**UNIT 2**

**INTRODUCTION TO COMPILER**

**Programming Language Basics**

* Programming languages are defined by **syntax** (rules for valid program structure) and **semantics** (meaning of valid programs).
* **Tokens** are the basic building blocks of a language, such as keywords, identifiers, and operators.
* **Lexical conventions** include rules for handling whitespace, comments, and delimiters, which are ignored during compilation but help in structuring the code.

**What is Lexical Analysis?**

* It reads the input program (a sequence of characters) and groups these into **lexemes**, which are sequences of characters matching patterns (e.g., if, x, +).
* For each lexeme, it generates a **token**, a data structure containing:
  + Token type (e.g., keyword, operator).
  + Attribute value (additional information, like variable names).

**Lexical Analysis – Role of Lexical Analyzer:**

* The **lexical analyzer** (or scanner) is the first phase of a compiler, responsible for converting a stream of characters into a stream of **tokens**.
* It reads the source code character-by-character and groups characters into meaningful units called **tokens**, such as keywords, identifiers, and operators.
* The lexical analyzer **ignores whitespace and comments**, as they are not relevant to the program’s meaning.
* It also **detects lexical errors**, such as invalid characters or unterminated strings, and reports them to the user.
* During tokenization, the lexical analyzer interacts with the **symbol table**, adding identifiers (e.g., variable names) and their attributes for later use.

**Lexical Analysis vs Parsing**

* **Simplicity**: Handling low-level tokenization separately makes the parser simpler and easier to design.
* **Efficiency**: Lexical analyzers use optimized techniques like **finite automata** to recognize tokens quickly.
* **Portability**: Lexical rules (e.g., handling whitespace) are less likely to change across machines compared to syntax rules.

**Tokens, Patterns, and Lexemes**

**Tokens**

* A **token** is a category or type of lexical unit in a programming language.
* Tokens represent meaningful groupings of characters, such as keywords, identifiers, operators, and literals.
* Examples of tokens include INT\_KEYWORD, IDENTIFIER, PLUS\_OPERATOR, and NUMBER.

**Patterns**

* A **pattern** is a rule or description that defines the valid form of lexemes for a specific token.
* Patterns are often expressed using **regular expressions** or **finite automata**.
* Example:
  + The pattern for an IDENTIFIER token might be: *"a sequence of letters and digits starting with a letter."*
  + The pattern for a NUMBER token could be: *"a sequence of digits optionally followed by a decimal point and more digits."*

**Lexemes**

* A **lexeme** is the actual sequence of characters in the source code that matches a pattern and is classified into a token.
* Example:
  + In the code int x = 5;, the lexeme int corresponds to the INT\_KEYWORD token.
  + The lexeme x matches the IDENTIFIER token.
  + The lexeme 5 matches the NUMBER token.

**Relationship**:

* The lexical analyzer uses **patterns** to identify **lexemes** in the source code and categorizes them into **tokens**.

**Lexical Errors**

**Definition**

* **Lexical errors** occur when the scanner encounters sequences of characters that do not match any valid pattern for tokens.

**Common Lexical Errors**

1. **Invalid Characters**:
   * Example: Using @ in a C program, where @ is not a valid operator or identifier.
2. **Unterminated Strings/Comments**:
   * Example: "Hello World (missing closing quote).
   * Example: /\* This is a comment (missing \*/).
3. **Malformed Numbers**:
   * Example: 123.45.67 (multiple decimal points).
4. **Illegal Identifiers**:
   * Example: 5var (starts with a digit).

**Error Recovery Strategies**

1. **Panic Mode**:
   * Skip characters until a valid token is found (e.g., skip until the next whitespace or delimiter).
2. **Delete/Insert Characters**:
   * Fix minor errors automatically (e.g., add a missing quote).
3. **Error Tokens**:
   * Insert a special error token and continue parsing.

**Input Buffering**

Lexical analyzers need to read source code character-by-character, but direct disk access for each character is **inefficient**. Input buffering minimizes disk I/O operations by reading large chunks of the source code into memory.

* **Buffer Pairs**
* **Concept**: Use **two buffers** (Buffer 1 and Buffer 2) to hold chunks of the source code.
* **How it works**:
  1. Load Buffer 1 with the first block of the source code.
  2. Load Buffer 2 with the next block when Buffer 1 is exhausted.
  3. Alternate between buffers to ensure continuous input flow.

**Example:**

Suppose the source code is split into blocks of size **N**:



* The lexical analyzer processes characters from Buffer 1.
* When it reaches the end of Buffer 1, it switches to Buffer 2 and reloads Buffer 1 with the next block.

**Advantages:**

* Reduces disk I/O operations by reading large blocks.
* Ensures seamless processing without waiting for disk reads.
* **Sentinels**
* **Purpose**: Simplify end-of-buffer checks.
* **Concept**: Place a **special marker (sentinel)** at the end of each buffer.
  + The sentinel is a unique character (e.g., EOF or #) that does not appear in the source code.
* **How it works**:
  1. Append a sentinel to the end of each buffer.
  2. The lexical analyzer processes characters until it encounters the sentinel.
  3. When the sentinel is reached, the buffer is reloaded.

**Example:**



* The # marks the end of the buffer.
* When the lexical analyzer reads #, it triggers a buffer switch.

**Advantages:**

* Eliminates repeated checks for buffer boundaries (e.g., if (current\_char == end\_of\_buffer)).
* Speeds up processing by reducing conditional checks.
* **Handling Lookahead**
* **Problem**: Some tokens require looking ahead (e.g., = vs ==).
* **Solution**:
  1. Use a **lookahead pointer** to peek at the next character.
  2. If the next character is in the same buffer, process it.
  3. If it crosses the buffer boundary, switch buffers and continue.

**Example:**

Processing ==:

* Read = from Buffer 1.
* Look ahead to the next character. If it is also =, consume it as part of the token ==.
* **Edge Cases**
* **Token spans across buffers**:
  + If a token starts near the end of a buffer (e.g., 123 split as 12 in Buffer 1 and 3 in Buffer 2):
    - Process 12 in Buffer 1.
    - Switch to Buffer 2 and append 3 to complete the token.

[Recognition of tokens using finite automata](https://www.btechvibes.com/2023/06/recognition-of-tokens-in-compiler-design.html)