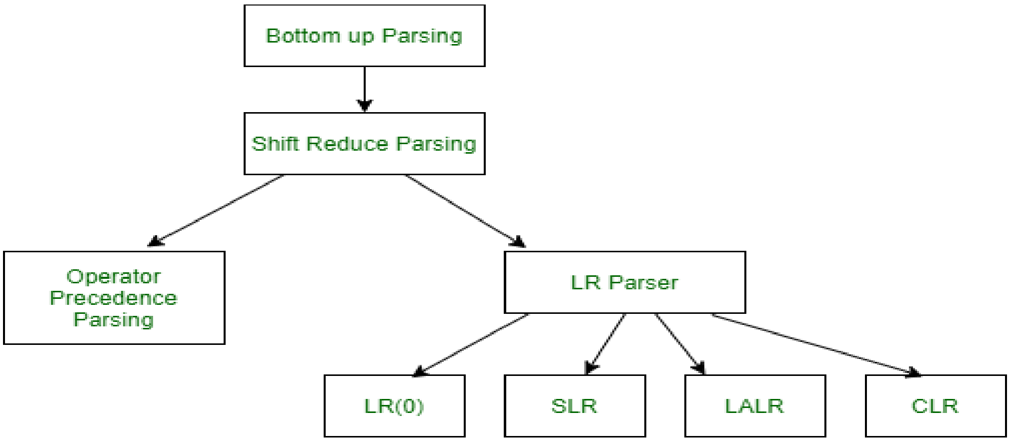
LR Parser

LR parsers are used to parse the large class of context free grammars used by computer programming language compiler and other associated tools. This technique is called LR(k) parsing.

- LR parsers can usually recognize all programming language construct that can be specified by context-free grammars.
- LR parsers detect errors fast.
- Drawback: it is too much work to construct an LR parser by hand.

It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.



• L is left-to-right scanning of the input.

• R is for constructing a right most derivation in reverse.

k is the number of input symbols of lookahead that are used in making parsing decisions.

LR parser consists of two parts, a driver routine and a parsing table.

The driver routine is same for all LR parsers only the parsing table changes from one parser to another.

The driver routine is simple to implement.

There are many different parsing table used in LR parser.

Some parsing table detect errors sooner than others.

Three different techniques for producing LR parsing tables are

• SLR(l) – Simple LR

Works on smallest class of grammar.

Few number of states, hence very small table.

Simple and fast construction.

Easy to implement

• LR(1) – LR parser

Also called as Canonical LR parser.

Generates large table and large number of states.

Slow construction.

Expensive to implement

• LALR(l) – Look ahead LR parser

Works on intermediate size of grammar.

Number of states are same as in SLR(l).

Works on most programming language

LR Parser

LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions

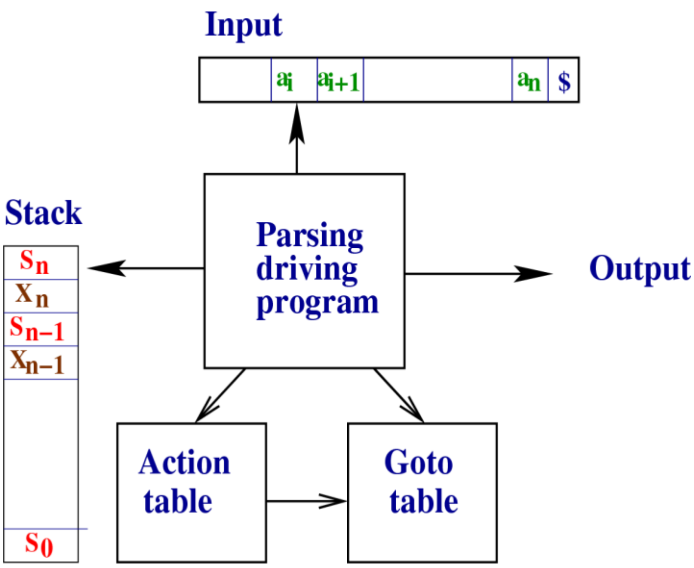
1. Action

2. Goto

The driver program is same for all LR parsers. Only the parsing table changes from one parser to another.

The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the information contained in the stack.

The stack holds a sequence of states, s_o, s_1, · ·· , S_m, where S_m is on the top.



