UNIT-II Syntax Analysis (Parser):

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar *CFG*, which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- A set of **non-terminals** *V*. Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** Σ. Terminals are the basic symbols from which strings are formed.
- A set of **productions** *P*. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol *S*; from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal *initiallythestartsymbol* by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular

Expression. That is, L = { w | w = w^R } is not a regular language. But it can be described by means of CFG, as illustrated below:

```
G = ( V, Σ, P, S )
```

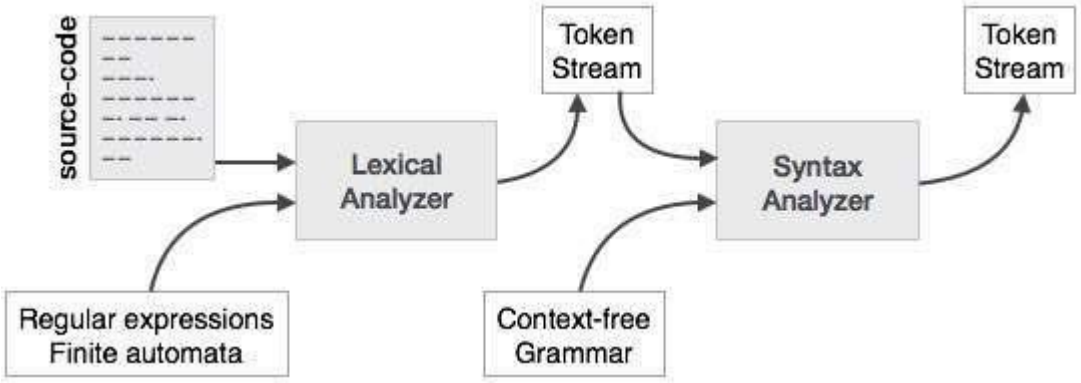
Where:

```
V = { Q, Z, N }
Σ = { 0, 1 }
P = { Q → Z | Q → N | Q → ε | Z → 0Q0 | N → 1Q1 }
S = { Q }
```

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code *tokenstream* against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

```
E → E + E
E → E * E
E → id
```

Input string: id + id * id

The left-most derivation is:

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

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

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

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

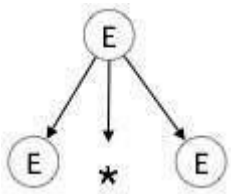
We take the left-most derivation of a + b * c

The left-most derivation is:

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

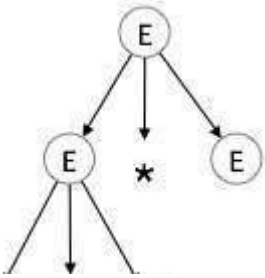
Step 1:

E → E * E



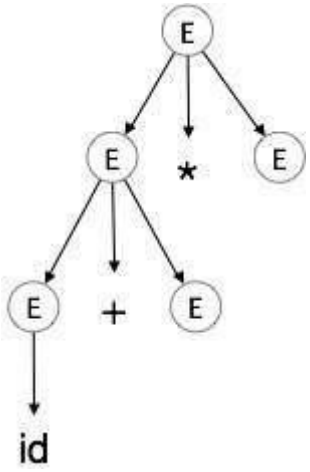
Step 2:

E → E + E * E



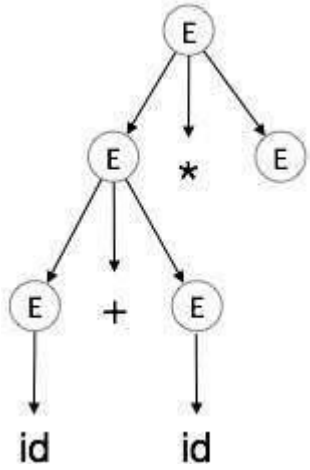
Step 3:

E → id + E * E



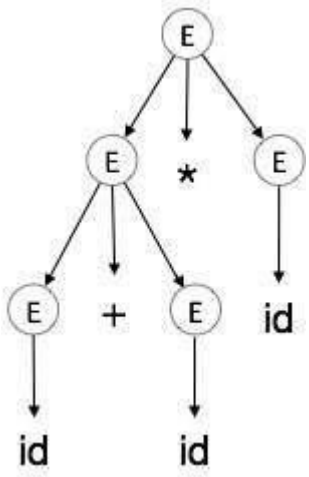
Step 4:

E → id + id * E



Step 5:

E → id + id * id



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

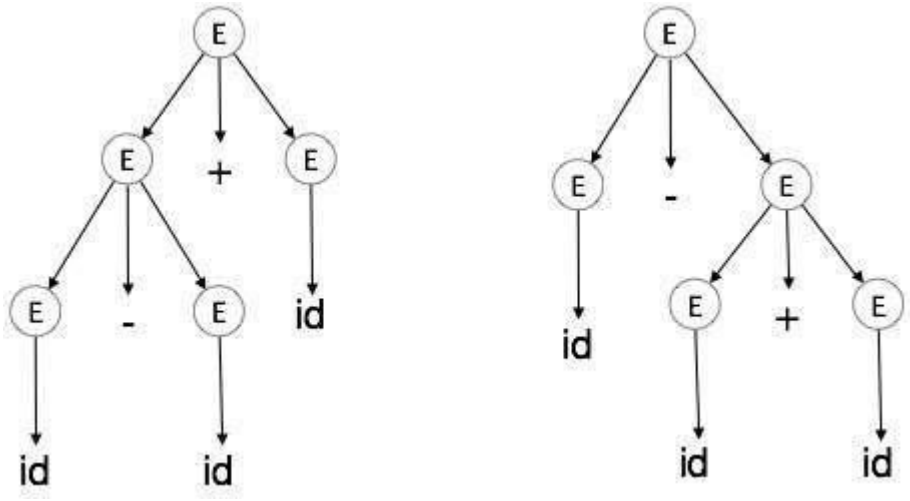
Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree *leftorrightderivation* for at least one string.

Example

```
E → E + E
E → E − E
E → id
```

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

```
id op id op id
```

it will be evaluated as:

```
(id op id) op id
```

For example, *id* + *id* + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

```
id op (id op id)
```

For example, id ^ id^id

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to 2 + 3 *4 and another corresponding to 2+3 ∗ 4. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * *multiplication* has precedence over + *addition*, so the expression 2+3*4 will always be interpreted as:

```
2 + (3 * 4)
```

These methods decrease the chances of ambiguity in a language or its grammar.

Left Recursion

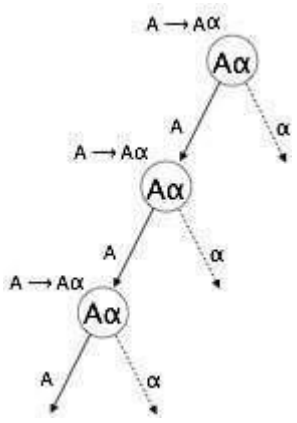
A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

Example:

```
(1) A => Aα | β

(2) S => Aα | β
    A => Sd
```

1 is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

2 is an example of indirect-left recursion.



