KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

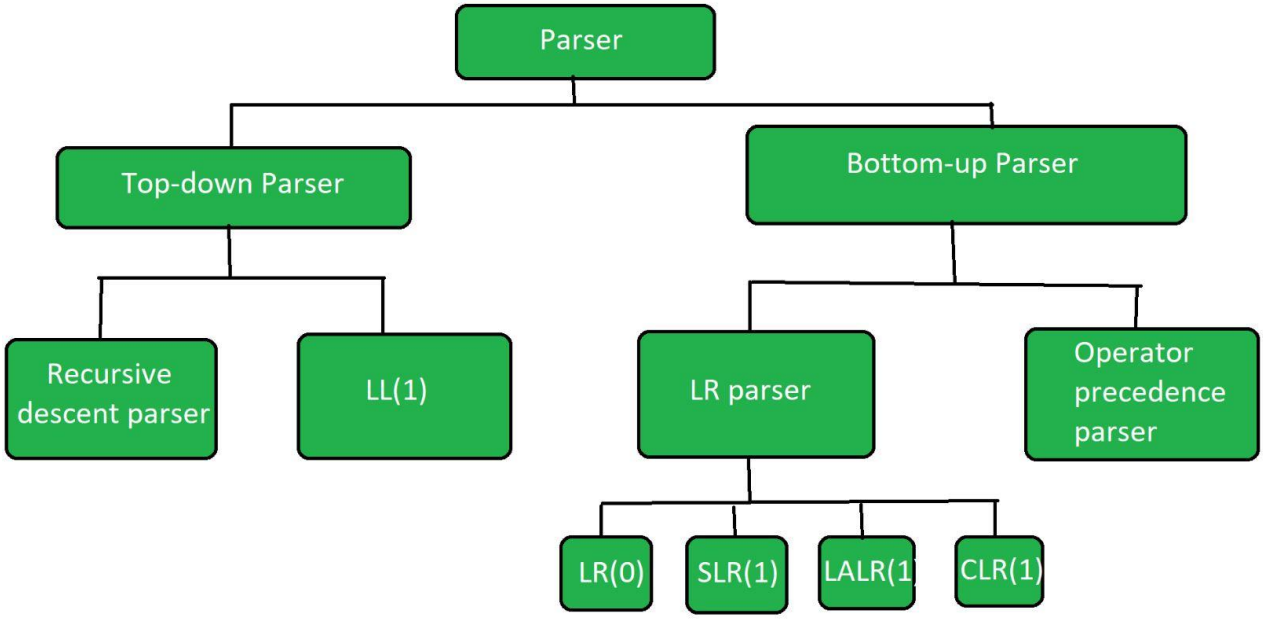
MODULE 3

Bottom-Up Parsing: Shift Reduce parsing - Operator precedence parsing (Concepts only)

LR parsing – Constructing SLR parsing tables, Constructing, Canonical LR parsing tables and Constructing LALR parsing tables.

PARSING

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer.*



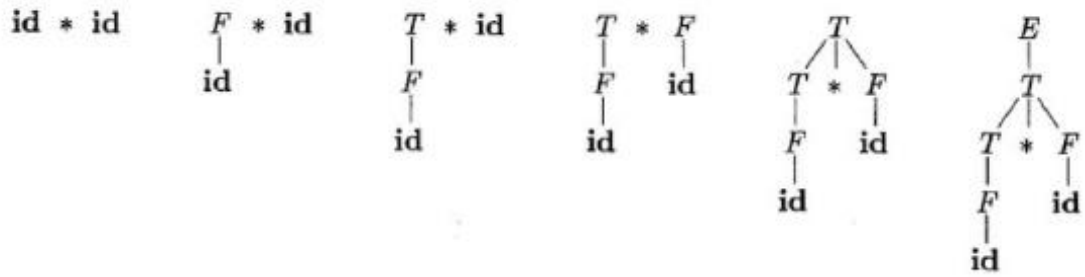
3.1 BOTTOM UP PARSING

A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse.

Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production.

When it finds such a substring, it *reduces* it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

- Bottom-up parse for **id*id**



3.1.1 SHIFT REDUCE PARSING

- Shift reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).
- This can be considered as the process of "reducing" a string w to the start symbol of

a grammar.

- At each reduction step parser searches for substrings of the working string that match the right side of some production.
- When it finds such a substring, it reduces it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.
- In Shift-reduce parsing a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	w $

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty as follows:

STACK	INPUT
$ S	$

- Upon entering this configuration, the parser halts and announces successful completion of parsing.
- There are actually four possible actions a shift-reduce parser can make:
  - **Shift**
  - **Reduce**
  - **Accept**
  - **Error**

1. **Shift:** The next input symbol is shifted onto the top of the stack.
2. **Reduce:** The parser knows the right end of the string to be reduced must be at the top of the stack. It must then locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept:** Announce successful completion of parsing.
4. **Error:** Discover a syntax error has occurred and calls an error recovery routine.

EXAMPLE

Following figure steps through the actions a shift-reduce parser might take in parsing the input string $id_1 * id_2$ according to the expression grammar.

E→ E + T | T

T→ T * F | F

F→ (E) | id

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

*Configurations of a shift-reduce parser on input idl*id2*

3.1.1.1 OPERATOR PRECEDENCE PARSING

- Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars.
- In an operator grammar, no production rule can have:
  - at the right side (no production).
  - two adjacent non-terminals at the right side.
- This property enables the implementation of efficient *operator-precedence parsers*.

EXAMPLE

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |
B→b	O→+|*|/	E/E | id
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three precedence relations between certain pairs of terminals as follows:

Relation	Meaning
$a <\cdot b$	*a* yields precedence to *b (b has higher precedence than a)*
$a =\cdot b$	*a* has the same precedence as *b*
$a \cdot> b$	*a* takes precedence over *b (b has lower precedence than a)*

- The intention of the precedence relations is to find the handle of a right sentential form,

  <. with marking the left end,

  =· appearing in the interior of the handle, and

  .> marking the right end.

- In our input string $a1a2...an$, we insert the precedence relation between the pairs of terminals.

- Example: Consider the string id + id * id and the grammar is:

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

- The corresponding precedence relations is

	id	+	*	$
Id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

- Then the string with the precedence relations inserted is:

  $ <. id .> + <. id .> * <. id .> $

- <. is inserted between the leftmost $ and id since <. is the entry in row $ and column id.