Action This function takes as arguments a state *i* and a terminal *a* (or $, the input end marker). The value of ACTION [i, *a]* can have one of the four forms:

i) Shift *j*, where *j* is a state.

ii) Reduce by a grammar production A—> β.

iii) Accept.

iv) Error.

Goto This function takes a state and grammar symbol as arguments and produces a state.

If GOTO [I_i ,A] = I_j, the GOTO also maps a state *i* and non terminal A to state *j*.

Input: A LR-Parser for an unambiguous context-free grammar G over an alphabet Σ and a word $w \in \Sigma^*$.

Output: An error if $w \notin L(G)$ or a rightmost derivation for w otherwise.

Set the cursor to the rightmost symbol of $w\$$
push the initial state s_0 on top of the empty stack
repeat
  let s be the state on top of the stack
  let a be the current pointed symbol in $w\$$
  if action$[s, a]$ = shift s' **then**
    push a on top of the stack
    push s' on top of the stack
    advance the cursor to the next symbol on the right in $w\$$
  else if action$[s, a]$ = reduce $A \longmapsto \beta$ **then**
    pop 2 $\mid \beta \mid$ symbols of the stack
    let s' be the state on top of the stack
    push A on top of the stack
    push goto$[s', A]$ on top of the stack
    output $A \longmapsto \beta$
  else if action$[s, a]$ = accept **then**
    return
  else
    error

Canonical collection of LR(0) items

A production with a dot at some position on the right-hand side of the production is called the LR (0) item.

Example: The possible LR (0) items for a production A → BCD
A → • BCD
A → B • CD
A → BC • D
A → BCD
And for the production A → ∈, LR (0) item

At any point of the parsing process, LR (0) item indicates how much portion of a production we have seen.
For example the send production A → B • CD

Indicates that we have just seen the input string derivable from B and we next expect to see the string derivable from CD

A collection of sets of LR (0) items is called Canonical LR(0) collection which is used in the construction of SLR functions
To construct the canonical LR(0) collection for a grammar we need to define augmented grammar and two functions CLOSURE and GOTO

Augmented Grammar − It is a new Grammar G' which contains a new production $S' \to S$ with all other productions of given grammar G.

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.

• Initially, add every item I to *CLOSURE(I).*

• If $A \longrightarrow \alpha B, \beta$ is in *CLOSURE(I)* and $B \longrightarrow \gamma$ is a production, then add the item $B \longrightarrow \bullet\ \gamma$ to *CLOSURE(i),* if it is not already there. Apply this rule until no more items can be added to *CLOSURE (!).*

Constructing SLR parsing table

Steps to produce SLR Parsing Table

    - Generate Canonical set of LR (0) items
    - Compute FOLLOW as required by Rule (2b) of Parsing Table Algorithm.
- **Step1**− Construct the Augmented Grammar and number the productions
- (0) $E' \to E$
- (1) $E \to E + T$
- (2) $E \to T$
- (3) $T \to T * F$
- (4) $T \to F$
- (5) $F \to (E)$
- (6) $F \to id$
- **Step2**− Apply closure to the set of items & find goto
- Square Boxes represent the new states or items, and Circle represents the repeating items.

