BOTTOM-UP PARSING

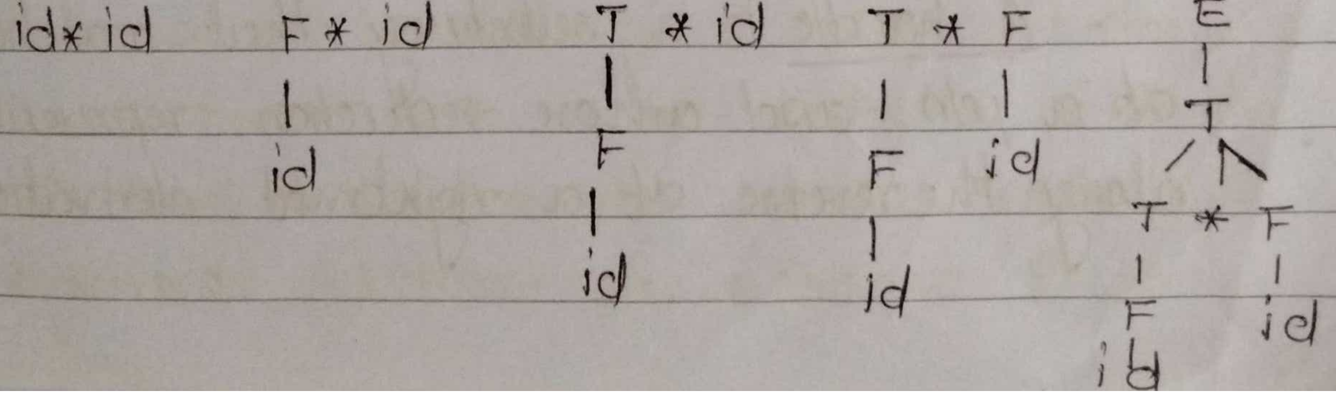
A bottom up parse corresponds to construction of a parse tree for an i/p string beginning at the leaves (the bottom) and working up towards the root (the top).

eg:1 bottom up parser for id*id. The grammar is

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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



eg:

Reduction Reduction is the 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 nonterminal at the head of the pdn.
- A reduction is the reverse of a step in derivation. (in derivation, a nonterminal in sentential form is replaced by the body of one of its pdn.
- The goal of bottom-up parsing is to construct a derivation in reverse.
- The key decisions during bottom-up parsing are about when to reduce and about what pdns to apply as the parse proceeds.
- The following corresponds to parse in eg ①.

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

This derivation is in fact a rightmost derivation

Handle Pruning (The process to construct a bottom-up parse is called handle Pruning)

Bottom-up parsing during a left to right scan of the i/p constructs a rightmost derivation in reverse.

- A handle is a substring that matches the body of a pdn, and whose reduction, represents one step along the reverse of a rightmost derivation.

eg: Adding subscripts to the tokens id for clarity the handles during the parsing of $id_1 * id_2$ according to expr. grammar in eg① is given in table below:

Right Sentential Form	Handle	Reducing Production
$id_1 * id_2$	id_1	F → id
$F * id_2$	F	T → F
$T * id_2$	id_2	F → id
T * F	T*F	T → T*F
T	T	E → T

Consider the sentential form T* id and the pdn E ⟶ T, Here T is not a handle in the sentential form T * id. If T was indeed replaced by E, we would get the string E*id.

Shift Reduce Parsing

It is a form of bottom up parsing in which a stack holds grammer symbols and an i/p buffer holds the rest of the string to be parsed. The handle always appears at the top of the stack. we use $ to mark the bottom of the stack and also the right end of the i/p. Initially the stack is empty and the string w is on the i/p as follows

STACK	INPUT
\$	w\$

During a left to right scan of the i/p string the parser shift zero or more i/p symbols on to the stack until it is ready to reduce a string β of grammar symbol on top of the stack. It then reduce β to the head of the appropriate production the parser repeat this cycle until it has detected an error or until the stack contains the start symbol and the i/p is empty.

STACK	INPUT
\$S	\$

Upon entering this configuration the parser stops and announces successful completion of parsing.

$E \rightarrow E+T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

STACK	INPUT	ACTION
$	$id_1 * id_2 \$$	shift
$\$id_1$	$* id_2 \$$	reduce F→id
$\$F$	$* id_2 \$$	reduce T→F
$\$T$	$* id_2 \$$	shift
$\$T*$	$id_2 \$$	shift
$\$T*id_2$	$	reduce F→id
$\$T*F$	$	reduce T→T*F
$\$T$	$	reduce E→T
$\$E$	$	accept

Four possible actions a shift reduce parser can make.

