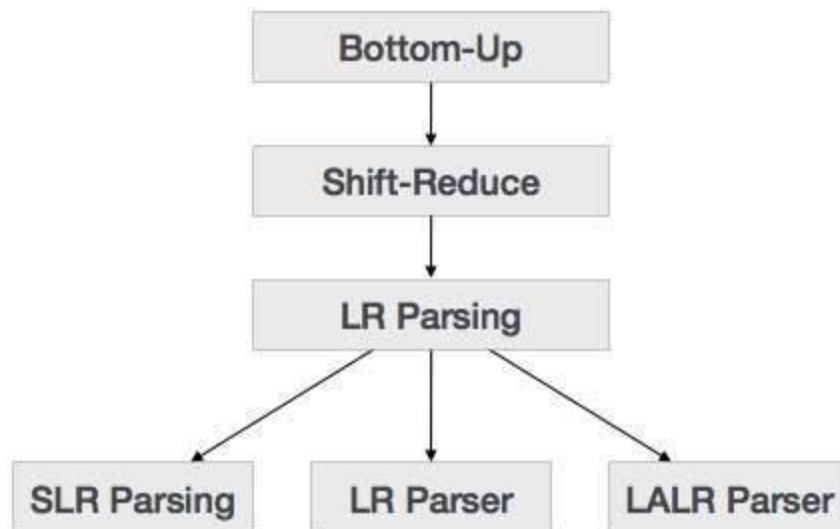


1. finit automata of LR(0) items and LR(0) Parsing
2. SLR(1) Parsing
3. LR(1) and LALR(1) Parsing
4. Yacc: An LALR(1) Parser Generator
5. Generation of TINY Parser Using Yacc
6. Error Recovery in bottom up parser.

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step :** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

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

- SLR(1) – Simple LR Parser:
 - Works on smallest class of grammar
 - Few number of states, hence very small table
 - Simple and fast construction
- LR(1) – LR Parser:
 - Works on complete set of LR(1) Grammar
 - Generates large table and large number of states
 - Slow construction
- LALR(1) – Look-Ahead LR Parser:
 - Works on intermediate size of grammar
 - Number of states are same as in SLR(1)

LL(1) Parsing:

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar. Then, for each nonterminal A,

1. Create an initial and final (return) state.
2. For each production $A \rightarrow X_1 X_2 \dots X_k$, create a path from the initial to the final state, with edges labeled X_1, X_2, \dots, X_k . If $A \rightarrow \epsilon$, the path is an edge labeled ϵ .

Transition diagrams for predictive parsers differ from those for lexical analyzers. Parsers have one diagram for each nonterminal. The labels of edges can be tokens or nonterminals. A transition on a token (terminal) means that we take that transition if that token is the next input symbol. A transition on a nonterminal A is a call of the procedure for A.

The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

$$\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \cap \text{First}(\alpha_3) \cap \dots \cap \text{First}(\alpha_n) = \emptyset$$

2. At most one of α and β can derive the empty string. *

$A \rightarrow \alpha \mid \beta$ one of them can be ϵ

3. If $\beta \Rightarrow^* \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$. Likewise, if $\alpha \Rightarrow^* \epsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

$A \rightarrow \alpha \mid \epsilon$ $\text{first}(\alpha) \cap \text{Follow}(A) = \varnothing$

Grammar

$S \rightarrow i E t S S' \mid a$

$S' \rightarrow e S \mid \epsilon$

$E \rightarrow b$

$\text{First}(S) = \{i, a\}$

$\text{Follow}(S) = \{\$, e\}$

$\text{First}(S') = \{e, \epsilon\}$

$\text{Follow}(S') = \{\$, e\}$

$\text{First}(E) = \{b\}$

$\text{Follow}(E) = \{b\}$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

For S' at e Note that the choice $S' \rightarrow \epsilon$ would prevent e from ever being put on the stack or removed from the input, and is surely wrong. 0

Algorithm depends on keeping track of the current state in the DFA of sets of items, we must modify the parsing stack to store not only symbols but also state numbers. we do this by pushing the new state number onto the parsing stack after each push of a symbol.

To begin a parse , we push the bottom marker \$ and the start states 0 onto the stack, so that at the beginning of the parse, the situation can be represented as

Parsing stack	Input
\$0	InputString \$

A grammar is said to be an LR(0) grammar if the above rules are unambiguous. This means that if a state contains a complete item $A \rightarrow a$, then it can contain no other items. Indeed , if such a state contains a “shift” item $A \rightarrow a.XB$ (x is terminal), then an ambiguity arises as to whether action(1) or action (2) is to be performed. This situation is called a shift-reduce conflict. Similarly, if such a stack contains another complete item $B \rightarrow b$. then an ambiguity arises as to which production to use for the reduction ($A \rightarrow a$ or $B \rightarrow b$). This situation is called a reduce –reduce conflict. Thus, a grammar is LR(0) if and only if each state is a shift state(a state containing only “shift” items) or a reduce state containing a single complete item.

