DIFFERENCE BETWEEN COMPILER AND INTERPRETER

1. A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
2. Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
3. List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
4. An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
5. The compiler produces object code whereas interpreter does not produce object code.
6. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed hence interpreter is less efficient than compiler.

Examples of interpreter: A UPS Debugger is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.

Examples of compiler: Borland c compiler or Turbo C compiler compiles the programs written in C or C++.

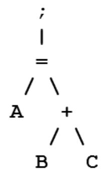
The Structure of a Compiler

Analysis determines the operations implied by the source program which are recorded in a tree structure. **Synthesis** takes the tree structure and translates the operations therein into the target program

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end. If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.

The Phases of a Compiler

Phase	Output	Sample
Programmer	**Source string**	`A=B+C;`
Scanner **(performs** ***lexical analysis*****)**	**Token string**	`'A', '=', 'B', '+', 'C', ';'` **And** ***symbol table*** **for identifiers**
Parser **(performs** ***syntax analysis*** **based on the grammar of the programming language)**	**Parse tree or abstract syntax tree**	
Semantic analyzer **(type checking, etc)**	**Parse tree or abstract syntax tree**	

Intermediate code generator	**Three-address code, quads, or RTL**	`int2fp B        t1`
		`+      t1   C   t2`
		`:=     t2       A`

Optimizer	**Three-address code, quads, or RTL**	`int2fp B        t1`
		`+      t1  #2.3  A`

Code generator	**Assembly code**	`MOVF  #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF  r2,A`

Peephole optimizer	**Assembly code**	`ADDF2 #2.3,r2`
		`MOVF  r2,A`

LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.

A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

- Scanning
- Tokenization

Token, Pattern and Lexeme

Token: Token is a valid sequence of characters which are given by lexeme. In a programming language,

- keywords,
- constant,

- identifiers,
- numbers,
- operators and
- punctuations symbols are possible tokens to be identified.

Pattern: Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.

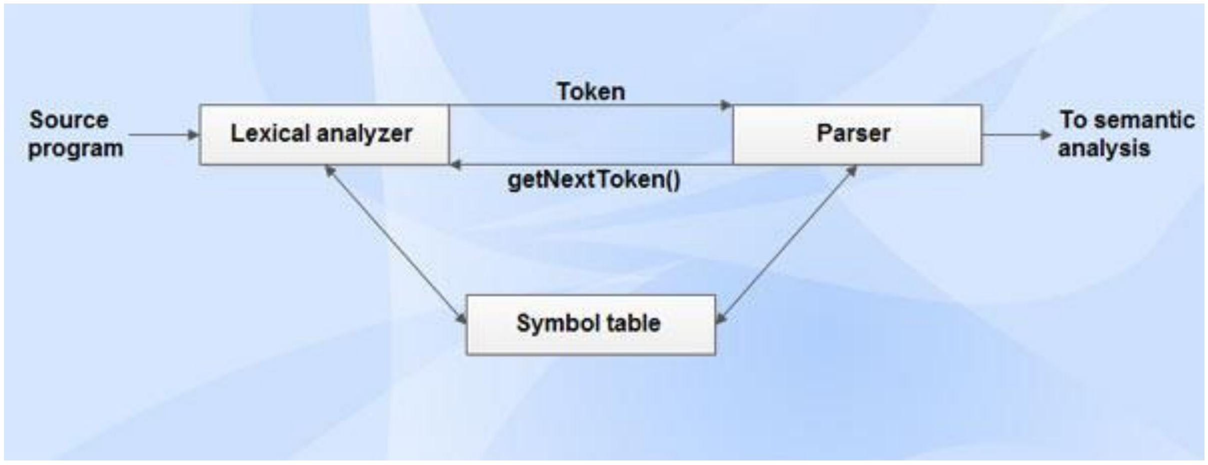
Lexeme: Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token. (eg.) c=a+b*5;

Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of Lexical Analyzer



Lexical analyzer performs the following tasks:

- Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
- Enters the identified token into the symbol table.
- Strips out white spaces and comments from source program.
- Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.

- Expands the macros if it is found in the source program.

Tasks of lexical analyzer can be divided into two processes:

> ***Scanning:*** Performs reading of input characters, removal of white spaces and comments.
>
> ***Lexical Analysis:*** Produce tokens as the output.

