Unit I – Part III

1. The structure of a compiler
2. *Phases of a compiler*
3. *Cousins of the Compiler*
4. *Grouping of Phases*
5. *Compiler construction tools*
6. *Lexical Analysis*
7. *Role of Lexical Analyzer*
8. *Input Buffering*

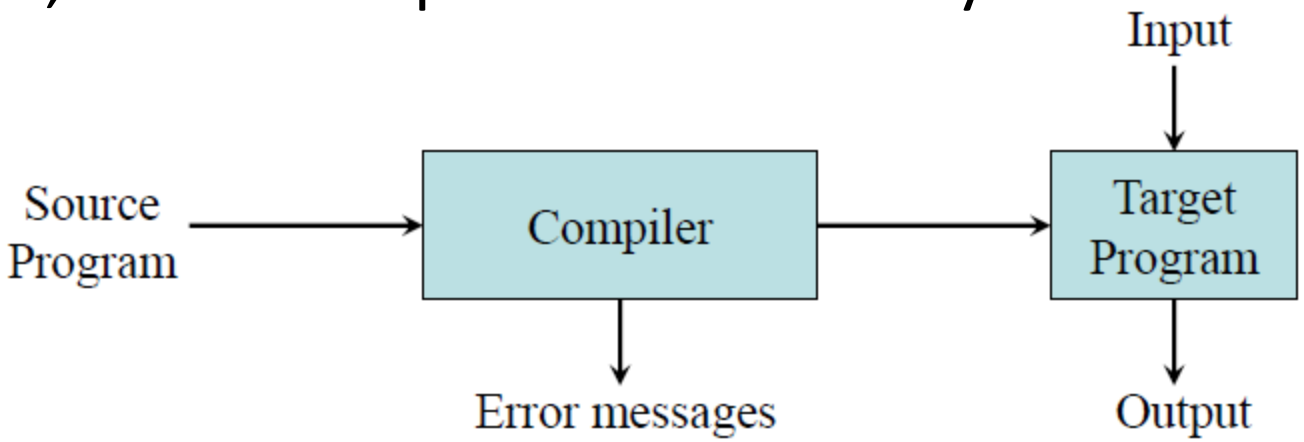
Why compilers?

- Software for early computers was primarily written in assembly language, and before that directly in machine code. It is usually more productive for a programmer to use a high-level language, and programs written in a high-level language can be reused on different kinds of computers.

- Because computer architecture is made up of electronic switches and cables that can only work with binary 1s and 0s, you need a compiler to translate your code from high level C++ to machine language that the CPU can understand.

Compilers

- A compiler is a translator that produces an output of low-level language (like an assembly or machine language) by taking an input of high-level language.
- The computer then processes the machine code for performing the corresponding tasks.
- Compilers check all types of errors, limits, and ranges. Thus, it's more intelligent.
- The run time of its program is longer, and it occupies more memory.



- Another way that compilers differ from one another is in the format of the target machine code they generate:

–Assembly or other source format

–Relocatable binary

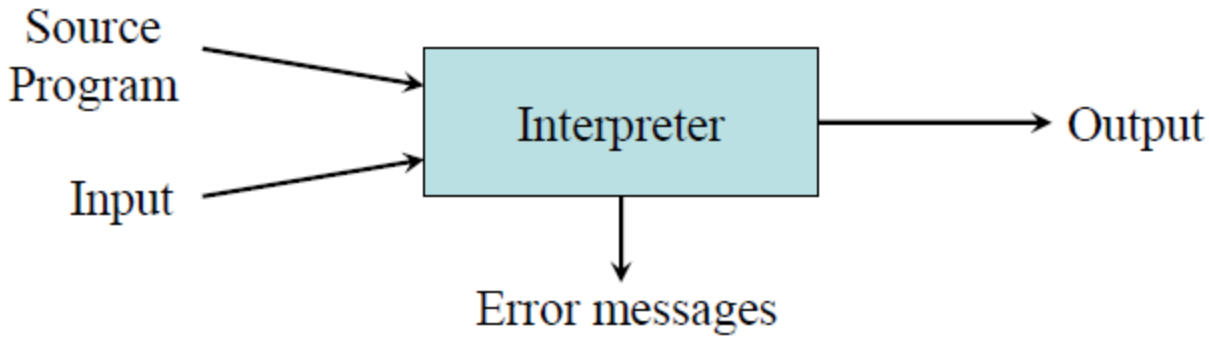
- Relative address (B+15)
- A linkage step is required

