LR Parsers

LR parsers are bottom-up parsers. In general, they are called LR(k) parsing where,

L → is for left to right scanning of input
R → is for constructing a rightmost derivation in reverse
and k → is for number of symbols of look ahead

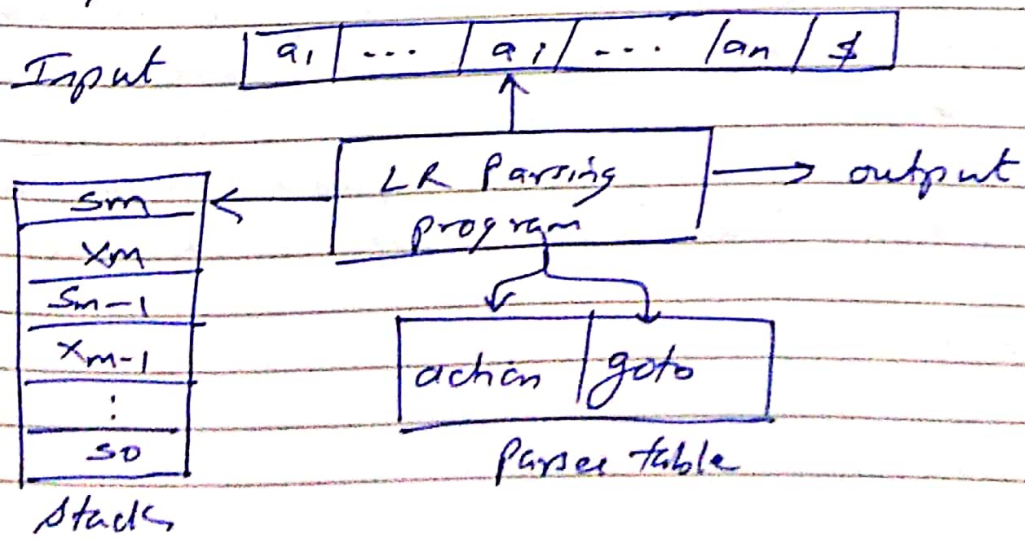
When (k) is omitted, it is assumed to be 1.

* Note that the class of grammars that can be parsed using LR methods is a proper superset of class of grammars that can be parsed with predictive parsers

Drawback: Too much work to construct LR parser.

Solution: Need specialized tools for LR parser generation.
[SLR, CLR and LALR].

Working model of LR parser.



Regarding stack:

Observe that stack contains string

$S0\ X_1\ S_1\ X_2\ S_2 \cdots X_{m-1}\ S_{m-1}\ X_m\ S_m$

↑ Bottom of stack

↑ Top of stack

Here, $S0, S_1, S_2 \cdots, S_{m-1}, S_m$ are all states.

and $X_1, X_2, \cdots X_{m-1}, X_m$ are symbols of grammar [terminal / Variable]

Note that 1) Bottom of stack is S0 which is like initial state and top of stack is S_m which is like current state.

2) Entire stack has alternating state and grammar symbol.

Regarding Parsing Action:

Assuming that the tos contains S_m and the input pointer points to symbol a_i. The parser program refers to the entry in parser table $[S_m, a_i]$ for parsing action. Parsing action can be shift or Reduce.

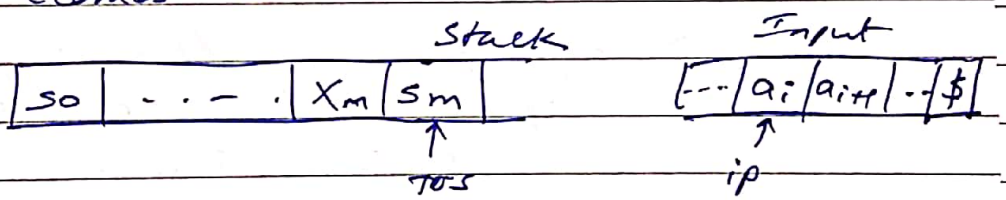
Regarding Parser Table:

Parser table has 2 parts, 1) Action and 2) Goto.

