**Chapter 2: Lexical Analyzer**

A **Lexical Analyzer** (also called a scanner or lexer) is the first phase of a compiler, responsible for reading the input source code and breaking it down into **tokens**. A token is a sequence of characters that represent a unit like keywords, operators, identifiers, literals, etc. The lexical analyzer eliminates whitespace, comments, and other non-essential characters from the input stream.

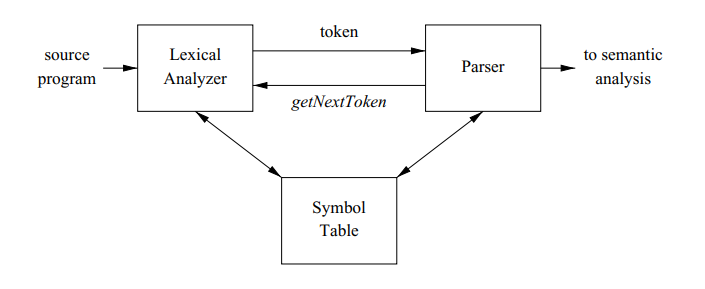


Figure 2.1: Interactions between the lexical analyzer and the parser

**Key Functions of a Lexical Analyzer:**

1. **Tokenization**:

The lexical analyzer reads the raw input source code and breaks it down into tokens, which are atomic units like keywords (if, while), operators (+, -, =), identifiers (variable or function names), and literals (constants, numbers, strings). These tokens are essential for the syntactic analysis in later stages.

1. **Pattern Recognition:**

The lexical analyzer uses regular expressions or similar techniques to define patterns for different token types. It matches lexemes in the input with these patterns to identify them. For example:

Identifiers might match the pattern [a-zA-Z\_][a-zA-Z0-9\_]\*

Integer literals might match the pattern [0-9]+

1. **Error Handling:**

It detects lexical errors, such as illegal characters or malformed tokens (e.g., an identifier starting with a number), and reports them. This ensures that the input to the parser is error-free in terms of basic token structure.

1. **Eliminating Non-Essential Data:**

The lexical analyzer removes whitespace, comments, and other characters that are irrelevant for the syntax analysis phase but necessary for readability and documentation in the source code.

1. **Symbol Table Interaction:**

The lexical analyzer adds new identifiers (variable names, function names, etc.) to the symbol table and associates them with token types, making it easier for the later phases to perform semantic analysis and code generation.

1. **Efficiency:**

By pre-processing the source code and turning it into tokens, the lexical analyzer streamlines the parsing process, making it more efficient. Parsing with raw characters would be far more complex and slower.

1. **Phases in Lexical Analysis:**
2. **Input Buffering**: Efficiently reading characters from the source code.
3. **Lexeme Recognition**: Identifying lexemes (token patterns) using regular expressions or other pattern-matching techniques.
4. **Token Generation**: Assigning a token type to the recognized lexeme (e.g., keyword, identifier).
5. **Error Handling**: Detecting illegal symbols and reporting them.

**Example of Token Generation:**

If a simple C program contains the following:

*int a = 5;*

The lexical analyzer would produce the following tokens:

* int → keyword
* a → identifier
* = → assignment operator
* 5 → numeric constant
* ; → semicolon

**Lexeme, Token and Pattern**

In the context of lexical analysis (the first phase of compilation), **tokens**, **patterns**, and **lexemes** are foundational concepts that describe how a compiler breaks down the source code into manageable pieces for further processing. Here's a detailed breakdown of each term:

**1. Lexeme:**

A **lexeme** is the sequence of characters in the source code that matches a pattern and forms the lowest-level syntactic unit. In simple terms, it is the raw input (a string of characters) that the lexical analyzer identifies as a valid token.

For example, in the statement:

*int x = 10;*

* *int*, *x*, =, *10*, and *;* are all lexemes.