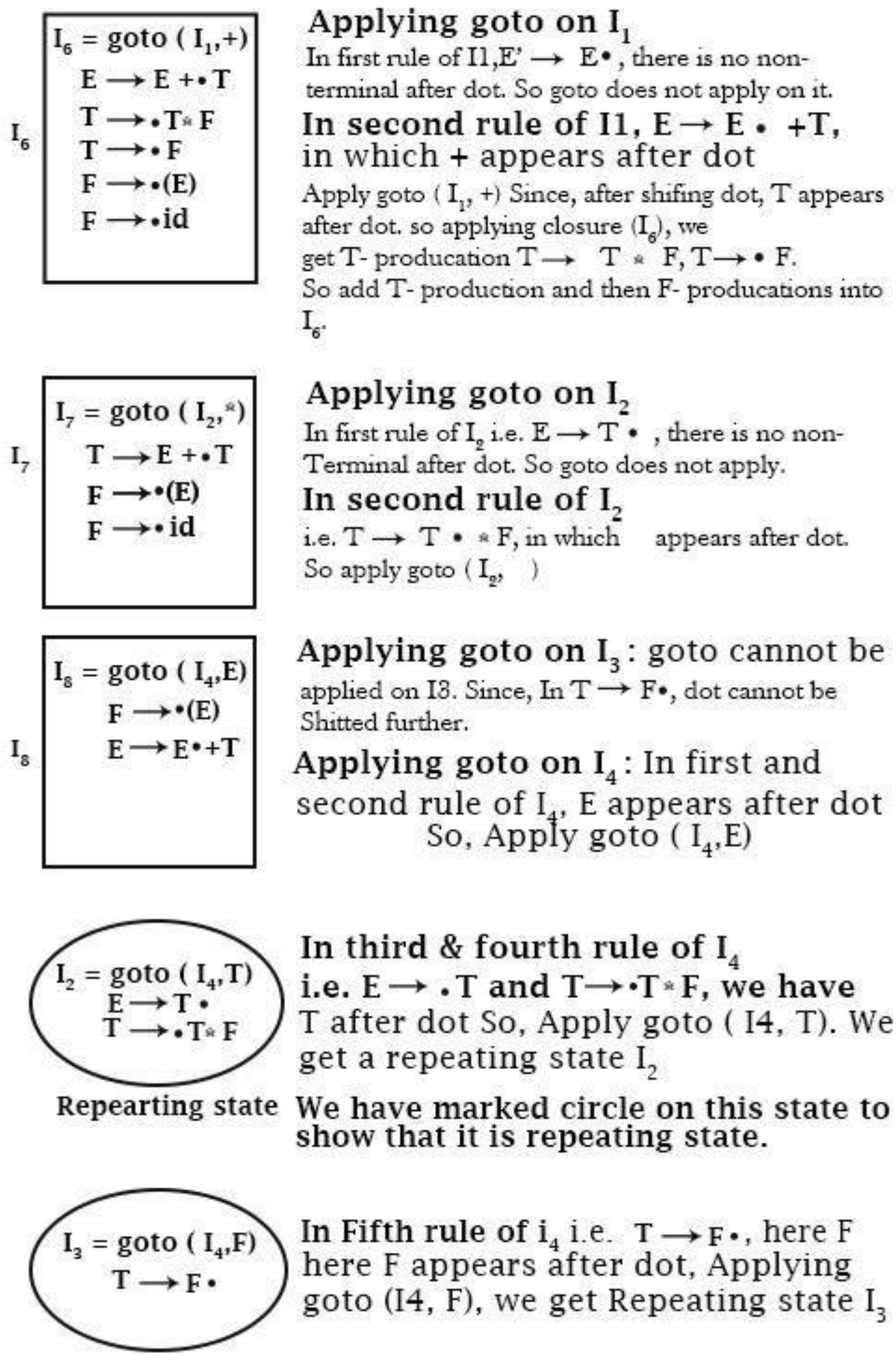
I_0	$\therefore$ Closure $(E' \to \cdot E)$ = $E' \to \cdot E$ $E \to \cdot E + T$ $E \to \cdot T$ $T \to \cdot T * F$ $T \to \cdot F$ $F \to \cdot (E)$ $F \to \cdot id$	Item I_0 is constructed starting from grammar $E' \to \cdot E$. As, there is non-terminal E after a dot, add all E – productions with a dot on the first position of the right-hand side. It means $E \to \cdot E + T$ and $E \to \cdot T$ will be added in I_0. But the rule $E \to \cdot T$ which we have added contains non-Terminal T immediately right to dot so we have to add T production, i.e., $T \to \cdot T *$ F and $T \to . F$ in I_0. Now, in production $T \to \cdot F$, F appears after the dot, So add F –productions also i.e., $F \to \cdot$ (E) and $F \to \cdot id$ into I_0. Next, we have to find item I_1 which will be derived from I_0.

-

I_1	I_1 = goto (I_0,E) $E' \to E\bullet$ $E \to E\bullet + T$	Applying goto on I_0 In first rule and second rule of I0 $E' \to E$ and $E \to \bullet\ E + T$. As E appears after dot. So, chose all those rules in I_0 in which E is after dt & shift the dot to right side. i.e. Apply goto (I_0, E)
I_2	I_2 = goto (I_0,T) $E' \to T\bullet$ $T \to T\bullet * F$	In third & fourth rule of I_0 in which T appears after dot i.e. $E \to \bullet\ T$, $T \to \bullet T*F$ Apply goto (I_0 T)
I_3	I_3 = goto (I_0,F) $T \to F\ \bullet$	In fifth rule i.e. $T \to \bullet F$ in I_0, F appears after dot Apply goto (I_0, F)
I_4	I_4 = goto (I_0,() $F \to (\bullet E)$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet\ T* F$ $T \to \bullet\ F$ $F \to (\bullet E)$ $F \to \bullet id$	In Sixth rule $F \to \bullet$ (E) here (appears after dot in I_0 Apply goto (I0, (). But after Shifting dot, we get non- terminal E after dot, so Apply closure on E, then on T, then on F.
I_5	I_3 = goto (I_0,id) $F \to id\bullet$	In last rule of I0 i.e. $F \to \bullet$ id here id appears after dot Apply goto (I_0, id)

-

- So, all rules of I_0have been completed by applying goto on I_0. Now, in the same manner apply goto on I_1, I_2 and then goes on.