Need of Lexical Analyzer

Simplicity of design of compiler: The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.

Compiler efficiency is improved: Specialized buffering techniques for reading characters speed up the compiler process.

Compiler portability is enhanced

Issues in Lexical Analysis

Lexical analysis is the process of producing tokens from the source program. It has the following issues:

- Lookahead
- Ambiguities

Lookahead: *Lookahead* is required to decide when one token will end and the next token will begin. The simple example which has lookahead issues are i *vs. if,* = *vs.* ==. Therefore a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

- Is if it is two variables *i* and *f* or if?
- Is == is two equal signs =, = or ==?
- arr(5, 4) vs. fn(5, 4) *II* in Ada (as array reference syntax and function call syntax are similar.

Hence, the number of lookahead to be considered and a way to describe the lexemes of each token is also needed.

Regular expressions are one of the most popular ways of representing tokens.

Ambiguities: The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

- The longest match is preferred.
- Among rules which matched the same number of characters, the rule given first is preferred.

Lexical Errors

- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.
- Misspelling of identifiers, keyword, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery Schemes

- Panic mode recovery
- Local correction
  - ✓ Source text is changed around the error point in order to get a correct text.
  - ✓ Analyzer will be restarted with the resultant new text as input.
- Global correction
  - ✓ It is an enhanced panic mode recovery.
  - ✓ Preferred when local correction fails.

Panic mode recovery: In panic mode recovery, unmatched patterns are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

(eg.) For instance the string fi is encountered for the first time in a C program in the context:

fi (a== f(x))

A lexical analyzer cannot tell whether *f* iis a misspelling of the keyword if or an undeclared function identifier.

Since *f* i is a valid lexeme for the token **id,** the lexical analyzer will return the token **id** to the parser.

Local correction: Local correction performs deletion/insertion and/or replacement of any number of symbols in the error detection point.

(eg.) In Pascal, c[i] '='; the scanner deletes the first quote because it cannot legally follow the closing bracket and the parser replaces the resulting'=' by an assignment statement.

Most of the errors are corrected by local correction.

(eg.) The effects of lexical error recovery might well create a later syntax error, handled by the parser. Consider

· · · for $tnight · · ·

The $ terminates scanning of *for.* Since no valid token begins with $, it is deleted. Then *tnight* is scanned as an identifier.

In effect it results,

· · · fortnight · · ·

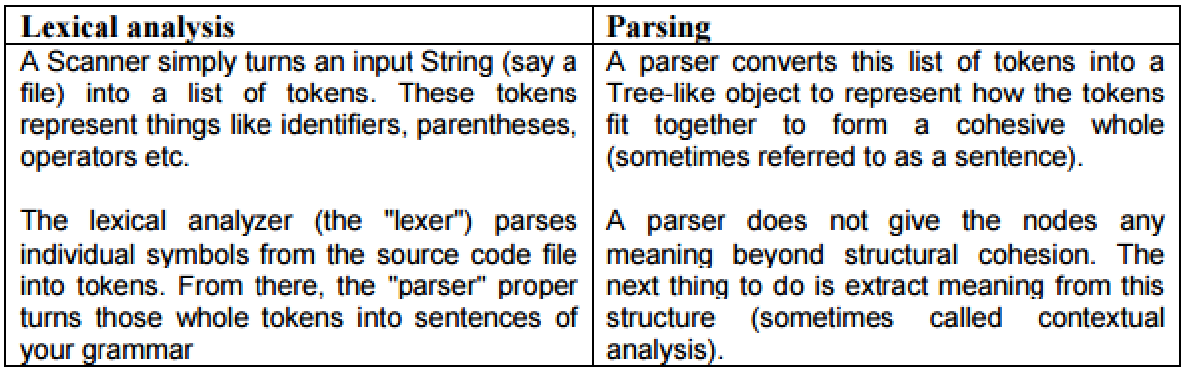
Which will cause a syntax error? Such *false errors* are unavoidable, though a syntactic error-repair may help.

Lexical error handling approaches

Lexical errors can be handled by the following actions:

- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

LEXICAL ANALYSIS VS PARSING:



Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).