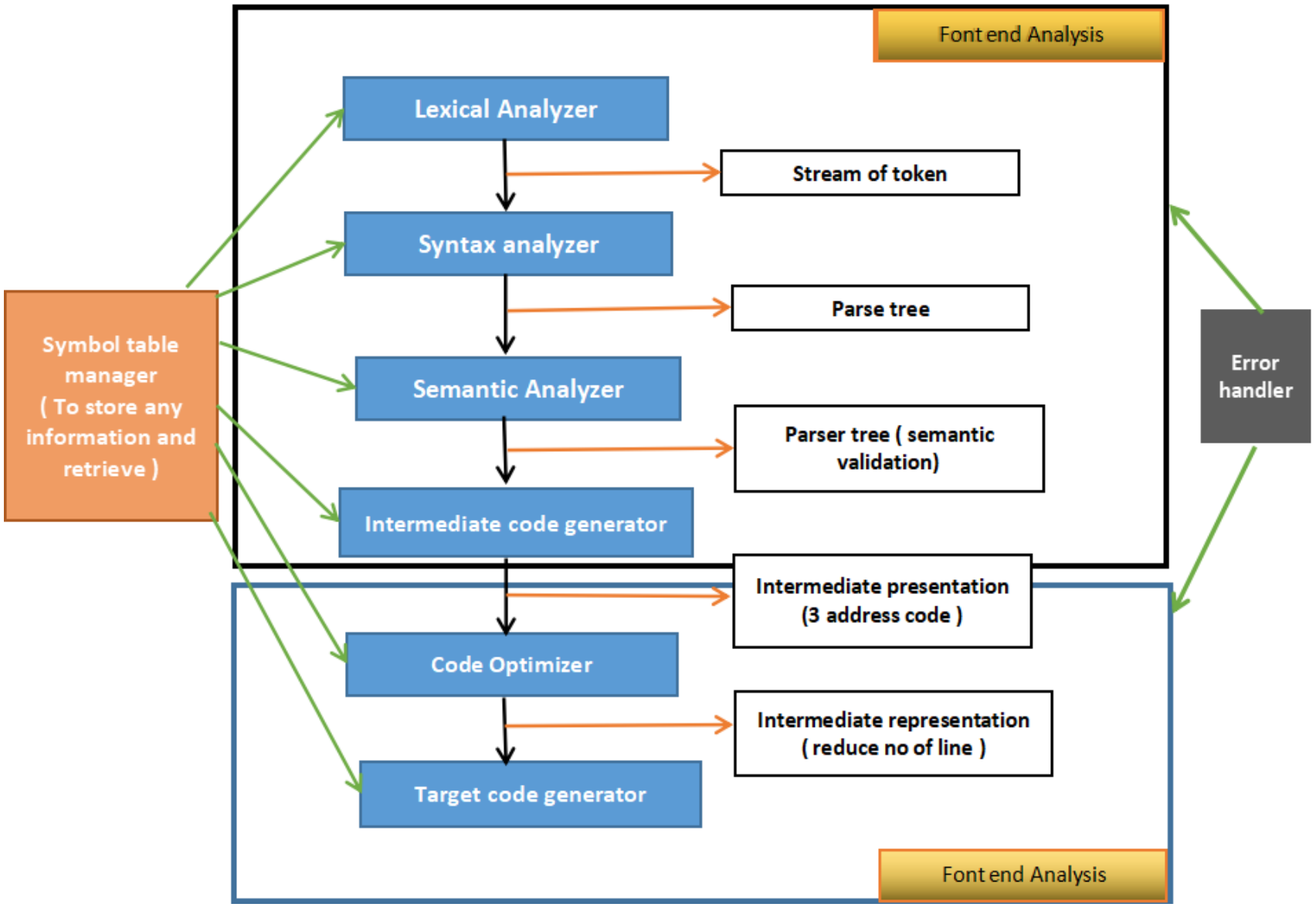
Compiler design

Phases of Compiler



The compilation of **compiler process** contains the sequence of various **phases**. Each phase takes source program in one representation and produces output in another representation. Each **phase** takes input from its previous stage.