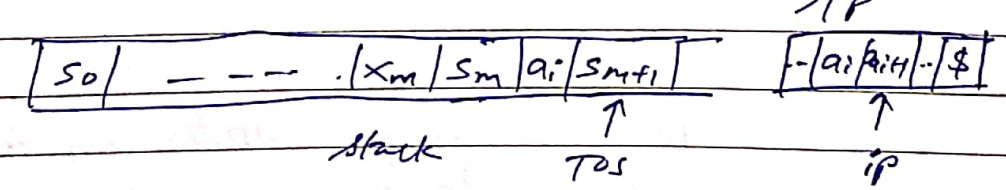
1) Action part of Parser Table:

For ~~each~~ tos s_m and input a_i, the action $[s_m, a_i]$ of parser table can be one of the following :-

a) shift s : Then the parser shifts the input symbol a_i on top of stack, input pointer advances to next symbol a_{i+1} and then the parser pushes the state s on top of stack. Hence new current state of stack becomes s.



action $[s_m, a_i] = s_{m+1}$



b) reduce $A \to \beta$: Then the parser performs reduce action by poping out β from ~~to~~ top of stack and replacing it by A.

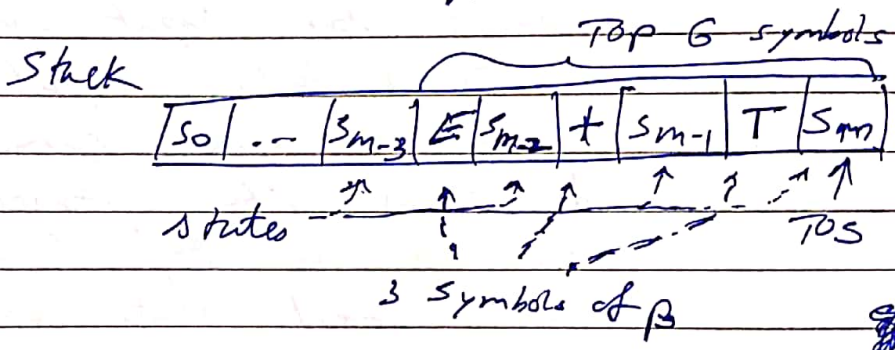
Assume that length of β is r.

[Example : for $E \to E+T$ as $A \to \beta$, length of β (i.e $E+T$) is 3]

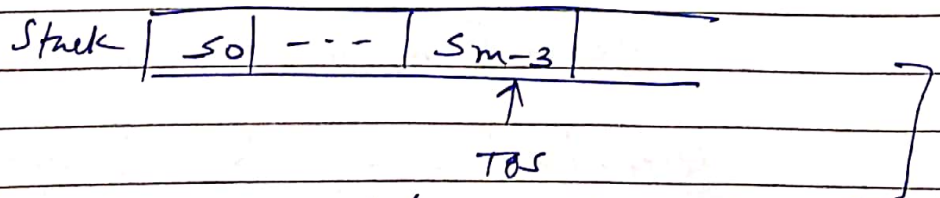
Here, parser pops top 2r symbols,

Note that alternate stack contents are grammer symbols and state. Hence for β of length r, poped contents will be 2r.

[For Example $E \to E+T$. as $A \to \beta$, stack will pop top 6 symbols, which will appear as



* In general, after it pops β from top of stack, the top will be a state S_{m-r}

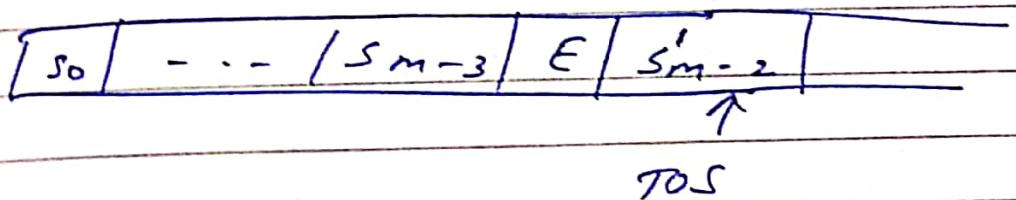


With S_{m-r} state on top of stack and the left variable of production as A, parser refers to the Goto part of parser table. goto $[S_{m-r}, A]$

