# Lexical Analysis:

A lexical analyzer is also called a "**Scanner**". Given the code's statement/ input string, it reads the statement from left to right character-wise. The input to a lexical analyzer is the pure high-level code from the preprocessor. It identifies valid lexemes from the program and returns tokens to the syntax analyzer, one after the other, corresponding to the ***getNextToken*** command from the syntax analyzer.


**There are three important terms to grab:**

1. **Tokens**: A Token is a pre-defined sequence of characters that cannot be broken down further. It is like an abstract symbol that represents a unit. A token can have an optional attribute value. There are different types of tokens:
    - Identifiers (user-defined)
    - Delimiters/ punctuations (;, ,, {}, etc.)
    - Operators (+, -, *, /, etc.)
    - Special symbols
    - Keywords
    - Numbers
2. **Lexemes**: A lexeme is a sequence of characters matched in the source program that matches the pattern of a token.
   **For example**: (, ) are lexemes of type punctuation where punctuation is the token.
3. **Patterns:** A pattern is a set of rules a scanner follows to match a lexeme in the input program to identify a valid token. It is like the lexical analyzer's description

of a token to validate a lexeme.

**For example**, the characters in the keyword are the pattern to identify a keyword. To identify an identifier the pre-defined set of rules to create an identifier is the pattern

| Token | Lexeme | Pattern |
|---|---|---|
| Keyword | while | w-h-i-l-e |
| Relop | < | <, >, >=, <=, !=, == |
| Integer | 7 | (0 - 9)*-> Sequence of digits with at least one digit |
| String | "Hi" | Characters enclosed by " " |
| Punctuation | , | ; , . ! etc. |
| Identifier | number | A - Z, a - z A sequence of characters and numbers initiated by a character. |

## Everything that a lexical analyzer has to do:

1. Stripping out comments and white spaces from the program
2. Read the input program and divide it into valid tokens
3. Find lexical errors
4. Return the Sequence of valid tokens to the syntax analyzer
5. When it finds an identifier, it has to make an entry into the symbol table.

**The questions here are:**

1. How does the lexical analyzer read the input string and break it into lexemes?
2. How can it understand the patterns and check if the lexemes are valid?
3. What does the Lexical Analyzer send to the next phase?

**We'll get into the details question-wise.**

First, the lexical analyzer has to read the input program and break it into tokens. This is achieved by a method called "**Input Buffering**".

## Input Buffering

For suppose, assume that the line of code is:

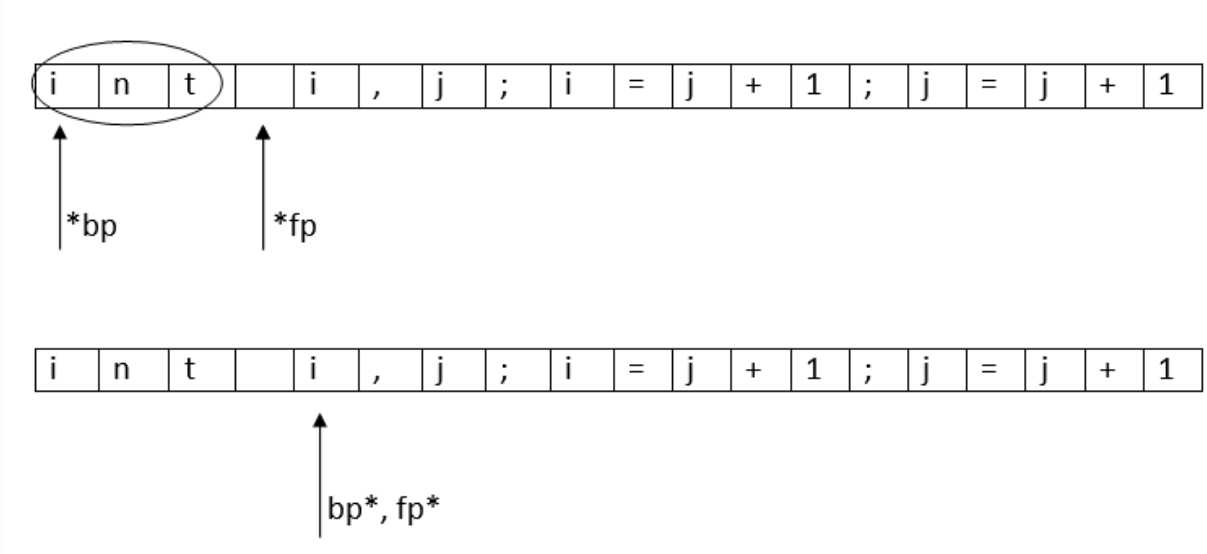
1. int i, j;
2. i = j + 1;
3. jj = j + 1;

The input is stored in buffers to avoid going to secondary memory.

