**COMPILER DESIGN**

**AUTHORS :**

**Alfred V.Aho**

**D.Ullman**

**UNIT-1**

**INTRODUCTION**

**\*Language Processors**

A language processor is a special type of a computer software that has the capacity of translator the source code or program codes into machine codes.

* A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious(long) task to write a computer program directly in machine code.
* The programs are written mostly in high level languages like Java, C++, Python etc. and are called **source code**.
* These source code cannot be executed directly by the computer and must be converted into machine language to be executed.
* Hence, a special translator system software is used to translate the program written in high-level language into machine code is called **Language Processor** and the program after translated into machine code (object program / object code).

The language processors can be any of the **following three types:**

1. **Compiler :**

The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.

**Example:** C, C++, C#, Java

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.



1. **Assembler**

The Assembler is used to translate the program written in Assembly language into machine code. The source program is an input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.



1. **Interpreter**

* The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter.
* If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message.
* The interpreter moves on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

**Example:** Perl, Python and Matlab.

**Difference between Compiler and Interpreter**

| **COMPILER** | **INTERPRETER** |
| --- | --- |
| A compiler is a program which converts the entire source code of a programming language into executable machine code for a CPU. | Interpreter takes a source program and runs it line by line, translating each line as it comes to it. |
| Compiler takes large amount of time to analyse the entire source code but the overall execution time of the program is comparatively faster. | Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower. |
| Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present anywhere in the program. | Its Debugging is easier as it continues translating the program until the error is met |
| Generates intermediate object code. | No intermediate object code is generated. |
| Examples: C, C++, Java | Examples: Python, Perl |
|  |  |

# \*Phases of a Compiler

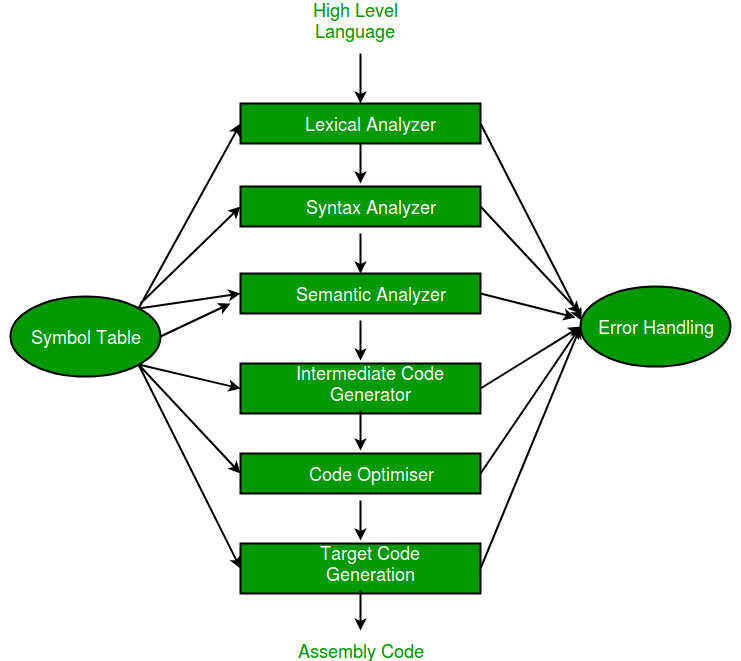
# We basically have two phases of compilers, namely Analysis phase and Synthesis phase.

# Analysis phase creates an intermediate representation from the given source code.

# Synthesis phase creates an equivalent target program from the intermediate representation.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

Let us understand the phases of a compiler.



**Symbol Table**

It is a data structure being used and maintained by the compiler, consists all the identifier’s name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The analysis of a source program is divided into **mainly three phases.** They are:

1. **Linear Analysis-**  
   This involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various **tokens** having a collective meaning.

types of **tokens**: Keywords, Identifiers, Constant, Strings, Operators, etc.

**Ex: int**  a = 12;

1. **Hierarchical Analysis-**  
   In this analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.
2. **Semantic Analysis-**  
   This phase is used to check whether the components of the source program are meaningful or not.

The compiler has **two modules namely front end and back end.**

Front-end constitutes of the Lexical analyzer, semantic analyzer, syntax analyzer and intermediate code generator.

And the rest are assembled to form the back end.