Handle And Handle Pruning

- In our input string $a1a2...an$, we insert the precedence relation between the pairs of terminals.
- Example: Consider the string id + id * id and the grammar is:
- Handle is a substring that matches with the right side of the production and if the substring matches with the right side of the production, it is reduced with the non-terminal on the left side of the production

EXAMPLE

Consider the grammar

E → E+E

E → E*E

E → (E)

E→ id

And the input string is $id_1 + id_2 * id_3$

The rightmost derivation is:

E → E + E

➔ **E + E * id_3**
➔ **E + id_2 * id_3**
➔ **Id_1 + id_2 * id_3**

In the above derivation, the underlined substrings are called handles.

HANDLE PRUNING

The process of obtaining rightmost derivation in reverse order is called "handle pruning". (i.e.) if w is a sentence or string of the grammar at hand, then w = γ_n where γ_n is the n^{th} right sentinel form of some rightmost derivation.

HANDLE PRUNING

- A "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation
- The handles during the parse of $\mathbf{id_1 * id_2}$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id_1} * \mathbf{id_2}$	$\mathbf{id_1}$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id_2}$	F	$T \rightarrow F$
$T * \mathbf{id_2}$	$\mathbf{id_2}$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

- The leftmost substring that matches the body of some production need not be a handle

3.1.2 LR PARSING

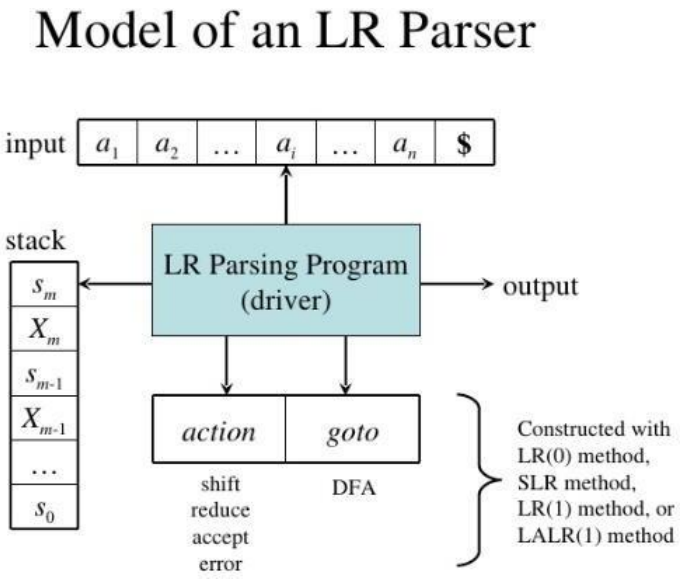
LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.

The technique is called LR(k) parsing; the "L" is for left to right scanning of input symbol, the "R" for constructing right most derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decision.

There are three types of LR parsing,

- LR(0)
- SLR (Simple LR)
- CLR (Canonical LR)
- LALR (Lookahead LR)

The schematic form of LR parser is given below.



Requirements of LR parser:

- Input Buffer
- Stack

- Parsing Table
- LR Parsing Program

Input Buffer

The parsing program reads characters from an input buffer one at a time.

Stack

- Parsing program uses a stack to store string of the form $s_0X_1s_1X_2s_2X_3$ ……X_ms_m where s_m is on the top.
- X_i is a grammar symbol and S_i is the state
- Each state symbol summarizes the information contained in the stack below it
- Combination of state symbol on stack top and current input symbol are used to index the parsing table and determine the shift reduce parsing decision

Parsing Table

Consist of 2 parts

1. parsing action function action
2. goto function goto

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:

i. Shift *j,* where *j* is a state.
ii. Reduce by a grammar production A---> β.
iii. Accept.
iv. Error.

Goto

This function takes a state and grammar symbol as arguments and produces a state. If GOTO [Ii ,A] = Ij, the GOTO also maps a state *i* and nonterminal A to state *j.*

Parsing Program works as follows

- It determines s_m, the state currently on top of stack and a_i the current input symbol
- It consults parsing action table entry for action[s_m,a_i] which can have 4 values:
- The function goto takes a state and grammar symbol as arguments and produces a state.

1. **Shift s, where s is a state**
2. **Reduce by a grammar production A→b**
3. **Accept**
4. **Error**

- The function goto takes a state and grammar symbol as arguments and produces a state.
- A configuration of LR parser is a pair whose first component is stack contents and second component is remaining input:

$$(s_0 X_1 s_1 X_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

- The next move of the parser is determined by reading ai, the current input symbol and sm, state on stack top and then consulting parsing action table entry action[sm,ai].
- The configuration resulting after each of four types of moves are as follows:

1. If action $[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0 X_1 s_1 X_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

Here the parser has shifted both the current input symbol a_i and the next state s, which is given in action $[s_m, a_i]$, onto the slack; a_{i+1} becomes the current input symbol.

2. If action $[s_m, a_i]$ =reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n \$)$$

where s = *goto* $[s_{m-r}, A]$ and r is the length of β , the right side of the production. Here the parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for *goto*$[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move.

3. If action $[s_m, a_i]$ = accept, parsing is completed.
4. If action $[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

LR Parsing Algorithm

INPUT : Input string w, LR-Parsing table with functions ACTION and GOTO for a 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 then executes the following program until an accept or error action is encountered.

set *ip* to point to the first symbol of w$

repeat forever begin

Iet s be the state on top of the slack and

a the symbol pointed to by *ip*;

if *action* [s, *a*]= shift s' then begin

push a then s' on lop of the stack;

advance *ip* to the next input symbol

end

else if *action* [s,a] = reduce $A \rightarrow \beta$ then begin

pop 2*|β| symbols off the stack ;

let s' be the state now on top *of* the stack;

push *A* then *goto* [s', A] on top of the stack;

output *the* production $A \rightarrow \beta$

end

else if *action* [s,a] = accept then

return

else error();

end

Different types of LR Parsers

All LR parsers use the same parsing algorithm but the difference is only in terms of the construction of parsing tables. There are three widely used algorithms available for constructing an LR parsing Tables. Different types of LR parsers are:

- **LR(0)**
  - The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR(0) parsing.
  - L stands for the left to right scanning
  - R stands for rightmost derivation in reverse
  - 0 stands for no. of input symbols of lookahead.
- **SLR(l) - Simple LR**
  - Works on smallest class of grammar.
  - Few number of states, hence very small table.
  - Simple and fast construction.
- **LR(1) - LR parser OR CLR Parsing**
  - Also called as Canonical 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).