Initially, **We used a One-buffer scheme**:

| i | n | t |  | i | , | j | ; | i | = | j | + | 1 | ; | j | = | j | + | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

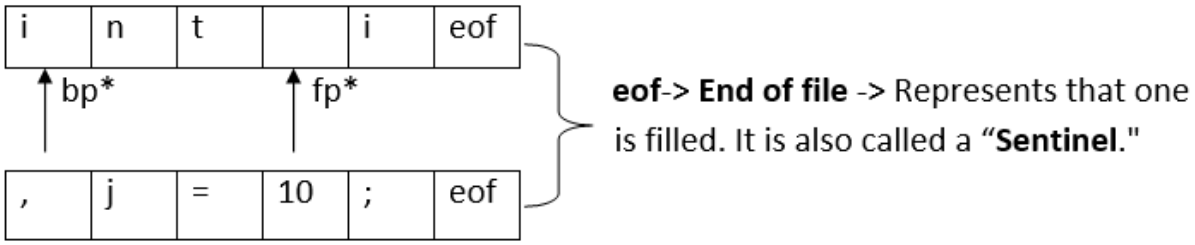
**Two pointers** are used to read and find tokens: ***bp (Beginning) and *fp (foreword)**. *bp is kept at the beginning, and *fp is traversed through the buffer. Once *fp finds a delimiter like white space or semicolon, the traversed part between *bp and the encountered delimiter is identified as a token. Now, *bp and *fp are set to the succeeding block of the delimiter to continue searching for tokens.




**The drawback of One-buffer schema**: When the string we want to read is longer than the buffer length, before reading the whole string, the end of the buffer is reached, and the whole buffer has to be reloaded with the rest of the string, which makes identification hard

Hence, the **Two Buffer scheme** is introduced.

Here, two buffers of the same size are used. The advantage is that when the first buffer is filled, the second buffer is loaded, and vice versa. We won't lose strings midways.




Whenever fp* moves forward, eof checks if it is reaching the end to reload the second buffer. So, this is how an input program is read, and tokens are divided.

The next question **is how the lexical analyzer can match patterns with lexemes to check the validity of lexemes with tokens.**

**Patterns:**

The Lexical analyzer has to scan and identify only a finite set of valid tokens/ lexemes from the program for which it uses patterns. Patterns are to find a valid lexeme from the program. These patterns are specified using "**Regular grammar**". All the valid tokens are given predefined patterns to check the validity of detected lexemes in the program.

## 1. Numbers

A number can be in the form of:

1. A whole number (0, 1, 2...)
2. A decimal number (0.1, 0.2...)
3. Scientific notation(1.25E), (1.25E23)

The grammar has to identify all types of numbers:

**Sample Regular grammar:**

1. Digit -> 0|1|....9
2. Digits -> Digit (Digit)*
3. Number -> Digits (.Digits)? (E[+ -] ? Digits)?
4. Number -> Digit+ (.Digit)+? (E[+ -] ? Digit+)?
   - ? represents 0 or only 1 occurrence of the previous expression
   - * represents 0 or more occurrences of the base expression
   - + represents 1 or more occurrences of the base expression

## 2. Delimiters

There are different types of delimiters like white space, newline character, tab space, etc.

**Sample Regular grammar:**

1. Delimiter -> ' ', '\t', '\n'
2. Delimiters -> delimiter (delimiter)*

## 3. Identifiers

**The rules of an identifier are:**

1. It has to start only with an alphabet.
2. After the first alphabet, it can have any number of alphabets, digits, and underscores.

**Sample Regular grammar:**

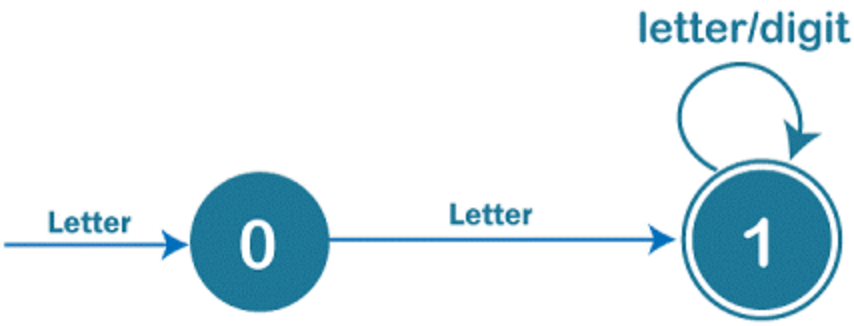
1. Letter -> a|b|....z
2. Letter -> A|B|...Z
3. Digit -> 0|1|...9
4. Identifier -> Letter (Letter/ Digit)*

Now, we have detected lexemes and pre-defined patterns for every token. The lexical analyzer needs to recognize and check the validity of every lexeme using these patterns.