1. [**Lexical Analyzer**](https://www.geeksforgeeks.org/compiler-lexical-analysis/)

* It is also called **scanner**.
* It takes the output of preprocessor (which performs file inclusion and macro expansion) as the input which is in pure high-level language. It reads the characters from source program and groups them into lexemes (sequence of characters that “go together”).
* Each lexeme corresponds to a token.
* Tokens are defined by regular expressions which are understood by the lexical analyzer.
* It also removes lexical errors (for e.g., erroneous characters), comments and white space.

1. [**Syntax Analyzer**](https://www.geeksforgeeks.org/compiler-design-introduction-to-syntax-analysis/)

* It is sometimes called as **parser**.
* It constructs the parse tree.
* It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.
* The parse tree is also called the derivation tree.
* Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

1. **Semantic Analyzer –** It verifies the parse tree, whether it’s meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking and Flow control checking.

Ex: Int a=12.678 or a++ or a<=10

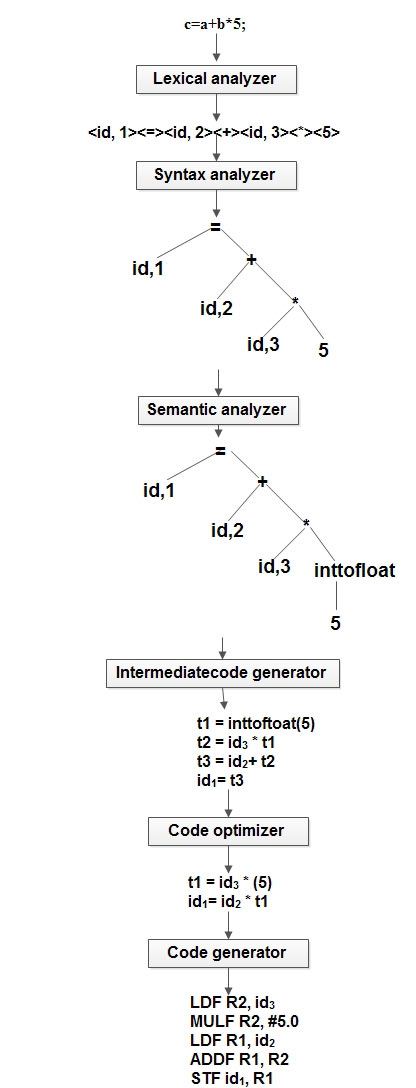
1. [**Intermediate Code Generator**](https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)**–** It generates intermediate code, that is a form which can be readily executed by machine. We have many popular intermediate codes.

Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

1. [**Code Optimizer**](https://www.geeksforgeeks.org/compiler-design-code-optimization/)**–** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimisation can be categorized into two types: machine dependent and machine independent.
2. **Target Code Generator –** The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

All these six phases are associated with the symbol table manager and error handler as shown in the above block diagram.



Note : Mistake code optimizer output (line 2) i.e., id1 = id2 + t1

**PASS AND PHASE**

A compiler can have many phases and passes.

* **Pass** : A pass refers to the traversal of a compiler through the entire program.
* **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

# \*Bootstrapping

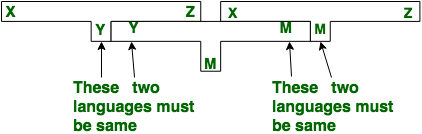
**Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

* Writing a compiler for any high-level language is a complicated process. It takes lot of time to write a compiler from scratch.
* Hence simple language is used to generate target code in some stages.
* To clearly understand the **Bootstrapping** technique, consider a following scenario.

Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is saying Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

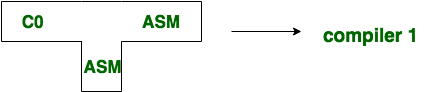
Following diagram illustrates the above scenario.

**Example:**  
We can create compiler of many different forms. Now we will generate.

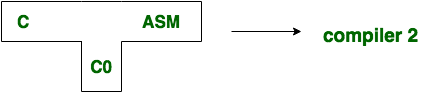


Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

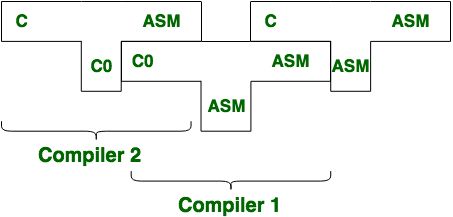
* **Step-1:** First we write a compiler for a small of C in assembly language.



