UNIT-I

INTRODUCTION TO LANGUAGE PROCESSING:

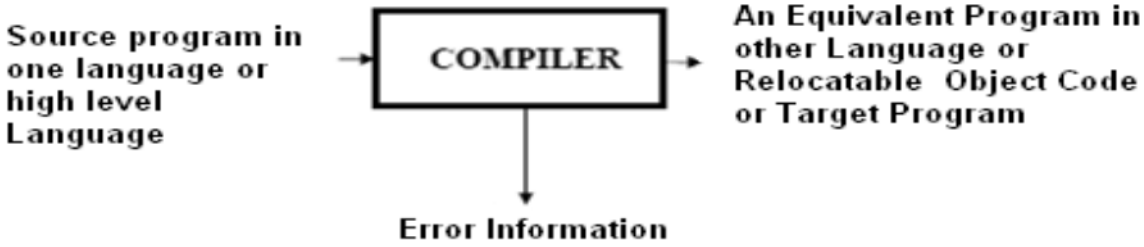
As Computers became inevitable and indigenous part of human life, and several languages with different and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software's have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGE TRANSLATORS :

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target]language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level language program.

∑ Two commonly Used Translators are Compiler and Interpreter

1. **Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



∑ If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

Figure1.1: Running the target Program

2. **Interpreter:** An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Figure 1.2: Running the target Program

LANGUAGE PROCESSING SYSTEM:

Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

Preprocessor: A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

- ∑ Collects all the modules, files in case if the source program is divided into different modules stored at different files.
- ∑ Expands short hands / macros into source language statements.

Compiler: Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

- ∑ Reports to its user the presence of errors in the source program.
- ∑ Facilitates the user in rectifying the errors, and execute the code.

Assembler: Is a program that takes as input an assembly language program and converts it into its equivalent machine language code.

Loader / Linker: This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

- ∑ **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.
- ∑ **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.

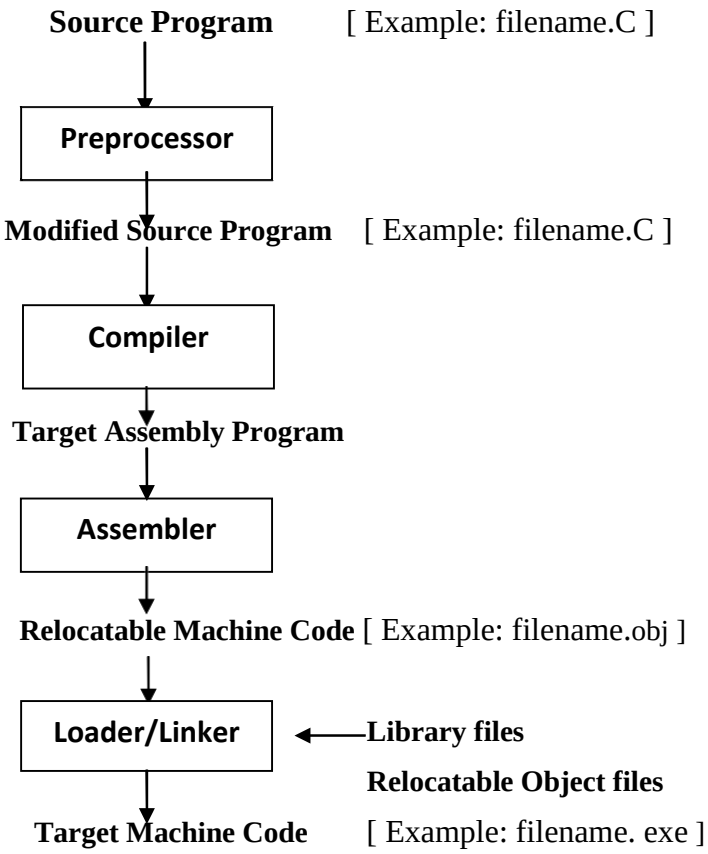


Figure1.3 : Context of a Compiler in Language Processing System

TYPES OF COMPILERS:

Based on the specific input it takes and the output it produces, the Compilers can be classified into the following types;

Traditional Compilers(C, C++, Pascal): These Compilers convert a source program in a HLL into its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code for another machine.

Incremental Compilers: These compilers separate the source into user defined–steps; Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into object code of another platform**.**

PHASES OF A COMPILER:

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions. Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)

Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. There fore a typical Compiler is having the following Phases:

1. Lexical Analyzer (Scanner), 2. Syntax Analyzer (Parser), 3.Semantic Analyzer, 4.Intermediate Code Generator(ICG), 5.Code Optimizer(CO) , and 6.Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some

cases. The description is given in next section.

