Unit-2

* **left recursion :-** The left recursion is the grammar of the form $A \longrightarrow A\alpha$

* if the left recursion is present in the grammar. then it creates serious problem, because of this left recursion the topdown parser problem may enter into the infinite loop

* To eliminate the left recursion we need to modify the grammar, if the grammar has the production rule with left recursion. then eliminating the left recursion then the by rewriting the productions as rules are.

* $A \longrightarrow A\alpha | \beta$

  $A \longrightarrow \beta A'$

  $A' \longrightarrow \alpha A' | \epsilon$

Ex:- 1) $E \longrightarrow E+T | T$ — by eliminating the left recursion

$E \longrightarrow TE'$

$E' \longrightarrow +TE' | \epsilon$

2) $T \longrightarrow T*F | F$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' | \epsilon$

3) $A \longrightarrow ABd | Aa | a$

i) $A \longrightarrow ABd | a$

ii) $A \longrightarrow Aa | a$

4) $B \longrightarrow Be | b$

$B \longrightarrow bB'$

$B' \longrightarrow eB' | \epsilon$

i) $A \longrightarrow ABd | a$

$A \longrightarrow aA'$

$A' \longrightarrow BdA' | \epsilon$

ii) $A \longrightarrow Aa | a$

$A \longrightarrow aA'$

$A \longrightarrow aA' | \epsilon$

* **Left factoring :-**

if the grammar is left factoring then it becomes suitable for the to use left factor is used when it is not clear which are the two alternatives to expand the non-terminal

* By left factoring we may be able to rewrite the production as another productions

* if $A \longrightarrow \alpha\beta_1 | \alpha\beta_2 \cdots$ then it is not possible to take a decision weather to choose first rule or second rule

In such a situation the above grammar can be left factored. has $A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2 \ldots$

T-L
B-R

Ex:- 1) $S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

$S \rightarrow iEtSS' / a$ where $[\beta_1 = \epsilon]$

$S' \rightarrow eS / a$

$E \rightarrow b$

2) $A \rightarrow aAB / aA / a$

$B \rightarrow bB / b$

$A' \rightarrow aAA' / a$

$A' \rightarrow B / a$

$\rightarrow B \rightarrow bB'$

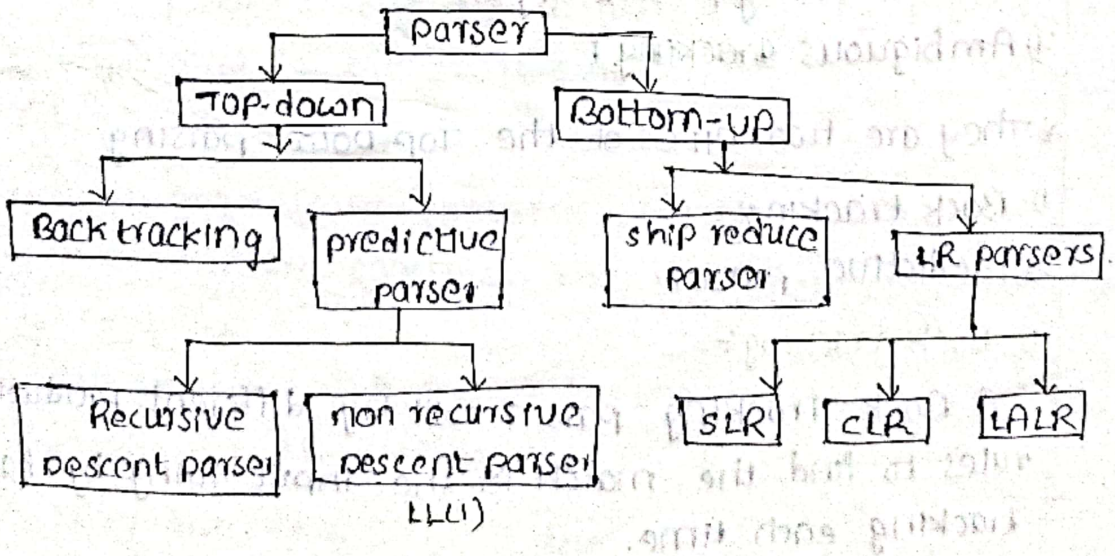
$B' \rightarrow B$

* Back tracking :-

Classification of the parsing techniques

1) when the parse tree can be constructed from roots and extended to leaves then that type of parse is called Top down parse tree.

2) when the parse tree canbe constructed from leaves to the roots then such type of parse is called Bottom up parse tree



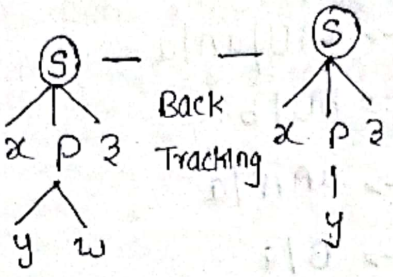
1) Top-Down parsing :-

A Top-Down parser a parse tree is generated from Top to Bottom. The left most derivation matches this type of requirement. The main task in Top-Down parsing is to find the appropriate production rule to produce current input string.

$S \rightarrow xPz$
$P \rightarrow yw/y$

Derive the input string xyz

$S \rightarrow xPz$
$S \rightarrow xyz$



In Top-Down parsing selection of proper rule is important and this selection is based on Trial and Error. we have to select a particular rule & it is not producing the current input string then we need to backtrack and then we have to try another production. This process has been repeated until we get the current input string.

Problems with Top-Down parsing :-

1) Back tracking
2) Left recursion
3) Left tracking / factoring
4) Ambiguous grammar

* They are two types of the Top-Down parsing

1) Back tracking
2) predictive parser

1) Back tracking :-

A Back Tracking parser will try different production rules to find the match for the input string by back tracking each time.

The back tracking is power full than predictive parsing

but this technique is slower and requires time.
Hence Back tracking is not preferred.

2) Predictive parser:
It tries to predict the next construction using one or more symbols from input string.

* They are two types of predictive parsers:-
1) Recursive descent parser
2) Non-recursive descent parser (or) LL(1) parser.

1) Recursive descent parser:-
The parser that uses the collection of recursive procedures for passing the given input string is called recursive descent parser. In this type of parser CFG is said to be used to produce recursive Routines.

For each non-terminal a separate procedure written which is corresponds to the non-terminal.

Steps for the Recursive descent parser:-
1) if the input string is non-terminal then a call to the procedure corresponding to the non-terminal.
2) if the input symbol is terminal then it is matching to the symbol from the input string
3) if the production rule has many alternatives all has to be combined to a single body of procedure.
4) The parser should be activated by the procedure corresponding to the start symbol.

Advantages:-
1) Recursive descent parser is simple to built.
2) They can be construct the with help of parse tree.

Disadvantages:-
1) Recursive descent parser are not very efficient has compared to the other parser techniques.
2) They are chances that the program may enter into the infinite loop.

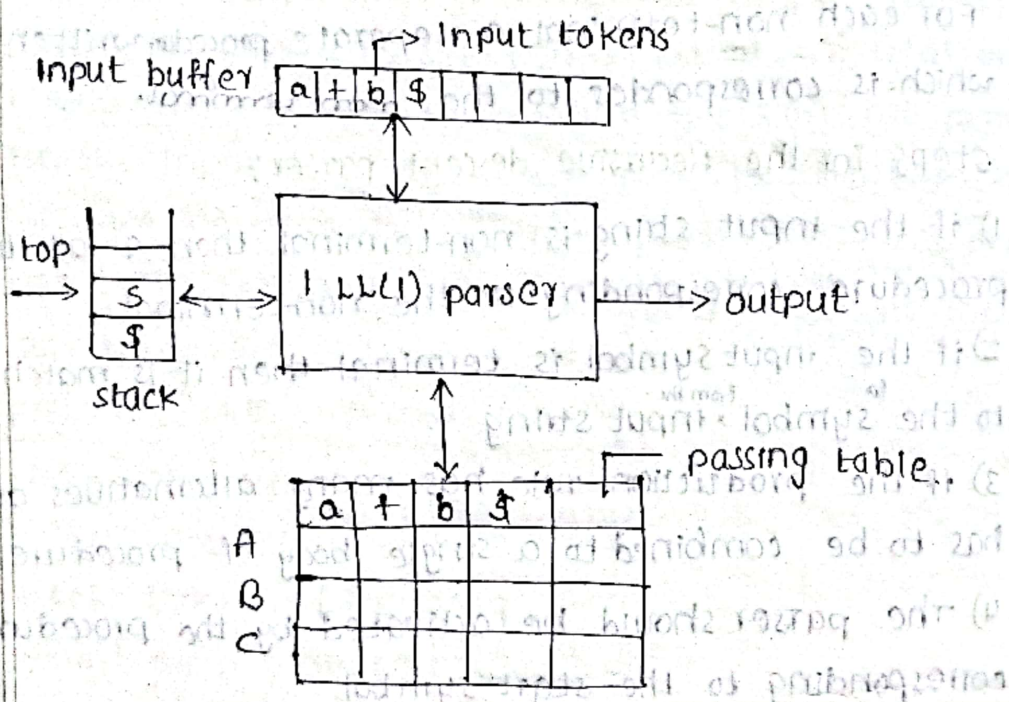
3) it can't provide good error messaging.
4) it is difficult to parser string if it is dif input string is wrong. long

preprocessing steps for the top-down parser

1) start symbol
2) Apply production rules
3) Match the input string
4) Parsing
5) Scan from left to right

2) Non Recursive descent parser (or) LL(1) parser:-

LL(1) — L-scan from left to right
L-It defines left most derivation.
1- Reads 1 character at a time.



