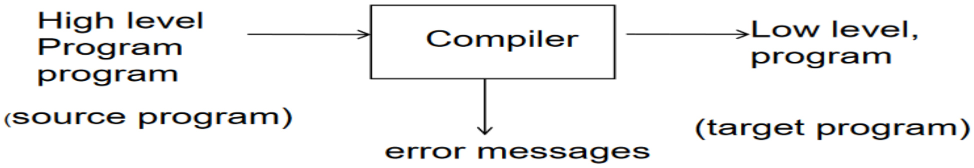
Introduction to Compiler Design

Compiler: A compiler is a software that translates code written in high-level language (i.e., source language) functionally equivalent into target language.

Ex: source languages like C, Java ... etc. Compilers are user friendly. The target language is like machine language, which is efficient for hardware.



A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. Compiler process goes through lexical, syntax, and semantic analysis at the front end, and code generation and optimization at a back-end.

Features of Compilers

- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

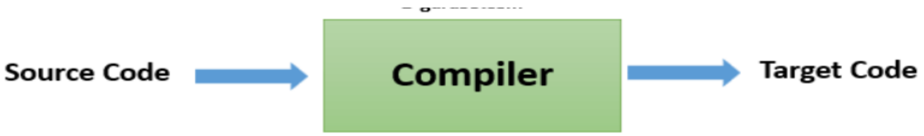
Types of Compiler

- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

Passes: The number of iterations to scan the source code, till to get the executable code is called as a pass.

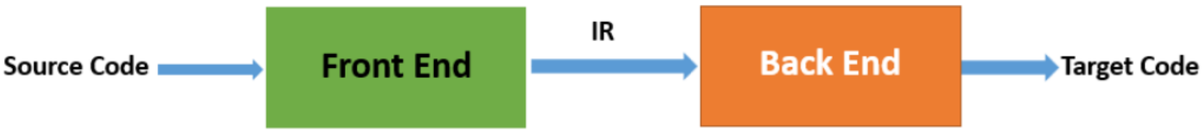
Compiler is two pass. Single pass requires more memory and multipass require less memory

Single Pass Compilers:



In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

Two Pass Compilers:

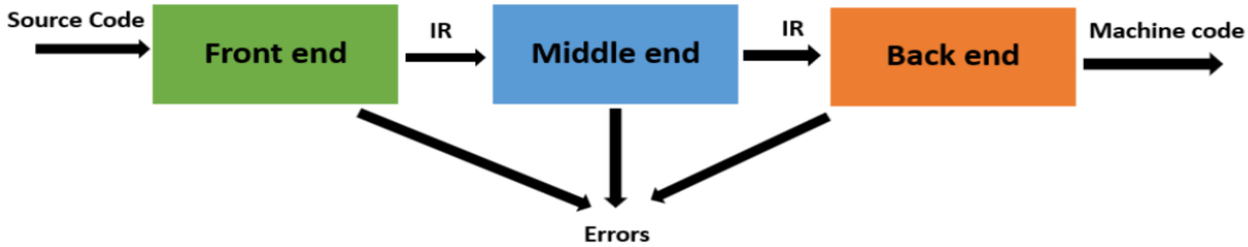


Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Multipass Compilers:



The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory. It is also known as 'Wide Compiler'.

Tasks of Compiler:

Main tasks performed by the Compiler are:

- Breaks up the up the source program into pieces and impose grammatical structure on them
- Allows you to construct the desired target program from the intermediate representation and also create the symbol table
- Compiles source code and detects errors in it
- Manage storage of all variables and codes.
- Support for separate compilation
- Read, analyze the entire program, and translate to semantically equivalent
- Translating the source code into object code depending upon the type of machine

History of Compiler:

Important Landmark of Compiler's history are as follows:

- The "compiler" word was first used in the early 1950s by Grace Murray Hopper
- The first compiler was build by John Backum and his group between 1954 and 1957 at IBM
- COBOL was the first programming language which was compiled on multiple platforms in 1960
- The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution

Language processing system:

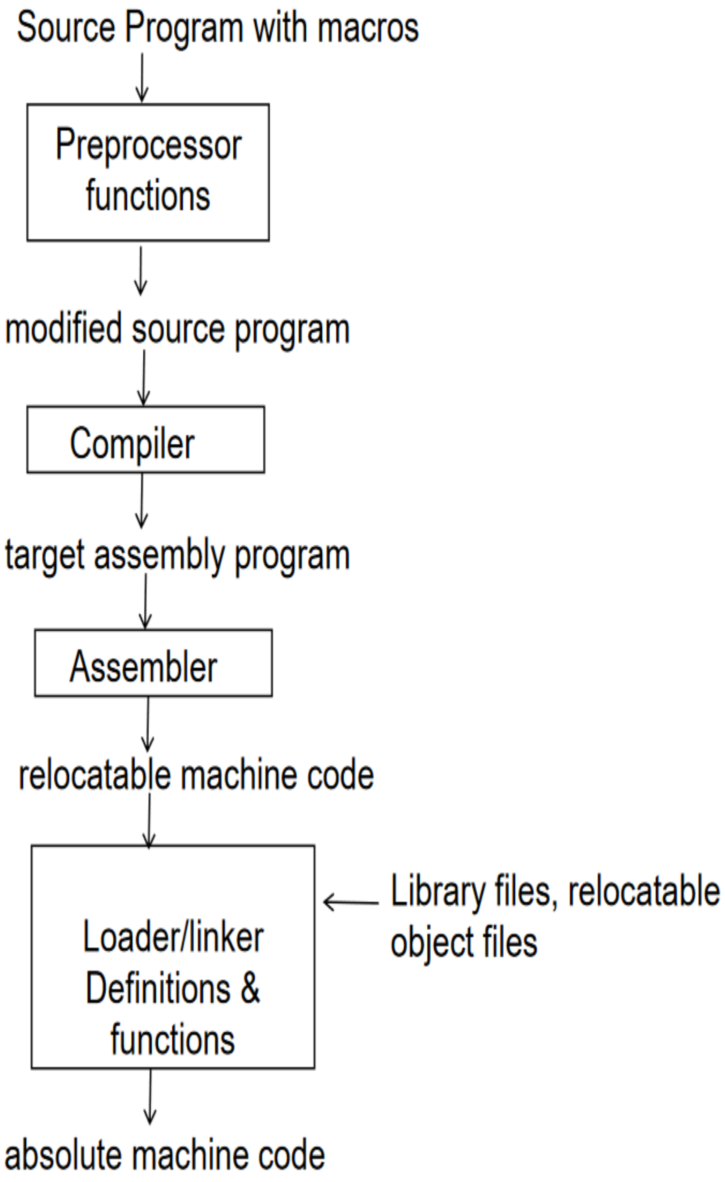
Before knowing about the concept of compilers, you first need to understand a few other tools which work with compilers.

Preprocessor: The preprocessor takes the source code as input and gives the expanded source code as output. As the source code includes header files, library files, macros.It evaluates the macros and includes files, performs conditional compilation etc.

Some languages like PASCAL, they will not perform preprocessing as it will not support #include and macros. So preprocessing is optional i.e. it depends on the language.

Loading: Loading is a process of getting an executable running on the machine.

Linking: Linking is a process of combining multiple code modules into a runnable



- **Preprocessor**: The preprocessor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.

- **Interpreter**: An interpreter is like Compiler which translates high-level language into low-level machine language. The main difference between both is that interpreter reads and transforms code line by line. Compiler reads the entire code at once and creates the machine code.

- **Assembler**: It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.

- **Linker**: The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.

- **Loader**: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.

- **Cross-compiler**: A cross compiler is a platform which helps you to generate executable code.

- **Source-to-source Compiler**: Source to source compiler is a term used when the source code of one programming language is translated into the source of another language.

Language Processors:

Translator: A software system that converts the source code from one form of the language to another form of language is called as translator. Widely used translators include

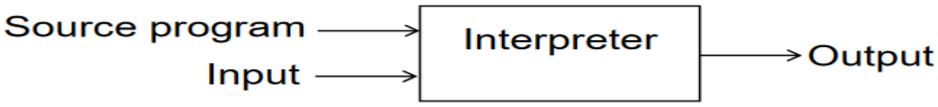
(1) Interpreter (2) Compiler (3) Assembler.

Interpreter compiles and executes programming language statements one by one in an interleaved manner.

Compiler converts source code of high level language into low level language.

Assembler converts assembly language code into binary code.

Interpreter: It is a computer program that executes instructions written in a programming language. It either execute the source code directly or translate source code into some efficient intermediate representation and immediately execute this.



Ex: Early versions of Lisp programming language, BASIC.

Why use a Compiler?