The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

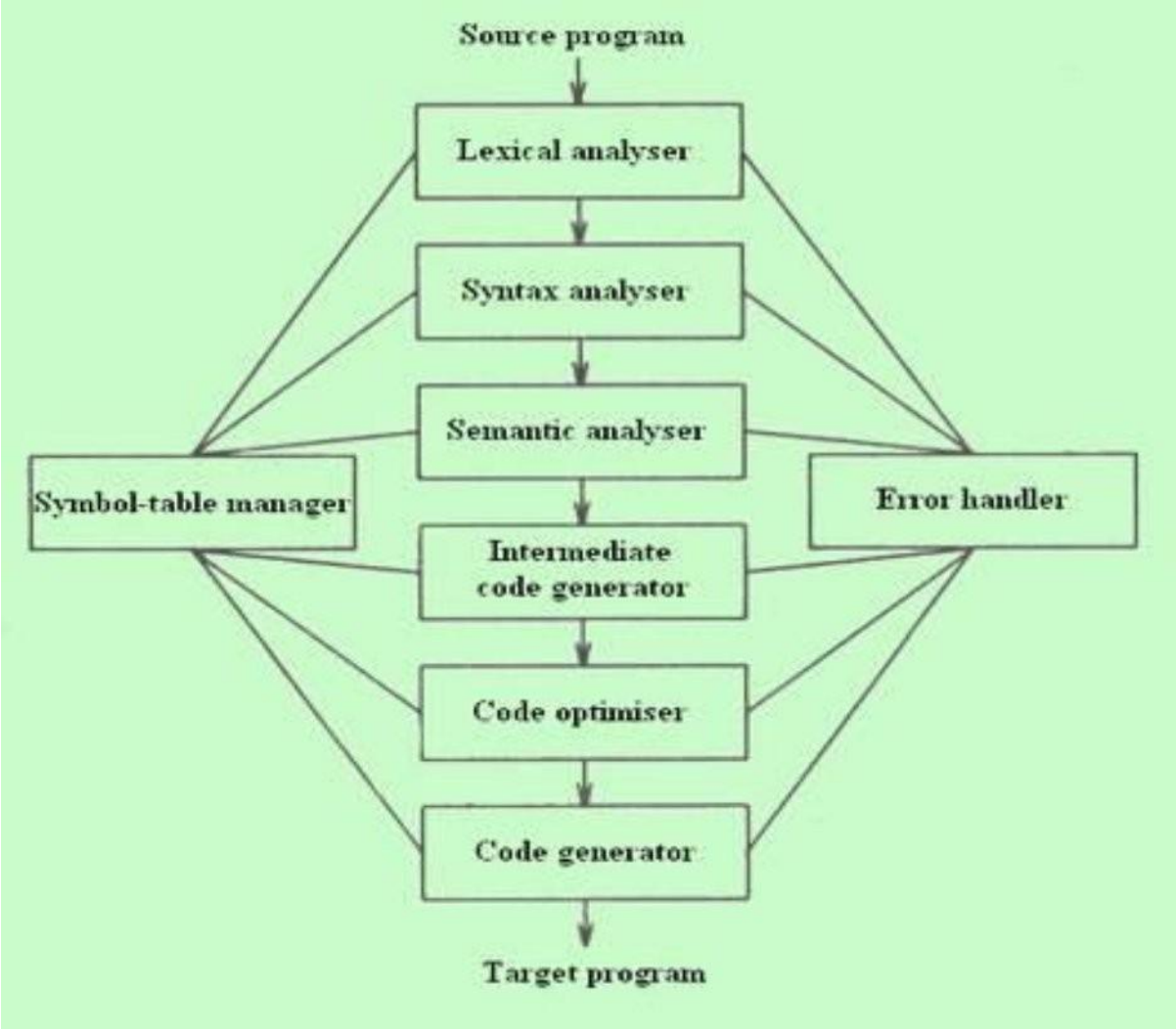


Figure1.4 : Phases of a Compiler

PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THE FRONT-END & BACK-END OF A COMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**, and **The Back-end**. This division is due to their dependence on either the Source Language or the Target machine. This model is called an Analysis & Synthesis model of a compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and those portions don't dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

- ∑ Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .
- ∑ The character sequence forming a token is called a **lexeme** of the token.
- ∑ The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn't exist.
- ∑ Also removes the Comments, and unnecessary spaces.

The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- ∑ Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves are tokens.
- ∑ The interior node of this tree represents the stream of tokens that logically belongs together.
- ∑ It means it checks the syntax of program elements.

SEMANTIC ANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it

may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

- ∑ The Syntactically and Semantically correct structures are produced here in the form of a Syntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- ∑ It should be easy to produce,and Easy to translate into the target program. Example intermediate code forms are:

- ∑ Three address codes,

- ∑ Polish notations, etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

- ∑ Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.