Lexical analysis : It reads the **program** and converts it into **token** . It converts a stream of lexeme into stream of **token**. It takes source code as input. It reads the source program one character at a time and convert it into meaningful **lexeme**. Lexical analyzer represents these lexeme in the form of tokens. **Lexeme means sequence of character.**

Syntax Analyzer : These next phase is called the **syntax analysis** or **parsing**. It takes the token product by **lexical analysis** as input and generates a **parse tree** (or syntax tree). In syntax analysis phase , the parser check that the expression made by the token is syntactically correct or not.

It takes all the token one by one and uses context free grammar to construct the parse tree.

Semantic analyzer

- It **verifies the parse tree** , whether its meaningful or not. For example , assignment of values is between compatible data type, and adding string to integer, also the **semantic analyzer** keeps track of identifiers, their types and expressions , whether identifiers are declared before use or not etc .

  The **semantic analyzer** produces an interpret **syntax tree** as an output.

Intermediate code generator

- In the **intermediate code generator**, compiler generates the source code into the **intermediate** code **. Intermediate code** is generated between high level and the machine language . The intermediate code should be generated in such a way that the program can be easily translated it into the target code. popular intermediates codes- three address code.
- Intermediate code is converted to machine language using the last two phases which are platform dependent.

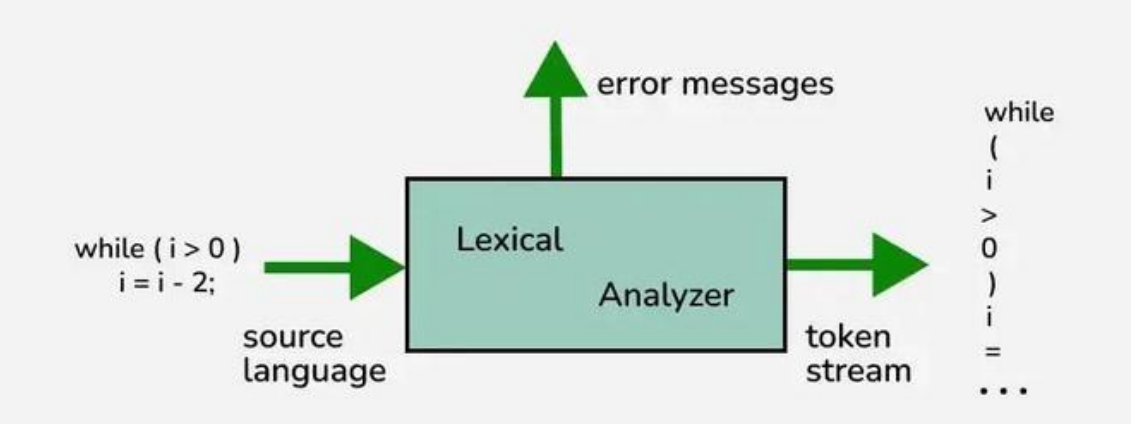
Code optimizer

- **Code optimizer** is an optional phase. It is used to improve i**ntermediate code** so that output of the program could run faster and take less space . It removes the unnecessary lines of the code and arrange the sequence of statements in order to speed up the program execution without wasting resources. It is platform/machine dependent or platform dependent.

Target code generator

- It is the final stage of the **compilation process**. It takes the optimizer intermediate code as input and write code that machine can understand. The code generator translates the **intermediate code** into sequence of relocatable machine code. It also machine dependent.

Introduction of Lexical Analysis



The lexical analyzer takes a source program as input, and produces a stream of tokens as output.

Categories of Tokens

- **Keywords:** In C programming, keywords are reserved words with specific meanings used to define the language's structure like if, else, for, and void. These cannot be used as variable names or identifiers, as doing so causes compilation errors. C programming has a total of 32 keywords.
- **Identifiers:** Identifiers in C are names for variables, functions, arrays, or other user-defined items. They must start with a letter or an underscore (_) and can include letters, digits, and underscores. C is case-sensitive, so uppercase and lowercase letters are different. Identifiers cannot be the same as keywords like if, else or for.
- **Constants:** Constants are fixed values that cannot change during a program's execution, also known as literals. In C, constants include types like integers, floating-point numbers, characters, and strings.
- **Operators:** Operators are symbols in C that perform actions on variables or other data items, called operands.
- **Special Symbols:** Special symbols in C are compiler tokens used for specific purposes, such as separating code elements or defining operations. Examples include **;** (semicolon) to end statements, **,** (comma) to separate values, **{}** (curly braces) for code blocks, and [] (square brackets) for arrays. These symbols play a crucial role in the program's structure and syntax.