A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

```
A => Aα | β
```

is converted into following productions

```
A => βA'
A'=> αA' | ε
```

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

```
START

Arrange non-terminals in some order like A1, A2, A3,…, An

   for each i from 1 to n
      {
      for each j from 1 to i-1
         {
         replace each production of form Ai ⟹Aj
         with Ai ⟹ δ1  | δ2 | δ3 |…| 
         where Aj ⟹ δ1 | δ2|…| δn  are current Aj productions
         }
      }
   eliminate immediate left-recursion

END
```

Example

The production set

```
S => Aα | β
A => Sd
```

after applying the above algorithm, should become

```
S => Aα | β
A => Aαd | βd
```

and then, remove immediate left recursion using the first technique.

```
A  => βdA'
A' => αdA' | ε
```

Now none of the production has either direct or indirect left recursion.

Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

```
A ⟹ αβ  | α | …
```

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal *ornon − terminal*. To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new

productions.

Example

The above productions can be written as

```
A => αA'
A'=> β | □ | …
```

Now the parser has only one production per prefix which makes it easier to take decisions.

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

```
α → t β
```

That is α derives t *terminal* in the very first position. So, t ∈ FIRSTα.

Algorithm for calculating First set

Look at the definition of FIRSTα set:

- if α is a terminal, then FIRSTα = { α }.
- if α is a non-terminal and α → Ɛ is a production, then FIRSTα = { Ɛ }.
- if α is a non-terminal and α → □1 □2 □3 … □n and any FIRST□ contains t then t is in FIRSTα.

First set can be seen as:

$$\text{FIRST}(\alpha) = \{ t \mid \alpha \xrightarrow{*} t\beta \} \cup \{ \varepsilon \mid \alpha \xrightarrow{*} \varepsilon \}$$

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Algorithm for calculating Follow set:

- if α is a start symbol, then FOLLOW = $
- if α is a non-terminal and has a production α → AB, then FIRST*B* is in FOLLOW*A* except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW*A* is in FOLLOWα.

Follow set can be seen as: FOLLOWα = { t | S *αt*}

Limitations of Syntax Analyzers

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.

Context-free languages and grammars

Push-down automata:

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is −

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

LL(1) grammars:

LL(1) Parsing: 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 Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

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 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.

Now, let's understand with an example.

Example 1: Consider the Grammar:

```
E --> TE'

E' --> +TE' | ε

T --> FT'

T' --> *FT' | ε

F --> id | (E)


*ε denotes epsilon
```

Step 1: The grammar satisfies all properties in step 1.

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

Find their First and Follow sets:

	First	Follow
E –> TE'	{ id, (}	{ $,) }
E' –> +TE'/ε	{ +, ε }	{ $,) }
T –> FT'	{ id, (}	{ +, $,) }
T' –> *FT'/ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Step 3: Make a parser table.

Now, the LL(1) Parsing Table is:

	id	+	*	(	)	$
E	E –> TE'			E –> TE'		
E'		E' –> +TE'			E' –> ε	E' –> ε
T	T –> FT'			T –> FT'		
T'		T' –> ε	T' –> *FT'		T' –> ε	T' –> ε
F	F –> id			F –> (E)		

As you can see that all the null productions are put under the Follow set of that symbol and all the remaining productions lie under the First of that symbol.

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

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

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

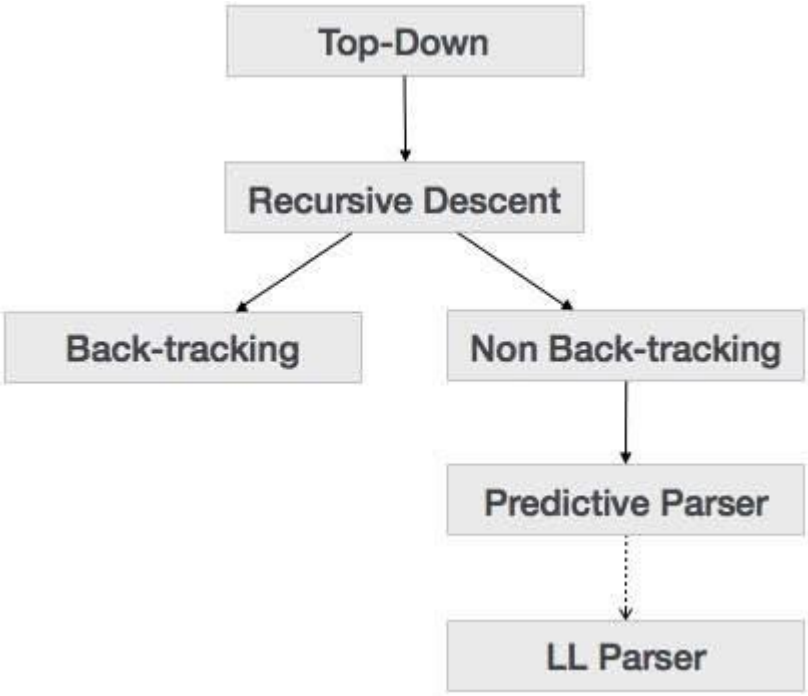
Step 3: Making a parser table
Parsing Table:

	(	)	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.

TOP DOWN PARSING:

The types of top-down parsing are depicted below:



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

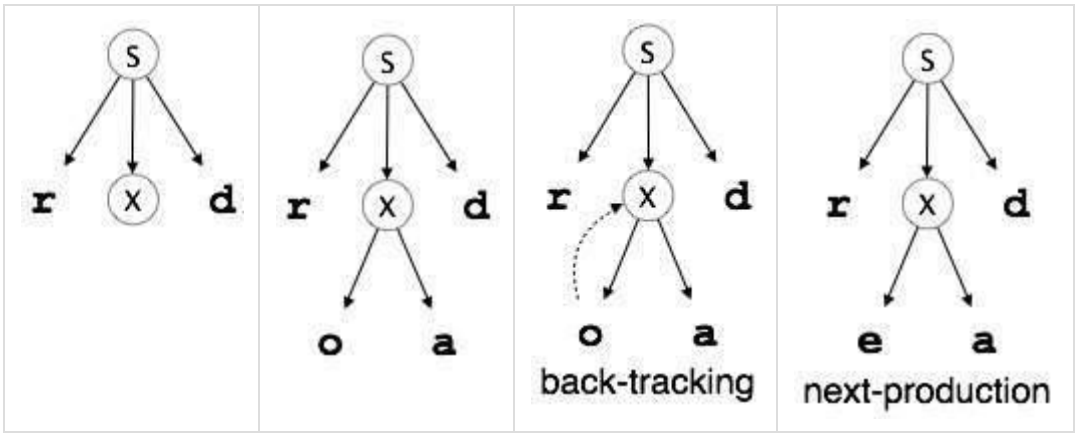
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd X → oa | ea Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