- ∑ Sometimes the data structures used in representing the intermediate forms may also be changed.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

- ∑ Memory locations are selected for each variable used, and assignment of variables to registers is done.

- ∑ Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

The input source program is **Position=initial+rate*60**

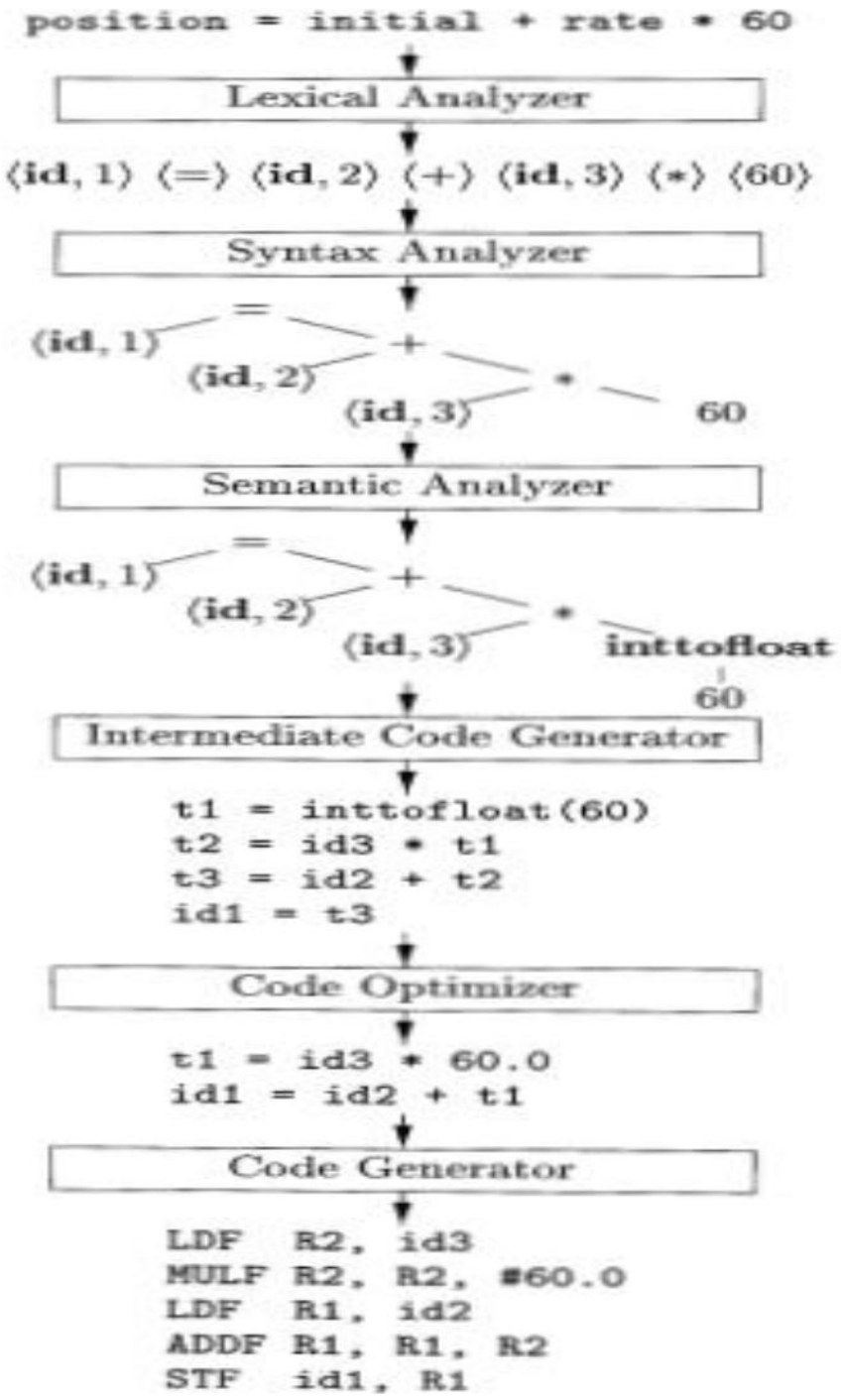


Figure1.5: Translation of an assignment Statement

LEXICAL ANALYSIS:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.

