MISR UNIVERSITY
FOR SCIENCE & TECHNOLOGY
College of Information Technology

جامعة مصر
للعلوم والتكنولوجيا
كلية تكنولوجيا المعلومات

LEXICAL ANALYZER

Build Scanner

Prepared By
Student Name Student ID
Youssef Ahmed Fawzy
200042728

Under Supervision
Name of Doctor: dr nehal
Name of T. A.: eng islam said

1. Introduction

Introduction to Lexical Analyzer

2.The **lexical analyzer** (also known as the scanner) is the first phase in the compilation process. Its primary function is to read the source code one character at a time and transform it into a sequence of tokens. Tokens are the smallest meaningful units of a program, such as keywords, identifiers, constants, operators, and punctuation.

3.By converting raw source code into a structured sequence of tokens, the lexical analyzer makes the job easier for the following compiler stages. It also eliminates unnecessary elements like whitespace and comments, which aren't relevant for syntax or semantic analysis.

4.The lexical analyzer is essential for error detection, ensuring that all characters and symbols are valid and correctly categorized. The tokens produced by this phase are then passed to the syntax analyzer, which uses them to construct the program's syntactic structure

1.1. Phases of Compiler

This image illustrates the **phases of a compiler**, breaking down the process of translating **source code** into **target code** (usually machine code or bytecode). It's divided into two main parts: **Analysis** and **Synthesis**.

🔍 Analysis Phase

This part analyzes the source code and checks for correctness.

1. **Lexical Analyzer**
   - **Input:** Source code
   - **Output:** Stream of tokens
   - **Job:** Breaks the code into tokens (keywords, identifiers, symbols, etc.)
2. **Syntax Analyzer** (Parser)
   - **Input:** Stream of tokens
   - **Output:** Abstract Syntax Tree (AST)
   - **Job:** Checks syntax rules (grammar of the language) and builds a tree structure (AST).
3. **Semantic Analyzer**

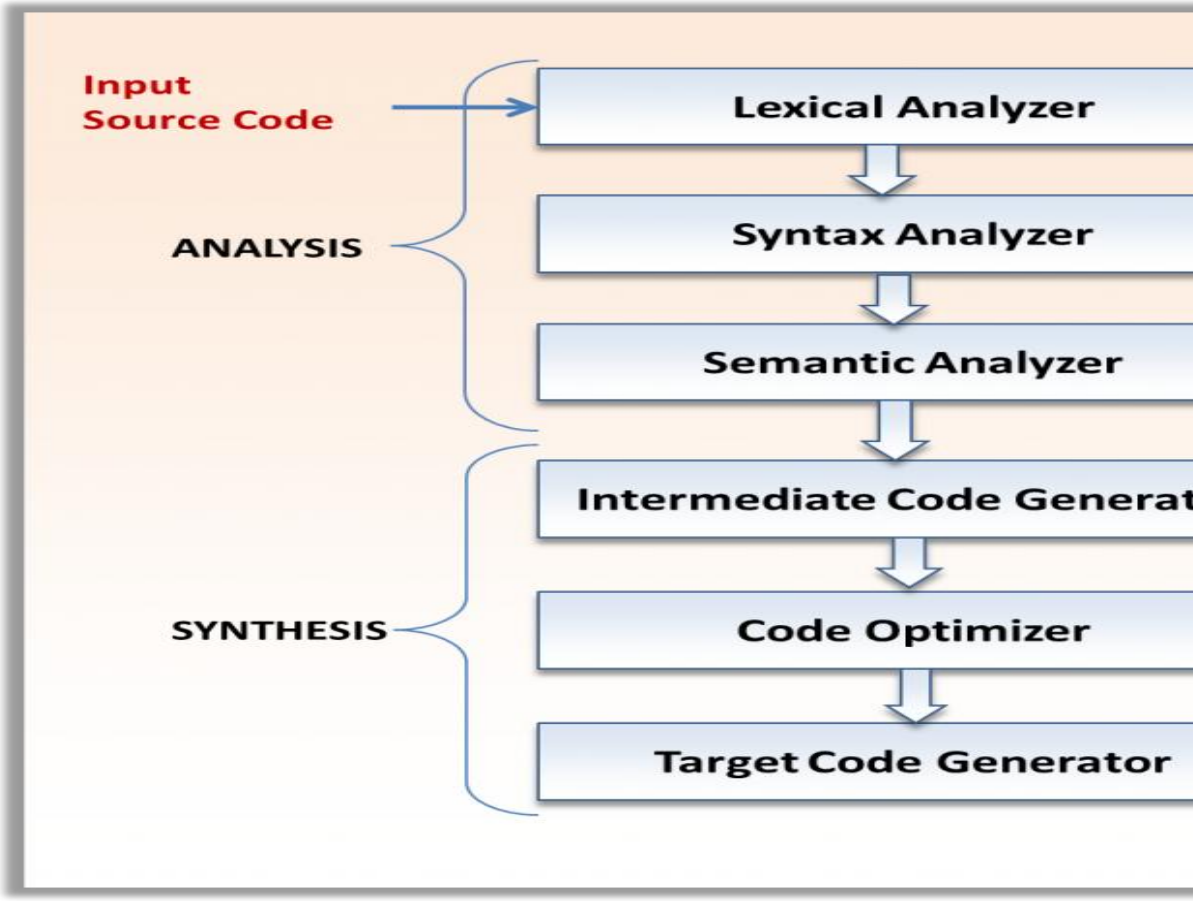
- **Input:** AST
- **Output:** Parse Tree
- **Job:** Ensures logical correctness (like type-checking, variable declarations, etc.)