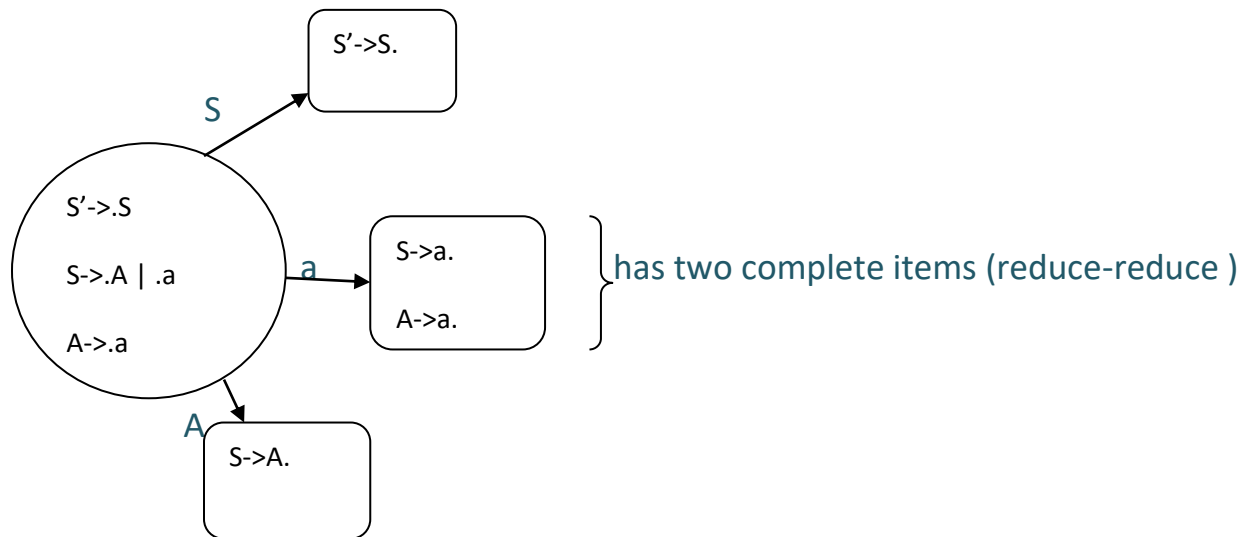
1.

Grammar

$S \rightarrow A \mid a$

$A \rightarrow a$

Check whether a given grammar is LL(1) ,LR(0) and SLR(1) or not?



For any bottom up parser grammar should not ambiguous.

In LR(0) It will conflict with (reduce- reduce) problems.like which one it will reduce like $S \rightarrow a \cdot$ or $A \rightarrow a \cdot$.

In SLR(1) it will check if $\text{follow}(S) \cap \text{follow}(A)$ should be empty otherwise it is not a SLR(1) parser.

$\text{Follow}(S) = \text{Follow}(A) = \{\$ \}$ so $\text{follow}(S) \cap \text{follow}(A)$ is not empty so it is also not a SLR(1) grammar.

In LL(1) grammar two production should not go into same cell.

$\text{First}(S) = \text{First}(A) = \{a\}$ mean production of S and production of A is going to the same so it is not LL(1) grammar also.

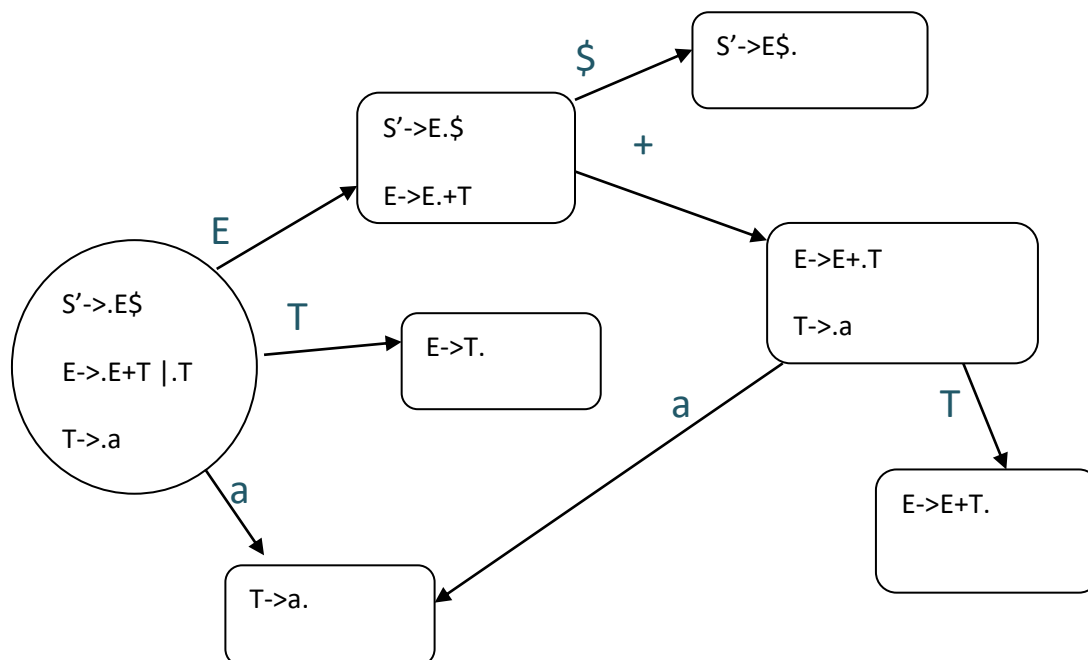
2.