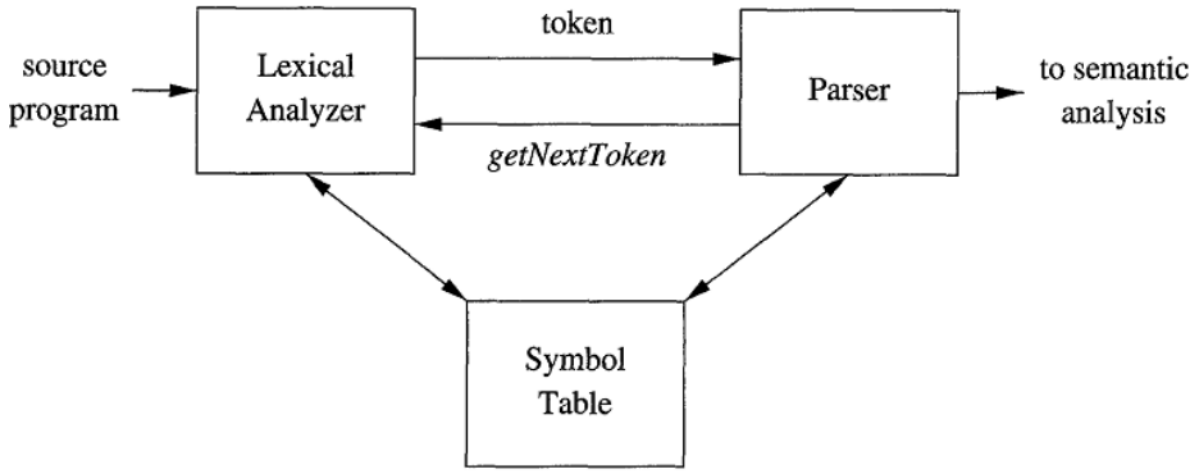


Figure 1.6 : Lexical Analyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example: In the following C language statement ,

printf ("Total = %d\n", score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n**" is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 1.7: Examples of Tokens

LEXICAL ANALYSIS Vs PARSING:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

- **1**. **Simplicity of design is the most important consideration.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

- **2. Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

- **3. Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

INPUT BUFFERING:

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

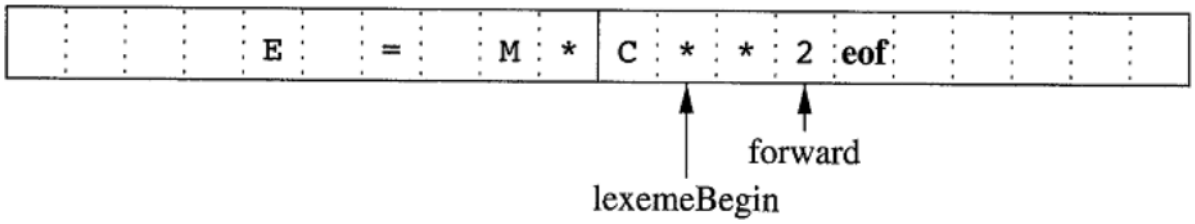


Figure1.8 : Using a Pair of Input Buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

∑ Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

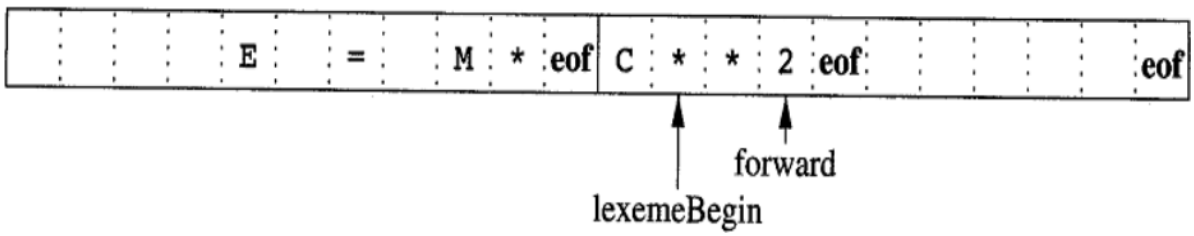


Figure1.8 : Sentential at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of

a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++ )
{
        case eof: if (forward is at end of first buffer )
                {
                  reload second buffer;
                  forward = beginning of second buffer;
                 }
                else if (forward is at end of second buffer )
                {
                  reload first buffer;
                  forward = beginning of first buffer;
                }
                  else    /* eof within a buffer marks the end of input */
                    terminate lexical analysis;
            break;
}
```

Figure 1.9: use of switch-case for the sentential

SPECIFICATION OF TOKENS:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

LEX the Lexical Analyzer generator

Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex .yy .c, it is a c program given for C Compiler, gives the Object code. Here we need to know how to write the Lex language. The structure of the Lex program is given below.

Structure of LEX Program : A Lex program has the following form:

Declarations
%%
Translation rules
%%
Auxiliary functions definitions

The declarations section : includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. It appears between %{. . .%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the form

Pattern {Action}

The auxiliary function definitions section includes the definitions of functions used to install identifiers and numbers in the Symbol tale.

LEX Program Example:

```
%{
/* definitions of manifest constants LT,LE,EQ,NE,GT,GE, IF,THEN, ELSE,ID, NUMBER,
RELOP */
%}
/* regular definitions */
delim          [ \t\n]
ws      {      delim}+
letter          [A-Za-z]
digit           [o-91
id              {letter} ({letter} | {digit})  *
number          {digit}+ (\ . {digit}+)? (E  [+-I]?{digit}+)?
%%
{ws}             {/* no action and no return */}
if               {return(1F) ; }
```