–Absolute binary

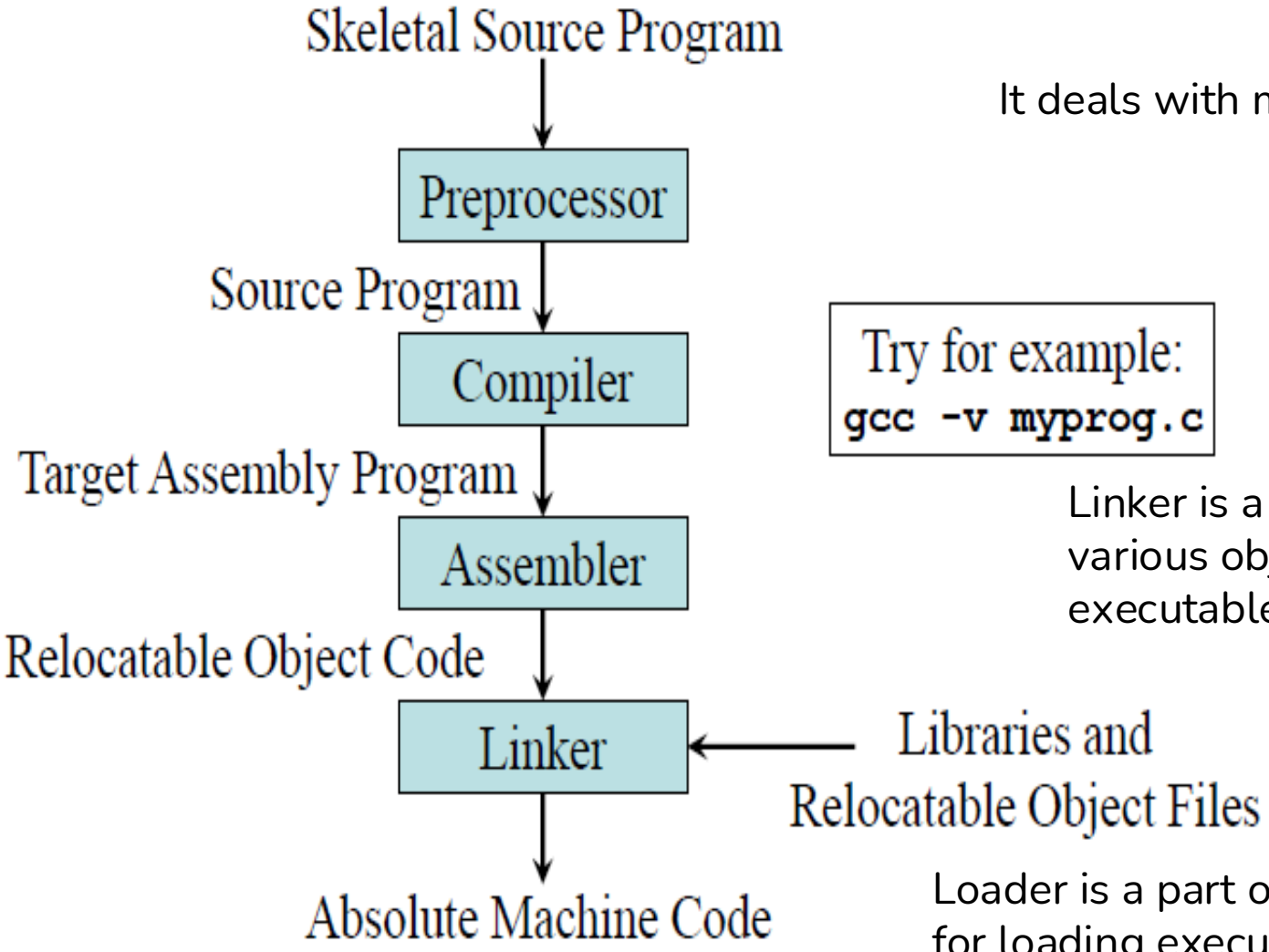
- Absolute address
- Can be executed directly

Interpreters

- It is a program that functions for the translation of a programming language into a comprehensible one. It is a computer program used for converting high-level program statements into machine codes. It includes pre-compiled code, source code, and scripts.
- An interpreter translates only one statement at a time of the program.
- They create an exe of the programming language before the program runs.



