UNIT 2

Syntax Analysis

Syntax analysis (parsing) is the second phase of the compilation process, following lexical analysis. Its primary goal is to verify the syntactical correctness of the source code. It takes the tokens generated by the lexical analyzer and attempts to build a Parse Tree or Abstract Syntax Tree (AST), representing the program's structure. During this phase, the syntax analyzer checks whether the input string adheres to the grammatical rules of the language using context-free grammar. If the syntax is correct, the analyzer moves forward; otherwise, it reports an error.

The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.

Syntax Analysis (also known as parsing) is the step after Lexical Analysis. The Lexical analysis breaks source code into tokens.

- Tokens are inputs for Syntax Analysis.
- The goal of Syntax Analysis is to interpret the meaning of these tokens.
- It checks whether the tokens produced by the lexical analyzer are arranged according to the language's grammar.
- The syntax analyzer attempts to build a Parse Tree or Abstract Syntax Tree (AST), which represents the program's structure.

Parsing Algorithms Used in Syntax Analysis

- **LL parsing:** This is a top-down parsing algorithm that starts with the root of the parse tree and constructs the tree by successively expanding non-terminals. LL parsing is known for its simplicity and ease of implementation.
- **LR parsing:** This is a bottom-up parsing algorithm that starts with the leaves of the parse tree and constructs the tree by successively reducing terminals. LR parsing is more powerful than LL parsing and can handle a larger class of grammars.
- **LR(1) parsing:** This is a variant of LR parsing that uses lookahead to disambiguate the grammar.
- **LALR parsing:** This is a variant of LR parsing that uses a reduced set of lookahead symbols to reduce the number of states in the LR parser.

- Once the parse tree is constructed, the compiler can perform semantic analysis to check if the source code makes sense and follows the semantics of the programming language.
- The parse tree or AST can also be used in the code generation phase of the compiler design to generate intermediate code or machine code .

Formalisms for Syntax Analysis in Compiler Design

In syntax analysis, various formalisms help in understanding and verifying the structure of the source code. Here are some key concepts:

1. Context-Free Grammars (CFG)

Context-Free Grammars define the syntax rules of a programming language. They consist of production rules that describe how valid strings (sequences of tokens) are formed. CFGs are used to specify the grammar of the language, ensuring that the source code adheres to the language's syntax.

2. Derivations

A derivation is the **process of applying the rules of a Context-Free Grammar to generate a sequence of tokens,** ultimately forming a valid structure. It helps in constructing a parse tree, which represents the syntactic structure of the source code.

3. Concrete and Abstract Syntax Trees

- **Concrete Syntax Tree (CST)**: Represents the full syntactic structure of the source code, including every detail of the grammar.
- **Abstract Syntax Tree (AST)**: A simplified version of the CST, focusing on the essential elements and removing redundant syntax to make it easier for further processing.

4. Ambiguity

Ambiguity occurs when a grammar allows multiple interpretations for the same string of tokens. This can lead to errors or inconsistencies in parsing, making it essential to avoid ambiguous grammar in programming languages.

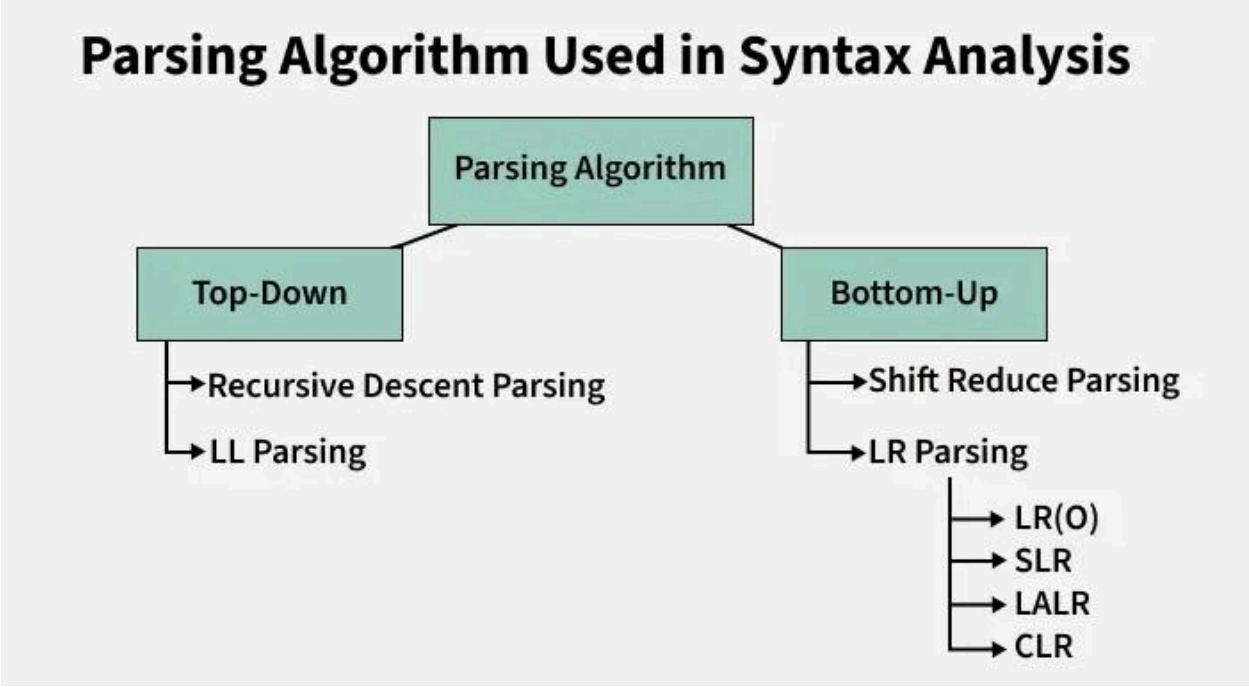
These formalisms are crucial for performing accurate syntax analysis and ensuring that the source code follows the correct grammatical structure.

Features of Syntax Analysis

- **Syntax Trees:** Syntax analysis creates a syntax tree, which is a hierarchical representation of the code's structure. The tree shows the relationship between the various parts of the code, including statements, expressions, and operators.
- **Context-Free Grammar:** Syntax analysis uses context-free grammar to define the syntax of the programming language. Context-free grammar is

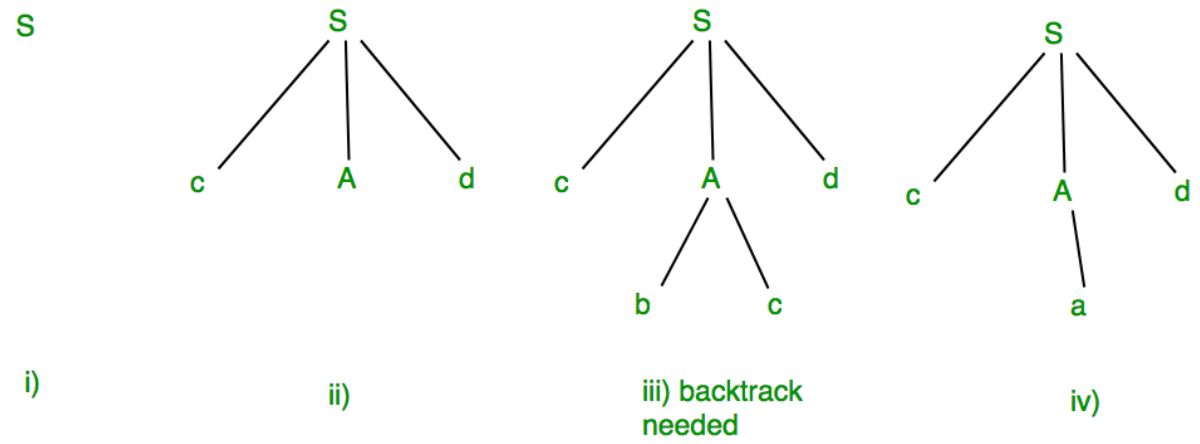
a formal language used to describe the structure of programming languages.

- **Top-Down and Bottom-Up Parsing:** Syntax analysis can be performed using two main approaches: top-down parsing and bottom-up parsing. Top-down parsing starts from the highest level of the syntax tree and works its way down, while bottom-up parsing starts from the lowest level and works its way up.
- **Error Detection:** Syntax analysis is responsible for detecting syntax errors in the code. If the code does not conform to the rules of the programming language, the parser will report an error and halt the compilation process.
- **Intermediate Code Generation:** Syntax analysis generates an intermediate representation of the code, which is used by the subsequent phases of the compiler. The intermediate representation is usually a more abstract form of the code, which is easier to work with than the original source code.
- **Optimization:** Syntax analysis can perform basic optimizations on the code, such as removing redundant code and simplifying expressions.



The pushdown automata (PDA) is used to design the syntax analysis phase.
The Grammar for a Language consists of Production rules.
Example: Suppose Production rules for the Grammar of a language are:

S -> cAd
A -> bc|a
And the input string is “cad”.
Now the parser attempts to construct a syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string “cad” it uses the rules as shown in the given diagram:



In step (iii) above, the production rule A->bc was not a suitable one to apply (because the string produced is “cbcd” not “cad”), here the parser needs to backtrack, and apply the next production rule available with A which is shown in step (iv), and the string “cad” is produced.
Thus, the given input can be produced by the given grammar, therefore the input is correct in syntax. But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.
Steps in Syntax Analysis Phase

- **Tokenization:** The input program is divided into a sequence of tokens, which are basic building blocks of the programming language, such as identifiers, keywords, operators, and literals.
- **Parsing:** The tokens are analyzed according to the grammar rules of the programming language, and a parse tree or AST is constructed that represents the hierarchical structure of the program.
- **Error handling:** If the input program contains syntax errors, the syntax analyzer detects and reports them to the user, along with an indication of where the error occurred.

- **Symbol table creation:** The syntax analyzer creates a symbol table, which is a data structure that stores information about the identifiers used in the program, such as their type, scope, and location.
- The syntax analysis phase is essential for the subsequent stages of the compiler, such as semantic analysis, code generation, and optimization. If the syntax analysis is not performed correctly, the compiler may generate incorrect code or fail to compile the program altogether.

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

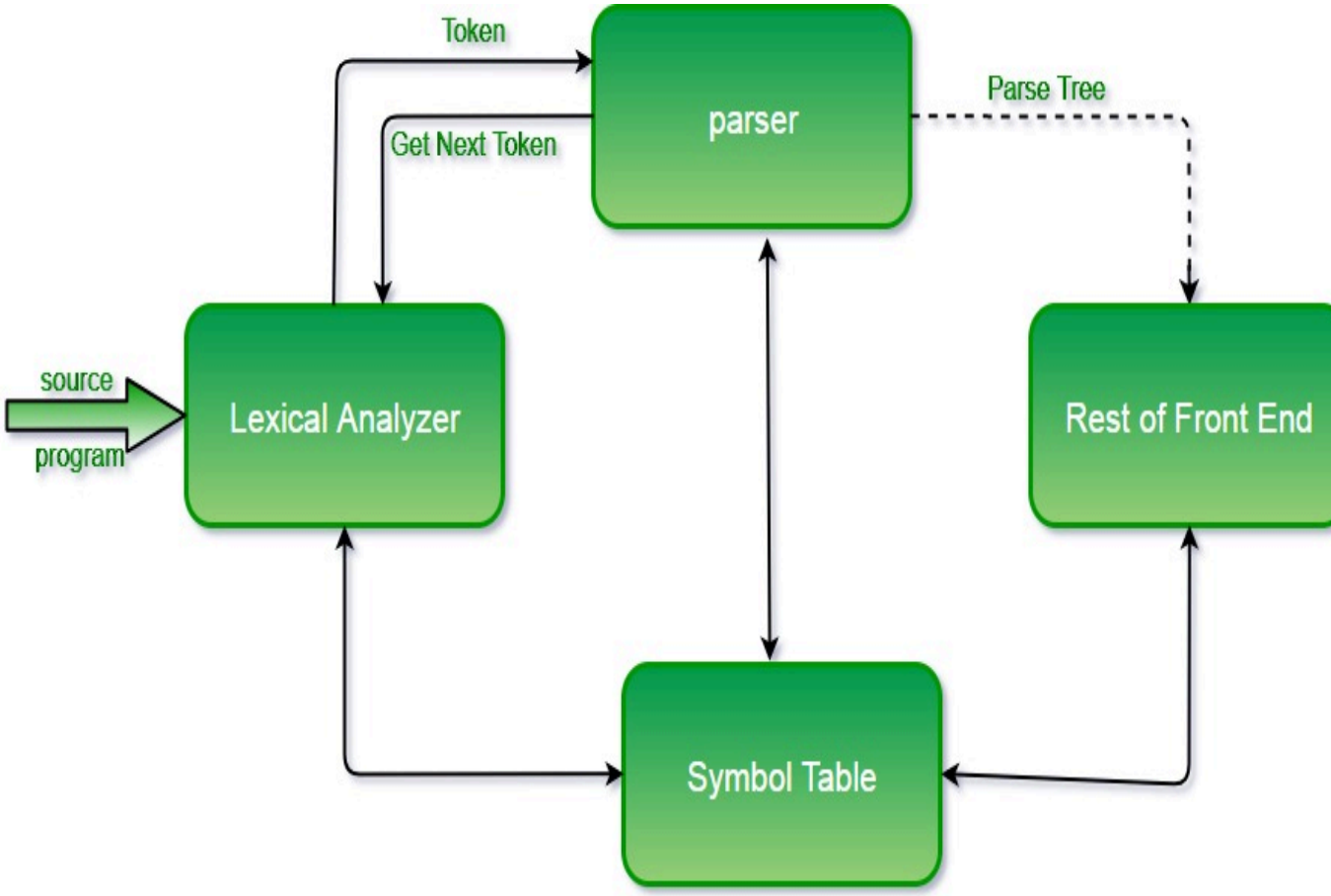


Fig. 2.1 Position of parser in comp Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.