In the input buffer, it is used to store the input tokens. this stack is used to hold the left sentential form.

* the symbols in the right hand side rule are pushed into the stack in reverse order from left to right
* use of the stack makes the algorithm non-recursive.
* the table has a row for non-terminals and column for terminals.

* The parser consults the table each time while taking the parser actions. This type of parsing method is called table recursive passing algorithm.

Construction of LL(1)

1) Eliminate left regression.
2) Eliminate left factoring
3) Compute first function FOLLOW function i.e FIRST() and FOLLOW()
4) constructing the parser (or) passing table using first and FOLLOW functions.
5) pass the input string with the help of predictive passing table.

Ex:- $E \rightarrow E+T$, consider the given grammar as follows.
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

-> it can be summarized as follows

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

Sol:- Given, $E \rightarrow E+T|T$
$T \rightarrow T*F|F$
$F \rightarrow (E)|id$

Step-1

$E \rightarrow E+T/T$ <- By using left regression, we get the productions.
$E \rightarrow TE'$
$E' \rightarrow +TE'/\epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'/\epsilon$
$F \rightarrow (E)|id$

Step:-2 In the above productions we don't have any left factoring productions.

Then moves to the step3 with the productions

FIRST(∝) :-

1) FIRST(∝) is a set of terminal symbols that are appears first at right side of the derivation of the alpha(∝).

2) if '∝' tends to 'ϵ' then ϵ is also in FIRST(∝).

Rules :-

1) If 'x' is a terminal, FIRST(x) = {x}

2) If 'x' is non terminal and x → ϵ is a production, then FIRST (x) = {ϵ}

3) If 'x' is non-terminal and x → $Y_1Y_2Y_3 \cdots Y_K$ then FIRST (X) = FIRST(Y_1) ∪ FIRST(Y_2) ∪ ..... FIRST(Y_K)

4) If Y_1 → ϵ then ADD FIRST (x) = FIRST (Y_2)

5) If Y_2 → ϵ then ADD FIRST (X) = FIRST (Y_3)

⋮

6) If Y_K → ϵ then ADD FIRST (X) = {ϵ}

Step:-3

FIRST() :-

The productions are :-

E → TE'	FIRST(E) = {(, id}
E' → +TE'/ϵ	FIRST(E') = {+, ϵ}
T → FT'	FIRST (T) = {(, id}
T' → *FT'/ϵ	FIRST(T') = {*, ϵ}
F → (E)/id	FIRST(F) = {(, id}

1) if FIRST(X) = FIRST(Y_1) add all symbols except 'ϵ', if Y_1 → ϵ then it may be replaced by 'ϵ' production so that, we need to add FIRST (Y_2) is also to the FIRST(X)

2) if Y_2 → ϵ then FIRST(Y_3) is also included in the FIRST(x), for suppose if $Y_1, Y_2, Y_3 \cdots Y_K$ all the deriving 'ϵ' then we need to add 'ϵ' to FIRST(x)

consider the grammar, S → ABCDE
A → a/ϵ
B → b/ϵ
C → c

D → d | ϵ
E → e | ϵ compute FIRST Function.

FIRST (S) = {a, b, c}
FIRST (A) = {a, ϵ}
FIRST (B) = {b, ϵ}
FIRST (C) = {c}
FIRST (D) = {d, ϵ}
FIRST (E) = {e, ϵ}

consider S → ABB | CbB | Ba
A → da | BC
B → g | ϵ compute FIRST function
C → h | ϵ

FIRST (S) = {d, g, h, ϵ, a, b}
FIRST (A) = {d, g, h, ϵ}
FIRST (B) = {g, ϵ}
FIRST (C) = {h, ϵ}

consider the grammar compute FIRST Function

S → EFG
E → gd
F → f | ϵ
G → K | l

FIRST (X) = {g}
FIRST (E) = {g}
FIRST (F) = {f, ϵ}
FIRST (G) = {k, l}

S → aXZh
X → cY
Y → bY | ϵ
Z → EF
E → g | ϵ
F → f | ϵ

FIRST (S) = {a}
FIRST (X) = {c}
FIRST (Y) = {b, ϵ}
FIRST (Z) = {g, f, ϵ}
FIRST (E) = {g, ϵ}
FIRST (F) = {f, ϵ}

FOLLOW() :-

FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right (A).

Rules :-

1) For the starts symbol, S FOLLOW(S) = {$} i.e place $ in FOLLOW(S)

2) if $A \longrightarrow \alpha B\beta$ then FOLLOW(B) = FIRST(β) except ϵ

3) if $A \longrightarrow \alpha B$ (or) $A \longrightarrow \alpha B\beta$ where FIRST(β) continus ϵ then FOLLOW(B) = FOLLOW(A)

4) if you want to find the FOLLOW() we need to check the right hand side of the production.

The productions are :-

$E \longrightarrow TE'$	FOLLOW(E) = {$,)}
$E' \longrightarrow +TE'/\epsilon$	FOLLOW(E') = {$,)}
$T \longrightarrow FT'$	FOLLOW(T) = {+,), $}
$T' \longrightarrow *FT'/\epsilon$	FOLLOW(T') = {+,), $}
$F \longrightarrow (E)/id$	FOLLOW(F) = {*, +,), $}

Step 4 : Algorithm for predictive parsing table.

A construction of predictive parsing table in important avitivity in predictive parsing method. This algorithm required FIRST and Follow functions

Algorithm rules :-

* For the rule $A \longrightarrow \alpha$ for grammar a

1) For each 'a' in FIRST(α), create entry $M[A, a] = A \longrightarrow \alpha$

2) FOR 'e' in FIRST(α) create entry $M[A, b] = A \longrightarrow \alpha$ where 'b' is the symbol from FOLLOW(A)

3) if ϵ is in FIRST(α) and $ is in FOLLOW(A) then create entry in the table $M[A, \$] = A \longrightarrow \alpha$

4) All other remaining entries in the table are as syntax error

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

step 5: predictive passing table.

Now, the input string id+id*id can be parsed using above table. At the initial configuration this stack will contain '$' symbol. And the input buffer has input string is placed.

stack	input	action
$E	id+id*id$	—
$E'T	id+id*id$	$E \to TE'$
$E'T'F	id+id*id$	$T \to FT'$
$E'T'id	id+id*id$	$F \to id$
$E'T'	+id*id$	—
$E'	+id*id$	$T' \to \epsilon$
$E'T+	+id*id$	$E' \to +TE'$
$E'T	id*id$	T —
$E'T'F	id*id$	$T \to FT'$
$E'T'id	id*id$	$F \to id$
$E'T'	*id$	—
$E'T'F*	*id$	$T' \to *FT'$
$E'T'F	id$	—
$E'T'id	id$	$F \to id$
$E'T'	$	—
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

∴ The string is accepted. string is empty.

LL(1) Grammars :-

1) Show that the following grammar is LL(1) grammar.

