- *Bottom up Parsing*
- *Reductions*
- *Handle Pruning*
- *Shift Reduce Parsing*
- *Problems related to Shift Reduce Parsing*
- *Conflicts during Shift Reduce Parsing*
- *Operator Precedence Parser*
- *Computation of LEADING*
- *Computation of TRAILING*
- *Problems related to LEADING AND TRAILING*
- *LR Parsers – Why LR Parsers*
- *Items and LR(0) Automation*
- *Closure of Item sets*
- *LR Parsing Algorithm*
- *SLR Grammars*
- *SLR Parsing Tables*
- *Problems related to SLR*
- *Construction of Canonical LR(1) and LALR*

- *Construction of LALR*
- *Problems related to Canonical LR(1) and LALR Parsing Table*
  - *Bottom up Parsing*
  - *Reductions*
  - *Handle Pruning*
  - *Shift Reduce Parsing*
  - *Problems related to Shift Reduce Parsing*
  - *Conflicts during Shift Reduce Parsing*
  - *Operator*
  -
  - *Precedence Parser*
  - *Computation of LEADING*
  - *Computation of TRAILING*
  - *Problems related to LEADING AND TRAILING*
  - *LR Parsers – Why LR Parsers*
  - *Items and LR(0) Automation*
  - *Closure of Item sets*
  - *LR Parsing Algorithm*

- *SLR Grammars*
- *SLR Parsing Tables*
- *Problems related to SLR*
- *Construction of Canonical LR(1) and LALR*
- *Construction of LALR*
- *Problems related to Canonical LR(1) and LALR Parsing Table*
- *Bottom up Parsing*
- *Reductions*
- *Handle Pruning*
- *Shift Reduce Parsing*
- *Problems related to Shift Reduce Parsing*
- *Conflicts during Shift Reduce Parsing*
-
- *Precedence Parser*
- *Computation of LEADING*
- *Computation of TRAILING*
- *Problems related to LEADING AND TRAILING*
- *LR Parsers – Why LR Parsers*

- *Items and LR(0) Automation*
- *Closure of Item sets*
- *LR Parsing Algorithm*
- *SLR Grammars*
- *SLR Parsing Tables*
- *Problems related to SLR*
- *Construction of Canonical LR(1) and LALR*
- *Construction of LALR*
- *Problems related to Canonical LR(1) and LALR Parsing Table*
- *Bottom up Parsing*
- *Reductions*
- *Handle Pruning*
- *Shift Reduce Parsing*
- *Problems related to Shift Reduce Parsing*
- *Conflicts during Shift Reduce Parsing*
- *Operator Precedence Parser*
- *Computation of LEADING*
- *Computation of TRAILING*

- *Problems related to LEADING AND TRAILING*
- *LR Parsers – Why LR Parsers*
- *Items and LR(0) Automation*
- *Closure of Item sets*
- *LR Parsing Algorithm*
- *SLR Grammars*
- *SLR Parsing Tables*
- *Problems related to SLR*
- *Construction of Canonical LR(1) and LALR*
- *Construction of LALR*
- *Problems related to Canonical LR(1) and LALR Parsing Table*

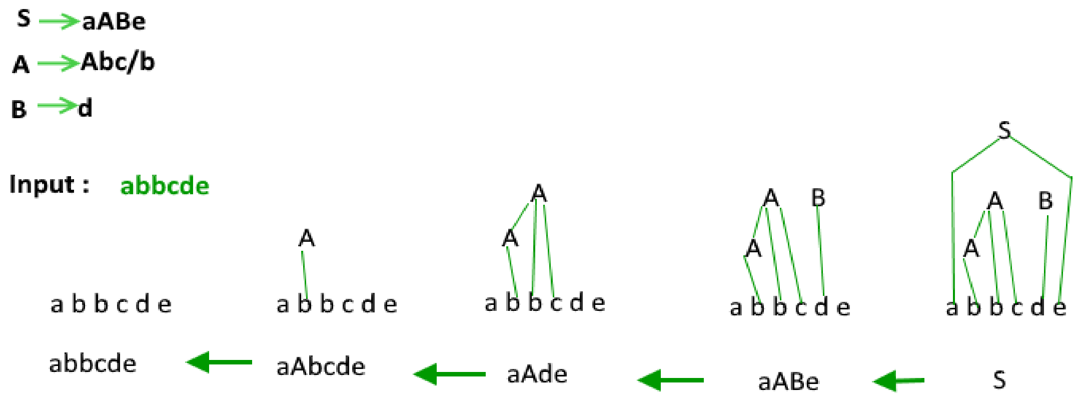
18CSC304J – COMPILER DESIGN
UNIT - III

Topics: Bottom up parsing, Reductions, Handle Pruning, Shift Reduce Parsing, Problems related to Shift Reduce Parsing, Conflicts during Shift Reduce Parsing, LR Parsers – Why LR Parsers, Items and LR(0) Automation, Closure of Item sets, LR Parsing Algorithm, Operator Precedence Parser, Computation of LEADING, Computation of TRAILING, Problems related to LEADING AND TRAILING, SLR Grammars, SLR Parsing Tables, Problems related to SLR, Construction of Canonical LR(1) and LALR, Construction of LALR, Problems related to Canonical LR(1) and LALR Parsing Table.

Textbook: Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*" Addison-Wesley, 1986.

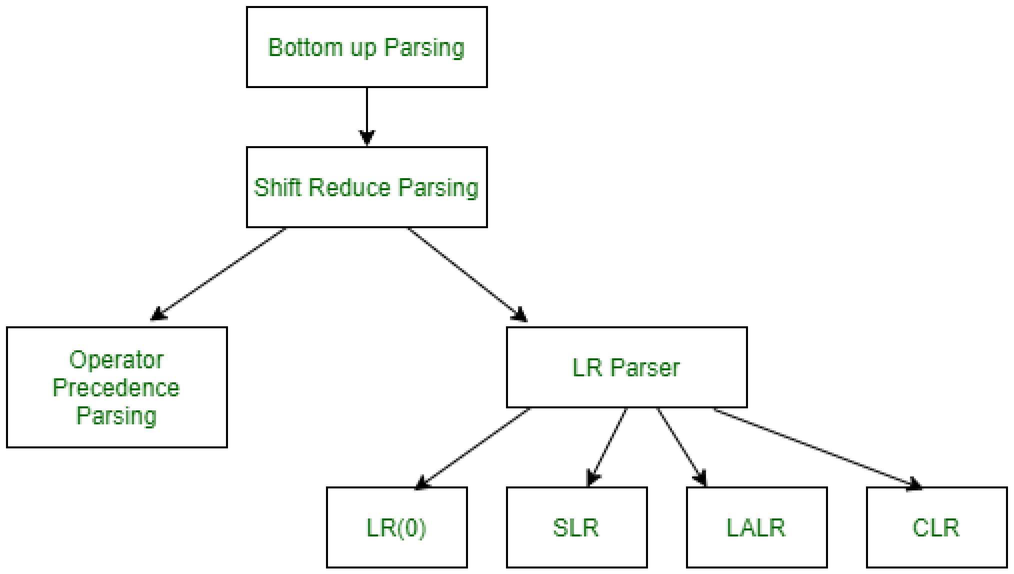
BOTTOM-UP PARSING - REDUCTION:

Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.



CLASSIFICATION OF BOTTOM-UP PARSERS:

- Bottom-up syntax analysis is also termed as shift-reduce parsing.
- The common method of shift-reduce parsing is called LR parsing.
- Operator precedence parsing is an easy-to-implement shift-reduce parser.



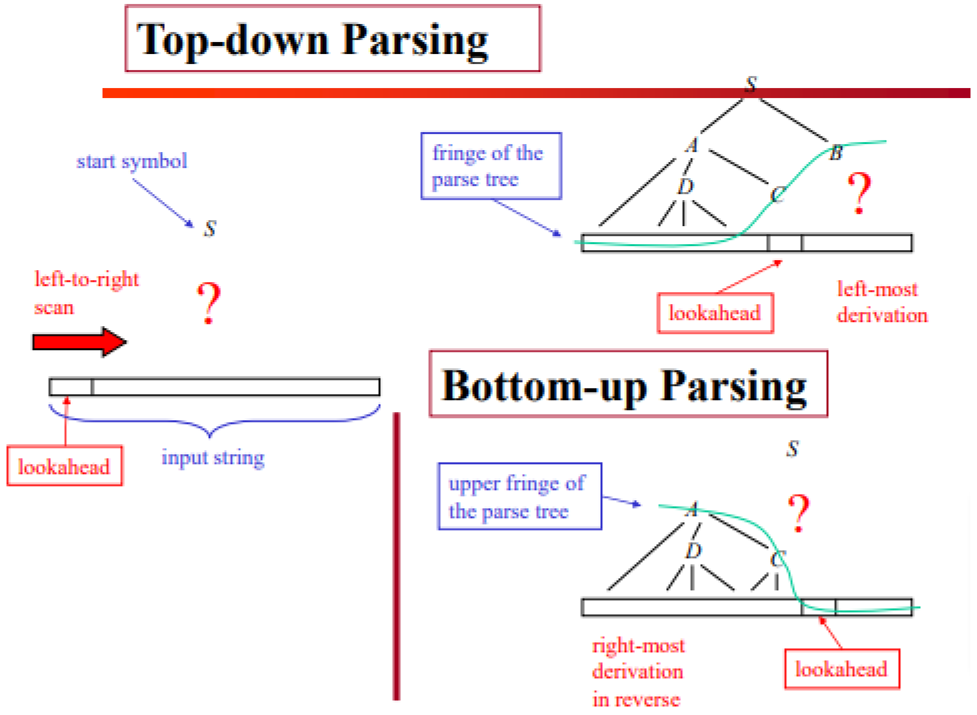
TOP-DOWN AND BOTTOM-UP PARSERS:

Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad “pick” ⇒ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production.
- Bottom-up parsers handle a large class of grammars



HANDLE AND HANDLE PRUNING:

Handle:

The handle is the substring that matches the body of a production whose reduction represents one step along with the reverse of a Rightmost derivation.
The handle of the right sequential form Y is the production of Y where the string S may be found and replaced by A to produce the previous right sequential form in RMD (Right Most Derivation) of Y.

Sentential form:

S => a here, 'a' is called sentential form, 'a' can be a mix of terminals and non-terminals.

Example:

Consider Grammar:	S -> aSa | bSb | ε
Derivation:	S => aSa => abSba => abbSbba => abbbba

Left Sentential and Right Sentential Form:

A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.
A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

Handle contains two things:

- Production
- Position

Handle Pruning:
If A⇢ β is a production then reducing β to A by the given production is called Handle Pruning ie., removing the children of A from the parse tree. A rightmost derivation in reverse can be obtained by handle pruning.

Steps to Follow:

1. Start with a string of terminals ‘w’ that is to be parsed.
2. Let w = γ_n, where γ_n is the nth right sequential form of an unknown Right Most Derivation (RMD).
3. To reconstruct the RMD in reverse, locate handle β_n in γ_n. Replace β_n with LHS of some $A_n \dashrightarrow \beta_n$ to get (n-1)th Right Sequential Form $\gamma_{n-1.}$ Repeat.

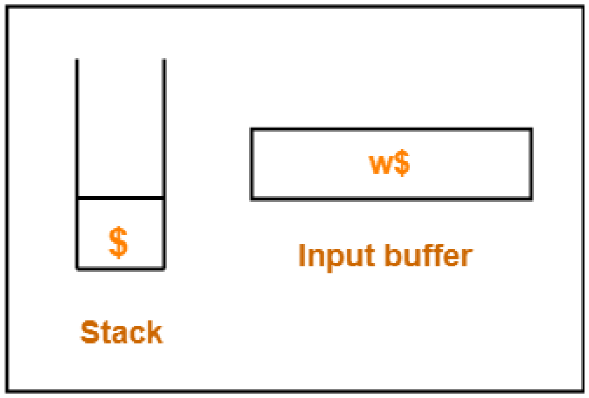
Example:

Right Sequential Form	Handle	Reducing Production
id + id * id	id	E ⇒ id
E + id * id	id	E ⇒ id
E + E * id	id	E ⇒ id
E + E * E	E + E	E ⇒ E + E
E * E	E * E	E ⇒ E * E
E (Root)		

SHIFT – REDUCING PARSER:

Reduced to
String
Start Symbol

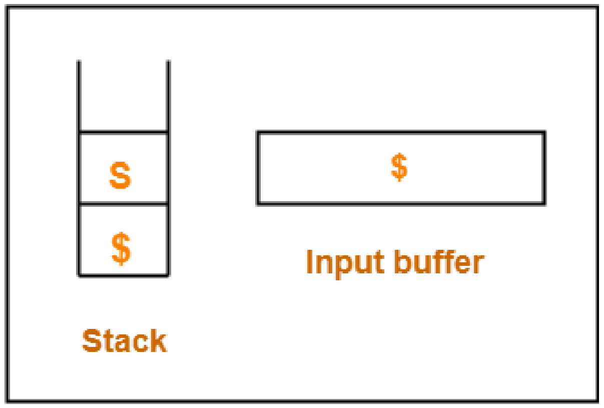
It is a type of bottom-up parser. In this parser, a stack holds the grammar symbol, and an input buffer holds the rest of the string that is set to be parsed.
The symbol **$** denotes the bottom of the stack and also the input's right side. While discussing the bottom-up parsing, we generally represent the top of the stack on the right-hand side. In the beginning, the stack is empty, and the string **a** is on the input side, as shown below:



Initial Configuration

This parsing generally performs two action shifts and reduce. But there are four possible actions that this parser can perform: shift, reduce, accept, error.
1) **Shift** – Shift operation shifts the input symbol onto the top of the stack.
2) **Reduce** – The top of the stack must hold the right end of that string, which is set to be reduced. It finds the left end of the string within the stack and decides which non – terminal can replace the string.

3) **Accept** – When we are only left with the start symbol in the stack, then the parsing action is called as Accept state.
4) **Error** – It detects the error and try to recover it.



Final Configuration

Example: 1
Consider the grammar:
A ⇢ A + A
A ⇢ A – A
A ⇢ (A)
A ⇢ a

Perform Shift Reduce parsing for input string: a1 - (a2 + a3)

Parsing Table:

Stack	Input	Action
$	a1 - (a2 + a3) $	Shift a1
$a1	- (a2 + a3) $	Reduce by A => a

$A	- (a2 + a3) $	Shift -
$A -	(a2 + a3) $	Shift (
$A - (	a2 + a3) $	Shift a2
$A-(a2	+a3)$	Reduce by A => a
$A - (A	+ a3) $	Shift +
$A - (A +	a3) $	Shift +
$A - (A + a3	) $	Reduce by A =>a
$A - (A + A	) $	Shift)
$A - (A + A)	$	Reduce by A => A +A
$A - (A)	$	Reduce by A => (A)
$A – A	$	Reduce by A => A - A
$A	$	Accept

Example: 2

Consider the grammar
E –> 2E2
E –> 3E3
E –> 4
Perform Shift Reduce parsing for input string “32423”.

Stack	Input Buffer	Parsing Action
$	32423$	Shift
$3	2423$	Shift
$32	423$	Shift
$324	23$	Reduce by E --> 4
$32E	23$	Shift
$32E2	3$	Reduce by E --> 2E2
$3E	3$	Shift
$3E3	$	Reduce by E --> 3E3
$E	$	Accept

Example: 3
Consider the grammar
S –> (L) | a
L –> L, S | S
Perform Shift Reduce parsing for input string (a, (a, a))

Stack	Input Buffer	Parsing Action
$	(a, (a, a)) $	Shift

Stack	Input Buffer	Parsing Action
\$ (	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

Stack	**Input Buffer**	**Parsing Action**
\$ (L, (L)	) \$	Reduce S → (L)
\$ (L, S	) \$	Reduce L → L, S
\$ (L	) \$	Shift
\$ (L)	\$	Reduce S → (L)
\$ S	\$	Accept

CONFLICTS DURING SHIFT-REDUCE PARSING:

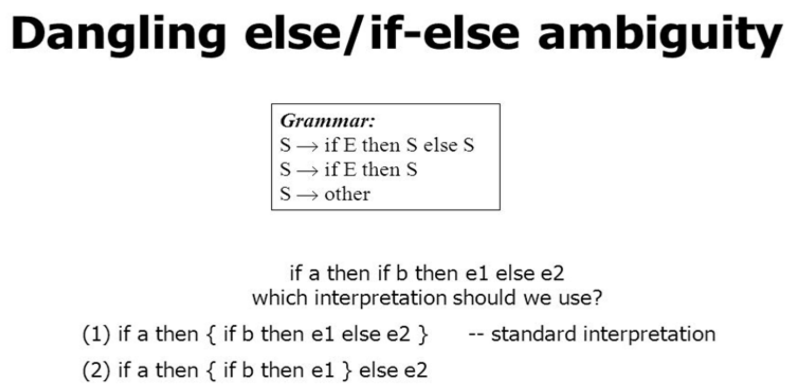
Shift-reduce parsing cannot be used in context free grammar.
Two types of conflicts in shift-reduce parsing:

1. **Shift-Reduce Conflict:**
   For every shift-reduce parser, such grammar can reach a configuration in which the parser cannot decide whether to shift or to reduce the current handle.

   Example:
   Consider the grammar,
   A ⇢ Sa
   B ⇢ Sab

Stack	Input Buffer	
…	…	
\$Sa	b…\$	//Conflict occur to do reduce of “Sa” to A or shift “b” from input buffer to stack.