Reasons for attractiveness 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.

Drawbacks 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.

Construction of LR Parsing Table

3 techniques for constructing LR parsing table for a grammar :-

1. **SLR**
2. **CLR**
3. **LALR**

Simple LR (SLR)

Easy to implement

Least powerful

Canonical LR

Most powerful

Most expensive

LookAhead LR(LALR)

Intermediate in power and cost between other 2

LR(0) PARSING

Steps for constructing the LR parsing table :

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

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

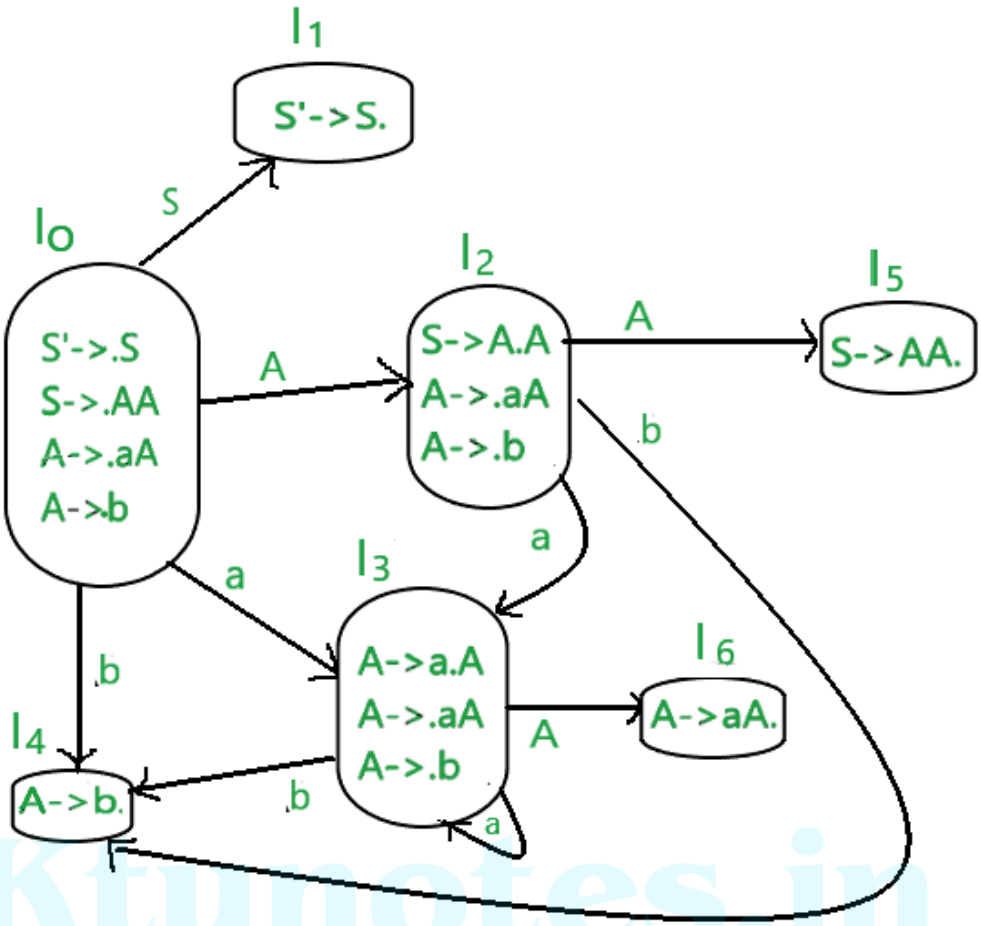
S–>AA
A–>aA|b

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



	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		

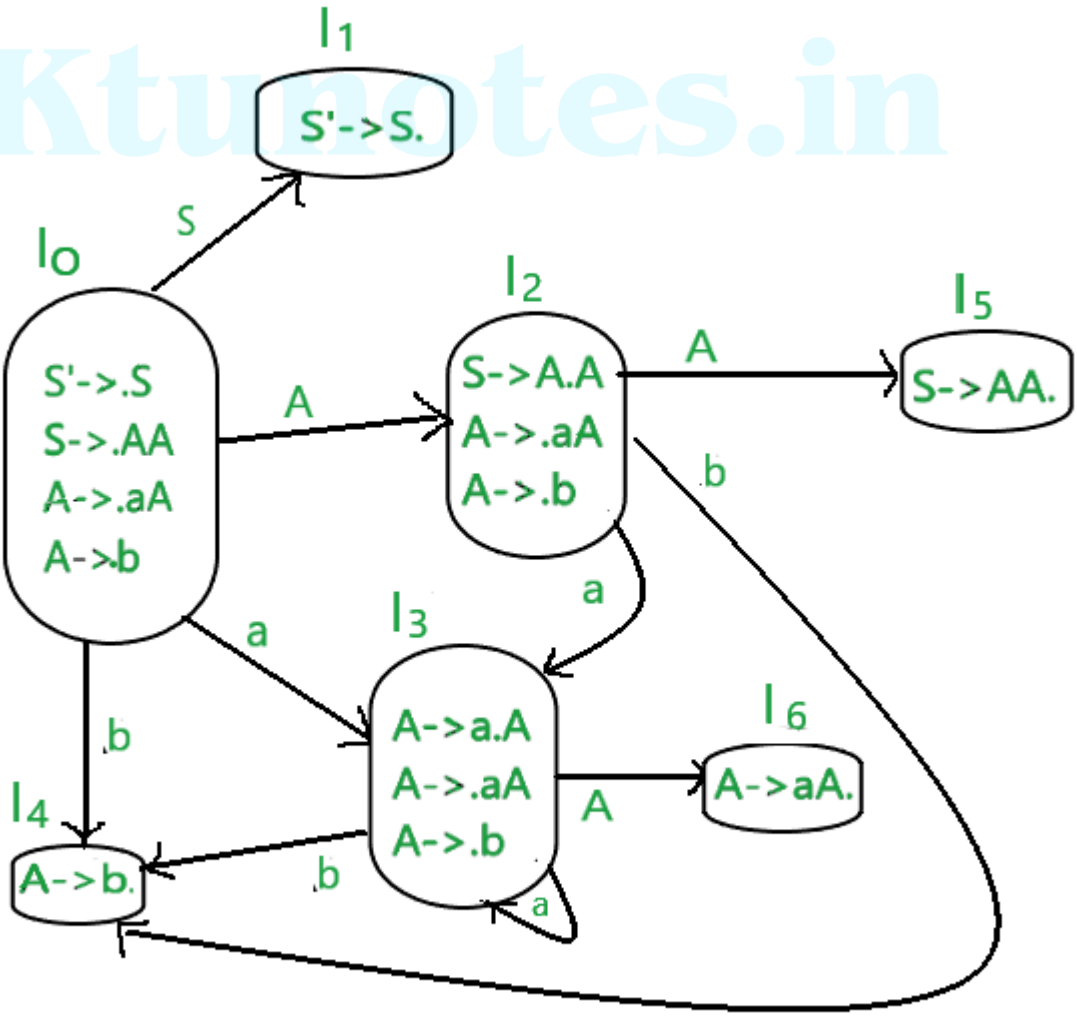
SLR Parser :

SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items

Steps for constructing the SLR parsing table :

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

EXAMPLE – Construct LR parsing table for the given context-free **grammar**
S–>AA
A–>aA|b
Solution:
STEP1 – 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]
STEP2 – 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}

STEP3 –

Find FOLLOW of LHS of production

FOLLOW(S)=$

FOLLOW(A)=a,b,$

To find FOLLOW of non-terminals, please read follow set in syntax analysis.

STEP 4-

Defining 2 functions:goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

	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		
6	R2	R2	R2		

Limitations Of SLR Parser

For an SLR parser, if we are in a state with the item A -> α ., then we reduce by A -> α iff the next input symbol 'a' belongs to Follow(A).

This is actually a weak rule and can lead to erroneous results and/or shift/reduce conflicts.

3.1.2.1 CONSTRUCTING CANONICAL LR(CLR) PARSING TABLES

- In SLR method, the state i makes a reduction by A→α when the current token is **a**:
  - if the A→α.is in the I_i and **a** is FOLLOW(A)
  - In some situations, when state i appears on stack top, the viable prefix βα on the stack is such that βA cannot be followed by a in right sentential form.
- Thus the reduction by A→α would be invalid on input a.
- Because of that we go for : Canonical LR Parser
  - In this, it is possible to carry more information in the state that will allow us to avoid some of these invalid reductions
- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

  $A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item

  (**a** is a terminal or end-marker.)

- Such an object is called LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect in an item of the form $[A \rightarrow \alpha.\beta,a]$, where β is not ∈.
  - But an item of the form $[A \rightarrow \alpha.,a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.
- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by A→α only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that closure and goto operations work a little bit different.

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha.B\beta,a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,b$ will be in the closure(I) for each terminal b in FIRST(βa) .

Construction of CLR (canonical LR) Parsing Tables

Algorithm: Construction of canonical- LR parsing tables.

INPUT: An augmented grammar G'.

OUTPUT: the canonical-LR parsing table functions ACTION AND GOTO for G'.

METHOD:

1. **Construct C' = { I_0, I_1, ……., I_n}, the collection of sets of LR(1) items for G'.**
2. **State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.**

   (a) **If [$A \rightarrow \alpha.a\beta,b$] is in I_i and GOTO (I_i, a) = I_j, then set ACTION [i, a] To "shift j". Here a must be a terminal.**

   (b) **If [$A \rightarrow \alpha.$, a] is in I_i, $A \neq S'$, then set ACTION[i, a] to "reduce $A \rightarrow \alpha$."**

   (c) **If [$S' \rightarrow S.$, $] is in I_i, then set ACTION[i, $] to "accept".**

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm falls to produce a parser in this case.

3. **The goto transitions for state i are constructed for all non terminals A using the rule: if GOTO(I_i, A) = I_j, then GOTO[i, A]= j.**
4. **All entries not defined by rules (2) and (3) are made "error".**
5. **The initial state of the parser is the one constructed from the set of items containing [$S' \rightarrow .S$, $].**

EXAMPLE

Construct CLR parsing table for the flowing grammar

S→ CC

C→ cC| d

AUGMENTED GRAMMAR

S'→ S

S→ CC

C→ cC

C→ d

CANONICAL COLLECTION OF LR(1) ITEMS

I_0

S' → •S , $
S→ •CC, $
C→ •cC, c | d
C→ •d, c | d

I_1 = GOTO (I_0, S)

S' → S•, $

I_2 = GOTO (I_0, C)

S → C•C, $
C → •cC, $
C → •d, $

I_3 = GOTO (I_0, c)

C → c•C, c | d
C → •c C , c | d
C → •d, c | d

I_4 = GOTO (I_0, d)

C → d• , c | d

I_5 = GOTO (I_2, C)

S → CC•, $

I_6 = GOTO (I_2, c)

C → c•C, $
C → •c C ,$
C → •d , $

I_7 = GOTO (I_2, d)

C → d• , $

I_4 = GOTO (I_0, d)

C → d• , c | d

I_5 = GOTO (I_2, C)

S → CC•, $

I_6 = GOTO (I_2, c)

C → c•C, $
C → •c C ,$
C → •d , $

I_7 = GOTO (I_2, d)

C → d• , $

I_8 = GOTO (I_3, C)

C → c C•, c | d

I_3 = GOTO (I_3, c)

C → c• C , c | d
C→ •cC, c | d
C→ •d, c | d

I_4 = GOTO (I_3, d)

C→ d •, c | d

I_9 = GOTO (I_6, C)