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Sol: Given grammar,

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

FIRST() :-

$FIRST(S) = \{a, b\}$

$FIRST(A) = \{\epsilon\}$

$FIRST(B) = \{\epsilon\}$

FOLLOW() :-

$FOLLOW(S) = \{\$\}$

$FOLLOW(A) = \{a, b\}$

$FOLLOW(B) = \{b, a\}$

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

→ Parse the string 'ba' by using the above stable.

stack	input	action
$S	ba$	—
$aBbB	ba$	$S \rightarrow BbBa$
$aBb	ba$	$B \rightarrow \epsilon$
$aBb	ba$	—
$aB	a$	—
$a	$	$B \rightarrow \epsilon$
$	$	$ -

∴ Given grammar is LL(1) grammar.

2) For the following grammar find the FIRST and FOLLOW for each terminal.

$S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAB \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

Sol: Given.

$S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAB \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

FIRST ():

$FIRST(S) = \{a, b, \epsilon\}$

$FIRST(A) = \{a, \epsilon\}$

$FIRST(B) = \{b, \epsilon\}$

FOLLOW():

$FOLLOW(S) = \{\$\}$

$FOLLOW(A) = \{b, \$\}$

$FOLLOW(B) = \{b, \$\}$

	a	b	$
S	$S \rightarrow aAB$	$S \rightarrow bA$	
A	$A \rightarrow aAB$		
B		$B \rightarrow bB$	

∴ It is a LL(1) grammar because it contains only one input string.

3) consider the grammare, prove that the grammar is LL(1) (or) not.

$S \rightarrow iCbSA \mid a$

$A \rightarrow eS \mid \epsilon$

$C \rightarrow b$

Sol: Given grammar

$S \rightarrow iCtSA \mid a$

$A \rightarrow eS \mid \epsilon$

$C \rightarrow b$

FIRST() :-

FIRST(S) = {i, a}

FIRST(A) = {e, ∈}

FIRST(C) = {b}

FOLLOW() :-

FOLLOW(S) = {$, e}

FOLLOW(A) = {e, $}

FOLLOW(C) = {t}

	i	t	a	e	b	$
S	S→iCtSA		S→a			
A				A→eS A→∈	A→∈	
C					C→b	

As we got multiple entries in the table, so the given grammar is not a LL(1) grammar.

u) consider the predictive parser for the given grammar and also derive the input string (a,a)

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

Sol:

Given grammar.

$S \to (L) \mid a$

$L \to L,S \mid S$

$S \to (L)$

$S \to (L,S)$

$S \to (S,a)$

$S \to (a,a)$

$L \to L,S \mid S$

$L \to SL'$ $\Rightarrow$

$L' \to ,L' \mid \epsilon$

$S \to (L) \mid a$

$L \to SL'$

$L' \to ,SL' \mid \epsilon$

$\therefore [A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon]$

FIRST(S) = { (, a }

FIRST(L) = { (, a }

FIRST(L') = { , , ∈ }

FOLLOW(S) = { $,) }

FOLLOW(L) = {) }

FOLLOW(L') = {) }

	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→∈		L'→,SL'	

The parser input string is (a,a)

stack	input	string
$ S	(a,a) $	—
$) L (	(a,a) $	S→(L)
$) L	a,a) $	—
$) L' S	a,a) $	L→SL'
$) L' a	a,a) $	S→a
$) L'	,a) $	—
$) L' S ,	,a) $	L'→,SL'
$) L' S	a) $	—
$) L' a	a) $	S→a
$) L'	) $	—
$)	) $	L'→∈
$	$	—

∴ stack is empty so both strings matched.

* Strategies to recover from syntax errors:

parsers have various strategies to recover from syntax errors. These are

1) panic mode error recovery
2) phrase level error recovery
3) Error productions
4) Global corrections

1) Panic mode error recovery:-

* These strategy is used by most parsing methods and is simple to implement.
* In these method on discovering error, the parser discards input symbol one at a time. This process is continue until one of a designated set of synchronizing tokens is found.
* thus, in panic mode error recovery an amount of input is skipped without changing.
* this method guarantees not to enter into the infinite loop.
* if they are less number of the errors in the statement then this choice is the best choice. i.e strategy

2) Error recovery in predictive parsing:-

An error is repeated during the predictive parsing when terminal on the top of the stack doesn't match the input symbol or) when non-terminal on the top of the stack and the input symbol in which the parsing table entry is empty.

* pandic mode error recovery is based on the idea of skipping symbols on the input in a selected set of synchronizing tokens.
* place all the symbols in the FOLLOW() into the synchro-nized set of non-terminals. if we skip the tokens until an element of the FOLLOW() is seen and pop variable from the stack.

* if parser looks up the entry as blank then the input symbol is skipped.
* if entry is synch then the non-terminal of the top of the stack is pop
* if token on the top of the stack doesn't match the input symbol then pop the symbol from the stack.

ex:
construct the grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' / \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \epsilon$
$F \rightarrow (E) / id.$

FIRST(E) = { (, id }
FIRST(E') = { +, ε }
FIRST(T) = { (, id }
FIRST(T') = { *, ε }
FIRST(F) = { (, id }

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +,), $ }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = { *, +,), $ }

	+	*	(	)	id	$
E	skip	skip	$E \rightarrow TE'$	sync	$E \rightarrow TE'$	sync
E'	$E' \rightarrow +TE'$	skip	skip	$E' \rightarrow \epsilon$ $E' \rightarrow +TE'$	skip	$E' \rightarrow \epsilon$
T	sync	skip	$T \rightarrow FT'$	sync	$T \rightarrow FT'$	sync
T'	$T' \rightarrow *FT'$	$T' \rightarrow *FT'$	skip	$T' \rightarrow \epsilon$	skip	$T' \rightarrow \epsilon$
F	sync	sync	$F \rightarrow (E) / id$	sync	$F \rightarrow (E)$	sync

- Derive the input string +id**id$

stack	input	action.
$E	+id**id$	—
$E	id**id$	$E \xrightarrow{skip} skip$
$E'T	id**id$	$E \rightarrow TE'$
$E'T'F	id**id$	$T \rightarrow FT'$
$E'T'id	id**id$	$F \rightarrow id$
$E'T'	**id$	—
$E'T'F*	**id$	$T' \rightarrow *FT'$

$E'T'F	*id$	—
$E'T'	*id$	Sync - error. pop 'F' from stack.
$E'T'F*	*id$	$T' \rightarrow *FT'$
$E'T'F	id$	—
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	—
$	$	$T' \rightarrow \epsilon$ $E' \rightarrow \epsilon$

Derive the input string by using above table.

() id * id $.

Given, input string is

() id * id $ By using passing table.

stack	input	action
$E	()id*id$	—
$E'T	()id*id$	$E \rightarrow TE'$
$E'T'F	()id*id$	$T \rightarrow FT'$
$E'T')E(	()id*id$	$F \rightarrow (E)$
$E'T')E	)id*id$	—
$E'T')	)id*id$	E → sync - error pop 'E' from stack
$E'T'	id*id$	—
$E'T'	*id$	$T' \rightarrow id$ (skip)
$E'T'F*	*id$	$T' \rightarrow *FT'$
$E'T'F	id$	—
$E'T'	id$	F → synch - error pop 'F' from stack
$E'	id$	$T' \rightarrow \epsilon$
$E'T+	+id$	$E' \rightarrow +TE'$
$E'T	id$	—
$E'T'F	id$	$T \rightarrow FT'$
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	—
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

2) phrase level error recovery :-

In this method of discovering error, parser perform global correction on remaining input.

* It can replace a the prefix of remaining input by some string. This helps the parser to continue its job.

* The global correction can be replacing , by ; (or) inserting missing semicolon. A type global correction is designed by compiler design. This method is used in many error repairing compilers.

* The drawback of this method is it finds difficult to handle the situations where the actual error has ocurred.

3) Error production :-

if we have a knowledge on common errors that can have encountered then we can reformed these errors. by updating (augmenting) the grammar of the corresponding language with error productions that generate the errors.

if error production is used during parsing we can generate an appropriate message and the parsing can be continued.

This method is extremely difficult to maintain because this (if we) will change the grammar then it becomes necessary to change the corresponding error productions.

4) Global correction :-

* We want such a compiler that makes a few very few changes that preprocessing an (correct) ranter input string

* They except less number of insertions, deletions and changes to recover from error.

* These methods increase time and space at parsing time.

2) Bottom-up parsing:

* we can construct the parse tree from Bottom to the top i.e leaf nodes to the root nodes.

Ex: consider the grammar $S \rightarrow aABe$
$A \rightarrow Abc|b$
$B \rightarrow d$
derive the input string 'abbcde' to get the start symbol.