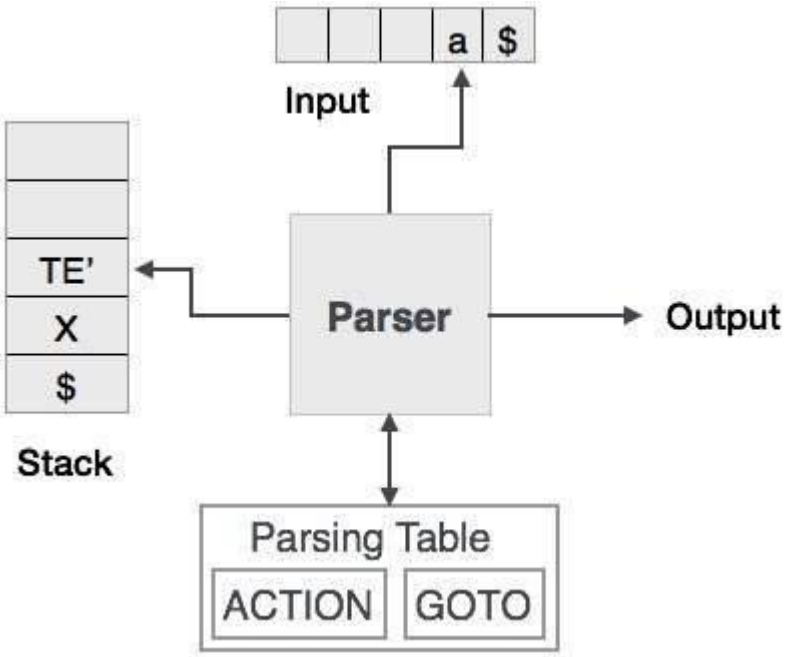
Now the parser matches all the input letters in an ordered manner. The string is accepted.



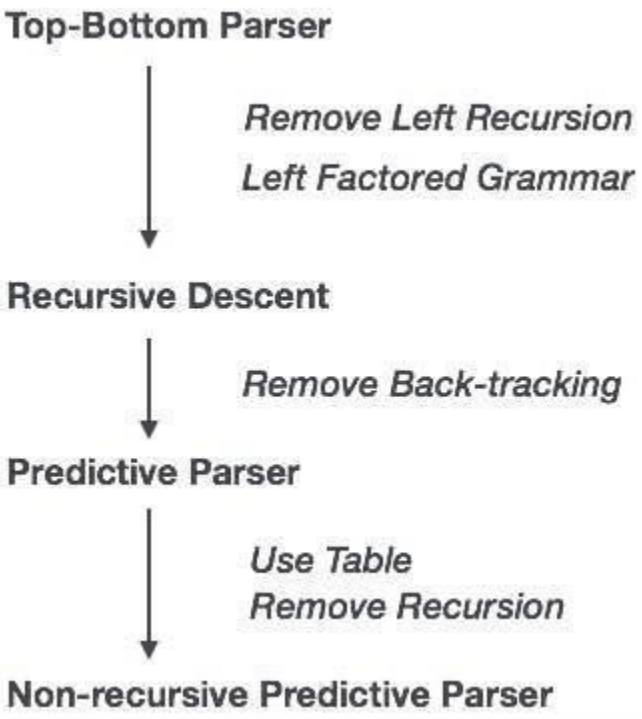
Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

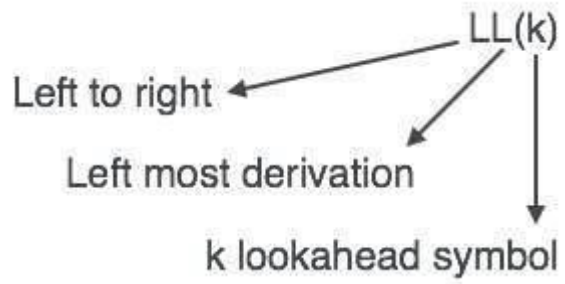


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

An LL Parser accepts LL grammar. LL 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.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

```
Input:
   string ω
   parsing table M for grammar G

Output:
   If ω is in L(G) then left-most derivation of ω,
   error otherwise.

Initial State : $S on stack (with S being start symbol)
   ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat
   let X be the top stack symbol and a the symbol pointed by ip.

   if X∈ Vt or $
      if X = a
```

```
      POP X and advance ip.
    else
      error()
    endif

  else        /* X is non-terminal */
    if M[X,a] = X → Y1, Y2,... Yk
      POP X
      PUSH Yk, Yk-1,... Y1 /* Y1 on top */
      Output the production X → Y1, Y2,... Yk
    else
      error()
    endif
  endif
until X = $  /* empty stack */
```

A grammar G is LL(1) if A → α | β are two distinct productions of G:

- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A)

Operator grammars:

A grammar that is used to define mathematical operators is called an **operator grammar** or **operator precedence grammar**. Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side. **Examples –** This is an example of operator grammar:

```
E->E+E/E*E/id
```

However, the grammar given below is not an operator grammar because two non-terminals are adjacent to each other:

```
S->SAS/a

A->bSb/b
```

We can convert it into an operator grammar, though:

```
S->SbSbS/SbS/a

A->bSb/b
```

Operator precedence parser – An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in any parser except operator precedence parser. There are two methods for determining what precedence relations should hold between a pair of terminals:

1. Use the conventional associativity and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗ a ⋖ b** This means a "yields precedence to" b. **a ⋗ b** This means a "takes precedence over" b. **a ≐ b** This means a "has same precedence as" b.

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Figure – Operator precedence relation table

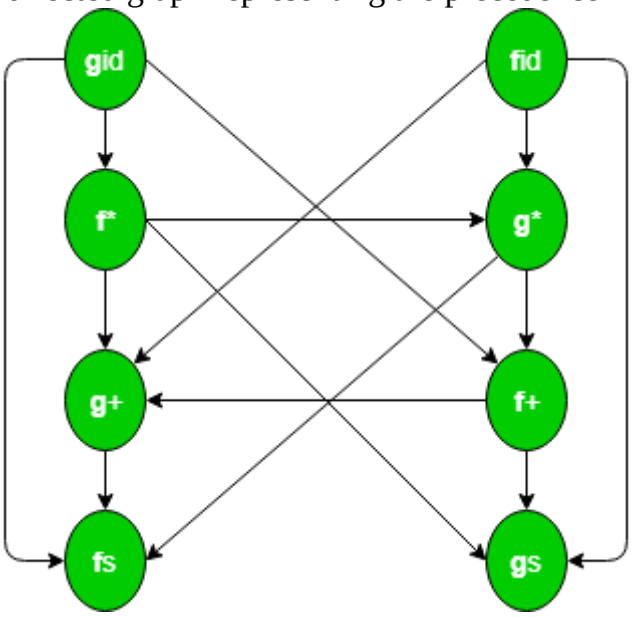
for grammar E->E+E/E*E/id There is not given any relation between id and id as id will not be compared and two variables can not come side by side. There is also a disadvantage of this table – if we have n operators then size of table will be n*n and complexity will be $0(n^2)$. In order to decrease the size of table, we use **operator function table**. Operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way. Operator precedence parsers use **precedence functions** that map terminal symbols to integers, and the precedence relations between the symbols are implemented by numerical comparison. The parsing table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers. We select f and g such that:

1. f(a) < g(b) whenever a yields precedence to b
2. f(a) = g(b) whenever a and b have the same precedence
3. f(a) > g(b) whenever a takes precedence over b

Example – Consider the following grammar:

```
E -> E + E/E * E/( E )/id
```

This is the directed graph representing the precedence function:



Since there is no cycle in the graph, we can make this function table:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

```
fid -> g* -> f+ ->g+ -> f$
```

```
gid -> f* -> g* ->f+ -> g+ ->f$
```

Size of the table is **2n**. One disadvantage of function tables is that even though we have blank entries in relation table we have non-blank entries in function table. Blank entries are also called error. Hence error detection capability of relation table is greater than function table.