- Compiler verifies entire program, so there are no syntax or semantic errors
- The executable file is optimized by the compiler, so it is executes faster
- Allows you to create internal structure in memory
- There is no need to execute the program on the same machine it was built
- Translate entire program in other language

- Generate files on disk
- Link the files into an executable format
- Check for syntax errors and data types
- Helps you to enhance your understanding of language semantics
- Helps to handle language performance issues
- Opportunity for a non-trivial programming project
- The techniques used for constructing a compiler can be useful for other purposes as well

Application of Compilers:

- Compiler design helps full implementation Of High-Level Programming Languages
- Support optimization for Computer Architecture Parallelism
- Design of New Memory Hierarchies of Machines
- Widely used for Translating Programs
- Used with other Software Productivity Tools

Phases of Complier

Phases: Compilation process is partitioned into some subproceses called phases.

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.

There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code. The phases of a compiler are:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator

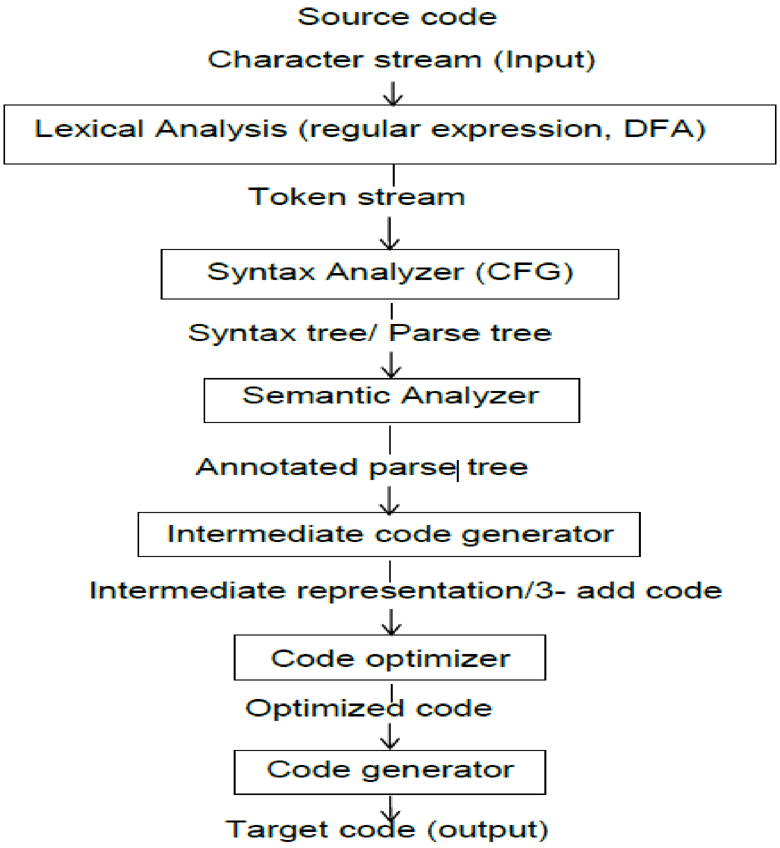


Fig: Phases of Compiler

Lexical analysis (or) Scanning:

It is the first phase of a compiler. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Ex: Consider the statement: if (a < b)
In this sentence the tokens are if, (, a, <, b,).
Number of tokens = 6
Identifiers: a, b
Keywords: if
Operators: <, (,)

Syntax Analyzer (or) Parser:

I. Tokens are grouped hierarchically into nested collections with collective meaning.

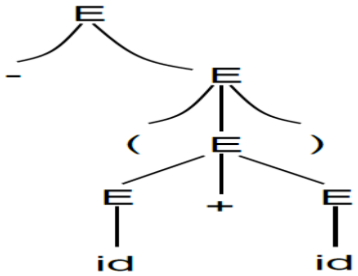
II. A context free grammar (CFG) specifies the rules or productions for identifying constructs

III. that are valid in a programming language. The output is a parse/ syntax/derivation tree.

Ex: Parse tree for –(id + id) using the following grammar:

E → E + E
E → E * E
E → -E
E → (E)
E → id

The parse tree for the string – (id + id) is



Semantic analysis:

✓ It checks the source program for semantic errors.
✓ Type checking is done in this phase, where the compiler checks that each operator has matching operands for semantic consistency with the language definition.
✓ Gathers the type information for the next phases.

Ex: 1 the bicycle rides the boy.
This statement has no meaning, but it is syntactically correct

Ex: 2 int a;
bool b;
char c;
c = a + b;
We cannot add integer with a Boolean variable and assign it to a character variable.