1. Shift – shift the next i/p symbol on the top of the stack.
2. Reduce – the right end of the string to be reduced must be at the top of the stack.
3. locate the left end of the string within the stack and decide with what non terminal to replace the string.

3. accept - announce successful completion of parser.

4. error - discover a syntax error and ~~hold~~ call~~ed~~ error recovery routine.

Conflit during shift reduce parsing

There are 2 types of conflits in SRP.

① shift or reduce conflit

The situation in which the parser cannot decide whether to shift or to reduce.

stmt → .if expr then stmt /
.if expr then stmt else stmt /
. other

STACK	INPUT
$ if expr then stmt	else stmt $

we cannot tell whether if expression then stmt is the handle. Here there is a shift or reduce conflit

② reduce-reduce conflit

The situation in which the parser cannot decide which of several productions to make

Consider the grammar,

1. stmt → id (parameter_list)
2. stmt → expr := expr
3. parameter_list → parameter_list/parameter
4. parameter → 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)$

After shifting, a shift reduce p would be in a configuration id(id, id) $.
There is a conflit that which production either's '5' or '7' can be used to reduce. This conflit is called reduce-reduce conflit.

OPERATOR PRECEDENCE PARSING

Operator Grammar:

It is a CFG that has the following properties:

- There is no ε production on the right side
- There is no adjacent nonterminals on the right side of the pdn.

E ⟶ EAE | (E) | -E | id

A ⟶ + | - | * | / | ↑

This is not an operator grammar because E ⟶ EAA has 3 consecutive nonterminals

This can be converted into an operator grammar ie,

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid id$

Operator precedence parser:

- It is a bottom up parser that interprets an operator grammar.
- Operator precedence parser is the only one parser that can parse ambiguous grammars.

Disadvantages:

- It is hard to handle tokens like the minus sign, which has two different precedences (depending on whether it is unary or binary).
- Only a small class of grammars can be parsed using operator precedence technique.

Advantage:

- Simple
- Numerous compilers use operator precedence parser for handling expressions.

- In operator precedence parsing, we define 3 disjoint precedence r/ns ~~bt~~ between certain pair of terminals ($<\cdot$, $=$ and $\cdot>$).

$a <\cdot b$: a yields precedence to b

$a \cdot> b$: a takes precedence over b

$a = b$: a has the same precedence as b

- The determination of correct presedence relation b/w terminals are based on the traditional notations of assosiativity and precedense of operator.

- The intention of precedence relation is to <u>find the handle</u> of a right sentential form.
- The handle can be find out by the following process.

1. scan the symbol from left end until the first ·> symbol is encountered.

2. Then scan backwords over any =· until a <· is encountered.

3. The handle contain everything to the left of the first greater (·>) than and to the right of the less than (<·) encountered in step 2.

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

Consider the i/p string id+id*id apply the precedence relation

E → E+E / E*E / id

~~w → id +id*id~~

w → id + id * id ⇒ $ id + id * id$.

$ <·id ·> + <· id ·> * <· id ·> $

$ <· + <· id > * <· id ·> $

$ <· + <· * <· id ·> $

$ < + <· * ·> $

$ <· + ·> $

$ $

Operator Precedence Relations from Associativity and Precedence.

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$

eg: If $*$ is having higher precedence than $+$ make $* \gtrdot +$ and $+ \lessdot *$

2. If θ_1 and θ_2 are operators of equal precedence then make $\theta_1 \gtrdot \theta_2$ and $\theta_2 < \theta_1$, if operators are left associative, or make $\theta_1 \lessdot \theta_2$ and $\theta_2 \gtrdot \theta_1$ if they are right associative

eg: If $+$ and $-$ are left associative, then make $+ \gtrdot +$, $+ \gtrdot -$, $- \gtrdot +$, and $- \gtrdot -$

If $\uparrow$ is right associative then make $\uparrow \lessdot \uparrow$.

3 make $\theta \lessdot id$, $id \gtrdot \theta$, $\theta \lessdot ($, $(\lessdot \theta$, $) \gtrdot \theta$, $\theta \gtrdot)$, $\theta \gtrdot \$$ and $\$ \lessdot \theta$ for all operators θ.
Also let

$(\doteq)$ $\$ \lessdot ($ $\$ \lessdot id$ $(\lessdot ($
$id \gtrdot \$$ $) \gtrdot \$$ $(\lessdot id$ $id \gtrdot)$ $) \gtrdot)$

These rules ensure that id will be reduced to E wherever found and (E) will be reduced to E wherever found.

- The operator precedence relations for grammar assuming

1. $\uparrow$ is of highest precedence and right associative
2. * and / are of next precedence and left associative
3. + and - are of lowest precedence and left associative is given below:

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

Operator Precedence Parsing Algorithms:

Input: An i/p string w and a table of precedence relations.