Given, abbcde

aAbcde [A→b]
aAde [A→Abc]
aABe [B→d]
S [S→aABe]
↳ start symbol

$E \rightarrow E+E|E*E|id$. derive the string id+id*id to get the start symbol

id+id*id
E+id*id [E→id]
E+E*E [E→id]
E+E [E→E*E]
E [E→E+E]

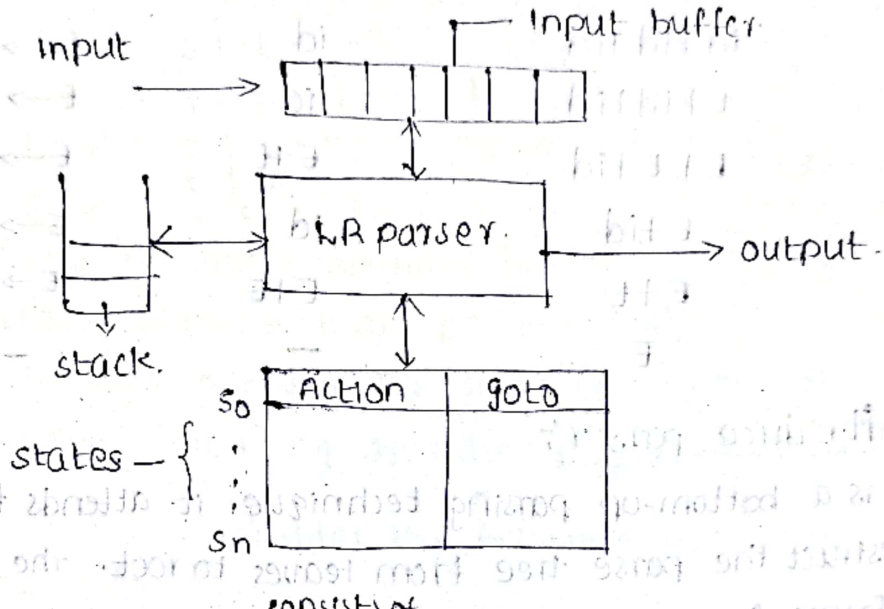
id*id+id
E*id+id [E→id]
E*E+E [E→id]
E*E [E→E+E]
E [E→E*E]

* Difference between LL parsers and LR parsers

LL(1)	LR(Parser)
first 'L' stands for Left to right scanning and second 'L' stand for LMD	1st 'L' stands for left to right scanning and 2nd R stands for RMD
2) it follows left most derivation.	2) it follows right most derivation.
3) In using LL(1) parsing tree is constructed from Top to Bottom	3) using LR parsing trees constructed from bottom to top
4) In LL(1) parsers non-terminals are expanded	4) in LR parsers terminals are compressed.

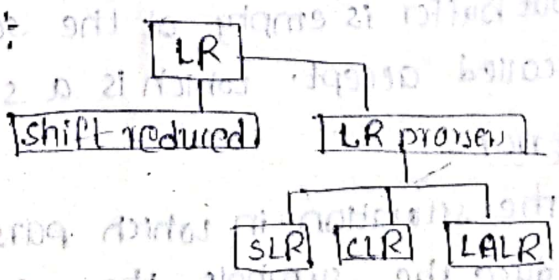
5) starts with an start symbol	5) starts with the input string.
6) LL(1) follows pre ordered traversal of tree.	6) LR follows post ordered traversal of tree.
7) it may use Back tracking (or) dynamic programming	7) it uses only dynamic programming
8) LL(1) is easy to write	8) LR is difficult to write.

* Model of an LR parser:-



* LR parser is consists of used for input buffer, stack, and parsing table.
* Input buffer is used to storeg the input string
* stack is used to store the grammar symbols
* The parsing table can be comperesed into two parts one is Action and goto respectively.
* there is a parsing algorithm which is a actually a program that reads a input symbol once at a time. from the input buffer.

Types of LR parser:-



Handle truning:

*the substring that could be reduced by the appropriate non-terminal, such substring is called handle truning.

*Hangling is the string of substring that matches the righthand side of the productions.

Ex: consider the grammar $E \rightarrow E+E|id$. derive the string id+id+id.

Right sentential form	Handle	production
id+id+id	id	$E \rightarrow id$
E+id+id	id	$E \rightarrow id$
E+E+id	E+E	$E \rightarrow E+E$
E+id	id	$E \rightarrow id$
E+E	E+E	$E \rightarrow E+E$
E	—	—

* shiftreduce parser:

It is a bottom-up parsing technique it attends to construct the parse tree from leaves to root. the parser performs four basic operations. they are:

1) shift —

moving the symbols from input buffer on to the stack. this action is called shift.

2) reduce —

if the handle appears on the top of the stack then reduction of it by appropriate rule.

3) accept —

if the stack contains start symbol only and the input buffer is empty at the same time then the action is called accept. which is a successfull parsering.

4) Error —

The situation in which parser cant either shift or reduce the symbols, they can't even perform the accept action it is called an error.

1) Ex:- consider the grammar E → E-E | E*E | id. perform shift reduce parsing of the input string id-id*id.

sol:-

stack	input	action
$	id-id*id$	shift
$id	-id*id$	reduce E→id
$E	-id*id$	shift
$E-	id*id$	shift
$E-id	*id$	reduce E→id
$E-E	*id$	shift
$E-E*	id$	shift
$E-E*id	$	reduce E→id
$E-E*E	$	reduce E→E*E
$E-E	$	reduce E→E-E
$E	$	accept.

Rules:-
- if the incoming operator has more priority than in stack operator than perform shift.
- if in stack operator has same or less operator than the priority of incoming operator perform reduce.

2) Example 2:- consider the following grammar.

S → TL;
T → int | float
L → L,id | id

pas the inputstring int id,id; using shift reduce parser:

stack	input	action
$	int id,id;$	shift
$int	id,id;$	reduce T→int
$T	id,id;$	shift
$Tid	,id;$	reduce id ie L→id
$TL	,id;$	shift
$TL,	id;$	shift
$TL,id	;$	reduce shift L→L,id
$TL;	;$	shift
$TL;	$	reduce S→TL;
$S	$	accept.

3. consider the following grammar $S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S$
parse the input string (a,(a,a)) using shift reduce parser

stack	input	action
$	(a,(a,a))$	shift
$(	a,(a,a))$	shift
$(a	,(a,a))$	reduce S→a
$(S	,(a,a))$	L→S
$(L	,(a,a))$	shift
$(L,	(a,a))$	shift
$(L,(	a,a))$	reduce S→a
$(L,(S	a,a))$	reduce L→S
$(L,(L		wrong

4) consider the grammar, design shift reduce parser.

10201

$S \rightarrow 0S0 \mid 1S1 \mid 2$

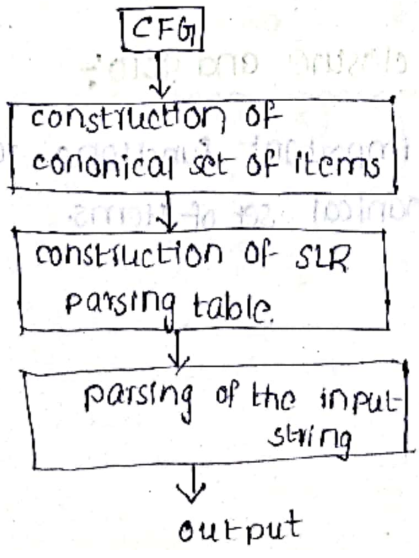
stack	input	action
$	10201$	shift
$1	0201$	shift
$10	201$	shift
$102	01$	reduce S→2
$10S	01$	shift
$10S0	1$	reduce S→0S0
$1S	1$	shift
$1S1	$	reduce S→1S1
$S	$	accept

stack	input	action.
$	(a,(a,a))$	shift
$(	a,(a,a))$	shift
$(a	,(a,a))$	reduce S→a
$(S	,(a,a))$	reduce L→S
$(L	,(a,a))$	shift
$(L,	(a,a))$	shift
$(L,(	a,a))$	shift
$(L,(a	,a))$	reduce S→a
$(L,(S	,a))$	reduce L→S
$(L,(L	,a))$	shift
$(L,(L,	a))$	shift
$(L,(L,a	))$	reduce S→a
$(L,(L,S	))$	reduce L→L,S
$(L,(L	))$	shift
$(L,(L)	)$	reduce S→(L)
$(L,S	)$	reduce L,S←L
$(L	)$	shift
$(L)	$	reduce S→(L)
$S	$	accept

1) SLR(1) parser :-

* We will start the simplest form of LR parser.
* It is the weakest of the 3 methods but it is easy to implement

construction of the SLR parsers :-



* The grammar for which SLR parser can be constructed is called SLR grammar.