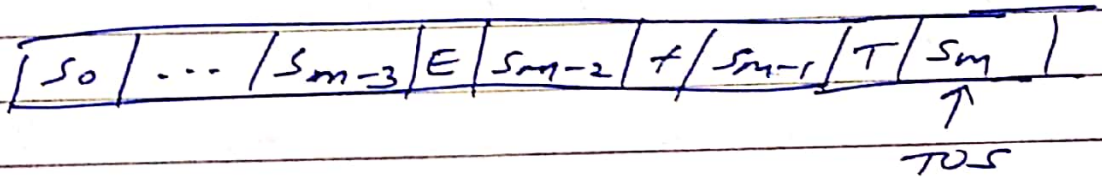
This provides the new current state $S_{m-(r-1)}$ (or S_{m-r+1})

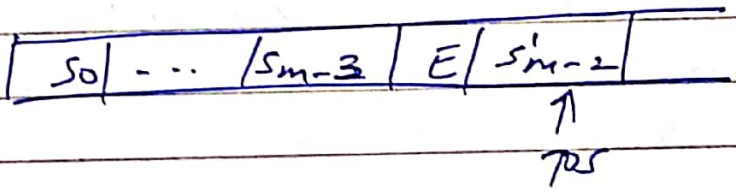
Now parser pushes A and state S_{m-r+1} on top of stack.

[For the given example

Stack 

Finally; with reduce $A \to \beta$ an entry of $[s_m, a_i]$ the stack is

Before 

After 

Note that s'_{m-2} can be a different state than s_{m-2}.]

c) accept : Parsing is completed

d) error : parser discovered an error.

Working of LR parser for grammar

$S \rightarrow CC$

$C \rightarrow aC \mid b$

The 3 productions are:

Production # numbers:

1) $S \rightarrow CC$
2) $C \rightarrow aC$
3) $C \rightarrow b$

Parser Table (PT) (How is this created? Ans: Will discuss later)

State	action			Goto	
	a	b	$	S	C
	(Terminals with $)			(Variables)	
0	s3	s4		1	2
1			Accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Action entry meaning.

s3 : Shift action and new state is 3

Illy s4 : Shift action and new state is 4

r3 : Reduce action & Reduce by production number 3# (ie $C \rightarrow b$).

Illy r2 : Reduce action - Reduce by production number 2# (ie $C \rightarrow aC$)

Understand the working of LR parser by parsing input string "aabab"

(I) Initially stack is initialized with "$0" where 0 represents initial state of stack and input is initialized with "aabab$".

Stack	Input
$0	aabab$

(II) With state on TOS as '0' and next input as 'a', the parser table entry PT[0, a] = s3.
Hence shift input 'a' on TOS and state of stack becomes 3.

Stack	Input
$0a3	abab$

(III) PT[3, a] = s3. Hence same action is repeated again.

Stack	Input
$0a3a3	bab$

(IV) PT[3, b] = s4. Hence the new state (after shift) is 4.

Stack	Input
$0a3a3b4	ab$.

(V) With state as '4' and input 'a'
PT [4, a] = r3.
r3 means - reduce by production number 3 (i.e C → b).
Hence pop 'b' from stack and push 'C'. To pop 'b', 2 elements are poped (as discussed earlier)

Intermediate - 1

Stack	Input
$ 0 a 3 a 3	a b $

Now the new state of stack (TOS) is 3. Push 'C' on 3.

Intermediate - 2

Stack	Input
$ 0 a 3 a 3 C	a b $

With state '3' and pushed 'C', refer goto [3, C] from PT.
goto [3, C] = 6.
Hence new state of stack is 6

Finally

Stack	Input
$ 0 a 3 a 3 C 6	a b $

(VI) PT entry for action [6, a] = r2.
Hence reduce by production 2 (i.e C → ac).
Therefore pop 'aC' from stack and push 'C'.

Like step IV, we get

Stack	Input
$ 0 a 3 C 6	a b $

~~Continuing~~ Similarly the parse actions are performed till error or i/p is accepted.

Stack	Input	PT entry
$ 0 a 3 C 6	a b $	[6, a] = r2
$ 0 C 2	a b $	[2, a] = s3
↳ goto[0, C] = 2		
$ 0 C 2 a 3	b $	[3, b] = s4
$ 0 C 2 a 3 b 4	$	[4, $] = r3
$ 0 C 2 a 3 C 6	$	[6, $] = r2
$ 0 C 2 C 5	$	[5, $] = r1
$ 0 S 1	$	[1, $] = Accept

⇒ Designing Techniques for Passes Table
↳ SLR, CLR, LALR