Output: If 'w' is well formed, a skeletal parse tree, with a placeholder nonterminal E labeling all interior nodes, otherwise, an error condn.

Method: Initially the stack contains $ and the i/p buffer contains the string w$.

```
1. set ip to point to the first symbol of w$
   repeat forever
   if $ is on top of stack and ip points to $ then
         accept and return.
   else begin
        let 'a' be the topmost terminal symbol on the
```

```
stack and let b be the symbol pointed to by ip
if a <· b or a ≐ b then begin
        push b onto the stack
        advance ip to the next i/p symbol
end
else if a ·> b then // reduce
      repeat
          pop the stack
      until the top stack element terminal is
      related by <· to the terminal most recently
                                           popped
else
      error();
end.
```

eg:

stack	i/p buffer	Action
$	a+(b*c)$	$<·a push
$a	+(b*c)$	a·>+ pop.
$	+(b*c)$	$<·a noop
$	+(b*c)$	$<·+ push
$+	(b*c)$	+<·(push
$+(	b*c)$	(<·b push
$+(b	*c)$	b·>* pop
$+(	*c)$	(<·b noop
$+(	*c)$	(<·* push
$+(*	c)$	*<·c push

$ + (* (	) $	(⋗) pop
$ + (*	) $	* <· (noop
$ + (*	) $	* ⋗) pop
$ + (	) $	(<· * noop
$ + (	) $	(≐) push
$ + ( )	$	) ⋗ $ pop
$ + (	$	(≐) pop
$ +	$	+ ⋗ (noop
~~$ +~~	$	$ (~~$ <· + no~~
$ +	$	$ <· + noop
$	$	Accept.

Compilers using operator precedence parsers need not store the table of precedence relations. In most cases the table can be encoded by two precedence functions f and g, which map terminal symbols to integers.

1. $f(a) < g(b)$ whenever $a \lessdot b$
2. $f(a) = g(b)$ whenever $a \doteq b$
3. $f(a) > g(b)$ whenever $a \gtrdot b$

Thus the precedence r/n b/w a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$.

LR parser is a bottom-up parser which is used to parse a large class of context-free grammars. In LR(k) parser; 'L' is for left to right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and 'k' for the number of input symbols of lookahead that are used in making parsing decisions.

LR parser is attractive because:

> LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.

> LR parsing method is the most general nonbacktracking shift-reduce parsing method known.

> The class of grammars that can be parsed using LR method is a proper superset of the class of grammars that can be parsed with predictive parsers.

> An LR parser can detect a syntactic error as soon as it is possible to do so on a left to right scan of the input.

The principal drawback of the method is that it is too much work to construct an LR parser by hand for a programming language.

The LR Parsing Algorithm.

The schematic form of an LR parser is shown below. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts, 'action' and 'goto.' The parsing program reads characters from input buffer one at a time. The ~~use~~stack store a string of the form $S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m$, where S_m is on top. Each X_i is a grammar symbol and S_i is state of DFA.

function action and a goto function goto. The parser determin S_m, the state currently on top of the stack, and a_i, the current input symbol, it then consult action[S_m, a_i], which can have one of four values:

1. Shift s, where 's' is a state.
2. reduce by a grammar production $A \to \beta$,
3. accept,
4. error.

The goto takes a state and grammar symbol as argumen and produces a state.

The parser table can be constructed using 3 method

1. Simple LR (SLR)
2. Canonical LR
3. Lookahead-LR (LALR).

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), a bottom-up parse for w; otherw 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 program below until an accept or error action is encountered.

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
    a the symbol pointed to by ip;
    if action[s, a] = shift s' then
```