Definition of LR of zero items and the related terms:

1)* The LR of (0) item for the grammar G is production rule in which symbol (.) is inserted at some position in RHS of the rule.

Ex:- S → .ABC
S → A.BC
S → AB.C
S → ABC.

2) The production S → ∈ generates only one item S →.

Augmented grammar:-

* If a grammar 'G' is having start symbol 'S' then Augmented grammar is a new grammar 'G' in which S is a new start symbol such that S' → S . The purpose of this grammar is to
The acceptance of the input when parser is about to reduce S' → S. It reaches to the acceptance string.

Kernel items:-

It is a collection of items S' → .S and all the items whose dots are not at the left most end of RHS of Rule.

Non-kernel items:-

The collection of all the items in which (.) are at left end of the RHS of the rule.

Functions and closure and goto:-

There are 3 important functions required to a collection of canonical set of items.

Viable and prefixes:-

It is a set of prefixes in the right sentisial form of production, $A \rightarrow \alpha \cdot \cdot$ This set can appear on the stack during shift (or) reduce action.

closure operation:-

* For a CFG if 'I' is the set of items then the function closure of 'I' be constructed using the following rules:

1) consider 'I' is a set of canonical items and initially every item 'I' is added to closure (I)

2) If Rule $A \rightarrow \alpha \cdot B\beta$ is rule in closure (I) and there is another rule for 'B' such as $B \rightarrow \cdot\delta$ then closure (I): $A \rightarrow \alpha \cdot B\beta$, $B \rightarrow \cdot\delta$

3) This rule has to be applied until no more new items can be added to closure (I).

Goto operation:-

* If there is a production $A \rightarrow \alpha \cdot B\beta$ then goto $(A \rightarrow \alpha \cdot B\beta, B) = A \rightarrow \alpha B \cdot \beta$ that means simple shifting of (·) one position over the grammar symbol.

* The rule $A \rightarrow \alpha \cdot B\beta$ is in 'I' then some goto() function can be written as goto(I, B)

Q: Consider the grammar $X \rightarrow Xb/a$. compute closure and goto operations.

Given grammar $X \rightarrow Xb/a$

I_0: $X \rightarrow \cdot Xb$
$X \rightarrow \cdot a$ } – closure operation

I_1: goto (I_0, X)
$X \rightarrow X \cdot b$ } – goto operation.

I_2: goto (I_0, a)
$X \rightarrow a \cdot$

2) consider the following grammar

$S \to AS \mid b$
$A \to SA \mid a$

compute closure and goto operation

$S \to AS$
$S \to b$
$A \to SA$
$A \to a$

} canonical set of items

I_0: $S \to \cdot AS$, $A \to \cdot SA$, $S \to \cdot b$, $A \to \cdot a$ } – closure operation

goto operation

I_1: goto(I_0, A)
$S \to A \cdot S$
$S \to \cdot bx$

I_2: goto(I_0, b)
$S \to b \cdot$

I_3: goto(I_0, S)
$A \to S \cdot A$
$A \to \cdot a$

I_4: goto(I_0, a)
$A \to a \cdot$

3) consider

$S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

compute goto and closure operations

Augmented grammar

$S \to Aa$
$S \to bAC$
$S \to BC$
$S \to bBa$
$A \to d$
$B \to d$

I_0:
$S^1 \to \cdot S$
$S \to \cdot Aa$
$S \to \cdot bAC$
$S \to \cdot BC$
$S \to \cdot bBa$
$A \to \cdot d$
$B \to \cdot d$

I_1: goto(I_0, A)
$S \to A \cdot a$

I_2: goto(I_0, b)
$S \to b \cdot AC$
$A \to \cdot d$
$S \to b \cdot Ba$
$B \to \cdot d$

I_3: goto(I_0, B)
$S \to B \cdot C$

I_4: goto(I_0, d)
$A \to d \cdot$
$B \to d \cdot$

I_5: goto(I_1, a)
$S \to Aa \cdot$

I_6: goto(I_2, A)
$S \to bA \cdot C$

I_7: goto(I_2, B)
$S \to bB \cdot a$

I_8: goto(I_3, C)
$S \to BC \cdot$

I_9: goto(I_6, C)
$S \to bAC \cdot$

I_{10}: goto(I_7, a)
$S \to bBa \cdot$

construction of canonical collection of set of items

1. For the grammar 'G' initially hadd $S \rightarrow \cdot S$
2. For each set of items I_i and for each grammar symbol X add closure $[I_i, X]$. This process should be repeated by applying goto(I_i, X) for each X in I_i. The set of items had to be constructed until no more set of items can be added.

Ex: consider the grammar.

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id.$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

step1: The Augmented grammar is

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

} - closure operation

I_0:

I_1: goto (I_0, E)

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot +T$

I_2: goto (I_0, T)

$E \rightarrow T \cdot$

$T \rightarrow T \cdot *F$

I_3 : goto(I_0, F)

T → F.

I_4 : goto(I_0, ()

F → (·E)

E → ·E+T

E → ·T

T → ·T*F

T → ·F

F → ·(E)

F → ·id

I_5 : goto(I_0, id)

F → id·

I_6 : goto(I_1, +)

E → E+·T

T → ·T*F

T → ·F

F → ·(E)

F → ·id.

I_7 : goto(I_2, *)

T → T*·F

F → ·(E)

F → ·id.

I_8 : goto(I_4, E)

F → (E·)

E → E·+T

I_9 : goto(I_6, T)

E → E+T·

T → T·*F

I_{10} : goto(I_7, F)

T → T*F·

I_{11} : goto(I_8,))

F → (E).

shift- next state
i.e dot after terminal

reduce : dot operations

Constructing SLR parsing table :

states	Action						goto		
	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					accept			
2	r_2	S_7		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_6			S_{11}					
9	r_1	S_7		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

For reduce operation, consider the given grammar as

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

Find the follow function of all the variables

FIRST(E) = {

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

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

FOLLOW(F) = {*, $, +,)}

parser input string :- always the top of stack should be state

By using the above parsing table. parser the input string id * id + id

stack	input	action
0	id*id+id$	shift
0id5	*id+id$	reduce by 6 [F→id] — 1×2=2 pop 2 symbols from stack
0F3	*id+id$	reduce by 4 T→F — 1×2=2 pop 2 symbols from stack
0T2	*id+id$	shift
0T2*7	id+id$	shift
0T2*7id5	+id$	reduce by 6 F→id — 1×2=2 pop 2 symbols from stack
0T2*7F10	+id$	reduce by 3 T→T*F — 3×2=6 pop 6 symbols from stack.
0T2	+id$	reduce by 2 E→T — 1×2=2 pop 2 symbols
0E1	+id$	shift
0E1+6	id$	shift
0E1+6id5	$	reduce by 6 F→id 1×2=2 pop 2 symbols
0E1+6F3	$	reduce by 4 T→F 1×2=2 pop 2 symbols
0E1+6T9	$	reduce by 1 E→E+T 3×2=6 pop 6 symbols
0E1	$	Accept

* parser a input string using parsing table.

1) Initially push '0' as initial state on to the stack, and place the input string with '$' as end marker from the input buffer.

2) if 'S' is state on the top of the stack and 'a' is a symbol in input buffer pointed by a look ahead pointer

a) if action of [S, a] = shift j; then push a, push j on to the stack.

b) if action of [S, a] = reduce, A → β then pop 2*|β| symbols, if 'i' is on the top of the stack then

push A, then push goto[i, A] on the top of the stack.

c) if action [s,a] = accept, then hat the parsing process which indicates successful parsing.

1) consider the following grammar, construct the SLR parsing table for this grammar also the input string is a*a+a.

$E \rightarrow E+T/T$

$T \rightarrow TF/F$

$F \rightarrow F*/a/b$

2) construct the following collection of LR(0 item sets) and draw the goto graph of the following grammar.

$S \rightarrow SS/a/\varepsilon$

indicate the context conflicts in the various states of the SLR parser.

3) construct the LR(0) parsing table for the following grammar

$S \rightarrow cA/ccB$ prove that the given grammar is not an SLR[1].