**2. Token:**

A **token** is the classified form of a lexeme. It consists of two parts:

* **Token Type**: A category or class that describes what kind of token it is (e.g., keyword, operator, identifier).
* **Token Value**: The actual lexeme that the token represents.

For example, from the same C code:

* int is a token of type KEYWORD.
* x is a token of type IDENTIFIER.
* 10 is a token of type NUMBER.

In the output from the lexical analyzer, a token would typically be represented like this:

*Token(KEYWORD, "int")*

*Token(IDENTIFIER, "x")*

*Token(OPERATOR, "=")*

*Token(NUMBER, "10")*

*Token(SEMICOLON, ";")*

**Tokens are defined for various constructs in a programming language:**

**I. One token for each keyword:**

Each keyword in the programming language is assigned its own token. Since the pattern for a keyword is the keyword itself, the lexer matches exact strings to recognize these tokens.

Example:

* **Lexeme**: int, while, return
* **Tokens**:Token(KEYWORD\_INT),Token(KEYWORD\_WHILE),Token(KEYWORD\_RETURN)

**II. Tokens for the operators:**

Operators can either be assigned individual tokens or grouped into classes. For instance, comparison operators (==, !=, >, <, etc.) can all be assigned a COMPARISON token, or they can be split into more specific tokens.

Example:

* **Lexemes**: +, -, ==, <, >
* **Tokens**:
  + Token(PLUS) for +
  + Token(MINUS) for -
  + Token(COMPARISON\_EQ) for ==
  + Token(COMPARISON\_LT) for <

**III. One token representing all identifiers:**

Identifiers (variable names, function names, etc.) typically match a pattern like [a-zA-Z\_][a-zA-Z0-9\_]\*, and they all map to a single token type, regardless of their specific name.

Example:

* **Lexemes**: x, sum, myVariable
* **Token**: Token(IDENTIFIER)

**IV. One or more tokens representing constants:**

Constants can be numbers (integers, floating-point numbers) or literal strings. These might be represented by separate tokens, like NUMBER for numeric constants and STRING for string literals.

Example:

* **Lexemes**: 100, 3.14, "hello"
* **Tokens**:
  + Token(NUMBER\_INT) for 100
  + Token(NUMBER\_FLOAT) for 3.14
  + Token(STRING\_LITERAL) for "hello"

**V. Tokens for each punctuation symbol:**

Punctuation symbols (like parentheses, commas, and semicolons) are typically assigned their own individual tokens, which are directly matched.

Example:

* **Lexemes**: (, ), ,, ;
* **Tokens**:
  + Token(LEFT\_PAREN) for (
  + Token(RIGHT\_PAREN) for )
  + Token(COMMA) for ,
  + Token(SEMICOLON) for ;

**3. Pattern:**

A **pattern** describes the rules or structure that defines which lexemes map to which tokens. The lexical analyzer uses these patterns to match and recognize lexemes in the input. Typically, patterns are defined using **regular expressions**.

For example, here are some patterns for different tokens:

* **Keyword**: A specific reserved word like int, while, if (often matched exactly).
* **Identifier**: A pattern for a variable name, which might be something like [a-zA-Z\_][a-zA-Z0-9\_]\* (starting with a letter or underscore and followed by letters, digits, or underscores).
* **Number**: A sequence of digits, like [0-9]+.

The pattern is what tells the lexical analyzer how to identify different types of lexemes and turn them into tokens.

**Summary Example:**

Given this C statement:

*int sum = 100;*

* **Lexemes**: int, sum, =, 100, ;
* **Tokens**:
  + *Token(KEYWORD, "int")*
  + *Token(IDENTIFIER, "sum")*
  + *Token(OPERATOR, "=")*
  + *Token(NUMBER, "100")*
  + *Token(SEMICOLON, ";")*
* **Patterns**:
  + Keyword pattern: specific keywords like int, float, return, etc.
  + Identifier pattern: [a-zA-Z\_][a-zA-Z0-9\_]\*
  + Number pattern: [0-9]+

**Relationship Between Tokens, Lexemes, and Patterns:**

* A **lexeme** is an actual sequence of characters from the source code.
* A **pattern** is a rule describing what sequence of characters qualifies as a lexeme for a specific token type.
* A **token** is the classified output (based on the pattern) that represents the lexeme and its type in the context of the language.