4. It performs error recovery.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use
3. Data type mismatch for an operation.

The above issues are handled by the Semantic Analysis phase.

Syntax error handling :

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling an identifier, keyword or operator.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the **parser discards input symbols one at a time until a synchronizing token is found**. The **synchronizing tokens are usually delimiters, such as semicolon or end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the **parser performs local correction on the remaining input** that allows it to continue. Example: **Insert a missing semicolon or delete an extraneous semicolon** etc.

Error productions:

The **parser is constructed using augmented grammar with error productions.** If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

WRITING A GRAMMAR

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a ***nonterminal* as its *left-hand side***, and a **sequence of one or more nonterminal and *terminal* symbols as its *right-hand side***. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of terminals,non-terminals, start symbols and productions.

Terminals: These are the basic symbols from which strings are formed.

Non-Terminals: These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol: Onenon-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and nonterminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Grammars are used to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable stmt to denote statements and variable expr to denote expressions, the production

stmt → **if** (expr) stmt **else** stmt

It specifies the structure of this form of conditional statement.

The Formal Definition of a Context-Free Grammar

A context-free grammar G is defined by the 4-tuple: G= (V, T, P S) where

1. V is a finite set of non-terminals (variable).
2. T is a finite set of terminals.
3. S is the start symbol (variable S∈V)
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Notational Conventions

These symbols are terminals:

- Lowercase letters early in the alphabet, such as a, b, c.
- Operator symbols such as +, *,- , / and so on.
- Punctuation symbols such as parentheses, comma, and so on.
- The digits 0 , 1 , . . . ,9.
- Boldface strings such as id or if, each of which represents a single terminal symbol

These symbols are non-terminals:

- Uppercase letters early in the alphabet, such as A, B, C.
- The letter S, which, when it appears, is usually the start symbol.
- Lowercase, italic names such as expr or stmt.

Example:

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

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

$F \rightarrow (E) \mid id$

Start symbol: E

Terminal: +, - , * , / , (,) , id

Non Terminal: E , T , F

Derivations: `

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example: Consider the following grammar for arithmetic expressions:

E → E+E | E*E | (E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- -(id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called **sentinel** forms.

Types of derivations:

The two types of derivation are:

- **Left most derivation:** In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- **Right most derivation:** In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement. Rightmost derivations are sometimes called canonical derivations.

Example: Given grammar G : E ⟶ E+E | E*E | (E) | - E | id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION

E → - E

E → - (E)

E → - (E+E)

E → - (id+E)

E → - (id+id)

RIGHTMOST DERIVATION

E → - E

E → - (E)

E → - (E+E)

E → - (E+id)

E → - (id+id)

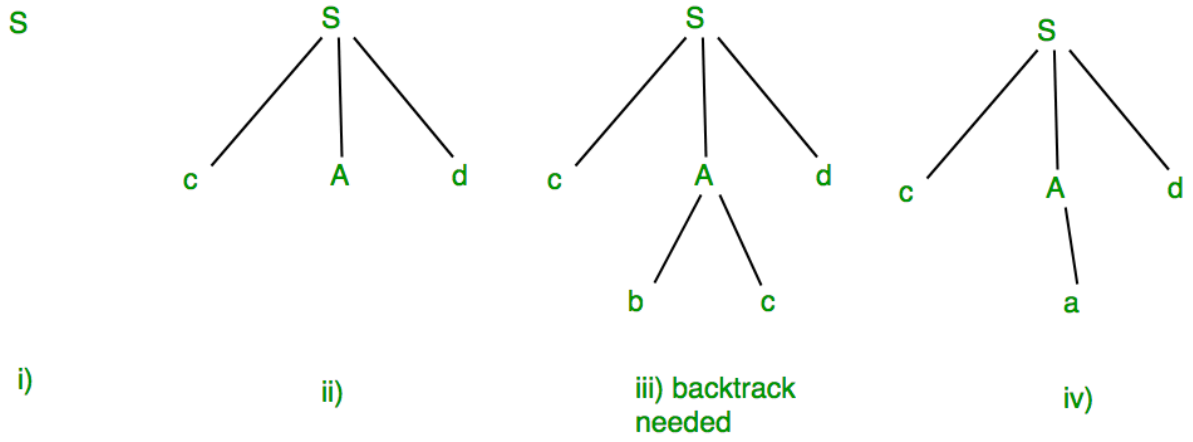
Parse Trees:

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

- **The root** of the tree represents the start symbol of the grammar.
- **Internal nodes** represent non-terminal symbols, which are expanded according to the production rules.
- **Leaf nodes** represent terminal symbols, which are the actual tokens from the input string.

Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.
Example: Suppose Production rules for the Grammar of a language are:

S -> cAd
A -> bc|a
And the input string is "cad".



Parsing

Parsing is the second phase of a compiler, also known as syntax analysis. Parsing is the process of analyzing the structure of a program's source code to ensure it follows the rules of the programming language.

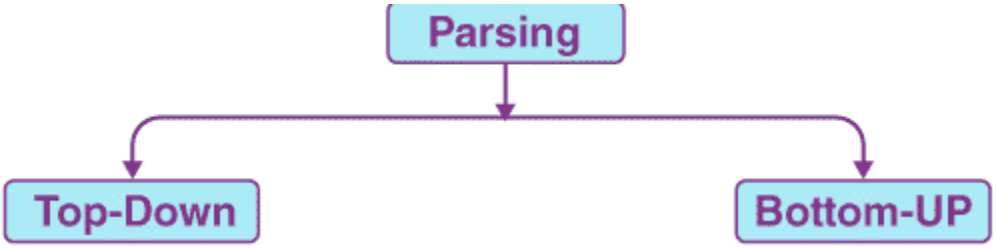
- The compiler receives tokens from the lexical analyzer, which is the first phase of compilation.
- The compiler then uses the tokens to build a parse tree, which is a hierarchical representation of the program's structure.
- The compiler uses the parse tree to check for syntax errors.

Parser is a compiler that is used to break the data into smaller elements coming from the lexical analysis phase.
A parser takes input in the form of a sequence of tokens and produces output in the form of a parse tree.
The parser is one of the phases of the compiler which takes a token of string as input and converts it into the corresponding **Intermediate Representation (IR)** with the help of an existing grammar. The parser is also known as Syntax Analyzer.

Types of Parsing

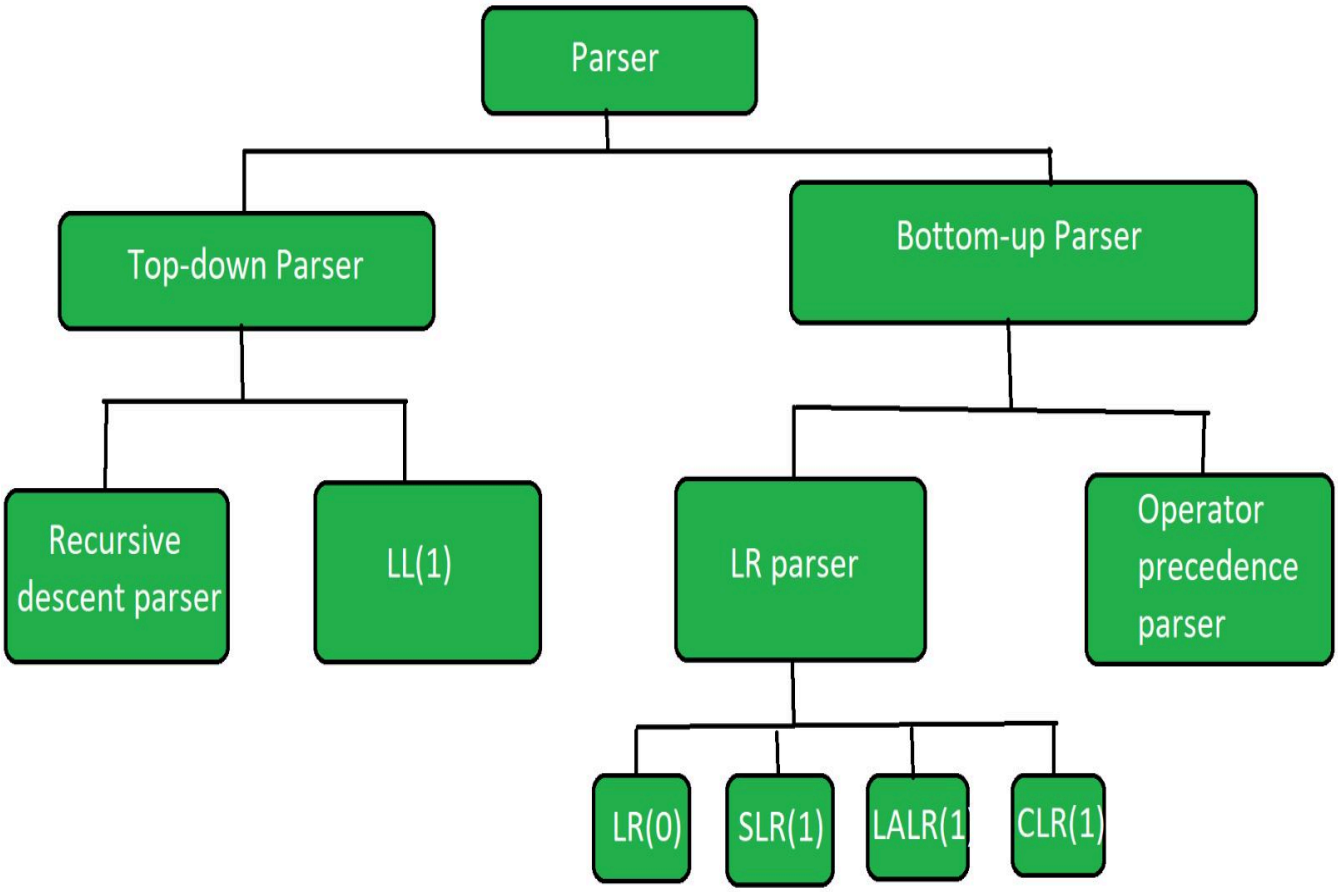
The parsing is divided into two types, which are as follows:

- Top-down Parsing
- Bottom-up Parsing



There are two types of Parsing:

- The Top-down Parsing
- The Bottom-up Parsing



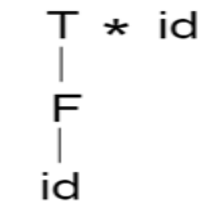
- **Top-down Parsing:** Top-down parser is the parser that generates a parse tree for the given input string with the help of grammar productions by expanding the non-terminals. It **starts from the start symbol and ends down on the terminals. It uses the leftmost derivation.**
    - **Recursive Descent Parsing:** Recursive descent parsing is a type of top-down parsing technique. This technique follows the process for every terminal and non-terminal entity. It **reads the input from left to right and constructs the parse tree from right to left.** As the **technique works recursively**, it is called recursive descent parsing.It basically generates the parse tree by using brute force and backtracking techniques.
    - **Back-tracking:** The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules.
    - **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.
- **Bottom-up Parsing:** The bottom-up parsing works just the reverse of the top-down parsing. It **first traces the rightmost derivation of the input until it reaches the start symbol.**