```
then        {return(THEN) ; }
else        {return(ELSE) ; }
(id)        {yylval = (int) installID(); return(1D);}
(number)    {yylval = (int) installNum() ; return(NUMBER) ; }
" < "       {yylval = LT; return(REL0P) ; )}
" <="       {yylval = LE; return(REL0P) ; }
"="         {yylval =  EQ  ; return(REL0P) ;  }
"<>"        {yylval = NE;  return(REL0P);}
"<"         {yylval =  GT;  return(REL0P);)}
"<="        {yylval  =  GE;  return(REL0P);}
%%
int installID0() {/* function to install the lexeme, whose first character is pointed to by yytext,
                    and whose length is yyleng, into the symbol table and return a  pointer
                    thereto */
int installNum() {/* similar to installID, but puts numerical constants into a separate table */}
```

Figure 1.10 : Lex Program for tokens common tokens

SYNTAX ANALYSIS (PARSER)

THE ROLE OF THE PARSER:

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the below Figure, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

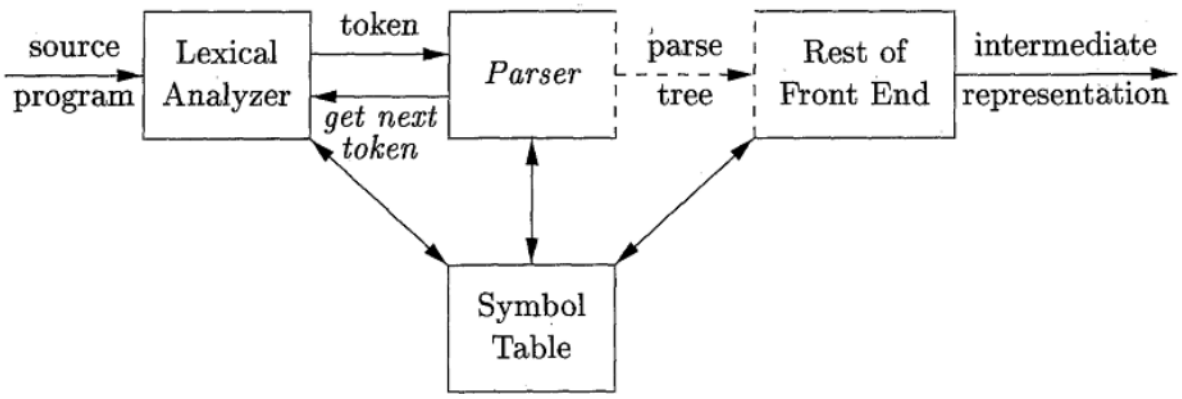


Figure2.1: Parser in the Compiler

During the process of parsing it may encounter some error and present the error information back to the user

Syntactic errors include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Based on the way/order the Parse Tree is constructed, **Parsing** is basically **classified** in to following two types:

1. **Top Down Parsing :** Parse tree construction start at the root node and moves to the children nodes (i.e., top down order).
2. **Bottom up Parsing:** Parse tree construction begins from the leaf nodes and proceeds towards the root node (called the bottom up order).

IMPORTANT (OR) EXPECTED QUESTIONS

1. What is a Compiler? Explain the working of a Compiler with your own example?
2. What is the Lexical analyzer? Discuss the Functions of Lexical Analyzer.
3. Write short notes on tokens, pattern and lexemes?
4. Write short notes on Input buffering scheme? How do you change the basic input buffering algorithm to achieve better performance?
5. What do you mean by a Lexical analyzer generator? Explain LEX tool.

UNIT-II

TOP DOWN PARSING:

- ∑ Top-down parsing can be viewed as the problem of constructing a parse tree for the given input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first left to right).
- ∑ Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

It is classified in to two different variants namely; one which uses Back Tracking and the other is Non Back Tracking in nature.

Non Back Tracking Parsing: There are two variants of this parser as given below.

1. **Table Driven Predictive Parsing :**
   i. LL (1) Parsing
2. **Recursive Descent parsing**

Back Tracking

1. Brute Force method

NON BACK TRACKING:

LL (1) Parsing or Predictive Parsing

LL (1) stands for, left to right scan of input, uses a Left most derivation, and the parser takes 1 symbol as the look ahead symbol from the input in taking parsing action decision.

A non recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

The table-driven parser in the figure has

- ∑ An input buffer that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.
- ∑ A stack, containing a sequence of grammar symbols with a $ at the bottom of the stack, which initially contains the start symbol of the grammar on top of $.
- ∑ A parsing table containing the production rules to be applied. This is a two dimensional array M [Non terminal, Terminal].
- ∑ A parsing Algorithm that takes input String and determines if it is conformant to Grammar and it uses the parsing table and stack to take such decision.

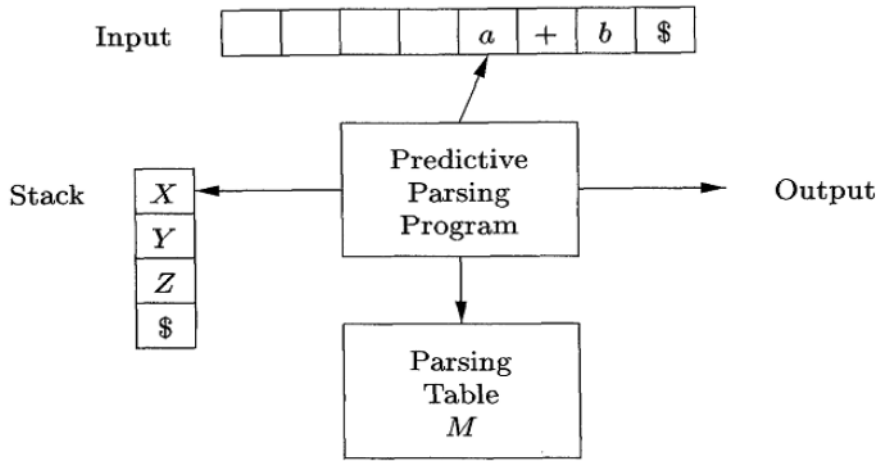