2. **Reduce-Reduce Conflict:**
   It cannot decide which of the several reductions to make, by knowing the entire stack contents and the next input symbol.



Dangling else/if-else ambiguity

Grammar:
S $\longrightarrow$ if E then S else S
S $\longrightarrow$ if E then S
S $\longrightarrow$ other

if a then if b then e1 else e2
which interpretation should we use?

(1) if a then { if b then e1 else e2 } -- standard interpretation

(2) if a then { if b then e1 } else e2

OPERATOR PRECEDENCE PARSER:

A parser that reads and understand an operator precedence grammar is called as Operator Precedence Parser. An operator precedence grammar has to satisfies the following 2 conditions:

1. There exists no production rule which contains ε on its RHS.
2. There exists no production rule which contains two non-terminals adjacent to each other on its RHS

It represents a small class of grammar. But it is an important class because of its widespread applications.

E → EAE | (E) | -E | id

A → + | - | x | / | ^

Operator Precedence Grammar ✗

E → E + E | E - E | E x E | E / E | E ^ E | (E) | -E | id

Operator Precedence Grammar ✓

Designing Operator Precedence Parser:

In operator precedence parsing,

1. We define precedence relations between every pair of terminal symbols.
2. We construct an operator precedence table.

Defining Precedence Relations:

The precedence relations are defined using the following rules:

Rule: 1

1. If precedence of b is higher than precedence of a, then we define a < b
2. If precedence of b is same as precedence of a, then we define a = b
3. If precedence of b is lower than precedence of a, then we define a > b

Rule: 2

1. An identifier is always given the higher precedence than any other symbol.

2. $ symbol is always given the lowest precedence.

Rule: 3

1. If two operators have the same precedence, then we go by checking their associativity.

Parsing A Given String:

The given input string is parsed using the following steps:

Step: 1

Insert the following:

1. $ symbol at the beginning and ending of the input string.
2. Include precedence operator between every two symbols of the string by referring the operator precedence table.

Step: 2

1. Start scanning the string from LHS in the forward direction until > symbol is encountered.
2. Keep a pointer on that location.

Step: 3

1. Start scanning the string from RHS in the backward direction until < symbol is encountered.
2. Keep a pointer on that location.

Step: 4

1. Everything that lies in the middle of < and > forms the handle.
2. Replace the handle with the head of the respective production.

Step: 5

1. Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

Precedence Rule:

Operators	Precedence	Association
↑	Highest	Right Associative
* and /	Next Highest	Left Associative

Operators	**Precedence**	**Association**
+ and −	Lowest	Left Associative

Operator Precedence Relations:

	+	−	*	/	↑	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
−	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
↑	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>
(	<.	<.	<.	<.	<.	<.	<.	=	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.		

Example:
Consider the following grammar,
E → EAE | id
A → + | *
Construct the operator precedence parser and parse the string id + id * id.

Step: 1
We convert the given grammar into operator precedence grammar.
The equivalent operator precedence grammar is
E → E + E | E * E | id

Step: 2

Given string is id + id * id, add \$ in the beginning and in the end.
\$ id + id * id \$
The terminal symbols in the grammar are {id, +, *, \$}
We construct the operator precedence table as

	id	**+**	*****	**\$**
id		>	>	>
+	<	>	<	>
*****	<	>	>	>
\$	<	<	<	

Parsing Given String:
Given string to be parsed is **id + id * id**.

Step: 1
We insert \$ symbol at both ends of the string as
\$ id + id * id \$

We insert precedence operators between the string symbols as
\$ < id > + < id > * < id > \$

Step: 2
We scan and parse the string as

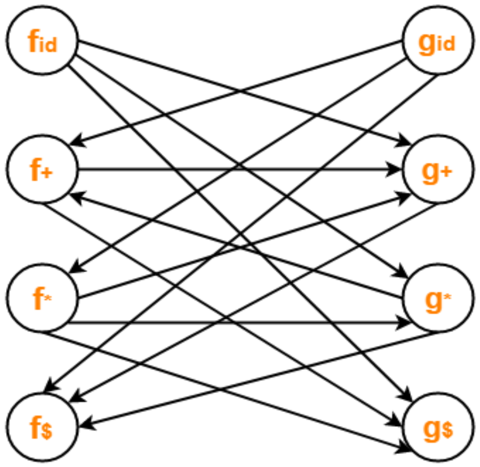
$\$\ \underline{<\textbf{id}>} + < \text{id} > * < \text{id} > \$$ //Consider > first, move backward for <
$\$\ E + \underline{<\textbf{id}>} * < \text{id} > \$$
$\$\ E + E * \underline{<\textbf{id}>}\ \$$
$\$\ E + E * E\ \$$ //Eliminate Non-terminals
$\$ + * \$$
$\$ < + \underline{<} * \underline{>}\ \$$ //Apply precedence
$\$\ \underline{< + >}\ \$$ //Apply precedence
$\$\ \$$

Precedence functions:
In practice, operator precedence table is not stored by the operator precedence parsers. This is because it occupies the large space. Instead, operator precedence parsers are implemented in a very unique style. They are implemented using operator precedence functions. Precedence functions perform the mapping of

terminal symbols to the integers. To decide the precedence relation between symbols, a numerical comparison is performed. It reduces the space complexity to a large extent. 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

The graph representing the precedence functions is



Graph Representing Precedence Functions

There are no cycles, so precedence function exists. Precedence functions are given as,
The path f_a is given by the number of edges from f_a to either $f_\$$ or $g_\$$
Path of $f_+ = 2$, $f_+ \rightarrow g_+ \rightarrow f_\$$
Path of $f_* = 4$, $f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
Path of $f_{id} = 4$, $f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
Path of $f_\$ = 0$, f_{id}
Path of $g_+ = 1$, $g_+ \rightarrow f_\$$
Path of $g_* = 3$, $g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
Path of $g_{id} = 5$, $g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
Path of $g_\$= 0$, $g_\$$

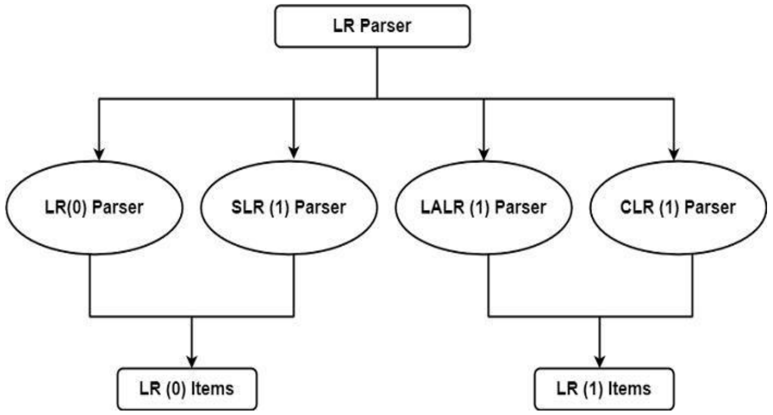
The resulting precedence functions are

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

LR PARSERS:

Description of LR parser:
LR parsers are table-driven, much like the non-recursive LL parsers. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

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



• SLR(l) – Simple LR
  o Works on smallest class of grammar.
  o Few numbers of state, hence very small table.
  o Simple and fast construction.

• LR(1) – LR parser
  o Also called as Canonical LR parser.
  o Works on complete set of LR(l) Grammar.
  o Generates large table and large number of states.
  o Slow construction.

- **LALR(l) – Look ahead LR parser**
  - o Works on intermediate size of grammar.
  - o Number of states are same as in SLR(l).

Advantages of LR parser:
- LR parsers can handle a large class of context-free grammars.
- The LR parsing method is a most general non-back tracking shift-reduce parsing method.
- An LR parser can detect the syntax errors as soon as they can occur.
- LR grammars can describe more languages than LL grammars.

Disadvantages of LR parsers:
- It is too much work to construct LR parser by hand. It needs an automated parser generator.
- If the grammar contains ambiguities or other constructs then it is difficult to parse in a left-to-right scan of the input.

LR(0) items:
An LR(0) item is a production of the grammar with exactly one dot on the right-hand side. For example, production $T \rightarrow T * F$ leads to four LR(0) items:

$$T \rightarrow \cdot T * F$$

$$T \rightarrow T \cdot * F$$

$$T \rightarrow T * \cdot F$$

$$T \rightarrow T * F \cdot$$

What is to the left of the dot has just been read, and the parser is ready to read the remainder, after the dot. Two LR(0) items that come from the same production but have the dot in different places are considered different LR(0) items.