```
            push a then s' on top of the stack;
            advance ip to the next input symbol,
        end
    else if action [s, a] = reduce $A \rightarrow \beta$ then
        begin
            pop $2*|\beta|$ symbol off the stack; - - - - -> eg:- $2*|E+T|$
                                                            $= 2*3 = 6$
            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
```

Example

Consider the either grammar for arithmetic expressions with binary operators + and *;

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

The figure below shows the parsing action and goto functions of LR parsing table for the above grammer. The code for actions are:

1. Si means shift and stack state i,
2. rj means reduce by production numbered j,
3. acc means accept
4. blank means error.

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

On input id*id+id, the sequence of stack and input content shown below.

stack	input	Action.
1) 0	id*id+id $	Shift.
2) 0 id5	*id+id $	reduce by F→id
3) 0F3	*id+id $	reduce by T→F
4) 0T2	*id+id $	Shift
5) 0T2*7	id+id $	Shift
6) 0T2*7id5	+id $	reduce by F→id
7) 0T2*7F10	+id $	reduce by T→T*F
8) 0T2	+id $	reduce by E→T
9) 0E1	+id $	Shift
10) 0E1+6	id $	Shift
11) 0E1+6id5	$	reduce by F→id
12) 0E1+6F3	$	reduce by T→F
13) 0E1+6T9	$	E→E+T.
14) 0E1	$	accept.

There are three methods to construct an LR parsing table from a grammar.

1. Simple LR or SLR :- is the weakest of the three in terms of the number of grammars for which it succeeds, but is the easiest to implement.
2. Canonical LR :- is the most powerful and most expensive.
3. Lookahead LR (LALR) :- is intermediate in power and cost between the other two.

SLR Parsing Tables

The parsing table constructed by this method as an SLR table, and the LR parser using an SLR parsing table is called an SLR parser. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.

The central idea in the SLR method is first to construct a DFA from the grammar to recognize viable prefixes[The set of prefixes of right sentential form that can appear on the stack of a shift-reduce parser are called viable prefixes].
To construct the DFA we make use of augmented gramma 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 production is to indicate to the parser when it should stop parsing and announce acceptance of the input.
Consider the following grammar:

E -> E+T | T

T -> T*F | F

F -> (E) | id

The augmented grammar is

$E' \to E$

$E \to E+T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

LR(0) items

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus production $A \to XYZ$ yields the four items

$A \to .XYZ$, $A \to X.YZ$, $A \to XY.Z$, $A \to XYZ.$

A production $A \to \epsilon$ generate only one item $A \to .$

If the length of the right side of the production is 'n', then there are (n+1) items can be generated.

The items that can be generated using the above productions are $T \to T*F$ are

$T \to .T*F$, $T \to T.*F$, $T \to T*.F$, $T \to T*F.$

Canonical collection

A set of items that corresponds to the states of DFA that recognizes viable prefixes is called canonical collection.

Construction of a DFA involves finding canonical collections. Canonical collection provide basis for SLR parsers. To construct canonical collection make use of the two functions.

1. Closure
2. goto.

Closure Operation.

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, every item in I is added 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 I, if it is not already there. We apply this rule until no more new items can be added to closure(I).

✓

In our example closure($E' \rightarrow .E$) =

$$\begin{aligned} \{ & E' \rightarrow .E, \\ & E \rightarrow .E+T, \\ & E \rightarrow .T, \\ & T \rightarrow .T*F, \\ & T \rightarrow .F, \\ & F \rightarrow .(E), \\ & F \rightarrow .id \} \end{aligned}$$

The goto operation.

goto(I,X) where I is set of items and X is grammar symbol, 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

In our example consider $I = \{ E' \rightarrow E., E \rightarrow E.+T \}$, then

$$\begin{aligned} goto[I,+] = \{ & E \rightarrow E+.T, \\ & T \rightarrow .T*F, \\ & T \rightarrow .F, \\ & F \rightarrow .(E), \\ & F \rightarrow .id \} \end{aligned}$$

Now we can construct the canonical collection of sets of LR(0) items for an augmented grammar G',

The algorithm for canonical collection is:

algorithm for canonical collection is :