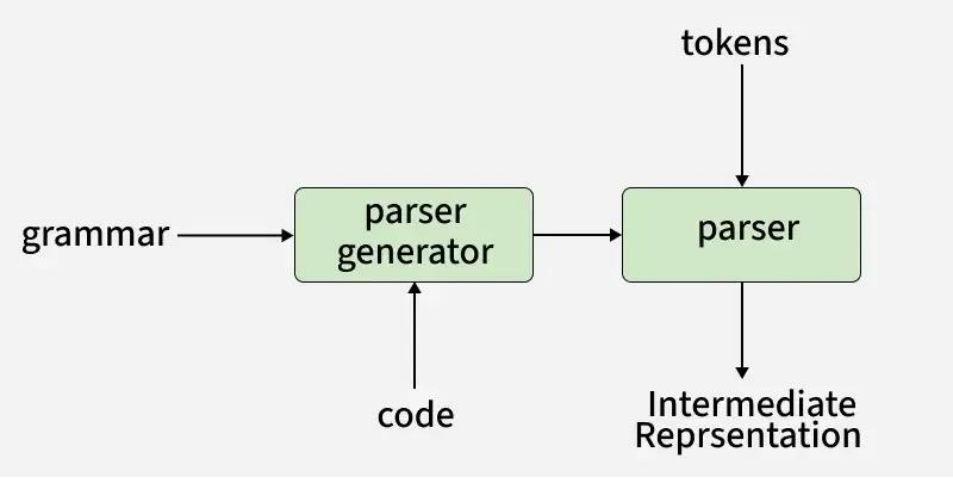
- *int main()*
  {
  // 2 variables
  int a, b;
  a = 10;
  return 0;
  }

All the valid tokens are:

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

Parsing

- Parsing, also known as syntactic analysis, is the process of analyzing a sequence of tokens to determine the grammatical structure of a program. It takes the stream of tokens, which are generated by a lexical analyzer or tokenizer, and organizes them into a parse tree or syntax tree.
- The parse tree visually represents how the tokens fit together according to the rules of the language's syntax. This tree structure is crucial for understanding the program's structure and helps in the next stages of processing, such as code generation or execution. Additionally, parsing ensures that the sequence of tokens follows the syntactic rules of the programming language, making the program valid and ready for further analysis or execution.