-

I_4 = goto (I_4,( )

$F \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$

In sixth rule of I_4, $F \rightarrow \bullet (E)$,in

Which (appears after dot. Applying goto (I_4,( ).
We get again I_4 state

I_5 = goto (I_4,id)

$F \rightarrow id \bullet$

In Seventh rule of I_4, $F \rightarrow \bullet id$

in Which id appears after dot. On shifting dot we get repearting state I_5

I_9

I_9 = goto (I_6,T)

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

Applying goto on I_5

In I_5, we have rule $F \rightarrow id *$ As dot cannot be shifted further. So, goto cannot be applied

Applying goto on I_6

In first and second rule of I_6, $E \rightarrow E+ \bullet T$ and $T \rightarrow \bullet T * F$, in which T appears after dot So, applying goto (I_6, T), we get new state I_9

I_3 = goto (I_6,F)

$T \rightarrow F \bullet$

In third rule of $I_{6,}$

i.e. $T \rightarrow \bullet F$, F appears after dot.
Applying goto (I_6,F), we get repeating state I_3

I_4 = goto (I_6,( )

$F \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$

In Fourth rule of $I_{6,}$

i.e. $F \rightarrow \bullet (E)$, in which (appears after dot.
So, applying goto (I6, ( ), we get repeating state I_4

•

I_5 = goto (I_6, id)
F → id●

In fifth rule of I_6, i.e., F → ●id, in which id appears after dot.
So, applying goto (I_6, id) will give repeating state I_5.

I_{10}

I_{10} = goto (I_7, F)
T → T*F ●

Applying goto on I_7
In first rule of I_7 i.e., T → T * ● F, We have F after dot.
So applying goto (I_7, F), we get new state I_{10}.

I_4 = goto (I_7, ( )
F → (●E)
E → ●E+T
E → ●T
T → ●T*F
T → ● F
F → ●(E)
F → ● id

In second rule of I_7. F → ●(E) We have (after dot.
So applying goto (I_7, ( ) we get again same state I_4

-

I_5 = goto (I_7, id)
F → id●

In third rule of I_7, i.e., F → ●id, in which id appears after dot.
So, applying goto (I_7, id) will give repeating state I_5.

I_{11}

I_{11} = goto (I_8, ,))
F → (E)●

Applying goto on I_8
In first rule of I_8 i.e., F → (E ●), in which) appears after dot.
So applying goto (I_8,)), we get new state I_{11}.

I_6 = goto (I_8, +)
E → E + ●T
T → ●T * F
T → ●F
F → ●(E)
F → ●id

In second rule of I_8. E → E ● + T, in which + appears after dot.
So applying goto (I_8, +) we get again repeating state I_6