```
procedure items (G')
begin
  C := closure ( $S' \rightarrow .S$ )
  repeat
    for each set of items I in C and each grammar
    symbol X, such that goto (I, X) is not empty, an
    not in C do
        add goto (I, X) to C.
  until no more set of items can be added to C.
end.
```

Eg:- $E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$.

Its augmented grammar is :

$E' \rightarrow .E$

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E)/id$.

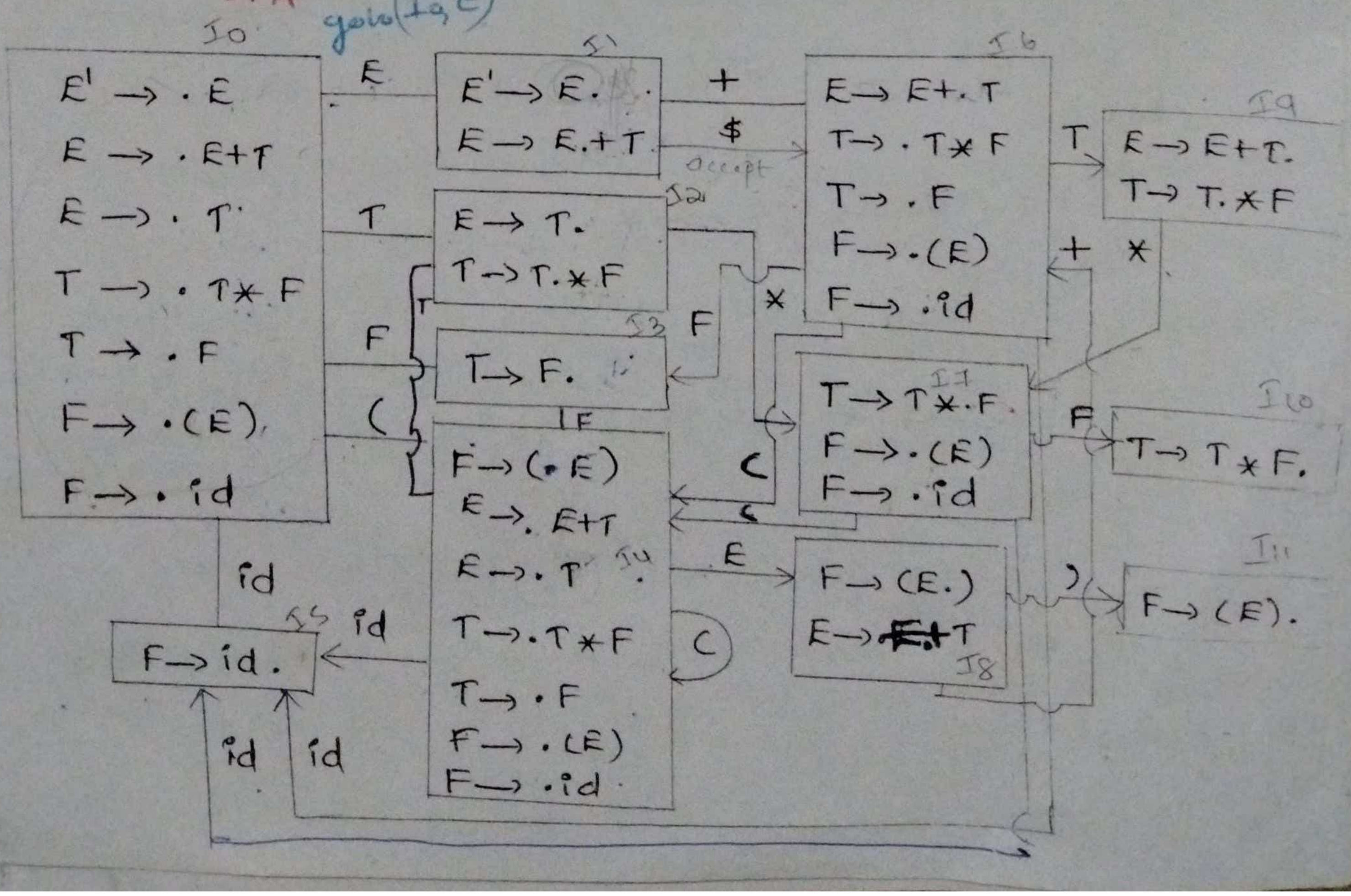
reductn no.

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

Now, closure of $E' \rightarrow .E$ = {
$E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$ } I_0

So,

DFA



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 $C = \{ I_0, I_1, I_2, \dots I_n \}$, the collection of sets of LR(0) items for G'.
2. State i is constructucted from I_i. The parsing actions for state i is 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 $[A \to \alpha .]$ 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 $[S' \to S.]$ is in I_i, then set action[i, \$] to 'accept'.

   if any conflicting actions are generated by the above rules, we say the grammar is not SLR(1).
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. The initial state of the parser is the one constructed from the set of items containing $[S' \to .S]$.

SLR(1) parse table for our example grammar is as shown below.

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

CANONICAL LR (CLR or LR).

In SLR, the FOLLOW set may contain more elements than actual lookahead (which can be drawback)

. In CLR, the lookahead symbol is calculated based on the pdn. Every pdn has a lookahead associated with it.

- CLR requires more memory consumption as it generates more no. of states as the lookahead symbols are associated with the pdn. And for different states lookahead symbols will vary

- The extra info of lookahead is incorporated into the state by redefining items to include a terminal symbol as a second component

The general form of an item becomes

$$\boxed{A \rightarrow \alpha \cdot \beta, a}$$

where $A \rightarrow \alpha\beta$ is a pdn, and a is a terminal or the right endmarker \$.

Such an object is called an LR(1) item. The 1 refers to the length of the 2nd component, called the lookahead of the item.

. The lookahead has no effect in an item of the form $A \rightarrow \alpha \cdot \beta, a$ where β is not ϵ.

But an item of the form $A \rightarrow \alpha \cdot, a$ calls for reduction by $A \rightarrow \alpha$ only if the next i/p symbol is a.

Constructing LR(1) set of Items:

- The method for building the collection of set of valid LR(1) items is essentially the same as that of canonical collection of sets of LR(0) items. The two procedures CLOSURE and GOTO has to be modified.