Preprocessors, Compilers, Assemblers, and Linkers



It deals with macro-processing, file inclusion etc.

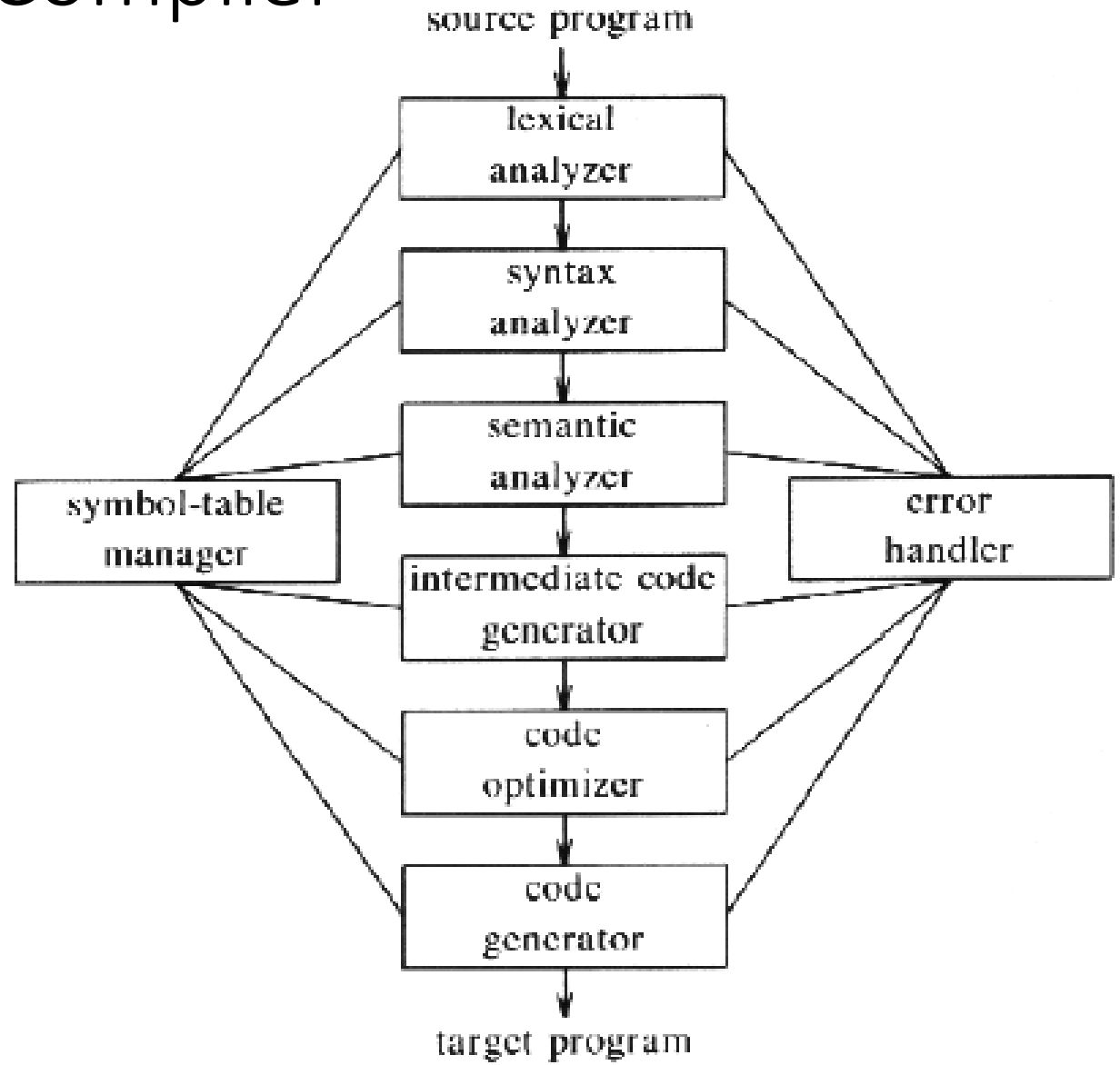
Try for example:

```
gcc -v myprog.c
```

Linker is a computer program that links and merges various object files together in order to make an executable file.

Loader is a part of operating system and is responsible for loading executable files into memory and execute them.