**Example (in context):**

For a mathematical expression like:

a + b = 10;

* **Lexemes**: a, +, b, =, 10, ;
* **Patterns**:
  + [a-zA-Z] for identifiers
  + [+] for the addition operator
  + [=] for the assignment operator
  + [0-9]+ for numbers
* **Tokens**:
  + *Token(IDENTIFIER, "a")*
  + *Token(OPERATOR, "+")*
  + *Token(IDENTIFIER, "b")*
  + *Token(OPERATOR, "=")*
  + *Token(NUMBER, "10")*
  + *Token(SEMICOLON, ";")*

**Attributes for tokens**

Tokens often have associated **attributes** to provide additional information about the lexeme that matches the token. These attributes are crucial for the subsequent phases of compilation, such as parsing and code generation. When multiple lexemes can match the same token type, the attribute helps **distinguish** between them.

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

**Key Concepts of Attributes for Tokens:**

1. **Token Name**:
   * The **token name** is the classification of the lexeme (e.g., IDENTIFIER, NUMBER, KEYWORD). It is used by the parser to influence parsing decisions.
2. **Attribute**:
   * The **attribute** of a token is a value that carries additional information about the lexeme. The attribute is especially important when a single token type can represent many different lexemes, such as IDENTIFIER or NUMBER.

**Examples of Tokens with Attributes:**

1. **Identifiers (IDENTIFIER)**:
   * **Lexeme**: The specific name of the identifier (e.g., sum, x, total).
   * **Attribute**: Typically, a **pointer to the symbol table** entry. The symbol table stores additional information about the identifier, such as its data type (e.g., int, float), scope, and location in the code.
   * **Usage**:
     + **Token Name**: IDENTIFIER
     + **Attribute**: Pointer to the symbol table entry (where detailed information about the identifier is stored).
2. **Literals/Constants (NUMBER, STRING\_LITERAL)**:
   * **Lexeme**: The exact value of the constant or literal (e.g., 100, 3.14, "hello").
   * **Attribute**: The actual **numeric value** or **string**. For numbers, this would be the parsed numeric value. For string literals, it would be the string content.
   * **Usage**:
     + **Token Name**: NUMBER, STRING\_LITERAL
     + **Attribute**: For numbers, the parsed value (100 or 3.14), and for strings, the literal string content ("hello").
3. **Keywords (KEYWORD)**:
   * **Lexeme**: The exact keyword matched (e.g., if, while, return).
   * **Attribute**: Usually, no additional attribute is needed because the keyword itself is unique. The token name is typically enough to identify the keyword.
   * **Usage**:
     + **Token Name**: KEYWORD\_IF, KEYWORD\_WHILE
     + **Attribute**: None (or sometimes the keyword string itself as a lexeme for clarity).
4. **Operators (PLUS, MINUS, ASSIGNMENT)**:
   * **Lexeme**: The operator symbol (e.g., +, -, =).
   * **Attribute**: Usually, operators don’t require attributes because the lexeme itself sufficiently identifies the operation. However, in certain cases, attributes like **precedence** or **associativity** might be included.
   * **Usage**:
     + **Token Name**: PLUS, MINUS, ASSIGNMENT
     + **Attribute**: None (or operator-specific information like precedence).
5. **Punctuation (SEMICOLON, LEFT\_PAREN)**:
   * **Lexeme**: Punctuation characters (e.g., ;, ().
   * **Attribute**: Usually, no attributes are necessary for punctuation because they are self-contained.
   * **Usage**:
     + **Token Name**: SEMICOLON, LEFT\_PAREN
     + **Attribute**: None.