$A \rightarrow cA/a$

$B \rightarrow ccB/b$

4) consider the grammar show that it is not SLR of (1) parsing

$S \rightarrow L=R/R$

$L \rightarrow *R/id$

$R \rightarrow L$

* **More powerfull LR parsers :-**

construct of CLR(1) parsing

1) the canonical set of items is the parsing technique in which lookA head symbol is generated while constructin the set of items.

Hence the collection of items is refered to as LR(1) The value '1' indicates that there is one lookA head symbol in set of items.

2) we follow the same steps in the SLR parsing Technique.

a) construct of the canonical set of items along with the lookA head.

b) Building canonical LR parsing table.

c) Parsing the input string using canonical LR parsing table

* Construction of canonical set of items along with look A head. LR(1) = LR(0) + look A head ($)

1) construct, For the grammar 'G' intially add $S' \rightarrow \cdot S$ in the set of item 'C'.

2) For each set of items of I_i in C and for each grammar symbol X add closure (I_i, X). this process is repeated by appling the goto (I_i, X). for each 'X' in I_i such that goto $[I_i, X)$ is not empty and not in C. This set of items as to be constructed until no more set of items are added to 'C'.

3) The closure() can be computed has for each item $A \rightarrow \alpha \cdot X\beta, a$ and $X \rightarrow \gamma$ and $b \in$ FIRST (β, a) such that $X \rightarrow \cdot\gamma$ and b is not in I then add $X \rightarrow \cdot\gamma, b$ to I

4) similary the goto function can be computed has for each item $[A \rightarrow \alpha \cdot X\beta, a]$ is in I and the rule $[A \rightarrow \alpha X \cdot \beta, a]$ is not in goto items then add $[A \rightarrow \alpha X \cdot \beta, a]$ to goto items.

This process repeated until no more items are set of added to the collection of C.

1) consider the following grammar, $S \rightarrow CC$
$C \rightarrow cC \mid d$

Sol: Given, $S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d$

Augmented grammar,
$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d.$

I_0: $S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

I_1: goto (I_0, S)
$S' \rightarrow S \cdot, \$$

I_2: goto (I_0, C)
$S \rightarrow C \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

I_3: goto (I_0, c)
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

I_4: goto (I_0, d)
$C \rightarrow d \cdot, c/d$

I_5: goto (I_2, C)
$S \rightarrow CC \cdot, \$$

I_6: goto (I_2, c)
$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

I_7: goto (I_2, d)
$C \rightarrow d \cdot, \$$

I_8: goto (I_3, C)
$C \rightarrow cC \cdot, c/d$

I_9: goto (I_6, C)
$C \rightarrow cC \cdot, \$$

Construction of canonical LR parser table :-

Algorithm:-

1) Initially construct set of items $C = \{I_0, I_1, \ldots I_n\}$ where C is the collection of set of LR(1) items for the grammar 'G'.

2) The parsing action are based on each item I_i. the actions are given as

a) if $[A \rightarrow \alpha . a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$, then create an entry an action table as action $[I_i, a] = \text{shift } j$

b) if there is a production $[A \rightarrow \alpha ., a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \rightarrow \alpha$. Here, 'A should not be 'S'.

c) if there is a production $[S' \rightarrow S., \$$ in $I_i]$ then action of $[i, \$]$ = accept.

3) The goto part of the LR table can be filled as the goto transaction of the states 'i' is consider for non-terminals only. if goto $[I_i, A) = I_j$ then goto $(I_i, A] = j$

4) All the entries donot defined by step (2) and step (3) are consider to be an error.

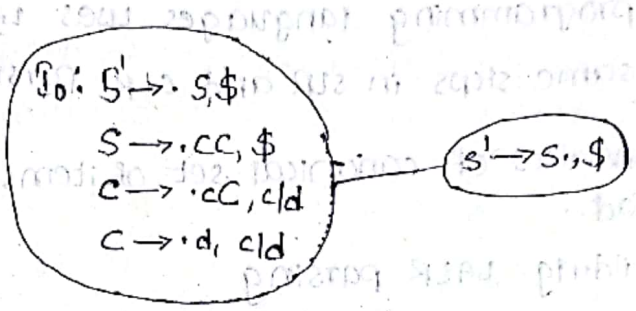
states	Action			goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

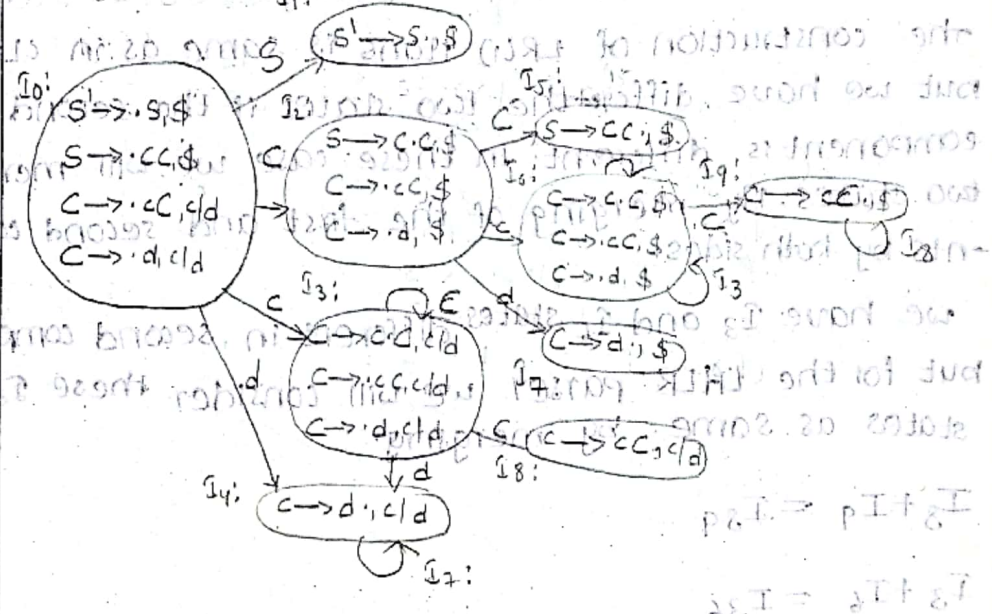
For reduction process, we consider the given grammar as

1) $S \rightarrow CC$
2) $C \rightarrow cC$
3) $C \rightarrow d$

and find out the dot symbol in the right most hand.

goto graph: deterministic finite automata.





1) construct a DFA whose states are the canonical collections of LR(1) items for the following augmented grammar.

1) $S \rightarrow A$
   $A \rightarrow BA \mid \epsilon$
   $B \rightarrow aB \mid b$

2) $S' \rightarrow S$
   $S \rightarrow CC$
   $C \rightarrow aC \mid d$

parse the string aabd

In

* LALR parsing :-

In this parser a look A head symbol is generated for each set of-

-The table updated by this method is similar in size then LR SO -

-The states of CLR and LALR are always same. most of the programming languages uses LALR parsers. we follow the same steps in SLR and CLR parsing.

1) construction of canonical set of items along with the look A head.

2) Building LALR parsing

3) parser the input string using the canonical LR parsing table.

construction of set of LR(1) items along with the LookAhead

the construction of LR(1) items is same as in CLR but we have differ the two states if the second component is different. In these case we will merge the two states. By merging of the first and second compone-nts by both sides.

we have I_3 and I_6 states different in second component but for the LALR parser we will consider these $I_3+I_6=2$ states as same. by merging.

$I_8 + I_9 = I_{89}$

$I_3 + I_6 = I_{36}$

$I_4 + I_7 = I_{47}$

construct of LALR parsing table. steps :-

1) construct the LR(1) set of items

2) Merge the two states I_i and I_j is I_{ij}

a) if the first component are matching then create a new state are replacing one of the older state as $I_{ij} = I_i \cup I_j$

3) The parsing actions are based on each item I_i and the actions are.

a) if $[A \rightarrow \alpha . a\beta, b]$ is in I_i and goto $[I_i, a] = I_j$

then create an entry in the action table as action [I, a] = shift j.

b) if there is a production [A → α·, a] in Ii then in action table action [Ii, a] = reduce by A → α and 'A' should not be S'.

c) If there is a production S' → S·, $ in Ii then action [i, $] = accept.

4) The goto path of the LR table are filled with, as the goto transitions for state 'i' is consider for non-terminal only.