Phases of Compiler



There are two parts to compilation:

- ***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

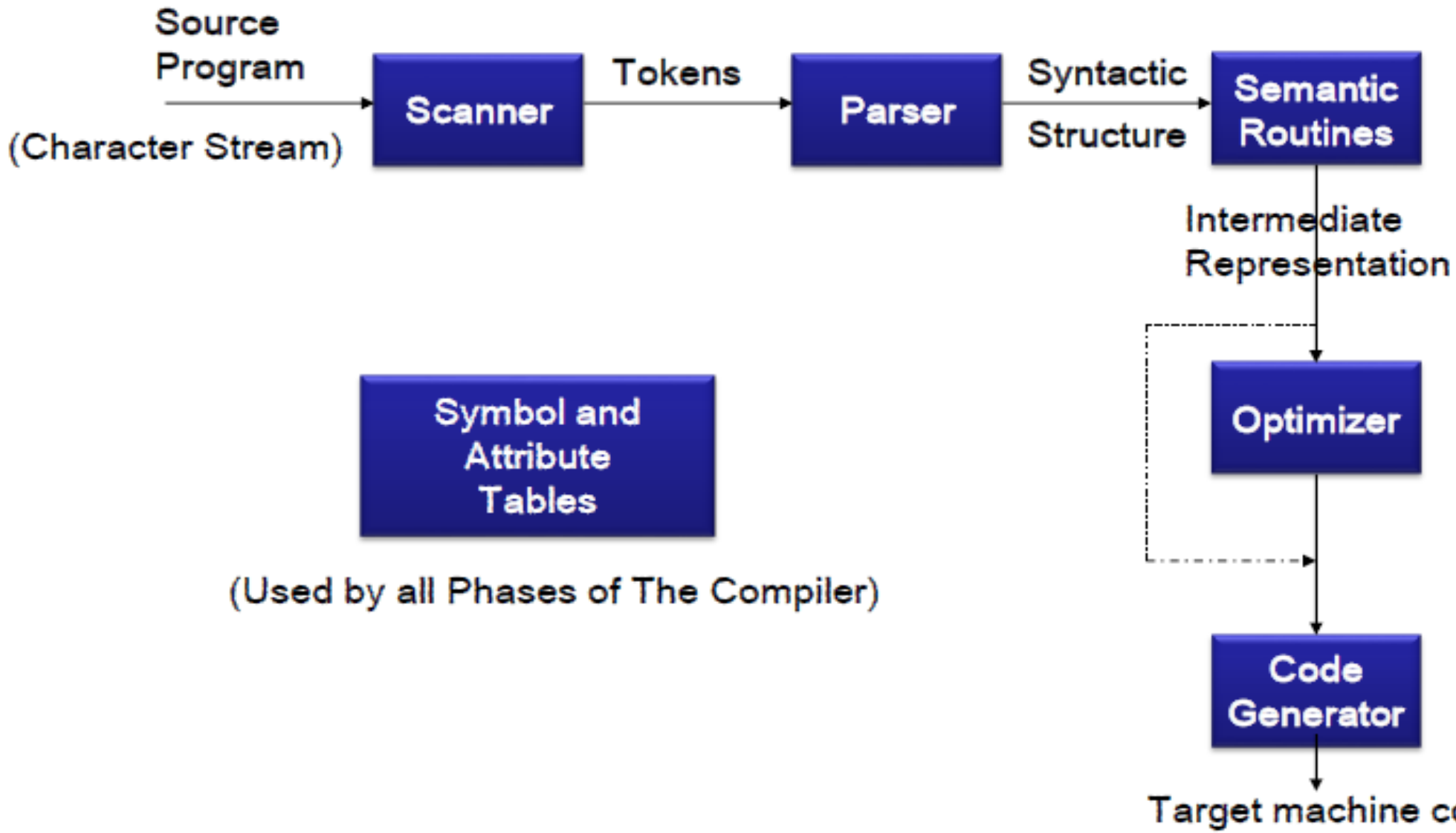
Grouping of Phases

- Generally, phases are divided into two parts:

1. Front End phases
2. Back End phases

1. **Front End phases:** The front end consists of those phases or parts of phases that are source language-dependent and target machine independents. These generally consist of **lexical analysis, semantic analysis, syntactic analysis, symbol table creation, and intermediate code generation**.

2. Back End phases: The portions of compilers that depend on the target machine and do not depend on the source language are included in the back end.


