KTU 2019 Scheme

MODULE - III

BOTTOM - UP PARSING

Prepared by,
Noufala . P. S
Asst. Professor
UEC, Thrissur

SYLLABUS :-

Handle Pruning, Shift Reduce parsing, Operator precedence parsing (concept only). LR Parsing - Constructing SLR, LALR and canonical LR parsing tables.

Bottom - Up Parsing :-

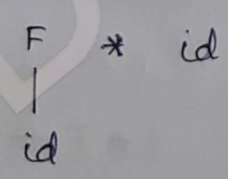
A bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

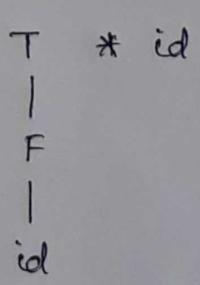
eg:- A bottom-up parse of the token stream id * id w.r.t the grammar, $E \rightarrow E + T \mid T$ is given below:

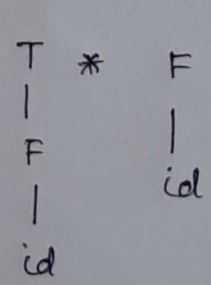
$T \rightarrow T * F \mid F$

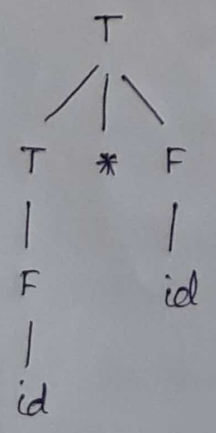
$F \rightarrow (E) \mid id$

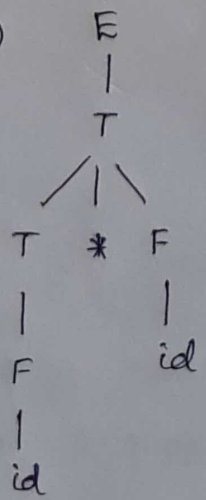
i) id * id

ii)


iii)


iv)


v)


vi)


Reductions :-

Bottom-up parsing can be considered as a process of "reducing" a string 'w' to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Our above considered example can be considered as a sequence of reductions,

id * id ; F * id ; T * id ; T * F ; T ; E

- A reduction is the reverse of a step in a derivation.
- The goal of bottom-up parsing is therefore to construct a derivation in reverse.

i.e $E \Rightarrow T$

$\Rightarrow T * F$

$\Rightarrow T * id$

$\Rightarrow F * id$

$\Rightarrow id * id$

- This derivation is infact a rightmost derivation.
- Hence, bottom-up parsing can be considered as the right-most derivation in reverse order.

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 (ie the bottom-up parsing).

For example, lets consider our previous example, id * id.

In this string, for clarity lets add the subscripts to the id.

ie, $id_1 * id_2$.

	RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
1)	$id_1 * id_2$	id_1	$F \rightarrow id$
2)	$F * id_2$	F	$T \rightarrow F$
3)	$T * id_2$	id_2	$F \rightarrow id$
4)	$T * F$	$T * F$	$T \rightarrow T * F$
5)	T	T	$E \rightarrow T$
6)	E		

- Here, In step 3, although T is the body of production $E \rightarrow T$, the symbol T is not a handle in the sentential form $T * id_2$. If T were indeed replaced by E, we could get the string $E * id_2$, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

A rightmost derivation in reverse can be obtained by "handle pruning". ie, the process of obtaining the rightmost derivation in reverse order is called "handle pruning".

- Bottom-up Parsing
  - Shift-reduce parsing
  - Operator Precedence parsing
  - LR parsing

SHIFT-REDUCE PARSING :-

- It is a form of bottom up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.
- '$' is used 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 the 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 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: ie,

STACK	INPUT
$ S	$

- Upon entering this configuration, the parser halts and announces successful completion of parsing.

eg:- let $w = id * id$ and the grammar is,

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$.

for simplicity, lets consider w as $id_1 * id_2$.

STACK	INPUT	ACTION
$	$id_1 * id_2$ $	shift
$ id_1	$* id_2$ $	reduce by $F \rightarrow id$
$ F	$* id_2$ $	reduce by $T \rightarrow F$
$ T	$* id_2$ $	shift
$ T *	id_2 $	shift
$ T * id_2	$	reduce by $F \rightarrow id$
$ T * F	$	reduce by $T \rightarrow T*F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	Accept.

- while the primary actions (or operations) are 'shift' and 'reduce', there are actually 4 possible actions a shift-reduce parser can make:

1) Shift
2) Reduce
3) Accept
4) Error

*) Shift :- Shift the next input symbol onto the top of the stack.

•) Reduce :- The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non-terminal to replace the string.

•) Accept :- Announce the successful completion of parsing.

•) Error :- Discover a syntax error and call an error recovery routine.

- The use of a stack in shift-reduce parsing is justified by an important fact : the handle will always eventually appear on top of the stack, never inside.

Conflicts during Shift-Reduce Parsing :-

- for some CFG, shift-reduce parsers cannot be used.
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make. (a reduce/reduce conflict).

Note :- An ambiguous grammar can never be LR.

Examples:

- Suppose that our grammar is,

$A \rightarrow Sa$

$B \rightarrow Sab$.

and the stack configuration is,

STACK	INPUT
$\$Sa$	$b \ldots \$$

- Now the parser is confused whether to shift b to the stack so as to obtain $B \rightarrow Sab$; or to reduce the stack using $A \rightarrow Sa$. This conflict is called shift/reduce conflict.

- Similarly, suppose the configuration is,

STACK.	INPUT
$\$Sa$	$b \ldots \$$

and our grammar is,

$A \rightarrow Sa \mid a$

$B \rightarrow Sa \mid b$

- Now the parser is confused about which reduction to choose. is $A \rightarrow Sa$ or $B \rightarrow Sa$. This conflict is known as reduce/reduce conflict.

OPERATOR PRECEDENCE GRAMMAR :-

- Any grammar G is called an operator precedence grammar if it meets the following two conditions;

① There exist no production rule which contain ε (epsilon) on its RHS. ie there should be no ε-production.