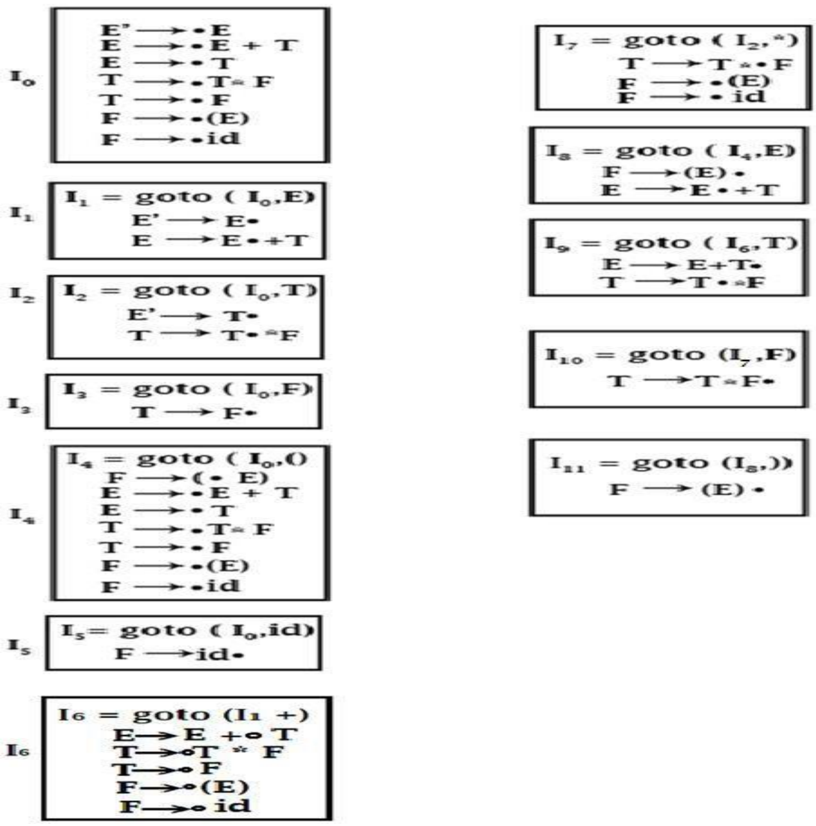
-

I_7= goto (I_9, *)
T → T * ●F
F → ●(E)
F → ● id

Applying goto on I_9
In first rule, dot is at the last position. So, goto does not apply.
In second rule, T → T ●* F, in which * appears after dot.
So, applying goto (I_9, *), we get repeating state I_7.

Applying goto on I_{10}, I_{11}. In T → T * F●, F → (E)● dot is at last position.
So goto does not apply. Following shows collection of items I_0 to I_{11}

-



-

LR Driver Program

The LR driver Program determines Sm, the state on top of the stack and ai , the Current Input symbol.

❖ It then consults Action[Sm, ai] which can take one of four values:

✓ Shift

✓ Reduce

✓ Accept

✓ Error

If Action[Sm, ai] = Shift S

✓ Where S is a State, then the Parser pushes ai and S on to the Stack.

❖If Action[Sm, ai] = Reduce A → β,

✓ Then ai and Sm are replaced by A

✓ if S was the state appearing below ai in the Stack, then GOTO[S, A] is consulted and the state pushed onto the stack

If Action[Sm, ai] = Accept,

✓ Parsing is completed

❖If Action[Sm, ai] = Error,

✓ The Parser discovered an Error.

GOTO Table

❖ The GOTO table specifies which state to put on top of the stack after a reduce

✓Rows are State Names;

✓Columns are Non-Terminals

The GOTO Table is indexed by a state of the parser and a Non Terminal (Grammar Symbol) ex : GOTO[S, A]

❖ The GOTO Table simply indicates what the next state of the parser if it has recognized a certain Non Terminal

To fill reduce state

Check in states I_0 to I_{11} whether . is at end. If .(dot) followed by non terminal find follow for the non terminal.

To Find Follow for E,T and F

Follow(E)={$,+,)}

Follow(T)={$,+,),*}

Follow(T)={$,+,),*}

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			

Table Construction

Put $ at the end of the string, i.e., id * id + id $.

Stack	Input String	Reason
0	id ∗ id + id	Action [0, id] = s5 ∴ Shift id and state 5

Stack	Input String	Reason
0 id 5	$*$ id + id $	Action [5,$*$] = r6. ∴ Reduce by F → id. goto(0, F) = 3
0 F 3	$*$ id + id $	Action [3,$*$] = r4, Reduce by T → F goto(0, T) = 2
0 T 2	$*$ id + id $	Action [2,$*$] = s7, shift $*$, 7
0T2*7	id + id $	Action [7, id] = s5, shift id, 5
0T2*7 id 5	+id $	Action [5, +] = r6, Reduce by F → id goto(7, F) = 10
0T2*7 F 10	+id $	Action [10, +] = r3S, Reduce by T → T $*$ F, goto(0, T) = 2
0 T 2	+id $	Action [2, +] = r2, Reduce by E → T goto(0, E) = 1
0 E 1	+id $	Action [1, +] = s6, Shift +, 6
0 E 1 + 6	id $	Action [6, id] = s5, Shift id, 5.
0 E 1 + 6 id 5	$	Action [5, $] = s6, Reduce by F → id, goto(6, F) = 3
0 E 1 + 6 F 3	$	Action [3, $] = r4, Reduce by T → F, goto(6, F) = 9
0 E 1 + 6 T 9	$	Action [9, $] = r1, Reduce by E → E + T, goto(0, E) = 1
0 E 1	$	Action [1, $] = accept