Source
Program
(Character Stream)
Scanner
Tokens
Parser
Syntactic
Structure
Semantic
Routines
Intermediate
Representation
Optimizer
Code
Generator
Target machine code
Symbol and
Attribute
Tables
(Used by all Phases of The Compiler)

Lexical Analyzer –

- It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language.
- It reads the characters from the 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 (e.g., erroneous characters), comments, and white space.

```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

All the valid tokens are:

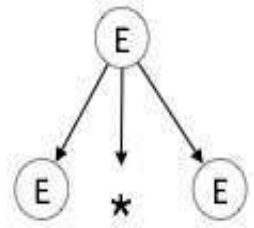
'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';'
'a' '=' '10' ';' 'return' '0' ';' '}'

Syntax Analyzer –

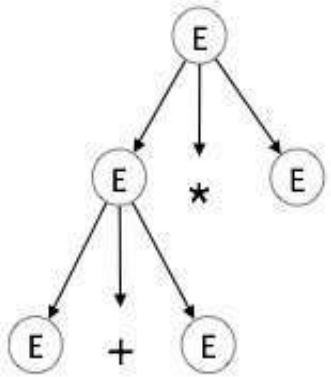
- It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.
- The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
- The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar.

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

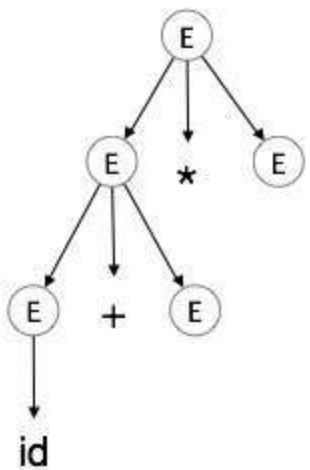
Step 1:

E → E * E	

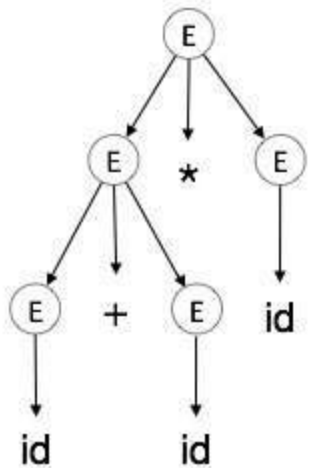
Step 2:

E → E + E * E	

Step 3:

E → id + E * E	

Step 5:

E → id + id * id	

- **Semantic Analyzer –** It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree.
- Type mismatch
- Reserved identifier misuse

Intermediate Code Generator – It generates intermediate code, which is a form that can be readily executed by a machine.

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

Till intermediate code, it is the 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.

- **Code Optimizer –** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered.

- **Target Code Generator –** The main purpose of the 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.

Error Detection and Reporting

- Syntax and semantic analysis handle a large fraction of errors

- Some of the Errors may occur in compilation phase

–Lexical phase: could not for many token

- e.g. **Misspelling** or Juxtaposing of characters

–Syntax phase: tokens violate structure rules

- e.g. **Unbalanced parenthesis, Missing punctuation operators or Undeclared variables**.

–Semantic phase: **no meaning of operations**

- Add an array name and a procedure name, Truncation of results or Unreachable Code.

Symbol Table

- **Symbol Table** is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- **Items stored in Symbol table:**
  - Variable names and constants
  - Procedure and function names
  - Literal constants and strings
  - Compiler generated temporaries
  - Labels in source languages

- It is used by various phases of the compiler as follows:-

  - **Lexical Analysis:** Creates new table entries in the table, for example like entries about tokens.
  - **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
  - **Semantic Analysis: Uses available information in the table to check for semantics** i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
  - **Intermediate Code generation: Refers symbol table for knowing how much and what type of run-time is allocated** and table helps in adding temporary variable information.
  - **Code Optimization: Uses information present in the symbol table for machine-dependent optimization**.
  - **Target Code generation: Generates code by using address information** of identifier present in the table.

Cousin's of the compiler

1. Preprocessors produce input to compilers. They may perform the following functions:

- ***Macro* processing:** A preprocessor may allow a user to define macros that are short hands for longer constructs.
- **File *inclusion:*** A preprocessor may include header files into the program text. For example, the C preprocessor causes the contents of the file <global. h> to replace the statement #include <global . h> when it processes a file containing this statement.
- **Rational preprocessors:** These processors **augment older languages with more modern flow-of-control** and data-structuring facilities. For example, such a preprocessor might provide the user with **built-in macros for constructs like while-statements or if-statements**, where none exist in the programming language itself.

2. **Assemblers:** Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses. A typical sequence of assembly instructions might be