Closures:
Suppose that S is a set of LR(0) items. The following rules tell how to build closure(S), the closure of S. We must add LR(0) items to S until there are no more to add. All members of S are in the closure(S).
Suppose closure(S) contains item $A \rightarrow \alpha{\cdot}B\beta$, where B is a nonterminal. Find all productions $B \rightarrow \gamma_1, \ldots, B \rightarrow \gamma_n$ with B on the left-hand side. Add LR(0) items $B \rightarrow \cdot\gamma_1, \ldots B \rightarrow \cdot\gamma_n$ to closure(S).
Example:
$E \rightarrow E + \cdot T$
Since there is an item with a dot immediately before nonterminal T, we add $T \rightarrow \cdot F$ and $T \rightarrow \cdot T * F$.
The set now contains the following LR(0) items.

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot F$$

$$T \rightarrow \cdot T * F$$

Now there is an item in the set with a dot immediately followed by F. So we add items $F \rightarrow \cdot n$ and $F \rightarrow \cdot (E)$.

The set now contains the following items.

E → E + · T

T → · F

T → · T * F

F → · n

F → · (E)

No more LR(0) items need to be added, so the closure is finished. What is the point of the closure? LR(0) item E → E + · T indicates that the parser has just finished reading an expression followed by a + sign. In fact, E + are the top two symbols on the stack.

Construct canonical LR(0) collection:
Augmented grammar is defined with two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then augmented grammar G' is G with a new start symbol S' —> S. The role of augmented production is to stop parsing and notify the acceptance of the input i.e., acceptance occurs when and only when the parser performs reduction by S' —> S.

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

I_0 State:
Adding a production called Augment production to the I_0 State and Compute the Closure
I_0 = Closure (S` → •S)
Add all productions starting with S into I_0 state because "•" is followed by the non-terminal. So, the I_0 State becomes
I_0 = S` → •S
S → •AA
Add all productions starting with "A" in modified I_0 state because "•" is followed by the non-terminal. So, the I_0 State becomes.
I_0= S` → •S
S → •AA
A → •aA
A → •b

I_1= Go to (I_0, S) = closure (S` → S•) = S` → S•

Here, the Production is reduced so close the State.
I_1= S` → S•

I_2= Go to (I_0, A) = closure (S → A•A)
Add all productions starting with A in to I_2 State because "•" is followed by the non-terminal. So, the I_2 State becomes
I_2 =S→A•A
    A → •aA
    A → •b
Go to (I_2, a) = Closure (A → a•A) = (same as I_3)
Go to (I_2, b) = Closure (A → b•) = (same as I_4)

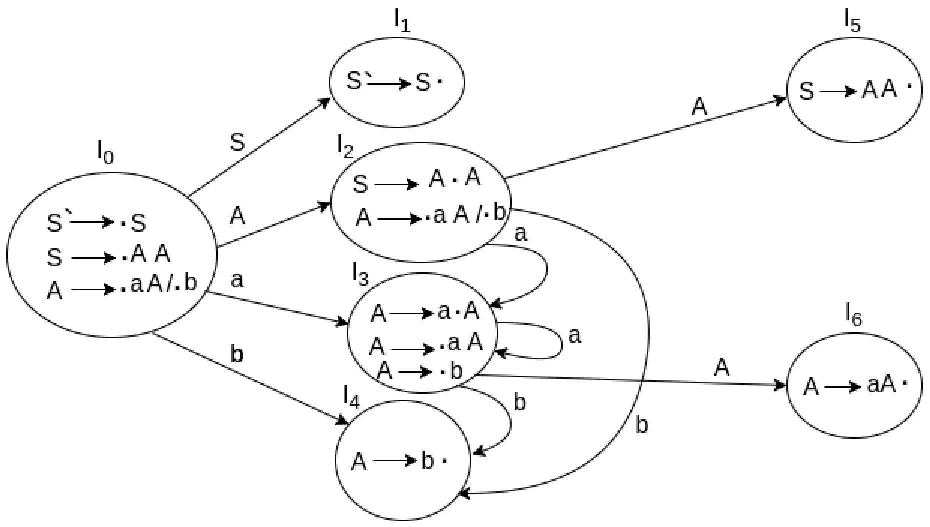
I_3= Go to (I_0, a) = Closure (A → a•A)
Add productions starting with A in I_3.
A → a•A
A → •aA
A → •b
Go to (I_3, a) = Closure (A → a•A) = (same as I_3)
Go to (I_3, b) = Closure (A → b•) = (same as I_4)

I_4= Go to (I_0, b) = closure (A → b•) = A → b•

I_5= Go to (I_2, A) = Closure (S → AA•) = S →AA•

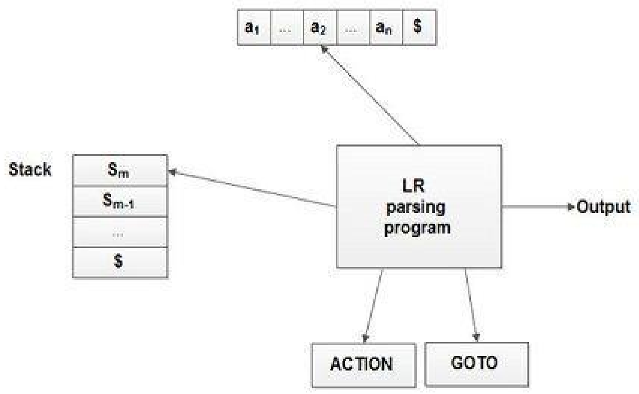
I_6= Go to (I_3, A) = Closure (A → aA•) = A → aA•

Drawing DFA:
The DFA contains the 7 states I_0 to I_6.



I_1, $I_{5,}$ I_6 are called final items, when the dot is present at the end ie., at the rightmost end. It tells us that, it has scanned all the symbols and time to reduce it.

Model of LR Parser:
LR parser consists of an input, an output, a stack, a driver program and a parsing table.
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.
3. 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. It has two functions:
1. Action
2. Goto
The driver program is same for all LR parsers. Only the parsing table changes from one parser to another. The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the information contained in the stack. The stack holds a sequence of states, s_o, s_1, … s_m, where s_m is on the top.



Action: This function takes as arguments a state *i* and a terminal *a* (or \$, the input end marker). The value of ACTION [i, a] can have one of the four forms:

1. Shift - If Action [S_m, a_i] = shift s then shift a_i with state s onto the stack

2. Reduce − If Action [S_m, a_i] = reduce A → β, then Parser perform reduction, here

s = goto[s_{m-r}, A]

r = length of β

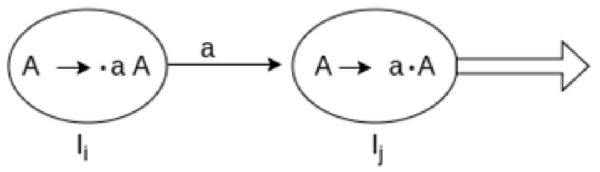
Number of symbols popped = 2r (r state symbols + r grammar symbols)

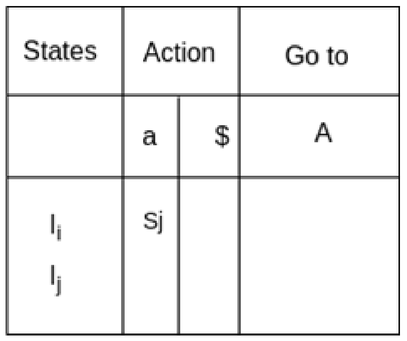
3. Accept − If Action [s_m, a_i] =accept, Parsing is completed.

4. Error − If Action [s_m, a_i] = error, Parser calls an error-correcting function.

Goto: This function takes a state and grammar symbol as arguments and produces a state.
If GOTO [I_i ,A] = I_j, the GOTO also maps a state *i* and non-terminal A to state *j*.

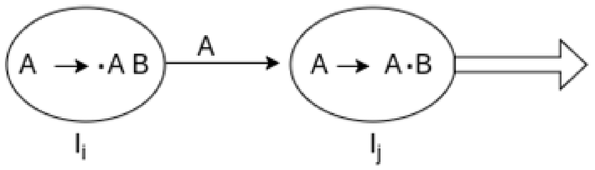
LR(0) Table:
If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.

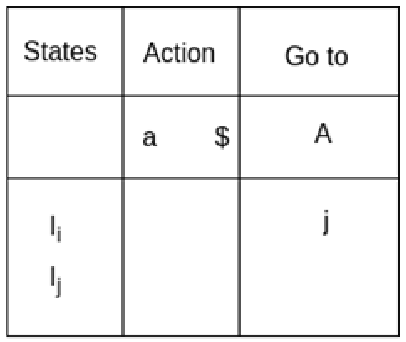




States	Action		Go to
	a	\$	A
I_i	Sj		
I_j			

If a state (I_i) is going to some other state (I_j) on a variable/non-terminal then it corresponds to Goto part.





States	Action		Go to
	a	$	A
I_i I_j			j

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		

Goto:
I_0 on S is going to I_1 so write it as 1.
I_0 on A is going to I_2 so write it as 2.
I_2 on A is going to I_5 so write it as 5.
I_3 on A is going to I_6 so write it as 6.