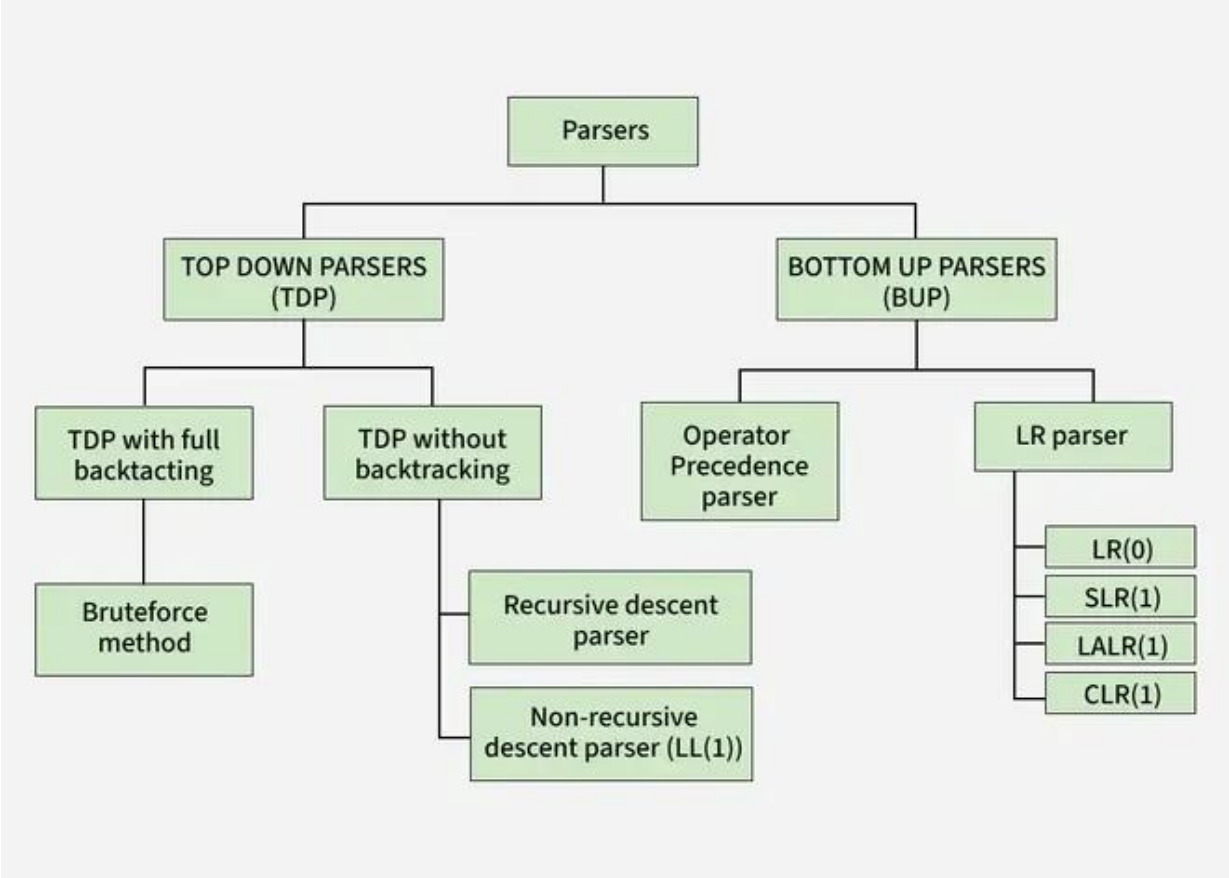


- **Role of Parser**
- A parser performs syntactic and semantic analysis of source code, converting it into an intermediate representation while detecting and handling errors.
- **Context-free syntax analysis**: The parser checks if the structure of the code follows the basic rules of the programming language (like grammar rules). It looks at how words and symbols are arranged.
- **Guides context-sensitive analysis**: It helps with deeper checks that depend on the meaning of the code, like making sure variables are used correctly. For example, it ensures that a variable used in a mathematical operation, like x + 2, is a number and not text.
- **Constructs an intermediate representation**: The parser creates a simpler version of your code that's easier for the computer to understand and work with.
- **Produces meaningful error messages**: If there's something wrong in your code, the parser tries to explain the problem clearly so you can fix it.
- **Attempts error correction**: Sometimes, the parser tries to fix small mistakes in your code so it can keep working without breaking completely.

- **Types of Parsing**

The parsing is divided into two types, which are as follows:

- Top-down Parsing
- Bottom-up Parsing



- **Top-Down Parsing**
- Top-down parsing is a method of building a parse tree from the start symbol (root) down to the leaves (end symbols). The parser begins with the highest-level rule and works its way down, trying to match the input string step by step.
- **Process**: The parser starts with the start symbol and looks for rules that can help it rewrite this symbol. It keeps breaking down the symbols (non-terminals) into smaller parts until it matches the input string.
- **Leftmost Derivation**: In top-down parsing, the parser always chooses the leftmost non-terminal to expand first, following what is called leftmost derivation. This means the parser works on the left side of the string before moving to the right.
- **Other Names**: Top-down parsing is sometimes called recursive parsing or predictive parsing. It is called recursive because it often uses recursive functions to process the symbols.
- Top-down parsing is useful for simple languages and is often easier to implement. However, it can have trouble with more complex or ambiguous grammars.