**Example of How Attributes Are Used:**

*int x = 100;*

* **Token Name**: KEYWORD\_INT
  + **Lexeme**: int
  + **Attribute**: None (keywords typically don’t need additional attributes).
* **Token Name**: IDENTIFIER
  + **Lexeme**: x
  + **Attribute**: Pointer to the symbol table entry for x, which contains details like x’s data type (int), scope, and memory location.
* **Token Name**: ASSIGNMENT
  + **Lexeme**: =
  + **Attribute**: None (the operator is self-contained).
* **Token Name**: NUMBER
  + **Lexeme**: 100
  + **Attribute**: The integer value 100, which the code generator will use to generate instructions.
* **Token Name**: SEMICOLON
  + **Lexeme**: ;
  + **Attribute**: None (punctuation does not require additional attributes).

**Use of Symbol Table:**

For identifiers, the token’s attribute often includes a **pointer to the symbol table**. The symbol table holds detailed information, such as:

* The **lexeme** (e.g., x).
* The **data type** (e.g., int).
* The **scope** (local or global).
* The **memory location** (used during code generation).
* Any **additional attributes** like whether the identifier is a function, variable, or constant.

**Alphabet, Language and Regular Expression (Regex)**

**Alphabet** refers to a finite set of symbols used to construct strings or input sequences during the **lexical analysis** phase. These symbols form the basic building blocks for recognizing tokens in source code.

An **alphabet** (denoted by Σ) is the set of **valid characters** that can be used to form tokens in the source language. These characters can include:

* **Letters** (uppercase and lowercase).
* **Digits** (numbers).
* **Punctuation marks** (e.g., ;, ,, .).
* **Special characters** (e.g., +, -, \*, /, =, etc.).
* **Whitespace** (spaces, tabs, newlines).

**Examples of Alphabets in Compiler Design:**

1. **Programming Languages**: Each language has its own alphabet. For example:
   * **C Language Alphabet**: Includes letters (a-z, A-Z), digits (0-9), symbols (+, -, \*, /, ;, etc.), and whitespace.
   * **Pascal Alphabet**: Includes letters (a-z, A-Z), digits, and various symbols like :=, >=, <=, etc.
2. **Lexical Analyzer**: The **lexer** or **scanner** uses this alphabet to recognize tokens by matching patterns (usually described by regular expressions) formed from the characters of the alphabet.

**Importance of Alphabet in Compiler Design:**

* The **alphabet** provides the raw material for constructing **lexemes** (basic units of meaning) in source code.
* Different programming languages define their own **alphabets**, specifying which characters are valid and how they are combined to form tokens.
* Understanding the alphabet helps in defining the **syntax** and **grammar** of the programming language, as well as in building **finite automata** that recognize valid tokens.

**Example:**

In a C program, the alphabet could include:

* **Letters**: a-z, A-Z
* **Digits**: 0-9
* **Operators**: +, -, \*, /
* **Punctuation**: ;, ,, (), {}, []
* **Whitespace**: Spaces, tabs, newlines

With this alphabet, the lexical analyzer can recognize valid identifiers, operators, keywords, and other tokens in the source code.

**Language** refers to a set of strings constructed from an alphabet that follows specific syntactic and semantic rules. Understanding languages in this context is crucial for the development and implementation of compilers and interpreters.

**Regular expressions (regex)** are essential for defining the structure of tokens in the **lexical analysis phase**. The lexical analyzer (lexer or scanner) uses regular expressions to recognize the basic building blocks (tokens) of a programming language, such as keywords, identifiers, operators, and literals.

**Role of Regular Expressions in Compiler Design:**

* Regular expressions are used to define **patterns** for lexical tokens.
* The **lexical analyzer** uses these patterns to match input sequences with token definitions.
* Once a match is found, the lexer produces a token (a terminal symbol) that is passed to the next phase of the compiler (syntax analysis).