* **Step-2:** Then using with small subset of C i.e. C0, for the source language C the compiler is written.



* **Step-3:** Finally, we compile the second compiler. using compiler 1 ,the compiler 2 is compiled.



* **Step-4:** Thus, we get a compiler written in ASM which compiles C and generates code in ASM.

# \*Compiler construction tools

* The compiler writer can use some specialized tools that help in implementing various phases of a compiler.
* These tools assist in the creation of an entire compiler or its parts.
* The most successful tools are those that hide the details of the algorithm and produce components.

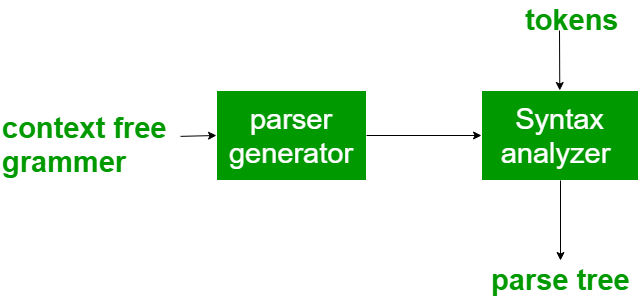
Some commonly used compiler construction tools include:

1. Parser generators.  
2. Scanner generators.  
3. Syntax-directed translation engines.  
4. Automatic code generators.  
5. Data-flow analysis engines.  
6. Compiler-construction toolkits.

1. **Parser Generator –**

* It produces syntax analysers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar.
* It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
* Ex: YACC, Bison

**Input:** Grammatical description of a programming language.  
**Output:** Syntax analyzers.

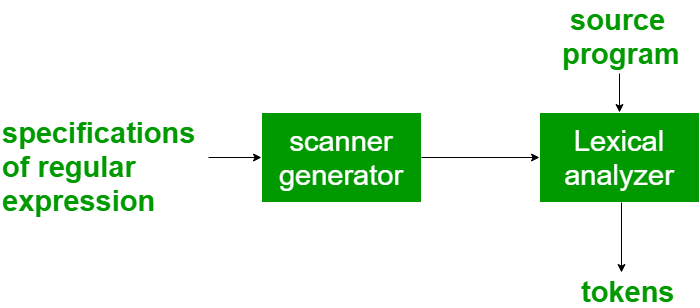


1. **Scanner Generator –**

* It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language.
* It generates a finite automaton to recognize the regular expression.
* Example: Lex,Flex

**Input:** Regular expression description of the tokens of a language  
**Output:** Lexical analyzers.

Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.



1. **Syntax directed translation engines –**

* It generates intermediate code with three address format from the input that consists of a parse tree.
* These engines have routines to traverse the parse tree and then produces the intermediate code.
* In this, each node of the parse tree is associated with one or more translations.

**Input:** Parse tree.  
**Output:** Intermediate code.

Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

1. **Automatic code generators –**

* It generates the machine language for a target machine.
* Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator.
* A template matching process is used.
* An intermediate language statement is replaced by its equivalent machine language statement using templates.

**Input:** Intermediate language.  
**Output:** Machine language.

Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

1. **Data-flow analysis engines –**

* It is used in code optimization.
* Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.

1. **Compiler construction toolkits –**

* It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

# \* Applications of Compiler Technology

* Compiler design helps in implementation Of High-Level Programming Languages.
* Techniques used in Lexical Ananlyzer can be used in text editor, information retrieval system, pattern recognition program.
* Techniques used in parser will be used in Query processing system such as SQL.
* Most of the techniques used in CD will be used in NLP system.
* Support optimization for Computer Architectures.
* Design of New Computer Architecture.
* Widely used for Translating Programs.
* Used with other Software Productivity Tools(type checking and bound checking).

# Compiler Design - Lexical Analysis

* Lexical analysis is the first phase of a compiler.
* It takes the modified source code from language pre-processors that are written in the form of sentences.
* The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
* If the lexical analyzer finds a token invalid, it generates an error.
* The lexical analyzer works closely with the syntax analyzer.
* It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

# Introduction of Lexical Analysis

* Lexical Analysis is the first phase of the compiler also known as a **scanner.**
* It converts the High-level input program into a sequence of **Tokens**.
* Lexical Analysis can be implemented with the [Deterministic finite Automata](http://quiz.geeksforgeeks.org/toc-finite-automata-introduction/).
* The output is a sequence of tokens that is sent to the parser for syntax analysis.