⚙ Synthesis Phase

This part generates the actual machine-level output.

4. **Intermediate Code Generator**
   - **Input**: Parse Tree
   - **Output**: Intermediate Code
   - **Function**: The intermediate code generator takes the parse tree as input and translates it into an intermediate representation of the program. This intermediate code serves as a bridge between the high-level source code and the machine code, and it is platform-independent, meaning it can be adapted for different target systems.
5. **Code Optimizer**
   - **Input**: Intermediate Code
   - **Output**: Optimized Code
   - **Function**: The code optimizer improves the efficiency of the intermediate code by applying optimization techniques. This includes eliminating redundant instructions, simplifying operations, and rearranging code to enhance performance, resulting in a more efficient version of the code.
6. **Target Code Generator**
   - **Input**: Optimized Code
   - **Output**: Target Code (Executable Code)
   - **Function**: The target code generator takes the optimized intermediate code and converts it into machine-specific code, which can be directly executed by the hardware. This phase produces the final executable program.

- **Job:** Converts to the final code for a specific platform/architecture.



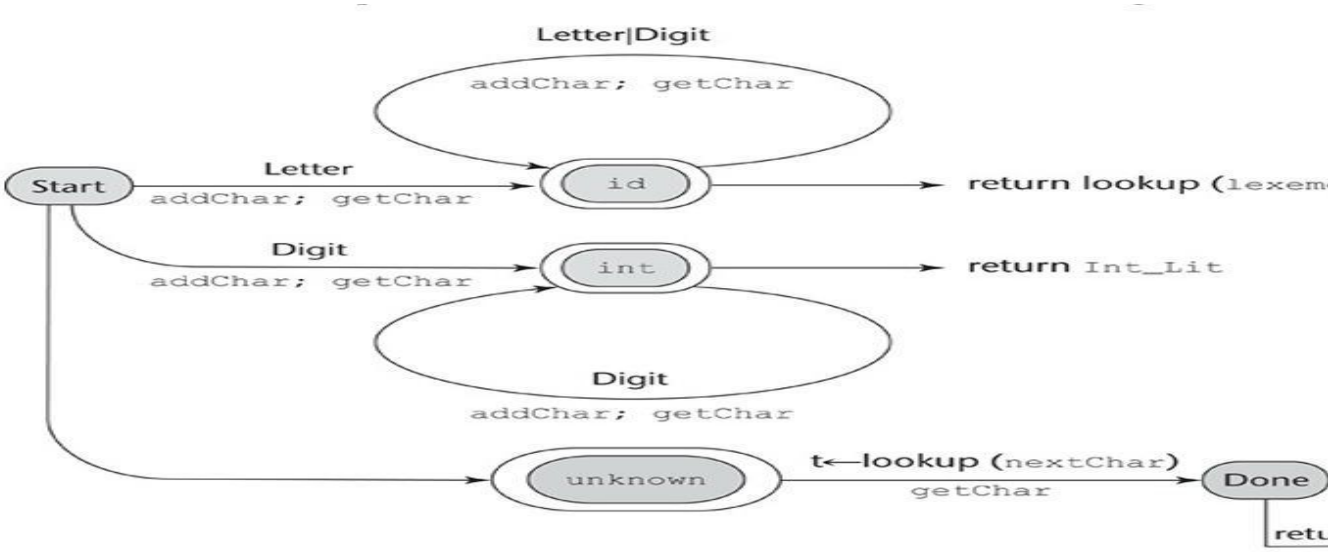
2. Lexical Analyzer

his image shows a **Finite State Machine (FSM)** diagram used by a **Lexical Analyzer** (scanner) to identify **tokens** in a programming language. The FSM describes how the lexical analyzer processes characters from source code and categorizes them into different types of tokens like **identifiers, integers, and unknown symbols**.