Grammar

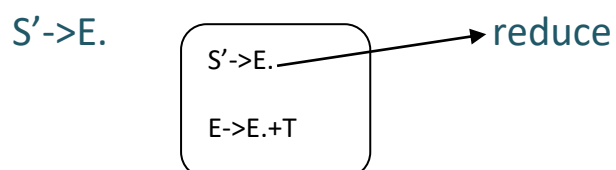
$E \rightarrow E+T \mid T$


$T \rightarrow a$

Check whether a grammar is LR(0) and SLR(1) or not?



In this grammar



$E \rightarrow E.T$  shift

Is not a shift reduce conflict because $S' \rightarrow E$ is augmented part means it is added by the user so it is not a shift reduce problem .so there is not any conflicts in this grammar so it is LR(0) and SLR(1) grammar.

3.

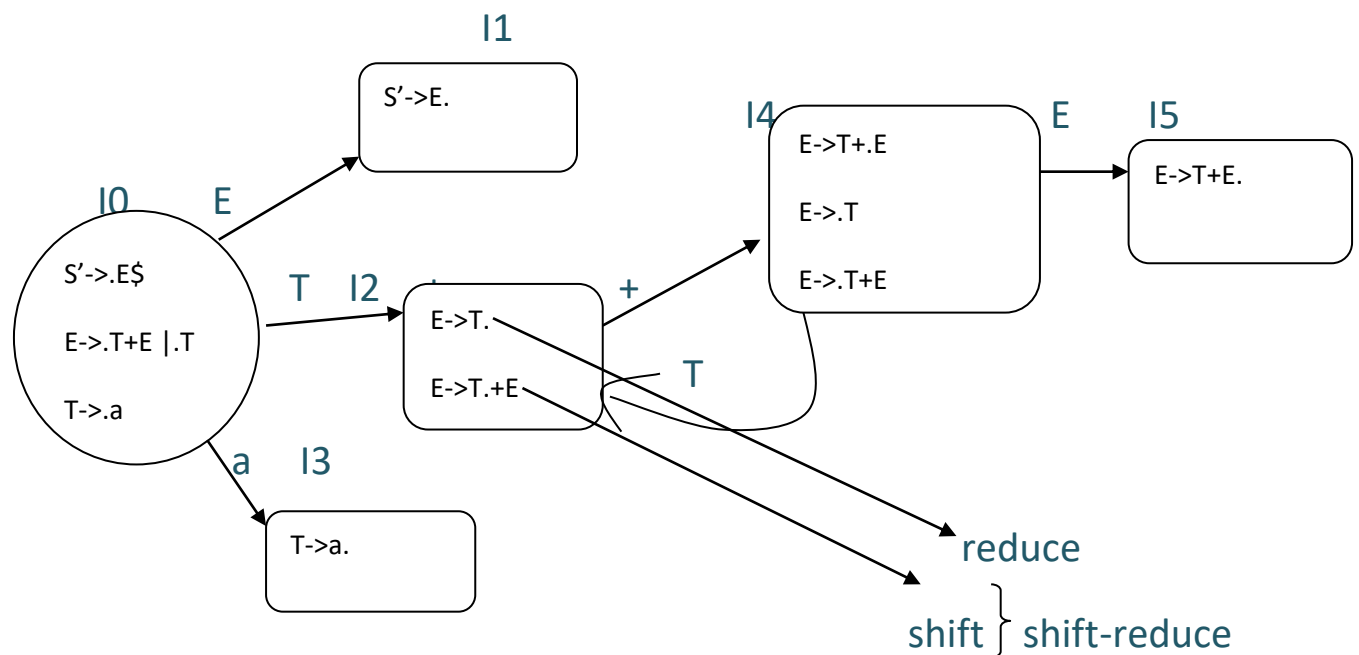
Grammar

$E \rightarrow T + E \quad (1) \mid T \quad (2)$

$T \rightarrow a \quad (3)$

Check whether a grammar is LR(0) and SLR(1) or not?

grammar is LR(0) and SLR(1) or not?



Because it has shift –reduce conflicts so it is not LR(0) parser.

In SLR(1) we have to check

$\text{follow}(E) = \{+, \$\}$

$\text{follow}(T) = \{\$, \}$

after putting the reduce move into follow of the left handside production .In given example the follow(T) and follow(E) symbols then we will checke if there is conflicts or not.

	a	+	\$	E	T
0	S3			1	2
1			ACCEPT		
2		S4	R2		
3		R3	R3		
4	S3			5	2
5			R1		

LR(0) Parser:

It is called LR(0) parser because it is not depending on the look ahead

It is used the shift and reduce in the parsing table.