**What is a token?**  
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

**Example of tokens:**

* Type token (id, number, real, . . . )
* Punctuation tokens (IF, void, return, . . . )
* Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name, etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',' ';' etc

**Example of Non-Tokens:**

Comments, pre-processor directive, macros, blanks, tabs, newline, etc.

**Lexeme:**

The sequence of characters matched by a pattern to form  
the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.

**eg-** “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;” .

**How Lexical Analyzer functions**  
1. Tokenization i.e. Dividing the program into valid tokens.  
2. Remove white space characters.  
3. Remove comments.  
4. It also provides help in generating error messages by providing row numbers and column numbers.

**For example,** in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Suppose we pass a statement through lexical analyzer –

**a = b + c** ;           It will generate token sequence like this:

**id=id+id**;            Where each id refers to it’s variable in the symbol table referencing all details.

**Exercise 1:**

int main()

{

// 2 variables

int a, b;

a = 10;

return 0;

}

All the valid tokens are:

'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';'

'a' '=' '10' ';' 'return' '0' ';' '}'

Above are the valid tokens.  
You can observe that we have omitted comments.

**Exercise 2:**

**Printf ( “ compiler design” ) ;**  
   
There are 5 valid token in this printf statement.

**Exercise 3:**  
Count number of tokens :

int main()

{

int a = 10, b = 20;

printf("sum is :%d",a+b);

return 0;

}

Answer: Total number of token: 27.

**Exercise 4:**

Count number of tokens :

int max(int i);

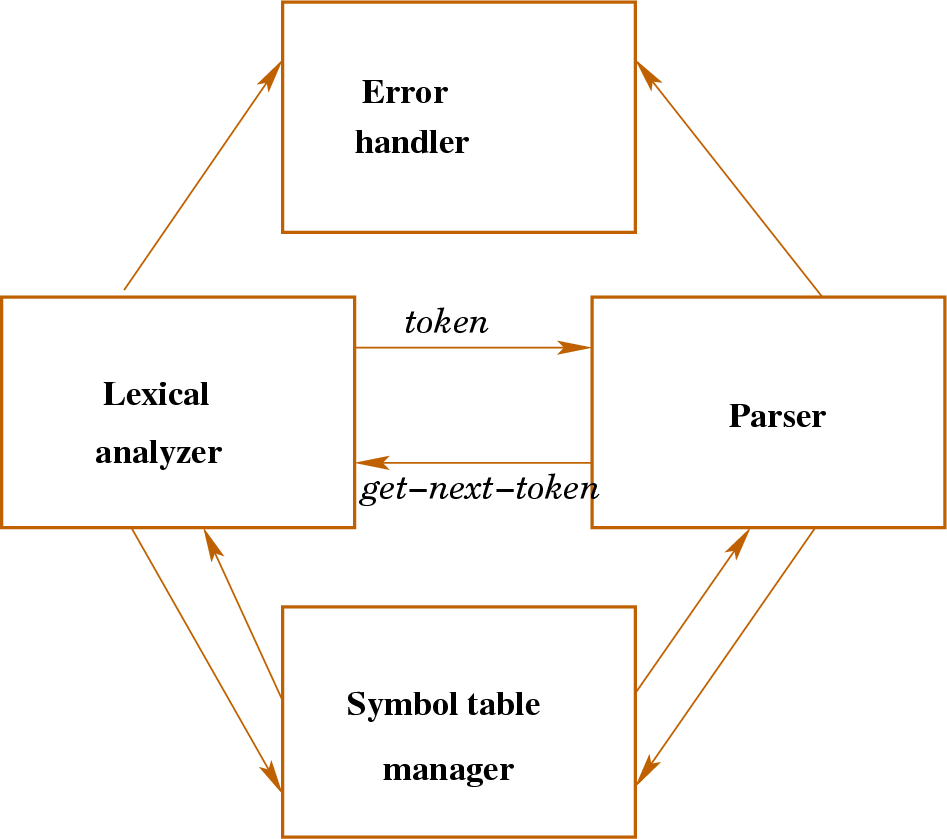
* Lexical analyzer first read **int** and finds it to be valid and accepts as token
* **max** is read by it and found to be a valid function name after reading**(**
* **int**  is also a token , then again**i** as another token and finally **;**

Answer: Total number of tokens 7:

int, max, ( ,int, i, ), ;

## **The role of the lexical analyzer in the compiler**

Upon receiving a get-next-tohen command from the parser, the lexical analyzer reads input characters until it can identify the next token.