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

- A parser that reads and understands an operator precedence grammar is called operator precedence parser.

Eg:- 1) $E \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$.

2) $E \rightarrow EOE$
$E \rightarrow id$
$O \rightarrow + | * | /$

3) $E \rightarrow E+E | T$
$T \rightarrow T*F | F$
$F \rightarrow id$

- In the above 3 grammars, (1) and (2) are not operator precedence grammars, but (3) is operator precedence.
ie, In (1), $E \rightarrow AB$, 2 non-terminals came together (A & B)
In (2), also $E \rightarrow EOE$. Non-terminals came togethes.

⇒ $E \rightarrow EAE | (E) | id$
$A \rightarrow + | - | * | /$ | ^

} This grammas is not operator precedence now. But it can be converted as,

$E \rightarrow E+E | E-E | E*E | E \wedge E | (E) | id$.

- Now it became operator precedence grammar.
- Note that, operator precedence can only established between the terminals of the grammar. It ignores the non-terminals.

Precedence Relations :-

- we define three precedence relations in operator precedence parsing.

	Relation	Meaning
(1)	$a \lessdot b$	a yields precedence to b. (i.e b has higher precedence than a)
(2)	$a \doteq b$	a has same precedence as b.
(3)	$a \gtrdot b$	a takes precedence over b (i.e b has lower precedence than a).

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

$E \longrightarrow EAE \mid id$

$A \longrightarrow + \mid *$

Soln:- Convert the grammar to operator precedence form.

i.e, $E \longrightarrow E+E \mid E*E \mid id$

Now, construct the operator precedence table. For that we have to consider only the terminal symbols - { +, *, id, $ }

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	Accept

Note :- •) id - id are not compared.

•) + $\gtrdot$ +

•) * $\gtrdot$ *

•) $ $ $\Rightarrow$ accept

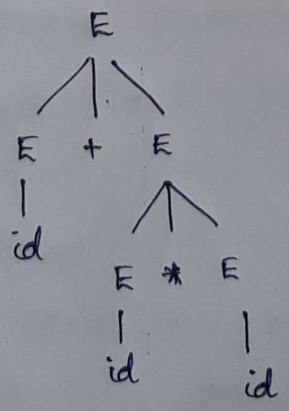
•) id $\xrightarrow[a,b]{}$ highest precedence

•) $ $\longrightarrow$ lowest precedence

- Now we have to parse the given string, id + id * id using the above constructed precedence table.

STACK	RELATION	INPUT	ACTION
$	<·	id + id * id $	Shift id
$id	·>	+ id * id $	Reduce E → id
$E	<·	+ id * id $	Shift +
$E +	<·	id * id $	Shift id
$E + id	·>	* id $	Reduce E → id
$E + E	<·	* id $	Shift *
$E + E *	<·	id $	Shift id
$E + E * id	·>	$	Reduce E → id
$E + E * E	·>	$	Reduce E → E * E
$E + E	·>	$	Reduce E → E + E
$E	Accept.	$	

- Now generate the parse tree using the bottom-up approach from the contents of the stack.



LR Parsing :-

•) LR(K) ⇒ Left-to-Right Scanning (L)
Right Most Derivation in reverse (R)
k no: of lookahead i/p symbols (K)

•) LR parsing
- → Simple LR (SLR)
- → Canonical LR (CLR)
- → LookAhead LR (LALR)

•) why LR parsers ? (advantages of LR)

LR parsers are attractive for a variety of reasons :

i) LR parsers can be constructed to recognize all pgming language constructs for which CFG can be written.

ii) LR parsing method is the most general non-backtracking shift-reduce parsing method.

iii) LR parser can detect syntax errors as soon as it is possible to do so on a left-to-right scan of the input.

iv) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods.

Drawback of LR method :

Too much work to construct an LR parser by hand. A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available. The most commonly used one is 'Yacc'.

Items and LR(0) Automaton:

An LR parser makes shift-reduce decisions by maintaining "states" to keep track of where we are in a parse. States represent set of "items." An LR(0) item (or simply item) of a grammar G_1 is a production of G_1 with a dot at some position of the body. Thus, production $A \rightarrow XYZ$ yields the below four items:

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

- The production $A \rightarrow \varepsilon$ generates only one item, $A \rightarrow .$

An item indicates how much of a production we have seen at a given point in the parsing process. For eg, the item $A \rightarrow .XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item $A \rightarrow X.YZ$ indicates that we have just seen on the input a string derivable from X, and that we hope next to see a string derivable from YZ.

One collection of sets of LR(0) items, called the canonical LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an LR(0) automaton.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions — CLOSURE and GOTO.

If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production S' → S. The purpose of this new starting ~~symbol~~ production is to indicate the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by S' → S.

CLOSURE of item sets :-

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1) Initially, add every item in I to CLOSURE(I).
2) If $A \rightarrow \alpha . B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Algorithm for computation of CLOSURE() :-

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
            for (each item A → α.Bβ in J)
                for (each production B → γ of G)
                    if (B → .γ is not in J)
                        add B → .γ to J;
        until no more items are added to J on one round;
        return J;
    }
```

The sets of items can be divided into two classes :-

1) Kernal Items :- The initial item, $S' \rightarrow .S$, and all items whose dots are not at the left end.

2) Nonkernal Items :- All items with their dots at the left end, except for $S' \rightarrow .S$.

The function GOTO() :-

Denoted by GOTO(I, X), where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $A \rightarrow \alpha X . \beta$ such that $A \rightarrow \alpha . X\beta$ is in I. It is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton corresponds to set of items, and GOTO(I, X) specifies the transition from the state for I under input X.

Algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' is,

```
void items(G') {
    C = { CLOSURE({ S' → .S }) };
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (GOTO(I, X) is not empty and not in C)
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Q) Draw the canonical LR(0) collection [or the LR(0) automaton] for the grammar,

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Numbering the prod"s :-

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

Sol" :- Step 1: Define the augmented grammar.

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

If I is the set of one item $\{E' \rightarrow .E\}$, then CLOSURE(I) contains the set of items I_0 (figure below) (ie, $E' \rightarrow .E$ is put in CLOSURE(I). Since there is an immediate non-terminal, E, to the right of a dot, we add the E-productions with dots at the left ends; $E \rightarrow .E+T$ and $E \rightarrow .T$. Now, there is an immediate non-terminal T, to the right of a dot. So we add $T \rightarrow .T*F$ and $T \rightarrow .F$. Next, there is an F to the right of a dot. Hence we have to add $F \rightarrow .(E)$ and $F \rightarrow .id$. Now, no other items need to be added.

Thus, I_0 becomes,

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Now compute GOTO(I, X) and create the remaining set of items.

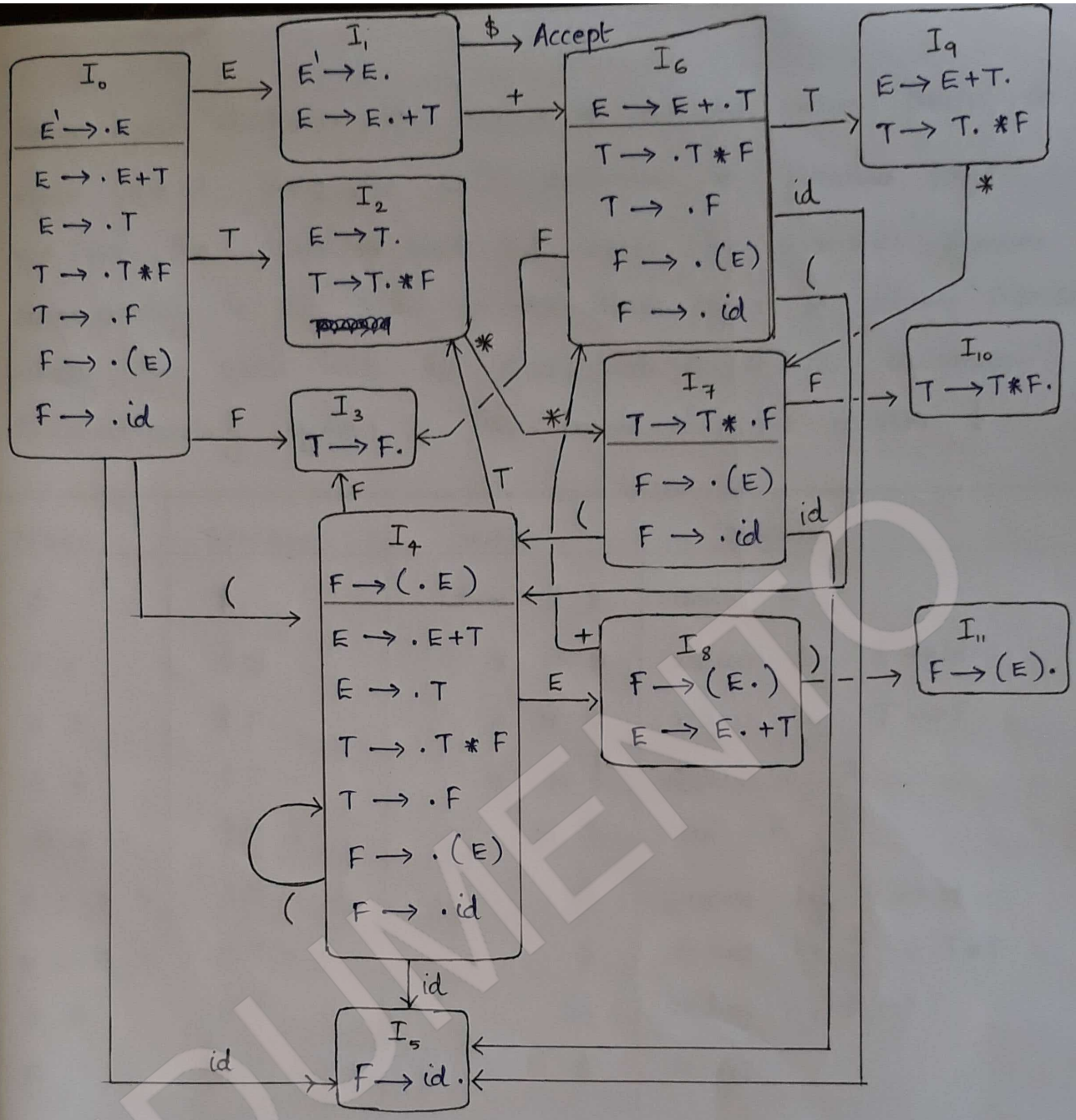


Fig:- LR(0) automaton or
canonical LR(0) collection.

•) How can LR(0) automata help with shift-reduce decisions? (ie, Use of the LR(0) automaton) :-

Shift-reduce decisions are made as follows. Suppose that the string γ of grammar symbols takes the LR(0) automaton from the state 0 to some state j. Then, shift on next input symbol 'a' if state j has a transition on 'a'. Otherwise we choose to reduce. The items in state j will tell us which production to use.

Below figure illustrates the actions of a shift-reduce parser on input id * id, using the LR(0) automaton in previous page. we use the stack to hold the states. The grammar symbols corresponding to the states on the stack appear in column SYMBOL Initially, the stack holds the start state 0 of the automaton. The corresponding symbol is the bottom-of-stack marker $.

STACK	SYMBOLS	INPUT	ACTION
0	$	id * id $	Shift to 5
0 5	$ id	* id $	reduce by F → id
0 3	$ F	* id $	reduce by T → F
0 2	$ T	* id $	Shift to 7
0 2 7	$ T *	id $	Shift to 5
0 2 7 5	$ T * id	$	reduce by F → id
0 2 7 10	$ T * F	$	reduce by T → T*F
0 2	$ T	$	reduce by E → T
0 1	$ E	$	Accept

At line no: 1, the next input symbol is id and state 0 has a transition on id to state 5. Therefore we shift. At line no: 2, state 5 (symbol id) has been pushed on to the stack. There is no transition from state 5 on input *, so we reduce using the production F → id.

~~with symbols, a reduction~~ with symbols, a reduction is implemented by popping the body of the production from the stack (on line 2, the body is id) and pushing the head of the production (in this case, F).

with states, we pop state 5 for symbol id, which brings state 0 to the top and look for a transition on F, the head of the production. In our figure, state 0 has a transition on F to state 3, so we push state 3, with corresponding symbol F. (see line no: 3)

The LR-parsing Algorithm :-

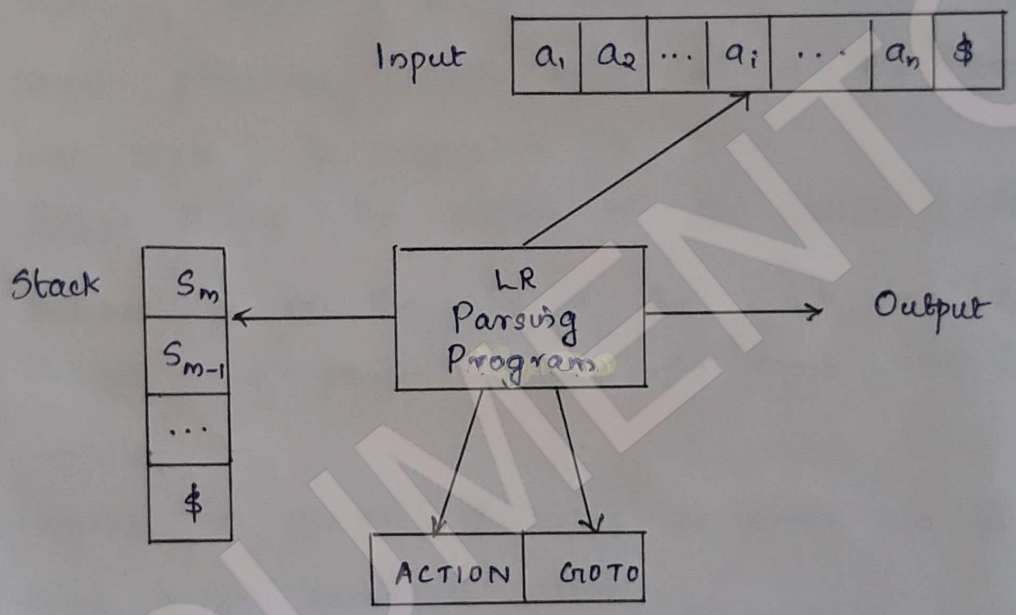


fig: Model of an LR parser

A schematic of an LR parser is shown above. It consists of an input, an output, a stack, a driver pgm and a parsing table that has two parts (ACTION & GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing pgm reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a state. The stack holds a sequence of states, $S_0 S_1 \dots S_m$, where S_m is on top.

Structure of the LR parsing table :-

The parsing table consists of two parts : a parsing-action function ACTION and a goto function GOTO.

1) The ACTION function takes as arguments a state i and a terminal 'a' (or $, the input endmarker). The value of ACTION [i, a] can have one of four forms :
   a) Shift j, where j is a state. The action taken by the parser effectively shifts input 'a' to the stack, but uses state j to represent 'a'.
   b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   c) Accept. The parser accepts the input and finishes parsing.
   d) Error. The parser discovers an error in its input and takes some corrective action.

2) The GOTO function, defined on sets of items, to states ; if GOTO $[I_i, A] = I_j$, then GOTO also maps a state i and a non-terminal A to state j.

LR parsing Algorithm :

INPUT : An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT : If w is in L(G), the reduction steps of 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 below pgm.

```
let 'a' be the first symbol of w$;
while(1) {  /* repeat forever */
    let s be the state on top of the stack;
    if ( ACTION [s, a] = shift t ) {
        push t onto the stack;
        let 'a' be the next input symbol;
    }else if (ACTION[s, a] = reduce A→β) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO [t, A] onto the stack;
        output the production A→β;
    }else if ( ACTION[s, a] = accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

fig: LR parsing program.

Constructing SLR-parsing tables :

The SLR method begins with LR(0) items and LR(0) automata. The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It also requires to find out the FOLLOW(A) for each non-terminal A of a grammar.

Algorithm: Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

Rule ~~Step~~ 1) Construct $C = \{ I_0, I_1, \ldots, I_n \}$, the collection of sets of LR(0) items for G'.

Rule ~~Step~~ 2) State i is constructed from I_i. The parsing actions for state i are determined as follows:

a) If $A \rightarrow \alpha . a\beta$ 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 $A \rightarrow \alpha .$ is in I_i, then set $ACTION[i, a]$ to "reduce $A \rightarrow \alpha$" for all 'a' in FOLLOW(A); here A may not be S'.

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 SLR(1). The algorithm fails to produce a parser in this case.

Rule ~~Step~~ 3) The goto transitions for state i are constructed for all non-terminals A using the rule: If ~~GOTO(~~ $GOTO(I_i, A) = I_j$, then $GOTO[i, A] = j$.

Rule 4 : All entries not defined by rules (2) and (3) are made "error".

Rule 5 : The initial state of the parser is the one constructed from the set of items containing $S' \rightarrow .S$

The parsing table consisting of the ACTION and GOTO functions determined by the above algorithm is called the SLR(1) table for G. An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be SLR(1) (or simply SLR).

eg: Construct the SLR table for our previous example.

[Note: reductions are put in their corresponding Follow]

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

/(8): Construct the SLR parsing table for the given grammar and check whether the given grammar is SLR or not.

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

Sol^n: Augmented grammar is,

$S' \rightarrow S$

$S \rightarrow L = R$

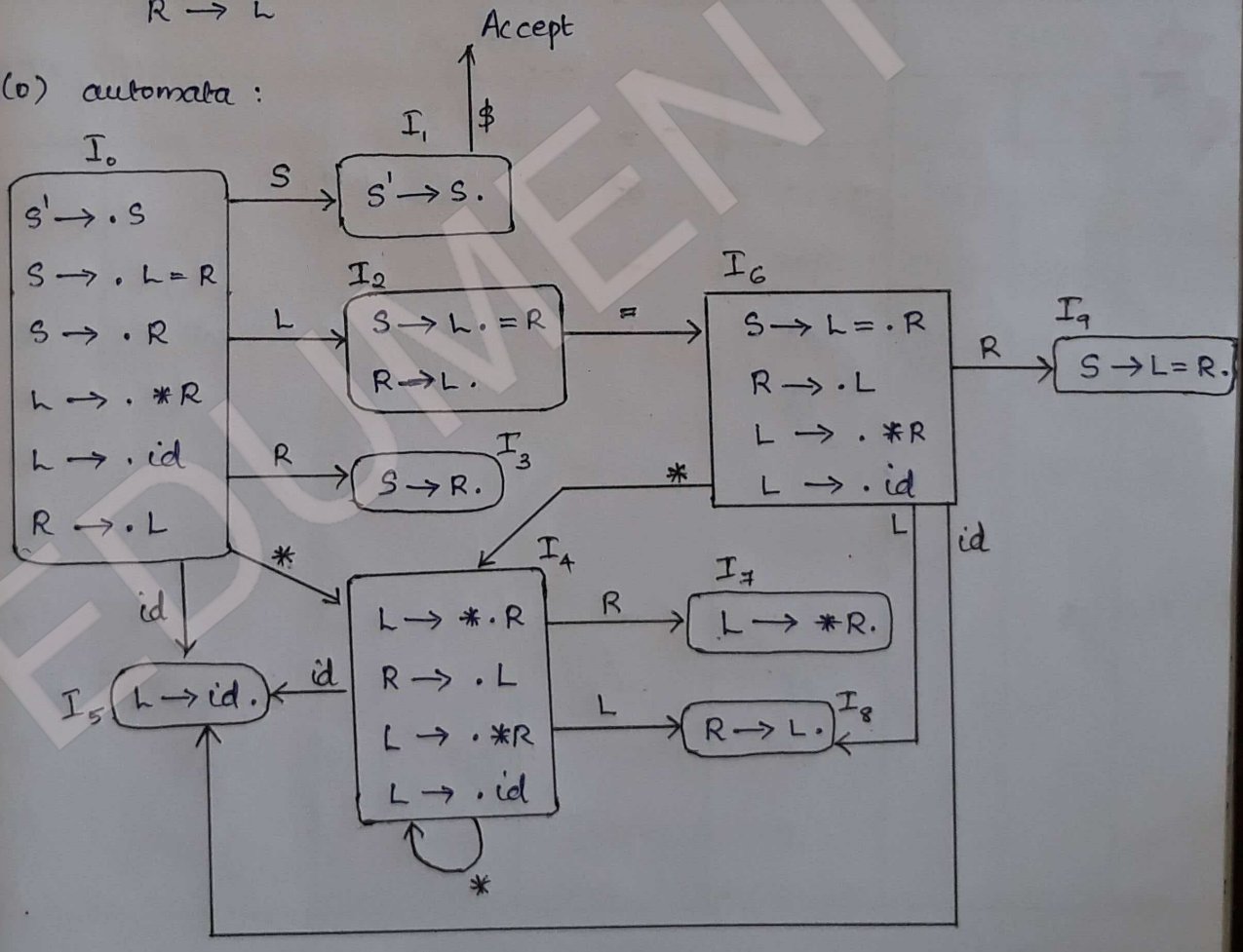
$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

LR(0) automata :



Now, let's number the productions :

1) $S \rightarrow L = R$ 2) $S \rightarrow R$

3) $L \rightarrow *R$ 4) $L \rightarrow id$

5) $R \rightarrow L$

Now, construct the SLR parsing table. For that first find out the FOLLOW of all non-terminals.

FOLLOW(S) = { $ }

FOLLOW(L) = { =, FL(R) }
= { =, $ }

FOLLOW(R) = { FL(S), FL(L) }
= { $, = }

Now, the parsing table is,

STATES	ACTION				GOTO		
	=	*	id	$	S	L	R
0		S_4	S_5		1	2	3
1				Accept			
2	S_6 / r_5			r_5			
3				r_2			
4		S_4	S_5			8	7
5	r_4			r_4			
6		S_4	S_5			8	9
7	r_3			r_3			
8	r_5			r_5			
9				r_1			

Here, in ACTION[2, =], there is a shift to state 6 as well as reduction of R → L. Hence, it is a shift-reduce conflict. Thus, the given grammar is not SLR.

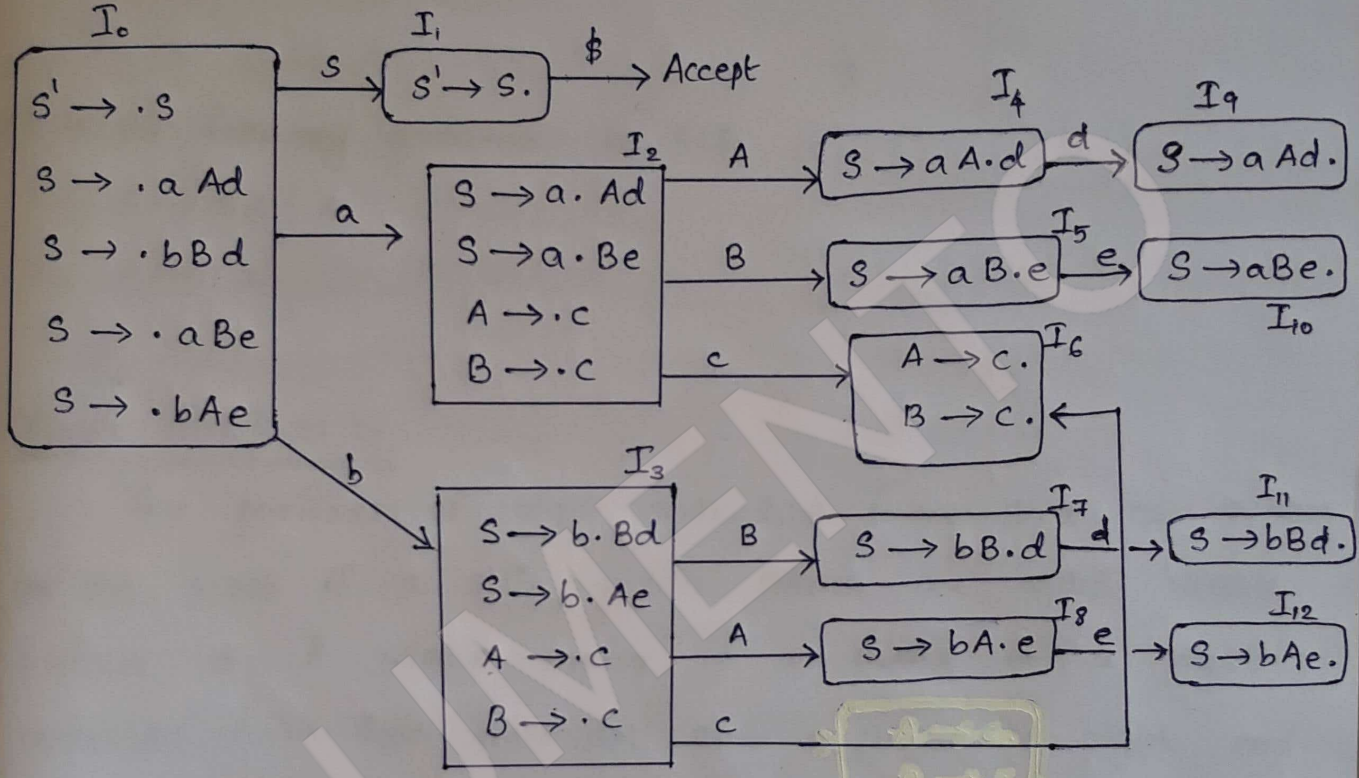
8) Construct the LR(0) collection of items for the given grammar.

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$ } (small letter 'c')

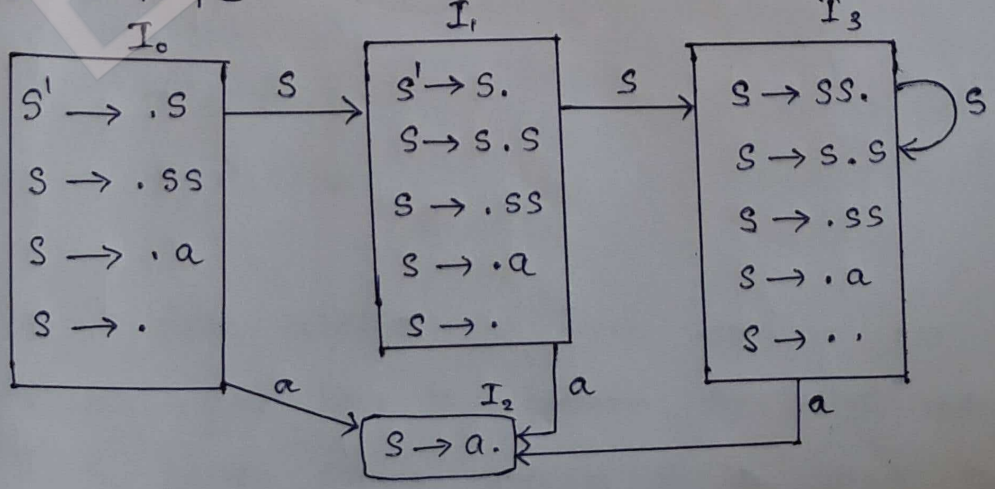
Soln:



9) Find the LR(0) items for the grammar

$S \rightarrow SS \mid a \mid \varepsilon$

Soln:



Practise questions :

1) Show that the following grammar is not SLR(1)

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

2) S.T → the following grammar is SLR

$S \rightarrow SA \mid A$

$A \rightarrow a$

Viable Prefixes :-

The prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called viable prefixes. ie, A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

eg: for the below grammar check whether id+id and E+id are viable prefix or not.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Soln :- To check whether the given prefixes are viable prefix or not, you have to perform the shift-reduce parsing. If the given prefix appears in the stack, then it is a viable prefix. Otherwise not.

Stack	Input	Action
$	id + id $	Shift id
$ id	+ id $	Reduce by F → id
$ F	+ id $	Reduce by T → F
$ T	+ id $	Reduce by E → T
$ E	+ id $	Shift +
$ E +	id $	Shift id
$ E + id		

Hence E+id is a viable prefix. But id+id is not a viable prefix.

2) For the given grammar, check whether ab, bb, Cb, abb are viable prefixes or not.

S → CC

C → aC | b.

Soln:

1)

Stack	Input	Action
$	a b $	shift a
$a	b$	Shift b
$ab	$	

⇒ ab is a viable prefix

2)

$	bb $	Shift b
$b	b$	Reduce by C → b
$C	b$	Shift b
$Cb	$	

⇒ Cb is a viable prefix

But bb is not a viable prefix.

Stack	Input	Action
$	abb $	Shift a
$a	bb $	Shift b
$ab	b $	Reduce by C → b
$aC	b $	Reduce by C → aC
$C	b $	Shift b
$Cb	$	Reduce by C → b
$CC	$	Reduce by S → CC
$S		

Hence abb is not a viable prefix.

Q) For the given grammar S → 0S1 | 01 write all the viable prefixes for the string 00001111.

Soln:-

Stack	Input	Action
$	00001111 $	Shift 0
$0	0001111 $	Shift 0
$00	001111 $	Shift 0
$000	01111 $	Shift 0
$0000	1111 $	Shift 1
$00001	111 $	Reduce by S → 01
$000S	111 $	Shift 1
$000S1	11 $	Reduce by S → 0S1
$00S	11 $	Shift 1
$00S1	1 $	Reduce by S → 0S1
$0S	1 $	Shift 1
$0S1	$	Reduce by S → 0S1
$S	$	

viable prefixes:- 0, 00, 000, 0000, 00001, 000S, 000S1, 00S, 00S1, 0S, 0S1, S.

Canonical LR(1) Items (CLR) :-

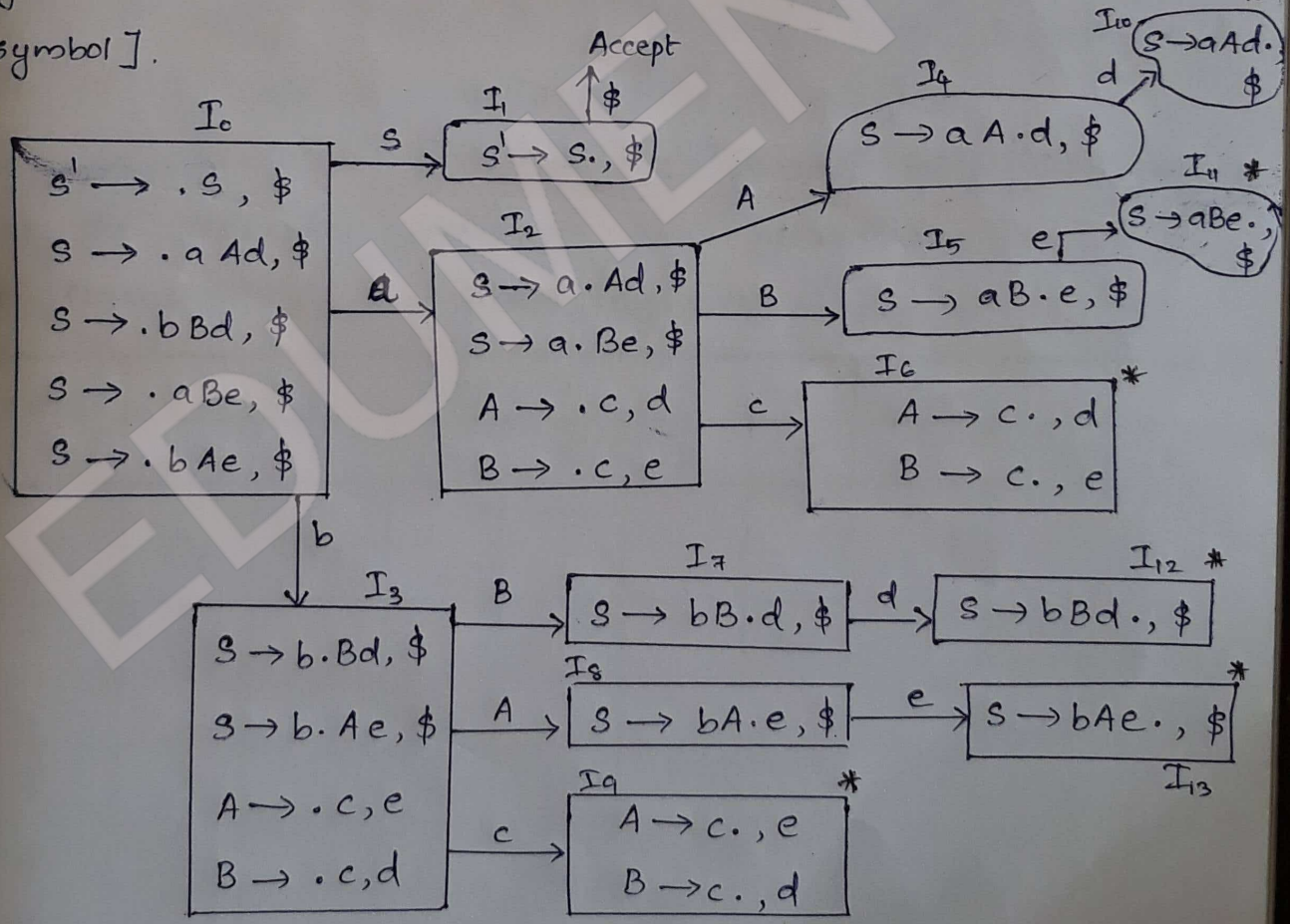
Q) Construct the set of LR(1) items and the CLR parsing table for the given grammar.

$S \rightarrow aAd$ ① | bBd ② | aBe ③ | bAe ④

⑤ $A \rightarrow c$

⑥ $B \rightarrow c$

Soln: [Note: In CLR, the construction of set of LR(1) items is similar to the construction of LR(0) items. But the difference is, in ~~CLR~~ LR(1), we have to consider a look-ahead symbol too. ie the 1 in LR(1) indicates '1' look-ahead symbol].



Note:- for the augmented production (ie $S' \rightarrow S$) in I_0, the lookahead will be always $ by default.

(For the remaining items we have to find out the look-ahead symbols. For that we need to know the FIRST().
eg, consider the set of canonical item I_2. In that, we have written $S \to a \cdot Ad, \$$ and $S \to a \cdot Be, \$$ directly from I_0 and hence their look-ahead remains the same, ie $ itself. But $A \to \cdot c$ and $B \to \cdot c$ are added newly to I_2, and Hence we have to find out their look-ahead.
for $A \to \cdot c$;- we expanded this production because of $S \to a \cdot Ad, \$$. Hence we have to find the first of whatever comes after A in $a \cdot Ad, \$$. Here it is 'd'. ie why $A \to \cdot c, d$. Similarly from $S \to a \cdot Be, \$$ it is e after B. Hence $B \to \cdot c, e$.)

Now, we have to construct the parsing table [CLR parsing table]
Here the reductions ~~with~~ will be entered in the corresponding look-ahead symbols only.

STATE	ACTION						GOTO		
	a	b	c	d	e	$	S	A	B
0	S_2	S_3					1		
1						Acc			
2			S_6					4	5
3			S_9					8	7
4				S_{10}					
5					S_{11}				
6				r_5	r_6				
7				S_{12}					
8					S_{13}				
9				r_6	r_5				

STATE	ACTION						GOTO		
	a	b	c	d	e	$	S	A	B
10						r_1			
11						r_3			
12						r_2			
13						r_4			

- Here there is no S/R or R/R conflicts. Hence the grammar is CLR.

LALR parsing :- (Look-Ahead LR)

- For constructing LALR parsing table, we have to first create the LR(1) collection of items (CLR items) and then look for the ~~it~~ boxes with same items (look-ahead can vary). Now we have to merge those states. This will reduce the no: of states. Note that the no: of states in SLR and LALR will be same. But CLR may have more no: of states.
- power :-

  SLR < LALR < CLR.

- CLR is most powerful LR parser. But it requires more m/m because of large no: of states and large parsing table.
- Here, the most commonly used LR parser is LALR.
- Now lets draw the LALR parsing table for our previous example.

ii, construct the LALR parsing table for,

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

Soln:- we have already constructed the LR(1) items. (ii for CLR). In that we can see that, the states I_6 and I_9 are having same items $A \rightarrow c.$ and $B \rightarrow c.$. Only their corresponding look-aheads vary. Hence we can ~~can~~ merge those two states into I_{69}.

I_{69}

$A \rightarrow c.$, d/e
$B \rightarrow c.$, d/e

Now in the LALR parsing table we can merge the states 6 and 9 as 69. The parsing table will become,

STATES	ACTION						GOTO		
	a	b	c	d	e	$	S	A	B
0	S_2	S_3					1		
1						acc			
2			S_{69}					4	5
3			S_{69}					8	7
4				S_{10}					
5					S_{11}				
69				r_5 / r_6	r_6 / r_5				
7				S_{12}					
8					S_{13}				
10					r_1				
11					r_3				
12					r_2				
13					r_4				

Here it is clear that ACTION [69, d] and ACTION [69, e] contains reduce-reduce conflicts. ie, $r_5 | r_6$. Hence the grammar is not LALR.

* Q) Construct CLR and LALR parsing table for the below grammar,

$S \rightarrow CC$

$C \rightarrow cC \mid d$.

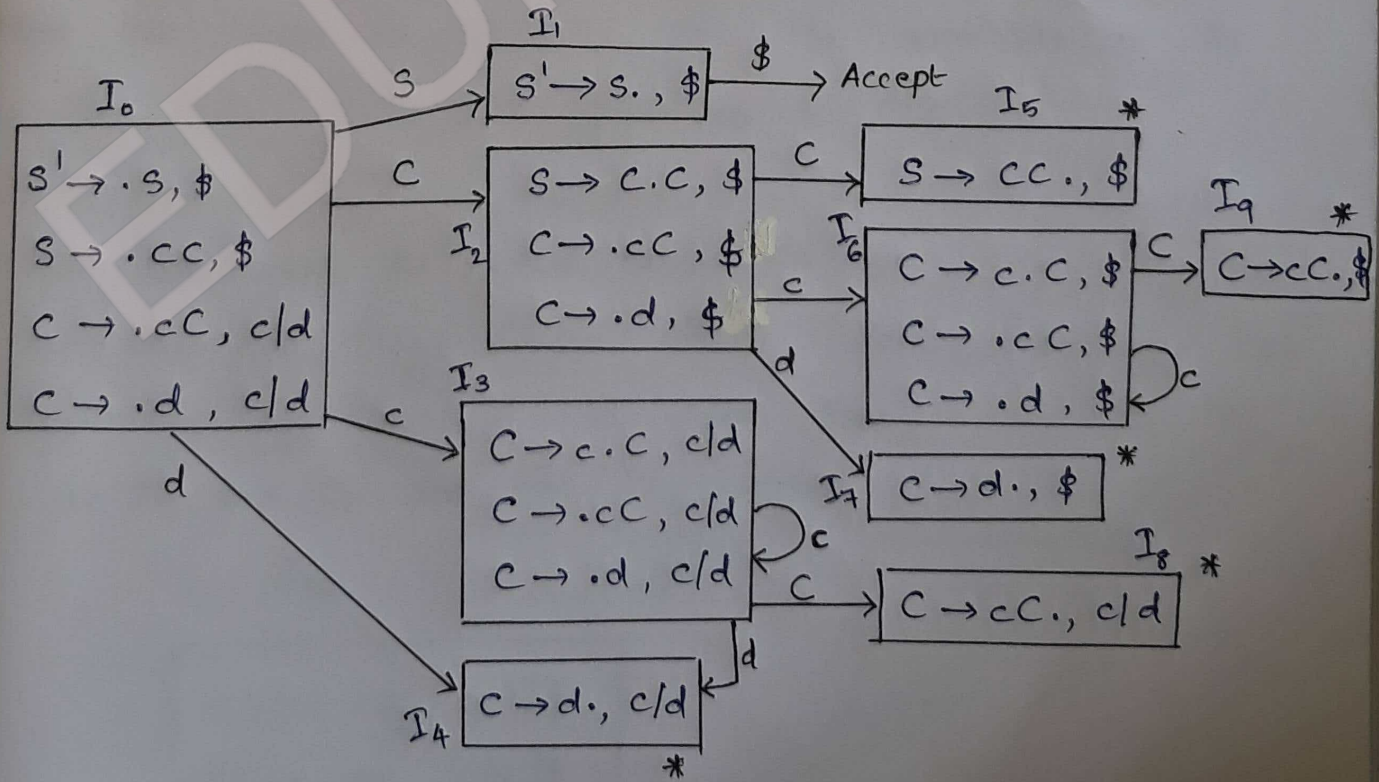
Sol? The augmented grammar is,

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

CLR collection of items (ie LR(1) items)



CLR parsing table :-

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

There is no s/r or r/r conflicts in the above table. Hence the grammar is CLR.

- Now lets create the LALR table for the same. For that we have to merge similar states. Here, from the CLR collection of items, it is clear that;
  - I_3 and I_6 can be merged.

I_{36}

$C \rightarrow c.C, c/d/\$$
$C \rightarrow .cC, c/d/\$$
$C \rightarrow .d, c/d/\$$

•) I_4 and I_7 can be merged.

I_{47} : $C \rightarrow d.,\ c/d/\$$

•) I_8 and I_9 can be merged.

I_{89} : $C \rightarrow cC.,\ c/d/\$$

Thus, the LALR parsing table will be,

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Here also there is no conflict. Hence the grammar is LALR. Thus we can conclude that the given grammar is CLR as well as LALR.