Intermediate code generation:

The intermediate representation should have two important properties:
(i) It should be easy to produce.
(ii) Easy to translate into the target program

“Three address code” is one of the common forms of Intermediate code.
Three address code consists of a sequence of instructions, each of which has at most three operands.

Ex:
```
id1 = id2 + id3 × 10;
t1 : = inttoreal(10)
t2 : = id3 × t1
t3 : = id2 + t2
id1: = t3
```

Code optimization:

The output of this phase will result in faster running machine code.
Ex: For the above intermediate code the optimized code will be

```
 t1:= id3 × 10.0
id1: = id2 + t1
```

In this we eliminated t2 and t3 registers.

Code Generation:

In this phase, the target code is generated.

- Generally the target code can be either a relocatable machine code or an assembly code.
- Intermediate instructions are each translated into a sequence of machine instructions.

• Assignment of registers will also be done.

```
MOV       id3, R2
MUL       # 60.0, R2
MOV        id2, R1
ADD       R2, R1
MOV       R1, id1
```

Lexical Analysis in Compiler Design

LEXICAL ANALYSIS is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

Programs that perform lexical analysis are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and pass the data to the syntax analyzer when it demands.

What's a lexeme?

A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

What's a token?

The token is a sequence of characters which represents a unit of information in the source program.

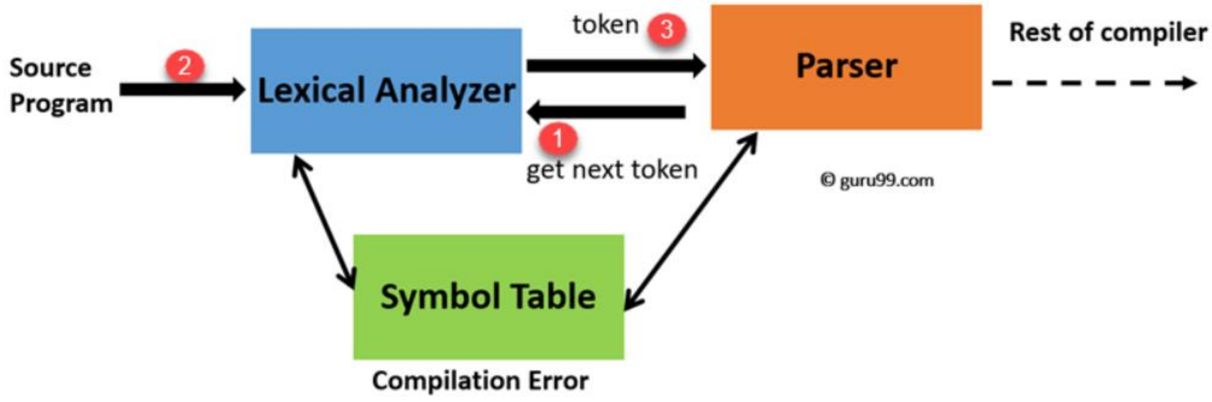
What is Pattern?

A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

Lexical Analyzer Architecture: How tokens are recognized:

The main task of lexical analysis is to read input characters in the code and produce tokens.

Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how this works-



1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

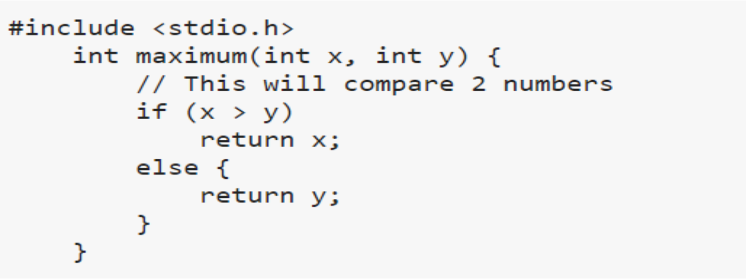
Roles of the Lexical analyzer:

Lexical analyzer performs below given tasks:

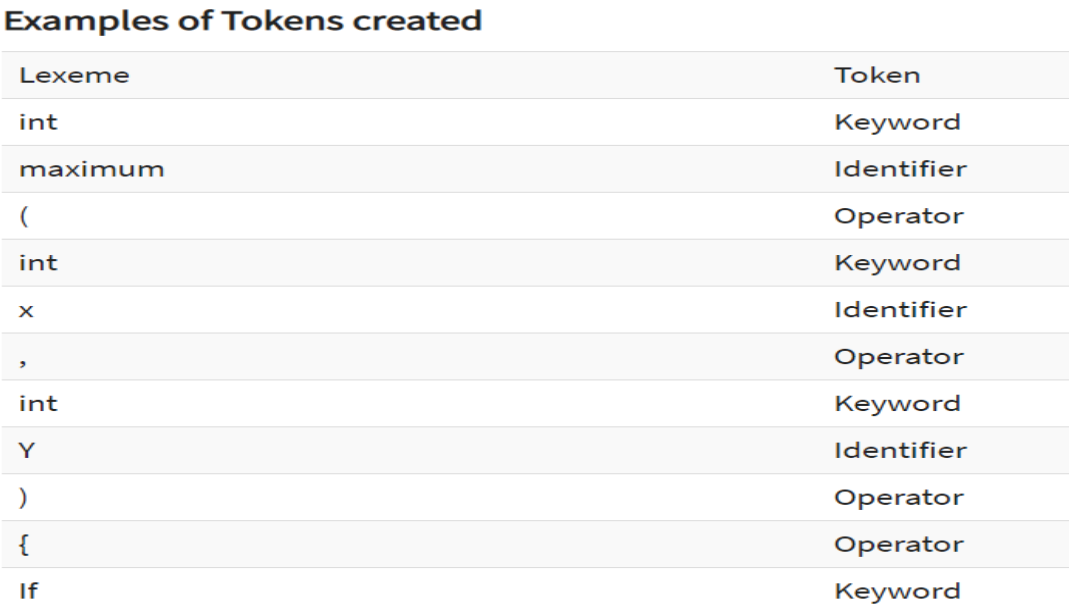
- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

Example of Lexical Analysis, Tokens, Non-Tokens

Consider the following code that is fed to Lexical Analyzer

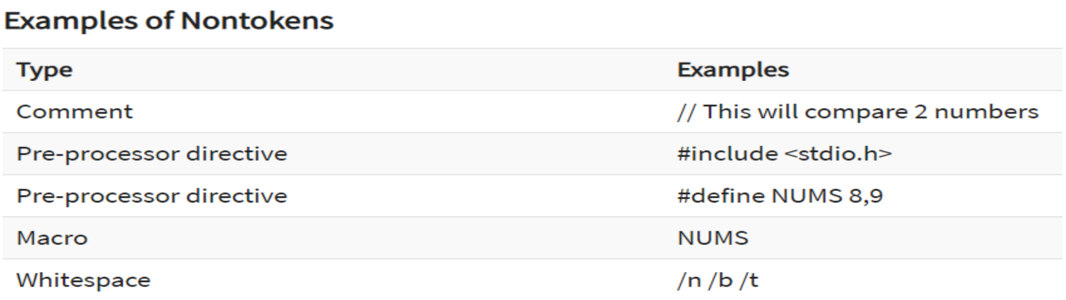


```
#include <stdio.h>
    int maximum(int x, int y) {
        // This will compare 2 numbers
        if (x > y)
            return x;
        else {
            return y;
        }
    }
```



Examples of Tokens created

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword



Examples of Nontokens

Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

Lexical Errors:

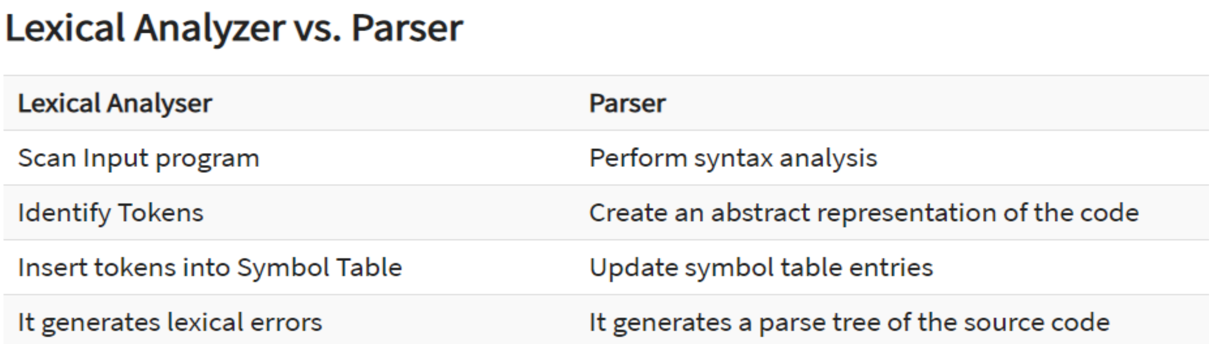
A character sequence which is not possible to scan into any valid token is a lexical error. Important facts about the lexical error:

- Lexical errors are not very common, but it should be managed by a scanner
- Misspelling of identifiers, operators, keyword are considered as lexical errors
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery in Lexical Analyzer:

Here, are a few most common error recovery techniques:

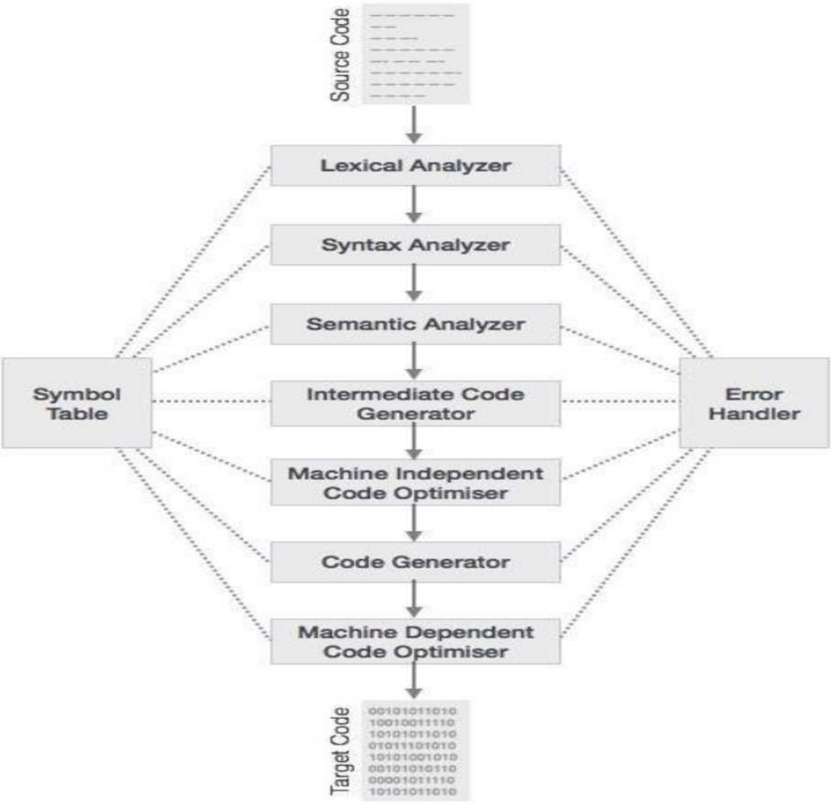
- Removes one character from the remaining input
- In the panic mode, the successive characters are always ignored until we reach a well-formed token
- By inserting the missing character into the remaining input
- Replace a character with another character
- Transpose two serial characters



Lexical Analyzer vs. Parser

Lexical Analyser	Parser
Scan Input program	Perform syntax analysis
Identify Tokens	Create an abstract representation of the code
Insert tokens into Symbol Table	Update symbol table entries
It generates lexical errors	It generates a parse tree of the source code

Secondary **phases of** compiler:



Symbol Table:

A symbol table contains a record for each identifier with fields for the attributes of the identifier. This component makes it easier for the compiler to search the identifier record and retrieve it quickly. The symbol table also helps you for the scope management. The symbol table and error handler interact with all the phases and symbol table update correspondingly.

Error Handling Routine:

In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: Unreachable statements
- Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis.

The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process. These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

Advantages of Lexical analysis:

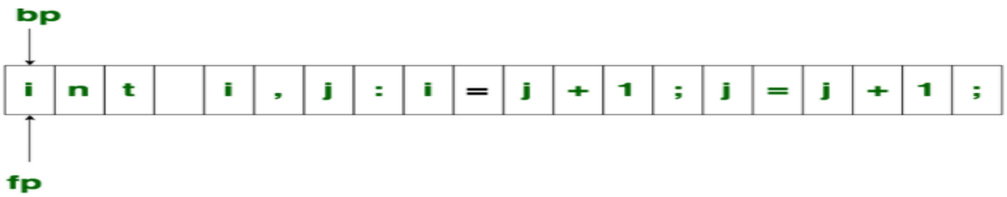
- Lexical analyzer method is used by programs like compilers which can use the parsed data from a programmer's code to create a compiled binary executable code
- It is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
- A separate lexical analyzer helps you to construct a specialized and potentially more efficient processor for the task

Disadvantage of Lexical analysis:

- You need to spend significant time reading the source program and partitioning it in the form of tokens
- Some regular expressions are quite difficult to understand compared to PEG or EBNF rules
- More effort is needed to develop and debug the lexer and its token descriptions
- Additional runtime overhead is required to generate the lexer tables and construct the tokens