# \*Input Buffering

# The lexical analyzer scans the input from left to right one character at a time.

# It uses two pointers beginning pointer ptr(bp) and forward pointer ptr(fp) to keep track of the pointer of the input scanned.

# 

# Initially both the pointers point to the first character of the input string as shown below.

# 

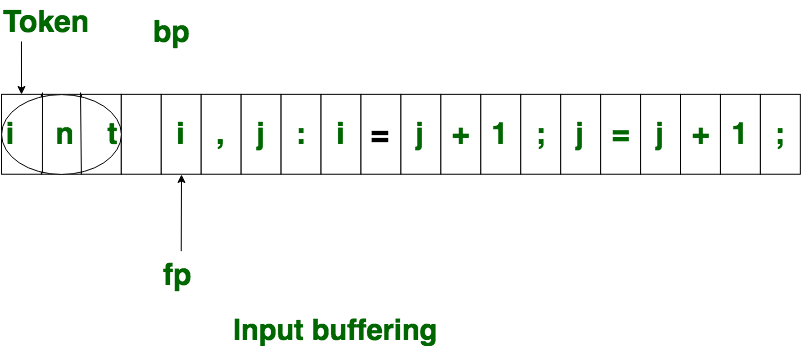
# 

* The forward ptr moves ahead to search for end of lexeme.
* As soon as the blank space is encountered, it indicates end of lexeme.
* In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. Then both the begin ptr(bp) and forward ptr(fp) are set at next token.
* The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.
* A block of data is first read into a buffer, and then second by lexical analyzer. there are **two methods** used in this context:

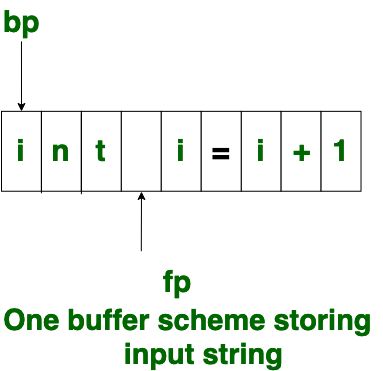
One Buffer Scheme, and

Two Buffer Scheme.

These are explained as following below.

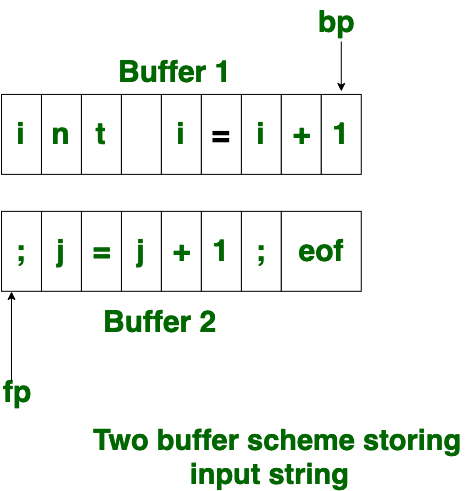


**1.One Buffer Scheme:**  
In this scheme, **only one buffer is used** to store the input string but **the problem** with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes **overwriting** the first of lexeme.



**2.Two Buffer Scheme:**

* To overcome the problem of one buffer scheme, in this method **two buffers are used** to store the input string.
* The first buffer and second buffer are scanned alternately.
* when end of current buffer is reached the other buffer is filled.
* The only **problem** with this method is that if length of the lexeme is longer, than the length of the buffer then scanning input cannot be scanned completely.
* Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer.
* Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer.
* when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started.
* In the same way when second **eof** is obtained then it indicates of second buffer.
* Alternatively, both the buffers can be filled up until end of the input program and stream of tokens is identified.
* This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.



**\*SPECIFICAT**[**ION OF TOKENS**](http://notes.pmr-insignia.org/)

There are 3 sp[ecifications of tokens:](http://notes.pmr-insignia.org/)

1. Strings
2. Language
3. Regular expr[ession](http://notes.pmr-insignia.org/)

## **Strings and** [**Languages**](http://notes.pmr-insignia.org/)

* An **alphabet** or character class is a finite set of symbols.
* A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
* A **language** is any countable set of strings over some fixed alphabet.
* In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
* The length of a string s, usually **written |s|,** is the number of occurrences of symbols in s.