C → c C• , $

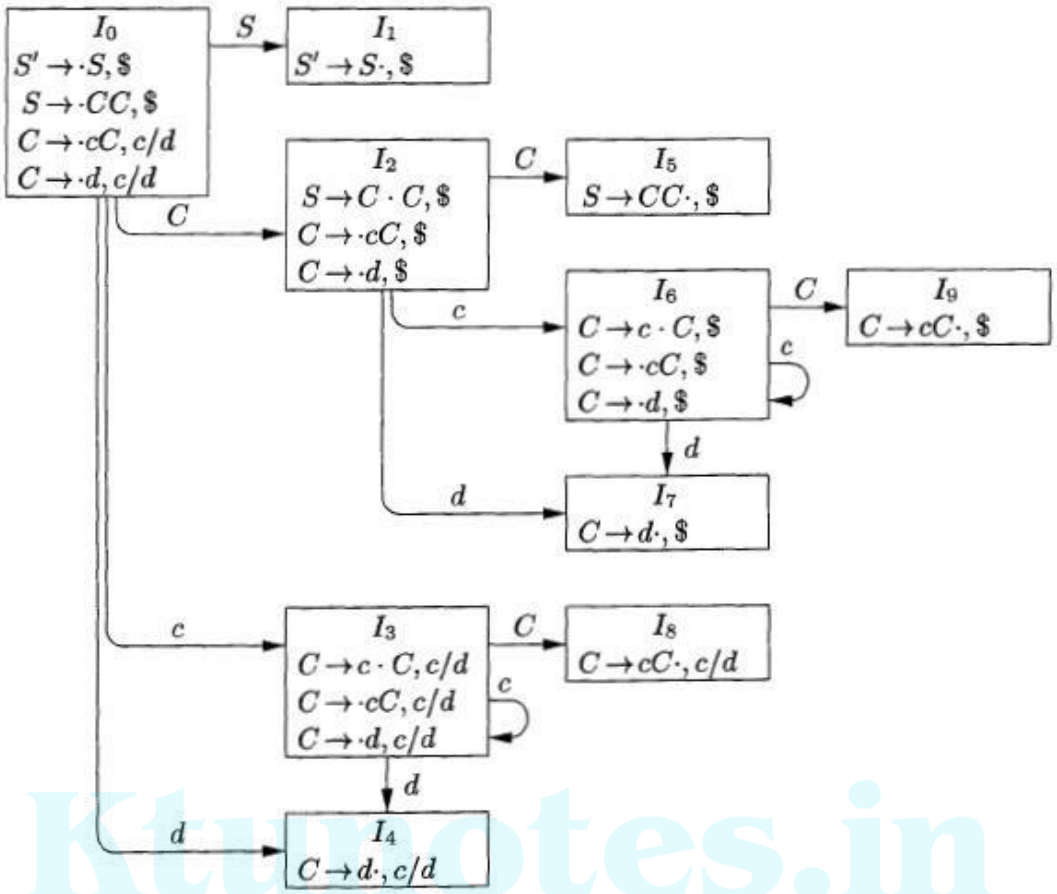
I_6 = GOTO (I_6, c)

C → c• C ,$
C→ •cC, $
C→ •d, $

I_7 = GOTO (I_6, d)

C→ d•, $

GOTO GRAPH FOR THE GRAMMAR



NUMBER THE PRODUCTIONS

1. S→ CC
2. C→ cC
3. C→ d

PARSING TABLE

I_1 = S' → S• , $

I_4 = C → d• , c | d

I_5 = S → CC•, $

I_7 = C → d• , $

I_8 = C → c C•, c | d

I_9 = C → c C• , $

STATE	ACTION			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		

3.1.2.2 CONSTRUCTING LALR PARSING TABLE

- **LALR** stands for **Lookahead LR.**
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- *yacc* creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

- Canonical LR(1) Parser ➔ LALR Parser

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component

Ex: $S \rightarrow L.=R,\$$ ➔ $S \rightarrow L.=R$ Core

$R \rightarrow L.,\$$ $R \rightarrow L.$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id.,=$ A new state: I_{12}: $L \rightarrow id.,=$

➔ $L \rightarrow id.,\$$

I_2: $L \rightarrow id.,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

We need to do following update in the parsing table.

- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
  I_4: C → d. , c/d I_7: C → d., $
- Replace I_4 and I_7 by a new state I_{47} consisting of (C → d., c/d/$)
- Similarly I_3 & I_6 and I_8 & I_9 form pairs
- Merge LR(1) items having the same core

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing – table functions ACTION and GOTO for G'

METHOD:

1. Construct C = { $I_0, I_1, \ldots, I_n$ }, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let C' = {$J_0, J_1, \ldots, J_m$}, the collection of sets of LR(1) items. The parsing action for state i are constructed from J_i in the same manner as in Algorithm. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1)
4. The GOTO table is constructed as follows. if J is the union of one or more sets of LR(1) items, that is, J = $I_1 \cup I_2 \cup \ldots \cup I_k$, then the $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1,X). Then GOTO (J, X) = K.

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

****

MODULE - 3

BOTTOM UP PARSING

Bottom Up Parsing

It is the reverse of top down parsing. In this we start with input symbol and finally leads to starting symbol.

eg:-

$S \longrightarrow aABe$
$A \longrightarrow Abc/b$
$B \longrightarrow d$

Derive Bottom up parsing for the string :
abbcde

→ abbcde
→ aAbcde
→ aAde
→ aABe
→ S

Handles

A substring that is matching with the left side of the given production is called a Handle.

$E \longrightarrow E+T/T$
$T \longrightarrow T*F/F$
$F \longrightarrow id$

Derive id + id * id

Leftmost Derivation

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

Rightmost Derivation

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

Handle Pruning

The Rightmost Derivation in reverse can be obtained by Handle Pruning.

$E \rightarrow E + E / E * E / (E) / id$

$id_1 + id_2 * id_3$ - Derive

$E \rightarrow E + E$
$\rightarrow E + E * E$
$\rightarrow E + E * id_3$
$\rightarrow E + id_2 * id_3$
$\rightarrow id_1 + id_2 * id_3$

Right Sentential Form	Handle	Action
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id_1$
$E + id_2 * id_3$	id_2	$E \rightarrow id_2$
$E + E * id_3$	id_3	$E \rightarrow id_3$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

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

Rightmost Derivation

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

Right Sent.	Handle	Action
$id * (id + id)$	id	$E \rightarrow id$
$E * (id + id)$	id	$E \rightarrow id$
$E * (E + id)$	id	$E \rightarrow id$
$E * (E + E)$	$E + E$	$E \rightarrow E + E$
$E * (E)$	(E)	$E \rightarrow (E)$
$E * E$	$E * E$	$E \rightarrow E * E$
E		

Shift Reduce Parser (SR Parser)

To implement SRP we use a stack to hold grammar symbols and an input buffer to hold the string 'w' to be parsed. There are 4 possible actions :-

(i) Shift

(ii) Reduce

(iii) Accept

(iv) Error

1. Construct shift reduce parser for the input string $w = cdcd$

$S \rightarrow CC$
$C \rightarrow cC/d$

$S \rightarrow CC$ $S \rightarrow CcC$
$S \rightarrow CcdC$
$S \rightarrow cCcd$
$S \rightarrow cCcd$
$S \rightarrow cdcd$