- Top-down parsers can be classified into two types based on whether they use backtracking or not:

1. Top-down Parsing with Backtracking

- In this approach, the parser tries different possibilities when it encounters a choice, If one possibility doesn't work (i.e., it doesn't match the input string), the parser backtracks to the previous decision point and tries another possibility.
- **Example**: If the parser chooses a rule to expand a non-terminal, and it doesn't work, it will go back, undo the choice, and try a different rule.
- **Advantage**: It can handle grammars where there are multiple possible ways to expand a non-terminal.
- **Disadvantage**: Backtracking can be slow and inefficient because the parser might have to try many possibilities before finding the correct one.

2. Top-down Parsing without Backtracking

- In this approach, the parser does not backtrack. It tries to find a match with the input using only the first choice it makes, If it doesn’t match the input, it fails immediately instead of going back to try another option.
- **Example**: The parser will always stick with its first decision and will not reconsider other rules once it starts parsing.
- **Advantage**: It is faster because it doesn’t waste time going back to previous steps.
- **Disadvantage**: It can only handle simpler grammars that don’t require trying multiple choices.

- **Bottom-Up Parsing**
- Bottom-up parsing is a method of building a parse tree starting from the leaf nodes (the input symbols) and working towards the root node (the start symbol). The goal is to reduce the input string step by step until we reach the start symbol, which represents the entire language.
- **Process**: The parser begins with the input symbols and looks for patterns that can be reduced to non-terminals based on the grammar rules. It keeps reducing parts of the string until it forms the start symbol.
- **Rightmost Derivation in Reverse**: In bottom-up parsing, the parser traces the rightmost derivation of the string but works backwards, starting from the input string and moving towards the start symbol.
- **Shift-Reduce Parsing**: Bottom-up parsers are often called shift-reduce parsers because they shift (move symbols) and reduce (apply rules to replace symbols) to build the parse tree.
- Bottom-up parsing is efficient for handling more complex grammars and is commonly used in compilers. However, it can be more challenging to implement compared to top-down parsing.

- Generally, bottom-up parsing is categorized into the following types:
- **1. LR parsing/Shift Reduce Parsing:** Shift reduce Parsing is a process of parsing a string to obtain the start symbol of the grammar.
- LR(0)
- SLR(1)
- LALR
- CLR
- **2. Operator Precedence Parsing:** The grammar defined using operator grammar is known as operator precedence parsing. In operator precedence parsing there should be no null production and two non-terminals should not be adjacent to each other.

- LR Parser **:**

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.

- **Description of LR parser :**
  The term parser LR(k) parser, here the L refers to the left-to-right scanning, R refers to the rightmost derivation in reverse and k refers to the number of unconsumed “look ahead” input symbols that are used in making parser decisions. Typically, k is 1 and is often omitted. A context-free grammar is called LR (k) if the LR (k) parser exists for it. This first reduces the sequence of tokens to the left. But when we read from above, the derivation order first extends to non-terminal.
- The stack is empty, and we are looking to reduce the rule by S’→S$.

  Using a “.” in the rule represents how many of the rules are already on the stack.
- A dotted item, or simply, the item is a production rule with a dot indicating how much RHS has so far been recognized. Closing an item is used to see what production rules can be used to expand the current structure. It is calculated as follows:

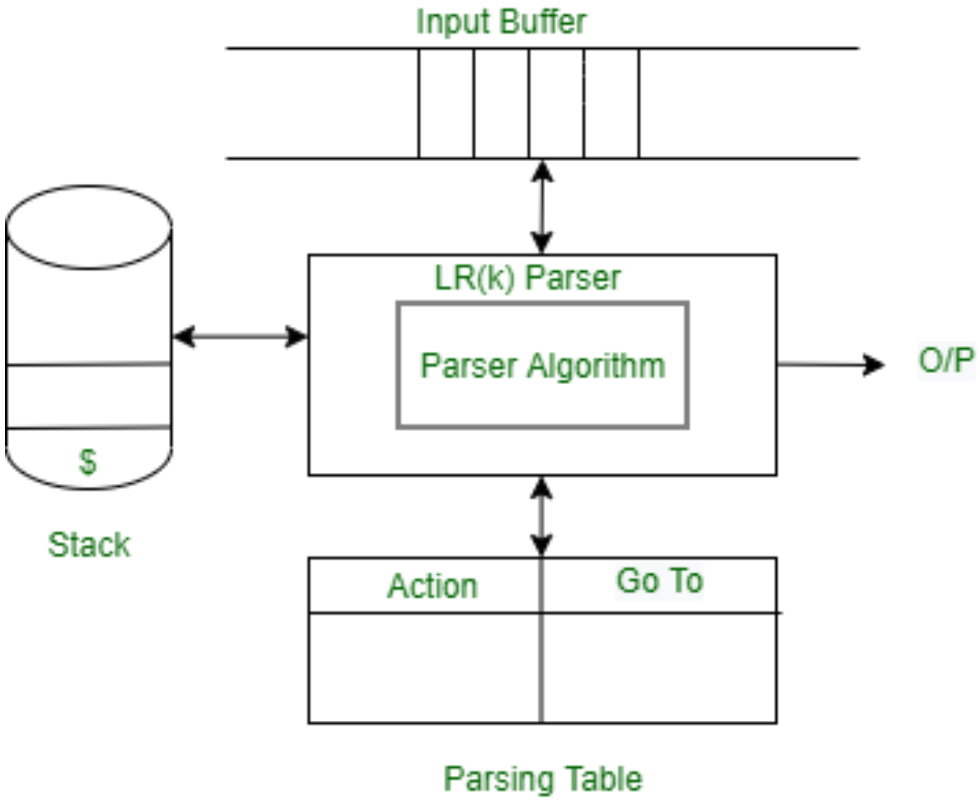
- **Rules for LR parser :**
  The rules of LR parser as follows.
- The first item from the given grammar rules adds itself as the first closed set.
- If an object is present in the closure of the form A→ α. β. γ, where the next symbol after the symbol is non-terminal, add the symbol's production rules where the dot precedes the first item.
- Repeat steps (B) and (C) for new items added under (B).
- **LR parser algorithm :**
  LR Parsing algorithm is the same for all the parser, but the parsing table is different for each parser. It consists following components as follows.
- **Input Buffer –**
  It contains the given string, and it ends with a $ symbol.

- **Stack –**
  The combination of state symbol and current input symbol is used to refer to the parsing table in order to take the parsing decisions.

- **Parsing Table** :

- Parsing table is divided into two parts- Action table and Go-To table. The **action table** gives a grammar rule to implement the given current state and current terminal in the input stream. There are four cases used in action table as follows.
- Shift Action- In shift action the present terminal is removed from the input stream and the state ***n*** is pushed onto the stack, and it becomes the new present state.
- Reduce Action- The number ***m*** is written to the output stream.
- The symbol ***m*** mentioned in the left-hand side of rule ***m*** says that state is removed from the stack.
- The symbol ***m*** mentioned in the left-hand side of rule **m** says that a new state is looked up in the goto table and made the new current state by pushing it onto the stack.

LR Parsing diagram



Left Recursion

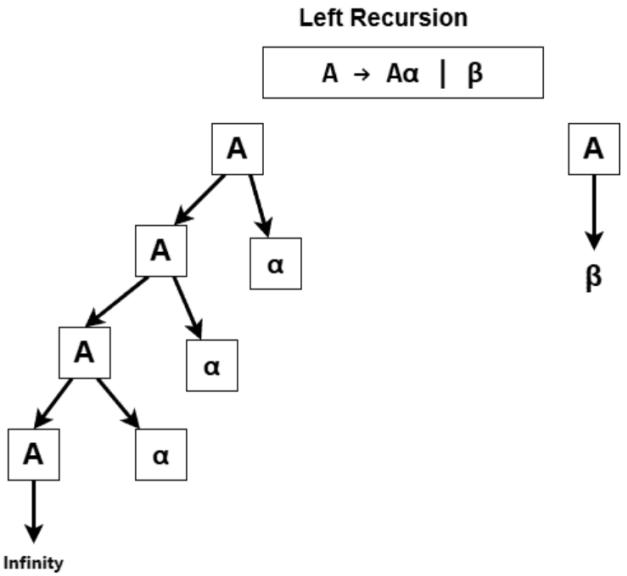
- Left recursion occurs when a non-terminal symbol in a grammar rule refers to itself as the leftmost symbol in its production:
- $$A \to A\alpha \mid \beta$$
- Here, *A* is a non-terminal, is a non-empty string, whereas is a string that can be empty.
- Furthermore, left recursion can be direct, as illustrated above, or indirect, where a non-terminal eventually refers back to itself through a chain of productions. For example:

$$A \to B\alpha$$
$$B \to A\beta$$

This recursion is indirect because *A* depends on *B*, which depends on *A*.