Action:
I_0, I_2 and I_3on a are going to I_3 so write it as S_3 which means that shift 3.
I_0, I_2 and I_3 on b are going to I_4 so write it as S_4 which means that shift 4.

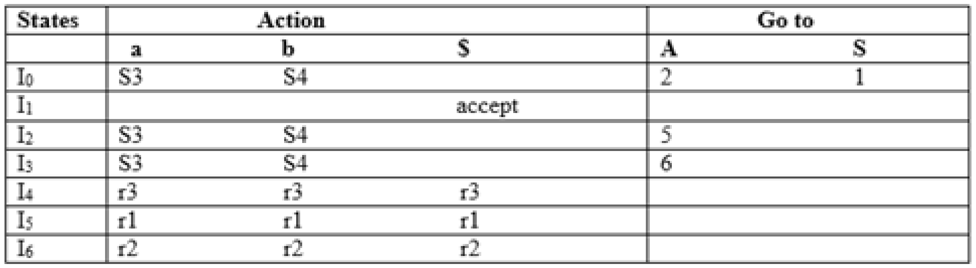
Final items:
I_4, I_5 and I_6 all states contain the final item because they contain • in the right most end. So, rate the production as production number.
Productions are numbered as follows:
S → AA ... (Production number is 1)

$A \rightarrow aA$ ... (Production number is 2)
$A \rightarrow b$ ... (Production number is 3)
I_1 contains the final item which drives ($S` \rightarrow S\bullet$), so action $\{I_1, \$\}$ = Accept. (Because it's an augmented production of the grammar)
I_4 contains the final item which drives $A \rightarrow b\bullet$ and that production corresponds to the production number 3 so write it as r3 in the entire row.
I_5 contains the final item which drives $S \rightarrow AA\bullet$ and that production corresponds to the production number 1 so write it as r1 in the entire row.
I_6 contains the final item which drives $A \rightarrow aA\bullet$ and that production corresponds to the production number 2 so write it as r2 in the entire row.

LR Parsing Algorithm:
Input: An input string w and an LR parsing table with functions action and goto for grammar G.
Output: If w is in L(G), a bottom-up parse for w; otherwise, an error indication.
Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and w $ in the input buffer. The parser executes the following program until an accept or error action is encountered.
Algorithm:

```
set ip to point to the first symbol of w;
repeat forever begin
        let s be the state on the top of the stack and
                'a' the symbol pointed by ip;
        if action[s,a] = shift s′ then begin
                push a then s′ on top of the stack;
                advance ip to the next input symbol;
         end
        else if action[s,a] = reduce A → β then begin
                pop 2∗ | β | symbols off the stack;
                let s ′ be the state now on top of the stack;
                push then goto[s′, A] on top of the stack;
                output the production A → β;
        end
        else if action[s,a] = accept then return
        else error()
end
```

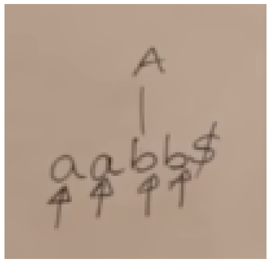
Example:
$S` \rightarrow S$
$S \rightarrow AA$ ... (Production number is 1)
$A \rightarrow aA$ ... (Production number is 2)
$A \rightarrow b$ ... (Production number is 3)
Input string aabb



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		

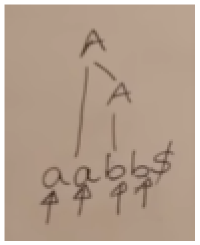
Stack: 0
Initially, stack has state 0.
Consider 1st symbol from input string aabb$ and compare input a and state 0 in table, it is s3 ie., shift3. Means shift a and 3 to stack.
Stack: 0 a 3
Increment input pointer, consider 2st symbol from input string aabb$ and compare input a and state 3 in table, it is s3 ie., shift3. Means shift a and 3 to stack.
Stack: 0 a 3 a 3
Increment input pointer, consider 3st symbol from input string aabb$ and compare input b and state 3 in table, it is s4 ie., shift4. Means shift b and 4 to stack.
Stack: 0 a 3 a 3 b 4
Increment input pointer, consider 4st symbol from input string aabb$ and compare input b and state 4 in table, it is r3 ie., reduce3. Means A ⟶ b (Production number: 3), reduce the previous symbol aabb$.
A ⟶ |b|, find the length of the right-side production. If length is 1, then pop 2 elements from stack.

Stack: 0 a 3 a 3 ~~b 4~~
Push left side of the production rule, A ⟶ b
We reduce b to A.



Stack: 0 a 3 a 3 A, consider 3 and A in table, it is 6.
Stack: 0 a 3 a 3 A 6
Increment input pointer, consider 4st symbol from input string aabb$ and compare input b and state 6 in table, it is r2 ie., reduce2. Means A ⟶ aA (Production number: 2)
A ⟶ |aA|, length is 2, so pop 4 elements from stack.

Stack: 0 a 3 ~~a 3 A 6~~



Push left side of the production rule, A → aA

Stack: 0 a 3 A

Compare 3 and A in table, it is 6

Stack: 0 a 3 A 6

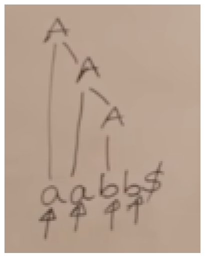
Increment input pointer, consider 4st symbol from input string aabb$ and compare input b and state 6 in table, it is r2 ie., reduce2. Means A → aA (Production number: 2)

A → |aA|, length is 2, so pop 4 elements from stack.

Stack: 0 ~~a 3 A 6~~

Push left side of the production rule, A → aA

Stack: 0 A



Compare 0 and A in table, it is 2

Stack: 0 A 2

Increment input pointer, consider 4st symbol from input string aabb$ and compare input b and state 2 in table, it is s4 ie., shift4

Stack: 0 A 2 b 4

Increment input pointer, consider 5st symbol from input string aabb$ and compare $ and 4 in table, it is r3 ie., reduce 3. A → b (Production number: 3)

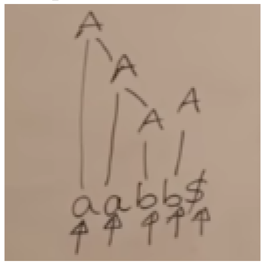
A → |b|, length is 1, so pop 2 elements from stack

Stack: 0 A 2 ~~b 4~~

A → b,

Stack: 0 A 2 A

Compare 2 and A in table, it is 5



Previous symbol b is reduced to A

Stack: 0 A 2 A 5

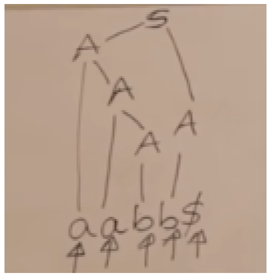
Increment input pointer, consider 5st symbol from input string aabb$ and compare $ and 5 in table, it is r1 ie., reduce 1. S → AA (Production number: 1)

A → |AA|, length is 2, so pop 4 elements from stack

Stack: 0 ~~A 2 A 5~~

S → AA

Stack: 0 S



Compare 0 and S, it is 1.

Stack: 0 S 1

Increment input pointer, consider 5st symbol from input string aabb$ and compare $ and 1 in table, it is accepted.

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

Example:
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

The canonical collection of LR(0) items are
I_0:
E' →. E
E →. E + T
E →. T
T →. T * F
T →. F
F →. (E)
F → .id

I_1: Goto((I_0, E)
E' → E. // Here, the Production is augmented production, which is considered as Accept in Parsing table.
E → E.+ T

I_2: Goto((I_0, T)
E → T.
T → T.* F

I_3: Goto((I_0, F)
T → F.

I_4: Goto((I_0, ()
F → (. E)
E →. E + T
E →. T
T →. T * F
T →. F

F →. (E)
F → .id

I_5: Goto((I_0, id)
F → id.

I_6: Goto((I_1, +)
E → E +. T
T →. T * F
T →. F
F →. (E)
F → .id

I_7: Goto((I_2, *)
T → T *. F
F →. (E)
F → .id

I_8: Goto((I_4, E)
F → (E.)
E → E. + T

I_2: Goto((I_4, T)
E → T.
T → T.* F

I_3: Goto((I_4, F)
T → F.

I_4: Goto((I_4, ()
F → (. E)
E →. E + T
E →. T
T →. T * F
T →. F
F →. (E)
F → .id

I_5: Goto((I_4, id)
F → id.

I_9: Goto((I_6, T)
E → E + T.
T → T. * F

I_3: Goto((I_6, F)
T → F.

I_4: Goto((I_6, ()
F → (. E)
E →. E + T
E →. T
T →. T * F
T →. F
F →. (E)
F → .id

I_5: Goto((I_6, id)
F → id.

I_{10}: Goto((I_7, F)
T → T * F.