**Basic Regular Expression Syntax**:

* + **Literal Characters**: Individual characters from the alphabet, e.g., a, b, 1, +.
  + **Alternation (|)**: Represents a choice between patterns, e.g., a|b matches either a or b.
  + **Concatenation**: Sequences of patterns, e.g., ab matches the string ab.
  + **Kleene Star (\*)**: Represents zero or more repetitions of a pattern, e.g., a\* matches ε (empty string), a, aa, aaa, etc.
  + **Grouping**: Parentheses are used to group expressions, e.g., (a|b)\* matches any combination of a and b (including the empty string).

**Regular Expression Patterns in Compiler Design:**

1. **Identifiers**:
   * Identifiers are sequences of letters and digits, starting with a letter.
   * **Regex**: [a-zA-Z][a-zA-Z0-9]\*
     + This pattern matches strings that start with a letter (uppercase or lowercase), followed by any number of letters or digits.
2. **Keywords**:
   * Keywords are specific reserved words in a programming language (e.g., if, while, return).
   * **Regex for individual keyword**: if|else|while|return
     + Each keyword can be represented as its exact string, separated by the | (OR) operator.
3. **Numbers**:
   * Integers and floating-point numbers.
   * **Integer Regex**: [0-9]+
     + Matches one or more digits.
   * **Floating-point Number Regex**: [0-9]+\.[0-9]+
     + Matches a sequence of digits, a decimal point, and another sequence of digits (e.g., 123.45).
4. **Operators**:
   * Arithmetic operators such as +, -, \*, /.
   * **Regex**: \+|\-|\\*|\/
     + Matches one of the operators (+, -, \*, or /).
5. **Whitespace**:
   * Whitespace (spaces, tabs, newlines) is often ignored by the lexer, but it must be recognized.
   * **Regex**: [\t\n\r ]+
     + Matches any number of whitespace characters (spaces, tabs, newlines).
6. **String Literals**:
   * Strings enclosed in double quotes.
   * **Regex**: "[^"]\*"
     + Matches a sequence of characters between double quotes, excluding the quotes themselves.
7. **Comments**:
   * Comments can be single-line (e.g., // comment) or multi-line (e.g., /\* comment \*/).
   * **Single-line Comment Regex**: \/\/.\*
     + Matches // followed by any characters until the end of the line.
   * **Multi-line Comment Regex**: /\\*[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*/
     + Matches /\*, followed by any sequence of characters, ending with \*/.

* **Regular Language**: A language that can be described by a regular expression. Regular languages can be recognized by finite automata and are typically used for defining tokens in programming languages.

**Example of Regular Expression:**

Let the alphabet Σ={a,b}\Sigma = \{a, b\}Σ={a,b}.

* **Regular Expression**: a\*
  + This matches the language {ϵ,a,aa,aaa,… }\{ \epsilon, a, aa, aaa, \dots \}{ϵ,a,aa,aaa,…}, i.e., any number of occurrences of the character a (including zero occurrences).
* **Regular Expression**: (a|b)\*

This matches the language of all possible strings made from a and b, including the empty string {ϵ,a,b,aa,ab,ba,bb,aaa,… }\{ \epsilon, a, b, aa, ab, ba, bb, aaa, \dots \}{ϵ,a,b,aa,ab,ba,bb,aaa,…}

For a language like **C**, here are a few important tokens defined using regular expressions:

* **Keywords**: if|else|while|for|return
* **Identifiers**: [a-zA-Z\_][a-zA-Z0-9\_]\*
* **Integer constants**: [0-9]+
* **Floating-point constants**: [0-9]+\.[0-9]+
* **Operators**: \+|\-|\\*|\/|==|!=|<=|>=|&&|\|\|
* **Parentheses, braces**: \(|\)|\{|\}

**Summary of Relationships:**

* **Alphabet**: A set of symbols (e.g., letters, digits).
* **Language**: A set of valid strings formed from an alphabet.
* **Regular Expressions**: Patterns used to describe sets of strings (languages) over an alphabet.