Constructing CLR parsing table

CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

LR(1) Parsing configurations have the general form:

A –> X1...Xi • Xi+1...Xj , a

❖ The Look Ahead Component ‘a’ represents a possible look-ahead after the entire right-hand side has been matched

❖ The Є appears as look-ahead only for the augmenting production because there is no lookahead after the end-marker

Steps for constructing CLR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found

3. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the CLR parsing table

Example:

Context Free Grammar: S → CC C → cC C → d

Augmented Grammar: S’ → • S$

S → •CC

C → •cC

C → •d

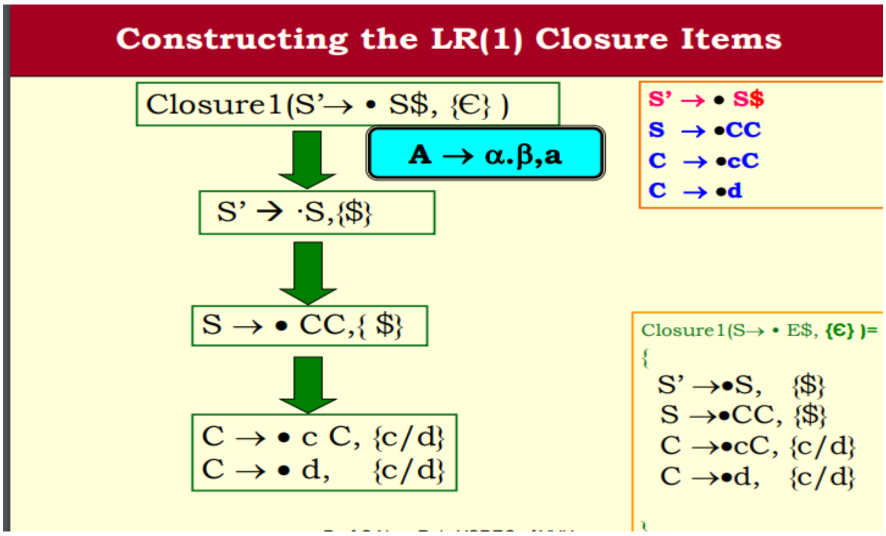
Constructing the LR(1) Closure Items

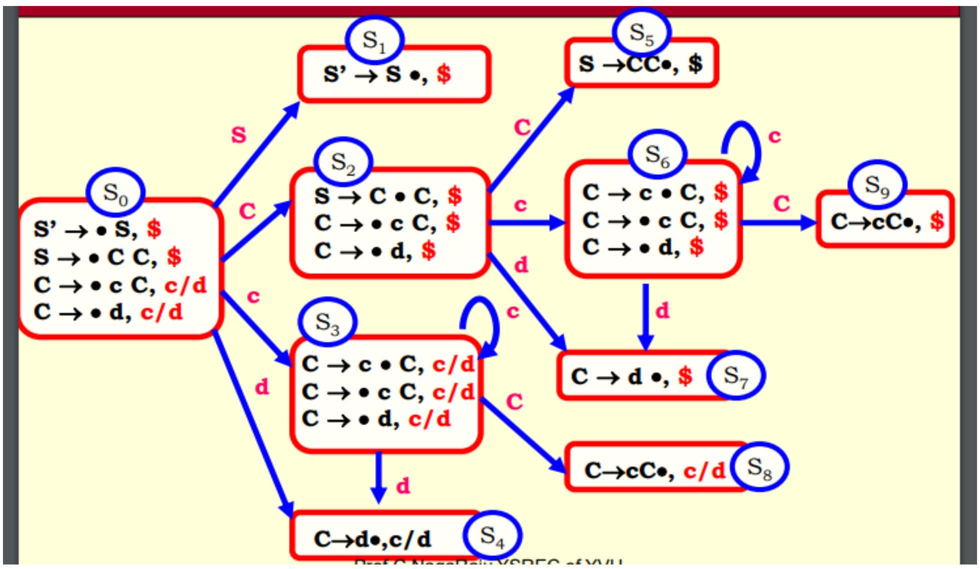
S’ → • S$

S → •CC

C → •cC

C → •d





Construction of Follow Function

S’ → S$

S → C C

C → c C

C → d

Follow (S) = { $ }

Follow (C) = { $,c, d }