I_4: Goto((I_7, ()
F → (. E)
E →. E + T
E →. T
T →. T * F
T →. F
F →. (E)
F → .id

I_5: Goto((I_7, id)
F → id.

I_{11}: Goto((I_8,))
F → (E).

I_6: Goto((I_8, +)

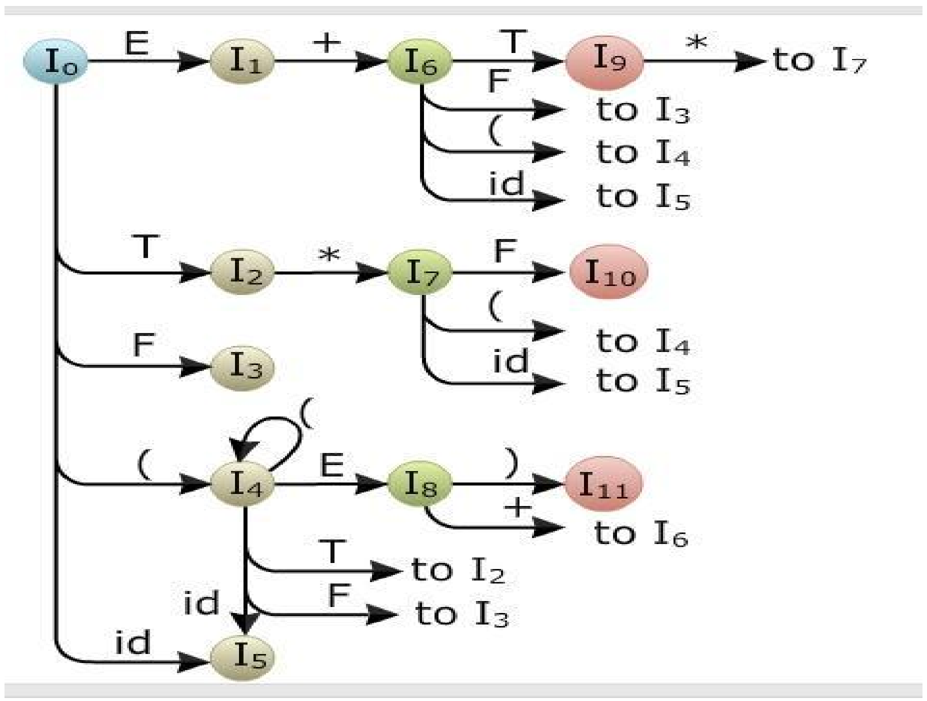
$E \rightarrow E +. T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow .id$

I_7: Goto((I_9, *)
$T \rightarrow T *. F$
$F \rightarrow . (E)$
$F \rightarrow .id$

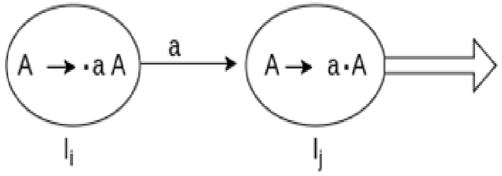
I_{10}: Goto((I_7, F)
$T \rightarrow T * F.$

I_{11}: Goto((I_8,))
$F \rightarrow (E).$

The DFA for the canonical set of SLR items is

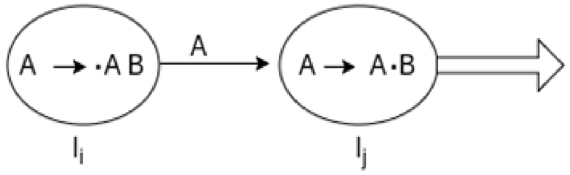


Construction of SLR Parsing Table:
The steps which use to construct SLR (1) Table is given below:
If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.



States	Action		Go to
	a	$	A
I_i	Sj		
I_j			

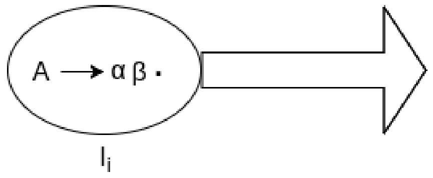
If a state (I_i) is going to some other state (I_j) on a variable/non-terminal then it corresponds to Goto part.



States	Action		Go to
	a	$	A
I_i			j
I_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
A -> αβ•
Follow(A) = {a}



States	Action			Go to	
	a	b	$	S	A
I_i	r2				

For the above example
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

The parsing table is

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6		s4		Acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			S4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Here, si means shift state i. ri means 'reduce by using production number I' and Acc means Accept, blank means error.

Goto:

Goto(I_0, E) = I_1
Goto(I_0, T) = I_2
Goto(I_0, F) = I_3
Goto(I_4, E) = I_8
Goto(I_4, T) = I_2
Goto(I_4, F) = I_3
Goto(I_6, T) = I_9
Goto(I_6, F) = I_3
Goto(I_7, F) = I_{10}

Action:

- **(i)** **Accept -** Action (1, $)
- **(ii)** **Error –** Blank entry cells in the table.
- **(iii)** **Shift –** For a specific state and for a specific terminal symbol, check DFA
- **(iv)** **Reduce –** Consider the final items and implement follow function to it.

Shift:

(I_0, id) $\rightarrow$ I_5
(I_0, () $\rightarrow$ I_4 and so on…

Reduce:
$I_{2,} I_{3,} I_{5,} I_{9,} I_{10 \text{ and}} I_{11}$ are final items.

Rules For Calculating Follow Function:

1. First put \$ (the end of input marker) in Follow(S), where S is the start symbol.
2. If there is a production $A \rightarrow \alpha B\beta$, $\beta \neq \in$ then everything in FIRST(β) except for ε is placed in FOLLOW(B).
   Follow (B) = First (β) - ε
3. If there is a production $A \rightarrow \alpha B$, then everything in FOLLOW(A) is in FOLLOW(B)
   Follow (B) = Follow (A)
4. If there is a production $A \rightarrow \alpha B\beta$, where FIRST($\beta$) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)
   Follow (B) = Follow (A)

$E \rightarrow E + T$ (Production number is 1)
$E \rightarrow T$ (Production number is 2)
$T \rightarrow T * F$ (Production number is 3)
$T \rightarrow F$ (Production number is 4)
$F \rightarrow (E)$ (Production number is 5)
$F \rightarrow id$ (Production number is 6)

Follow(E) = {\$,), +}
Follow(T) = {\$, +,), *}
Follow(F) = {\$, +, *,)}
Parsing of the input string **id * id + id** using the SLR parser table for the grammar

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

State	Stack	Input	Action
1	0	id * id + id \$	Shift
2	0 id5	* id + id \$	reduce by F →id
3	0 F3	* id + id \$	reduce by T → F
4	0 T2	* id + id \$	shift
5	0 T2 * 7	id + id \$	shift
6	0 T2 * 7 id 5	+ id \$	Reduce by F → id
7	0 T2 * 7 F 10	+ id \$	Reduce by T → T * F

8	0 T2	+ id $	Reduce by E → T
9	0 E 1	+ id $	Shift
10	0 E 1 + 6	id $	Shift
11	0 E1 + 6 id 5	id $	Reduce by F → id
12	0 E1 + 6 F 3	$	Reduce by T → F
13	0 E1 + 6 T 9	$	E → E + T
14	0 E1	$	Accept

The given input string is accepted by the grammar.

<u>LR(1) PARSING (OR) CLR(1) PARSING:</u>

CLR refers to canonical LR. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the greater 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 that where we place the final item. The look ahead always adds $ symbol for the argument production.

Example:
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, $ // add $
S → •AA, $ //Consider S` → •S, $, compute first ($) = {$}
A → •aA, a/b //Consider S → •AA, $, compute first(A$) = first(A) = {a,b}
A → •b, a/b

Rules:
For Closure Function:
$A \to \alpha.B\beta$, a
$B \to .\gamma$, First(βa)
For Reduce:
$A \to \alpha.$, a

I_0 State:
Add Augment production to the I0 State and Compute the Closure
I_0 = Closure (S\` → •S)
Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I_0 State becomes
I_0 = 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.
I_0 = S\` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I_1= Go to (I_0, S) = closure (S\` → S•, $) = S\` → S•, $

I_2= Go to (I_0, A) = closure (S → A•A, $)
Add all productions starting with A in I_2 State because "." is followed by the non-terminal. So, the I_2 State becomes
I_2= S → A•A, $
A → •aA, $
A → •b, $

I_3= 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 I_3 State becomes
I_3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b
Go to (I_3, a) = Closure (A → a•A, a/b) = (same as I_3)
Go to (I_3, b) = Closure (A → b•, a/b) = (same as I_4)