Stack	Input Buffer	Action
$	cdcd$	Shift c
$c	dcd$	Shift d
$cd	cd$	Reduce C→d
$cC	cd$	Reduce C→cC
~~SHIFT~~		
$cC	cd$	Shift c
$cCc	d$	Shift d
$cCcd	$	Reduce C→d
$cCcC	$	Reduce C→cC
$CcC	$	Reduce C→cC
$CC	$	Reduce S→CC
$S	$	Accept

2. $w = abbcde$

$S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$

$S \rightarrow aABe$
$\rightarrow aAde$
$\rightarrow aAbcde$
$\rightarrow abbcde$

Stack	Input Buffer	Action
$	abbcde$	Shift a
$a	bbcde$	Shift b
$ab	bcde$	Reduce A→b
$aA	bcde$	Shift b
$aAb	cde$	Shift c
$aAbc	de$	Reduce A→Abc
$aA	de$	Shift d
$aAd	e$	Reduce B→d
~~Shift e~~		
$aAB	e$	Shift e
$aABe	$	Reduce S→aABe
$S	$	Accept

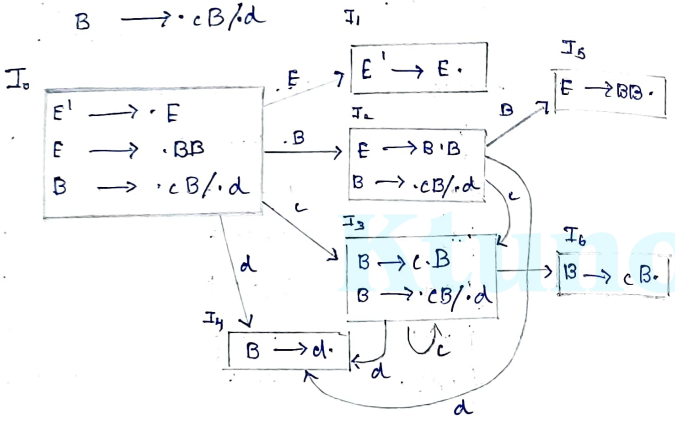
LR PARSING

1. $E \rightarrow BB$
   $B \rightarrow cB/d$

Input string cddd

Ans :- Augmented Grammer

$E' \rightarrow \cdot E$
$E \rightarrow \cdot BB$
$B \rightarrow \cdot cB/\cdot d$



Construction of LR Parsing Table.

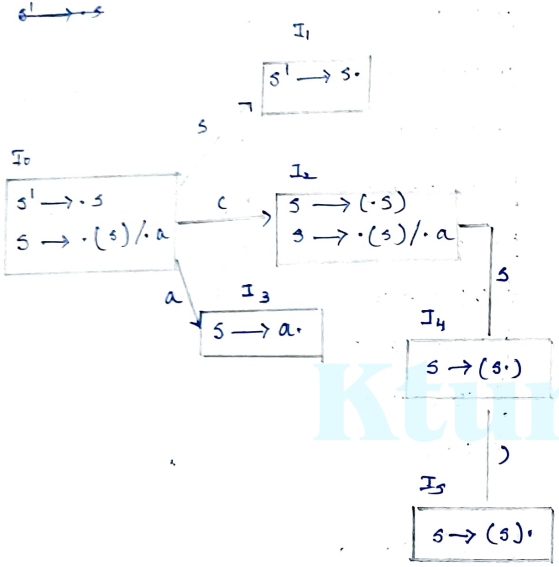
	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Stack	Input	Action
$0	cc dd $	Shift S3
$0C3	cdd $	Shift S3
$0C3C3	dd $	Shift S4
$0C3C3d4	d $	Reduce r_3
$0C3C3B5	d $	
$0C3C3B6	d$	Reduce r_2
$0C3C3B6	d$	Reduce r_2
$0B2d4	d$	Shift S_4
$0B2d4	$	Reduce r_3
$0B2B5	$	Reduce r_1
$0E1	$	Accept

2. $S \rightarrow (S)/a$

Check whether (a) is accepting or not.

$S' \rightarrow \cdot S$ ①
$S \rightarrow \cdot (S)/\cdot a$ ②
(a)



	Action				Goto
	(	)	a	$	S
I_0	S2		S3		1
I_1				Accept	
I_2	S2		S3		4
I_3	r2	r2	r2	r2	
I_4		S5			
I_5	r1	r1	r1	r1	

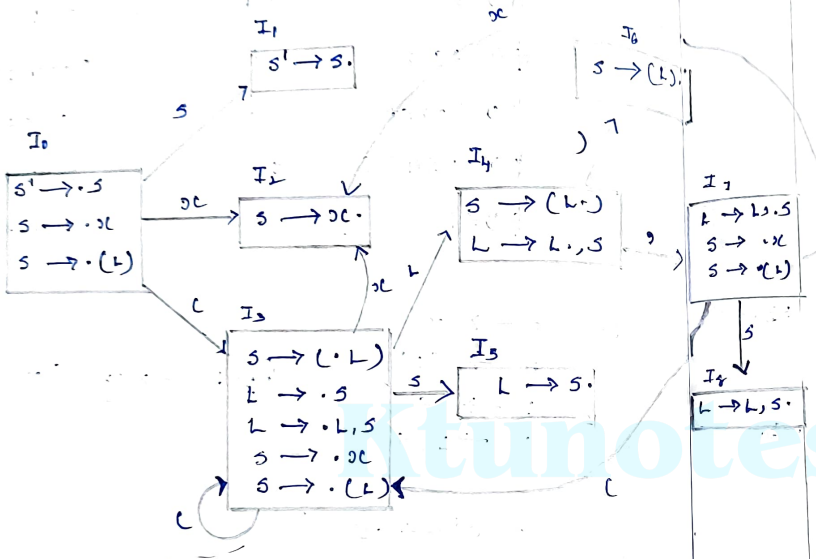
Stack	Input	Action
$0	(a)$	Shift S2
$0(2	a)$	Shift S3
$0(2a3	)$	Reduce r2
$0(2aS4	)$	Shift S5
$0(2S4)5	$	reduce r1
$0S1	$	Accept

3. $S \rightarrow x$
$S \rightarrow (L)$
$L \rightarrow S$
$L \rightarrow L, S$

Input string (x, (x))

0 $S' \rightarrow \cdot S$
1 $S \rightarrow (L)\cdot x$
2 $S \rightarrow \cdot (L)$
3 $L \rightarrow \cdot S$
4 $L \rightarrow L, S$

I_0
$S' \rightarrow S$
$S \rightarrow \cdot x$
$S \rightarrow \cdot (L)$



	Action					Goto	
	x	(	)	,	$	S	L
I_0	S2	S3				1	
I_1					Accept		
I_2	r1	r1	r1	r1	r1		
I_3	S2	S3				5	4
I_4			S6	S7			
I_5	r3	r3	r3	r3	r3		
I_6	r2	r2	r2	r2	r2		
I_7	S2	S3				8	
I_8	r4	r4	r4	r4	r4		

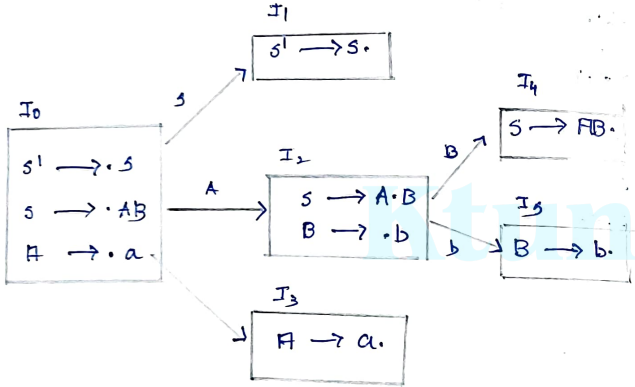
Stack	Input	Action
$0	(x,(x))	Shift S3
$0(3	x,(x))	Shift S2
$0(3x2	,(x))	Reduce r1
$0(3S5	,(x))	Reduce r3
$0(3L4	,(x))$	Shift S7
$0(3L4,7	(x))$	

SLR Parsing

S → AB
S → ·AB

S → AB
A → a
B → b

⓪ S' → ·S
① S → ·AB
② A → ·a
③ B → ·b



	FIRST	FOLLOW
S → AB	{a}	{$}
A → a	{a}	{b}
B → b	{b}	{$}

	Action			Goto		
	a	b	$	S	A	B
I_0	S3			1	2	
I_1			Accept			
I_2		S5				4
I_3		r2				
I_4			r1			
I_5			r3			

Stack	Input	Action
$0	ab$	Shift S3
$0a3	b$	Reduce r2
$0A2	b$	Shift S5
$0A2b5	$	Reduce r3
$0A2B4	$	Reduce r1
$0S1	$	Accept

CLR(1) Parsing

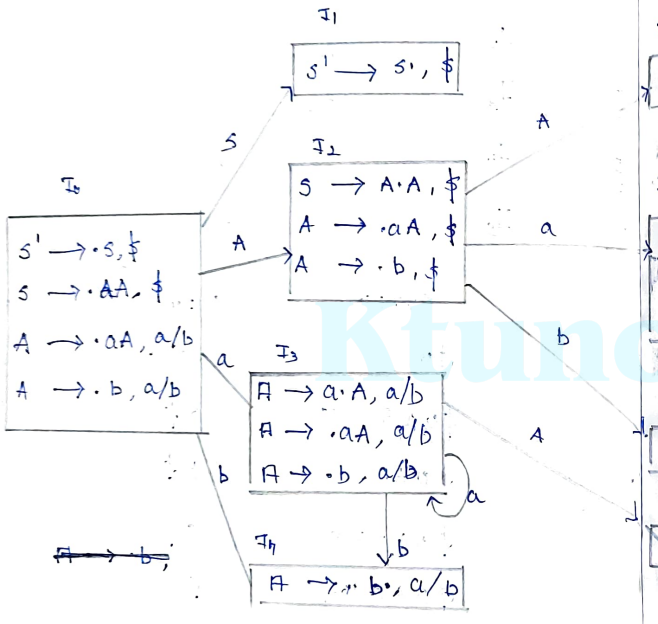
$S \rightarrow AA$

$A \rightarrow aA / b$

$S' \rightarrow .S$ ⓪

$S \rightarrow .AA$ ①

$A \rightarrow .aA$ ② / $.b$ ③



	Action			Goto	
	a	b	$	S	A
I_0	s3	s4		1	2
I_1			Accept		
I_2	s6	s7			5
I_3	s3	s4			8
I_4	r3	r3			
I_5			r1		
I_6	s6	s7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

Stack	Input	Action
$0	abab$	shift s3
$0a3	bab$	shift s4
$0a3b4	ab$	Reduce r3
$0a3A8	ab$	Reduce r2
$0aA2	ab$	shift s6
$0A2a6	b$	shift s7
$0A2a6b7	$	Reduce r3

$ 0 A 2 A 6 b A 9	$ Reduce r2
$ 0 A 2 A 5	$ Reduce r1
$ 0 5 1	$ Accept

LALR(1)

S → Bb/cAb/d

B → ab/c

A → e

Check whether the input string 'ceb' is accepting or not.

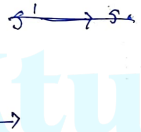
Also construct LALR(1) Parser.

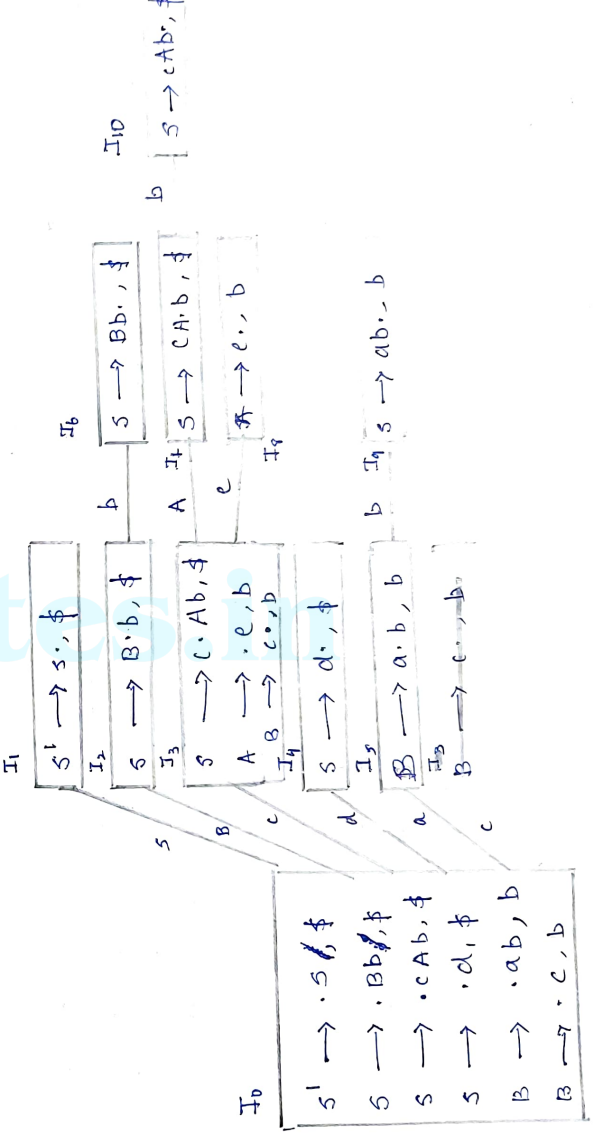
① S' → ·S ②

② S → ·Bb/·cAb/·d ③

④ B → ·ab/·c ⑤

A → ·e ⑥





	Action						GOTO		
	a	b	c	d	e	$	S	A	B
I_0	s5		s3	s4			1		2
I_1						Accept			
I_2		s6							
I_3					s8			7	
I_4						r3			
I_5		s9							
I_6						r1			
I_7		s10							
I_8		r6							
I_9		r4							
I_{10}						r2			

Stack	Input	Action
$0	ceb$	Shift s3
$0c3	eb$	Shift s8
$0c3e8	b$	Reduce r6
$0c3A7	b$	Shift s10
$0c3A7b10	$	Reduce r2
$051	$	Accept

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Input string

ab

Augmented Grammar

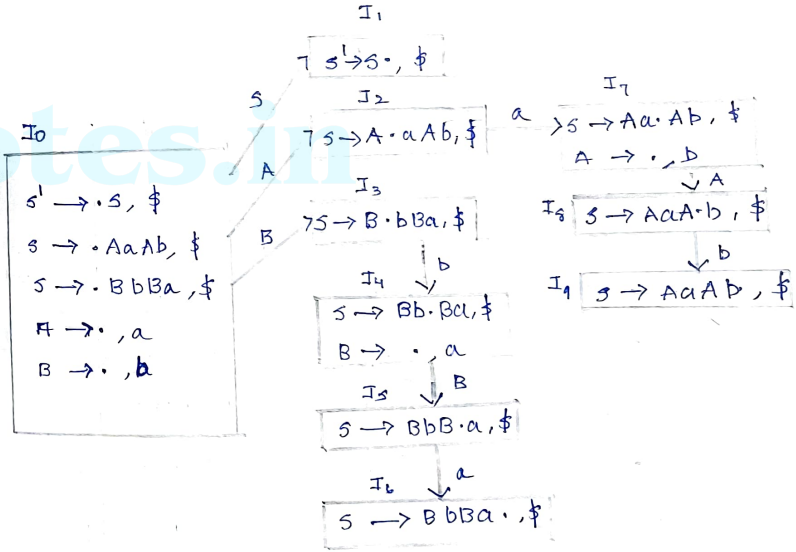
$S' \rightarrow \cdot S$ ⓪

$S \rightarrow \cdot AaAb$ ①

$S \rightarrow \cdot BbBa$ ②

$A \rightarrow \cdot$ ③

$B \rightarrow \cdot$ ④



	Action			Goto		
	a	b	$	S	A	B
I_0	r_3	r_4		1	2	3
I_1			Accept			
I_2	s_7					
I_3		s_4				5
I_4	s_4					
I_5	s_6					
I_6			r_2			
I_7		r_3			8	
I_8		s_9				
I_9			r_1			

Stack	Input	Action
$0	abb$	Reduce r_3
$0A2	abb$	Shift s_7
$0A2a7	bb$	Reduce r_3
$0A2a7A8	bb$	Shift s_9
$0A2a7A8b9	$	Reduce r_1
$0S1	$	Accept

Operator Precedence Parser

Operator Precedence Parser is used to define mathematical operators

Ambigous Grammar are allowed in this parser.

	id	+	*	$
id	-	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	-

Stack	Input	Action
$	id + id * id $	$ <· id - shift
$id	+ id * id $	id ·> + - reduce
$E	+ id * id $	$ <· + - shift
$E +	id * id $	+ <· id - shift
$E + id	* id $	id ·> * - reduce
$E + E	* id $	+ <· * - shift
$E + E *	id $	* <· id - shift
$E + E * id	$	id ·> $ - Reduce
$E + E * E	$	* ·> $ - Reduce
$E + E	$	+ ·> $ - reduce
$E	$	

1. Find First and Follow of given productions. construct LL(1) parsing table.

	FIRST	FOLLOW
$S \to A(B/GbB/Ba$	{d, b, g, h, ε}	{$}
$A \to dA/BC$	{d, g, h, ε}	{h, g, $}
$B \to g/\varepsilon$	{g, ε}	{a, $, h}
$C \to h/\varepsilon$	{h, ε}	{g, $, h}

2. Check whether the production is LL(1) or not

a) $S \to aSbS/bSaS/\varepsilon$

FIRST	FOLLOW
{a, b, ε}	{$, a, b}

	a	b	$
S	$S \to aSbS$ $S \to \varepsilon$	$S \to bSaS$ $S \to \varepsilon$	$S \to \varepsilon$

b)

	FIRST	FOLLOW
$S \to (L)^{①}/a^{②}$	{(, a}	{), ,, $}
$L \to SL'^{③}$	{(, a}	{)}
$L' \to {}^{④}\varepsilon/,{}^{⑤}SL'$	{ε, ,}	{)}

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \varepsilon$		$L' \to ,SL'$	

It is LL(1)

c)

	FIRST	FOLLOW
$S \to iEtSS'/a$	{i, a}	{e, $}
$S' \to \varepsilon/eS$	{ε, e}	{e, $}
$E \to c$	{c}	{t}

	i	a	c	e	$	t
S	$S \to iEtSS'$	$S \to a$				
S'				$S' \to \varepsilon$ $S' \to eS$		$S' \to \varepsilon$
E			$E \to c$			

It is not LL(1)