Example

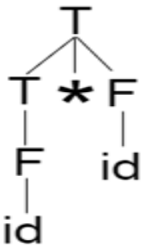
Production

$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow id$
$F \rightarrow T$
$F \rightarrow id$

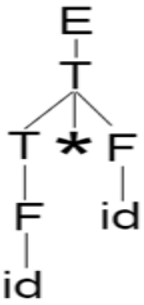
Parse Tree representation of input string "id * id" is as follows:



Step 4



Step 5



Step 6

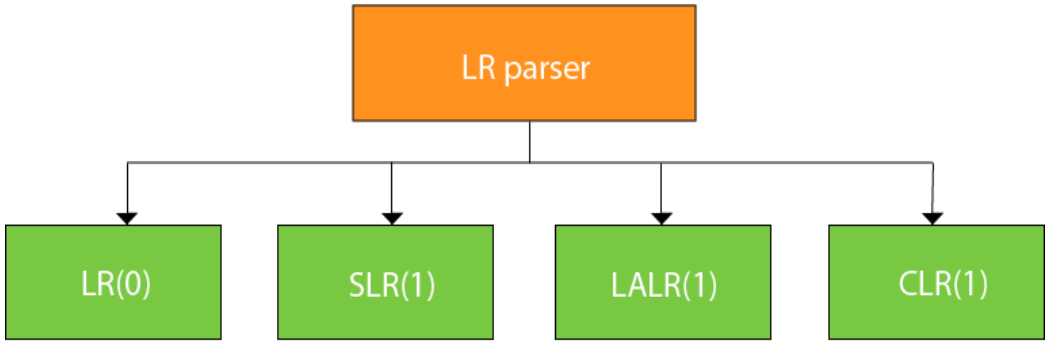


Fig: Types of LR parser

- **Shift-Reduce** Parsing: Shift-reduce parsing works on two steps: Shift step and Reduce step.
    - **Shift step:** The shift step indicates the increment of the input pointer to the next input symbol that is shifted.
    - **Reduce Step:** When the parser has a complete grammar rule on the right-hand side and replaces it with RHS.
- **LR Parsing:** LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to right tracing, and R stands for the **rightmost derivation in reverse.**

Why is parsing useful in compiler designing?

In the world of software, every different entity has its criteria for the data to be processed. So **parsing is the process that transforms the data in such a way so that it can be understood by any specific software.**

The Technologies Use Parsers:

- The programming languages like Java.
- Database languages like SQL.
- The Scripting languages.

- The protocols like HTTP.
- The XML and HTML.

How to find first and follow:

FIRST and FOLLOW in compiler design are two grammatical functions that help you enter table entries. We will discuss the First and Follow in detail below. If the compiler knew ahead of time what the "initial character and follow up of the string produced when a production rule is applied," it might carefully choose which production rule to apply by comparing it to the current character or token in the input string it sees.

First(): FIRST () is a function that specifies the set of terminals that start a string derived from a production rule. **It is the first terminal that appears on the right-hand side of the production.** For example,

If the Input string is
T->*FT'/ε
Here we find out that T has two productions like T->*FT' and T->ε, after viewing this we found the first of T in both the production statement which is * and ε.
Then the first of the string will be **{*,ε}.**

Rules to find First(): To find the first() of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

- If A is a terminal, then First(A) is {A}.
- If A is a non-terminal and A tends to aα is production, then add 'a' to the first of A. if A->ε, then add null to the First(A).
- If A_>BC then if First(B)=ε, then First(A) = { First(B)-ε} U First(C).
- If A->BC, then if First(A)=B, then First(B)=teminal but null then First(A)=First(B)=terminals.

Follow(): Follow () is a set of terminal symbols that can be displayed just to the right of the non-terminal symbol in any sentence format. It is the first terminal appearing after the given non-terminal symbol on the right-hand side of production.

For example,
If the input string is

E->TE', F->(E)/id
Here we found that on the right-hand side of the production statement where the E occurs, we only found E in the production F->(E)/id through which we found the follow of E.
Then the output Follow of E = {) }, as ')' is the terminal in the input string on the right-hand side of the production.

Rules to find Follow() To find the follow(A) of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

- If S is start symbol then, $ is a follow of 'S'(start symbol).
- If A->αBβ, β!=ε, then follow(B) = first(β)
  If first (β) doesnot contain ε
- If A->αB or A->αBβ where First(β)=ε, then Follow(B) = Follow(A).

Assumptions

- Epsilon is represented by '#'.
- Productions are of the form A=B, where 'A' is a single Non-Terminal and 'B' can be any combination of Terminals and Non- Terminals.
- The L.H.S. of the first production rule is the start symbol.
- Grammar is not left recursive.
- Each production of a non-terminal is entered on a different line.
- Only Upper Case letters are Non-Terminals and everything else is a terminal.
- Do not use '!' or '$' as they are reserved for special purposes.

Example of First and Follow Let us consider grammar to show how to find the first and follow in compiler design.

E->TE'
E'->+TE'/ε
T->FT'
T'->*FT'/ε
F->(E)/id

Here,
Terminals are id, *, +, ε, (,)

- Non-terminals are E, E', T, T', F

- Now let's try to find the first of 'E'. here on the right-hand side of the production E->TE' is T which is a non-terminal but we have to find the terminals so to find terminals we move to the production T->FT' in which the first element is again a non-terminal, so we move to the third production F->(E)/id in which the first element is a terminal which will be the first of E.
- So, First(E)={(, id}
- Now let's try to find the follow of 'E', to find this we find the production in which 'E' is on the right-hand side and we get production which is F->(E)/id, so the follow will be the next non-terminal followed by the terminals which are ')' and in the follow '$' is always added. So the follow(E)={$,)}
- On repeating the above steps to find the first and follow in compiler design, we get

E->TE'
E'->+TE'/ε
T->FT'
T'->*FT'/ε
F->(E)/id

Non-terminals	First()	Follow()
E	{(,id)}	{$,)}
E'	{+,∈}	{$,)}
T	{(,id)}	{$,+,)}
T'	{*,∈}	{+,),$}
F	{(,id}	{+,),$,*}

Example of First and Follow:

Problem-01:

Calculate the first and follow functions for the given grammar-

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \in$

$D \rightarrow EF$

$E \rightarrow g / \in$

$F \rightarrow f / \in$

Solution-

First Functions-

First(S) = { a }

First(B) = { c }

First(C) = { b , ∈ }

First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }

First(E) = { g , ∈ }

First(F) = { f , ∈ }

Follow Functions-

Follow(S) = { $ }

Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }

Follow(C) = Follow(B) = { g , f , h }

Follow(D) = First(h) = { h }

Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }

Follow(F) = Follow(D) = { h }

Problem-02:

Calculate the first and follow functions for the given grammar-

$S \rightarrow A$

$A \rightarrow aB / Ad$

$B \rightarrow b$

C → g

Solution-

We have-

The given grammar is left recursive.

So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

S → A

A → aBA'

A' → dA' / ∈

B → b

C → g

Now, the first and follow functions are as follows-

First Functions-

First(S) = First(A) = { a }

First(A) = { a }

First(A') = { d , ∈ }

First(B) = { b }

First(C) = { g }

Follow Functions-

Follow(S) = { $ }

Follow(A) = Follow(S) = { $ }

Follow(A') = Follow(A) = { $ }

Follow(B) = { First(A') – ∈ } ∪ Follow(A) = { d , $ }

Follow(C) = NA

<u>Problem-03:</u>

Calculate the first and follow functions for the given grammar-

S → (L) / a

L → SL'

L' → ,SL' / ∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { (, a }

First(L) = First(S) = { (, a }

First(L') = { , , ∈ }

Follow Functions-

Follow(S) = { $ } ∪ { First(L') – ∈ } ∪ Follow(L) ∪ Follow(L') = { $, , ,) }
Follow(L) = {) }
Follow(L') = Follow(L) = {) }

Problem-04:

Calculate the first and follow functions for the given grammar-
S → AaAb / BbBa
A → ∈
B → ∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { First(A) – ∈ } ∪ First(a) ∪ { First(B) – ∈ } ∪ First(b) = { a , b }
First(A) = { ∈ }
First(B) = { ∈ }

Follow Functions-

Follow(S) = { $ }
Follow(A) = First(a) ∪ First(b) = { a , b }
Follow(B) = First(b) ∪ First(a) = { a , b }

Problem-05:

Calculate the first and follow functions for the given grammar-
E → E + T / T
T → T x F / F
F → (E) / id

Solution-

We have-
The given grammar is left recursive.
So, we first remove left recursion from the given grammar.
After eliminating left recursion, we get the following grammar-
E → TE'
E' → + TE' / ∈
T → FT'
T' → x FT' / ∈
F → (E) / id
Now, the first and follow functions are as follows-

First Functions-

First(E) = First(T) = First(F) = { (, id }
First(E') = { + , ∈ }
First(T) = First(F) = { (, id }
First(T') = { x , ∈ }
First(F) = { (, id }

Follow Functions-

Follow(E) = { $,) }
Follow(E') = Follow(E) = { $,) }
Follow(T) = { First(E') – ∈ } ∪ Follow(E) ∪ Follow(E') = { + , $,) }
Follow(T') = Follow(T) = { + , $,) }
Follow(F) = { First(T') – ∈ } ∪ Follow(T) ∪ Follow(T') = { x , + , $,) }

Problem-06:

Calculate the first and follow functions for the given grammar-
S → ACB / CbB / Ba
A → da / BC
B → g / ∈
C → h / ∈

Solution-

The first and follow functions are as follows-

First Functions-

First(S) = { First(A) – ∈ } ∪ { First(C) – ∈ } ∪ First(B) ∪ First(b) ∪ { First(B) – ∈ } ∪ First(S) = { d , g , h , ∈ , b , a }
First(A) = First(d) ∪ { First(B) – ∈ } ∪ First(C) = { d , g , h , ∈ }
First(B) = { g , ∈ }
First(C) = { h , ∈ }

Follow Functions-

Follow(S) = { $ }
Follow(A) = { First(C) – ∈ } ∪ { First(B) – ∈ } ∪ Follow(S) = { h , g , $ }
Follow(B) = Follow(S) ∪ First(a) ∪ { First(C) – ∈ } ∪ Follow(A) = { $, a , h , g }
Follow(C) = { First(B) – ∈ } ∪ Follow(S) ∪ First(b) ∪ Follow(A) = { g , $, b , h }

Left Recursion

A context-free grammar is said to be left recursive if it contains a production rule where the non-terminal on the left-hand side of the rule also appears as the first symbol on the right-hand side. In other words, the grammar is trying to define a non-terminal in terms of itself, creating a recursive loop.

This can be represented formally as **−**

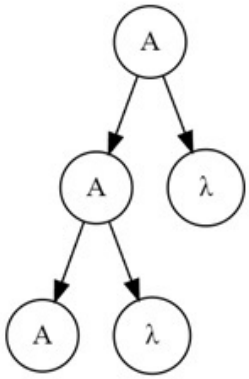
$$A \rightarrow A\alpha | \beta$$

Where **−**

A is a non-terminal symbol.
α represents a sequence of terminals and/or non-terminals.
β represents another sequence of terminals and/or non-terminals.

The most important part here is the presence of A on both sides of the production rule, with it appearing first on the right-hand side.

To visualize this, consider the following **parse tree −**



It is generated by a left-recursive grammar. As the grammar recursively expands the non-terminal 'A' on the left, the tree grows indefinitely downwards on the left side. This continuous expansion makes it unsuitable for top-down parsing, as the parser could get trapped in an infinite loop, trying to expand 'A' repeatedly.

Problem of Left Recursion for Top-Down Parsing

As we have seen, the top-down parsing works by starting with the start symbol of the grammar and attempting to derive the input string by applying production rules.

When encountering a left-recursive rule, the parser keeps expanding the same non-terminal, leading to an infinite loop. This inability to handle left recursion directly is a significant drawback of top-down parsing methods.

Eliminating Left Recursion

To solve this we can eliminate immediate left recursion from a grammar without altering the language it generates. The general approach involves introducing a new non-terminal and rewriting the recursive rules.
Let's illustrate this with our previous example −

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

We can eliminate the left recursion by introducing a new non-terminal 'A'` and rewriting the rule as follows −

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

In this **transformed grammar** −

A now derives the non-recursive part 'β' followed by the new non-terminal A'.