Where we make the table using terminals and non-terminals (also called variable) .

Using terminals we do action and goto for the variables.

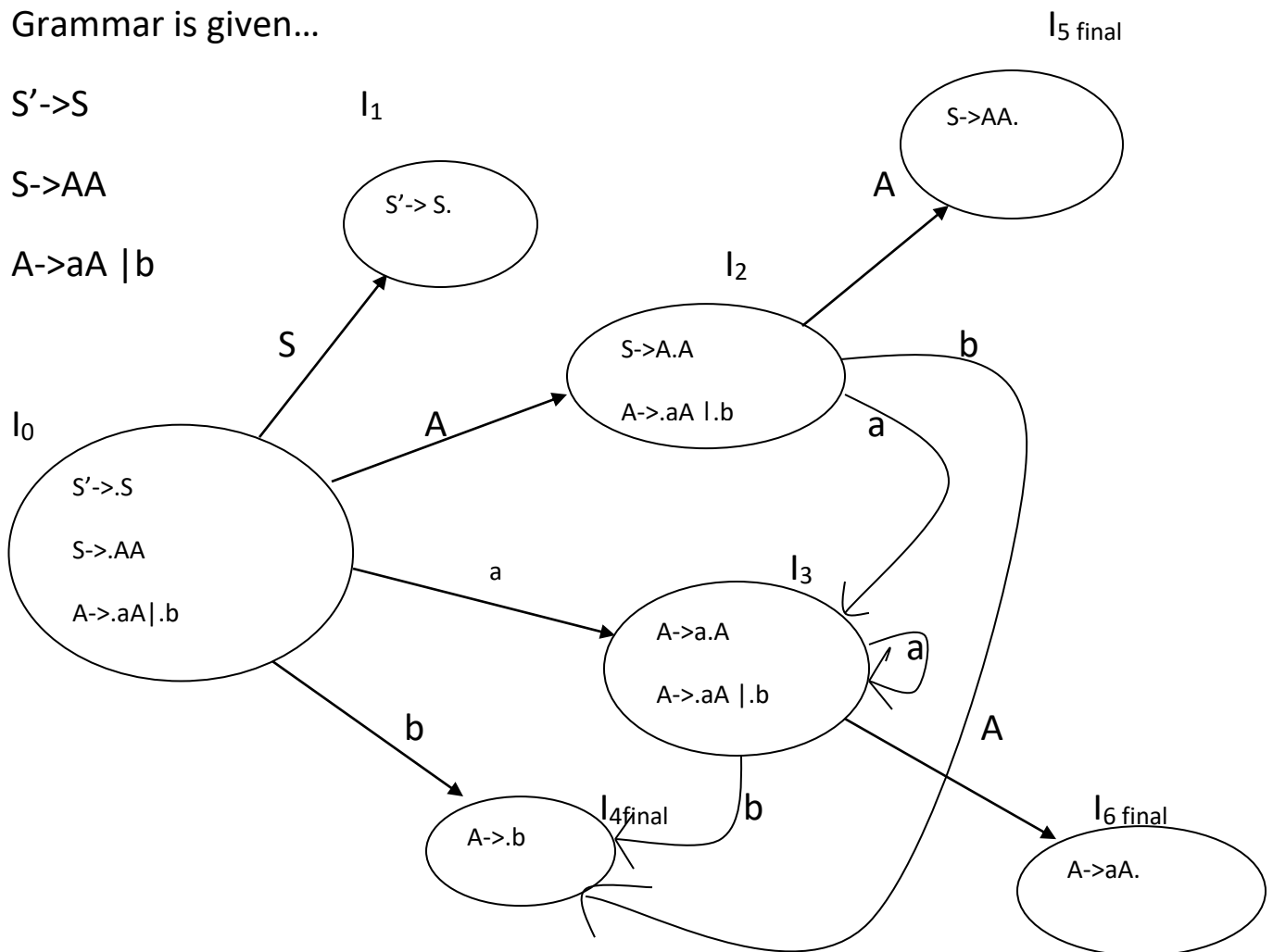
We have two state final and non-final state. In final state we do reduce and using non-final state we use shift.

Grammar is given...

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$



Augumented production which have the item in the right hand side of variable.

In above diagram I_4 I_5 and I_6 are the final item which is also called the augumented production .

	a	b	\$	A	S
0	S3	S4		2	1
1	ACCEPT
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

.