```
SetOfItems CLOSURE (I)
{
  repeat
    for (each item [$A \rightarrow \alpha \cdot B\beta, a$] in I ]
      for (each pdn $B \rightarrow \gamma$ in G')
        for (each terminal b in FIRST ($\beta a$))
          add [$B \rightarrow \cdot \gamma, b$] to set I;
  until no more items ~~are~~ added to I;
  return I;
}
```

```
SetOfItems GOTO (I, X)
{
  initialize J to the empty set;
  for (each item [$A \rightarrow \alpha \cdot X\beta, a$] in I )
    add item [$A \rightarrow \alpha X \cdot \beta, a$] to set J;
  return CLOSURE (J);
}
```

```
void items (G')
{
  initialise C to {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;
}
```

eg: Suppose given $S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

<u>step 1</u>: Create Augmented grammar

$S' \rightarrow \cdot S$

$S \rightarrow \cdot CC$

$C \rightarrow \cdot cC$

$C \rightarrow d$

In canonical LR, along with the pdn, lookahead symbol is passed.

<u>step 2</u>: Compute Closure:

We begin by computing the closure of $S' \rightarrow \cdot S, \$$ here \$ is the lookahead symbol as it is the accepting state.

Consider the algm for CLOSURE (I)

here I contains $S' \rightarrow \cdot S, \$$, which is similar to $A \rightarrow \alpha \cdot B\beta, a$

Here, $A = S'$

$\alpha = \varepsilon$

$B = S$

$\beta = \varepsilon$ and

$a = \$$

Function closure tells to add $B \rightarrow \cdot \gamma, b$, for each pdn $B \rightarrow \gamma$ and terminal b in FIRST (βa), Thus, must add $S \rightarrow CC$ and here b is \$ $\because \beta = \varepsilon$ and $a = \$$, b may be only \$

Illy compute closure for all items.

I_0 : $S' \rightarrow \cdot S, \$$

$S \rightarrow \cdot CC, \$$

$$\left.\begin{array}{l} C \rightarrow \cdot cC, c/d \\ C \rightarrow \cdot d, c/d \end{array}\right\} \Rightarrow \begin{array}{l} C \rightarrow \cdot cC, c \\ C \rightarrow \cdot cC, d \\ C \rightarrow \cdot d, c \\ C \rightarrow \cdot d, d \end{array}$$

$B\beta, a$
$\beta = c$
$c, \$ = \$$
$(\varepsilon, a = \$) = \$$

Step 3 :