A' handles the recursion, deriving either α A' (continuing the recursion) or ϵ (empty string), which terminates the recursion.

Let us see this through an example for a better view
Consider a simplified arithmetic expression grammar −

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

This grammar contains left recursion in the rules for both `E` and `T`. Let's eliminate it −

Eliminating Left Recursion in `E`:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$

Eliminating Left Recursion in `T`:

$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$

The final transformed grammar, free from left recursion, becomes −

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$

T′→∗FT′ | ε
F→(E)|id

Recursive Descent Parser

A **recursive descent parser** is a top-down parser that processes input based on a set of recursive functions, where each function corresponds to a grammar rule. It parses the input from left to right, constructing a parse tree by matching the grammar's production rules. This parser is simple to implement and is suitable for LL(1) grammars, where decisions can be made based on a single lookahead token. While straightforward, recursive descent parsers struggle with left-recursive grammars and may require grammar transformations to handle such cases effectively.

- In recursive descent parsing, the **parser may have more than one production** to choose from for a single instance of input, **whereas in predictive parser, each step has at most one production to choose**. There might be instances where there is no production matching the input string, making the parsing procedure to fail.
- The major approach of recursive-descent parsing is to relate each non-terminal with a procedure. The objective of each procedure is to **read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree** for the non-terminal. The structure of the procedure is prescribed by the productions for the equivalent non-terminal.

For example, input stream is a + b$.
lookahead == a
match()
lookahead == +
match ()
lookahead == b
…….
…….
In this manner, parsing can be done.

Example − Write down the algorithm using Recursive procedures to implement the following Grammar.

Before removing left recursion	After removing left recursion
E –> E + T | T T –> T * F | F F –> (E) | id	E –> T E' E' –> + T E' | e T –> F T' T' –> * F T' | e F –> (E) | id

E → TE′
E′ → +TE′
T → FT′
T′ →* FT′|ε
F → (E)|id

Solution

```
Procedure E ( )
{
        T ( );
        E′( );
}
```

$E \rightarrow TE'$

```
Procedure E′( )
{
        If input symbol ='+' then
                advance( );
                T ( );
                E′( );
}
```

$E \rightarrow + TE'$

```
Procedure T( )
{
        F ( );
        T′( );
}
```

$T \rightarrow F\,T'$

```
Procedure T′( )
{
        If input symbol ='*' then
        advance( );
        F ( );
        T′( );
}
```

$T' \rightarrow * \; FT'$

```
Procedure F ( )
{
        If input symbol ='id' then          F → id
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'               F → (E)
        advance ( );
        else error ( );
        else error ( );
}
```

Example: E->iE’

E’->+iE’/ε

INPUT i+i$

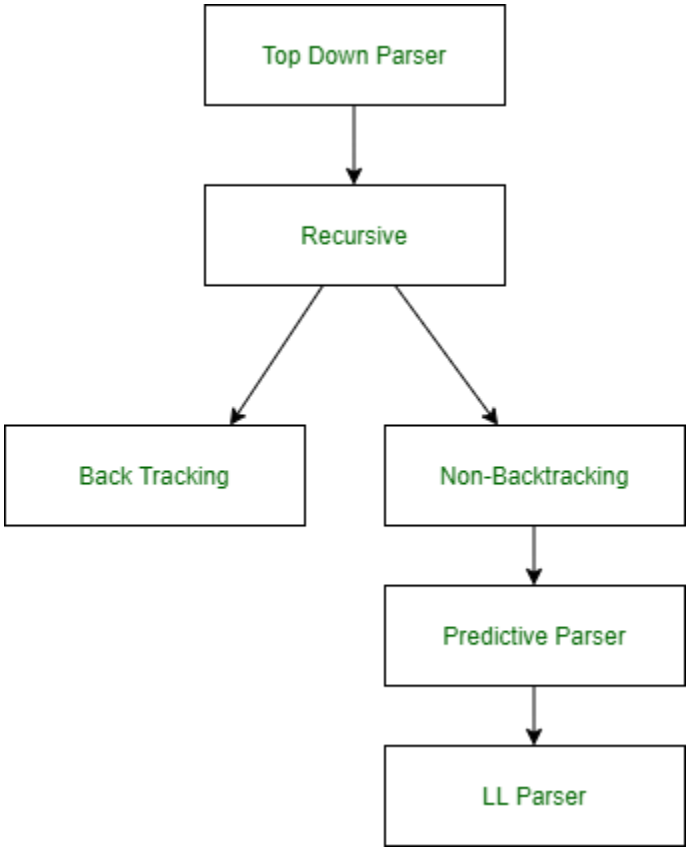
E’()
{
(If input ==+)
{
if input ++
if (input ==i)
input ++;
E’();
}
else
return ;
}
	E()
	{
	if (input==i)
	input ++;
	E’();
	}
		main()
		{
		E();
		if (input==$)
		parsing successful
		}

One of the major drawbacks of recursive-descent parsing is that it can be implemented only for those languages which support recursive procedure calls and it suffers from the problem of left-recursion.

Predictive Parser

A **predictive parser is a recursive descent parser with no backtracking or backup.** It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.
Consider
A -> A1 | A2 | ... | An

If the non-terminal is to be further expanded to 'A', the rule is selected based on the current input symbol 'a' only.



Predictive Parser Algorithm :

1. Make a transition diagram(DFA/NFA) for every rule of grammar.
2. Optimize the DFA by reducing the number of states, yielding the final transition diagram.
3. Simulate the string on the transition diagram to parse a string.

4. If the transition diagram reaches an accept state after the input is consumed, it is parsed.

Consider the following grammar –

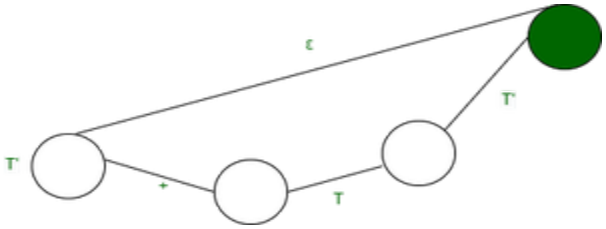
E->E+T|T
T->T*F|F
F->(E)|id
After removing left recursion, left factoring
E->TT'
T'->+TT'|ε
T->FT''
T''->*FT''|ε
F->(E)|id

STEP 1:

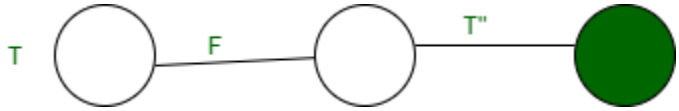
Make a transition diagram(DFA/NFA) for every rule of grammar.

- E->TT’

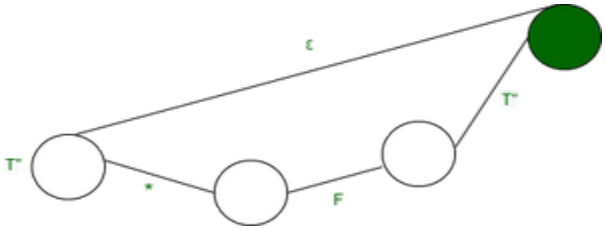
- T’->+TT’|ε



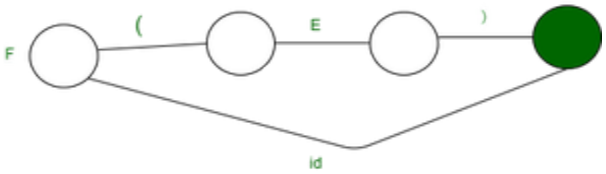
- T->FT”



- T”->*FT”|ε



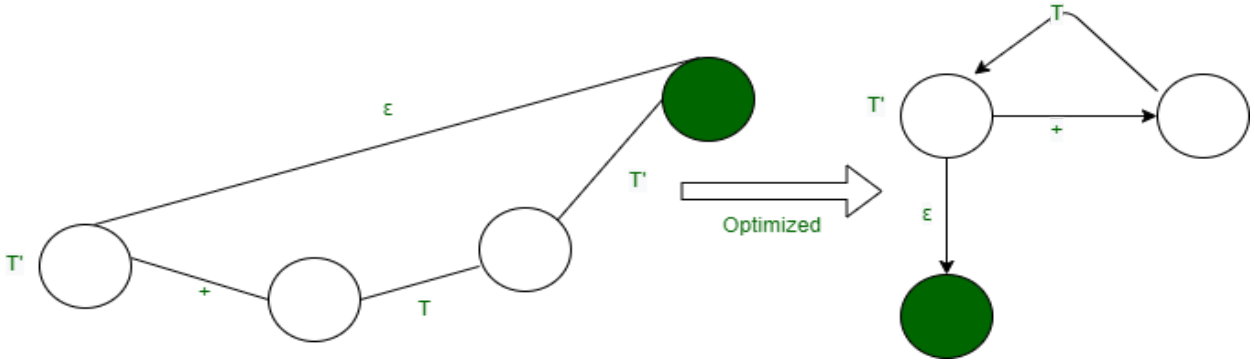
- F->(E)|id



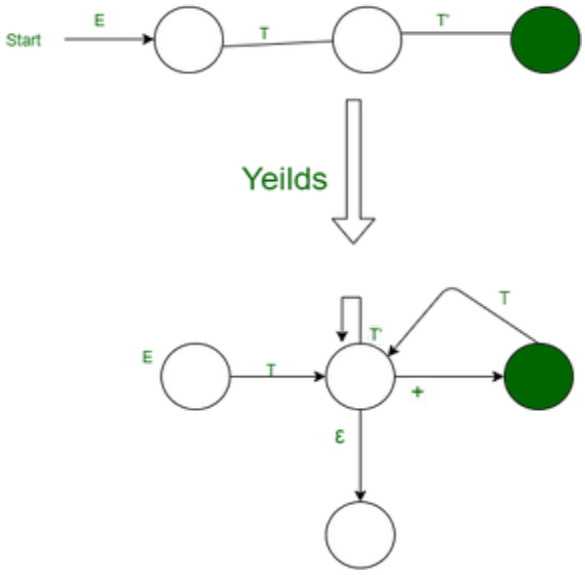
STEP 2:

Optimize the DFA by decreases the number of states, yielding the final transition diagram.

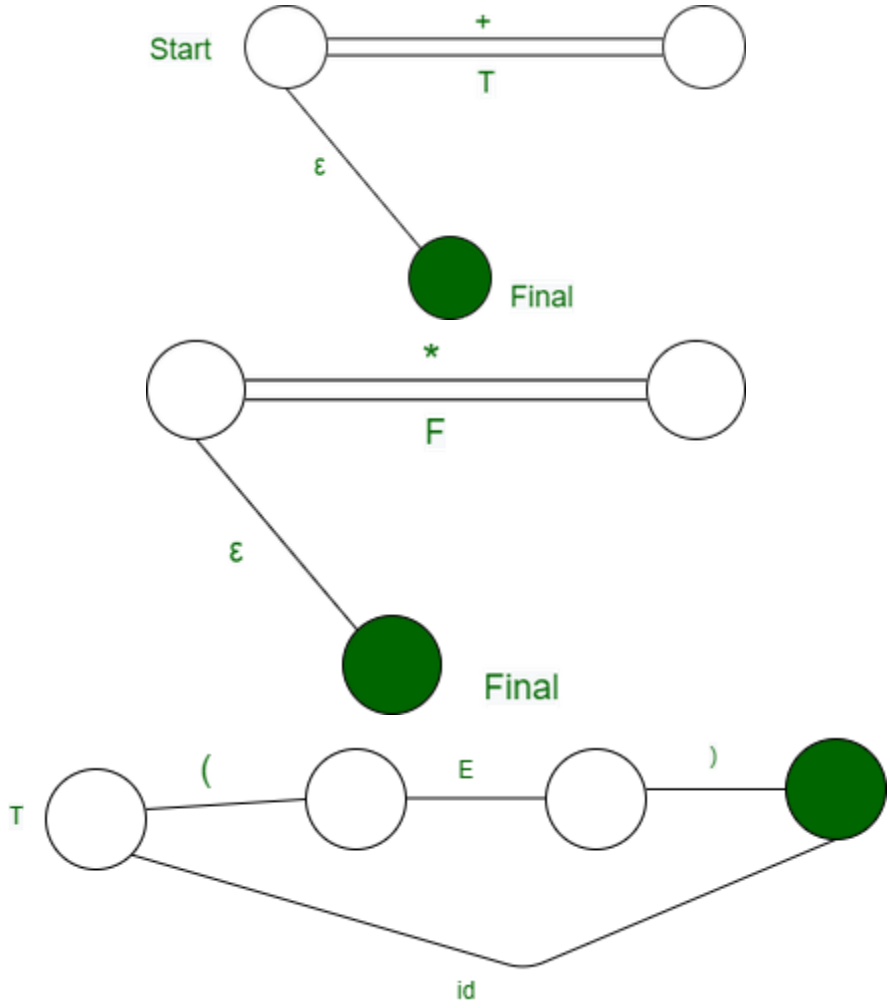
- T'->+TT'|ε



It can be optimized ahead by combining it with DFA for E->TT'