**To recognize and verify the tokens, the lexical analyzer builds Finite Automata for every pattern**. Transition diagrams can be built and **converted into programs** as an intermediate step. Each state in the transition diagram represents a piece of code. Every identified lexeme walks through the Automata. The programs built from Automata can consist of switch statements to keep track of the state of the lexeme. The lexeme is verified to be a valid token if it reaches the final state.

Here are some transition diagrams. These are just examples drawn manually, but the compiler's original rules and pattern programs are way more complicated as they have to recognize all types of lexemes whichever way they are used.

## 1. Identifiers

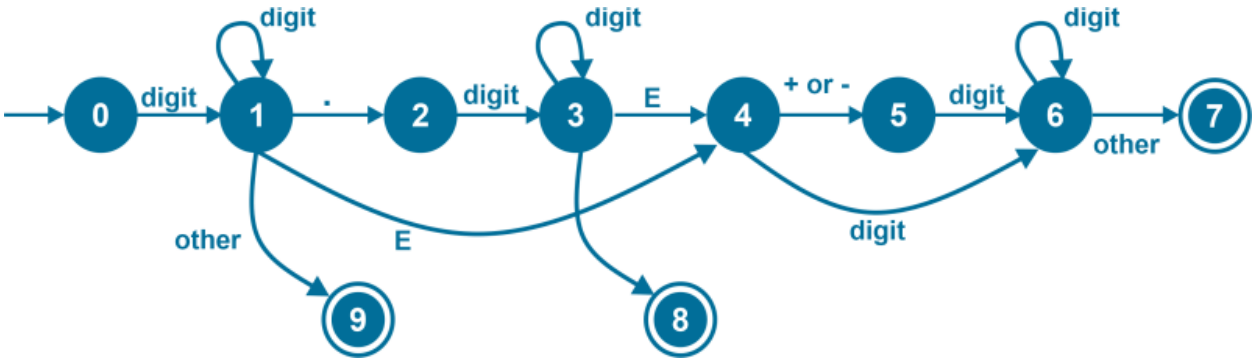



## 2. Delimiters


**White spaces:**

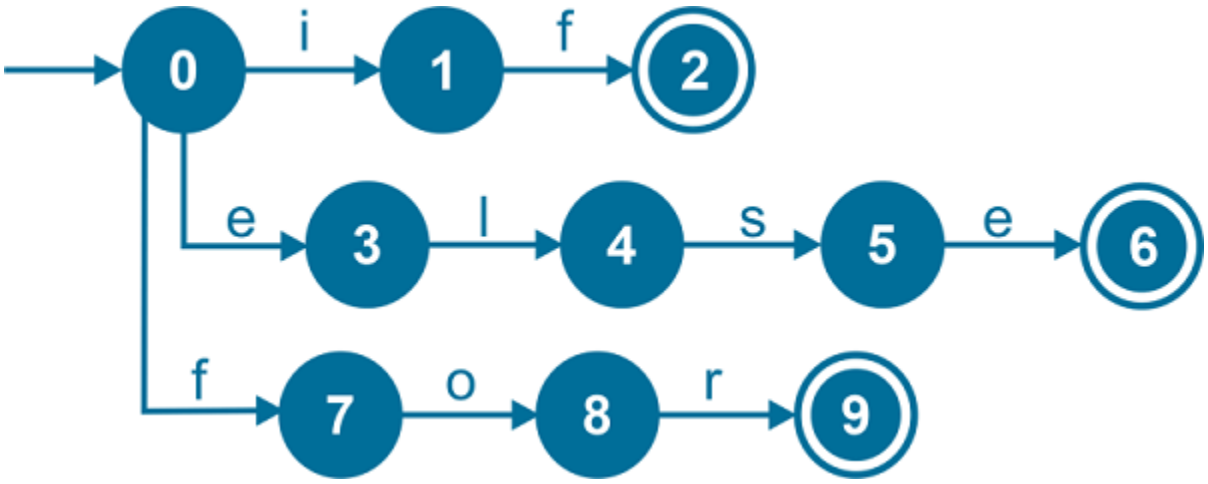
When a compiler recognizes a white space or other separating characters like '\t' and '\n', it doesn't send it to the syntax analyzer. It rather starts the whole lexical analysis process from the immediate next token. This is called Stripping the spaces from a program.

## 3. Numbers




## 4. Keywords

Identifies if, else, and for. As mentioned earlier, a keyword's letters are the pattern to identify a keyword.