Action			Goto		
	a	b	\$	A	S
0	S3	S4		2	1
1	ACCEPT
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Input string is String="aabb"

$S' \rightarrow S$ I_1

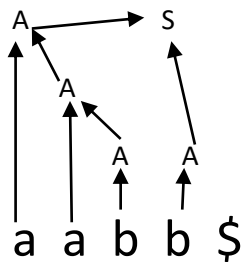
$S \rightarrow AA$ (1)

$A \rightarrow aA$ (2) | b (3)

0	a	3	A	3	B	4	A	6	A
	3	3	2	2	1	1	2	2	3
6	A	2	B	4	A	5	S	1->	accept
3	5	5	4	4	5	5			

In shift part we shift (S_n) the terminal and the position of terminals in given table and then we increment the pointer in the string.

In reduce part we reduce or pop the production on the right hand side of given production in the table (R_n) and the push the non-terminal or variable or left part of production into the stack. In reduce part we don't increment the position of the pointer in given string.



LR Parsing Algorithm

Here we describe a skeleton algorithm of an LR parser:

```
token = next_token()

repeat forever
    s = top of stack

    if action[s, token] = "shift si" then
        PUSH token
        PUSH si
        token = next_token()

    else if action[s, token] = "reduce A ::=  $\beta$ " then
        POP 2 *  $|\beta|$  symbols
        s = top of stack
        PUSH A
        PUSH goto[s, A]

    else if action[s, token] = "accept" then
        return

    else
        error()
```

SLR(1) Parser:

It increases the power of LR(0) parsing significantly, however by using the next token in the input string to direct its actions.

First, it consults the input token before a shift to make sure that an appropriate DFA transition exists.

Second, it uses the Follow set of a nonterminal, as constructed in to decide if a reduction should be performed.

THE SLR(1) parsing algorithm.

1. If state s contains any item of the form $A \rightarrow a.Xb$, where X is a terminal. And X is the next token in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item $A \rightarrow aX.b$
2. If state s contains the complete item $A \rightarrow y.$, and the next token in the input string is in $\text{Follow}(A)$, then the action is to reduce by the rule $A \rightarrow y$. A reduction by the rule $S' \rightarrow S$, where S is the start state, is equivalent to acceptance; this will happen only if the next input token is $\$$. In all other cases, the new state is computed as follows. Remove the string y and all of its corresponding states from the parsing stack.
3. Indeed, the Follow set for the augmented start state S' of any grammar is always the set consisting only of $\$$, since S' appears only in the grammar rule $S' \rightarrow S$.