Accordingly, we optimize the other structures to produce the following DFA



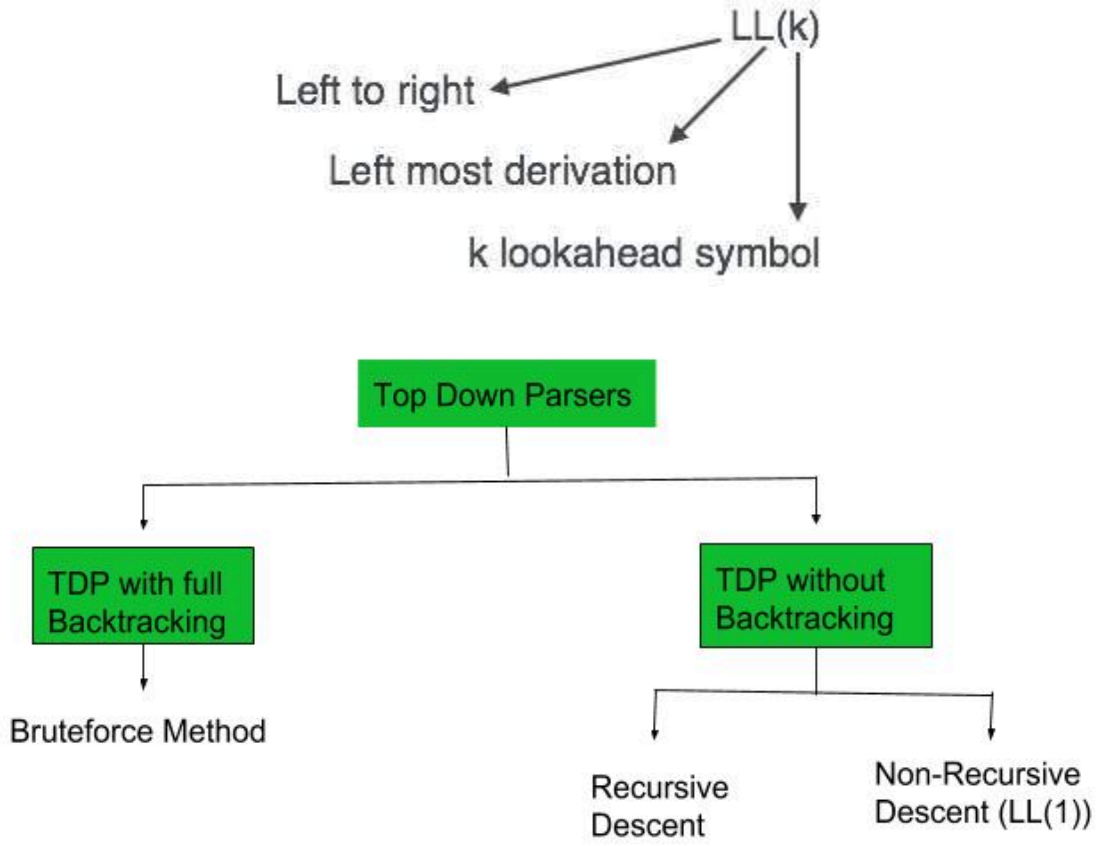
STEP 3:
Simulation on the input string.
Steps involved in the simulation procedure are:

1. Start from the starting state.
2. If a terminal arrives, consume it, move to the next state.
3. If a non-terminal arrives, go to the state of the DFA of the non-terminal and return on reaching up to the final state.
4. Return to actual DFA and Keep doing parsing.
5. If one completes reading the input string completely, you reach a final state, and the string is successfully parsed.

LL Parser

An LL Parser accepts LL grammar. L**L grammar is a subset of context-free grammar but with some restrictions to get the simplified version**, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

Here the 1st **L** represents that the scanning of the Input will be done from the **Left to Right manner** and the **second L shows that in this parsing technique**, we are going to use the **Leftmost Derivation Tree**. And finally, the **1 represents the number of look-aheads, which means how many symbols you will see when you want to make a decision.**



Model of LL Parser in Compiler Design:

The **three models** of LL parser in compiler design are the following:

- **Input:** This contains a string that will be parsed with the end-marker $.

- **Stack:** A predictive parser sustains a stack. It is a collection of grammar symbols with the dollar sign ($) at the bottom.
- **Parsing table:** M[A, S] is a two-dimensional array, where A is a non-terminal, and S is a terminal. With the entries in this table, it becomes effortless for the top-down parser to choose the production to be applied.

LL(1)

LL(1) parsing is a simple, powerful tool. It analyzes some grammar and compilers use it. **Its efficiency and error handling m m make it useful in programming language design.** But, it's not universal. LL(1) parsing tables are key to understanding language and compilers.

Conditions for an LL(1) Grammar

To construct a working LL(1) parsing table, a grammar must satisfy these conditions:

- No Left Recursion: Avoid recursive definitions like A -> A + b.
- Unambiguous Grammar: Ensure each string can be derived in only one way.
- Left Factoring: Make the grammar deterministic, so the parser can proceed without guessing.

Algorithm to Construct LL(1) Parsing Table

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as a terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Example 1:

The grammar is given below:

G --> SG'
G' --> +SG' | ε
S --> FS'
S' --> *FS' | ε
F --> id | (G)

Step 1: Each of the properties in step 1 is met by the grammar.

	First	Follow
G --> SG'	{ id, (}	{ $,) }
G' --> +SG' | ε	{ +, ε }	{ $,) }
S --> FS'	{ id, (}	{ +, $,) }
S' --> *FS' | ε	{ *, ε }	{ +, $,) }
F --> id | (G)	{ id, (}	{ *, +, $,) }

Step 2: Determine first() and follow().

Step 3: The parsing table for the above grammar will be:

	id	+	*	(	)	$
G	G -> SG’			G -> SG’		
G’		G’ -> +SG’			G’ -> ε	G’ -> ε
S	S -> FS’			S -> FS’		

S'		**S' -> ε**	**S' -> *FS'**		**S' -> ε**	**S' -> ε**
F	**F -> id**			**F -> (G)**		

As you can see, all of the null productions are grouped under that symbol's Follow set, while the remaining creations are grouped under its First.

Note: Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

Example 2: Consider the Grammar

S --> A | a

A --> a

Step 1: The grammar does not satisfy all properties in step 1, as the grammar is ambiguous. Still, let's try to make the parser table and see what happens

Step 2: Calculating first() and follow()

Find their First and Follow sets:

	First	**Follow**
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

Step 3: Make a parser table.

Parsing Table:

	a	**$**
S	S –> A, S –> a	
A	A –> a	

Here, we can see that there are two productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser.

Trick – Above grammar is ambiguous grammar. So the grammar does not satisfy the essential conditions. So we can say that this grammar is not feasible for LL(1) Parser even without making the parse table.

Example 3: Consider the Grammar

S -> (L) | a

L -> SL'

L' ->)SL' | ε

Step1: The grammar satisfies all properties in step 1

Step 2: Calculating first() and follow()

	First	**Follow**
S	(, a	$,)
L	(, a	)
L'	), ε	)

Step 3: Making a parser table

Parsing Table:

	(	**)**	**a**	**$**
S	S -> (L)		S -> a	
L	L -> SL'		L -> SL'	

L'		L'->(SL' L'->ε		

Here, we can see that there are two productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser. Although the grammar satisfies all the essential conditions in step 1, it is still not feasible for LL(1) Parser. We saw in example 2 that we must have these essential conditions and in example 3 we saw that those conditions are insufficient to be a LL(1) parser.

Advantages of Construction of LL(1) Parsing Table

- **Clear Decision-Making**: With an LL(1) parsing table, the parser can decide what to do by looking at just one symbol ahead. This makes it easy to choose the right rule without confusion or guessing.
- **Fast Parsing**: Since there's no need to go back and forth or guess the next step, LL(1) parsing is quick and efficient. This is useful for applications like compilers where speed is important.
- **Easy to Spot Errors**: The table helps identify errors right away. If the current symbol doesn't match any rule in the table, the parser knows there's an error and can handle it immediately.
- **Simple to Implement**: Once the table is set up, the parsing process is straightforward. You just follow the instructions in the table, making it easier to build and maintain.
- **Good for Predictive Parsing**: LL(1) parsing is often called "predictive parsing" because the table lets you predict the next steps based on the input. This makes it reliable for parsing programming languages and structured data.

Shift Reduce Parser

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the **parse tree is constructed from leaves(bottom) to the root(up)**. A more general form of the shift-reduce parser is the LR parser.
This **parser requires some data structures** i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

- **Shift:** This involves **moving symbols from the input buffer onto the stack.**
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. **RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.**
- **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Examples

Before coming to examples, we must remember some rules.

- **Rule 1:** If the priority of the incoming operator is higher than the operator's priority at the top of the stack, then we perform the **shift action.**
- **Rule 2:** If the priority of the incoming operator is equal to or less than the operator's priority at the top of the stack, then we perform the reduce action.

Now we will take some examples to understand shift-reduce parsing.
There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Example 1

Consider the following grammar.
E → E + E
E → E x E
E → id
Perform shift-reduce parsing for the input string “id + id x id”.
Solution: We will take a stack containing the $ symbol and place the given input string in the input buffer.

Note: The priority order of operators is: id > x > + > E > $.

Stack	Input Buffer	Parsing Action
$	id+idxid$	

The top of the stack contains $ while the current input symbol is the starting symbol, i.e., id. The priority of id > $. Thus, we will follow rule 1 and will perform the shift operation.

Stack	Input Buffer	Parsing Action
$	id+idxid$	Shift
$id	+idxid$	

Now, the symbol at the top of the stack is id, and the current input symbol is +. Since the priority of + < id, we will follow rule 2 and perform the reduce operation. For performing the reduce operation, we will check the grammar containing id on the right-hand side. Since E ⟶ id, we can reduce id by E.

Stack	Input Buffer	Parsing Action
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	

Next, the priority of + > E, perform the shift operation.

Stack	Input Buffer	Parsing Action

$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	

Next, the priority of id > +, perform the shift operation.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	

Next, the priority of x < id, perform the reduce operation. Since in the grammar E ⟶ id, we can reduce id by E.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id

$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id
$E+E	xid$	

Next, the priority of x > E, perform the shift operation.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id
$E+E	xid$	Shift
$E+Ex	id$	

Next, the priority of id > x, perform the shift operation.

Stack	**Input Buffer**	**Parsing Action**

$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id
$E+E	xid$	Shift
$E+Ex	id$	Shift
$E+Exid	$	

Next, the priority of $ < id, perform the reduce operation. Since in the grammar E ⟶ id, we can reduce id by E.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id

$E+E	xid$	Shift
$E+Ex	id$	Shift
$E+Exid	$	Reduce by E ⟶ id
$E+ExE	$	

Next, the priority of $ < E, we will perform the reduce operation. Since there is no grammar containing E, we will check the grammar containing xE. However, there is also no grammar containing xE, check the grammar containing ExE. We find that E ⟶ E x E. So, we will reduce ExE by E.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id
$E+E	xid$	Shift
$E+Ex	id$	Shift
$E+Exid	$	Reduce by E ⟶ id

$E+ExE	$	Reduce by E ⟶ E x E
$E+E	$	

Next, the priority of $ < E, we will perform the reduce operation. Since there is no grammar containing E, we will check the grammar containing +E. However, there is also no grammar containing +E, check the grammar containing E+E. We find that E ⟶ E + E. So, we will reduce E+E by E.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E ⟶ id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E ⟶ id
$E+E	xid$	Shift
$E+Ex	id$	Shift
$E+Exid	$	Reduce by E ⟶ id
$E+ExE	$	Reduce by E ⟶ E x E
$E+E	$	Reduce by E ⟶ E + E

$E	$	

Now, the stack only contains the starting symbol of the input string E and the input buffer is empty, i.e., includes the $ symbol. Hence, the parser will accept the string and will be completed.