- C

```
#include<stdlib.h>

#include<stdio.h>

#include<string.h>
```

```
// function f to exit from the loop
// if given condition is not true
void f()
{
    printf("Not operator grammar");
    exit(0);
}

void main()
{
    char grm[20][20], c;

    // Here using flag variable,
    // considering grammar is not operator grammar
    int i, n, j = 2, flag = 0;

    // taking number of productions from user
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%s", grm[i]);

    for (i = 0; i < n; i++) {
        c = grm[i][2];
```

```
        while (c != '&#092;&#048;') {



            if (grm[i][3] == '+' || grm[i][3] == '-'

                || grm[i][3] == '*' || grm[i][3] == '/')



                flag = 1;



            else {



                flag = 0;

                f();

            }



            if (c == '$') {

                flag = 0;

                f();

            }



            c = grm[i][++j];

        }

    }



    if (flag == 1)

        printf("Operator grammar");

}
```

```
Input :3
A=A*A
B=AA
A=$

Output : Not operator grammar

Input :2
A=A/A
B=A+A

Output : Operator grammar
```

$ is a null production here which are also not allowed in operator grammars. **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.

Features:

Operator Grammar:
Operators with Precedence: An operator grammar includes operators with different levels of precedence and associativity. The grammar specifies the syntactic structure of expressions, which can be used to derive parse trees for expressions.
Priority and Associativity: Operator grammar provides a way to define the priority and associativity of operators, which is essential for parsing expressions correctly.
Easy to Read: Operator grammar is easy to read and understand, making it a popular choice for designing the syntax of programming languages.

Ambiguity Resolution: Operator grammar helps to reduce ambiguity in expressions by specifying the order in which operators are applied.

Precedence Parser:

Efficient Parsing: Precedence parser can parse expressions efficiently and without requiring backtracking, which makes it faster than other parsing techniques.

Bottom-Up Parsing: Precedence parser is a bottom-up parser that can handle operator precedence and associativity.

Error Recovery: Precedence parser can perform error recovery by detecting errors in expressions and skipping over them to continue parsing the remaining input.

Flexible: Precedence parser can handle a wide range of grammars and can be easily extended to handle new operators or operators with different precedence levels.

LR(0):

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.

Q. Construct an LR parsing table for the given context-free grammar –

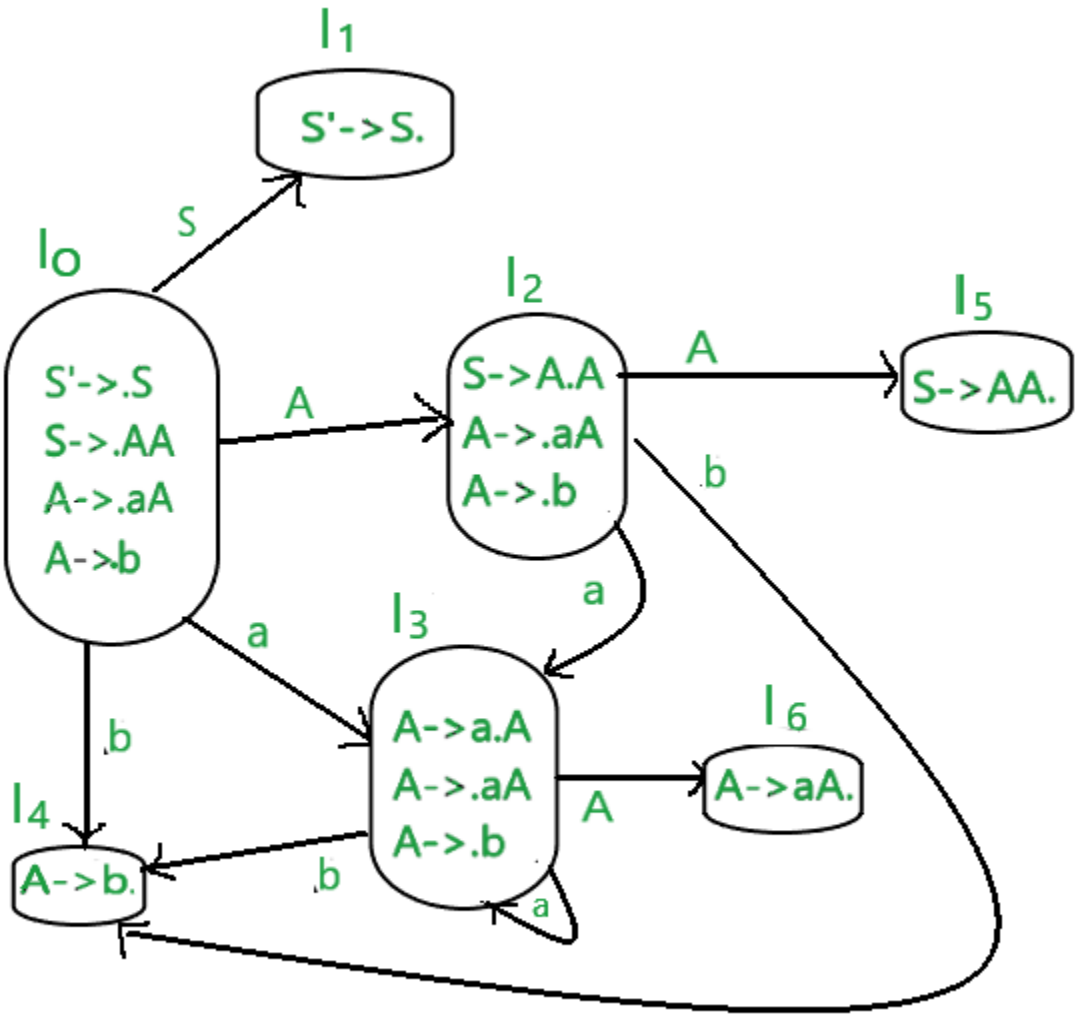
S–>AA
A–>aA|b

Solution :

STEP 1- Find augmented grammar –
The augmented grammar of the given grammar is:-

S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

STEP 2 – Find LR(0) collection of items
Below is the figure showing the LR(0) collection of items. We will understand everything one by one.