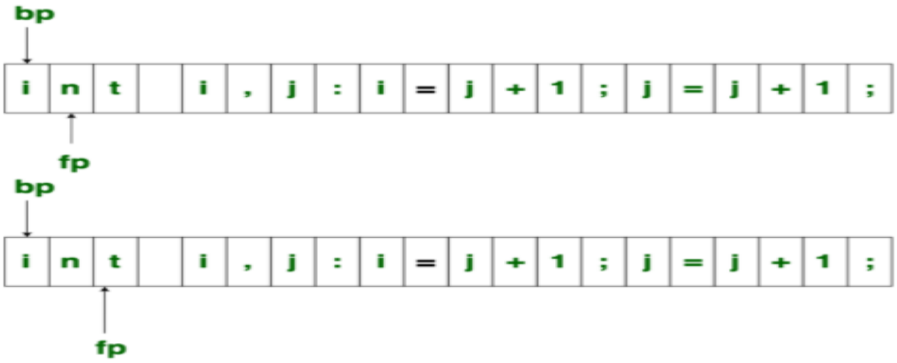
Input Buffering in Compiler Design:

- The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(bp) and forward to keep track of the pointer of the input scanned.



Initial Configuration

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

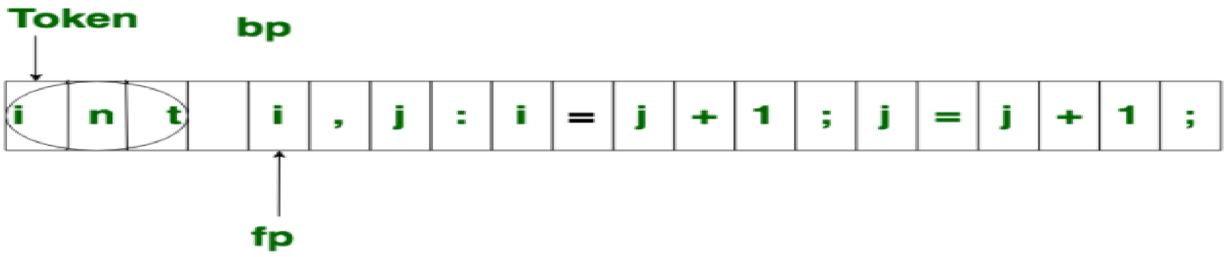

bp
i n t   i , j : i = j + 1 ; j = j + 1 ;
fp
bp
i n t   i , j : i = j + 1 ; j = j + 1 ;
fp

Input Buffering

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.

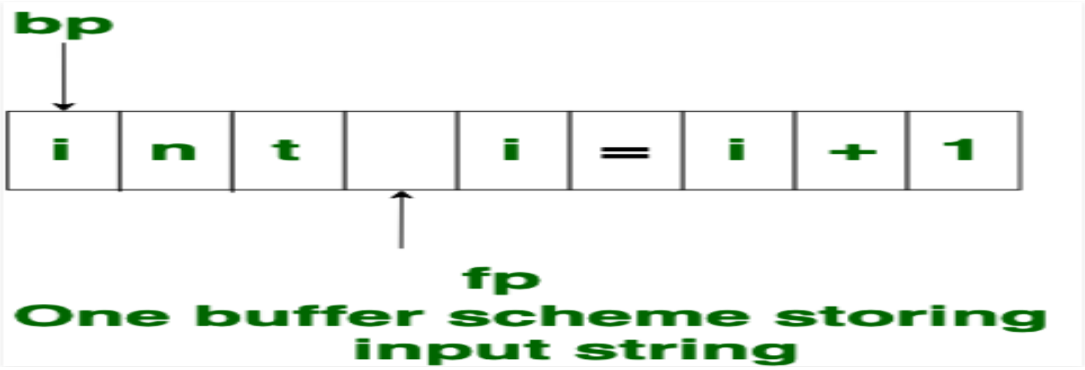
The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.

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.