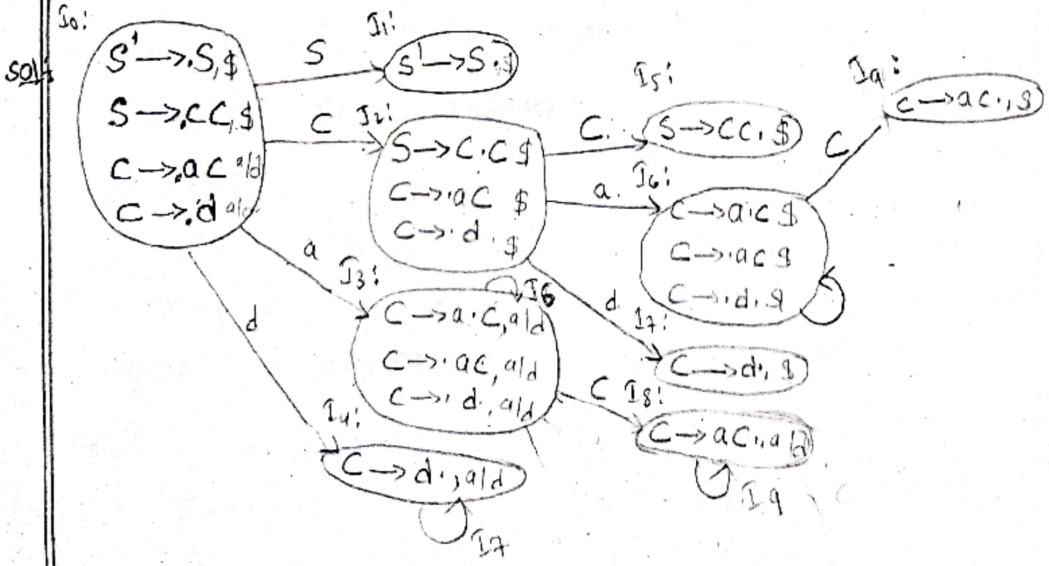
if goto [Ii, A] = Ij then goto [Ii, A] = j

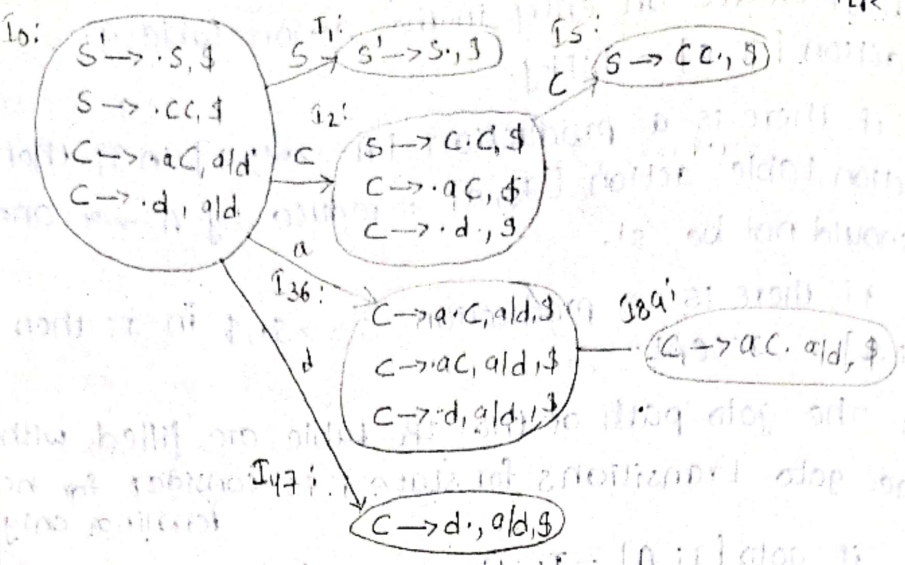
5) if parsing action confict occurs then algorithm fails to produce LALR parser and the grammar is not LALR (1). All the entries not defined by step 3, 4 are are consider to be an error.

1) Construct LALR gra parsing table for the given grammar

S → Aa
S → bAc
S → dc
S → bda.
A → d.

1) S' → S
S → CC
C → aC | d. CLR





states	action				
	a	d	$	S	C
0	s_{36}	s_{47}		1	2
1			accept		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

comparison of SLR and CLR and LALR.

	SLR	CLR	LALR
1)	It is the smallest in size	It is largest in size.	LALR and SLR are in same size
2)	It is the easiest method.	This method is most powerful than SLR and LALR.	The LALR method is applicable to wider.
3)	This method expo-ses less syntactic features than that of LR parser.	This method exposes less synthetic features than LR parser.	Most of the syntactic features of languages are expressed in LALR
4)	Every detection is no immediate SLR	Immediate error detection is done by the LR parser.	Error detection is not immediate in LALR
5)	It requires less time and space complexity.	It requires more time and space complexity	The time and space and complexity is more but effective method used for LALR parser directly.

* **Dangling else Ambiguity:**

* In the parsing methods if the ambiguous grammar is used then the conflicts occur and then we cannot parse the input string.

* if the 2 entries that can appear in the parsing table M[A,a] are for reduce action then Redue – conflict occurs

* if 1 entry is for shift and other for reduce action in M[A,a] then shift reduce conflict occurs.

using dangling else ambiguity:

* consider the grammar S → iSeS/iS/a. iSeS = if expression then else statement.

construct SLR parsing

S → iSeS

S → iS

S → a

Agmented grammar

I_0 : S → · iSeS

S → · iS

S → · a

S' → · S

I_1 : goto (I_0, S)

$S' \rightarrow S.$

I_2 : goto (I_0, i)

$S \rightarrow i.SeS$

$S \rightarrow .iSeS$

$S \rightarrow .iS$

$S \rightarrow .a$

I_3 : goto (I_0, a)

$S \rightarrow a.$

I_4 : goto (I_2, S)

$S \rightarrow iS.eS$

$S \rightarrow iS.$

I_5 : goto (I_4, e)

$S \rightarrow iSe.S$

$S \rightarrow .iS$

$S \rightarrow .a$

I_6 : goto (I_5, S)

$S \rightarrow iSeS.$

States	Action				goto
	i	a	e	$	S
0	S_2	S_3			1
1				accept	
2	S_2	S_3			4
3			r_3	r_3	
4			r_2/S_5	r_2	
5	S_2	S_3			6
6			r_1	r_1	

For the parsing table,

* FOLLOW(S) = {e, $}. so we build the SLR parser table as clearly in the above table Action (4,e) is a shift reduce conflict.

so we will now try to resolve it consider the input "iiaea" for parsing

stack	input	action
$0	iiaca$	shift
$0i2	iaca$	shift
$0i2i2	aca$	shift
$0i2i2a3	ca$	reduce by 3 S→a
$0i2i2S4	ca$	reduce by r_2 S→iS
$0i2S4	ca$	reduce by r_2 S→iS
$0i2S1	ca$	error!

The choice of r_2 for in Action(4,c) is not valid. Hence, we try to resolve by choosing the shift action.

stack	input	action
$0	iiaca$	shift
$0i2	iaca$	shift
$0i2i2	aca$	shift
$0i2i2a3	ca$	reduce by 3 S→a
$0i2i2S4	ca$	shift
$0i2i2S4c5	a$	shift
$0i2i2S4c5a3	$	reduce by 3 S→a
$0i2i2S4c5S3	$	reduce by 3 S→a
$0i2i2S4c5S6	$	reduce by 1 S→iSeS
$0i2S4	$	reduce by 2 S→iS
$0S1	$	Accept
$	$	accept

* Error recovery in LR parser :-

The LR parser is a table given parser method in which the blank entries treated as errors. when we compare any programs we get syntax errors: these error are usually denoted by user-friendly Error message.

consider the grammar $E \rightarrow E+E / E*E / (E) / id$ SLR?

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$.

Augmented grammar.

I_0: $E' \rightarrow \cdot E$

$E \rightarrow \cdot E+E$

$E \rightarrow \cdot E*E$

$E \rightarrow \cdot (E)$

$E \rightarrow \cdot id$

I_1: goto(I_0, E)

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot +E$

$E \rightarrow E \cdot *E$

I_2: goto(I_0, ()

$E \rightarrow (\cdot E)$

$E \rightarrow \cdot E+E$

$E \rightarrow \cdot E*E$

$E \rightarrow \cdot (E)$

$E \rightarrow \cdot id$.