The terminals of this grammar are {a,b}
The non-terminals of this grammar are {S,A}

RULE – if any nonterminal has ' . ' preceding it, we have to write all its production and add ' . ' preceding each of its-production.
RULE – from each state to the next state, the ' . ' shifts to one place to the right.

- In the figure, I_0 consists of augmented grammar.
- I_0 goes to I_1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accepted state.
  S is seen by the compiler
- I_0 goes to I_2 when ' . ' of 1st production is shifted towards the right (S->A.A) . A is seen by the compiler
- I_0 goes to I_3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . a is seen by the compiler.
- I_0 goes to I_4 when ' . ' of the 3rd production is shifted towards the right (A->b.) . b is seen by the compiler.
- I_2 goes to I_5 when ' . ' of 1st production is shifted towards the right (S->AA.) . A is seen by the compiler
- I_2 goes to I_4 when ' . ' of 3rd production is shifted towards the right (A->b.) . b is seen by the compiler.
- I_2 goes to I_3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . a is seen by the compiler.
- I_3 goes to I_4 when ' . ' of the 3rd production is shifted towards the right (A->b.) . b is seen by the compiler.
- I_3 goes to I_6 when ' . ' of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler
- I_3 goes to I_3 when ' . ' of the 2nd production is shifted towards the right (A->a.A) . a is seen by the compiler.

STEP3 – defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table

- $ is by default a nonterminal that takes the accepting state.
- 0,1,2,3,4,5,6 denotes $I_0, I_1, I_2, I_3, I_4, I_5, I_6$
- I_0 gives A in I_2, so 2 is added to the A column and 0 rows.
- I_0 gives S in I1, so 1 is added to the S column and 1 row.
- similarly, 5 is written in A column and 2nd row, 6 is written in A column and 3 rows.
- I_0 gives an in I_3 to .so S3(shift 3) is added to a column and 0 rows.
- I_0 gives b in I4, so S4(shift 4) is added to the b column and 0 rows.
- Similarly, S3(shift 3) is added on a column and 2,3 rows, S4(shift 4) is added on b column and 2,3 rows.

- I_4 is reduced state as ' . ' is at the end. I_4 is the **3**rd production of grammar. So write r**3**(reduce 3) in terminals.
- I_5 is reduced state as ' . ' is at the end. I_5 is the **1**st production of grammar. So write r**1**(reduce 1) in terminals.
- I_6 is reduced state as ' . ' is at the end. I_6 is the **2**nd production of grammar. So write r**2**(reduce 2) in terminals.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

As each cell has only 1 value in it, hence, the given grammar is LR(0)

SLR(1):

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.

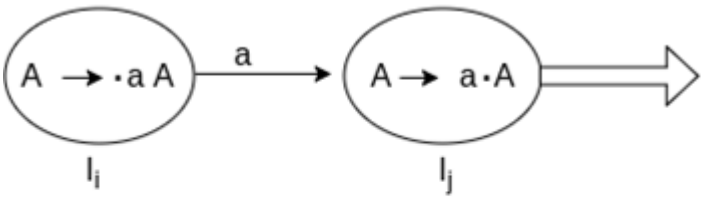
Various steps involved in the SLR (1) Parsing:

- 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

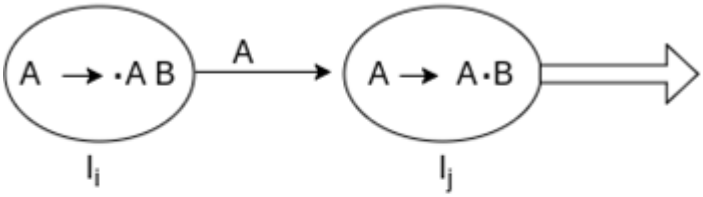
The steps which use to construct SLR (1) Table is given below:

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



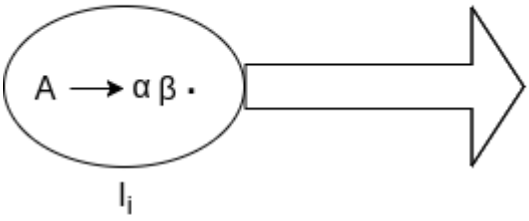
States	Action		Go to
	a	$	A
I_i I_j			j

If a state (I_i) 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

1. S -> •Aa
2. A->αβ•

1. Follow(S) = {$}
2. Follow (A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

SLR (1) Grammar

S → E
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)

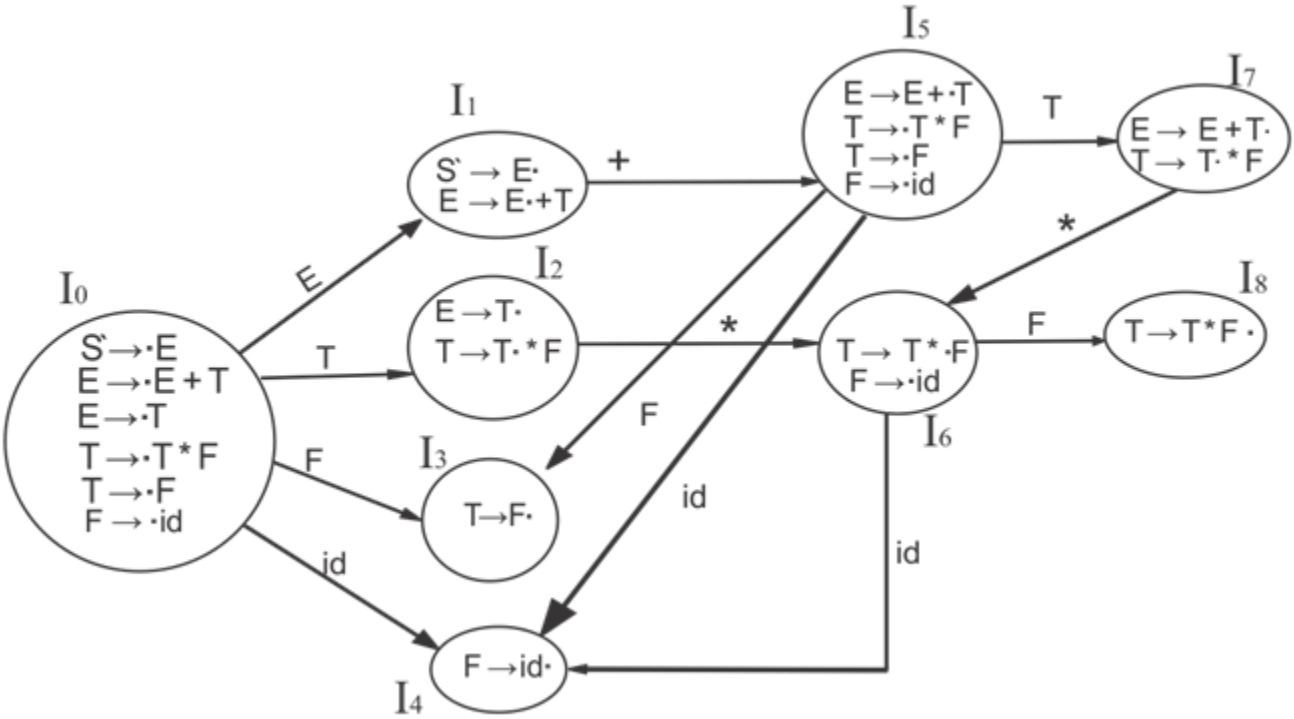
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

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

LR(1):

LR(1) items

We need a way to bring the notion of following tokens much closer to the productions that use them.