For example,

**banana** is a string of length **six.**

|banana| = 6

| | = 0

The empty string, **denoted ε,** is the string of **length zero.**

## **Operations on strings**

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s.

For example, ban is a prefix of banana.

1. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.

For example, nana is a suffix of banana.

1. A **substring** of s is obtained by deleting any prefix and any suffix from s.

For example, nan is a substring of banana.

1. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
2. A **subsequence of s** is any string formed by deleting zero or more not necessarily consecutive positions of s.

For example, baan is a subsequence of banana.

## **Operations on languages:**

The following [are the operations that can be applied to languages:](http://notes.pmr-insignia.org/)

1.Union 2.Concatenati[on](http://notes.pmr-insignia.org/) 3.Kleene closur[e](http://notes.pmr-insignia.org/) 4.Positive clos[ure](http://notes.pmr-insignia.org/)

The following [example shows the operations on strings:](http://notes.pmr-insignia.org/) Let L={0,1} and S={a,b,c}

1. Union [: L U S={0,1,a,b,c}](http://notes.pmr-insignia.org/)
2. Concatena[tion : L.S={0a,1a,0b,1b,0c,1c}](http://notes.pmr-insignia.org/)
3. Kleene clos[ure : L](http://notes.pmr-insignia.org/)\*[={ ε,0,1,00,01,10,11}](http://notes.pmr-insignia.org/)
4. Positive clos[ure : L](http://notes.pmr-insignia.org/)+[={0,1,00,10,01,11}](http://notes.pmr-insignia.org/)

## **Regular Expr**[**essions**](http://notes.pmr-insignia.org/)

Each regular expression **r** denotes a **language L(r).**

Here are the rules that define the regular expressions over some **alphabet Σ** and the languages that those expressions denote:

1. **ε** is a regular expression, and **L(ε) is { ε },** that is, the language whose sole member is the empty string.
2. If “a” is a symbol in Σ, then “a” is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with “a” in its one position.
3. Suppose **r and s are regular expressions** denoting the languages **L(r) and L(s).**
   1. Then, **(r)|(s)** is a regular expression denoting the language L(r) U L(s).
   2. **(r)(s)** is a regular expression denoting the language L(r)L(s).
   3. **(r)\*** is a regular expression denoting (L(r)) \*.
   4. **(r)** is a regular expression denoting L(r).
4. The **unary operator \*** has highest precedence and is left associative.
5. **Concatenation** has second highest precedence and is left associative.
6. **|** has lowest precedence and is left associative.

## **Regular set**

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are **equivalent** and Write **r = s.**

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance,

r|s = s|r is **commutative.**

r|(s|t) =(r|s) |t is **associative.**

## **Regular Definitions**

Giving [names to regular expressions is referred to as a Regular definition. If Σ](http://notes.pmr-insignia.org/) is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r 1

d2 → r2

………

dn → rn

1. Each di is a [distinct name.](http://notes.pmr-insignia.org/)
2. Each ri is a r[egular expression over the alphabet Σ U {dl, d2, . . . , di-l}.](http://notes.pmr-insignia.org/)

**Example:** Ident[ifiers is the set of strings of letters and digits beginning with a letter.](http://notes.pmr-insignia.org/) Regular definition for this set:

letter → A | B [| …. | Z | a | b | …. | z |](http://notes.pmr-insignia.org/)

digit → 0 | 1 | […. | 9](http://notes.pmr-insignia.org/)

id → letter ( l[etter | digit ) \*](http://notes.pmr-insignia.org/)

## **Shorthands**

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

## **One or more instances (+):**

* The unary postfix operator + means “one or more instances of”.
* If r is a regular expression that denotes the language L(r), then ( r )+ is a regular expression that denotes the language (L (r ))+
* Thus, the regular expression **a+ denotes the set of all strings of one or more a’s.**
* The operator + has the same precedence and associativity as the operator \*.

## **Zero or one instance ( ?):**

* The unary postfix operator **? means “zero or one instance of”.**
* The notation r? is a shorthand for r | ε.
* If “r‟ is a regular expression, then ( r )? is a regular expression that denotes the language L( r ) U { ε }.

