
* **Arithmetic operators** +, /, and \* are recognized as ADD\_OP (21), DIV\_OP (24), and MULT\_OP (23) respectively.
* The **number** 10 is recognized as an integer literal with token code 10 and type INT\_LIT.
* The **semicolon** ; indicates the end of the statement and is given token code -1 and type SEMICOLON.
* **Global Variables**

These hold the current state of parsing:

* + inputStr: the hardcoded input string
  + lexeme: stores the currently parsed token
  + charClass: indicates if a character is a letter, digit, etc.
  + nextToken: stores the identified token code
* **Functions**
  + getChar(): Reads the next character and determines its class
  + addChar(): Adds the current character to the lexeme
  + lookup(): Maps a symbol to its corresponding token code
  + getNonBlank(): Skips whitespace
  + lex(): Main lexical analyzer logic
* **Main Execution**

The main() function initializes the scan and loops through all characters until the end of input is reached.

1. **Software Tools**
   1. **Computer Program**

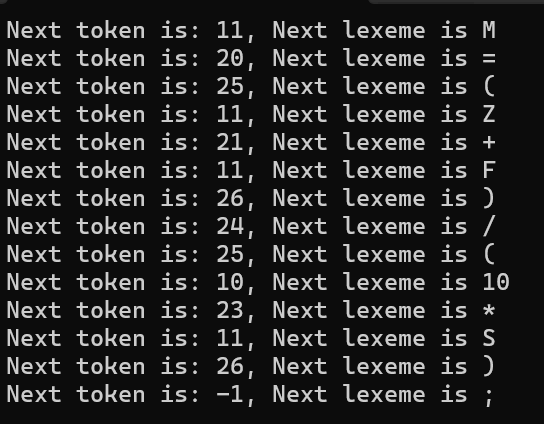
Visual studio 2022

* 1. **Programming Language**

C++

1. **Implementation of a Lexical Analyzer**
2. #include <iostream>
3. #include <cctype>
4. #include <string>
5. using namespace std;
6. #define LETTER 0
7. #define DIGIT 1
8. #define UNKNOWN 99
9. #define END\_OF\_FILE -1
10. #define INT\_LIT 10
11. #define IDENT 11
12. #define ASSIGN\_OP 20
13. #define ADD\_OP 21
14. #define SUB\_OP 22
15. #define MULT\_OP 23
16. #define DIV\_OP 24
17. #define LEFT\_PAREN 25
18. #define RIGHT\_PAREN 26
19. string lexeme;
20. char nextChar;
21. int charClass;
22. int nextToken;
23. int inputIndex = 0;
24. string inputStr = "M=(Z+F)/(10\*S);";
25. void addChar();
26. void getChar();
27. void getNonBlank();
28. int lookup(char ch);
29. int lex();
30. void addChar() {
31. lexeme += nextChar;
32. }
33. void getChar() {
34. if (inputIndex < inputStr.length()) {
35. nextChar = inputStr[inputIndex++];
36. if (isalpha(nextChar))
37. charClass = LETTER;
38. else if (isdigit(nextChar))
39. charClass = DIGIT;
40. else
41. charClass = UNKNOWN;
42. }
43. else {
44. charClass = END\_OF\_FILE;
45. }
46. }
47. void getNonBlank() {
48. while (isspace(nextChar))
49. getChar();
50. }
51. int lookup(char ch) {
52. switch (ch) {
53. case '(':
54. addChar(); nextToken = LEFT\_PAREN; break;
55. case ')':
56. addChar(); nextToken = RIGHT\_PAREN; break;
57. case '+':
58. addChar(); nextToken = ADD\_OP; break;
59. case '-':
60. addChar(); nextToken = SUB\_OP; break;
61. case '\*':
62. addChar(); nextToken = MULT\_OP; break;
63. case '/':
64. addChar(); nextToken = DIV\_OP; break;
65. case '=':
66. addChar(); nextToken = ASSIGN\_OP; break;
67. default:
68. addChar(); nextToken = END\_OF\_FILE; break;
69. }
70. return nextToken;
71. }
72. int lex() {
73. lexeme = "";
74. getNonBlank();
75. switch (charClass) {
76. case LETTER:
77. addChar();
78. getChar();
79. while (charClass == LETTER || charClass == DIGIT) {
80. addChar();
81. getChar();
82. }
83. nextToken = IDENT;
84. break;
85. case DIGIT:
86. addChar();
87. getChar();
88. while (charClass == DIGIT) {
89. addChar();
90. getChar();
91. }
92. nextToken = INT\_LIT;
93. break;
94. case UNKNOWN:
95. lookup(nextChar);
96. getChar();
97. break;
98. case END\_OF\_FILE:
99. nextToken = END\_OF\_FILE;
100. lexeme = "EOF";
101. break;
102. }
103. cout << "Next token is: " << nextToken << ", Next lexeme is " << lexeme << endl;
104. return nextToken;
105. }
106. int main() {
107. getChar();
108. while (nextToken != END\_OF\_FILE) {
109. lex();
110. }
111. return 0;
112. }

**OUTPUT:**

****

1. **References**

CONCEPTS OF PROGRAMMING LANGUAGES TWELFTH EDITION ROBERT W. SEBESTA University of Colorado at Colorado Spring