GOTO (I_0, S) $\Rightarrow S' \rightarrow S \cdot, \$ = I_1$

for GOTO (I_0, C) $\Rightarrow S \rightarrow C \cdot C, \$$

Now take CLOSURE

$C \rightarrow \cdot cC, \$$

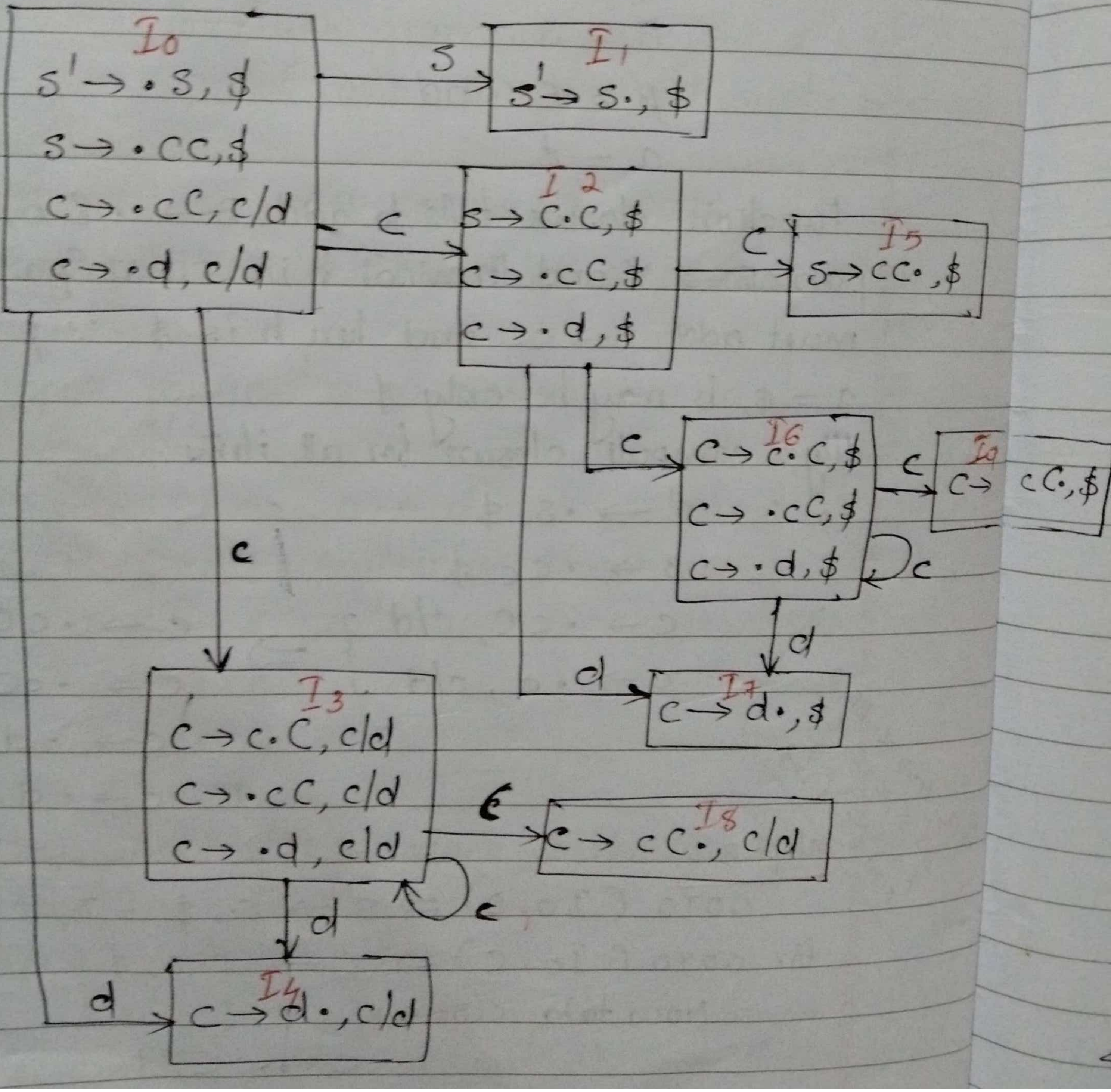
$C \rightarrow \cdot d, \$$

ie, I_2 : $S \to C \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

Illy carry out for the remaining states, and generate DFA or LR(1) automaton.



CANONICAL LR(1) PARSING TABLE CONSTRUCTION

LR(1) ACTION and GOTO fns are constructed from LR(1) items. The fns are represented by a table.

ALGORITHM: Constn of LR(1) Parsing Table

INPUT: Augmented grammar, G'

OUTPUT: Canonical LR parsing table fns ACTION and GOTO for G'

Method:

1. Construct $C' = \{I_0, I_1, \dots I_n\}$, the collection of sets of LR(1) items for G'.
2. state i of the parser is constructed from I_i. The parsing faction for state i is determined as follows.
   a) If $[A \rightarrow \alpha \cdot a\beta, b]$ 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 $[S' \rightarrow S\cdot, \$]$ is in I_i, then set ACTION$[i, \$]$ to "accept".

   If any conflicting actions from the above rules, we say, the grammar is not LR(1).
3. The GOTO transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) =$ 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 state set of items containing [S' → •S, $]

So the table formed for the grammar

1. S → CC
2 C → cC
3 C → d is given below

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		

LALR (LOOKAHEAD LR) PARSER.

Memory consumption is very high for CLR as the no. of states generated is more. (because of lookaheads included in each pdn)

· This can be reduced, if same states are merged which are having same items with same or different lookaheads.

LR parsers using LALR parsing table is called LALR parser. The table obtained by it are considerably smaller than CLR tables.

- The SLR and LALR tables of a grammar always have the same no. of states

- Consider the example discussed in the previous section.