Figure 2.2: Model for table driven parsing

The Steps Involved In constructing an LL(1) Parser are:

1. Write the Context Free grammar for given input String
2. Check for Ambiguity. If ambiguous remove ambiguity from the grammar
3. Check for Left Recursion. Remove left recursion if it exists.
4. Check For Left Factoring. Perform left factoring if it contains common prefixes in more than one alternates.
5. Compute FIRST and FOLLOW sets
6. Construct LL(1) Table
7. Using LL(1) Algorithm generate Parse tree as the Output

Context Free Grammar (CFG): CFG used to describe or denote the syntax of the programming language constructs. The CFG is denoted as G, and defined using a four tuple notation.

Let G be CFG, then G is written as, G= (V, T, P, S)

Where

- ∑ V is a finite set of Non terminal; Non terminals are syntactic variables that denote sets of strings. The sets of strings denoted by non terminals help define the language generated by the grammar. Non terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

- ∑ T is a Finite set of Terminal; Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "'terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer.

- ∑ S is the Starting Symbol of the grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. P is finite set of Productions; the productions of a grammar specify the manner in which the

terminals and non terminals can be combined to form strings, each production is in α->β form, where α is a single non terminal, β is (VUT)*.Each production consists of:

(a) A non terminal called the head or left side of the production; this production defines some of the strings denoted by the head.

(b) The symbol ->. Some times: = has been used in place of the arrow.

(c) A body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non terminal at the head can be constructed.

∑ Conventionally, the productions for the start symbol are listed first.

Example: Context Free Grammar to accept Arithmetic expressions.

The terminals are **+, ***, **-**, **(,)**, **id**.

The **Non terminal symbols** are **expression**, **term**, **factor** and expression is the starting symbol.

expression → ***expression + term***
expression → ***expression – term***
expression → ***term***
term → ***term * factor***
term → ***term / factor***
term → ***factor***
factor → ***(expression)***
factor → ***id***

Figure 2.3 : Grammar for Simple Arithmetic Expressions

Notational Conventions Used In Writing CFGs:

To avoid always having to state that "these are the terminals," "these are the non terminals," and so on, the following notational conventions for grammars will be used throughout our discussions.

1. These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, e.

(b) Operator symbols such as **+**, ***,** and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1. . . 9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

2. These symbols are non terminals:

(a) Uppercase letters early in the alphabet, such as A, B, *C.*

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as *expr* or *stmt.*

(d) When discussing programming constructs, uppercase letters may be used to represent Non terminals for the constructs. For example, non terminal for expressions, terms, and factors are often represented by E, T, and F, respectively.

Using these conventions the grammar for the arithmetic expressions can be written as

E → E + T | E – T | T
T → T * F | T / F | F

F → (E) | id

DERIVATIONS:

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a Non terminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree as well as the bottom construction of the parse tree.