## **Character Classes:**

* The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
* Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
* We can describe identifiers as being strings generated by the regular expression, **[A–Za–z]**[**[A–Za–z0–9]\***](http://notes.pmr-insignia.org/)

## **Non-regular S**[**et**](http://notes.pmr-insignia.org/)

A lang[uage which cannot be described by any regular expression is a non-re](http://notes.pmr-insignia.org/)gular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

**\*RECOGNITI**[**ON OF TOKENS**](http://notes.pmr-insignia.org/)

Consider the [following grammar fragment:](http://notes.pmr-insignia.org/)

**stmt → if expr** [**then stmt**](http://notes.pmr-insignia.org/)

**| if expr** [**then stmt else stmt**](http://notes.pmr-insignia.org/)

**| ε**

**expr → term** [**relop term**](http://notes.pmr-insignia.org/)

**| term**

**term → id**

**| num**

**Ex: if ( a<10)**

**Printf(“am inside if”);**

**a=a+1;**

**else**

**Printf(“ am inside else”);**

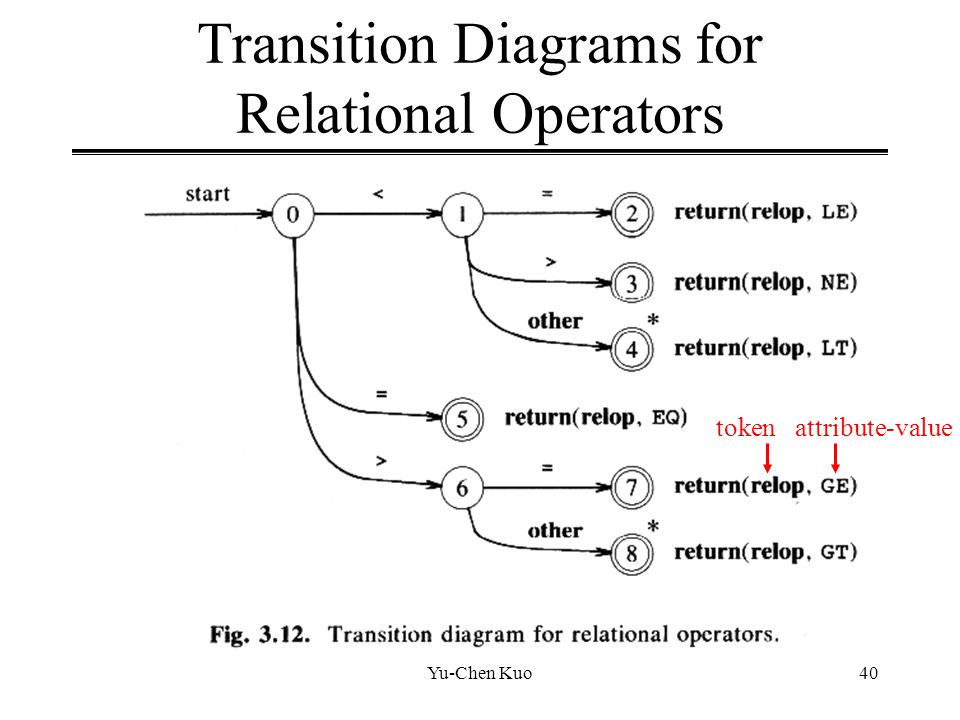
where the **terminals** if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

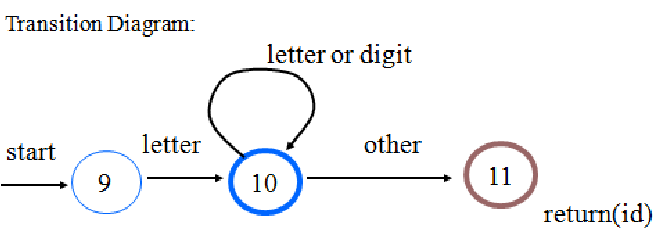
|  |  |  |
| --- | --- | --- |
| **if** | **→** | **if** |
| **then** | **→** | **then** | |
| **else** | **→** | **else** | |
| **relop** | **→** | **<|<=|=|<>|>|>=** | |
| **id** | **→** | **letter(letter|digit)\*** | |
| **num** | **→** | **digit+ (.digit+)?(E(+|-)?digit+)?** | |

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

## **Transition diagrams**

It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.