Stack	**Input Buffer**	**Parsing Action**
$	id+idxid$	Shift
$id	+idxid$	Reduce by E → id
$E	+idxid$	Shift
$E+	idxid$	Shift
$E+id	xid$	Reduce by E → id
$E+E	xid$	Shift
$E+Ex	id$	Shift
$E+Exid	$	Reduce by E → id
$E+ExE	$	Reduce by E → E x E
$E+E	$	Reduce by E → E + E
$E	$	Accept

Example 2:

Consider the following grammar.
S → S + S
S → S - S
S → (S)
S → a1/a2/a3
Perform shift-reduce parsing for the input string “a1-(a2+a3)”.
Solution: The priority order of operators is:
a1/a2/a3 > () > +/- > S > $.

Stack	Input Buffer	Parsing Action
$	a1-(a2+a3)$	Shift
$a1	-(a2+a3)$	Reduce by S → a1
$S	-(a2+a3)$	Shift
$S-	(a2+a3)$	Shift
$S-(	a2+a3)$	Shift
$S-(a2	+a3)$	Reduce by S → a2
$S-(S	+a3)$	Shift
$S-(S+	a3)$	Shift
$S-(S+a3	)$	Reduce by S → a3
$S-(S+S	)$	Shift
$S-(S+S)	$	Reduce by S → S + S

$S-(S)	$	Reduce by S → (S)
$S-S	$	Reduce by S → S - S
$S	$	Accept

Example 3 – Consider the grammar

S –> (L) | a

L –> L , S | S

Perform Shift Reduce parsing for input string "(a, (a, a)) ".

Priority Order: a > () > , > S > $

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a

$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L, (L, S	)) $	Reduce L →L, S
$ (L, (L	)) $	Shift
$ (L, (L)	) $	Reduce S → (L)
$ (L, S	) $	Reduce L → L, S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

Advantages of Shift Reduce Parsing

- **Efficiency:** Shift-reduce parsing is generally more efficient than other parsing techniques, especially for larger and more complex grammars.
- **Bottom-up Parsing:** Shift-reduce parsers work in a bottom-up manner. It allows them to handle a broader class of grammar, including left-recursive grammar.
- **Generality:** Shift-reduce parsing is a general technique applicable to various programming languages and grammar. It makes it versatile and widely applicable.

- **Error Recovery:** Shift-reduce parsers can recover errors by detecting and handling errors encountered during parsing. It allows the process to continue with subsequent input.
- **Handling Ambiguity:** Shift-reduced parsers can effectively handle grammar and provide multiple parse trees or interpretations when required.

Disadvantages of Shift Reduce Parsing

- **Shift-Reduce Conflicts:** Shift-reduce parsing can encounter conflicts when multiple actions are applicable at a given parsing state, which results in shift-reduce conflicts.
- **Ambiguity Resolution:** While shift-reduce parsers can handle ambiguity, resolving ambiguities can be complex and require additional grammar modifications or priority rules.
- **Grammar Design Complexity:** Designing an efficient and unambiguous grammar suitable for shift-reduce parsing can be challenging and require revision and careful consideration.
- **Handling Left Recursion:** Shift-reduce parsers may struggle with left-recursive grammars, increasing parsing time and leading to inefficiencies.
- **Limited Lookahead:** Shift-reduce parsers typically have limited lookahead capabilities, making it challenging to make optimal parsing decisions in complex situations.

Operator precedence grammar.

A grammar that is **used to define mathematical operators** is called an **operator grammar** or **operator precedence grammar**.

Operator precedence parsing is a type of Shift Reduce Parsing. In operator precedence parsing, an operator grammar and an input string are fed as input to the operator precedence parser, which may generate a parse tree. Note that a parse tree will only be generated if the input string gets accepted.

In operator precedence parsing, the shift and reduce operations are done based on the priority between the symbol at the top of the stack and the current input symbol.

The operator precedence parser performs the operator precedence parsing using operator grammar.

Such grammars have the restriction that:

- No production has either an empty right-hand side (null productions)
- Two adjacent non-terminals in its right-hand side.

Operator precedence can only be established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

- a ⋗ b means that terminal "a" has the higher precedence than terminal "b".
- a ⋖ b means that terminal "a" has the lower precedence than terminal "b".
- a ≐ b means that the terminal "a" and "b" both have same precedence.

Operator Precedence Parser Algorithm :

1. If the front of input $ and top of stack both have $, it's done
   else
2. compare front of input b with ⋗
   if b! = '⋗'
   then push b
   scan the next input symbol
3. if b == '⋗'
   then pop till ⋖ and store it in a string S
   pop ⋖ also
   reduce the popped string
   if (top of stack) ⋖ (front of input)
   then push ⋖ S
   if (top of stack) ⋗ (front of input)
   then push S and goto 3

Example 1:

Consider the following grammar
T ⟶ T+T/TxT/id
With the help of the above grammar, parse the input string “id+idxid”.
Solution:
Step 1:

Stack	Relation	Input Buffer	Parsing Action
$		id+idxid$	

Step 2: The given grammar T ⟶ T+T/TxT/id is an operator grammar. So we don't need to modify it.
Step 3: Operator Precedence Table
For drawing the operator precedence table, we will take the terminals present in the grammar on the left-hand side and right-hand side.

- If the symbol on the left-hand side has higher precedence than the right-hand side symbol, insert the ⋗ symbol in the table.
- If the symbol on the left-hand side has lower precedence than the right-hand side symbol, insert the ⋖ symbol in the table.
- If the symbol on the left-hand side has equal precedence to the right-hand side symbol, insert nothing in the case of terminals, ⋗ symbol in the operators, and A in the case of the $ symbol in the table.

Thus, the operator precedence table for the given grammar will be-

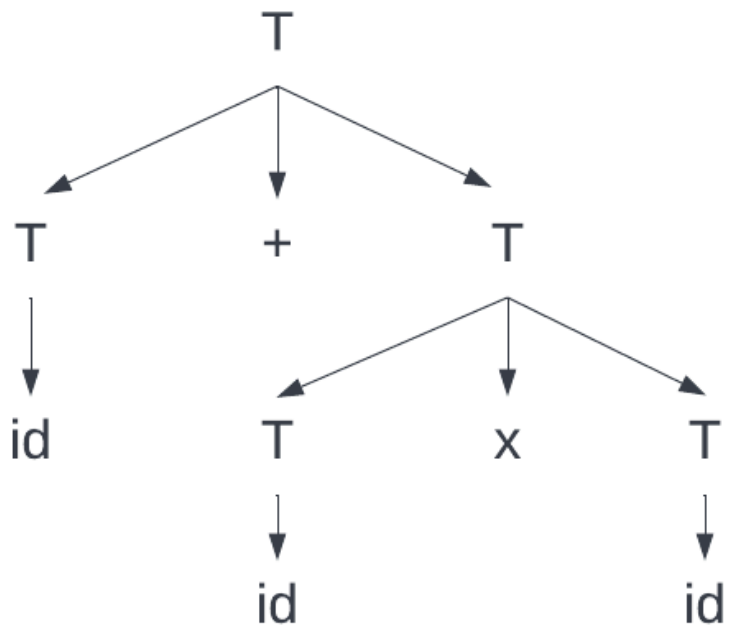
	+	x	id	$
+	⋗	⋖	⋖	⋗
x	⋗	⋗	⋖	⋗
id	⋗	⋗	—	⋗
$	⋖	⋖	⋖	A

Step 4: Parsing the input string.
We will use the operator precedence table to parse the given input string. We will compare the symbol at the top of the stack and the current input symbol. Based

on the precedence relation, we will perform either the shift operation or the reduced operation.
Thus, the parsing will be as follows.

Stack	Relation	Input Buffer	Parsing Action
$	⋖	id+idxid$	Shift
$id	⋗	+idxid$	Reduce by T ⟶ id
$T	⋖	+idxid$	Shift
$T+	⋖	idxid$	Shift
$T+id	⋗	xid$	Reduce by T ⟶ id
$T+T	⋖	xid$	Shift
$T+Tx	⋖	id$	Shift
$T+Txid	⋗	$	Reduce by T ⟶ id
$T+TxT	⋗	$	Reduce by T ⟶ T x T
$T+T	⋗	$	Reduce by T ⟶ T + T
$T	A	$	Accept

The given string is parsed and accepted.
Step 5: Generating the parse tree.
We will start from the bottom of the stack and generate the parse tree.



Example 2:

Consider the following grammar-
T ⟶ TAT | id
A ⟶ + | x
Construct the operator precedence parser and parse the string id + id x id.
Solution-
Step-01:
We convert the given grammar into operator precedence grammar.
The equivalent operator precedence grammar is-
T ⟶ T + T | T x T | id

Example 3:
Consider the following grammar-
S ⟶ (L) | a
L ⟶ L , S | S
Construct the operator precedence parser and parse the string (a , (a , a)).
Solution-
The terminal symbols in the grammar are { (,) , a , , }
We construct the operator precedence table as-

	a	(	)	,	$
a	>	>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

Operator Precedence Table
Parsing Given String-
Given string to be parsed is (a , (a , a)).
We follow the following steps to parse the given string-
Step-01:
We insert $ symbol at both ends of the string as-
$ (a , (a , a)) $
We insert precedence operators between the string symbols as-
$ < (< a > , < (< a > , < a >) >) > $
Step-02:

We scan and parse the string as-
$ < (< a > , < (< a > , < a >) >) > $
$ < (S , < (< a > , < a >) >) > $
$ < (S , < (S , < a >) >) > $
$ < (S , < (S , S) >) > $
$ < (S , < (L , S) >) > $
$ < (S , < (L) >) > $
$ < (S , S) > $
$ < (L , S) > $
$ < (L) > $
$ < S > $
$ $

Example 4: Consider the following grammar:

E ⟶ E+T/T

T ⟶ TxV/V

V ⟶ a/b/c/d

With the help of the above grammar, parse the input string “a+bxcxd”.

Solution:

Step 1:

Stack	Relation	Input Buffer	Parsing Action
$		a+bxcxd$	

Step 2: The grammar E ⟶ E+T/T, T ⟶ TxV/V, V ⟶ a/b/c/d is an operator grammar. So we don’t need to modify it.

Step 3: Operator Precedence Table

The operator precedence table for the given grammar will be-

	+	x	a	b	c	d	$
+	⋗	⋖	⋖	⋖	⋖	⋖	⋗
x	⋗	⋗	⋖	⋖	⋖	⋖	⋗

a	>	>	—	—	—	—	>
b	>	>	—	—	—	—	>
c	>	>	—	—	—	—	>
d	>	>	—	—	—	—	>
$	<	<	<	<	<	<	**A**

Step 4: Parsing the input string.

Stack	Relation	Input Buffer	Parsing Action
$	<	a+bxcxd$	Shift
$a	>	+bxcxd$	Reduce by V ⟶ a
$V	<	+bxcxd$	Shift
$V+	<	bxcxd$	Shift
$V+b	>	xcxd$	Reduce by V ⟶ b
$V+V	<	xcxd$	Shift
$V+Vx	<	cxd$	Shift
$V+Vxc	>	xd$	Reduce by V ⟶ c
$V+VxV	>	xd$	Reduce by T ⟶ V

$V+VxT	>	xd$	Reduce by E → T
$V+VxE	>	xd$	Error

Since no production contains E on its right-hand side, this parsing will lead to an error.

Thus, the given input string cannot be parsed using the precedence operator by the given grammar. So, we cannot generate a parse tree.

Advantages –

1. It can easily be constructed by hand.
2. It is simple to implement this type of parsing.

Efficient parsing: Precedence parsers can parse operator grammars in linear time, making them much more efficient than other parsing techniques.
Easy to implement: Operator grammars are relatively easy to define and implement, making them a popular choice for describing the syntax of programming languages.
Improved readability: Using operator precedence parsing can make the syntax of a programming language more readable and easier to understand, as operators can be grouped according to their precedence levels.
Error detection: Precedence parsers can detect certain types of errors, such as syntax errors and operator precedence errors, which can help programmers to debug their code more easily.
Flexibility: Operator grammars and precedence parsers are very flexible, allowing for a wide range of syntax structures to be described, including those with complex operator precedence rules.
Modular design: Precedence parsers can be designed to work with other parsing techniques, such as top-down and bottom-up parsers, allowing for a modular design that can be easily extended or modified.
Disadvantages –

1. It is hard to handle tokens like the minus sign (-), which has two different precedence (depending on whether it is unary or binary).
2. It is applicable only to a small class of grammars.

LR Parser

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compilers 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 production by building up from the leaves.** LR parsers are the most powerful parser of all deterministic parsers in practice.