when we look at the states I_4 and I_7, we'll find that the I^{st} component of the items (core) are same but the lookaheads are different (c/d for I_4 and \$ for I_7).

~~Since~~

Similarly I_3 and I_6 are form another pair with core

$\{ C \rightarrow c.C,\ C \rightarrow .cC,\ C \rightarrow d. \}$

One more pair I_8 and I_9 is common with

$\{ C \rightarrow cC. \}$

LALR Parsing table:

ALGORITHM:

I/P: An augmented grammar G'

O/P: LALR parsing table fns ACTION & GOTO for G'

Method:

1. construct $C = \{I_0, I_1 \cdots I_n\}$ the collection set of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replaces these sets by their union.
3. let $C' = \{J_0, J_1, \cdots J_m\}$ be the resulting sets of LR(1) items. The parsing action for state i are constructed from J_i in the same manner as canonical LR parsing table algm. If there is a parsing action conflict, the algm 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, ie, $J = I_1 \cup I_2 \cup \cdots I_K$, then the core of GOTO (I_1, X), GOTO $(I_2, X) \cdots$ GOTO(I_K, X) are same. since $I_1, I_2 \cdots I_K$ have the same core. let K be union of all sets of items having the same core as GOTO (I_1, X). Then GOTO $(J, X) = K$

- The collection of sets of items constructed in step 3 is called LALR (1) collections.

eg: Consider the grammar $S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC / d$

and canonical parsing table.

Here I_3 and I_6 have the same core. So we can merge these to two states

I_{36} : $C \rightarrow c \cdot C,\ c/d|\$$

$C \rightarrow \cdot cC,\ c/d|\$$

$C \rightarrow \cdot d,\ c/d|\$$

Illy I_4 and I_7 are replaced by their union

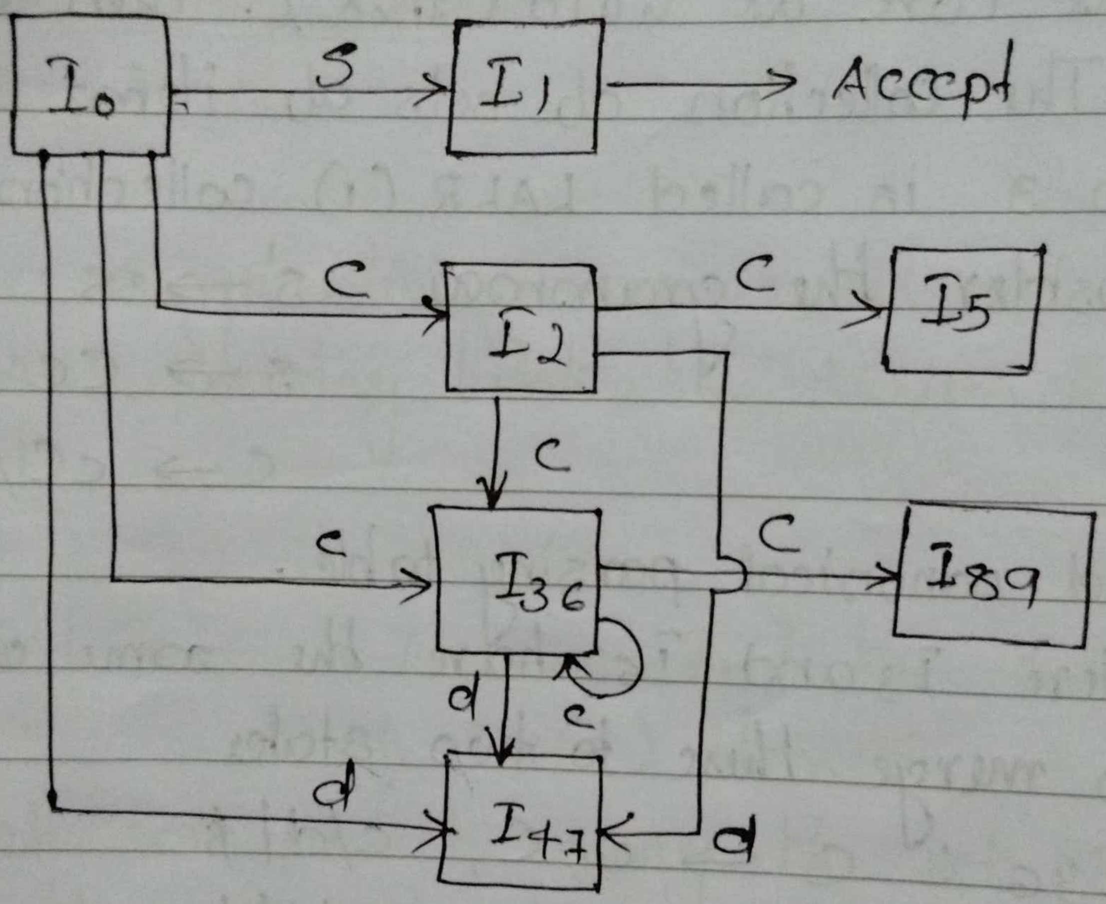
I_{47} : $C \rightarrow d \cdot,\ c/d|\$$

and I_8 and I_9 are replaced by their union

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

The LALR parsing table is shown below:

	ACTION			GOTO	
state	c	d	\$	S	C
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		


I0
S
I1
Accept
c
I2
c
I5
c
c
I36
c
I89
d
c
d
I47
d