```
MOV a, R1
ADD 2, R1
MOV R1, b
```

This code moves the contents of the address **a** into register **R1**, then adds the constant 2 to it, treating the contents of register **R1** as a fixed-point number and finally stores the result in the location named by b. Thus, it computes

b : = a + 2 .

3. Two pass assembly: The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once.

- In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table (separate from that of the compiler). **Identifiers are assigned storage locations as they are encountered for the first time**,
- In the second pass, the assembler scans the input again. This time, it **translates each operation code into the sequence of bits** representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table.

4. Loader and Linker: Usually, a program called a loader performs the two functions of loading and linking . The process of loading consists of taking **relocatable machine code, altering the relocatable addresses, and plating the altered instructions and data in memory at the proper locations**.

The linker allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

Compiler-construction tools

- Shortly after the first compilers were written, systems to help with the compiler-writing process appeared. These systems have often been referred to as *compiler-compilers, compiler-generators,* or translator-writing systems. Some compiler construction tools are:

1. Parser generators: These produce syntax analyzers, normally from input that is based on a context-free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler. This phase is now considered one of the easiest to implement**.**

***2. Scanner* generators:** These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.

3. Syntax-directed translation engines: These produce collections of routines that walk the parse tree, generating intermediate code. The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

4. Automatic *code generators:* Such a **tool takes a collection of rules** that define the translation of each operation of the **intermediate language into the machine language** for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data; e.g.. variables may be in registers, in a fixed (static) location in memory, or may be allocated a position on a stack. The basic technique is "template matching." The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template. Since there are usually many options regarding where variables are to be placed (eg.,in one of several registers or in memory), there are many possible ways to "tile" intermediate code with a given set of templates, and it is necessary to select a good tiling without a combinatorial explosion in running time of the compiler.

***5. Data flow* engines:** Much of the information needed to perform *good* code optimization involves "data-Row analysis," the gathering of information about how values are transmitted from *one* part of a program to each other part. Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate *code* statements and the information being gathered.

Lexical Analysis

- 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 (do, while, for, if, else)
- Constant (10, 20, 'a', 3.4, "c programming")
- Identifiers (int money, double accountBalance;)
- numbers,
- Operators (unary, binary, +, -, *, /, ++, --)
- Special characters in C ([], (), {}, ‘,’ , #)

are possible tokens to be identified.

- **Keywords:** Keywords are predefined, reserved words used in programming that have special meanings to the compiler.

auto	double	int	struct
break	else	long	switch
continue	for	signed	void
do	if	static	while

- **Identifiers:** Identifier refers to name given to entities such as variables, functions, structures etc.

```
int money;
double accountBalance;
```

- **Constants in C:** A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

- **Literals:** Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables.
- For example, const int a=10; is a constant integer expression in which 10 is an integer literal.

```
const int a=23;  // constant integer literal
```

- **Strings in C**: Strings in C are always represented as an array of characters having null character '\0' at the end of the string.