Problem in Left Recursion

- When some parsers attempt to parse input based on a left-recursive rule, such as , they can get stuck in an infinite loop, repeatedly trying to match *A* without ever reaching . This problem is common in top-down parsers (like recursive descent parsers), which expand rules starting from the left side:



This is the case with top-down parsers, such as LL parsers. These parsers try to predict which rule to apply based on the next input token, moving from left to right. However, they can't do that with a left-recursive rule, as they keep "recursing" without progressing through the input string. As a result, we get an infinite loop in the parser.

- For instance, let's say we have a left-recursive rule for parsing a list of terms separated by commas:

$$\text{List} \rightarrow \text{List}\ ','\ \text{Term} \mid \text{Term}$$

With this rule, the parser starts by trying to match *List*. Since it also sees *List* on the right-hand side, it tries to match it again, creating a never-ending cycle.

How to Eliminate Left Recursion?

We need to eliminate left recursion to make a grammar compatible with LL parsing. The general approach involves restructuring the rules so that recursion happens at the end of a production rather than at the beginning.
For instance, for a rule such as:

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

- the trick is to replace it with:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

This new setup uses a helper non-terminal, *A′*, which lets us handle the recursion “from the right” instead of directly looping
back to *A* (and denotes the empty string).
For example, suppose the input is *cdd*, and the grammar before elimination is:

$$A \to A\alpha \mid \beta$$
$$\alpha = d, \quad \beta = c$$

Using the left-recursive rule, a top-down parser would attempt:

$$A \to A\alpha \to A\alpha\alpha \to A\alpha\alpha\alpha \to \dots \text{ (infinite recursion)}$$

But after eliminating left recursion, the grammar becomes:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$
$$\alpha = d, \quad \beta = c$$

- For our previous input *cdd*, parsing now proceeds as:

$$A \to \beta A' \to cA'$$
$$A' \to \alpha A' \to dA'$$
$$A' \to \alpha A' \to dA'$$
$$A' \to \epsilon \text{ (ending recursion)}$$

Therefore, the transformation ensures the input *cdd* is successfully parsed, avoiding infinite recursion and making the grammar suitable for LL parsers.

- **Left Factoring**
- Left factoring is a technique that simplifies grammar rules to make parsing more efficient and less error-prone. This approach is especially helpful for top-down parsers, which can run into trouble when faced with ambiguous or left-recursive grammars.
- **The basic idea behind left factoring is to rewrite the rules that share beginnings (prefixes) so that the parser doesn't need to backtrack or guess which rule to apply.**

Why Do We Use Left Factoring

- Top-down parsers process input by scanning symbols from left to right and matching them against the grammar rules. **If two or more rules start with the same symbols, the parser can't immediately decide which one to follow. This results in backtracking or even parsing errors,** making the process less efficient.
- Left factoring removes this ambiguity by isolating the common parts of rules, reducing the need for the parser to backtrack. With left factoring, we get a clear, step-by-step process that the parser can follow smoothly.
- Let's look at this grammar that's not left-factored:

$$A \to \alpha\beta \mid \alpha\gamma$$

- In this example, both rules for *A* start with the symbol α. Without left factoring, the parser has to guess or backtrack after reading , which can lead to parsing issues.
- To apply left factoring, we rewrite this rule to group the common prefix α :

$$A \to \alpha A'$$
$$A' \to \beta \mid \gamma$$

- Now, the parser matches α and then decides between β and ϒ based on what follows. This way, the parser no longer needs to backtrack, so parsing is smoother.