Let me break it down step-by-step:

Start State

- This is where scanning begins.
- It checks whether the first character is a **Letter**, **Digit**, or **Other**.



Identifier (id) Recognition

- **If the first character is a Letter**:
  - It calls `addChar` and `getChar` to collect the full word.
  - Then moves to the `id` state.
  - It continues adding characters as long as they are letters or digits.
  - **Returns:** `lookup(lexeme)` – determines if it's a keyword or a user-defined identifier.

Integer (int) Recognition

- **If the first character is a Digit**:
  - It transitions to the `int` state.
  - It loops here, collecting all digits (multi-digit numbers).
  - **Returns:** `Int_Lit` – recognized as an integer literal.

Unknown Symbol Handling

- **If the character is not a letter or digit**, it is considered an **unknown** or **special character**.
  - Goes to the `unknown` state.
  - Uses `lookup(nextChar)` to determine what type of symbol it might be (e.g., +, =, etc.).
  - Transitions to `Done`, and then **returns** the token.

Functions Used

- **addChar:** Adds current character to the token.
- **getChar:** Moves to the next character in the input.
- **lookup:** Determines the token type based on the collected string or character.
- **return:** Yields the final token to the next phase in compilation.

3. Software Tools

3.1. Computer Program

Visual studio code2017

3.2. Programming Language

C++ programming language

4. Implementation of a Lexical Analyzer

include <iostream> // For input/output operations#

include <cctype> // For character checking functions like #

isalpha, isdigit

include <string> // For using the string class#

;using namespace std

```
Character classes //
define LETTER 0#
define DIGIT 1#
define UNKNOWN 99#

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#
define END_OF_FILE -1   // End of file/input#

Global variables //
int charClass;          // Class of the current character
string lexeme;          // Current lexeme being built
char nextChar;          // Current character
int lexLen = 0;         // Length of the lexeme
```

```
int nextToken;          // Token type of the current lexeme

string input;           // Input string to analyze
int inputIndex = 0;     // Current index in the input string

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

} ()int main
input = "x = 5 + y * (10 - 3)"; // The input string to tokenize

getChar(); // Load the first character

Keep analyzing tokens until the end of input //
} while (nextToken != END_OF_FILE)
;()lex
{
```

```
return 0; // Exit program
{

Adds nextChar to lexeme //
} ()void addChar
;lexeme += nextChar
;++lexLen
{

Reads the next character from input and classifies it //
} ()void getChar
} if (inputIndex < input.length())
nextChar = input[inputIndex++]; // Move to next character

Classify the character //
if (isalpha(nextChar))
;charClass = LETTER
else if (isdigit(nextChar))
;charClass = DIGIT
else
;charClass = UNKNOWN
} else {
```

```
charClass = END_OF_FILE; // No more characters left
{
{

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

Determines token type for single-character operators/symbols //
} int lookup(char ch)
} switch (ch)
;case '(': addChar(); return LEFT_PAREN
;case ')': addChar(); return RIGHT_PAREN
;case '+': addChar(); return ADD_OP
;case '-': addChar(); return SUB_OP
;case '*': addChar(); return MULT_OP
;case '/': addChar(); return DIV_OP
;case '=': addChar(); return ASSIGN_OP
;default: addChar(); return UNKNOWN
{
```

```
{

Main lexical analyzer function //
} ()int lex
lexeme = "";     // Reset lexeme
lexLen = 0;      // Reset lexeme length
getNonBlank();   // Skip any spaces

} switch (charClass)
If it starts with a letter, it's an identifier //
:case LETTER
;()addChar
;()getChar
} while (charClass == LETTER || charClass == DIGIT)
;()addChar
;()getChar
{
;nextToken = IDENT
;break

If it starts with a digit, it's an integer literal //
:case DIGIT
```

```
addChar();
getChar();
} while (charClass == DIGIT)
addChar();
getChar();
{
nextToken = INT_LIT;
break
```

```
Next token is: 11, Next lexeme is: x
Next token is: 20, Next lexeme is: =
Next token is: 10, Next lexeme is: 5
Next token is: 21, Next lexeme is: +
Next token is: 11, Next lexeme is: y
Next token is: 23, Next lexeme is: *
Next token is: 25, Next lexeme is: (
Next token is: 10, Next lexeme is: 10
Next token is: 22, Next lexeme is: -
Next token is: 10, Next lexeme is: 3
Next token is: 26, Next lexeme is: )
Next token is: -1, Next lexeme is:
EOF
```

5. References

CONCEPTS OF PROGRAMMING LANGUAGES

TWELFTH EDITION

ROBERT W. SEBESTA

University of Colorado at Colorado Springs

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.