## **A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER**

There is a wide range of tools for constructing lexical analyzers.

* LEX
* FLEX

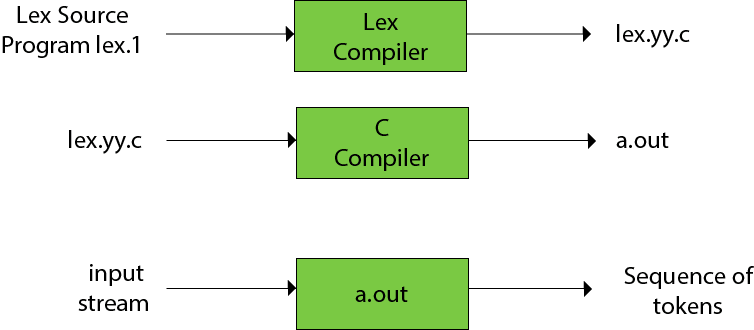
# LEX

# [What is LEX?](https://ecomputernotes.com/compiler-design/lex-use-of-lex)

* + - Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
    - Lex tool itself is a lex compiler.
* Lex is a program that generates lexical analyzer.
* It is used with YACC parser generator.
* The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
* It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

### **The function of Lex is as follows:**

* Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
* Finally C compiler runs the lex.yy.c program and produces an object program a.out.
* a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



## **Structure of Lex Programs**

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

**{ definitions }**

**%%**

**{ *Translation* rules }**

**%%**

**{ user subroutines }**

**Definitions** include declarations of constant, variable and regular definitions.

**Translation Rules** define the statement of form p1 {action1} p2 {action2}....pn {action}. It contains regular expressions and code segments.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

**User subroutines** are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

* + - ***yylval***is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
    - The attribute value can be numeric code, pointer to symbol table or nothing.
    - Another tool for lexical analyzer generation is Flex.
    - Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
    - Once a match is found, the associated action takes place to produce token.
    - The token is then given to parser for further processing.

**Example:**

|  |  |
| --- | --- |
| /\*Declarations section start here\*/ | |
|  |  |
|  | /\* Auxiliary declarations start here\*/ |
|  |  |
|  | %{ |
|  | #include <stdio.h> |
|  | int global\_variable; |
|  | %} |
|  |  |
|  | /\*Auxiliary declarations end & Regular definitions start here\*/ |
|  |  |
|  | number [0-9]+ //Regular definition |
|  | op [-|+|\*|/|^|=] //Regular definition |
|  |  |
|  | /\*Declarations section ends here\*/ |
|  |  |
|  | %% |
|  |  |
|  | /\* Rules \*/ |
|  |  |
|  | %% |
|  |  |
|  | /\* Auxiliary functions \*/ |

Rules in a LEX program consists of two parts :

1. The pattern to be matched

2. The corresponding action to be executed

**Example:**

|  |  |
| --- | --- |
| /\* Declarations\*/ | |
|  | %% | |
|  |  | |
|  | {number} {printf(“ number”);} | |
|  | {op} {printf(“ operator”);} | |
|  |  | |
|  | %% | |
|  | /\* Auxiliary functions \*/ | |

The pattern to be matched is specified as a regular expression.

Sample Input/Output for the above example:

I: 234

O: number

I: \*

O: operator

I: 2+3

O: number operator number

LEX obtains the regular expressions of the symbols 'number' and 'op' from the declarations section and generates code into a function *yylex()* in the *lex.yy.c* file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

### Auxiliary functions

LEX generates C code for the rules specified in the Rules section and places this code into a single function called *yylex()*.In addition to this LEX generated code, the programmer may wish to add his own code to the *lex.yy.c* file. The auxiliary functions section allows the programmer to achieve this.

|  |  |  |  |
| --- | --- | --- | --- |
| /\* Declarations \*/ | | | |
|  | | | %% | |
|  | | | /\* Rules \*/ | |
|  | | | %% | |
|  | | |  | |
|  | | | int main() | |
|  | | { |
|  | | yylex(); | | |
|  | | return 1; | | |
|  | } |

The auxiliary declarations and auxiliary functions are copied as such to the lex.yy.c file

Once the code is written, *lex.yy.c* maybe generated using the command *lex "filename.l"* and compiled as *gcc lex.yy.c*