∑ Derivations are classified in to **Let most Derivation** and **Right Most Derivations.**

Left Most Derivation (LMD):

It is the process of constructing the parse tree or accepting the given input string, in which at every time we need to rewrite the production rule it is done with left most non terminal only.

Ex: - If the Grammar is E-> **E+E | E*E | -E| (E) | id** and the input string is **id + id* id**

The production **E -> - E** signifies that if E denotes an expression, then **–** E must also denote an expression. The replacement of a single E by - E will be described by writing

E => -E which is read as **"E derives _E"**

For a general definition of derivation, consider a non terminal A in the middle of a sequence of grammar symbols, as in αAβ, where α and β are arbitrary strings of grammar symbol. Suppose A ->γ is a production. Then, we write αAβ => αγβ. The symbol => means "derives in one step". Often, we wish to say, "Derives in zero or more steps." For this purpose, we can use the symbol $\overset{*}{\Rightarrow}$, If we wish to say, "Derives in $\overset{+}{\Rightarrow}$ one or more steps." We cn use the symbol $\overset{+}{\Rightarrow}$. If S $\overset{*}{\Rightarrow}$ a, where S is the start symbol of a grammar G, we say that α is a sentential form of G.

The Leftmost Derivation for the given input string **id + id* id** is

E => <u>E</u> +E

=> id + E

=> id + E * E

=> id + id * E

=> id + id * id

NOTE: Every time we need to start from the root production only, the under line using at Non terminal indicating that, it is the non terminal (left most one) we are choosing to rewrite the productions to accept the string.

Right Most Derivation (RMD):

It is the process of constructing the parse tree or accepting the given input string, every time we need to rewrite the production rule with Right most Non terminal only.

The Right most derivation for the given input string **id + id* id** is

E => E + E
=> E + E * E
=> E + E * id
=> E + id * id
=> id + id * id

NOTE: Every time we need to start from the root production only, the under line using at Non terminal indicating that, it is the non terminal (Right most one) we are choosing to rewrite the productions to accept the string.

What is a Parse Tree?

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals.

- ∑ Each interior node of a parse tree represents the application of a production.
- ∑ All the interior nodes are Non terminals and all the leaf nodes terminals.
- ∑ All the leaf nodes reading from the left to right will be the output of the parse tree.
- ∑ If a node n is labeled X and has children n1,n2,n3,…nk with labels X1,X2,…Xk respectively, then there must be a production A->X1X2…Xk in the grammar.

Example1:- Parse tree for the input string - **(id + id) using** the above Context free Grammar is

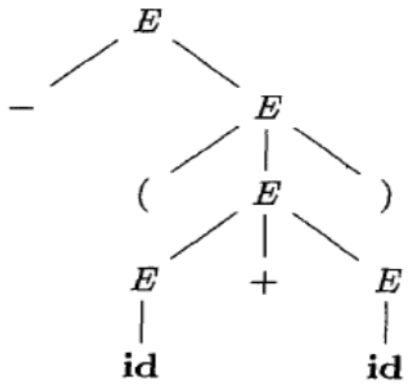


Figure 2.4 : Parse Tree for the input string - **(id + id)**

The Following figure shows step by step construction of parse tree using CFG for the parse tree for the input string - **(id + id).**

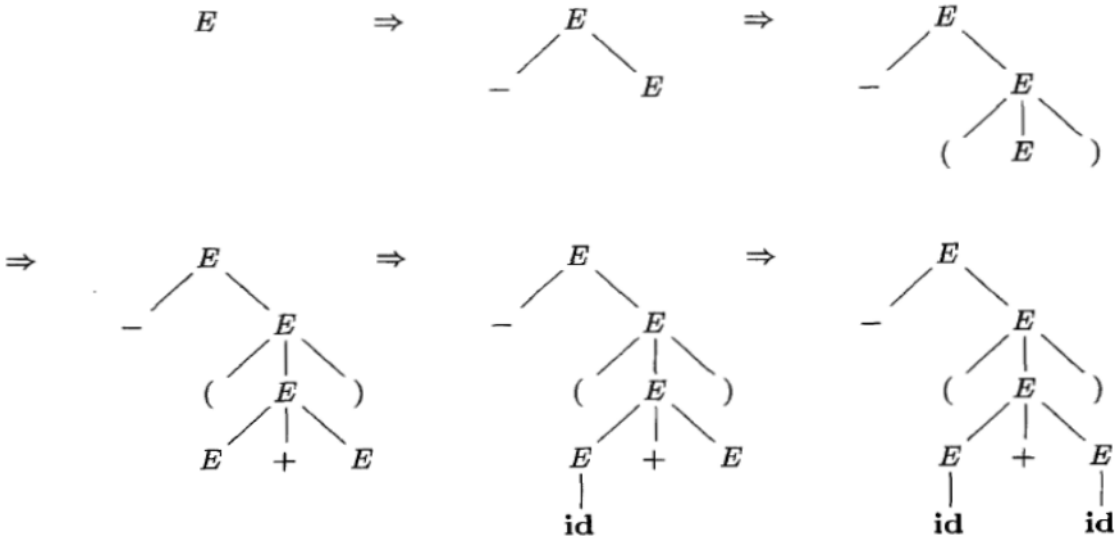


Figure 2.5 : Sequence outputs of the Parse Tree construction process for the input string **–(id+id)**

Example2:- Parse tree for the input string **id+id*id** using the above Context free Grammar is

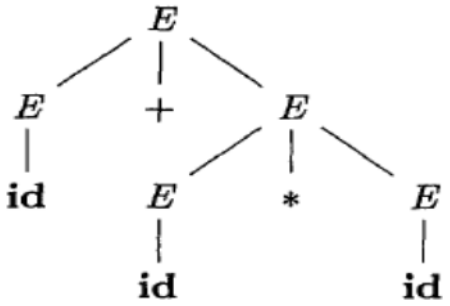


Figure 2.6: Parse tree for the input string id+ id*id

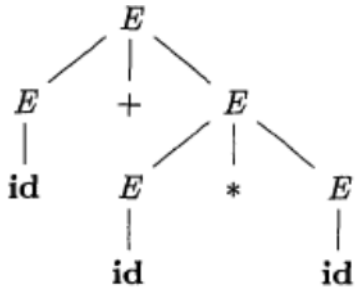
AMBIGUITY in CFGs:

Definition: A grammar that produces more than one parse tree for some sentence (input string) is said to be ambiguous.

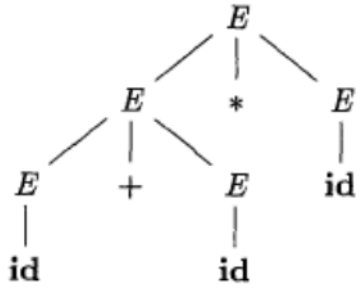