I_3: goto(I_0, id)

$E \rightarrow id \cdot$

I_4: goto(I_1, +)

$E \rightarrow E+\cdot E$

$E \rightarrow \cdot E+E$

$E \rightarrow \cdot E*E$

$E \rightarrow \cdot (E)$

$E \rightarrow \cdot id$

I_5: goto(I_1, *)

$E \rightarrow E*\cdot E$

$E \rightarrow \cdot E+E$

$E \rightarrow \cdot E*E$

$E \rightarrow \cdot (E)$

$E \rightarrow \cdot id$

I_7: goto(I_4, E)

$E \rightarrow E+E \cdot$

I_6: goto(I_2, E)

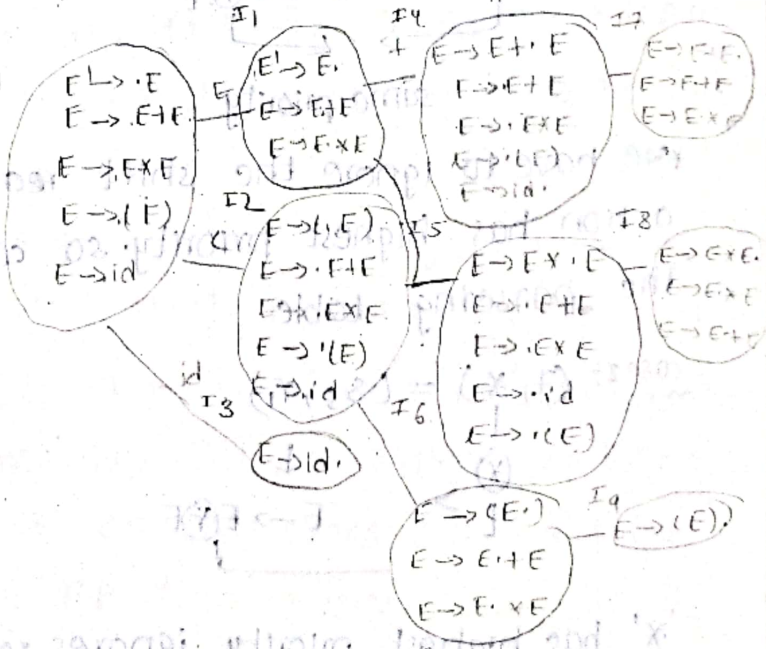
$E \rightarrow (E \cdot)$

I_8: goto(I_5, E)

$E \rightarrow E*E \cdot$

I_9: goto(I_6,))

$E \rightarrow (E) \cdot \cdot$

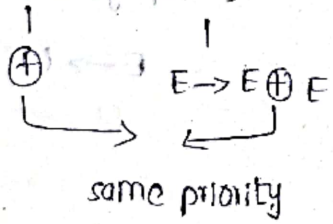


states	action						goto
	+	*	(	)	id	$	E
0	e_1	e_1	S_2	e_2	S_3	e_1	1
1	S_4	S_5	e_3	e_2	e_3	accept	
2	e_1	e_1	S_2	e_2	S_3	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	r_4
4	e_1	e_1	S_2	e_2	S_3	e_1	7
5	e_1	e_1	S_2	e_2	S_3	e_1	8
6	S_4	S_5	e_3	S_9	e_3	e_4	
7	S_4/r_1	S_5/r_1	r_1	r_1	r_1	r_1	r_1
8	S_4/r_2	S_5/r_2	r_2	r_2	r_2	r_2	r_2
9	r_3	r_3	r_3	r_3	r_3	r_3	r_3

FOLLOW(E) = {+, *, $,)}

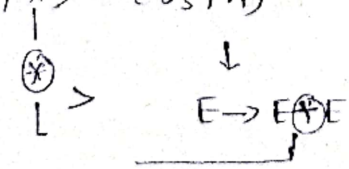
* in order to reduce the conflicts in SLR parsing. consider the conflict states i.e I_7 and I_8.

case1:- $(7, +) = (S_4, r_1)$



*we have to ignore the shift reaction and reduce action has highest priority. so cancel shift action in the parsing table.

case2:- $(7, *) = (S_5, r_1)$



'*' has highest priority, ignores reduce action, shift action has the * so cancel reduce action in the parsing table.
highest priority

case3 :- (8, +) = (s4, r2)

+ → < E → E + E

* ignore shift. cancel shift. Reduce has highest priority.

case4 :- (8, *) = (s5, r2)

* → E → E * E

some priority

* ignore shift, cancel shift in the parsing Table. use reduce.

* during error detection and recovering process. we have replaced some blank entries by particular reduction rules which means we are postponing the error detection for more reductions are done and the error will be introduced before any shift action takes place.

consider the set of items generated, for obtaining the error messages :-

1) I_0 : $E' \rightarrow .E$

which means that there is no symbol before dot. In such a case, if, +i * i $ comes in the Input string. then we say that operand is missing

2) I_1 :- $E' \rightarrow E.$

we ultimately reduce E → id. missing operator (or) may be unbalanced right paranthesis

3) I_6 :- goto(I_2, E) - E → (E.) if we reduce E → id. then the rule becomes E → (id] we expect ( ) -> closed paranthesis and if dollar ($) comes and then the error will be missing R.P from all these situations we conclude some Error messages are :-

e_1: These errors are in the state I_0, I_2, I_4 & I_5 which indicates that operand should appear before operator and Hence the above error message will be missing

e_2: These error is in ')' is in ; (colon) and the states I_0, I_2, I_4, I_5 indicates unbalancing right paranthesis Hence the error message will be unbalancing right paranthesis

e_3: The operator is expected in this case of error has it is state I_1 (or) I_6. the error message will be missing operator.

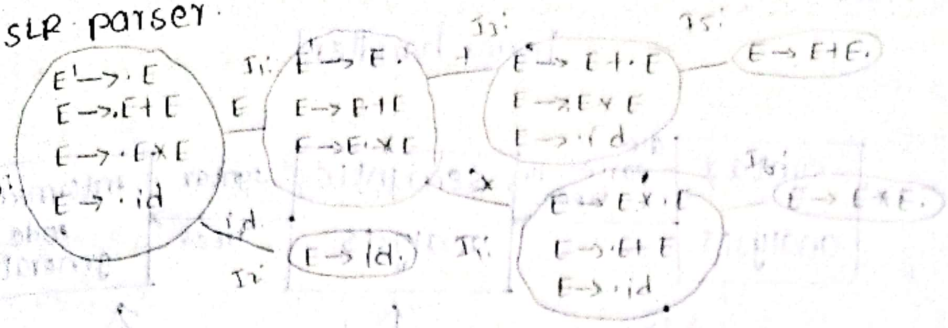
e_4: These are occurs ie errors occurs at state I_6 in dollor column this state (ex) at the end of expression Hence the error message will be missing right paranthesis.

id+)$ parser the string in the above table

stack	input	action
$0	id+)$	shift
$0id3	+)$	reduce by r_4 $E \to id$ 1x2=2 pop
$0E1	+)$	shift
$0E1+4	)$	Error, e_2, action is remove the right paran-thesis from the input
$0E1+4	$	Error e_1, action is push id on to the stack and change the state to 3
$0E1+4id3	$	reduce r_4 pop from the top of the stack
$0E1+4E7	$	reduce by R_1, pop 6 symbols from the stack $E \to E+E$
$0E1	$	accept

* Handling Ambiguous grammar using LR parsing:

consider the grammar E → E+E | E*E | id. construt SLR parser



states	ACTION				goto
	+	*	id	$	E
0			S_2		1
1	S_3	S_4		Accept	
2	r_3	r_3		r_3	
3			S_2		5
4			S_2		6
5	r_1/S_3	$r_1 S_4$		r_1	
6	$r_2 S_3$	$r_2 S_4$		r_2	

stack	input	action
$0	id+id*id$	shift by 2
$0id2	+id*id$	reduce by 3 E→id (pop 2 elements)
$0E1	+id*id$	shift by 3
$0E1+3	id*id$	shift by 2
$0E1+3id2	*id$	reduce by 3 E→id
$0E1+3E5	*id$	shift by 4
$0E1+3E5*4	id$	shift by 2
$0E1+3E5*4id2	$	reduce by 3 E→id
$0E1+3E5*4E6	$	reduce by 3 E→E*E (3×2=6)
$0E1+3E5	$	reduce by 2 =4
$0E1	$	Accept