```
char a[10] = "javatpoint";
```

- **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;

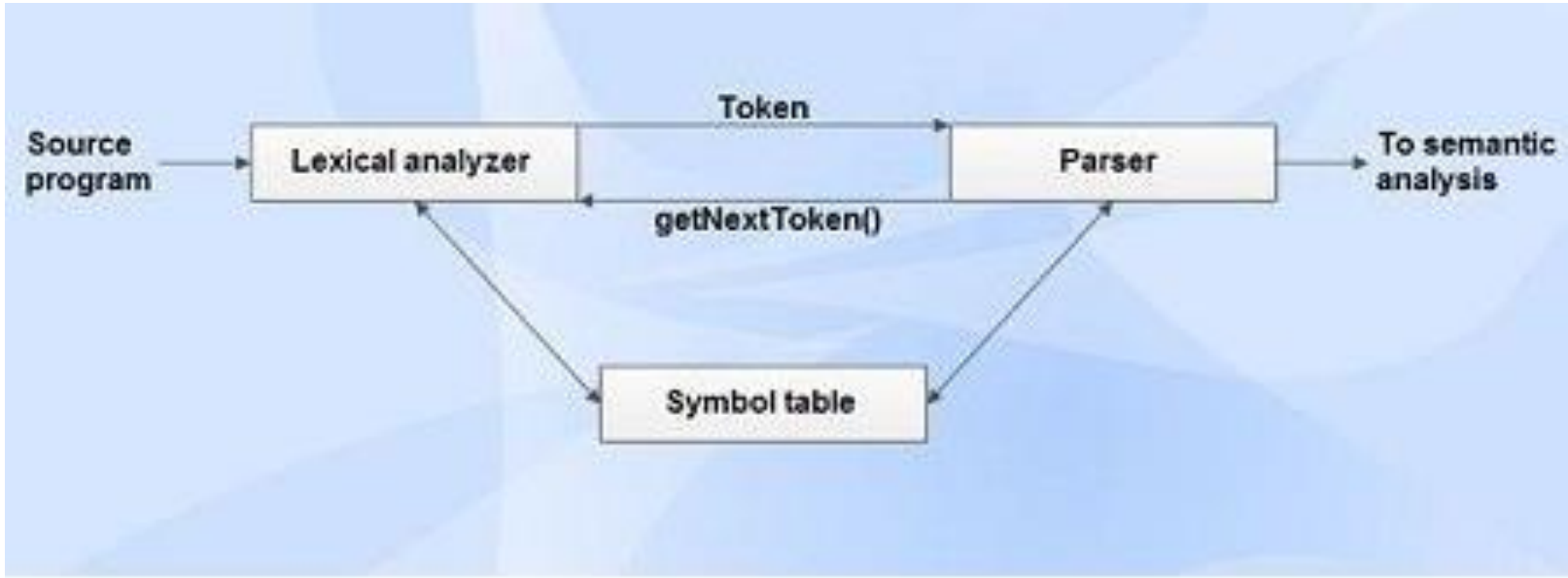
The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

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

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.

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.

Input Buffering

- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
- Hence a two-buffer scheme is introduced to handle large lookaheads safely.
- Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

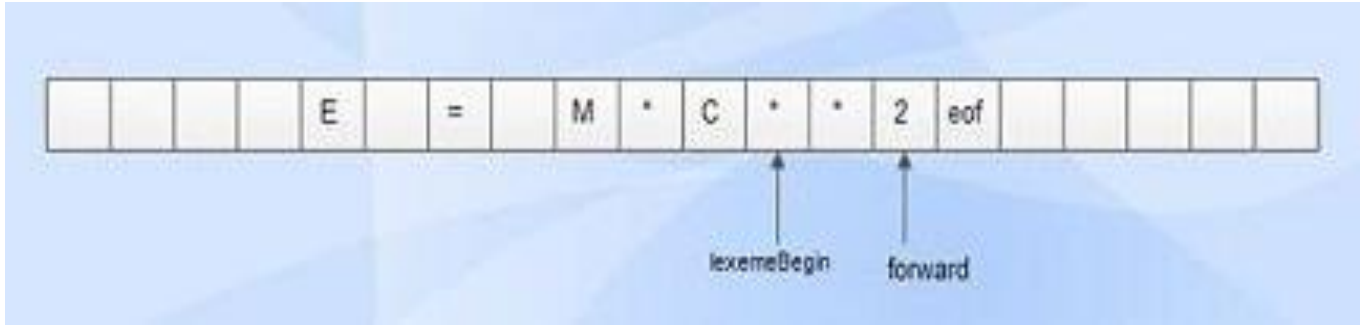
(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pairs

- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Scheme

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.
- *eof* is inserted at the end if the number of characters is less than N.



Pointers

- Two pointers *lexemeBegin* and *forward* are maintained.
- ***lexeme Begin*** points to the beginning of the current lexeme which is yet to be found.
- ***forward*** scans ahead until a match for a pattern is found.
- Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
- **(eg.)** DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;
- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

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.

Language: Set of strings

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 following example shows the operations on strings: Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : L^*={ ε,0,1,00....}
4. Positive closure : L^+={0,1,00....}

Regular Expressions: 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). Then,

a) (r)|(s) is a regular expression denoting the language L(r) U L(s).

b) (r)(s) is a regular expression denoting the language L(r)L(s).

c) (r)* is a regular expression denoting (L(r))*.

d) (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.