An *LR(1) item* has the form [*I*, *t*] where *I* is an LR(0) item and *t* is a token.

As the dot moves through the right-hand side of *I*, token *t* remains attached to it. LR(1) item [$A \rightarrow \alpha \cdot$, *t*] calls for a reduce action when the lookahead is *t*.

The follow context is determined when an LR(1) item [$A \rightarrow \cdot \alpha$, *t*] is created, and is carried along with it. Token *t* is a token that can follow *A in the context in which the A occurs*. This does not suffer from the difficulty of using FOLLOW sets to determine when to do reduce actions.

Closures of sets of LR(1) items

The closure of a set *S* of LR(1) items for augmented grammar *G* is computed as follows.

```
closure(S)
For each item [A → α · B β, t] in S,
  For each production B → γ in G,
    For each token b in FIRST(βt),
      Add [B → · γ, b] to S
```

You need to continue adding LR(1) items until no more can be added, just like for LR(0) closures.

Building the LR(1) finite-state machine.

Suppose that *G* is an augmented grammar with augmenting production $S' \rightarrow S$. The initial state has kernel {[$S' \rightarrow \cdot S$, $]}. Form its closure.

For each state, you need to build other state (if not yet built) and add transitions to them as follows.

Suppose that state *q* contains set *I* of LR(1) items.

For each symbol *s* (either a token or a nonterminal) that immediately follows a dot in an LR(1) item [$A \rightarrow \alpha \cdot s\beta$, *t*] in set *I*, let I_s be the set of all LR(1) items in *I* where *s* immediately follows the dot.

Move the dot to the other side of *s* in each of them. That set becomes the kernel of state *q*′, and you make a transition from *q* to *q*′ on *s*. As usual, form the closure of the set of LR(1) items in state *q*′.

Example (Page 262, Dragon Book)

For example, let *G* be the following grammar.

$$0.\ S' \rightarrow S$$
$$1.\ S \rightarrow C\ C$$
$$2.\ C \rightarrow \mathbf{c}\ C$$
$$3.\ C \rightarrow \mathbf{d}$$

The LR(1) finite-state machine for *G* is as follows.

State 0.
[*S'* → · *S*, $]
[*S* → · *C C*, $]
[*C* → · c *C*, c]
[*C* → · c *C*, d]
[*C* → · d, c]
[*C* → · d, d]
On *S* goto 1
On *C* goto 2
On c goto 3
On d goto 4

State 1.
[*S'* → *S* ·, $]

State 2.
[*S* → *C* · *C*, $]
[*C* → · c *C*, $]
[*C* → · d, $]
On *C* goto 5
On c goto 6
On d goto 7

State 3.
[*C* → c · *C*, c]
[*C* → c · *C*, d]
[*C* → · c *C*, c]
[*C* → · c *C*, d]
[*C* → · d, c]
[*C* → · d, d]
On *C* goto 8
On c goto 3
On d goto 4

State 4.
[*C* → d ·, c]
[*C* → d ·, d]

State 5.
[*S* → *C C* ·, $]

State 6.
[*C* → c · *C*, $]
[*C* → · c *C*, $]
[*C* → · d, $]
On *C* goto 9
On c goto 6
On d goto 7

State 7.

[*C* → **d ·, $]**

State 8.
[*C* → **c *C* ·, **c**]**
[*C* → **c *C* ·, **d**]**

State 9.
[*C* → **c *C* ·, $]**

The LR(1) parsing table is as follows.

	Actions			Goto	
	c	**d**	**$**	**S**	**C**
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

An abbreviation

It is common to have sets of LR(1) items where several of the LR(1) items contain the same LR(0) item. For example, state 8 above is

State 8.
[*C* → **c *C* ·, **c**]**
[*C* → **c *C* ·, **d**]**

Notation [$C \rightarrow$ **c** C ·, **c**/**d**] abbreviates a combination of those two items. State 8 can be rewritten as

State 8.
[*C* → **c *C* ·, c/d]**

LALR(1) parsers

Now we are able to explain what a LALR(1) parser is. Start with the LR(1) finite-state machine.

Compare each pair of states to one another by looking only at the LR(0) items that the LR(1) items contain.

If two states have exactly the same LR(0) items, combine those states into a single state by combining their LR(1) items.

The LR(1) finite state machine above is changed to the following.

State 0.

[S′ → · S, $]
[S → · C C, $]
[C → · c C, c/d]
[C → · d, c/d]
On S goto 1
On C goto 2
On c goto (3,6)
On d goto (4,7)

State 1.
[S′ → S ·, $]

State 2.
[S → C · C, $]
[C → · c C, $]
[C → · d, $]
On C goto 5
On c goto (3,6)
On d goto (4,7)

State (3,6).
[C → c · C, c/d/$]
[C → · c C, c/d/$]
[C → · d, c/d/$]
On C goto (8,9)
On c goto (3,6)
On d goto (4,7)

State (4,7).
[C → d ·, c/d/$]

State 5.
[S → C C ·, $]

State (8,9).
[C → c C ·, c/d/$]

The LALR(1) parsing table is as follows.

	Actions			Goto	
	c	**d**	**$**	**S**	**C**
0	s(3,6)	s(4,7)		1	2
1			acc		
2	s(3,6)	s(4,7)			5
(3,6)	s(3,6)	s(4,7)			(8,9)
(4,7)	r3	r3	r3		
5			r1		
(8,9)	r2	r2	r2		

Notice that

1. there are fewer states, and
2. there are no conflicts.

The LALR(1) parser always has exactly the same states as the SLR(1) parser. But, because it does not use the FOLLOW sets, it avoids some reduce actions that might cause conflicts.

Combining similar states does not always work. Sometimes, parsing conflicts arise. That means that LR(1) parsers more powerful than LALR(1) parsers.

But the state reduction is significant. For a typical programming language grammar, the LALR(1) finite-state machine can have an order of magnitude fewer states than the LR(1) finite-state machine.

A LALR(1) parser for a programming language like Java would have roughly 1000 states, where the LR(1) table would have roughly 10,000 states.

LALR(1):

LALR Parser is Look Ahead LR Parser. It is intermediate in power between SLR and CLR parser. It is the compaction of CLR Parser, and hence tables obtained in this will be smaller than CLR Parsing Table.

Here, first of all, we will construct LR (1) items. Next, we will look for the items having the same first component, and they are merged to form a single set of items. It means the states have the same first component, but the different second component can be integrated into a single state or item.

For Example.

Suppose if

I_4: C → d · , c | d

I_7: C → d · , $

Both items or states (I_4 and I_7) have the same first component, i.e., d · , but a different second component, i.e., c | d in I_4 and $ in I_7.

∴ these states can be merged to give

I_{47}: C → d · , c |d | $

Construction of LALR Parsing Table

Algorithm

Input − Augmented Grammar G′

Output − LALR Parsing Table

Method

- Construct LR (1) set of items, i.e., construct
  $C = \{I_0, I_1, I_2 \ldots . . I_n\}$
- Select the similar states having the same core, or first component and merge them into one.
  Let $C' = \{J_0, J_1, J_2 \ldots . . J_m\}$ be the resulting set.
- Construct Parsing Action for state J_1 similar to CLR construction. If there is a conflict in the Parsing Table, the algorithm can be considered to fail to produce an LALR parser.
- Construct goto actions as below.

Let goto [J,∗] = K where J is the union of one or more states of C.

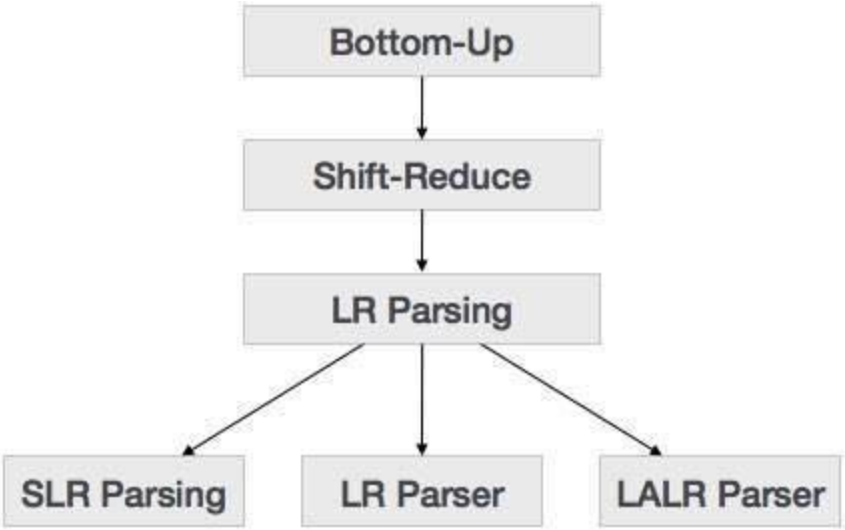
i.e., $J = I_1 \cup I_2 \ldots .\cup I_m$, then

then K = goto $(I_1,*) \cup$ goto $(I_2,*) \ldots .\cup$ goto $(I_m,*)$

Working of LALR Parser

BOTTOM UP PARSING:

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

LR Parsing Algorithm

Here we describe a skeleton algorithm of an LR parser:

```
token = next_token()

repeat forever
  s = top of stack

  if action[s, token] = “shift si” then
```

```
    PUSH token
    PUSH si
    token = next_token()

  else if action[s, token] = "reduce A::= β" then
    POP 2 * |β| symbols
    s = top of stack
    PUSH A
    PUSH goto[s,A]

  else if action[s, token] = "accept" then
    return

  else
    error()
```

LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Ambiguity and LR parsing:

LR Parser can be used to parse ambiguous grammars. LR parser resolves the conflicts (shift/reduce or reduce/reduce) in parsing table of ambiguous grammars based on certain rules (precedence and/or associativity of operators) of the grammar.

Example:

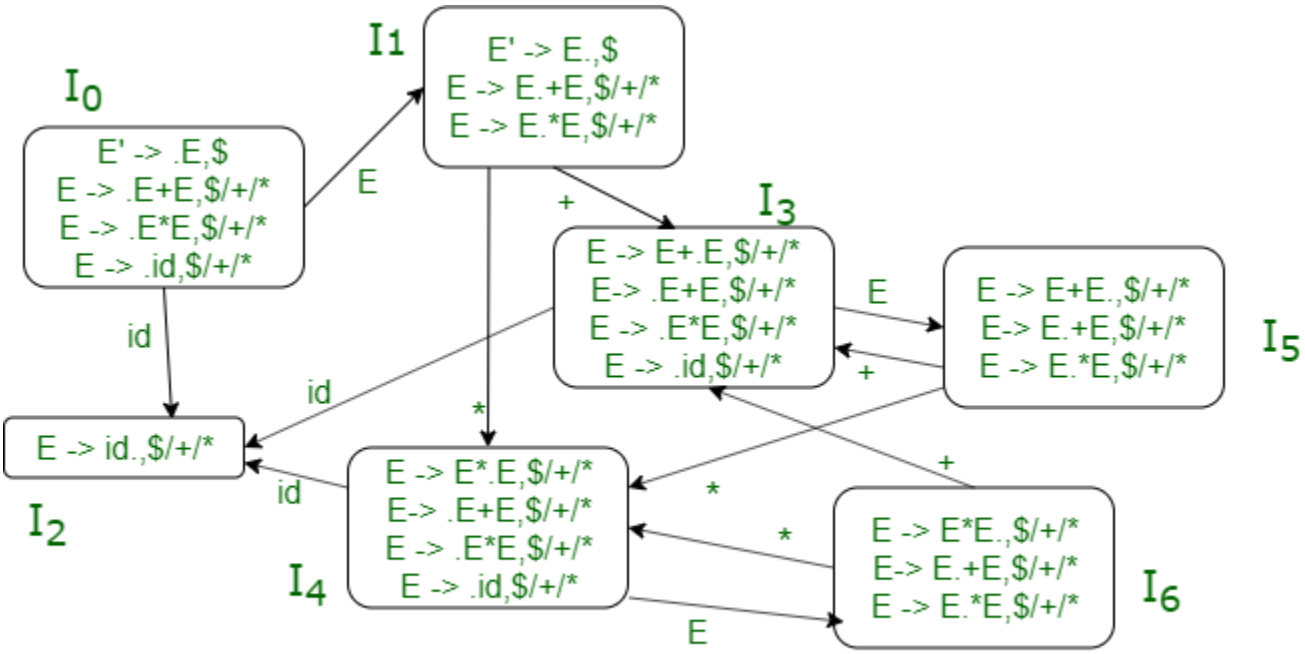
Lets take the following ambiguous grammar:

```
E -> E+E

E -> E*E

E -> id
```

Lets assume, the precedence and associativity of the operators (+ and *) of the grammar are as follows:

- “+” and “*” both are left associative,
- Precedence of “*” is higher than the precedence of “+”.

If we use LALR(1) parser, the LR(1) item DFA will be:



LR(1) item DFA

From the LR(1) item DFA we can see that there are shift/reduce conflicts in the state I_5 and I_6. So the parsing table is as follows:

	id	+	*	$	E
I_0	S_2				1
I_1		S_3	S_4	accept	
I_2		r_3	r_3	r_3	
I_3	S_2				5
I_4	S_2				6
I_5		r_1/S_3	r_1/S_4	r_1	
I_6		r_2/S_3	r_2/S_4	r_2	

Parsing Table

There are both shift and reduce moves in I_5 and I_6 on “+ and “*”. To resolve this conflict, that is to determine which move to keep and which to discard from the table we shall use the precedence and associativity of the operators. Consider the input string:

```
id + id + id
```

Lets look at the parser moves till the conflict state according to the above parsing table.

Stack	Symbol	Input
0		id + id + id $
0 2	id	+ id + id $
0 1	E	+ id + id $
0 1 3	E +	id + id $
0 1 3 2	E + id	+id $
0 1 3 5	E + E	+id $ reduce
0 1	E	+id $

Parser 1

Stack	Symbol	Input
0		id + id + id $
0 2	id	+ id + id $
0 1	E	+ id + id $
0 1 3	E +	id + id $
0 1 3 2	E + id	+id $
0 1 3 5	E + E	+id $ Shift
0 1 3 5 3	E + E +	id $

Parser 2

- If we take the reduce move of I_5 state on symbol “+” as in parser 1, then the left “+” of the input string is reduced before the right “+”, which makes “+” left associative.

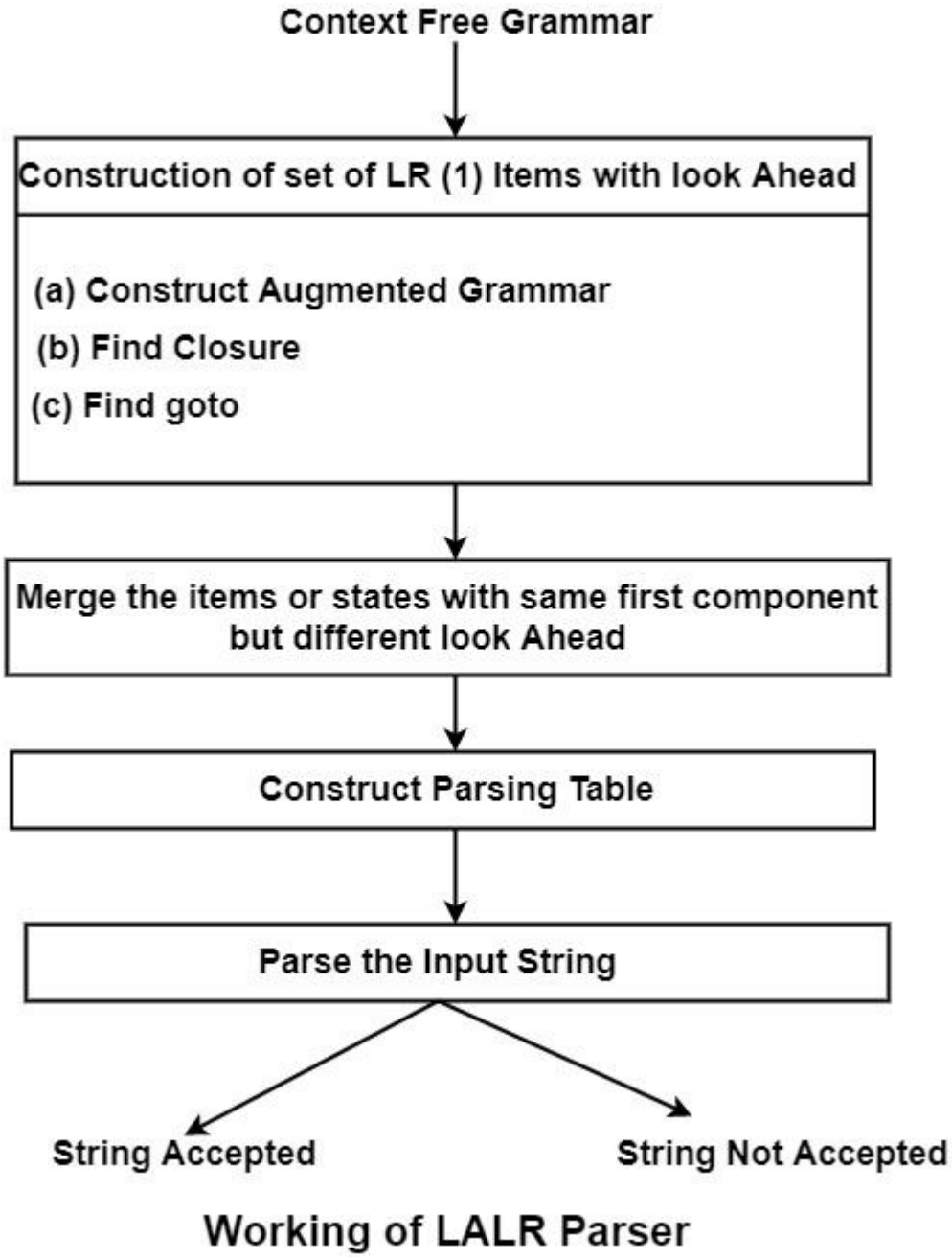
- If we take the shift move of I_5 state on symbol “+” as in parser 2, then the right “+” of the input string is reduced before the left “+”, which makes “+” right associative.

Similarly, Taking shift move of I_5 state on symbol “*” will give “*” higher precedence over “+”, as “*” will be reduced before “+”. Taking reduce move of I_5 state on symbol “*” will give “+” higher precedence over “*”, as “+” will be reduced before “*”. Similar to I_5, conflicts from I_6 can also be resolved.
According to the precedence and associativity of our example, the conflict is resolved as follows,

- The shift/reduce conflict at I_5 on “+” is resolved by keeping the reduce move and discarding the shift move, which makes “+” left associative.
- The shift/reduce conflict at I_5 on “*” is resolved by keeping the shift move and discarding the reduce move, which will give “*” higher precedence over “+”.
- The shift/reduce conflict at I_6 on “+” is resolved by keeping the reduce move and discarding the shift move, which will give “*” higher precedence over “+”.
- The shift/reduce conflict at I_6 on “*” is resolved by keeping the reduce move and discarding the shift move, which makes “*” left associative.

Generally, the parser generator tool **YAAC** resolves conflicts due to ambiguous grammars as follows,

- Shift/reduce conflict in the parsing table is resolved by giving priority to shift move over reduce move. If the string is accepted for shift move, then reduce move is removed, otherwise shift move is removed.
- Reduce/reduce conflict in the parsing table is resolved by giving priority to first reduce move over second reduce move. If the string is accepted for first reduce move, then second reduce move is removed, otherwise first reduce move is removed.



Working of LALR Parser

LALR Grammar − A grammar where the LALR parsing table has no multiply represented entries are said to be LALR (1) or LALR grammar.