Statement \rightarrow if-stmt | other

If-stmt \rightarrow if (exp) statement | if (exp) statement else statement

exp \rightarrow 0 | 1

S \rightarrow I | other

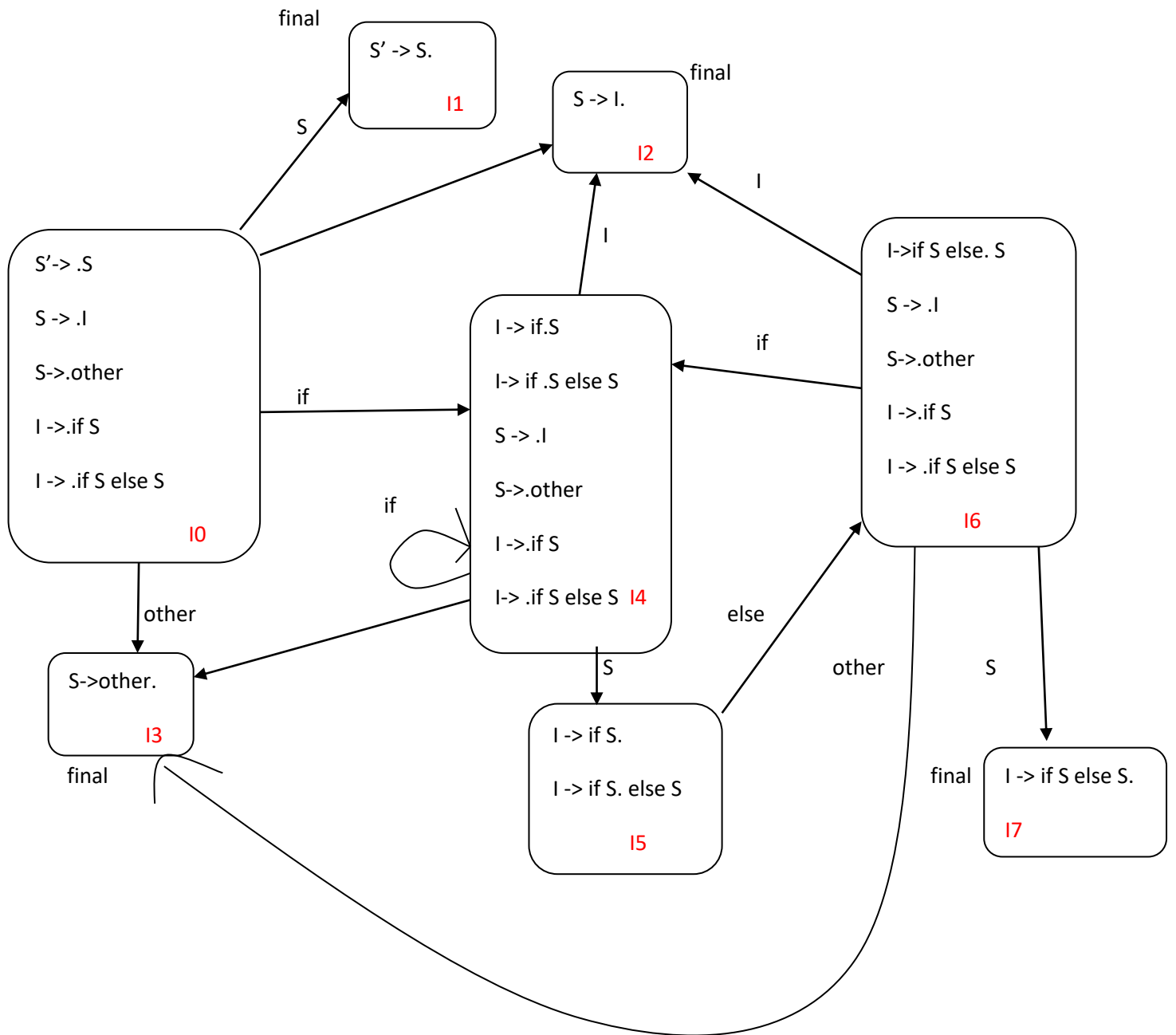
I \rightarrow if S | if S else S

$\text{Follow}(I) = \text{Follow}(I) = \{\$, \text{else}\}$

In SLR(1) we put the reduce position which have the follow of left symbol of production rule.

If state s contains any item of the form $A \rightarrow a.Xb$, where X is a terminal. pushed on the stack is the state containing the item $A \rightarrow aX.b$

If state s contains the complete item $A \rightarrow y.$, and the next token in the input string is in $\text{Follow}(A)$.



	Input				Goto	
	if	else	other	\$	S	I
0	S4		S3		1	2
1	ACCEPT
2		R1		R1		
3		R2		R2		
4	S4		S3		5	2
5		S6		R3		
6	S4		S3		7	2
7		R4		R4		

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. We shall have more to say about how such error recovery routines work in Sections 4.8.3 and 4.9A.

2. We extend the **GOTO** function, defined on sets of items, to states: if **GOTO** $[li, A] = lj$, then **GOTO** also maps a state i and a nonterminal A to state j .

LR(1) parser:

LALR(1) CLR(1)

LR(1) item=LR(0) items + lookahead

Examples:

Grammar:

$S \rightarrow AA$ (1)

$A \rightarrow aA$ (2)

$A \rightarrow b$ (3)

rules

$A \rightarrow \alpha.B\beta, a|b$ B is non-terminal

$B \rightarrow \cdot\lambda, \text{first}(\beta, a|b)$ $\text{first}(\beta, a|b) = \{c,d\}$

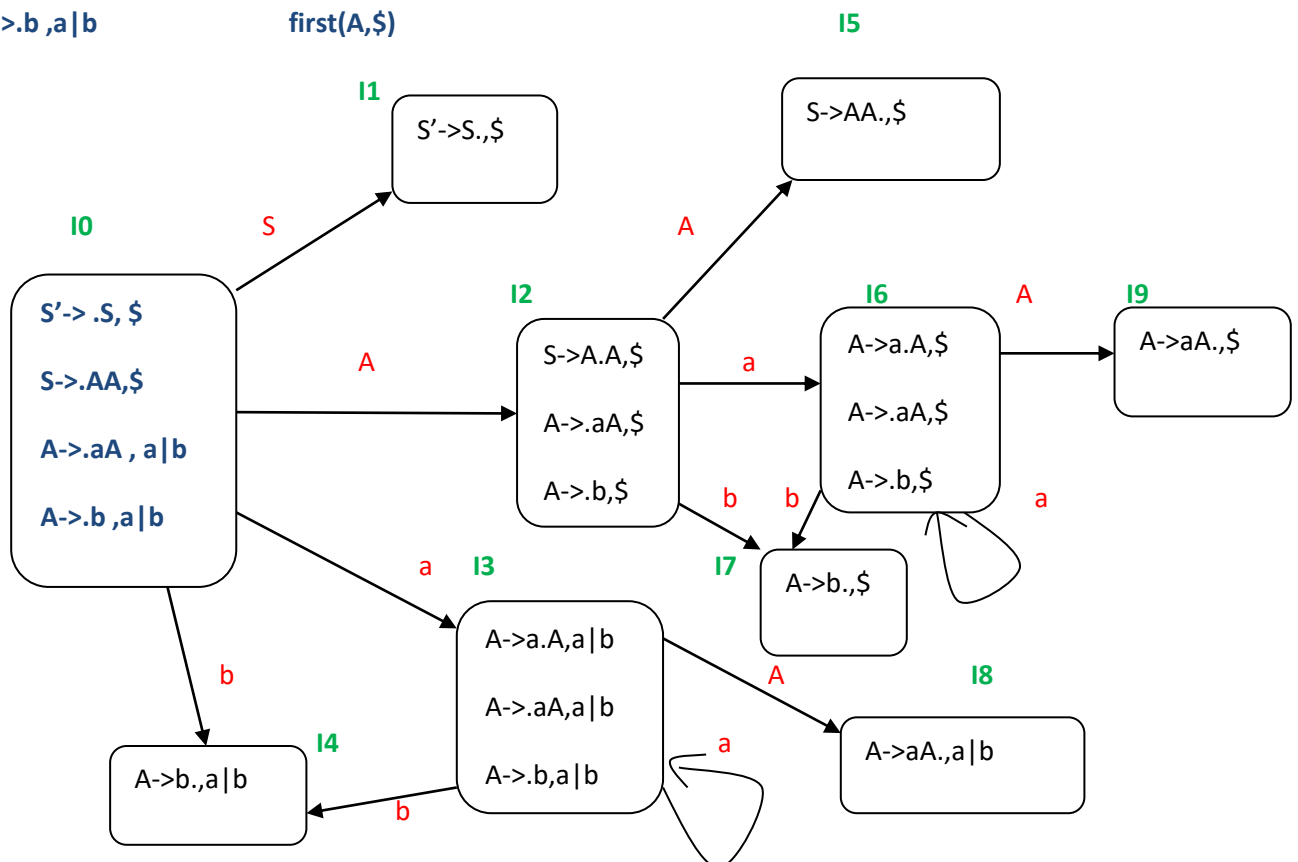
$B \rightarrow \cdot\lambda, c|d$ $\lambda \in (T \cup N)^*$