I_4= Go to (I_0, b) = closure (A → b•, a/b) = A → b•, a/b

I_5= Go to (I_2, A) = Closure (S → AA•, $) =S → AA•, $

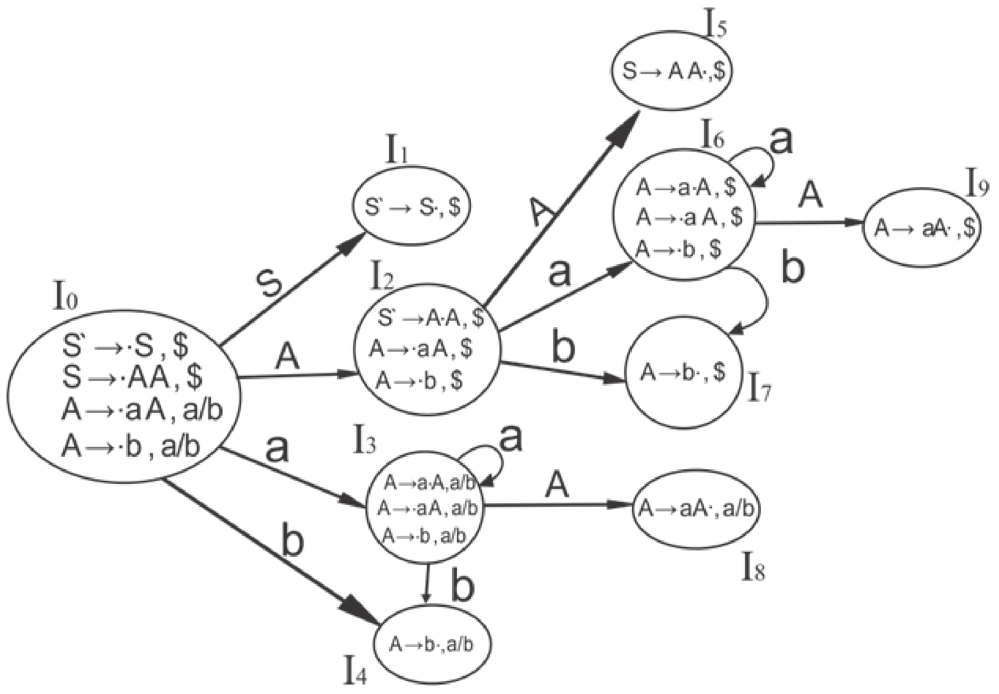
I_6= Go to (I_2, a) = Closure (A → a•A, \$)
Add all productions starting with A in I_6 State because "." is followed by the non-terminal. So, the I_6 State becomes
I_6 = A → a•A, \$
A → •aA, \$
A → •b, \$
Go to (I_6, a) = Closure (A → a•A, \$) = (same as I_6)
Go to (I_6, b) = Closure (A → b•, \$) = (same as I_7)

I_7= Go to (I_2, b) = Closure (A → b•, \$) = A → b•, \$

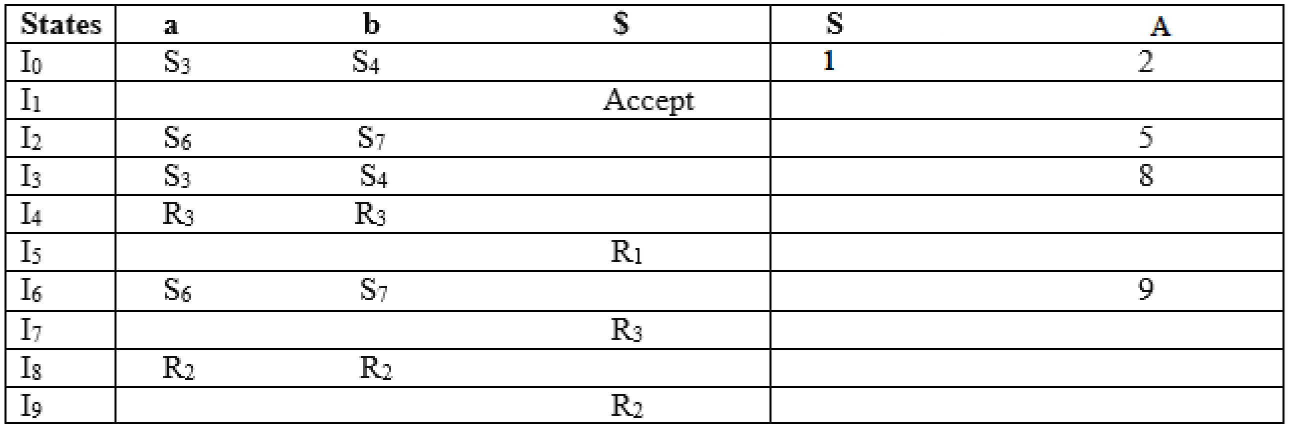
I_8= Go to (I_3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I_9= Go to (I_6, A) = Closure (A → aA•, \$) = A → aA•, \$

Drawing DFA:



Construction of parsing table:



States	a	b	$	S	A
I_0	S_3	S_4		1	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.
I_4 contains the final item which drives (A → b•, a/b), so action { I_4, a} = R3, action {I4, b} = R3.
I_5 contains the final item which drives (S → AA•, $), so action { I_5, $} = R1.
I_7 contains the final item which drives (A → b•,$), so action { I_7, $} = R3.
I_8 contains the final item which drives (A → aA•, a/b), so action { I_8, a} = R2, action {I8, b} = R2.
I_9 contains the final item which drives (A → aA•, $), so action { I_9, $} = R2.

<u>LALR (1) PARSING:</u>

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 lookahead is 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:
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

I_0 State:
Add Augment production to the I_0 State and Compute the Closure L
I_0 = Closure (S` → •S)
Add all productions starting with S in to I_0 State because "•" is followed by the non-terminal. So, the I_0 State becomes
I_0 = S` → •S, $
        S → •AA, $
Add all productions starting with A in modified I_0 State because "•" is followed by the non-terminal. So, the I_0 State becomes.
I_0 = S` → •S, $
        S → •AA, $
        A → •aA, a/b
        A → •b, a/b
I_1= Go to (I_0, S) = closure (S` → S•, $) = S` → S•, $

I_2= Go to (I_0, A) = closure (S → A•A, $)
Add all productions starting with A in I_2 State because "•" is followed by the non-terminal. So, the I_2 State becomes
I_2= S → A•A, $
        A → •aA, $
        A → •b, $

I_3= Go to (I_0, 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 I_3 State becomes
I_3 = A → a•A, a/b
        A → •aA, a/b
        A → •b, a/b
Go to (I_3, a) = Closure (A → a•A, a/b) = (same as I_3)
Go to (I_3, b) = Closure (A → b•, a/b) = (same as I_4)

I_4= Go to (I_0, b) = closure (A → b•, a/b) = A → b•, a/b
I_5= Go to (I_2, A) = Closure (S → AA•, $) =S → AA•, $
I_6= Go to (I_2, a) = Closure (A → a•A, $)

Add all productions starting with A in I_6 State because "•" is followed by the non-terminal. So, the I_6 State becomes
I_6 = A → a•A, $
        A → •aA, $
        A → •b, $

Go to (I_6, a) = Closure ($A \rightarrow a\bullet A$, \$) = (same as I_6)
Go to (I_6, b) = Closure ($A \rightarrow b\bullet$, \$) = (same as I_7)

I_7= Go to (I_2, b) = Closure ($A \rightarrow b\bullet$, \$) = $A \rightarrow b\bullet$, \$
I_8= Go to (I_3, A) = Closure ($A \rightarrow aA\bullet$, a/b) = $A \rightarrow aA\bullet$, a/b
I_9= Go to (I_6, A) = Closure ($A \rightarrow aA\bullet$, \$) $A \rightarrow aA\bullet$, \$

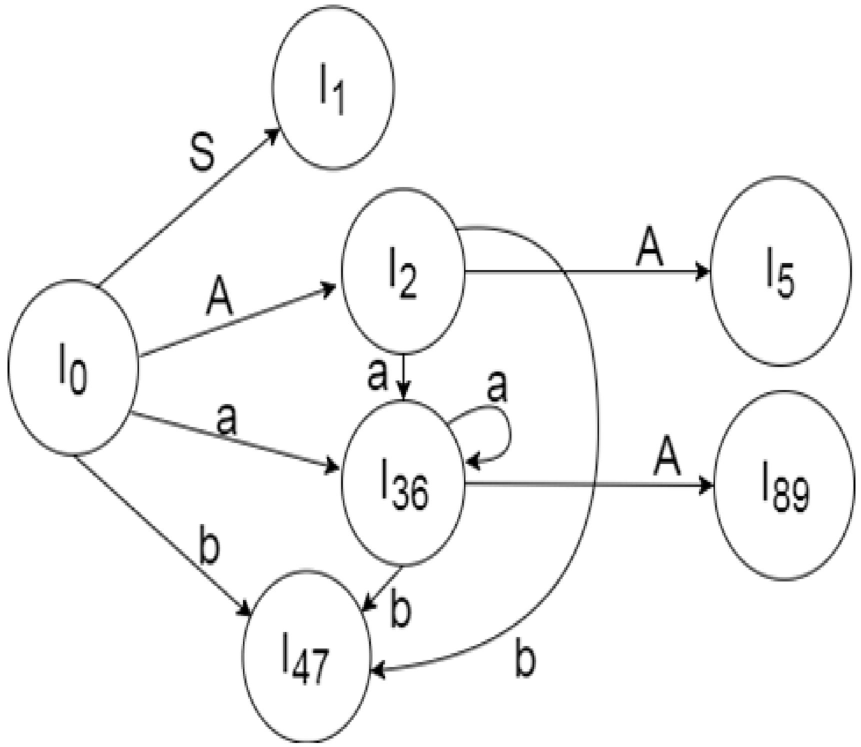
If we analyze then LR (0) items of I_3 and I_6 are same but they differ only in their lookahead.
I_3 = {$A \rightarrow a\bullet A$, a/b
$A \rightarrow \bullet aA$, a/b
$A \rightarrow \bullet b$, a/b}
I_6 = {$A \rightarrow a\bullet A$, \$
$A \rightarrow \bullet aA$, \$
$A \rightarrow \bullet b$, \$}
Clearly I_3 and I_6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I_{36}.