In other words, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Or If the right hand production of the grammar is having two non terminals which are exactly same as left hand side production Non terminal then it is said to an ambiguous grammar.
Example : If the Grammar is **E-> E+E | E*E | -E| (E) | id** and the Input String is **id + id* id**

Two parse trees for given input string are



(a)



(b)

Two Left most Derivations for given input String are :

E => E +E
=> id + E
=> id + E * E
=> id + id * E
=> id + id * id

(a)

E => E * E
=> E+E*E
=> id + E * E
=> id+ id* E
=> id + id * id

(b)

The above Grammar is giving two parse trees or two derivations for the given input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: Since Ambiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, **s**ometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

$$A \rightarrow A\alpha \mid \beta$$

This can be written as (introduce one new non terminal in the place of second non terminal)

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \alpha\, A' \mid \varepsilon$$

Example : Let the grammar is **E⟶ E+E | E*E | -E| (E) | id .** It is shown that it is ambiguous that can be written as

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

In the above grammar the 1st and 2nd productions are having ambiguity. So, they can be written as

E-> E+E | E*E this production again can be written as
E-> E+E | β , where β is E*E
The above production is same as the general form. so, that can be written as
E->E+T|T
T->β

The value of β is E*E so, above grammar can be written as

1) **E->E+T|T**
2) **T-> E*E** **The first production is free from ambiguity** and substitute E->T in the 2nd production then it can be written as
   T-> T*T | -E| (E) | **id** this production again can be written as
   T->T*T | β where β is -E| (E) | id, introduce new non terminal in the Right hand side production then it becomes
   T->T*F | F
   F-> -E | (E) | id now the entire grammar turned in to it equivalent unambiguous,

The Unambiguous grammar equivalent to the given ambiguous one is

1) **E → E + T | T**
2) **T → T * F | F**
3) **F → -E | (E) | id**

LEFT RECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation A=>Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

The General form of Left Recursion is

$$A \rightarrow A\alpha \mid \beta$$

The above left recursive production can be written as the non left recursive equivalent :

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid €$$

Example : - Is the following grammar left recursive? If so, find a non left recursive grammar equivalent to it.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow -E \mid (E) \mid id$$

Yes ,the grammar is left recursive due to the first two productions which are satisfying the general form of Left recursion, so they can be rewritten after removing left recursion from **E → E + T**, and **T→ T * F** is

$$E \rightarrow TE'$$
$$E' \rightarrow +T\ E' \mid €$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *F\ T' \mid €$$
$$F \rightarrow (E) \mid id$$

LEFT FACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.

For example, in the following grammar there are n A productions have the common prefix **α,** which should be removed or factored out without changing the language defined for A.

$$A \rightarrow \alpha A1 \mid \alpha A2 \mid \alpha A3 \mid \alpha A4 \mid \ldots \mid \alpha An$$

We can factor out the **α** from all n productions by adding a new A production $A \rightarrow \alpha A'$, and rewriting the A′ productions grammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1 \mid A2 \mid A3 \mid A4 \ldots \mid An$$

FIRST and FOLLOW: