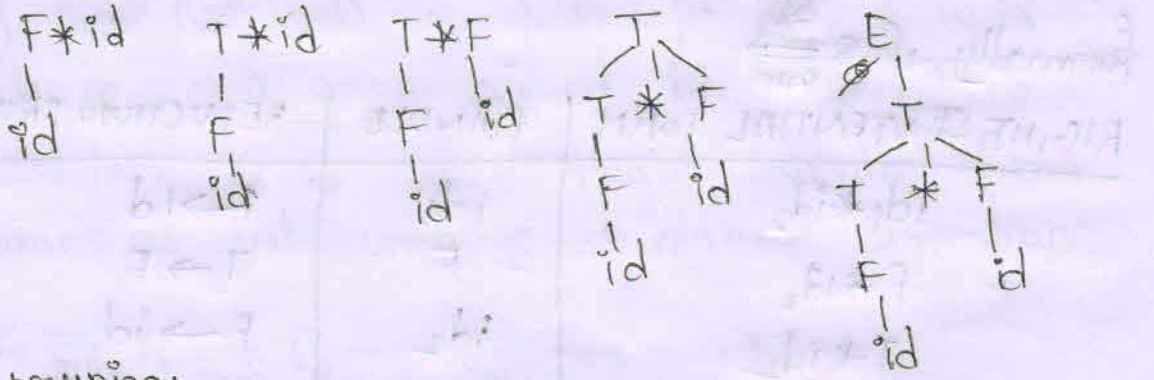
UNIT-3

Bottom up parsers:

Introduction:

→ A bottom-up parse 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 parses for 'id * id



Handle pruning:

Bottom up parsing during a left to right scan of the input constructs a right most derivation in reverse. Informally a 'handle' is a substring that matches the body of the production, and whose reduction represents one step along the reverse of the right-most derivation

Example:

adding subscripts to the tokens id for clarity, the handles during the parse of $id_1 * id_2$ a/c to the expressions grammar → $\left\{ \begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{array} \right\}$ are shown in the figure.

Although T is the body of the production $E \to T$, the symbol T is not a handle in the sentential form $T * id_2$. If T were indeed replaced by E, we would get a 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

~~formally, if $S \overset{*}{\underset{rm}{\Longrightarrow}}$~~

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
F	$T * F$	$E \to T * F$

Handles during a parse of $id_1 * id_2$ (a)

Formally, if $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta w$, as in figure (b), then production $A \to \beta$, in the position following α is a handle of $\alpha \beta w$. Alternatively, a handle of a right sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right sentential form in a rightmost derivation of γ

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we prefer to the body β rather than $A \to \beta$ as a handle. Note we "a handle" rather than "the handle", because

the grammar could be ambigous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambigous, then every right-sentential form of the grammar has Exactly one handle.

A right-most derivation in reverse can be obtained by "handle pruning". That is, we start with a string of terminals w to be parsed. If w is a sentence of a grammar at hand, then let $w = \gamma_n$, where γ_n is the n^{th} right sentential form of some as yet unknown rightmost derivation.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

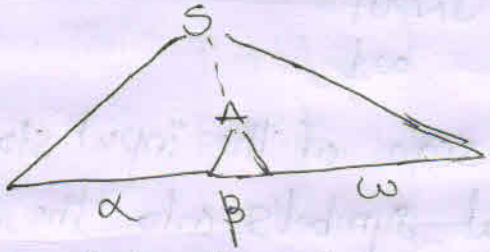


figure (b). A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of relevant production $A_n \to \beta_n$ to obtain the previous right sentential form γ_{n-1}. Note that we do not know how handles are to be found, but we shall see methods of doing so shortly

We the repeat this process. That is we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right sentential form γ_{n-2}. If by continuing this process we produce a right sentential form consisting only of the start symbol S, the we halt & that is successfull.

Shift-Reduce parsing:

Shift-reduce parsing 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.

→ we use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather on the left as we did for top-down parsing.

Initially, the stack is empty, and string w is on the input, as follows:

STACK	INPUT
$	w$

During left to right scan of the input string, the parser shifts zero/more input symbols onto the stack until it is ready to reduce a string β of the 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 input is empty.

Actions in shift-reduce parsing:

1. Shift: shift the next input symbol onto the top of stack
2. 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

3. Accept: Announce successfull Completion of parsing

4. Error: Discover a syntax error and can an error recovery routine

Find the handles for the given RSF and Construct shift-reduce parser:

1. $E \to E+T \mid T$
   $T \to T*F \mid F$
   $F \to (E) \mid id$

i/p: id+id
id+id*id

→ RMD

$E \to E+T$
$\Rightarrow E+F$
$\Rightarrow E+id$
$\Rightarrow T+id$
$\Rightarrow F+id$
$\Rightarrow id+id$

RSF	Handle	Action
id_1+id_2	id_1	$F \to id$
$F+id_2$	F	$T \to F$
$T+id_2$	T	$E \to T$
$E+id_2$	id_2	$F \to id$
$E+F$	F	$T-F$
$E+T$	$E+T$	$E \to E+T$
E		

Stack	RSF	Action
$	$id_1+id_2\$$	shift id_1
$\$id_1$	$\$id_2\$$	reduce $F \to id$
$\$F$	$+id_2\$$	reduce $T \to F$
$\$T$	$+id_2\$$	reduce $E \to T$
$\$E$	$+id_2\$$	shift +
$\$E+$	$+id_2\$$	shift id_2
$\$E+id_2$	$\$$	reduce $F \to id_2$
$\$E+F$	$\$$	reduce $T \to F$
$\$E+T$	$\$$	reduce $E \to T$
$\$E$		Success

ip: id+id*id

RMD:

$E \to E+T$
$\Rightarrow E+T*F$
$\Rightarrow E+T*id$
$\Rightarrow E+F*id$
$\Rightarrow E+id*id$
$\Rightarrow T+id*id$
$\Rightarrow F+id*id$
$\Rightarrow id+id*id$

RSF	Handle	Action
$id_1+id_2*id_3$	id_1	$F \to id$
$F+id_2*id_3$	F	$T \to F$
$T+id_2*id_3$	T	$E \to T$
$E+id_2*id_3$	id_2	$F \to id_2$
$E+F*id_3$	F	$T \to F$
$E+T*id_3$	id_3	$F \to id_3$
$E+T*F$	$T+F$	$T \to F*F$
$E+T$	$E+T$	$E \to E+T$
E		

Stack	RSF	Action
$	$id_1+id_2*id_3\$$	shift id_1
$\$id_1$	$+id_2*id_3\$$	$F \to id_1$
$\$F$	$+id_2*id_3\$$	$T \to F$
$\$T$	$+id_2*id_3\$$	$E \to T$
$\$E$	$+id_2*id_3\$$	Shift +
$\$E+$	$id_2*id_3\$$	Shift id_2
$\$E+id_2$	$*id_3\$$	reduce $F \to id$
		$T \to F$
		shift *
		shift id_3
$\$E+T*id_3$	$\$$	reduce $F \to id_3$
$\$E+T*F$	$\$$	reduce $T \to T*F$
		reduce $F \to E+T$
$\$E$	$\$$	Success

2) $S \to 0S1 \mid 01$ i/p: 000111

$S \xrightarrow{rm} 0S1$

$\Rightarrow 00S11$

$\Rightarrow 000111$

RSF	Handle	Action
000111	01	$S \to 0S1$
00S11	0S1	$S \to 0S1$
0S1	0S1	$S \to 0S1$

Stack	RSF	Action
$	000111$	Shift 0
$0	00111$	shift 0
$00	0111 $	shift 0
$000	111 $	shift 1
$0001	11 $	reduce $S \to 01$
$00S	11$	shift 1
$00S1	1$	reduce $S \to 0S1$
$0S	1$	shift 1
$0S1	$	reduce $S \to 0S1$
$S	$	success

3) $S \to SS+ \mid SS* \mid a$ i/p: aaa*a++

$S \to SS+$

$\Rightarrow SSS++$

$\Rightarrow SSa++$

$\Rightarrow SSS*a++$

$\Rightarrow SSa*a++$

$\Rightarrow Saa*a++$

$\Rightarrow aaa*a++$

RSF	Handle	Action
aaa*a++	a	$S \to a$
Saa*a++	a	$S \to a$
SSa*a++	a	$S \to a$
SSS*a++	SS*	$S \to SS*$
SSa++	a	$S \to a$
SSS++	SS+	$S \to SS+$
SS+*	SS+	$S \to SS+$
S		

Stack	RSF	Action
$	aaa*a++$	shift a
$a	aa*a++$	reduce S→a
$S	aa*a++$	Shift a
$Sa	a*a++$	reduce S→a
$SS	a*a++$	Shift a
$SSa	*a++$	reduce S→a
$SSS	*a++$	Shift *
$SS*	a++$	reduce S→SS*
$SS	a++$	Shift a
$SSa	a++$	reduce S→a
$SSa	a++$	Shift +
$SS	+$	reduce S→SS+
$SS+	+$	Shift +
$S	$	reduce S→SS+
	$	Success

Types of conflicts in shift-reduce parsers

Conflicts during shifts-Reduce parsing:

There are CFG's for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents & the next input symbol

Types:

i) shift/reduce conflict:

→ Cannot decide whether to shift or to reduce
Called shift-reduce conflict

ii) redu/reduce conflict:

→ Cannot decide whether which of several reductions to make Called reduce-reduce conflicts

eg: Consider the grammar

1) E → E+E
   | E-E
   | NUM
   | ID

i/p: 2+3*4

No.	Stack	operation/grammar
1	2 NUM	shift 2
2	E	reduce E→NUM
3	E+	shift +
4	E+3	shift 3
5	E+E	reduce E→NUM
6	E	reduce E→E+E or shift *

ie Shift reduce conflict

eg 2) An ambiguous grammar can never be LR. for eg:
Consider the dangling-else grammar

stmt → if expr then stmt
| if expr then stmt else stmt
| other

If we have a shift-reduce parser in configuration

STACK	INPUT
...if expr then stmt	else ... $

→ we cannot tell whether if expr then stmt is the handle no matter what happens below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the else on the input, it might be correct to reduce if expr then stmt to stmt, or it might be correct to shift else then to look for another stmt to complete the alternative if expr then stmt else stmt

eg 3) (1) stmt → id (parameter_list)
(2) stmt → expr := expr
(3) parameter_list → parameter_list , parameter
(4) parameter_list → parameter
(5) parameter → id
(6) expr → id (expr_list)
(7) expr → id
(8) expr_list → expr_list , expr
(9) expr_list → expr

STACK	INPUT
... id (id	, id) ...

In the problem id on the top of the stack must be reduced, but by which production? The correct choice is production (5) if p is a procedure, but production (7) if p is an array. The stack does not tell which; information in the symbol table obtained from the declaration of p must be used

So, In this case we have the conflict that reduce id by parameter or expr. It is called as reduce-reduce conflict.

SLR parses: | SLR(1) | LR(0)

└ Lookahead symbol

└→ rmd in reverse

└→ Scan from left to right

└→ Simple

1. No left recursion and left factoring
2. Augment the grammar
3. Number the productions
4. Find the closure of item set (LR(0) item) & goto function
5. find first and follow set
6. Construct the SLR parsing table

   ├→ shift reduce

   ├→ reduce reduce
7. parse the input string

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 $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more ~~New~~ new Items can be added to CLOSURE(I).

Algorithm for Computation of CLOSURE

~~SetOfItems CLOSURE (I) {~~

A convenient way to implement the function closure is to keep a boolean array added, indexed by the nonterminals of G_1, such that added[B] is set to true if and when we add the item $B \to .\gamma$ for each B-production $B \to \gamma$.

```
SetOfItems CLOSURE(I){
    J=I;
    repeat
        for(each item A → α.Bβ in J)
            for(each production B → γ of G1)
                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;
}
```

Note that if one B-production is added to the closure of I with the dot at the left end, then all B-productions will be similarly added to the closure. Hence, it is not necessary in some circumstances actually to list the items $B \to .\gamma$ added to I by CLOSURE. A list of nonterminals B whose productions were so added will suffice.

We deride all the sets of items of interest into two classes.

1. Kernel items:
   The initial item, $S' \rightarrow .S$, all items whose dots are not at the left end.
2. Non-kernel items:
   all items with their dots at the left end, except for $S' \rightarrow .S$.

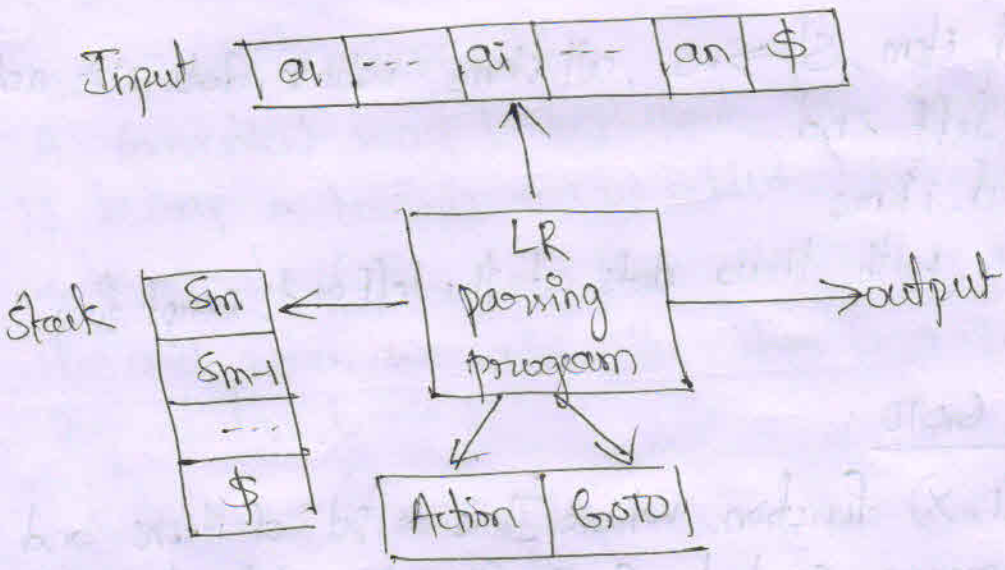
The Function GOTO:

The GOTO (I, X) function, 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. Intuitively, the GOTO function is used to define the transactions in the LR(0) automaton for a grammar. The states of the automaton correspond to set of items, and GOTO(I, X) specifies the transition from the state for I under input X

Algorithm for Computation of LR(0) items:

```
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 set of items are added to C
        on a round;
}
```

The LR- Parsing Algorithm



Model of LR parser:

A schematic of an LR parser is shown in figure It Consists of an input, an output, a stack, a driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time, where a shift-reduce parser would shift a symbol, an LR parser shifts a ~~table~~ state. Each state summarizes the information contained in the stack below it

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:

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

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

If any conflicting actions results from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce in this case.

3. The goto transitions for state i are constructed for all nonterminals 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. Initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

<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> Reduces $A \to \beta$. 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. ~~we shall have more to.~~

2. We Extend 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 nonterminal A to state j

Algorithm for LR parsing:

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G_1.

OUTPUT: If w is in $L(G_1)$, the reduction steps of a bottom-up parse for w; otherwise, an error ~~production~~ Indication

METHOD: Initially, the parser has S_0 on its stack, where S_0 is the initial state, and w$ is in the input buffer.

```
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;
output the production A → β;
} else if (ACTION[s, a] = accept) break; /* parsing is done */
else call error-recovery routine;
}

Constructing the SLR-parsing tables:

The SLR method for constructing parsing tables is a good starting point for studying LR parsing. We shall refer to the parsing table constructed by this method as an SLR table, and to an LR parser using an SLR-parsing table as an SLR parser. The other two methods augment the SLR method with lookahead information.

Algorithm for constructing an SLR parsing table:

INPUT: An augmented grammar G'.

OUTPUT: The SLR-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(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \to \alpha . a\beta]$ 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 $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'.

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

If any conflicting actions results from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce in this case.

3. The goto transitions for state i are constructed for all nonterminals 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. Initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

SLR

Bottom-up parsers:

1) $S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$. i/p: (a, a)

1. Don't remove any LR and LF
2. Seperate and numbering the productions
   1. $S \rightarrow (L)$
   2. $S \rightarrow a$
   3. $L \rightarrow L,S$
   4. $L \rightarrow S$
3. Find first and follow set

	S	L
FIRST	(a	(a
FOL	$) ,	) ,

4. Augment the grammar.

$S' \rightarrow S$

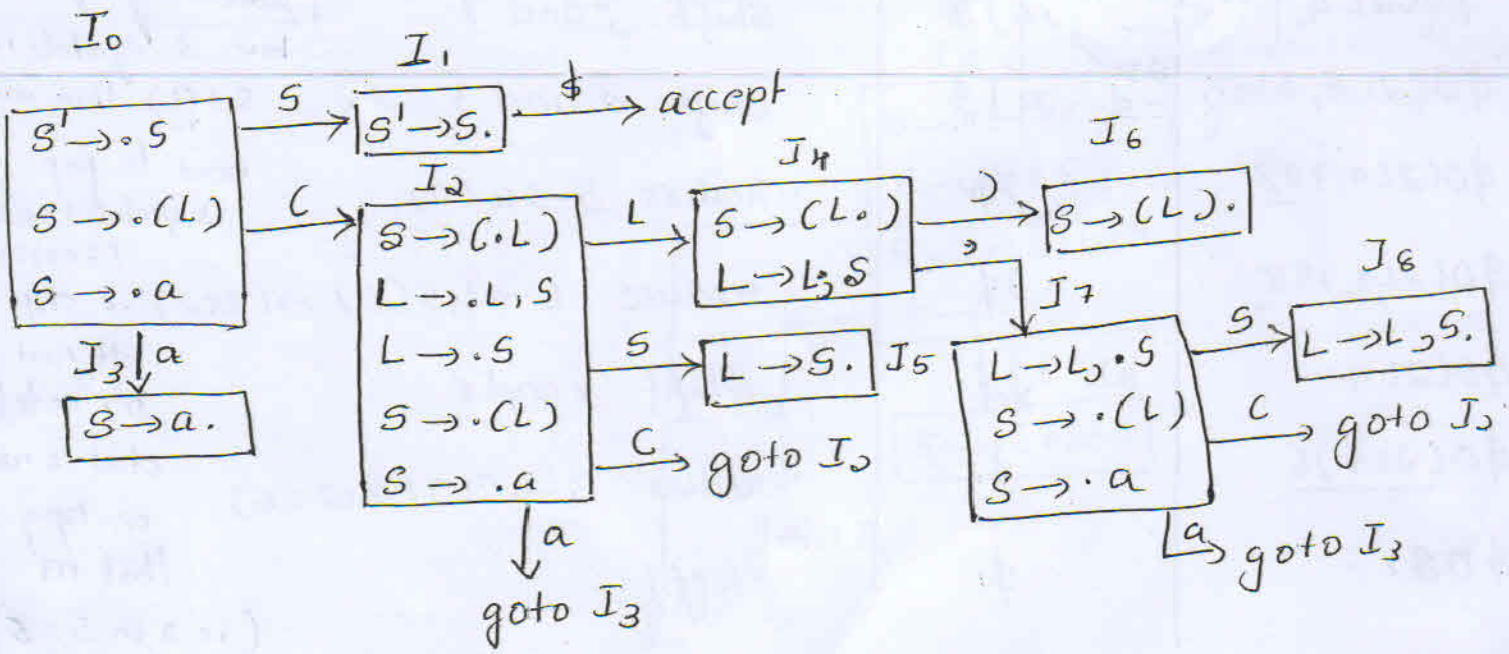
$S \rightarrow (L)$

$S \rightarrow a$

$L \rightarrow L,S$

$L \rightarrow S$

5. Construct closure item set



6. parse table.

	(	a	)	$	,	S	L
0	S_2	S_3				1	
1				accept			
2	S_2	S_3				5	4
3			r_2	r_2	r_2		
4			S_6		S_7		
5			r_4	r_4	r_4		
6			r_1	r_1	r_1		
7	S_2	S_3				8	
8			r_3		r_3		

The given grammar is SLR(1)
∵ no multiple entries

7. i/p: (a,a)

Stack	Input	Action.
$0	(a,a)$	shift (& 2
$0(2	a,a)$	shift a, & 3
$0(2a3	,a)$	reduce S→a (2)
$0(2S5	,a)$	reduce L→S (2)
$0(2L4	,a)$	Shift , and 7
$0(2L4,7	a)$	shift a and 3
$0(2L4,7a3	)$	reduce S→a (2)
$0(2L4,7S8	)$	reduce L→L,S (6) → (3x2) & shift
$0(2L4	)$	Shift) and 6
$0(2L4)6	$	reduce S→(L) (3x2=6)
$0S1	$	accept.

(a) after shifting 'S' we have to see 2 ons = 5

for while reducing, if u have 1 symbol on RHS, then pop 1 symbol & 1 number - totally 2

same way if we have 3 symbols ie S→(L) 1 2 3 then we need to pop 3 symbols + 3 nos ∴ (3+2)=6 & shift left most NT on to the stack & number on top of that NT (ie 2 on S=5)

$S \to iEtS' | iEtSeS | a$ i/p : aa+a*

$E \to b$

(2)

1. Don't remove any LR and LF
2. Seperate and number the production → 1. $S \to iEtS$ 2 $S \to iEtSeS$ 3. $S \to a$ 4 $E \to b$
3. Find first and follow set.
4. Augment the grammar

	S	E
FIRST	i a	b
FOL	$ e	t

$S' \to S$

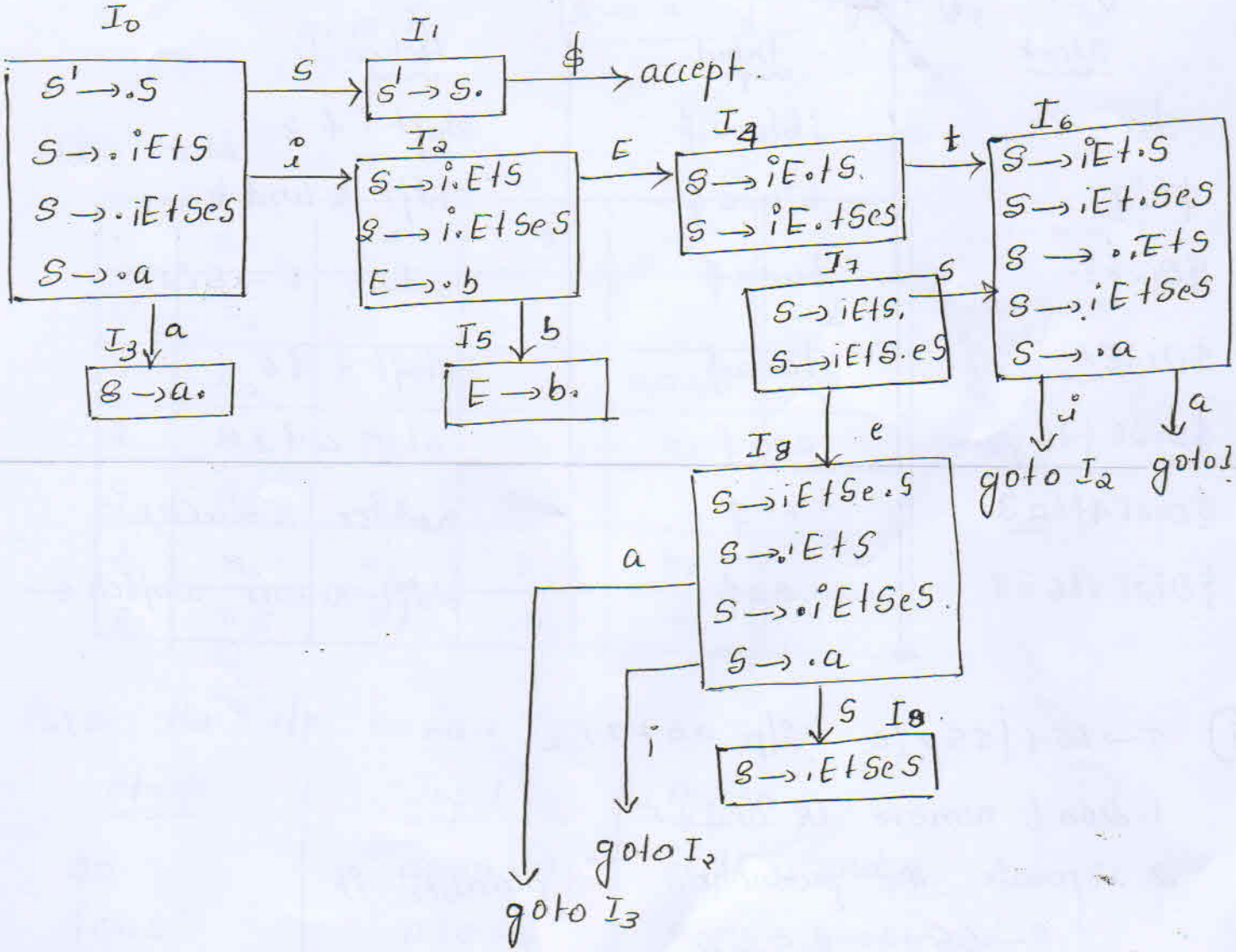
$S \to iEtS$

$S \to iEtSeS$

$S \to a$

$E \to b$.

5. Construct LL(0) or closure item set



6. Parse table.

	e	a	i	t	$	b	S	E
0		S_3	S_2				1	
1.					acept			
2						S_5		4
3	r_3				r_3			
4				S_6				
5				r_4				
6		S_3	S_2				7	
7	S_8/r_1				r_1		8	
8		S_3	S_2					
9	r_2				r_2			

The given grammar is not SLR. There is shift reduce conflicts

7. String parsing i/p: ibtaea

Stack	Input	Action:
$0	ibtaca$	shift i & 2
$0i2	btaea$	shift s and b
$0i2b5	taca$	reduce E → b(2)
$0i2E4	taca$	shift t & 6
$0i2E4t6	aca$	shift a & 3
$0i2E4t6a3	ea$	reduce S → a(2)
$0i2E4t6S7	ea$	shift-reduce conflict ←

(3.) S → SS+ | SS* | a i/p: aa+a*.

1. Don't remove LR and LF
2. Seperate the production and number it
   1. S → SS+ 3. S → a
   2. S → SS*

3. Find first and follow set

	FIRST	FOLLOW
S	a	$ + * a

4. Augment the grammar

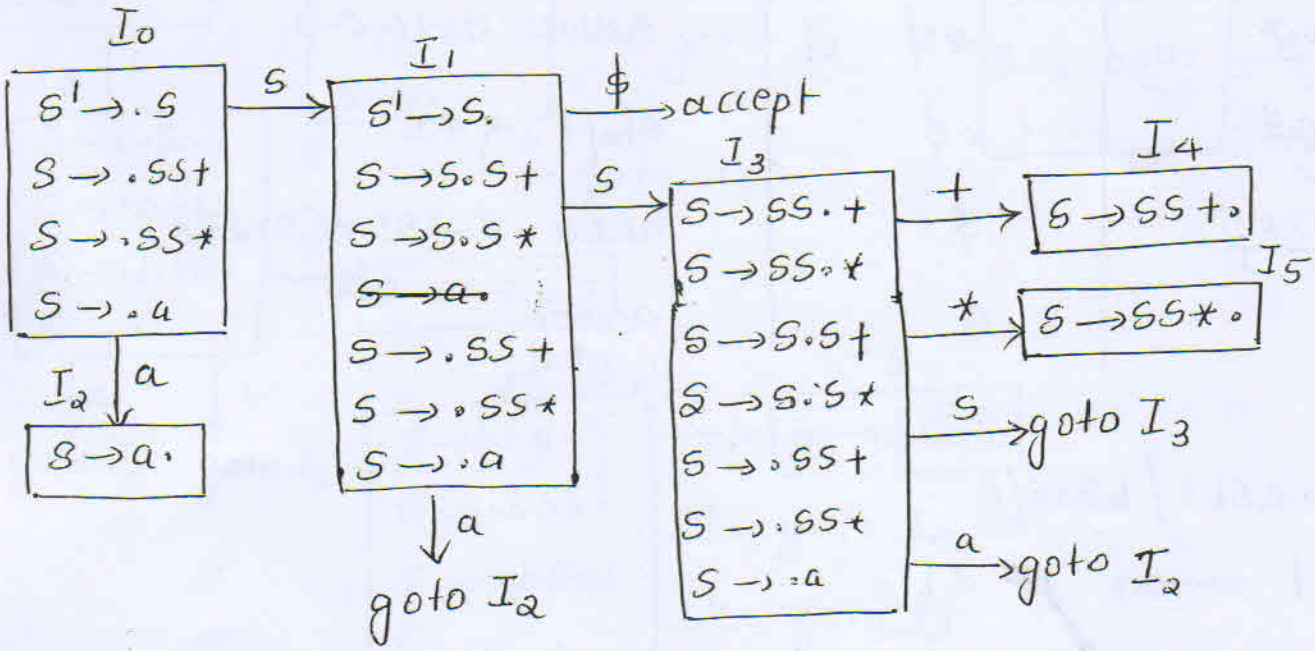
$S' \to S$

$S \to SS+$

$S \to SS*$

$S \to a$

4. Construct closure item set.



5. Parse table.

	a	+	*	$	S
0	S_2				1
1	S_2			accept	3
2	r_3	r_3	r_3	r_3	
3	S_4	S_4	S_5		3
4	r_1	r_1	r_1	r_1	
5	r_2	r_2	r_2	r_2	

The given grammar is SLR.

6. Parse the i/p : aa+a*

Stack	Input	Action
$0	aa+a*$	shift a and 2
$0a2	a+a*$	reduce S → a (2)
$0S1	a+a*$	shift a & 2

$0S1a2	+a*$	reduce S→a (2)
$0S1 S3	+a*$	shift + & 4
$0S1S3+4	a*$	redue S→SS+ (3×2=6)
$0S1	a*$	shift a & 2
$0S1a2	*$	redue S→a (2)
$0S1S3	*$	shift * & 5
$0S1S3*5	$	redue S→SS* (3×2=6)
$0S1	$	accept.

=

④ S → aSbS | bSaS | ϵ

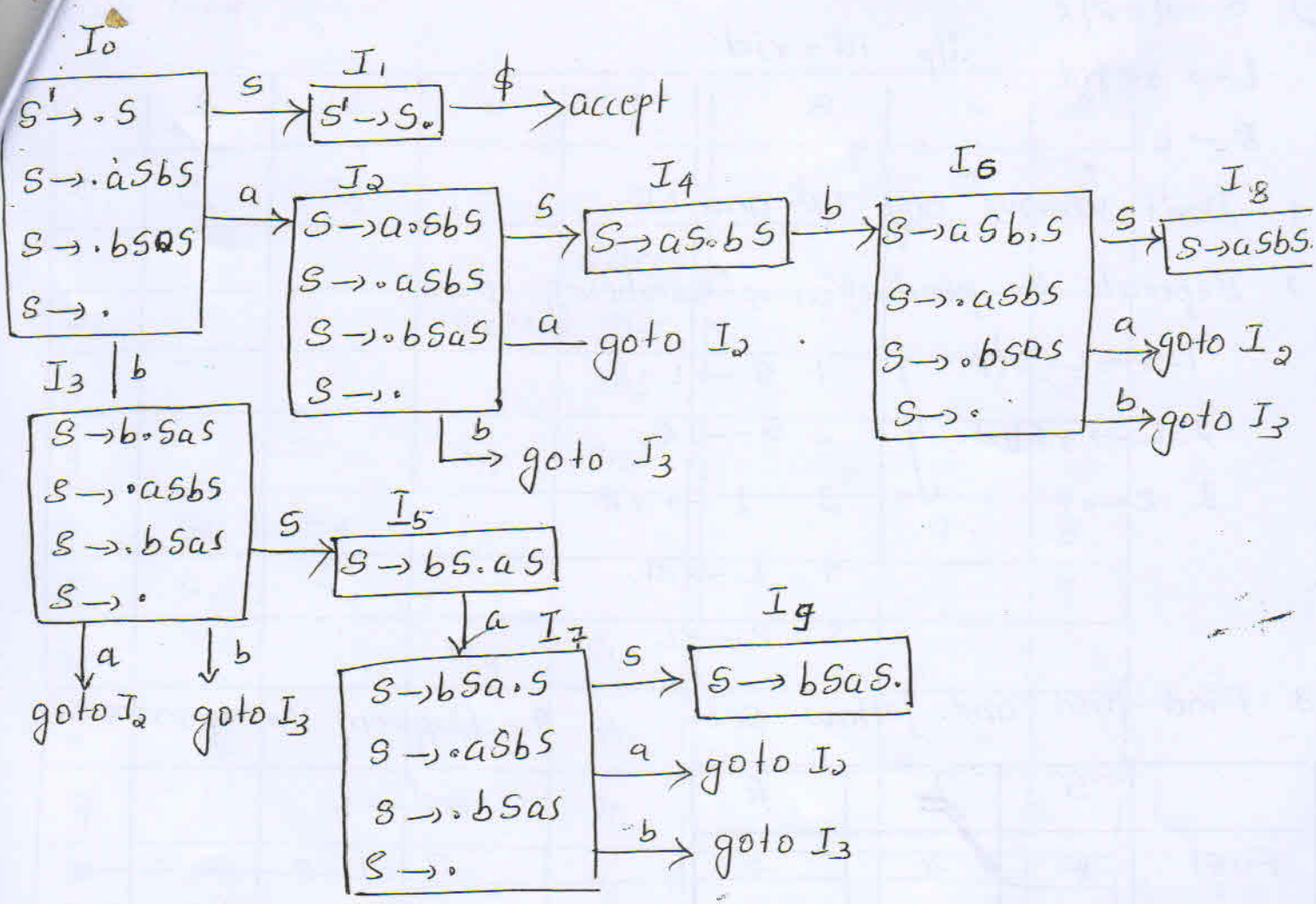
1. Don't remove LR & LF
2. Seperate the grammas and number them.
   1. S → aSbS
   2. S → bSaS
   3. S → ϵ
3. Find first and follow set

	S
FIRST	a b ϵ
FOL	$ a b

4. Augment the gramma

   S' → S
   S → aSbS
   S → bSaS
   S → ϵ

5. Construct the closure item set



6. parse table.

	a	b	$	S
0	S_2/r_3	S_3/r_3	r_3	1
1			accept	
2	S_2/r_3	S_3/r_3	r_3	4
3	S_2/r_3	S_3/r_3		5
4		S_6		
5	S_7			
6	S_2/r_3	S_3/r_3	r_3	8
7	S_2/r_3	S_3/r_3	r_3	9
8	r_1	r_1	r_1	
9	r_2	r_2	r_2	

∴ The given gramman is not SLR
Shift reduce conflicts.

Stack	Input	Action
$0	aabbab$	shift reduce conflict

5) $S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

i/p : $id = *id$.

1. Don't remove any LR and LF
2. Seperate the production & numbering them.

1. $S \rightarrow L = R \mid R$.
2. $L \rightarrow *R \mid id$.
3. $R \rightarrow L$

1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow *R$.
4. $L \rightarrow id$
5. $R \rightarrow L$

3. Find first and follow set.

	S	L	R
First	*	*	*
	id	id	id
Follow	$	=	$
		$	=

4. Augment the grammar.

$S' \rightarrow S$

$S \rightarrow L = R$

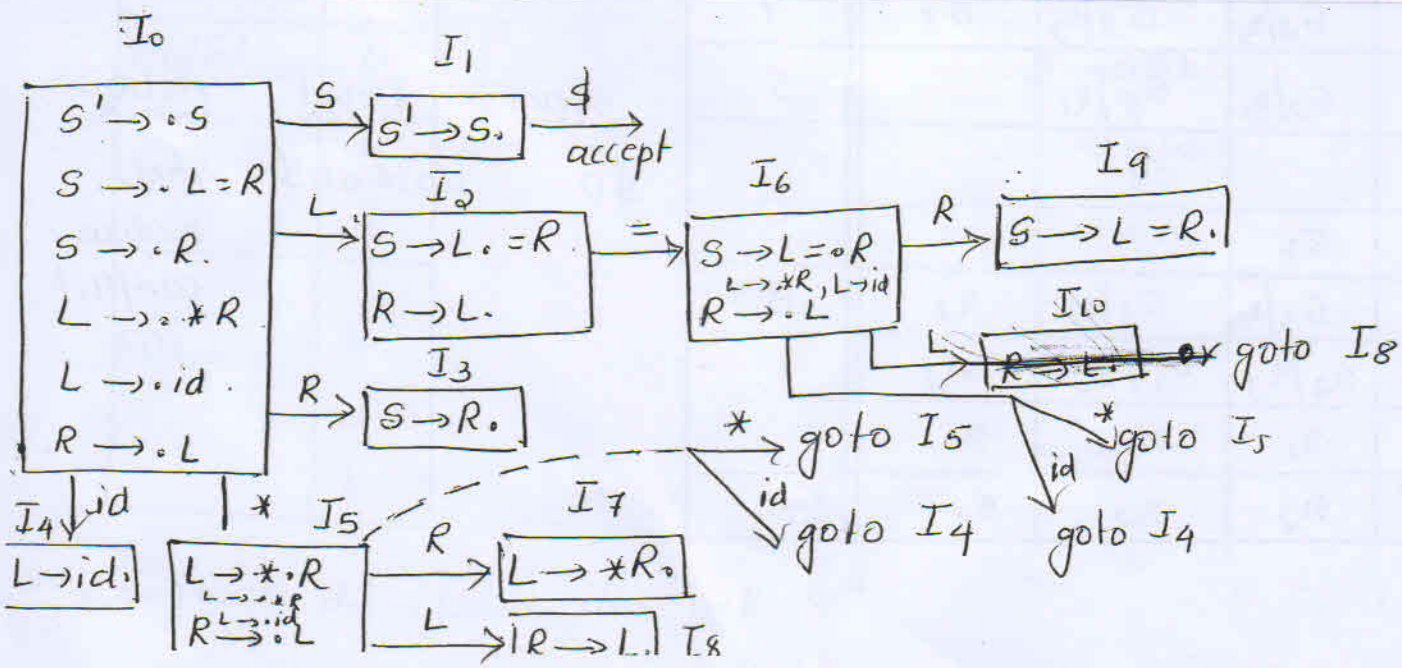
$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

5. Construct closure item set



Parse table.

	*	id	=	$	S	L	R
0	S_5	S_4			1	2	3
1				accept			
2			S_6/r_5	r_5			
3				r_2			
4			r_4	r_4			
5	S_5	S_4				7	8
6	S_5	S_4				8	9
7			r_3	r_3			
8			r_5	r_5			
9			~~r_1~~	r_1			
10			r_5	r_5			

7. i/p: id=*id.

This is not SLR(1) ∵ ~~multiple~~ entries of shift reduce conflict

Stack	Input	Action.
$0	id=*id$	shift id & 4
$0id4	=*id$	reduce L→id (2)
$0L2	=*id$	shift = & 6 ⟶ shift reduce conflict
~~$0L2=6~~	*id$	~~shift *5~~
~~$0L2=6*5~~	id$	shift id & 4
$0L2=6*5id4	$	reduce L→id (2)
$0L2=6*5L7	$	reduce L→*R (2×2=4)
$0L2=6L8	$	reduce R→L (2)
$0L2=6R9	$	reduce R_1 S→L=R (6)
$0S1	$	accept

(6) $S \rightarrow AaAb \mid BbBa$ is LL(1) but not SLR(1).

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

LL(1):

1. Remove left recursion — not required
2. Remove left factoring — not required.
3. Find first and follow set

	S	A	B
first	a b	ϵ	ϵ
follow	$	a b	b a.

4. construct parse table.

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

5. parse the i/p.

stack i/p action

Not needed.

∴ The given grammar is LL(1) ∵ no multiple entries.

SLR(1):

1. Don't remove any LR & LF
2. Seperate the productions and number them.

1. $S \rightarrow AaAb$
2. $S \rightarrow BbBa$
3. $A \rightarrow \epsilon$
4. $A \rightarrow \epsilon$

3. Find first and follow Set.

	S	A	B
first	a b	ϵ	ϵ
follow	$	a b	a b

4. Augment the grammar

$S' \to S$

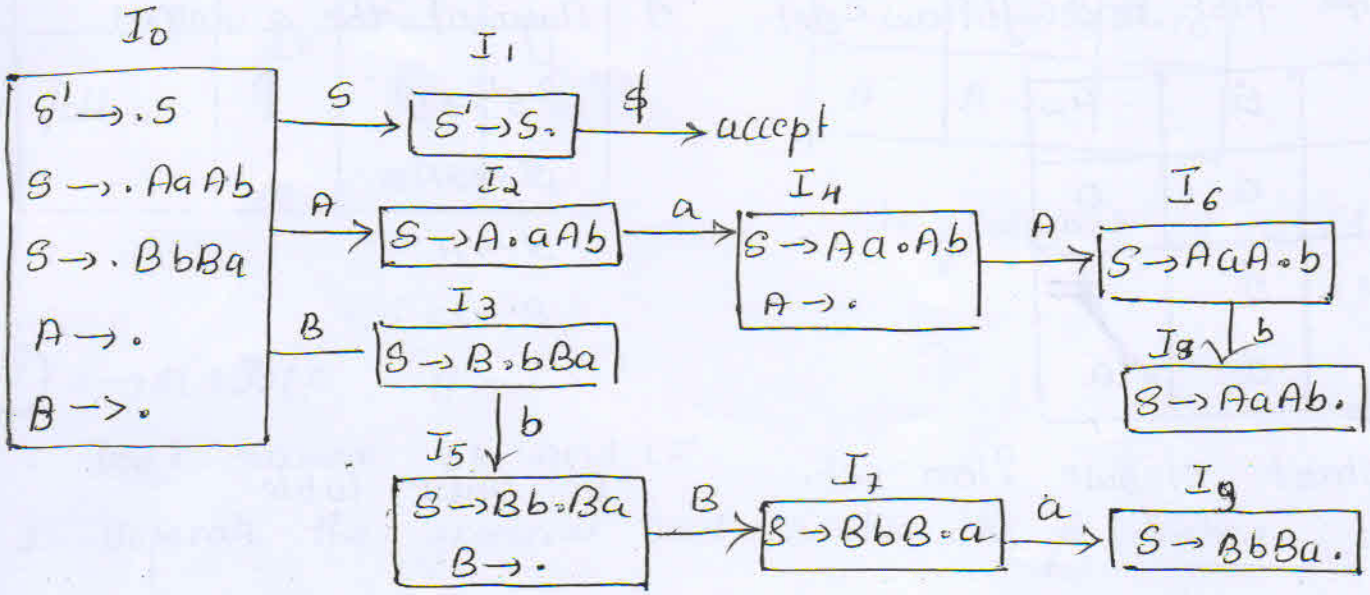
$S \to AaAb$

$S \to BbBa$

$A \to \epsilon$

$B \to \epsilon$

5. Construct closure item set.



6. parse table.

	a	b	$	S	A	B
0	r_3/r_4	r_3/r_4		1	2	3
1			accept			
2	S_4					
3		S_5				
4	r_3	r_3			6	
5	r_4	r_4				7
6		S_8				
7	S_9					
8			r_1			
9			r_2			

The given grammar is not SLR(1) ∵ reduce-reduce conflicts.

7) $S \to SA \mid A$
$A \to a$

is SLR(1) and ~~but not~~ LL(1)

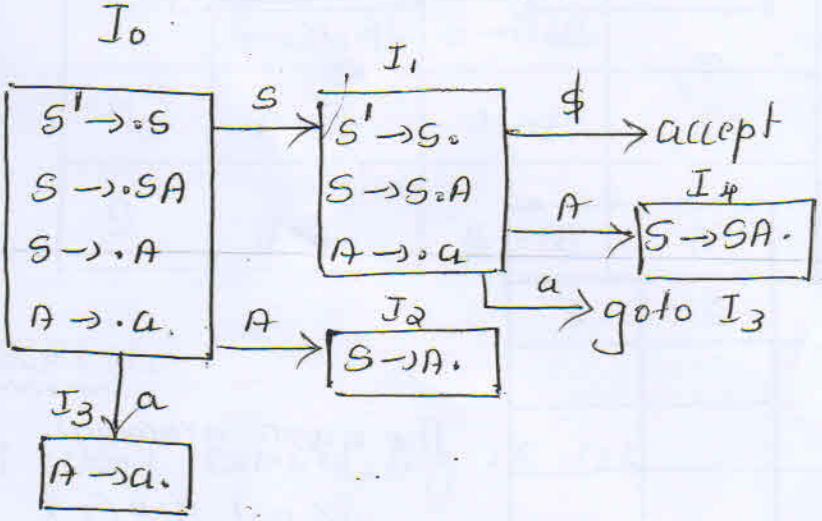
SLR(1)

1. Don't remove any LR & LF
2. Seperate and number the production.
   1. $S \to SA$
   2. $S \to A$
   3. $A \to a$
3. Find first and follow set.

	S	A
first	a	a
follow	$	$
	a	a

4. Augment the grammar
   $S' \to S$
   $S \to SA$
   $S \to A$
   $A \to a$.

5. Construct closure item set.



6. parse table

	a	$	S	A
0	S_3		1	2
1	S_3	accept		4
2	r_2	r_2		
3	r_3	r_3		
4	r_1	r_1		

∴ The given grammar is SLR

LL(1):

1. Remove left recursion
   $A \to A\alpha \mid B \Rightarrow A \to BA'$
   $A' \to \alpha A' \mid \epsilon$

$S \to SA/A$ $\quad S \to AS'$

$\alpha \quad \beta$ $\quad S' \to AS'/\epsilon$

$A \to a$

2. Remove left factoring -not required.

3. Find first and follow set

	S	S'	A
first	a	a ϵ	a
follow	$	$ a	$ a

4. parse table.

	a	$
S	$S \to AS'$	
S'	$S' \to AS'$	$S' \to \epsilon$
A	$A \to a$	

∴ The grammar is LL(1)

(8) $S \to S(S)S/\epsilon$ i/p : (()

1. Don't remove LR and LF.

2. Seperate the grammar and number the production.

   1. $S \to S(S)S$

   2. $S \to \epsilon$

3. Find first and follow set

	S		S
FIRST	ϵ (	FOLLOW	$ ( )

4. Augment the grammar

$S' \to S$

$S \to S(S)S$

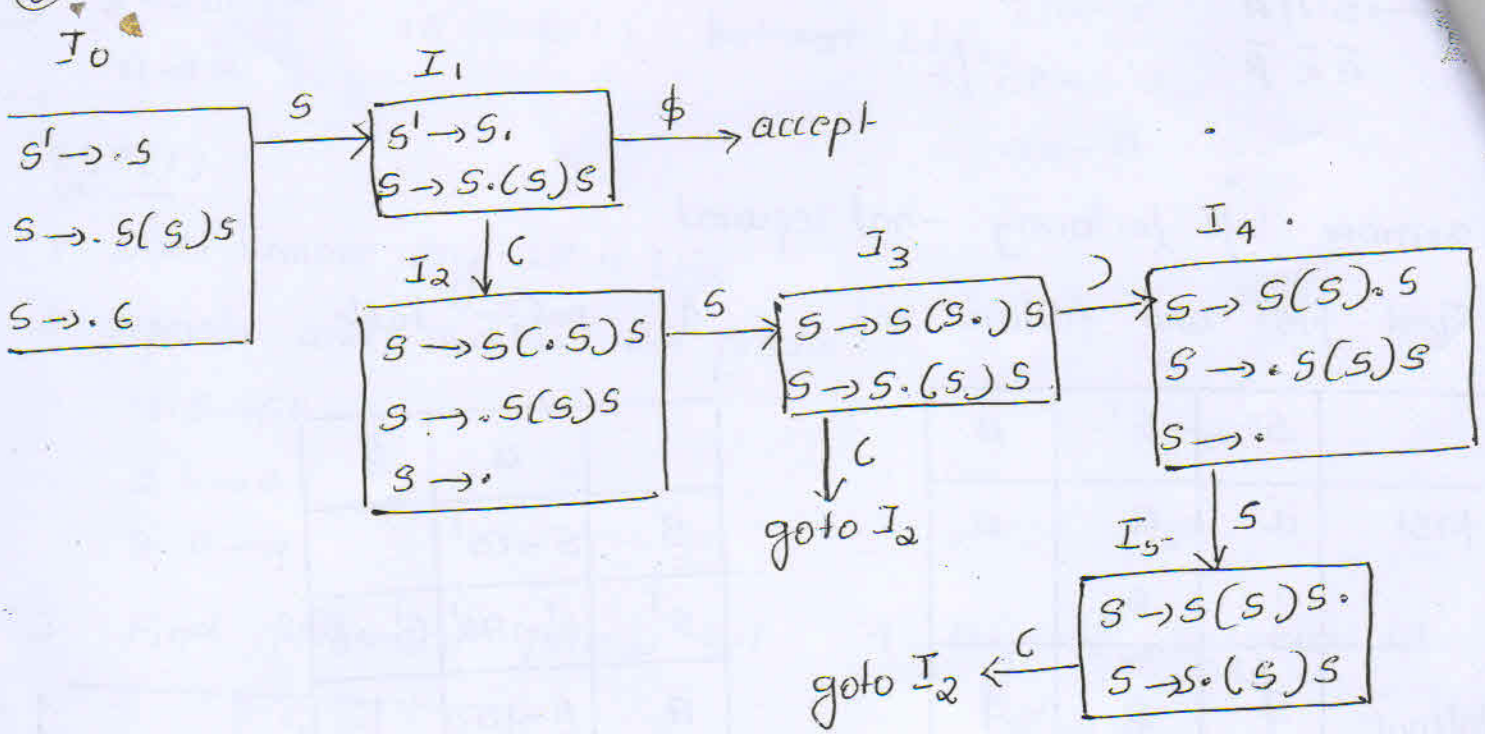
$S \to \epsilon$

5. Construct closure item set (next page)

6. Construct parsing table. (next page)

7. Parse the i/p string (next page)

⑤



⑥

	C	$	)	S
0	r_2	r_2	r_2	1
1	S_2	accept		
2	r_2	r_2	r_2	3
3	S_2		S_4	
4	r_2	r_2	r_2	5
5	S_2 / r_1	r_1	r_1	

The given grammar is not SLR(1)

∵ shift reduce conflict.

Stack	I/P	Action
$0	C()$	r_2, S → ∈
$0S1	C()$	shift C & 2
$0S1C2	()$	r_2, S → ∈
$0S1C2S3	()$	S_2, C & 2
$0S1C2S3C2	)$	shift) & 4
$0S1C2S3C2S4	)$	S → ∈
$0S1C2S3C2S3	)$	shift) & 4
$0S1C2S3C2S3)4	$	S → ∈
$0S1C2S3C2S3	$	

accept

I/p : (())

↳ shift reduce conflict.

⑦

Stack	Input	Action
$0	(())	r_2, S → ∈

$0S1	(()()) $	shift (& 2
$0S1(2	()()) $	$r_3, S \to G$
$0S1(2S3	())) $	shift (& 3
$0S1(2S3(3	)) $	shift) & 4
$0S1(2S3(3)4	) $	$r_2, S \to G$
$0S1(2S3(3)4S5	$	shift reduce conflict

```
else if ( ACTION [s, a] = reduce A → B ) {
    pop |β| sym of stack ;
    let state t now be on top of stack ;
    [ t, A ] onto stack ;
    output the production A → β ;
}
else if ( ACTION [s, a] = accept ) break ;
else call error recovery routine ;
}
```

Construct the SLR parsing table for the following grammar and parse the i/p string:

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

i/p : id+id*id .

→ 1. Don't remove any LR & LF

2. Seperate productions and number them from one to N.

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

3. Find FIRST and FOLLOW set.

	E	T	F
FIRST	(id	(id	(id

	id	id	id
FOLLOW	$	$	$
	)	)	)
	+	+	+
		*	*

4. Augment the grammar.

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

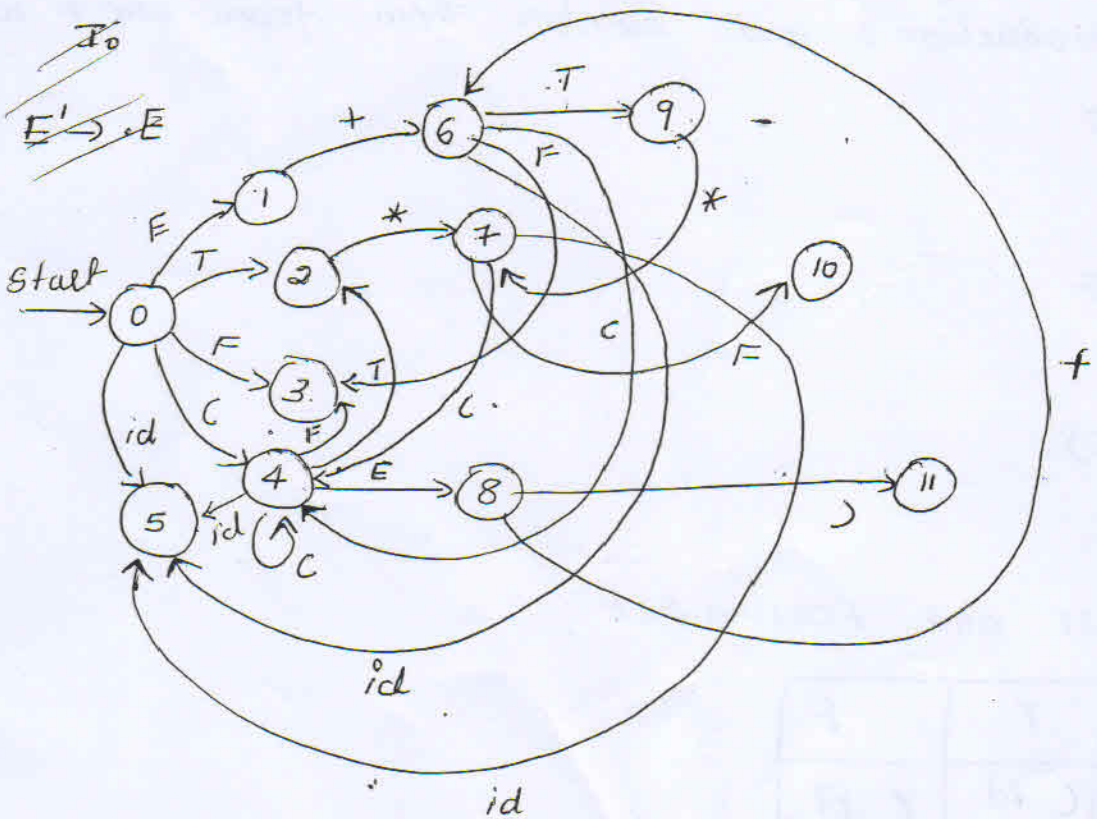
$T \rightarrow F$

$F \rightarrow (E)$

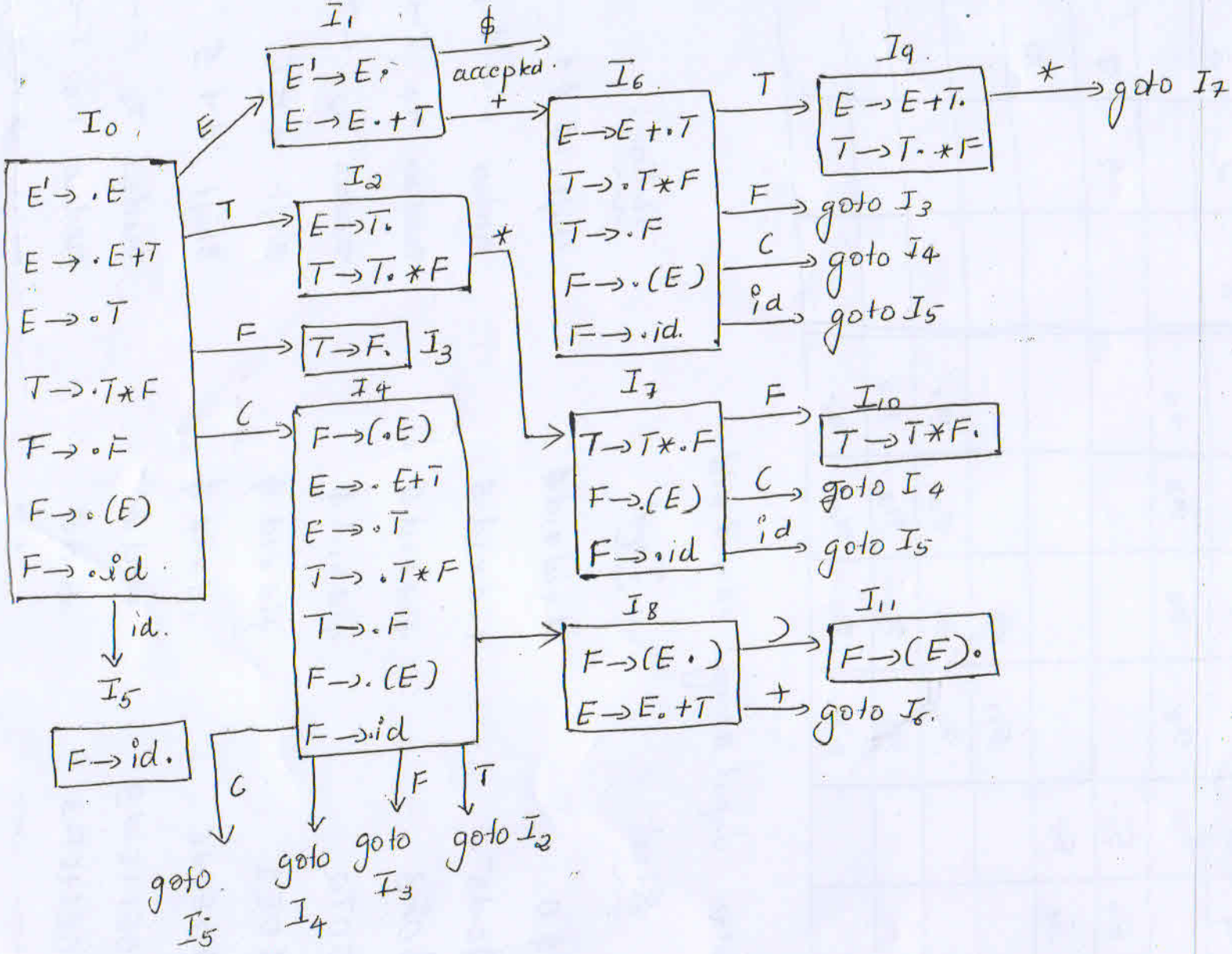
$F \rightarrow id$

5. Construction of closure (LR(0)) item set.

Transition diagram:



Construction of closure of item Set



6. Parsing table.

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

7. Parse input string: id + id * id.

Stack	Input	Action:
$0	id + id * id $	shift id & 5
$0 id 5	+ id * id $	reduce F → id. (1 x 2 = 2)
$0 F 3	+ id * id $	reduce r_4 T → F (2)
$0 T 2	+ id * id $	reduce r_2 E → T (2)
$0 E 1	+ id * id $	shift + & 6
$0 E 1 + 6	id * id $	shift id & 5
$0 E 1 + 6 id 5	* id $	reduce r_6 F → id (2)
$0 E 1 + 6 F 3	* id $	reduce r_4 T → F (2)
...	...	...

$0E1+6T9*7	id $	shift id & 5
$0E1+6T9*7id5	$	reduce F→id (5)
$0E1+6T9*7F10	$	reduce 3 T→T*F (6)
$0E1+6T9	$	reduce 1, E→E+T
$0E1	$	accept

9. S→+SS|*SS|a i/p: +*aaa

10. S→S+S|SS|(S)|S*|a i/p: (a+a)*a.

Ans:

9.1. Don't remove LR and LF.

2. Seperate and give the number

1. S→+SS
2. S→*SS
3. S→a.

3. Find first and follow set

	S		S
FIRST	+	FOLLOW	$
	*		+
	a		*

4. Augment the grammar

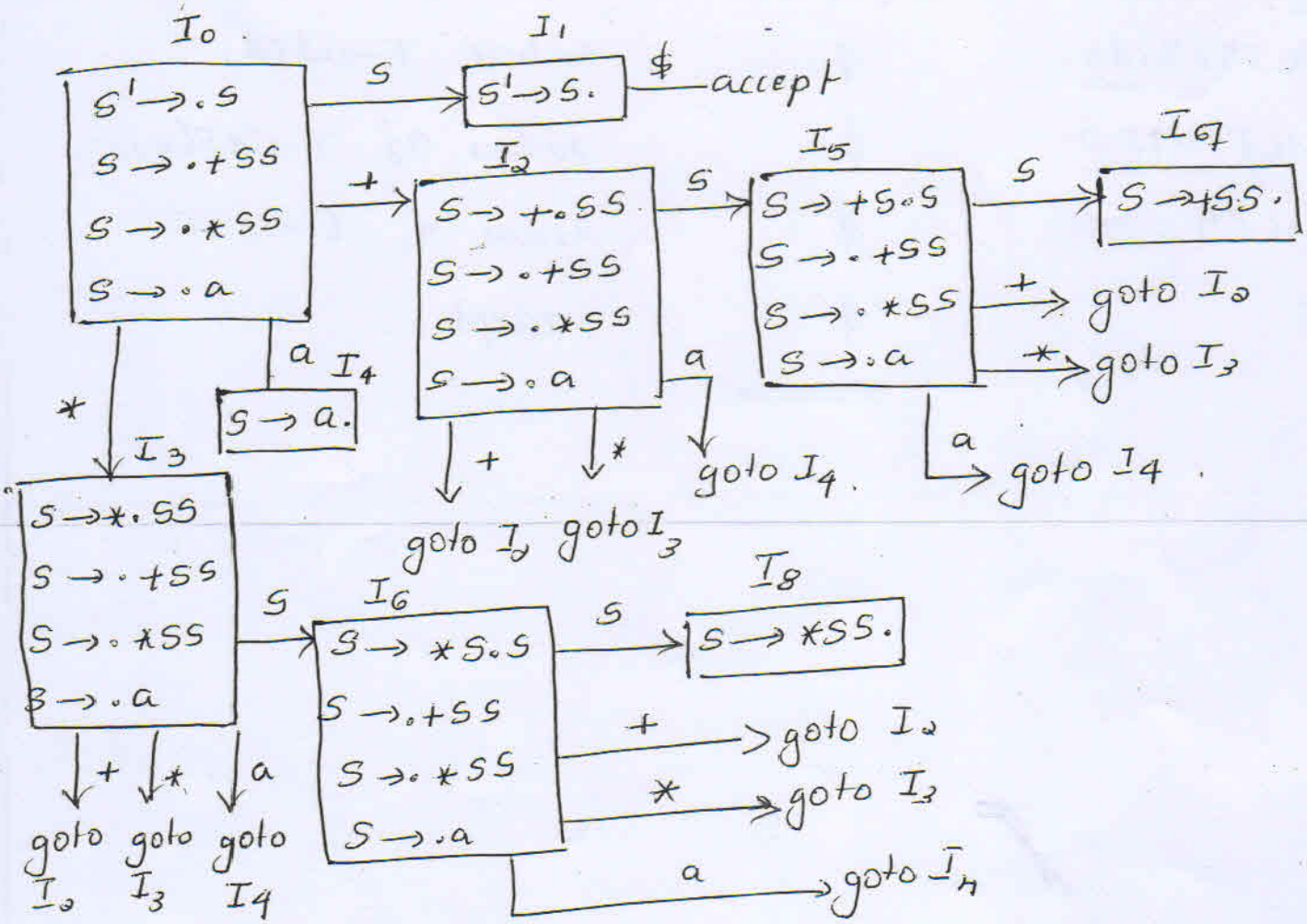
S'→S

S→+SS

S→*SS

S→a.

5. Construct closure item set.



6. Construct parse table.

	+	*	a	$	S
0	S_2	S_3	S_4		1
1				accept	
2	S_2	S_3	S_4		5
3	S_2	S_3	S_4		6
4	r_3	r_3	r_3	r_3	
5	S_2	S_3	S_4		7
6	S_2	S_3	S_4		8
7	r_1	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	

7. parse the i/p string.

Stack	Input	Action :
$0	+*aaa$	shift +, 2
$0+2	*aaa$	shift *, 3
$0+2*3	aaa$	shift a & 4
$0+2*3a4	aa$	r_3, S→a(2)
$0+2*3S6	aa$	shift a & 4
$0+2*3S6a4	a$	r_3, S→a(2)
$0+2*3S6S8	a$	r_2, S→*SS(6)
$0+2S5	a$	shift a & 4
$0+2S5a4	$	reduce S→a(2)
$0+2S5S7	$	S→+SS(6)
$0S1	$	accept

10. S→S+S|SS|(S)|S*|a

1. Don't remove any LR & LF.
2. Seperate the productions and number them.
   1. S→S+S
   2. S→SS
   3. S→(S)
   4. S→S*
   5. S→a
3. Find FIRST and FOLLOW set

	S		S
FIRST	a (	FOLLOW	$ +) * a (

4. Augment the grammar.

$S' \to S$

$S \to S+S$

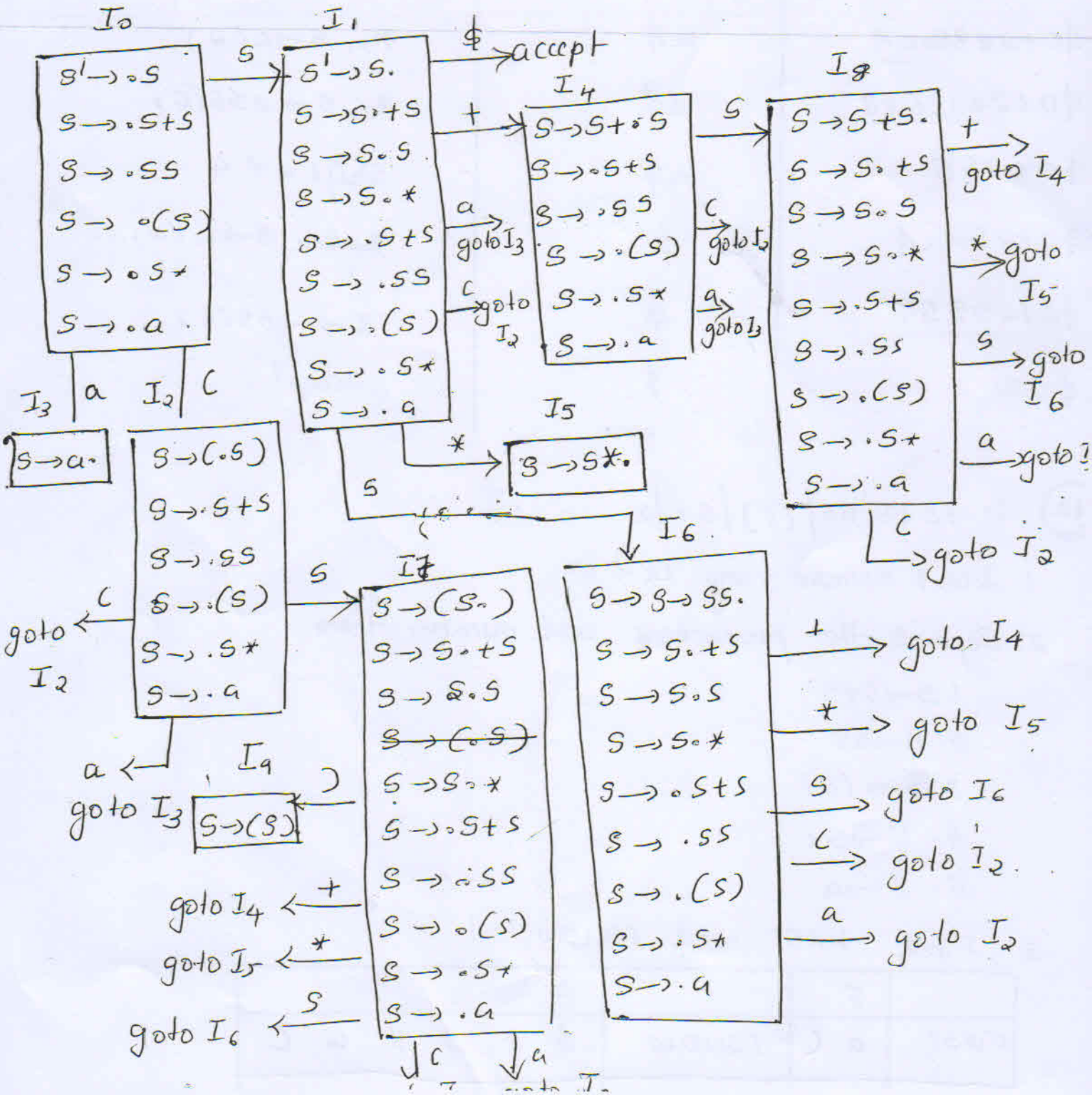
$S \to SS$

$S \to (S)$

$S \to S*$

$S \to a$

5. Find closure item set



	a	+	(	)	*	$	S.
0	S_3		S_2				1
1	S_3	S_4	S_2		S_5	accept	6
2	S_3		S_2				7
3	r_5	r_5	r_5	r_5	r_5	r_5	
4	S_3		S_2				8
5	r_4	r_4	r_4	r_4	r_4	r_4	
6	S_3/r_2	S_4/r_2	S_2/r_2	r_2	S_5/r_2	r_2	
7	S_3	S_4	S_2	S_9	S_5		6
8	S_3/r_1	S_4/r_1	S_2/r_1	r_1	S_5/r_1	r_1	6
9	r_3	r_3	r_3	r_3	r_3	r_3	

∴ The given grammar is not SLR(1) ∵ shift reduce conflict.

∴ parse the i/p string

Stack	Input	Action
$0	(a+a)*a$	shift (& 3
$0(3	a+a)*a$	S → a (2)
$0S1	a+a)*a$	shift a & 3
$0S1a3	+a)*a$	S → a(2).
$0S1S6	+a)*a$	shift reduce conflict.

Viable prefix:

The prefix of the RSF that can appear on the stack of shift reduce parser are called viable prefix. In other words the viable prefix is a prefix of a RSF that doesnot continue passt the right end of the right most handle of that sentential form. By this it is always possible to add terminal

Viable prefix:

→ The prefix on the Right Sentential form that can be appear on the stacks of S-R parser are called Viable prefixes. In other words the Viable prefix is a prefix is a prefix of Right sentential form That does not Continue past the right End of the rightmost handle of that sentential form by this It is always possible to add terminal symbols to the End of the Viable prefix to obtain The Right Sentential form

→ The Viable prefix is based on the fact that LR(0) automata, recognizes Viable prefixes

→ we say item $A \to \beta_1 . \beta_2$ is valid for a Viable prefix $\alpha\beta_1$ If there is a derivation of the form $S \xRightarrow[rm]{*} \alpha Aw \xRightarrow[rm]{} \alpha\beta_1\beta_2 w$

→ In general, an Item will be valid for many Viable prefixes

here $A \to \beta_1 . \beta_2$ tells us whether to shift / reduce when we find $\alpha\beta_1$ on the parsing stack. In particular If $\beta_2 \neq \varepsilon$, Then we have to shift onto the stack If $\beta_2 = \varepsilon$, then $A \to \beta_1$ is the handle and we have to reduce by the production

eg: $E \xRightarrow[rm]{*} F * id \xRightarrow[rm]{} (E) * id$

at various times, during parse the stack will hold (, (E, (E) which are the viable prefix. but it should not hold (E)* because, (E) is a handle, which the parser must reduce to F, before shifting *