I_{36} = {$A \rightarrow a\bullet A$, a/b/\$
$A \rightarrow \bullet aA$, a/b/\$
$A \rightarrow \bullet b$, a/b/\$}

The I_4 and I_7 are same but they differ only in their look ahead, so we can combine them and called as I_{47}.
I_{47} = {$A \rightarrow b\bullet$, a/b/\$}

The I_8 and I_9 are same but they differ only in their look ahead, so we can combine them and called as I_{89}.
I_{89} = {$A \rightarrow aA\bullet$, a/b/\$}

Drawing DFA:



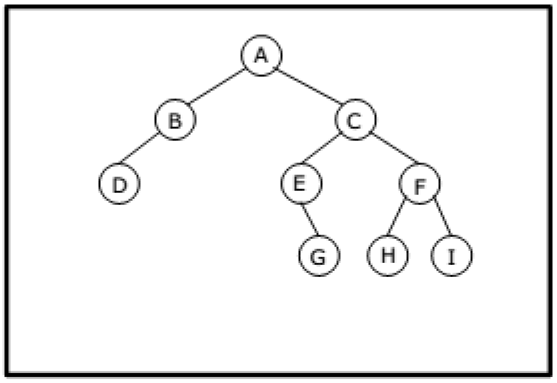
LALR (1) Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		

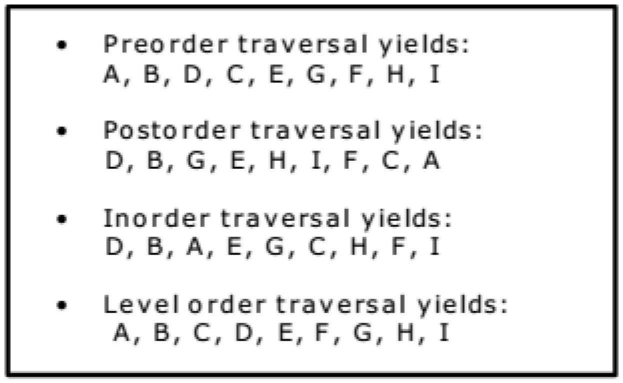
Parsing of input string happens in a similar way like previous parsers.

DIFFERENCE BETWEEN LL AND LR PARSER:

LL Parser	**LR Parser**
First L of LL is for left to right and second L is for leftmost derivation.	L of LR is for left to right and R is for rightmost derivation.
It follows the left most derivation.	It follows reverse of right most derivation.
Using LL parser, parser tree is constructed in top-down manner.	Parse tree is constructed in bottom-up manner.
In LL parser, non-terminals are expanded.	In LR parser, terminals are compressed/reduced.
Starts with the start symbol(S).	Ends with start symbol(S).
Ends when stack used becomes empty.	Starts with an empty stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parser tree.
LL is easier to write.	LR is difficult to write.
Example: LL(1)	Example: LR(0), SLR(1), LALR(1), CLR(1)



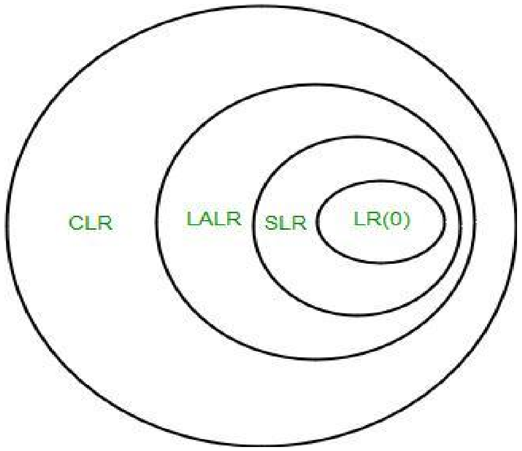
Binary Tree



- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

NOTE:
If number of states LR(0) = n1, number of states SLR = n2, number of states LALR = n3, number of states CLR = n4 then, n1 = n2 = n3 <= n4



DIFFERENCE BETWEEN LR(0) AND LR(1) ALGORITHM

- LR(0) requires just the canonical items for the construction of a parsing table, but LR(1) requires the lookahead symbol as well.
- LR(0) reduces at all items whereas LR(1) reduces only at lookahead symbols.
- LR(0) = canonical items and LR(1) = LR(0) + lookahead symbol.
- In canonical LR(0) collection, state/item is in the form:
  E.g. $I_2 = \{ C \rightarrow AB\bullet \}$

whereas in canonical LR(1) collection, state/item is in the form:
E.g. I_2 = { C →d•, e/d} where e & d are lookahead symbol.

- Construction of canonical LR(0) collection starts with C = {closure({S‘→.S})} whereas Construction of canonical LR(1) collection starts with C = {closure({S‘→.S, $})} where S is the start symbol.
- LR(0) algorithm is simple than LR(1) algorithm.
- In LR(0), there may be conflict in parsing table while LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table.

COMPUTATION OF LEADING AND TRAILING:

LEADING(A):
(i) If A→αaβ, α is single variable or ε
LEADING(A) = {a}
(ii) If A→Baβ
LEADING(A) = LEADING(B)

Example:
S → a | ↑ | (T)
T → T , S | S

S → a | ↑ | (T) // Rule (i)
LEADING(S) = {a, ↑, (}

T → T , S //Rule (i) and Rule(ii)
LEADING(T) = {, LEADING(T) // Same meaning, so remove LEADING(T)
and
T → S // Rule (ii)
= {, LEADING(S)
= {, a, ↑, (}

Trailing(S)=

TRAILING(A):
(i) If A→αaβ, β is single variable or ε
TRAILING(A) = {a}
(ii) If A→βaB
TRAILING(A) = TRAILING(B)

Example:
S → a | ↑ | (T)
T → T , S | S

S → a | ↑ | (T)

TRAILING(S) = {a, ↑,)}
$T \rightarrow T , S$
TRAILING(S) = {, TRAILING(S)}
and
$T \rightarrow S$ // Rule (ii)
= {, TRAILING(S)}
= {, a, ↑,)}

Operator precedence relation table:

1. Set $ < Leading(S) and Trailing(S) > $ (where S is the start symbol)
2. $A \rightarrow X_1, X_2, X_3 \ldots X_n$
   a. If X_i and $X_{i+1} \in T$,
      $\mathbf{X_i = X_{i+1}}$
   b. If X_i and $X_{i+2} \in T$ and $X_{i+1} \in N$ or V,
      $\mathbf{X_i = X_{i+2}}$
   c. If $X_i \in T$, $X_{i+1} \in N$ or V
      $\mathbf{X_i < Leading(X_{i+1})}$
   d. If $X_i \in$ V or N, $X_{i+1} \in T$
      $\mathbf{Trailing(X_i) > X_{i+1}}$

As per rule: 1 LEADING(S) = {, ,a, ↑, (}
$ < ,
$ < a
$ < ↑
$ < (

TRAILING(S) = {, , a, ↑,)}
, > $
a > $
↑ > $
) > $

	a	↑	(	)	,	$
a				>	>	>
↑				>	>	>
(	<	<	<	=	<	

)				>	>	>
,	<	<	<	>	>	
$	<	<	<			

As per rule: 2(b) S → (T)

(=)

As per rule: 2(c) S → (T)

(< Leading(T) = {, a, ↑,)}

(< ,

(< a

(< ↑

(< (

And

T → T , S

, < Leading(S) = {a, ↑, (}

, < a

, < ↑

, < (

As per rule: 2(d) S → (T)

Trailing(T) = {, a, ↑,)} >)

, >)

a >)

↑ >)

) >)

And

T → T , S

Trailing(T) = {, a, ↑,)} > ,

, > ,

a > ,

↑ > ,

) > ,