## 5. Relational Operators

**GE**: Greater than or equal to

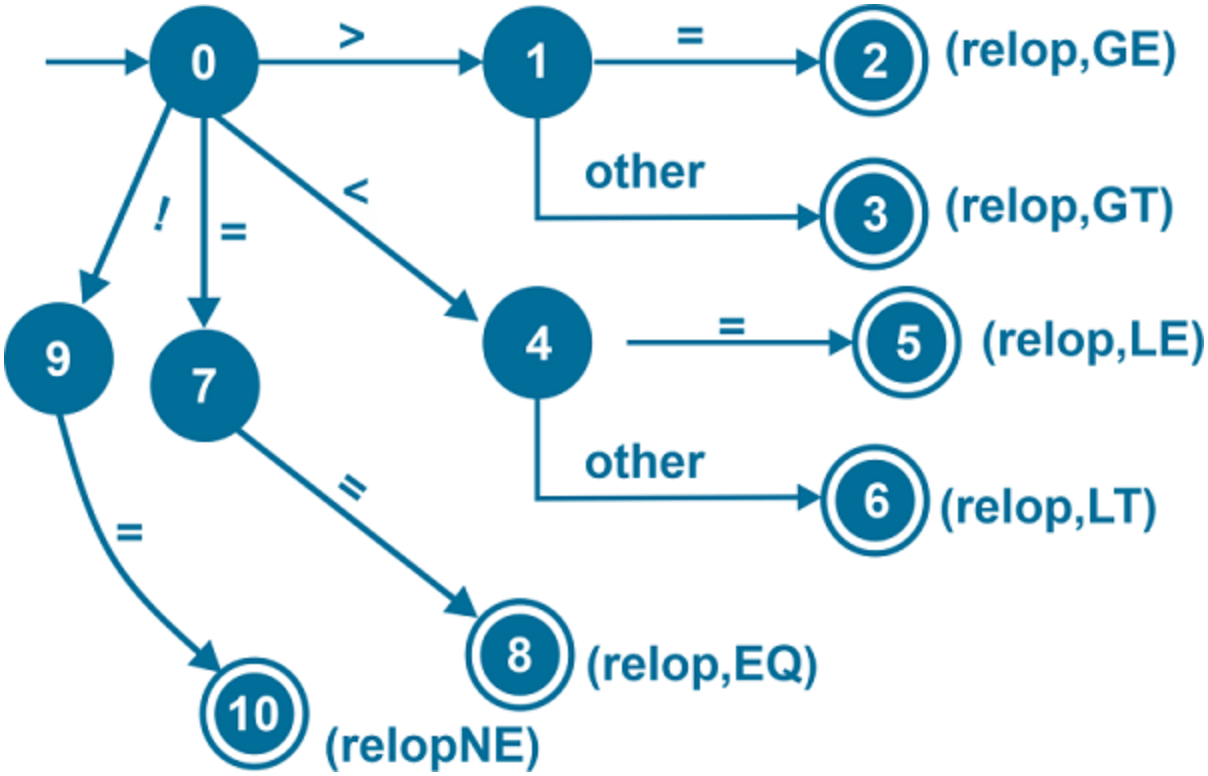
**LE**: Less than or equal to

**GT**: Greater than

**LT**: Less than

**EQ**: Equals to

**NE**: Not equal to




## Attributes for Tokens:

In a program, many lexemes can correspond to one token. We learned that the lexical analyzer sends a sequence of tokens to the next phase. Still, the rest of the phases need additional information about the lexeme to perform different operations.

Both 0 and 1 are identified as Numbers. But, if we send that there is a Number in the program, it isn't sufficient for the Code generator. Hence, the **tokens are sent as a pair of <Token name, Attribute value> to the Syntax analyzer.**

In the case of complex tokens like Identifiers, The Attribute value is a pointer pointing to the identifier's entry in the symbol table to associate more information about the identifier.

**Now, what exactly does the Lexical Analyzer send the Syntax Analyzer?**

Let us take an example of grammar for a simple if-else branching statement:

1. Stmt -> if expr then Stmt
2. If expr then stmt else Stmt
3. ε

4. expr -> term relop term
5. term
6. term -> id
7. number

Here is the output of the lexical analyzer to the next phase for this snippet:

| LEXEMES | TOKEN NAME | ATTRIBUTE VALUE |
|---|---|---|
| Any white space | - | - |
| if | if | - |
| then | then | - |
| else | else | - |
| Any Identifier | id | Pointer to symbol table entry of id |
| Any number | number | Pointer to symbol table entry of id |
| < | relop | LT |
| > | relop | GT |
| >= | relop | GE |
| <= | relop | LE |
| == | relop | EQ |
| <> | relop | NE |

A lexeme is like an instance of a token, and the attribute column is used to show which lexeme of the token is used. For every lexeme, the 1$^{st}$ and 2$^{nd}$ columns of the above table are sent to the Syntax Analyzer.