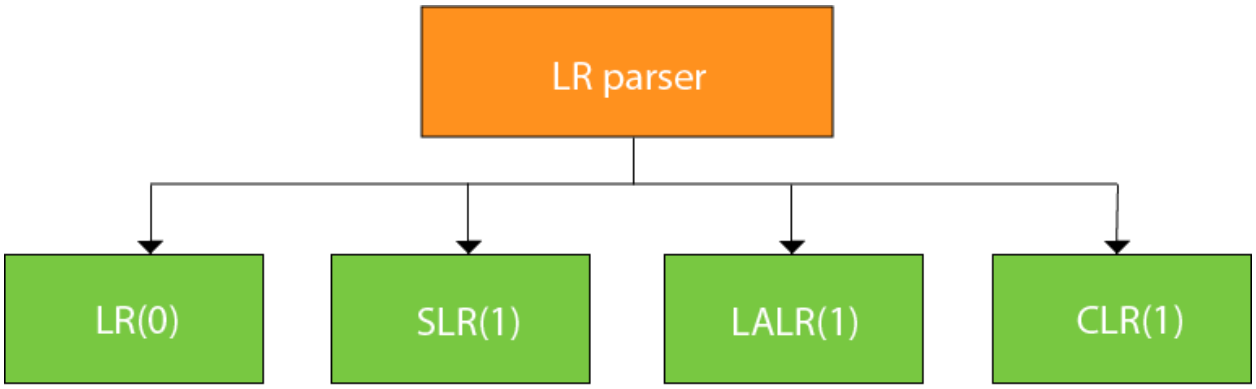
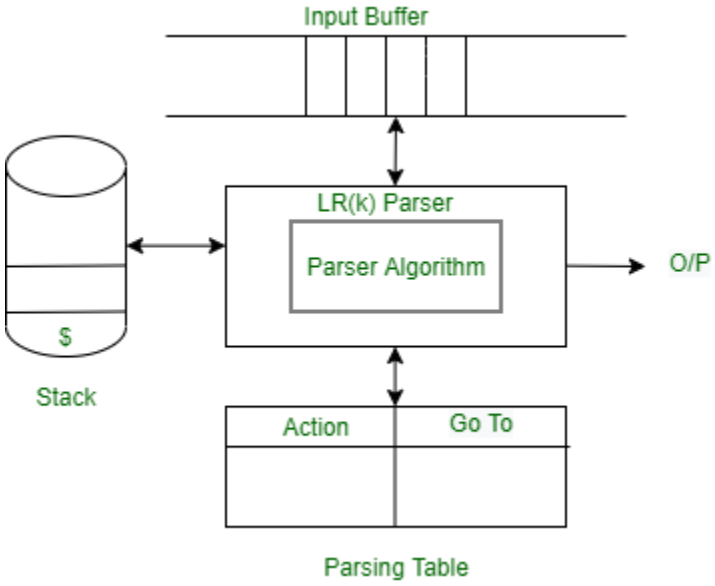


Fig: Types of LR parser



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.

1. The stack is empty, and we are looking to reduce the rule by S'→S$.
2. Using a "." in the rule represents how many of the rules are already on the stack.
3. 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.

1. The first item from the given grammar rules adds itself as the first closed set.
2. 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.
3. Repeat steps (B) and (C) for new items added under (B).

LR parser algorithm :

LR Parsing algorithm is the same for all the parsers, but the parsing table is different for each parser. It consists following components as follows.

1. **Input Buffer –**

   It contains the given string, and it ends with a $ symbol.
2. **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 the action table as follows.

1. 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.

2. Reduce Action- The number ***m*** is written to the output stream.
3. The symbol ***m*** mentioned in the left-hand side of rule ***m*** says that state is removed from the stack.
4. 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.

An accept - the string is accepted
No action - a syntax error is reported

- **Note –**
  The go-to table indicates which state should proceed.
- Canonical Collection of LR(0) items
- An LR (0) item is a production G with a dot at some position on the right side of the production.
- LR(0) items are useful to indicate how much of the input has been scanned up to a given point in the process of parsing.
- In the LR (0), we place the reduced node in the entire row.

Example

Given grammar:

S → AA
A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •S
S → •AA
A → •aA
A → •b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S
S → •AA

Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S

S → •AA

A → •aA

A → •b

I1= Go to (I0, S) = closure (S` → S•) = S` → S•

Here, the Production is reduced so close the State.

I1= S` → S•

I2= Go to (I0, A) = closure (S → A•A)

Add all productions starting with A in to I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2 =S→A•A

A → •aA

A → •b

Go to (I2,a) = Closure (A → a•A) = (same as I3)

Go to (I2, b) = Closure (A → b•) = (same as I4)

I3= Go to (I0,a) = Closure (A → a•A)

Add productions starting with A in I3.

A → a•A

A → •aA

A → •b

Go to (I3, a) = Closure (A → a•A) = (same as I3)

Go to (I3, b) = Closure (A → b•) = (same as I4)

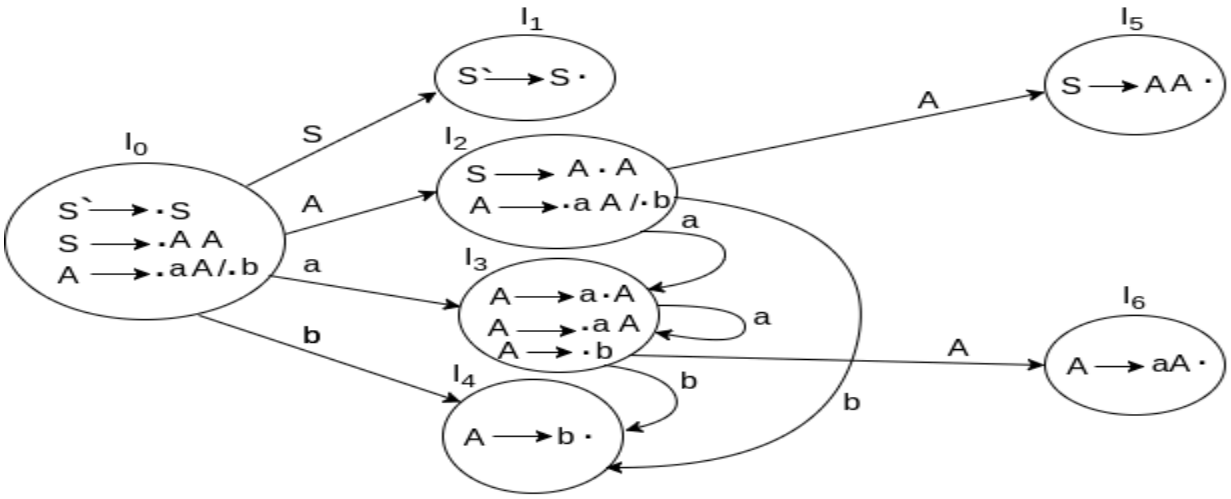
I4= Go to (I0, b) = closure (A → b•) = A → b•

I5= Go to (I2, A) = Closure (S → AA•) = SA → A•

I6= Go to (I3, A) = Closure (A → aA•) = A → aA•

Drawing DFA:

The DFA contains the 7 states I0 to I6



LR(0) Table

- If a state is going to some other state on a terminal then it corresponds to a shift move.
- If a state is going to some other state on a variable then it corresponds to going to move.
- If a state contains the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Explanation:

- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.
- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2and I3on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.

- I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as production number.

Productions are numbered as follows:

S → AA ... (1)
A → aA ... (2)
A → b ... (3)

- I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR (1) Parsing

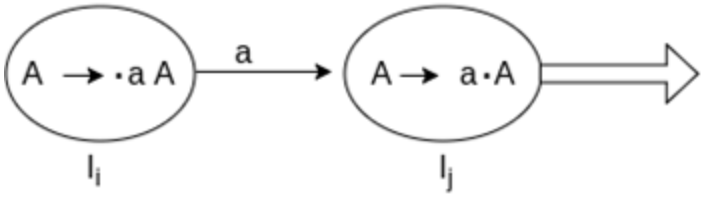
- SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table.To construct SLR (1) parsing table, we use canonical collection of LR (0) item.
- In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table

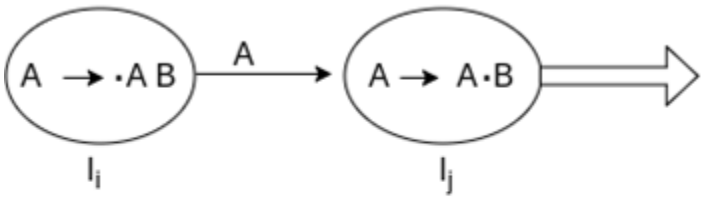
SLR (1) Table Construction

- The steps which use to construct SLR (1) Table is given below:
- If a state (Ii) is going to some other state (Ij) on a terminal then it corresponds to a shift move in the action part.



States	Action		Go to
	a	$	A
I_i I_j	Sj		

- If a state (Ii) is going to some other state (Ij) on a variable then it correspond to go to move in the Go to part.

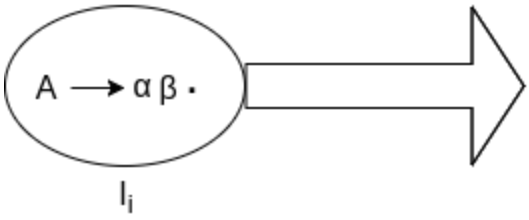


States	Action		Go to
	a	$	A
I_i I_j			j

If a state (Ii) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

Example

S -> •Aa
A->αβ•
Follow(S) = {$}
Follow (A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

SLR (1) Grammar

E → E + T | T
T → T * F | F
F → id

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •E)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •E
E → •E + T
E → •T

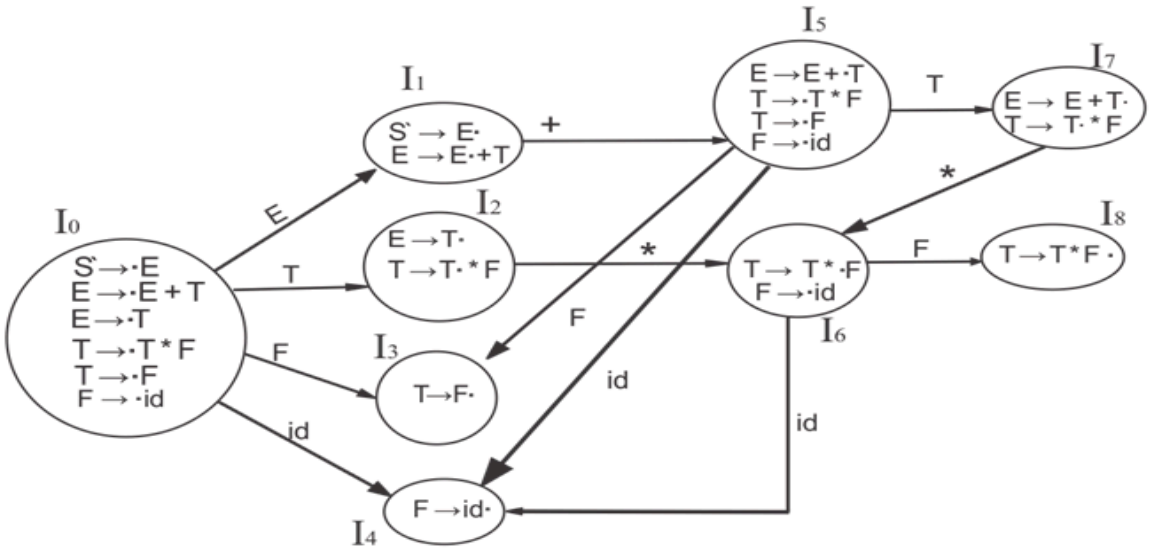
Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •E
E → •E + T

E → •T
T → •T * F
T → •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)
I2= Go to (I0, T) = closure (E → T•T, T• → * F)
I3= Go to (I0, F) = Closure (T → F•) = T → F•
I4= Go to (I0, id) = closure (F → id•) = F → id•
I5= Go to (I1, +) = Closure (E → E +•T)
Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes
I5 = E → E +•T
T → •T * F
T → •F
F → •id
Go to (I5, F) = Closure (T → F•) = (same as I3)
Go to (I5, id) = Closure (F → id•) = (same as I4)
I6= Go to (I2, *) = Closure (T → T * •F)
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes
I6 = T → T * •F
F → •id
Go to (I6, id) = Closure (F → id•) = (same as I4)
I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Drawing DFA:



SLR (1) Table

States	Action				Go to		
	id	+	*	**$**	**E**	**T**	**F**
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Explanation:

First (E) = First (E + T) ∪ First (T)
First (T) = First (T * F) ∪ First (F)
First (F) = {id}
First (T) = {id}
First (E) = {id}
Follow (E) = First (+T) ∪ {$} = {+, $}
Follow (T) = First (*F) ∪ First (F)
= {*, +, $}
Follow (F) = {*, +, $}

- I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
- I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4
- I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5
- I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
- I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

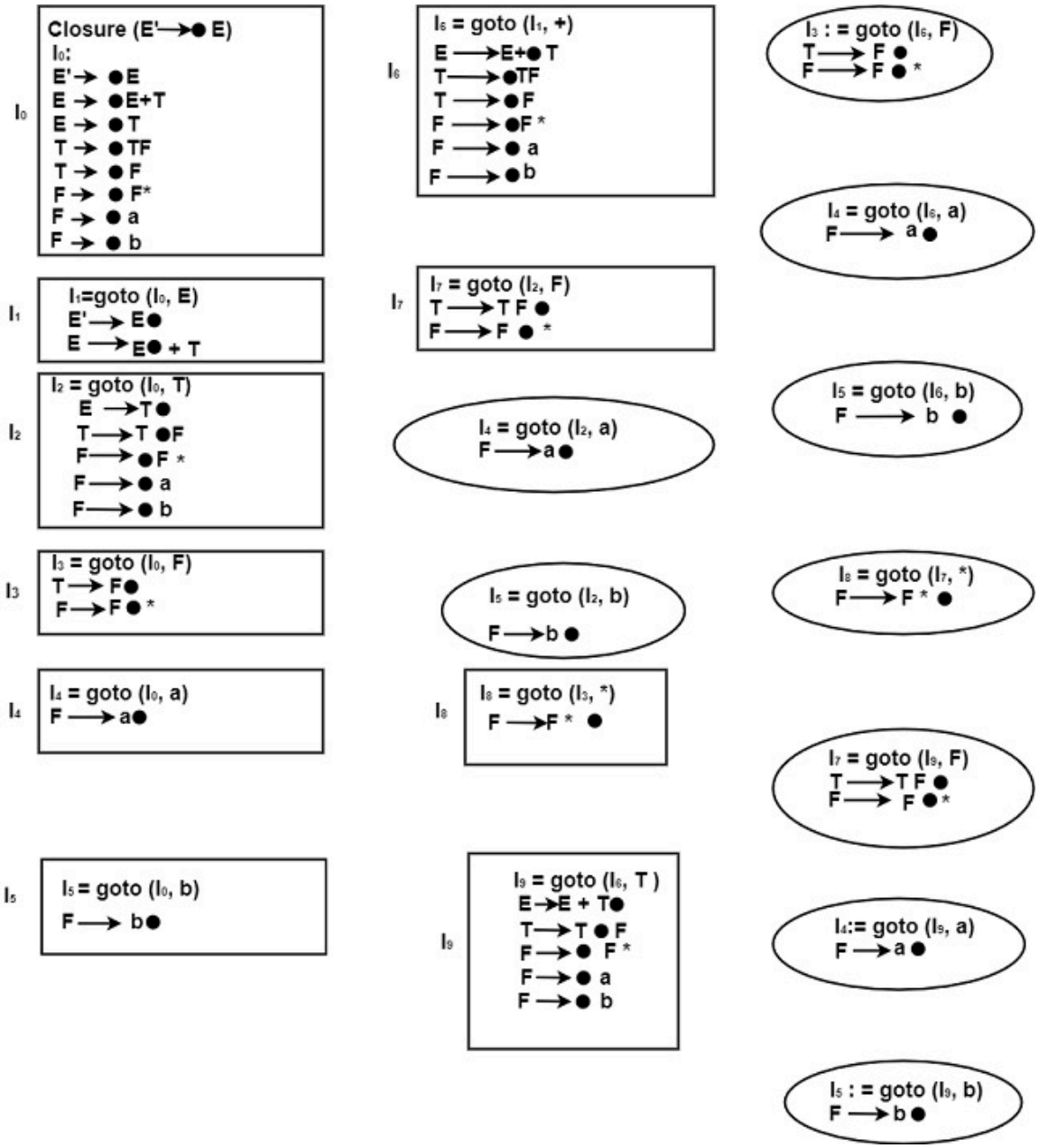
Construct the SLR Parsing table for the following grammar. Also, Parse the input string a * b + a.

E → E + T|T
T → TF|F
F → F*|a|b.

Solution

Step1 − Construct the augmented grammar and number the productions.
(0) E′ → E
(1) E → E + T
(2) E → T
(3) T → TF
(4) T → F
(5) F → F ∗
(6) F → a
(7) F → b.

Step2 − Find closure & goto Functions to construct LR (0) items.
Box represents the New states, and the circle represents the Repeating State.



Computation of FOLLOW

We can find out

FOLLOW(E) = {+, $}

FOLLOW(T) = {+, a, b, $}

FOLLOW(F) = {+,*, a, b, $}

LR Parsing Table

State	Action					goto		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

Parsing for Input String a * b + a –

Stack	Input String	Action
0	a * b + a $	Shift
0 a 4	* b + a $	Reduce by F → a.
0 F 3	* b + a $	Shift
0 F 3 * 8	b + a $	Reduce by F → F *
0 F 3	b + a $	Reduce by T → F
0 T 2	b + a $	Shift

0 T 2 b 5	+a $	Reduce by F → b
0 T 2 F 7	+a $	Reduce by T → TF
0 T 2	+a $	Reduce by E → T
0 E 1	+a $	Shift
0 E 1 + 6	a$	Shift
0 E 1 + 6 a 4	$	Reduce by F → a
0 E 1 + 6 F 3	$	Reduce by T → F
0 E 1 + 6 T 9	$	Reduce by E → E + T
0 E 1	$	Accept

CLR (1) Parsing

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.
In the CLR (1), we place the reduce node only in the lookahead symbols.
Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

LR (1) item

- LR (1) item is a collection of LR (0) items and a look ahead symbol.
- LR (1) item = LR (0) item + look ahead
- The look ahead is used to determine where we place the final item.
- The look ahead always add $ symbol for the argument production.

Example

CLR (1) Grammar

S → AA

A → aA

A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $

S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

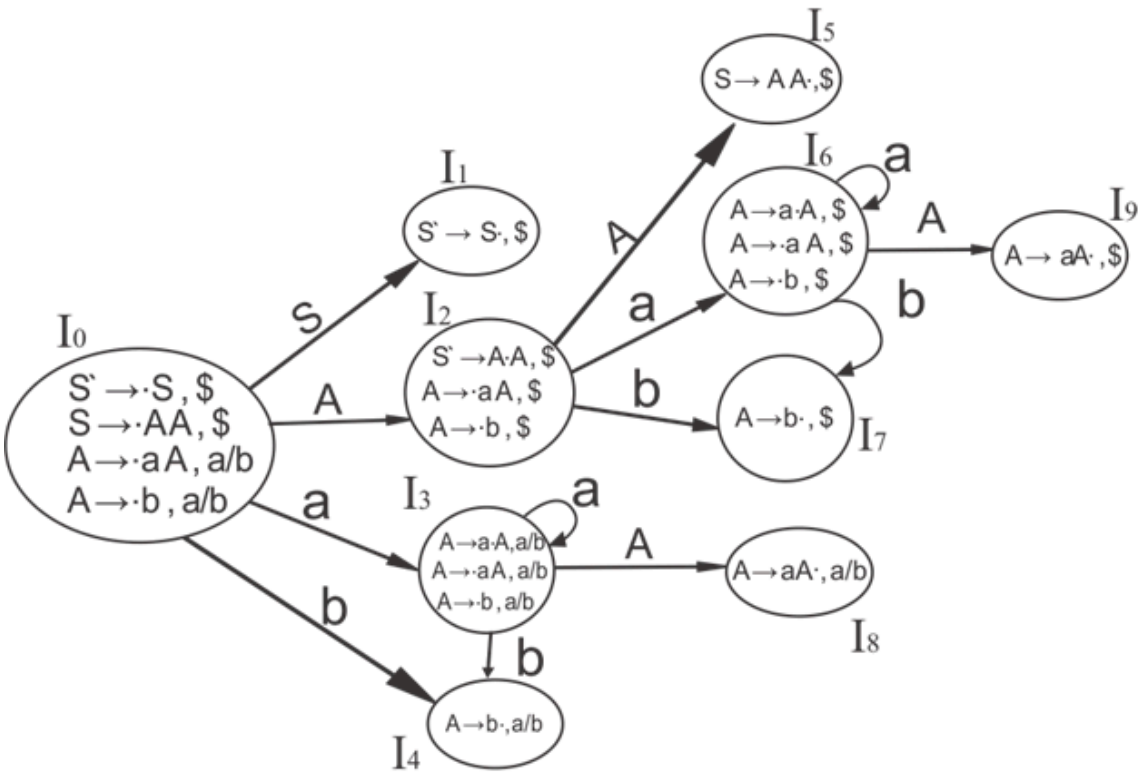
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



CLR (1) Parsing table:

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Productions are numbered as follows:

S → AA ... (1)
A → aA ....(2)
A → b ... (3)

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives (A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.
I5 contains the final item which drives (S → AA•, $), so action {I5, $} = R1.
I7 contains the final item which drives (A → b•,$), so action {I7, $} = R3.
I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.
I9 contains the final item which drives (A → aA•, $), so action {I9, $} = R2.

LALR (1) Parsing

- LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
- In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
- LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

LALR (1) Grammar

S → AA
A → aA
A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)
Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes
I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)
Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)
I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)
Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes
I6 = A → a•A, $
A → •aA, $
A → •b, $
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)
I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $
If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b
}
I6= { A → a•A, $
A → •aA, $
A → •b, $
}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$
A → •aA, a/b/$
A → •b, a/b/$
}

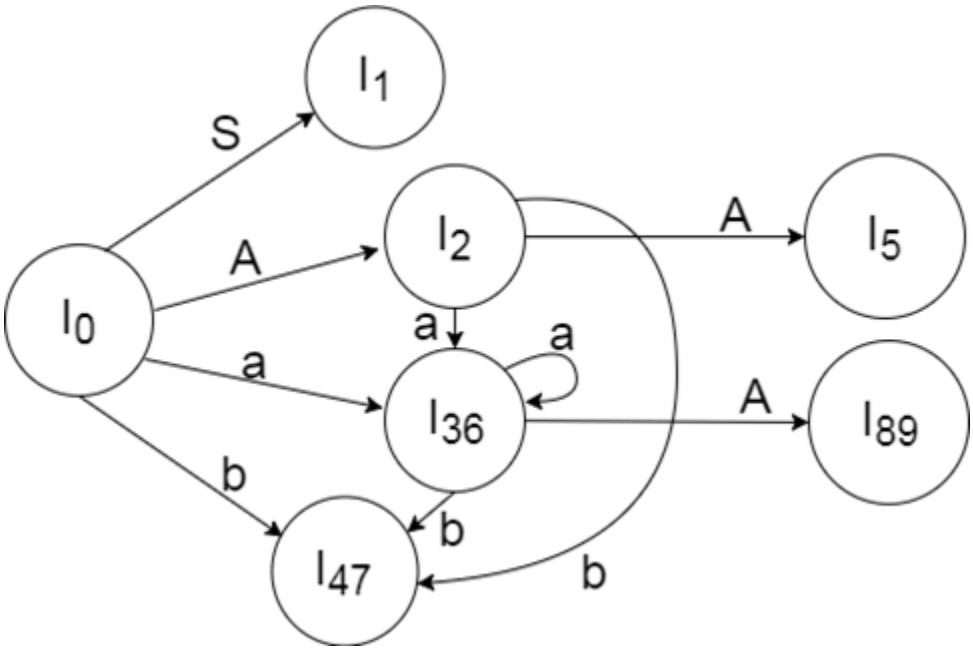
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	S	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Difference Between Bottom-Up and Top-Down Parser

Feature	Top-down Parsing	Bottom-up Parsing
Direction	Builds tree from **root to leaves**.	Builds tree from **leaves to root**.
Derivation	Uses **leftmost derivation**.	Uses **rightmost derivation in reverse**.
Efficiency	Can be slower, especially with backtracking.	More efficient for complex grammars.
Example Parsers	Recursive descent, LL parser.	Shift-reduce, LR parser.