$S' \rightarrow \cdot S, \$$

$S \rightarrow \cdot AA, \$$

$A \rightarrow \cdot aA, a|b$ $\text{first}(A, \$)$

$A \rightarrow \cdot b, a|b$ $\text{first}(A, \$)$



CLR(1) parsing table:

position	ACTION	Goto
----------	--------	------

	a	b	\$	S	A
0	S3	S3		1	2
1			Accept		
2	S6	S7		5	
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

I3, I6 => I₃₆

I4, I7 => I₄₇

I8, I9 => I₈₉

CLR(1) >= LR(0) = SLR(1) = LALR(1)

In LALR(1) table we combine the state which have same production rules but different in lookahead.

So when we remove one row then we combine or take the union of both and put in one of them row and then remove the other row of same production state.

To convert the CLR(1) we have to write row 36=(36₁ union 36₂) 47=(47₁ union 47₂) and 89=(89₁ union 89₂).

S36 ,S47 and S89 are the state so R1,R2,R3 are the production number.so we only effect on the state not in production rule number.

position	ACTION	Goto
----------	--------	------

	a	b	\$	S	A
0	S36	S36		1	2
1			Accept		
2	S36	S47		5	
36	S36	S47			89
47	R3	R3			
5			R1		
36	S36	S47			89
47			R3		
89	R2	R2			
89			R2		

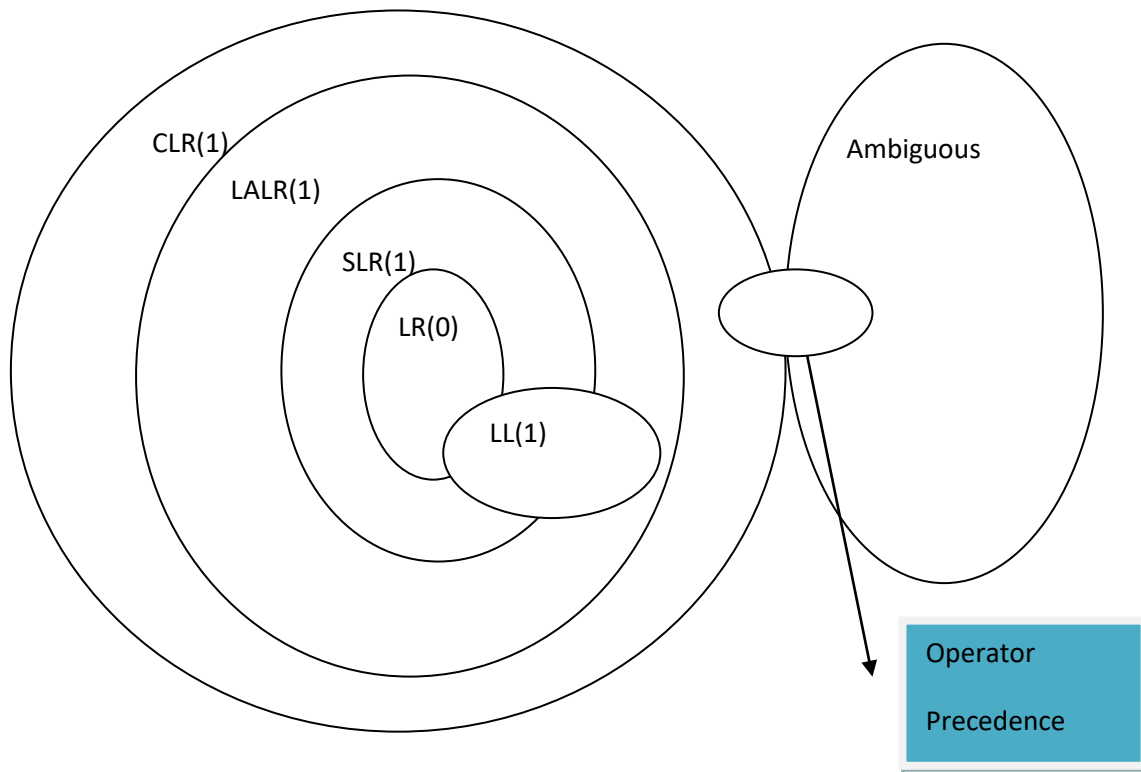
LALR(1) parsing table:

position	ACTION	Goto
----------	--------	------

	a	b	\$	S	A
0	S36	S36		1	2
1			Accept		
2	S36	S47		5	
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Diagram of grammar according to the power :

Unambiguous grammar...



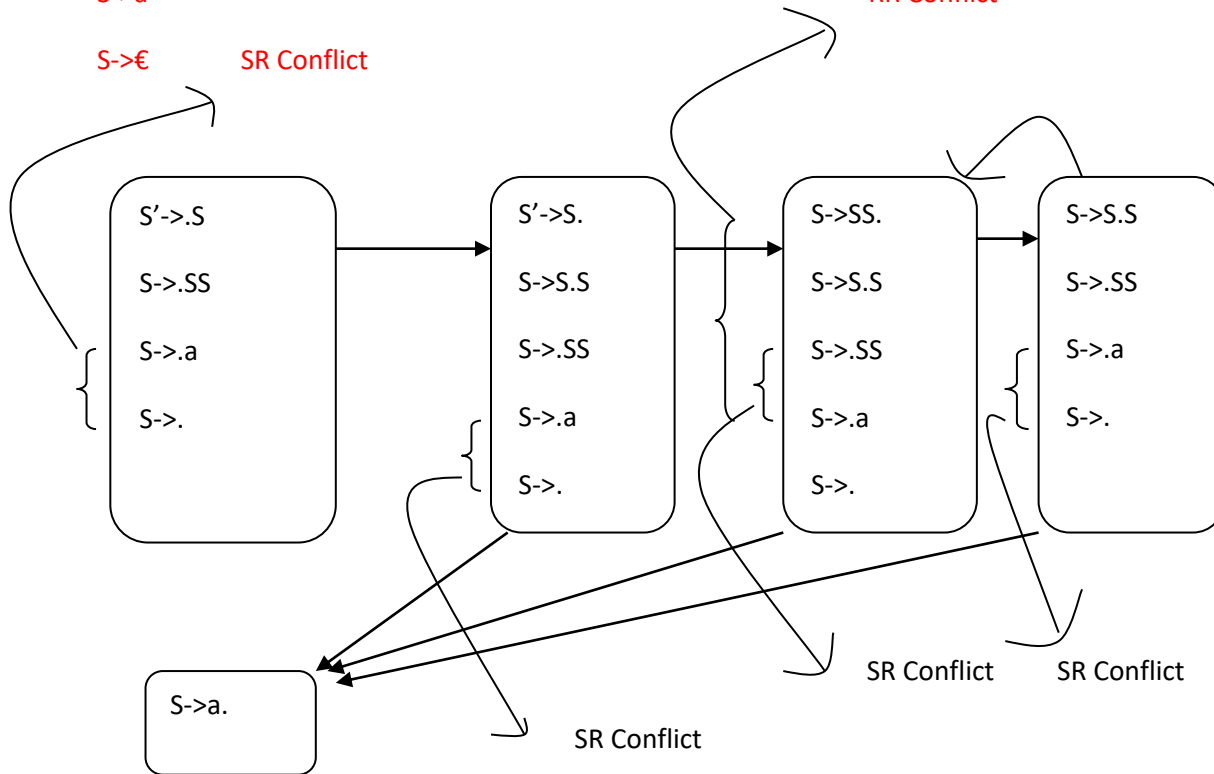
1. Find the number of SR and RR conflict in dfa with LR(0) items

$S \rightarrow SS$

$S \rightarrow a$

$S \rightarrow \epsilon$ SR Conflict

RR Conflict



4 SR Conflict and 1RR Conflict...

2.

$E \rightarrow E + n$

$E \rightarrow E * n$

$E \rightarrow n$

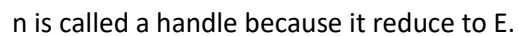
For the string $n+n+n$, the handles in right sentential form of reduction are...

$E \Rightarrow E * n$

$\Rightarrow E + n * n$

$\Rightarrow n + n + n$





Given grammar is

S->a

$$N(\text{LALR}(1)) = n^3$$

We also know that no of state in CLR(1) can greater than LALR(1) So we check by drawing the diagram of CLR(1).