Input buffering

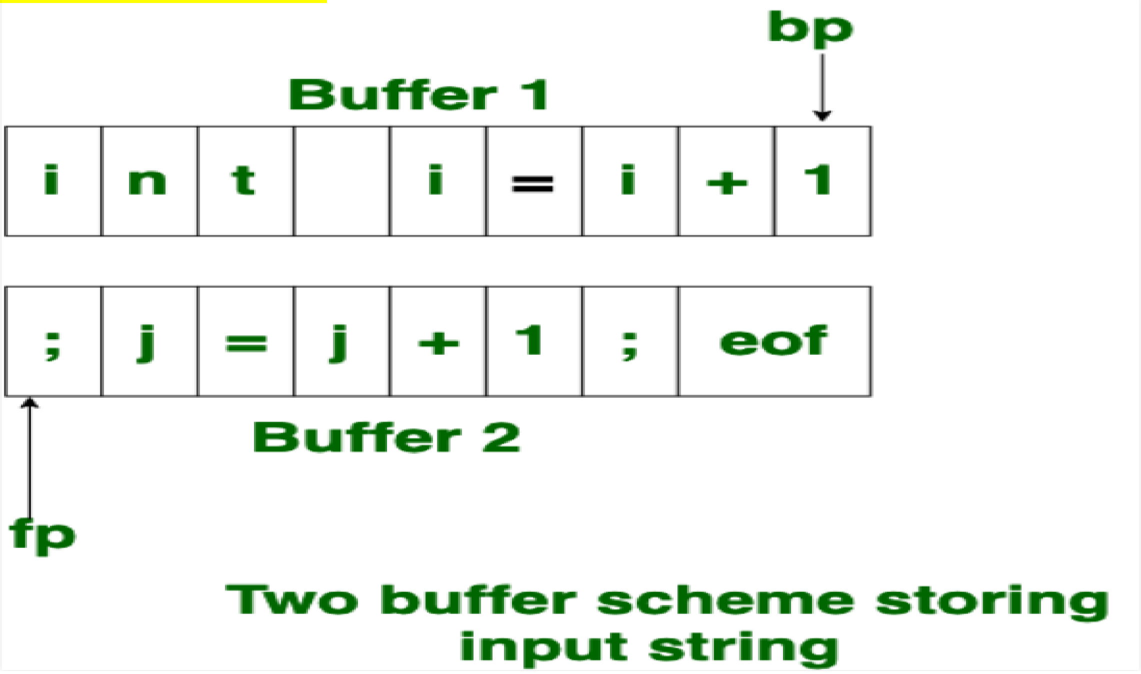
One Buffer Scheme:
In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



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 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.

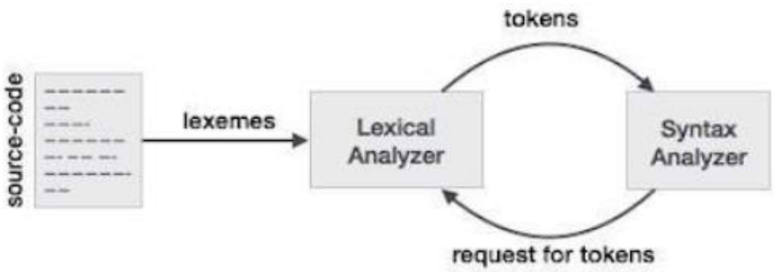


SPECIFICATION AND RECOGNITION OF TOKENS:

Lexical Analysis:

1. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

2. If the lexical analyzer finds a token invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens:

1. Lexemes are said to be a sequence of characters (alphanumeric) in a token.

2. There are some predefined rules for every lexeme to be identified as a valid token.

3. These rules are defined by grammar rules, by means of a pattern.

4. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

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

SPECIFICATION OF TOKENS:

There are 3 specifications of tokens:

1)Strings 2) Language 3)Regular expression

Strings and Languages

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. 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.
2. 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.
3. A substring of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.
4. 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.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s
6. For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The various operations on languages are:

· Union of two languages L and M is written as

L U M = {s | s is in L or s is in M}

- Concatenation of two languages L and M is written as

LM = {st | s is in L and t is in M}

- The Kleene Closure of a language L is written as

L* = Zero or more occurrence of language L.

Notations:

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Representing valid tokens of a language in regular expression:

If x is a regular expression, then:

- x* means zero or more occurrence of x.
- x+ means one or more occurrence of x.
- x? means at most one occurrence of x

Representing occurrence of symbols using regular expressions:

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions:

Decimal = $(\text{sign})^{?}(\text{digit})^{+}$

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Finite Automata:

1. Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.

2. If the input string is successfully processed and the automata reaches its final state, it is accepted.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

· **States**: States of FA are represented by circles. State names are written inside circles.

· **Start state**: The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.

· **Intermediate states**: All intermediate states have at least two arrows; one pointing to and another pointing out from them.

· **Final state**: If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles.

· **Transition**: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state.