Compiler Design
Chapter 2: Parsing

GATE CS Lectures
by Monalisa

- **Section 7: Compiler Design(≅5 mark)**
- Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation . Local optimization, Data flow analyses: constant propagation, liveness analysis, common subexpression elimination.

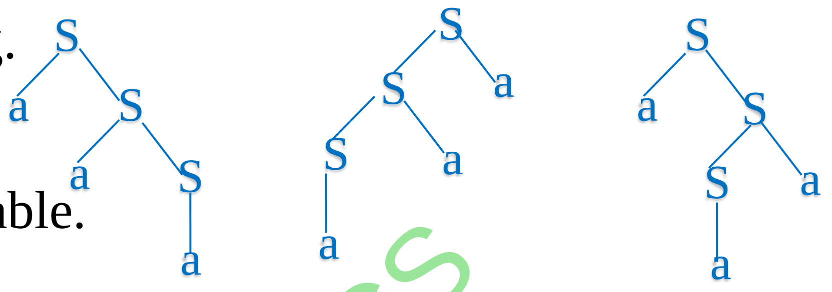
- Chapter 1: Introduction to Compiler [Language processing System ,Compiler ,Phases of Compiler , Lexical Analysis]
- Chapter 2: Parsing [Syntax Analysis , CFG, Ambiguous Grammar , Recursive Grammar ,Left Factoring ,Top down parser : LL(1),FIRST & FOLLOW , Bottom up parser : shift-reduce parsing ,LR(0),SLR(1),CLR(1), LALR(1), Operator Precedence grammar]
- Chapter 3: SDT , Code optimization &Runtime environments

Syntax Analysis

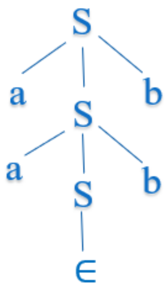
- Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.
- The parsing technique is implemented by CFG.
- **Functions of the parser :**
- 1. It verifies the structure generated by the tokens based on the grammar.
- 2. It constructs the parse tree.
- 3. It reports the errors.
- 4. It performs error recovery.
- Parser can detect errors during construction of syntax tree and grammar of language.
- Ex: such as an arithmetic expression with unbalanced parentheses.
- Parser cannot detect errors such as:
- 1. Variable re-declaration
- 2. Variable initialization before use.
- 3. Data type mismatch for an operation.
- The above issues are handled by Semantic Analysis phase

- **Error-Recovery Strategies**
- 1. Panic mode, 2. Phrase level, 3. Error productions, 4. Global correction
- **Types of parser :**
- There are two types of parsers for grammars: topdown and bottom-up.
- Top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from left to right,one symbol at a time.
- **CFG:** Finite Set of rules which are used to generate the string is called as grammar.
- It has 4 tuples G=(V,T,P,S)
- Classification of Grammar
- Grammar can be classified in two ways
- 1.Based on Derivation tree
  - Ambiguous Grammar
  - Unambiguous Grammar
- 2.Based on number of string
  - Recursive Grammar
  - Non Recursive Grammar

- <u>Ambiguous Grammar</u>: The grammar is said to be ambiguous if more than one parse tree exist for at least one string.
- Ex:S→aS|Sa|a ,w=aaa

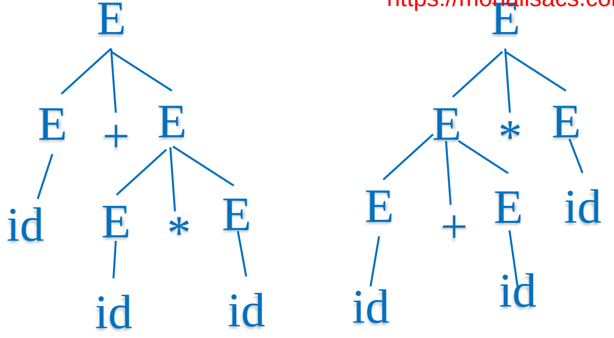


- Ambiguity of CFG is undecidable.
- <u>Unambiguous Grammar</u> :



- The grammar is said to be unambiguous if there exist unique parse tree for every input string. Ex:S →aSb| ∈,w=aabb
- No algorithm exist to convert ambiguous grammar to unambiguous grammar except operator grammar.
- The Ambiguous grammar which can't be converted to unambiguous is called inherent Ambiguous grammar .
- Operator |Expression grammar can be converted to unambiguous by redefine grammar using associativity & operator precedence.
- Precedence: id,bracket > ^ > *,| > + , -
- ^ is right associative,*,|,+,- are left associative.

- **Removal of Ambiguity from Expression Grammar**
- E→E+E | E-E | E*E | E^E | (E) | id
- W=id + id * id
- This is a ambiguous grammar.
- In parse tree highest precedence operator is always at lower level than lower precedence.
- It grow left side if operator is left associative & grow right if it is right associative
- Lets rewrite unambiguous grammar



Operator	Associativity	Variable	Grammar
+,-	Left	E	E→E+F | E-F|F
*	Left	F	F→F*G|G
^	Right	G	G→H^G|H
( ),id		H	H→(E)|id

- Find associativity & operator precedence of all the operator ?
- S→S@W |W
- W→W#Y|Y
- Y→Y$A|A
- A→B%A|A&B|id
- Sol:@<#<$<%,&,id
- Left associative @,#$,&
- Right associative %
- GATE2000-21, ISRO2015-24:Given the following expression grammar:
- E → E * F | F + E | F
- F → F - F | id
- which of the following is true?
- (A) * has higher precedence than +
- (B) – has higher precedence than *
- (C) + and - have same precedence
- (D) + has higher precedence than *
- Ans: (B) – has higher precedence than *

- Recursive Grammar :If at least one production contain same variable both at LHS and RHS. Ex:S →aSb| ∈
- Non Recursive Grammar :If no Production contain same variable both at LHS and RHS
- Ex:S→aA|b,A→a
- Non Recursive →Finite Language
- Recursive →Infinite Language
- **Types of Recursion:**
- 1.Left Recursion
- The Grammar is said to be left recursive if left most variable of RHS is same as variable of LHS.
- Ex:A→Aa|b
- 2.Right Recursion
- The Grammar is said to be right recursive if the right most variable of RHS is same as variable of LHS.
- Ex: A→aA|b
- 3.General Recursion
- The Grammar is said to be general recursive if it is neither left nor right recursive . Ex :A→aAb|b

- If the grammar is left recursive then parser may goes to infinite loop.
- To avoid looping we need to convert left recursive grammar to right recursive grammar.
- **Conversion of LRG→RRG:**
- $1. A \rightarrow A\alpha|\beta$ $\Rightarrow A \rightarrow \beta A'$
- $\beta\alpha^*$ $\Rightarrow A' \rightarrow \alpha A'|\epsilon$
- $2. A \rightarrow A\alpha_1|\ A\alpha_2|\ldots\ldots\ A\alpha_n|\ \beta$ $\Rightarrow A \rightarrow \beta A'$
- $\Rightarrow A' \rightarrow \alpha_1 A'|\ \alpha_2 A'|\ldots..\ \alpha_n A'|\epsilon$
- $3. A \rightarrow A\alpha|\ \beta_1|\beta_2|\ldots\ldots\ \beta_n$ $\Rightarrow A \rightarrow \beta_1 A'|\ \beta_2 A'|\ldots\ldots.\ \beta_n A'$
- $\Rightarrow A' \rightarrow \alpha A'|\epsilon$
- $4. A \rightarrow A\alpha_1|\ A\alpha_2|\ldots A\alpha_n|\ \beta_1|\beta_2|\ldots\beta_n$ $\Rightarrow A \rightarrow \beta_1 A'|\ \beta_2 A'|\ldots\ldots.\ \beta_n A'$
- $\Rightarrow A' \rightarrow \alpha_1 A'|\ \alpha_2 A'|\ldots..\ \alpha_n A'|\epsilon$
- Ex 1: $A \rightarrow Aab|c$ $\Rightarrow A \rightarrow cA'$
- $\Rightarrow A' \rightarrow ab\ A'|\ \epsilon$
- Ex 2: $S \rightarrow SaS|bS|a$ $\Rightarrow S \rightarrow aS'|bSS'$
- $\Rightarrow S' \rightarrow aS\ S'|\epsilon$
- Ex 3: $E \rightarrow E+E|E*E|(E)|id$ $\Rightarrow E \rightarrow idE'|(E)E'$
- $\Rightarrow E' \rightarrow +EE'|*EE'|\epsilon$

- **Grammar with common prefix:**
- If more than one production start with same sequence of grammar symbol then the grammar is called as Grammar with common prefix.
- Ex: $A \to aAa|aAb|\epsilon$
- Left Factoring:
- Left factoring is a grammar transformation that is useful for top-down parsing.
- The process of removing common prefix or eliminating nondeterminism is called as left factoring.
- $A \to \alpha\beta_1|\alpha\beta_2|\alpha\beta_3$ $\Rightarrow A \to \alpha A'$
  $\Rightarrow A' \to \beta_1|\beta_2|\beta_3$
- Ex 1: $A \to ab|ac|ad|ae$ $\Rightarrow A \to aB$
- $\Rightarrow B \to b|c|d|e$
- Ex 2: $E \to E+E|E*E|(E)|id$ $\Rightarrow E \to EE'|(E)|id$
- $\Rightarrow E' \to +E|*E$
- Ex 3: $S \to SaSbS/SbSaS/\epsilon$ $\Rightarrow S \to SS'|\epsilon$
- $\Rightarrow S' \to aSbS|bSaS$
- The grammar with both left & right recursive is always ambiguous.
- Left factoring will not remove ambiguity.

- Classification of Parser

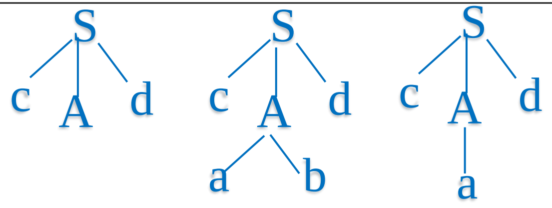
1) Top down parser
2) Bottom up parser

- **Top down Parser:**
- The process of constructing parse tree starting with root & going upto the leaves or children is called top down parsing.
- Top down parser simulate left most derivation.
- It takes the grammar which is free from ambiguity , left recursion & common prefix.
- Top down parser is very slow . Average time complexity $O(n^3)$,n=number of token.
- Types of top-down parsing :

1) Recursive descent parsing/Bruteforce Technique [with backtracking]
2) Predictive parsing(LL1) [without backtracking]

- **Recursive descent parsing:**
- This parsing method may involve backtracking, that is, making repeated scans of the input.
- Backtracking is costly. Debugging is very difficult.

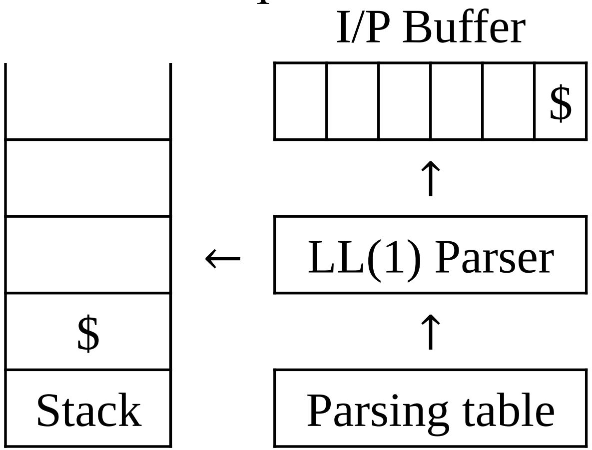
- Ex: Consider the grammar S→cAd
- A→ab|a
- input string w=cad.



- Step1:Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w.
- Step2: The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’.
- Expand A using the first alternative.
- Step3: The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’.
- But the third leaf of tree is ‘b’ which does not match with the input symbol ‘d’ Hence discard the chosen production and reset the pointer to second position.
- This is called **backtracking**.
- Step4: Now try the second alternative for A .
- If matching doesn’t occur then match with alternative.
- If it match at least one alternative then parsing is successful else fail.

- **Predictive parsing:**
- No backtracking.
- Grammar must be free from ambiguity , left recursion & common prefix.
- **LL(1) Parser:**
- The first "L" : scanning the input from left to right,the second "L":producing a leftmost derivation, and the "1" :using one input symbol of look ahead at each step to make parsing action decisions.
- The current parsing symbol is called look ahead symbol.
- Block Diagram of LL(1) Parser:
- LL(1) parser consist of 3 component

1) Input Buffer
2) Parse stack
3) Parse table



- **LL(1) Grammar:**
- The grammar for which LL(1) parser can be constructed is called LL(1) grammar.
- The grammar is LL(1) if its parse table is free from multiple entries.
- **Function used to construct LL(1) parse table**
- 1.FIRST(X), 2.FOLLOW(A) [X∈V+T,A ∈V]

- **FIRST(X):**
- FIRST (x) is set of all terminals that may begin any sentential form or production.
- The first terminal which can be derived from a variable in process of derivation.
- Rules for FIRST( ):

1) If X is terminal, then FIRST(X) is {X}.
2) If X→ ε is a production, then add ε to FIRST(X).
3) If X is non-terminal and X → aα is a production then add a to FIRST(X).
4) If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST (X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1) ,…., FIRST(Yi-1); that is, Y1 ,..Yi-1 => ε.If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).
5) If X → Y & both are non-terminal then FIRST(X)=FIRST(Y).

- Ex 1:A →a|b|∈
- FIRST(A)={a,b,∈}

- Ex 2:S→aSb|bSa|∈
- FIRST(S)={a,b,∈}

- Ex 3:S→aA|bB
- A →aA|b
- B →b | ∈
- FIRST(S) ={a,b}
- FIRST(A)={a,b}
- FIRST(B)={b,∈}

- Ex 4: $S \rightarrow Aa$
- $A \rightarrow b \mid \epsilon$
- FIRST(S) ={a,b}
- FIRST(A)={b,∈}

- Ex 7: $S \rightarrow ABCDE$
- $A \rightarrow a|\epsilon$
- $B \rightarrow b|\epsilon$
- $C \rightarrow c|\epsilon$
- $D \rightarrow d$
- $E \rightarrow e|\epsilon$
- FIRST(S) ={a,b,c,d}
- FIRST(A)={a,∈}
- FIRST(B)={b,∈}
- FIRST(C)={c,∈}
- FIRST(D)={d}
- FIRST(E)={e,∈}

- Ex 5: $S \rightarrow AB$
- $A \rightarrow a \mid \epsilon$
- $B \rightarrow b|c$
- FIRST(S) ={a,b,c}
- FIRST(A)={a,∈}
- FIRST(B)={b,c}

- Ex 8: $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'|\epsilon$
- $F \rightarrow (E)|id$
- FIRST(E) ={(,id}
- FIRST(E′)={+,∈}
- FIRST(T)={(,id}
- FIRST(T′)={*,∈}
- FIRST(F)={(,id}

- Ex 6: $S \rightarrow AB$
- $A \rightarrow aA \mid \epsilon$
- $B \rightarrow bB|\epsilon$
- FIRST(S) ={a,b,∈}
- FIRST(A)={a,∈}
- FIRST(B)={b, ∈}

- **FOLLOW(A):**
- A terminal which can follow a variable during process of derivation.
- FOLLOW(A) is the set of all terminals that may followed to the right of A in any production or any sentential form.
- **Rules for FOLLOW( ):**

1) If S is a start symbol, then FOLLOW(S) contains \$.
2) If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in FOLLOW(B).
3) If there is a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then FOLLOW(B)= FOLLOW(A) ∪ FIRST(β)- ε.
4) If $S \rightarrow \alpha A$ or $S \rightarrow A$ then FOLLOW(A)= FOLLOW(S)

- Ex-1: $A \rightarrow a|\epsilon$
- FOLLOW(A)={\$}
- Ex-2: $A \rightarrow A(A)|\epsilon$
- FOLLOW(A)={\$,(,)}
- Ex-3: $S \rightarrow aA$
- $A \rightarrow aAb|Sa|\epsilon$
- FOLLOW(S)={\$,a}
- FOLLOW(A)={\$,a,b}

- Ex-4:
- S →aAB
- A→aAc|∈
- B→bB|a
- FOLLOW(S)={$}
- FOLLOW(A)={a,b,c}
- FOLLOW(B)={$}
- Ex-6:
- S→ABCDE
- A →a|∈
- B →b|∈
- C →c|∈
- D →d
- E →e|∈
- FOLLOW(S) ={$}
- FOLLOW(A)={b,c,d}
- FOLLOW(B)={c,d}
- FOLLOW(C)={d}
- FOLLOW(D)={e,$}
- FOLLOW(E)={$}

- Ex-5:
- S →AB
- A→aA|∈
- B→bB|∈
- FOLLOW(S)={$}
- FOLLOW(A)={b,$}
- FOLLOW(B)={$}
- Ex-7:
- E→TE′
- E′→+TE′ |∈
- T→FT′
- T′→∗FT′|∈
- F→(E)|id
- FOLLOW(E) ={),$}
- FOLLOW(E′)={),$}
- FOLLOW(T)={+,),$}
- FOLLOW(T′)={+,),$}
- FOLLOW(F)={*, +,),$}

Construction of predictive / LL(1) Parse Table:

- For each production $A \rightarrow \alpha$ of the grammar , do the following:

1) For each terminal a in FIRST(A) add $A \rightarrow \alpha$ to M[A,a].
2) If ∈ is in FIRST(A),then for each terminal b in FOLLOW(A),add $A \rightarrow \alpha$ to M[A,b]. If ∈ is in FIRST(A) and $ is in FOLLOW(A),add $A \rightarrow \alpha$ to M[A,$] as well.

- The grammar G is LL(1) if predictive parse table is free from multiple entries.
- Ex-1: A →aA |b

	FIRST	FOLLOW
A	a,b	$

	a	**b**	**$**
A	A →aA	A →b	

- Ex-2:
- S→aA|Bb
- A →aA|b
- B →bB|∈

	FIRST	FOLLOW
S	a,b	$
A	a,b	$
B	b,∈	b

	a	**b**	**$**
S	S→aA	S→Bb	
A	A →aA	A →b	
B		B→bB B →∈	

- Since there are more than one production, the grammar is not LL(1) grammar.
- All ∈ production should be placed FOLLOW of LHS variable.

- Ex-3:
- S→Aa|bB
- A →aA|c
- B →bB|∈

	FIRST	FOLLOW
S	a,b,c	$
A	a,c	a
B	b, ∈	$

	a	**b**	**c**	**$**
S	S→Aa	S→bB	S→Aa	
A	A→aA		A→c	
B		B →bB		B →∈

- Ex-4:
- S→Aa|bB
- A →aA|Bb|d
- B →SB|b

	FIRST	FOLLOW
S	a,b,d	$,a,b,d
A	a,b,d	a
B	a,b,d	b,a,d,$

	a	**b**	**d**	**$**
S	S→Aa	S→bB S→Aa	S→Aa	
A				
B				

- If any terminal is repeated in FIRST(A) then the grammar is not LL(1)
- Ex-5: S→(S)| ∈

	FIRST	FOLLOW
S	(, ∈	$,)

	(	**)**	**$**
S	S→(S)	S→∈	S→∈

- Ex-6:
- $E \to TE'$
- $E' \to +TE' | \epsilon$
- $T \to FT'$
- $T' \to *FT' | \epsilon$
- $F \to (E) | id$

	FIRST	FOLLOW
E	(,id	),$
E′	+, ∈	),$
T	(,id	+,),$
T′	*, ∈	+,),$
F	(,id	+,*,),$

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

Short Cut method for testing LL(1) Grammar:

- 1.If the grammar is free from ∈ production & for every production of the form $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | \ldots\ldots \alpha_n$ the set FIRST(α_1)∩ FIRST(α_2)….. FIRST(α_n) =∅ then grammar is LL(1).
- Ex-4
- S →Aa|bB
- A →aA|Bb|d
- B →SB|b
- 2.If the grammar contain ∈ production & for every production of the form $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | \ldots\ldots \alpha_n$ the set FIRST(α_1)∩ FIRST(α_2)….. FIRST(α_n) =∅ & for every production $A \rightarrow \alpha |$ ∈ then FIRST(α)∩FOLLOW(A) =∅ then grammar is LL(1).
- Ex-2
- S →aA|Bb
- A →aA|b
- B →bB|∈
- 3.Every ambiguous grammar is not LL(1) .
- 4.Every left recursive grammar is not LL(1).
- 5.Every grammar having common prefix is not LL(1).

- Ex-1: A →ab|bc|d
- Ex-2: A →ab|ac|d
- Ex-3: S →aSb |∈
- Ex-4: S →aSb|bSa|∈
- Ex-5: S →aA |BbA
- A →aA|b
- B →bB|∈
- Ex-6: S →Aa|bB
- A →bA|dB|∈
- B →aBb|d
- Ex-7: S →aSbS|bSaS| ∈
- Ex-8: S →aABb
- A →c|∈
- B →d|∈

- Ex-1: {a}∩{b}∩{d} LL(1)
- Ex-2: {a}∩{a}∩{d} not LL(1)
- Ex-3: {a}∩{$,b} LL(1)
- Ex-4: {a}∩ {b} ∩{a,b,$}not LL(1)
- Ex-5: {a}∩{b}
- {a}∩{b}
- {b}∩{b} Not LL(1)
- Ex-6: {a,b,d} ∩{b} not LL(1)
- Ex-7: {a} ∩{b} ∩{a,b,$} not LL(1)
- Ex-8:
- {c} ∩{d,b}
- {d} ∩{b} LL(1)

- LL(1) Parsing process using stack:

1) Push the start symbol of the grammar into the stack.
2) Compare the topmost symbol of stack to look ahead symbol.
3) If matching occurs (x=a≠$) then pop off & increment the input pointer
4) If matching doesn't occur (x ≠ a ≠$) then perform the push operation again compare the top of the stack with look ahead symbol.
5) After reading the complete string if the stack is empty(x=a=$ }then parsing is successful .

- x= top of stack symbol,a=current input symbol,$=end marker.
- Ex-1:S→(S)| ∈ ,w=(())

Stack	i/p string	Action
$	(())$	Push(S)
$S	(())$	Push(S→(S))
$)S(	(())$	Pop
$)S	())$	Push(S→(S))
$))S(	())$	Pop
$))S	))$	Push(S→∈)
$))	))$	Pop
$)	)$	Pop
$	$	accept

	(	)	$
S	S→(S)	S→∈	S→∈

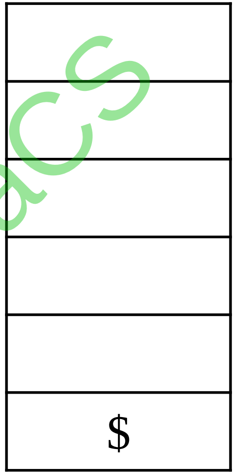
$

- Number of different push operation =3

- Ex-2:
- S →AA
- A →aA|b
- W=abab

	a	b	$
S	S →AA	S →AA	
A	A →aA	A →b	

Stack	**i/p string**	**Action**
$	abab$	Push(S)
$S	abab$	Push(S →AA)
$AA	abab$	Push(A →aA)
$AAa	abab$	Pop
$AA	bab$	Push(A →b)
$Ab	bab$	Pop
$A	ab$	Push(A →aA)
$Aa	ab$	Pop
$A	b$	Push(A →b)
$b	b$	Pop
$	$	Accept



- Number of different push operation =4
- Minimum number of distinct push operation=n[without ∈ production]
- n=number of tokens
- In case of ∈ Minimum number of distinct push operation= n-1

Bottom up Parser:

- Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
- Ex: $E \rightarrow E+T|T$, $T \rightarrow T*F|F$, $F \rightarrow (E)|id$
- W=id*id
- A general type of bottom-up parser is a shift-reduce parser.
- The class of grammars for which shift-reduce parsers can be built, the LR grammars.
- Bottom up parser simulates reverse of right most derivation.
- Bottom up parser is more powerful than top down parser.
- Average time complexity $O(n^3)$,n=number of token.
- Bottom up parser takes unambiguous grammar for LR parsing.

Reductions :

- Bottom-up parsing as the process of "reducing" a string w to the start symbol of grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production.
- Ex:A sequence of reductions id*id, F*id, T*id, T*F, T, E.
- A reduction is the reverse of a step in a derivation .

Handle Pruning:

- A **handle** of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production is possible.
- The process of finding & reducing the handle is called as **handle pruning**.

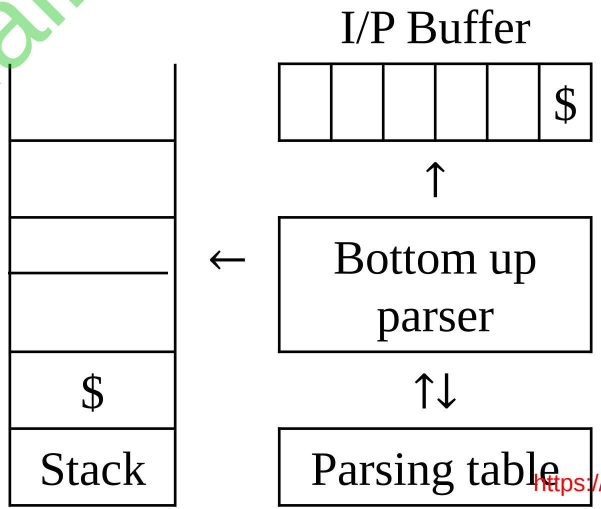
Sentential form	Handle	Reducing Production
id_1*id_2	id_1	F→id
$F*id_2$	F	T→F
$T*id_2$	id_2	F→id
T*F	T*F	T→T*F
T	T	E→T

Grammar:
E→E+T|T
T→T*F|F
F→(E)|id

Block diagram of bottom up parser:

- It consist of 3 component

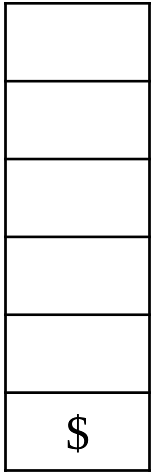
1) Input buffer
2) Parse stack
3) Parse table



- **Shift-Reduce Parsing:**
- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- We use $ to mark the bottom of the stack and also the right end of the input.
- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string of grammar symbols on top of the stack.
- It then reduces to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty
- There are four possible actions a shift-reduce parser:
  (1) shift, (2) reduce, (3) accept, (4) reject
- Shift: Shift the next input symbol onto the top of the stack.
- Reduce: The parser replaces the handle within a stack with a variable.
- Accept: At the end of parsing if the stack contains only the start symbol then the string is accepted and parsing is successful.

- Reject: At the end of parsing if the stack contains anything other than start symbol then the string is reject and parsing is unsuccessful.
- Ex-1:
- S →AA
- A →aA|b
- W=abab

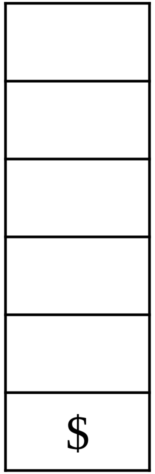
Stack	i/p string	Action
$	abab$	Shift
$a	bab$	Shift
$ab	ab$	Reduce(A→b)
$aA	ab$	Reduce(A→aA)
$A	ab$	Shift
$Aa	b$	Shift
$Aab	$	Reduce(A→b)
$AaA	$	Reduce(A→aA)
$AA	$	Reduce(S→AA)
$S	$	Accept



- Number of different reduce operation =3
- Maximum number of reduce moves that can be taken by Shift reduce parser / bottom up parser for a grammar with no ∈ and unit production to parse a string of n token is n-1.
- |abab|=4
- Number of reduce operation =4-1=3

- Ex-2:
- $E \rightarrow E+T|T$
- $T \rightarrow T*F|F$
- $F \rightarrow (E)|id$
- $W = id_1 * id_2$

Stack	i/p string	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



- Number of different reduce operation =4
- Maximum number of reduce moves =n-1[without unit, ∈ production]
- $| id_1*id_2 |=3-1=2$
- In case of unit production n-1+number of unit production.
- 2+2[E→T, T→F]=4

Conflicts During Shift-Reduce Parsing

- ***1. Shift-reduce conflict***: The parser cannot decide whether to shift or to reduce.
- Consider grammar: E→E+E | E*E| id ,input :id+id*id

Stack	i/p string	Action	Stack	i/p string	Action
$E+E	*id$	Reduce E→E+E	$E+E	*id$	Shift
$E	*id$	Shift	$E+E*	id$	Shift
$E*	id$	Shift	$E+E*id	$	Reduce E→id
$E*id	$	Reduce E→id	$E+E*E	$	Reduce E→E*E
$E*E	$	Reduce E→E*E	$E+E	$	Reduce E→E+E
$E	$	Accept	$E	$	Accept

- ***2.Reduce-reduce conflict:*** The parser cannot decide which of several reductions to make. Consider grammar:M →R+R|R+c , R →c,input :c+c

Stack	i/p string	Action	Stack	i/p string	Action
$	c+c$	Shift	$	c+c$	Shift
$c	+c$	Reduce R→c	$c	+c$	Reduce R→c
$R	+c$	Shift	$R	+c$	Shift
$R+	c$	Shift	$R+	c$	Shift
$R+c	$	Reduce R→c	$R+c	$	Reduce M→R+c
$R+R	$	Reduce M→R+R	$M	$	Accept
$M	$	Accept			

- **Classification of bottom up parser**

1) Operator Precedency parser
2) LR Parser
   - LR(0) item: LR(0), SLR(1)
   - LR(1) item: CLR(1),LALR(1)

- **Operator-precedence parsing**
- An efficient way of constructing shift-reduce parser is called operator precedence parsing .
- Operator-grammar: These grammars have the property that no production on right side is ε or has two adjacent non terminals.
- Example: Consider the grammar: **E → EAE | (E) | -E | id**

  A → + | - | * | / | ^
- The right side EAE has three consecutive non-terminals, so not operator grammar.
- The grammar can be written as follows:
- **E → E+E | E-E | E*E | E/E | E^E | -E | id**
- Operator grammar can be ambiguous or unambiguous.

- In operator grammar every terminal is operator.
- Only terminal are used for operator precedency grammar
- Operator grammar work on precedency & associativity property.
- Operator precedence grammar :the operator grammar for which an operator precedency parser can be constructed is called operator grammar.
- **Operator precedence relations**:
- There are three precedence relations namely <. , = , .>

1) a < . b :a yields precedence to b.b reduce before a.
2) a = b :a has the same precedence as b. reduce according to associativity.
3) a . > b :a takes precedence over b.a reduce before b.

- **Rules for constructing precedence parse table:**
- Let θ_1 & θ_2 be two operations.

1) If θ_1 has higher precedence than θ_2, then make θ_1 .> θ_2 and θ_2 <. θ_1.
2) If θ_1 and θ_2 , are of equal precedence, then make θ_1 .> θ_2 and θ_2 .> θ_1 if operators are left associative , θ_1 <. θ_2 and θ_2 <. θ_1 if right associative.

- ‘id’ > ‘^’ is right-associative > ‘*’ , ‘/’ left-associative and > ‘+’ ,‘-’ left-associative >$

- **Operator precedence parsing algorithm:**
- Let a is the top of stack & b is the look ahead symbol.

1) If a <. b ,or a = b then shift b onto the stack; advance ip to the next input symbol;
2) Else if a . > b then /*reduce*/
   repeat {pop the stack until the top stack terminal is related by <.to the terminal most recently popped}
3) If a=b=$ parsing successful .

- Stack implementation of operator precedence parsing:
- Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm.
- The initial configuration of an operator precedence parsing is stack $,input w $
- Advantages of operator precedence parsing:

1) It is easy to implement.
2) Once an operator precedence relation is made between all pairs of terminals of a grammar ,the grammar can be ignored.

- Disadvantages of operator precedence parsing:

1) It is hard to handle tokens like the minus sign (-) which has two different precedence.
2) Only a small grammar can be parsed using operator-precedence parser.

- Consider Grammar:
- **E → E+E | E-E | E*E | E/E | E^E | (E) | id**
- Input string : id + id * id

Stack	i/p string	Action
$	<. id+id*id$	Shift
$id	.> +id*id$	Pop
$	<. +id*id$	Shift
$+	<. id*id$	Shift
$+id	.> *id$	Pop
$+	<. *id$	Shift
$+*	<. id$	Shift
$+*id	.> $	Pop
$+*	.> $	Pop
$+	.> $	Pop
$	$	Accept

	+	-	*	/	^	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
-	.>	.>	<.	<.	<.	<.	<.	.>	.>
*****	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
^	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>
(	<.	<.	<.	<.	<.	<.	<.	=	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.		

- **Introduction to LR Parsing:**
- An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing.
- The ‘L’ is for **left-to-right** scanning of the input, the ‘R’ for constructing **a rightmost derivation in reverse**, and the ‘k’ for the number of input symbols .
- **Advantages of LR parsing:**
- It recognizes all programming language constructs for which CFG can be written.
- It is an efficient non-backtracking shift-reduce parsing method.
- It detects a syntactic error as soon as possible.
- A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive/LL(1) parser.
- **Drawbacks of LR method:**
- It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.
- Types of LR parsing method:
- LR(0),SLR(1)- Simple LR ,Easiest to implement, least powerful.
- CLR(1)- Canonical LR ,Most powerful, most expensive.
- LALR(1)- Look-Ahead LR
- Intermediate in size and cost between SLR & CLR.

- **Construct LR Parse Table:**

1. Obtain the augmented grammar.
2. Construct the canonical collection of LR items.
3. Draw the LR Automata.
4. Construct the parse table from LR Automata.

- Augmented grammar:
- 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 grammar which is obtained by adding 1 more production before start symbol is called as augmented grammar.
- LR(0) item:
- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
- An LR(0) item of a grammar G is a production of G with a dot at some position of the body of RHS.
- A →XYZ
- LR(0) items A →.XYZ , A →X.YZ , A →XY.Z , A →XYZ.
- A →X.YZ indicates that we have just seen on the input a string derivable from X and next to see a string derivable from YZ.
- Item A →XYZ. indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

- The production $A \rightarrow \epsilon$ generate only one item , $A \rightarrow .$
- **Function used to generate LR(0) item**

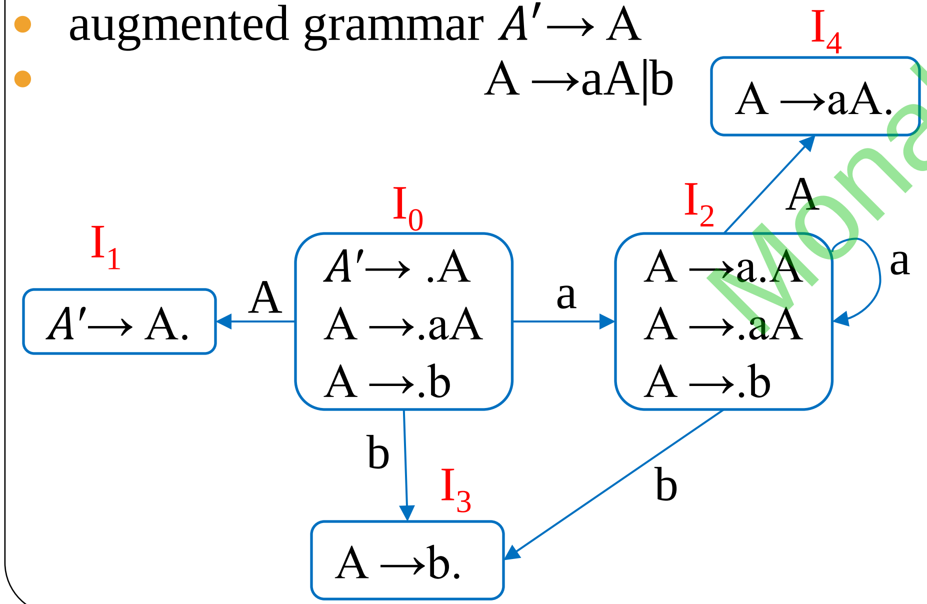
1) Closure(I) [I set of items]
2) Goto(I,x) [x grammar symbol]

- **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 two rules:

1) Initially ,add every item in I to CLOSURE(I).I_0=CLOSURE(S'→**.**S)
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 CLOSURE(I),if it is not already there . apply this rule until no more new items can be added to CLOSURE(I).

- **Goto (I,x)**
- Goto $(A \rightarrow \alpha.x\beta, x) = (A \rightarrow \alpha x.\beta)$
- **Structure of the LR Parsing Table:**
- The parsing table consist of two functions: 1.ACTION, 2.GOTO.
- ACTION function takes as argument a state I and a terminal or $.
- ACTION part contains shift & reduce of terminal.
- If x is a terminal & goto (I,x) =I_j then place S_j in ACTION.
- If parser accepts the input and finishes parsing ,then place **acc** in $ column of ACTION part.

- If the set I contain a final item then place r_i under all the terminal in ACTION part. r_i = reduce by the production numbered i.
- ➢ GOTO function takes as argument a state I and a non terminal and contains only shift operation of non terminal.
- If x is a non terminal & goto (I,x) =I_j then place j in GOTO.
- LR(0) grammar:
- The grammar G is said to LR(0) if its parse table is free from multiple entries.
- Ex 1:A →aA|b
- augmented grammar $A' \rightarrow$ A
- A →aA|b



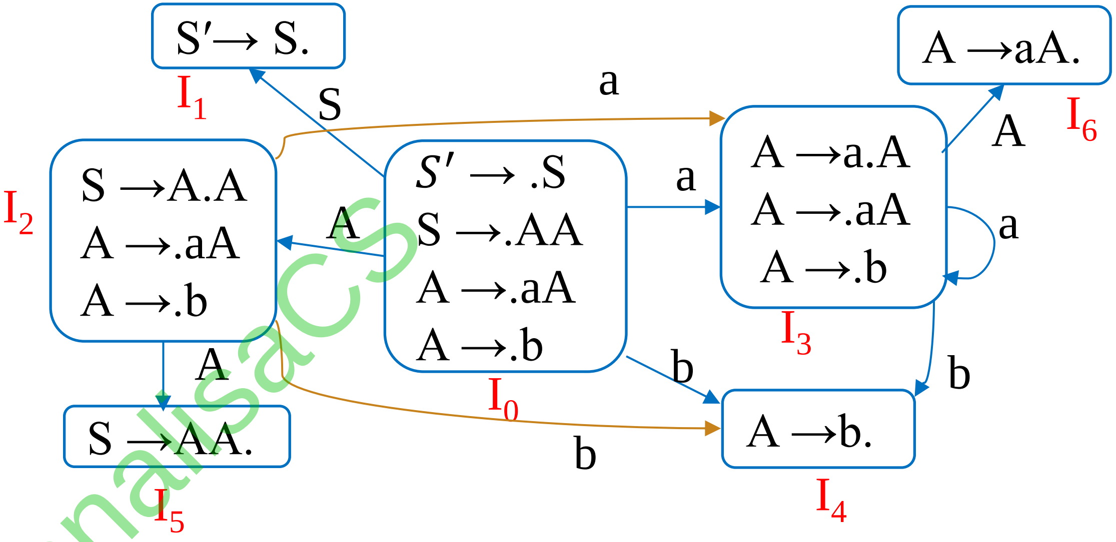
	ACTION			GOTO
	a	**b**	**$**	**A**
0	S_2	S_3		1
1			Acc	
2	S_2	S_3		4
3	r_2	r_2	r_2	
4	r_1	r_1	r_1	

- LR(0) Grammar

- Ex 2: S →AA
- A →aA
- A →b

- Augmented grammar
- S' → S
- S →AA
- A →aA
- A →b

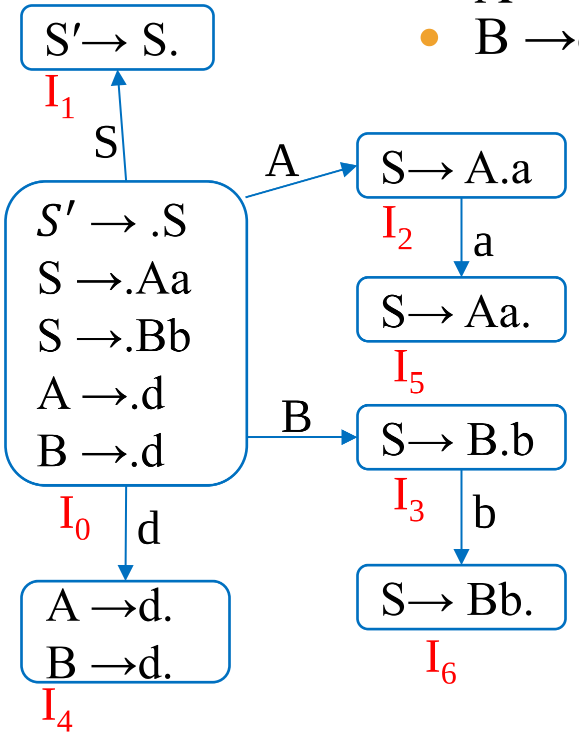
	ACTION			GOTO	
	a	**b**	**$**	**S**	**A**
0	S_3	S_4		1	2
1			Acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		



- LR(0) Grammar

- Ex 3: S →Aa |Bb
- A →d
- B →d

- Augmented grammar
- S′ → S
- S →Aa
- S →Bb
- A →d
- B →d

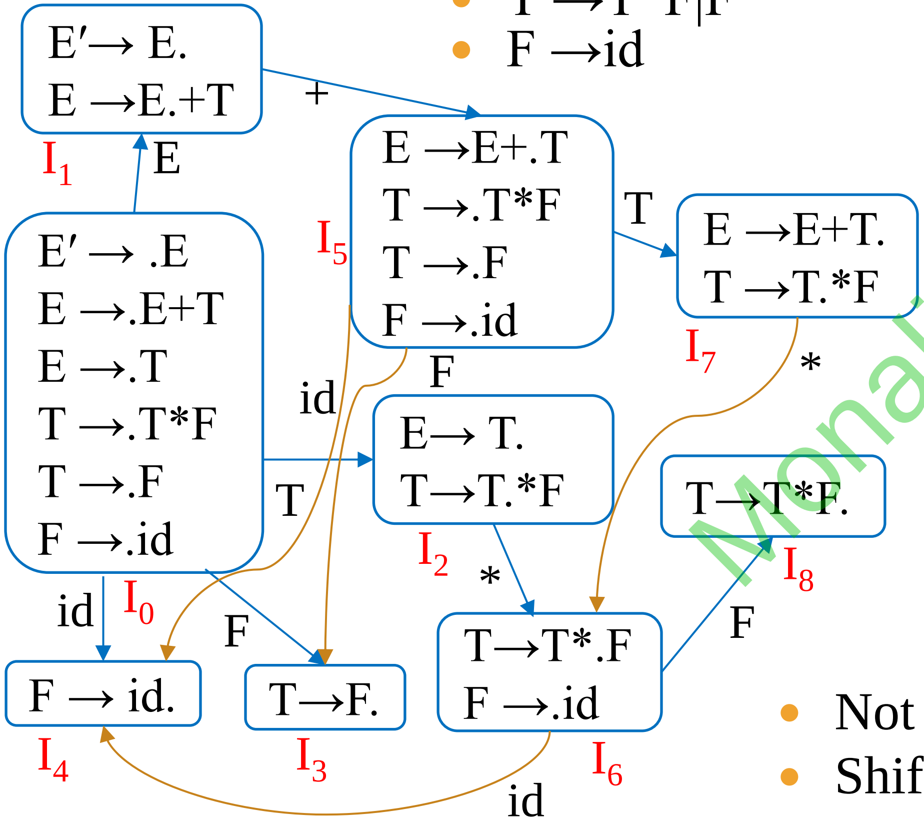


ACTION					GOTO		
	a	**b**	**d**	**$**	**S**	**A**	**B**
0			S_4		1	2	3
1				Acc			
2	S_5						
3		S_6					
4	r_3/r_4	r_3/r_4	r_3/r_4	r_3/r_4			
5	r_1	r_1	r_1	r_1			
6	r_2	r_2	r_2	r_2			

- Not LR(0) Grammar.
- Reduce Reduce conflict present.

- Ex 4: E →E+T |T
- T →T*F|F
- F →id

- Augmented grammar
- E′ → E
- E →E+T |T
- T →T*F|F
- F →id



	ACTION				GOTO		
	id	**+**	*****	**$**	**E**	**T**	**F**
0	S_4				1	2	3
1		S_5		Acc			
2	r_2	r_2	r_2/S_6	r_2			
3	r_4	r_4	r_4	r_4			
4	r_5	r_5	r_5	r_5			
5	S_4					7	3
6	S_4						8
7	r_1	r_1	r_1/S_6	r_1			
8	r_3	r_3	r_3	r_3			

- Not LR(0) Grammar.
- Shift Reduce Conflict present.

- **LR-parsing algorithm:**
- Initially, the parser has 0 on its stack, where 0 is the initial state,and w$ in the input buffer
- let **a** be the first symbol of w$;
- while(1)
- {let **s** be the state on top of the stack;
- if (ACTION[s, a] = shift t) {
- push t onto the stack; }
- else if (ACTION[s, a] = reduce (A→β){
- pop β symbols off the stack;
- let state **t** be on top of the stack;
- push GOTO[t, A] onto the stack;}
- else if (ACTION[s, a] = accept) break;
- else call error-recovery routine;
- }

Stack	Symbols	i/p string	Action
0	$	id*id$	Shift to 4
04	$id	*id$	Reduce F→id
03	$F	*id$	Reduce T→F
02	$T	*id$	Shift to 6
026	$T*	id$	Shift to 4
0264	$T*id	$	Reduce F→id
0268	$T*F	$	Reduce T→T*F
02	$T	$	Reduce E→T
01	$E	$	Accept

	ACTION				GOTO		
	id	+	*	$	E	T	F
0	S_4				1	2	3
1		S_5		Acc			
2	r_2	r_2	r_2/S_6	r_2			
3	r_4	r_4	r_4	r_4			
4	r_5	r_5	r_5	r_5			
5	S_4					7	3
6	S_4						8
7	r_1	r_1	r_1/S_6	r_1			
8	r_3	r_3	r_3	r_3			

- **Conflicts in LR Parsing:**
- ***1. Shift-reduce conflict*** :
- The parser cannot decide whether to shift or to re
- If the same state has both shift & reduce option th
- ***2.Reduce-reduce conflict*** :
- The parser cannot decide which of several reduct
- If the same state contain more than one final item
- The grammar is LR(0) if & only if it is free from

Stack	i/p string	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

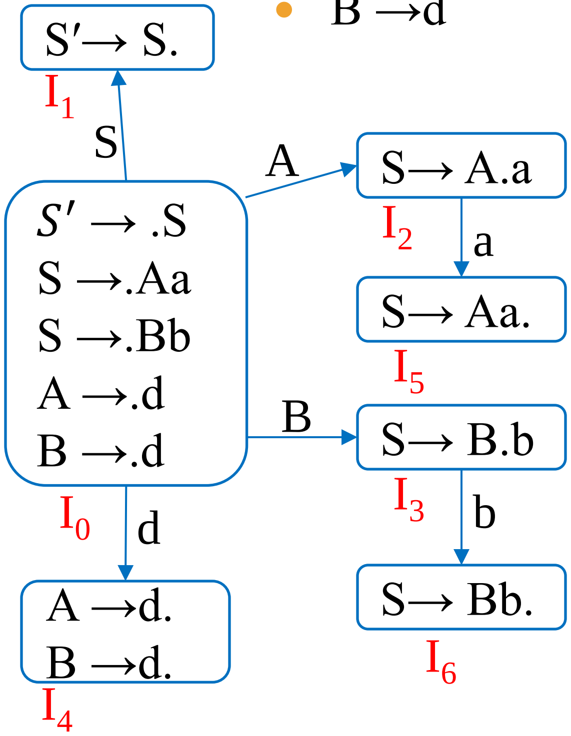
- **Viable Prefixes:**
- The prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called *viable prefixes.*
- A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.
- Not all prefixes of right-sentential forms can appear on the stack.
- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes
- Consider grammar :E →E+T |T , T →T*F|F , F →id
- Examples of Right sentential form : $E \Rightarrow T \Rightarrow T*F \Rightarrow T*id_2 \Rightarrow F*id_2 \Rightarrow id_1*id_2$
- Examples of viable prefix : id_1,F,T,T*,$T*id_2$,T*F,E

- **SLR(1) Parser:**
- The SLR method begins with LR(0) items and LR(0) automata.
- **Constructing an SLR-parsing table.**
- **ACTION** :
- (a) If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i ,a) = I_j , then set ACTION[i,a] to "S_j" ; a=terminal.
- (b) If $[A \rightarrow \alpha.]$ is in I_i , then set ACTION[i,a] to "r_j" for all **a** in FOLLOW(A);j is reduction number,A$\neq S'$.
- (c) If [S′→S.] is in I_i, then set ACTION[i, $] to "accept".
- If any conflicting actions result from the above rules, we say the grammar is not SLR(1).
- **GOTO** :
- The GOTO transitions for state i are constructed for all non terminals A using the rule: If GOTO(I_i , A) = I_j , then GOTO[i , A] = j.
- If SLR(1) parsing table is free from multiple entries then grammar called SLR grammar.
- We usually omit the (1) after the SLR, since we shall not deal here with parsers having more than one symbol of lookahead.
- Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1).

- Ex 1:
- S →Aa |Bb
- A →d
- B →d

- Augmented grammar
- $S' \rightarrow S$
- S →Aa
- S →Bb
- A →d
- B →d

- Follow(S)={$}
- Follow(A)={a}
- Follow(B)={b}



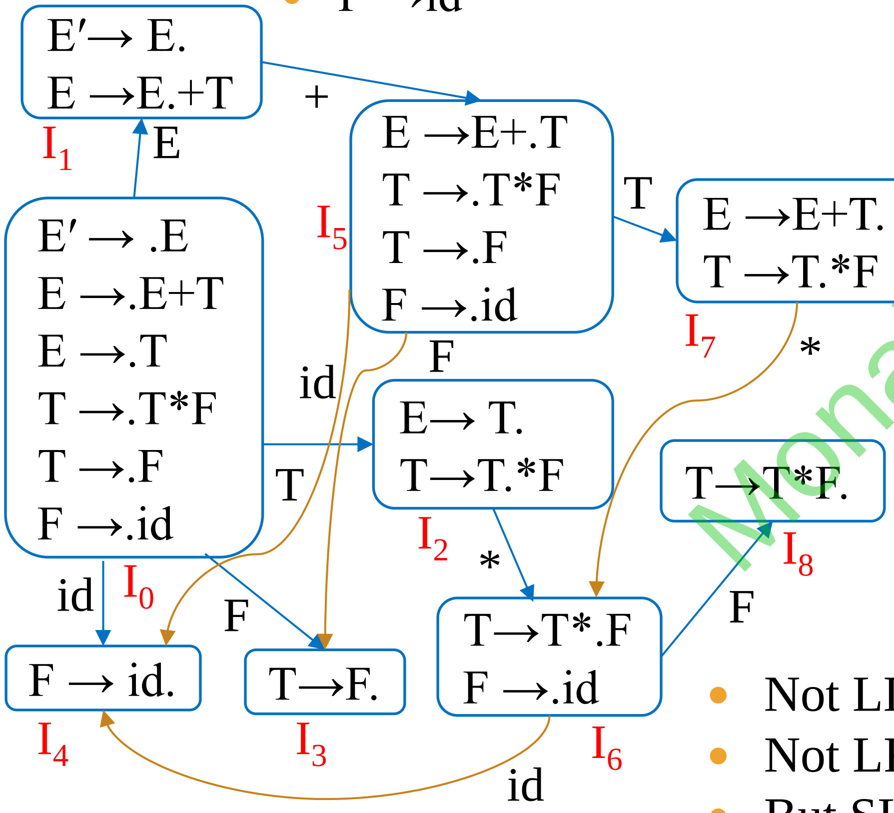
	ACTION				GOTO		
	a	**b**	**d**	**$**	**S**	**A**	**B**
0			S_4		1	2	3
1				Acc			
2	S_5						
3		S_6					
4	r_3	r_4					
5				r_1			
6				r_2			

- Not LL(1) Grammar . First(Aa)∩ First(Bb)=d
- Not LR(0) Grammar . RR conflict present.
- But SLR(1) Grammar

- Ex 2:
- E →E+T |T
- T →T*F|F
- F →id

- Augmented grammar
- E′ → E
- E →E+T |T
- T →T*F|F
- F →id

- Follow (E)={$,+}
- Follow (T)={$,*,+}
- Follow (F)={$,*,+}



	ACTION				GOTO		
	id	+	*	$	E	T	F
0	S_4				1	2	3
1		S_5		Acc			
2		r_2	S_6	r_2			
3		r_4	r_4	r_4			
4		r_5	r_5	r_5			
5	S_4					7	3
6	S_4						8
7		r_1	S_6	r_1			
8		r_3	r_3	r_3			

- Not LL(1) grammar , Left recursive
- Not LR(0) Grammar , SR Conflict.
- But SLR(1) Grammar

Conflicts in SLR Parsing:

- ***1. Shift-reduce conflict*** :
- If follow(B)∩ x=∅, SR conflict in LR(0) but not in SLR(1).
- If follow(B)∩ x≠ ∅ SR conflict in both LR(0) & SLR(1).
- ***2.Reduce-reduce conflict*** **:**

$A \rightarrow \alpha.xB$
$B \rightarrow \gamma.$

$A \rightarrow \alpha.$
$B \rightarrow \gamma.$

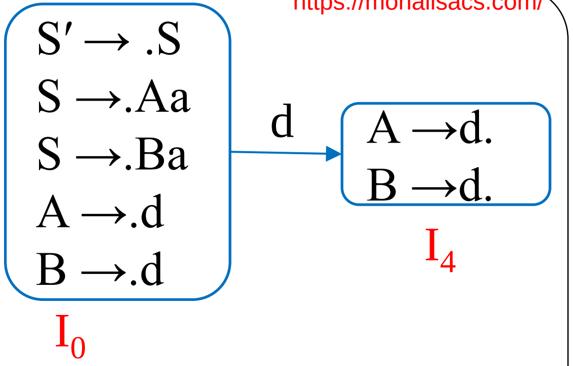
- If follow(A)∩ follow(B) =∅, RR conflict in LR(0) but not in SLR(1).
- If follow(A)∩ follow(B) ≠ ∅ RR conflict in both LR(0) & SLR(1)
- The grammar is SLR if & only if it is free from both SR & RR conflict.
- Ex 3:
- S →AaAb|BbBa
- A→∈
- B→∈

$S' \rightarrow .S$
$S \rightarrow .AaAb$
$S \rightarrow .BbBa$
$A \rightarrow .$
$B \rightarrow .$

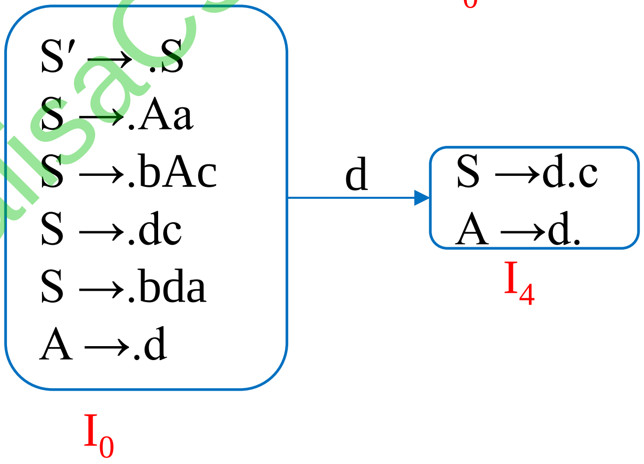
I_0

- Follow (S)={$}
- Follow (A)={a,b}
- Follow (B)={a,b}
- Follow(A)∩ Follow(B)={a,b} ≠ ∅
- RR conflict in both LR(0) & SLR(1)
- Not LR(0) grammar.
- Not SLR(1) grammar.
- LL(1) grammar
- First(AaAb) ∩First(BbBa)=∅

- Ex 4:
- S →Aa|Ba
- A→d
- B→d
- Follow (S)={$}
- Follow (A)={a}
- Follow (B)={a}
- Follow (A)∩ Follow(B)={a} ≠ ∅
- RR conflict in both LR(0) & SLR(1)
- Not LR(0) grammar .
- Not SLR(1) grammar.
- Not LL(1) grammar
- First(Aa) ∩First(Ba)={d}



- Ex 5:
- S →Aa|bAc|dc|bda
- A→d
- Not LL(1) grammar
- First(Aa) ∩First(dc)={d}
- Follow(A)∩ c={c}
- SR conflict in both LR(0) & SLR(1)
- Not LR(0) grammar .
- Not SLR(1) grammar.



- SLR(1) is more powerful than LR(0)
- Every LR(0) grammar is SLR(1) but converse not true.
- The number of entries in LR(0) table ≥ number of entries in SLR(1) table .
- Both table differ only in ACTION part not GOTO part.
- SLR(1) is more efficient than LR(0).
- **CLR(1) or LR(1) parser:**
- LR(1) =LR(0) +1 Look ahead symbol
- LR(1) determines the reduction dependency on the LA symbol.
- The redundant reduction can be removed hence it is called canonical LR(1).
- **LR(1) item:**
- [A $\rightarrow \alpha . \beta$,a],A $\rightarrow \alpha\beta$ is a production, β is not ε,a is a terminal or right endmarker $.
- 1 refers to the length of second component called lookahead of the item.
- The lookahead has no effect on item.
- [A $\rightarrow \alpha$. ,a], is a reduction if next input symbol is a.
- The set of a's will always be a proper subset of FOLLOW(A).

- **Constructing LR(1) Sets of Items:**
- The method for building the collection of sets of valid LR(1) items is same as the one for building the canonical collection of sets of LR(0) items.
- We need only to modify the two procedures CLOSURE and GOTO.
- **CLOSURE(I)** {repeat
-   for (each item [A → α. Bβ,a] in I)
-     for (each production B →γ in G')
-       for (each terminal b in FIRST(βa))
-         add [B →. γ ,b] to set I;
- until no more items are added to I;
- return I;}
- **GOTO(I, X)** {
-     for (each item [A → α.Xβ,a] in I)
-       add item [A → αX.β,a] to set J ;
- return CLOSURE(J); }

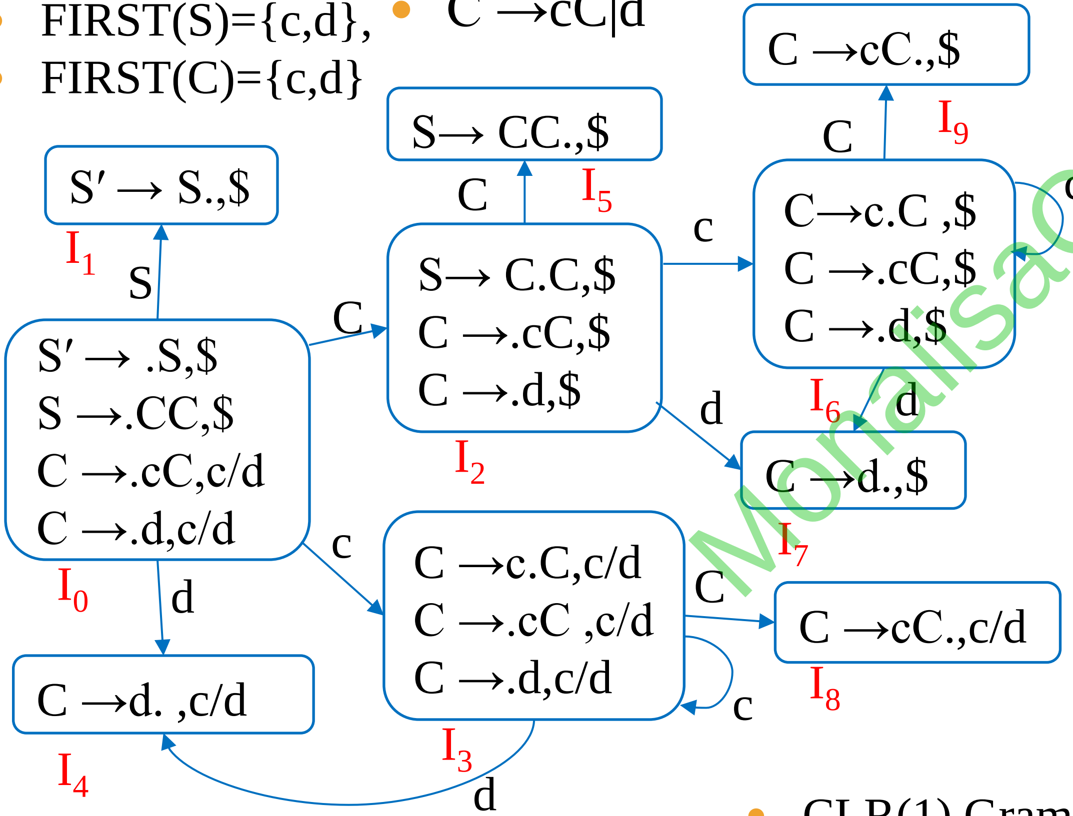
- void items(G') {
- initialize C to {CLOSURE [S′→.S,$]} ;
- repeat
- for (each set of items I in C)
-    for (each grammar symbol X)
-      if (GOTO(I, X) is not empty and not in C)
- add GOTO(I, X) to C;

until no new sets of items are added to C;
}

- **Canonical LR(1) Parsing Tables:**
- Algorithm : Construction of canonical-LR parsing tables.
- **ACTION:**
- (a) If $[A \rightarrow \alpha.\, a\beta, b]$ 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 $[A \rightarrow \alpha.\, ,a]$ is in I_i, A≠ S′, then set ACTION[i,a] to "r_j" ;j is reduction number.
- (c) If $[S' \rightarrow S.]$ is in I_i, then set ACTION[i, \$] to "accept".
- If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.
- **GOTO:**
- The GOTO transitions for state i are constructed for all non terminals A using the rule: If GOTO(I_i , A) = I_j , then GOTO[i , A] = j.
- A LR parser using this table is called a CLR(1) parser.
- If the table has no multiple entries then the given grammar is called LR(1) grammar.

- Ex 1:
- S →CC
- C →cC|d
- FIRST(S)={c,d},
- FIRST(C)={c,d}

- Augmented grammar
- S′ → S
- S →CC
- C →cC|d



ACTION				GOTO	
State	**c**	**d**	**$**	**S**	**C**
0	S_3	S_4		1	2
1			acc		
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		

- CLR(1) Grammar

Conflicts in CLR Parsing:

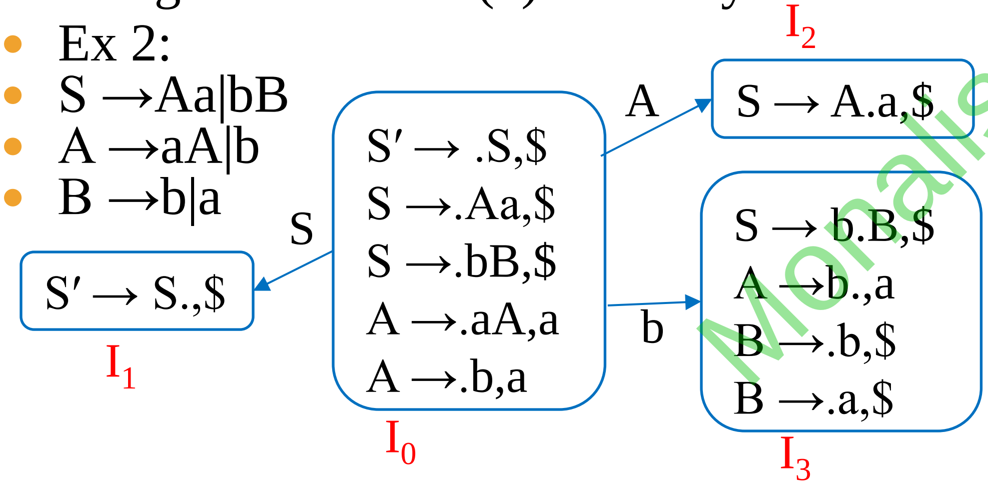
- ***1. Shift-reduce conflict*** :
- Shift terminal ∩ reduction look ahead symbol≠ ∅,SR conflict in LR(1)

$A \rightarrow \alpha.a\beta,b$
$B \rightarrow \gamma.\ ,a$

- ***2.Reduce-reduce conflict*** **:**

$A \rightarrow \alpha.,a$
$B \rightarrow \gamma.,a$

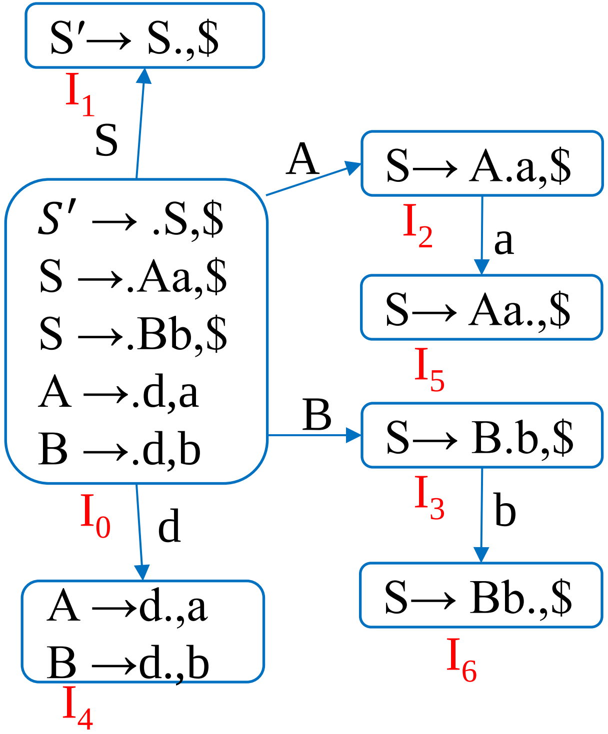
- r_i look ahead symbol ∩ r_j look ahead symbol ≠ ∅ then its RR conflict in LR(1).
- The grammar is LR(1) if & only if it is free from both SR & RR conflict.
- Ex 2:
- S →Aa|bB
- A →aA|b
- B →b|a



- a ∩ a=a Shift reduce conflict present.
- The grammar is not LR(1)
- FOLLOW(A) ∩ {a}=a ,SR conflict for both LR(0) &SLR
- Not LR(0), SLR grammar
- FIRST(Aa) ∩ FIRST(bB) =b
- Not LL(1) grammar

- The grammar is not LL(1) or LR as it is ambiguous.
- For string "ba" there are more than one parse tree.

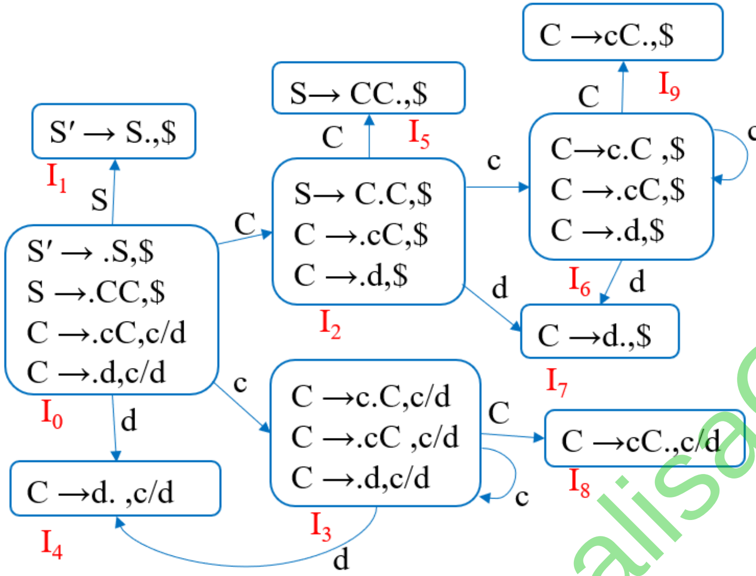
- Ex 3:
- S →Aa |Bb
- A →d
- B →d



- a ∩b=∅ ,No R-R conflict present for CLR.
- The grammar is LR(1) or CLR
- FOLLOW(A) ∩FOLLOW(B)= ∅ ,No R-R conflict for SLR
- The grammar is SLR .
- But for LR(0) R-R conflict present.
- The grammar is not LR(0)
- FIRST(Aa) ∩ FIRST(Bb) =d
- Not LL(1) grammar

- **LALR(1) Parser:**
- Minimal LR(1) Automata.
- The Automata of CLR(1) parser may contain some states with same production part and different look ahead part.
- Make all these state to single state by union of LA part & again draw the automata & construct the parse table ,which called as LALR table.
- If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar.
- The collection of sets of items are called LALR(1) collection.
- The SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like C.
- The canonical LR table would typically have several thousand states for the same-size language.
- CLR(1) is more powerful than LALR(1) & LALR(1) is more powerful than SLR(1).
- Every LALR(1) grammar is CLR(1) But every CLR(1) need not be LALR(1).
- If CLR(1) have RR conflict or may not have RR conflict ,still LALR(1) may have RR Conflict . LALR(1) have SR conflict if and only if CLR(1) have SR conflict .
- The grammar which is not CLR also not LALR.
- Every SLR(1) grammar is LALR(1) but reverse may not true.
- Number of states in CLR(1) Automata $\geq$ LALR(1) Automata.

- Ex 1:
- $S \rightarrow CC$
- $C \rightarrow cC|d$



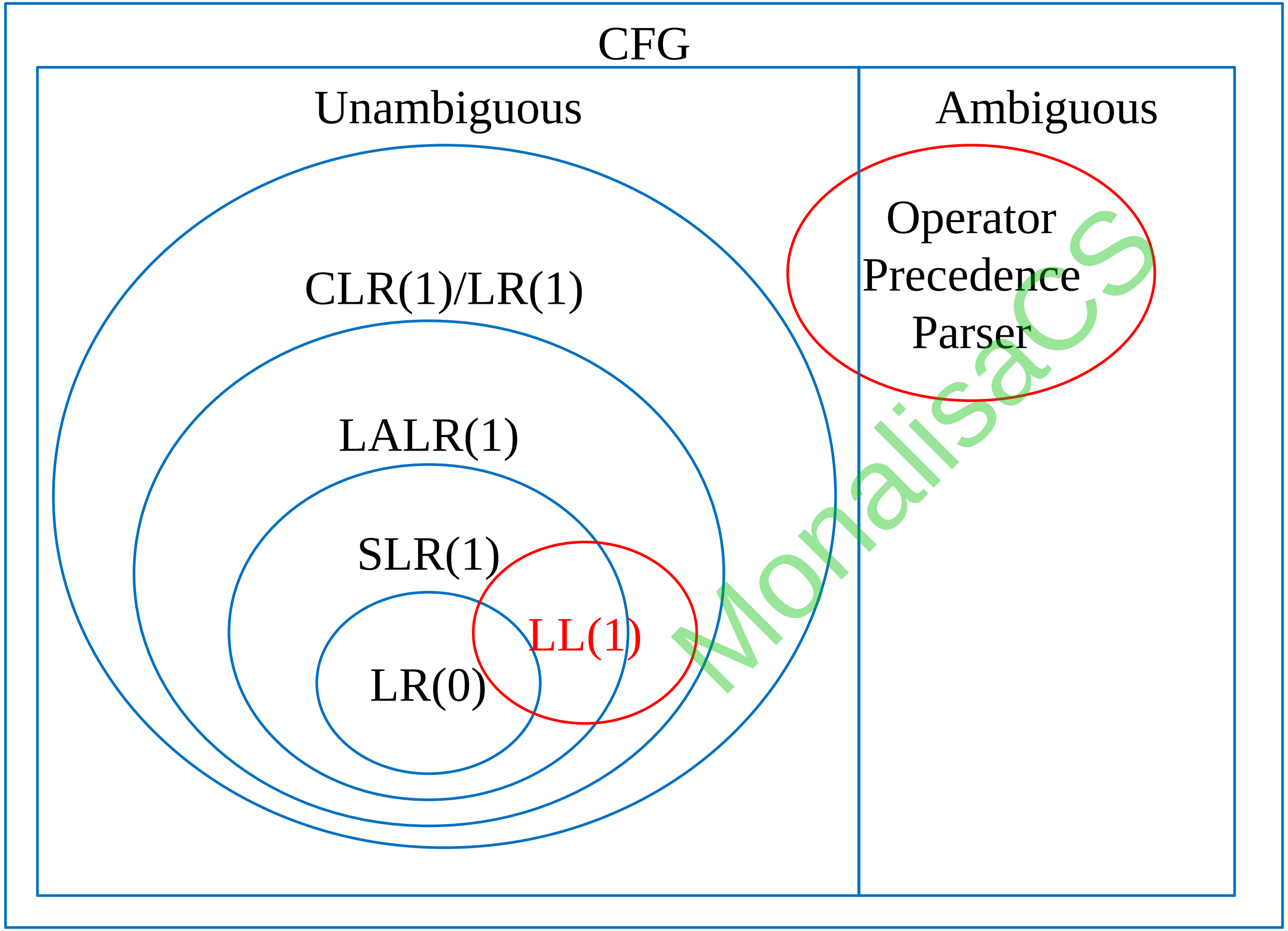
	ACTION			GOTO	
State	**c**	**d**	**$**	**S**	**C**
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

- I_3 and I_6 are replaced by their union.
- I_{36}: $C \rightarrow c.C,c/d/\$$
- $\quad C \rightarrow .cC,c/d/\$$
- $\quad C \rightarrow .d,c/d/\$$
- I_4 and I_7 are replaced by their union.
- I_{47}: $C \rightarrow d.,c/d/\$$
- I_8 and I_9 are replaced by their union.
- I_{89}: $C \rightarrow cC.,c/d/\$$

- The grammar is LR(0),SLR,CLR(1)& LALR
- LALR parsing table is same as SLR table.
- The grammar is also LL(1).

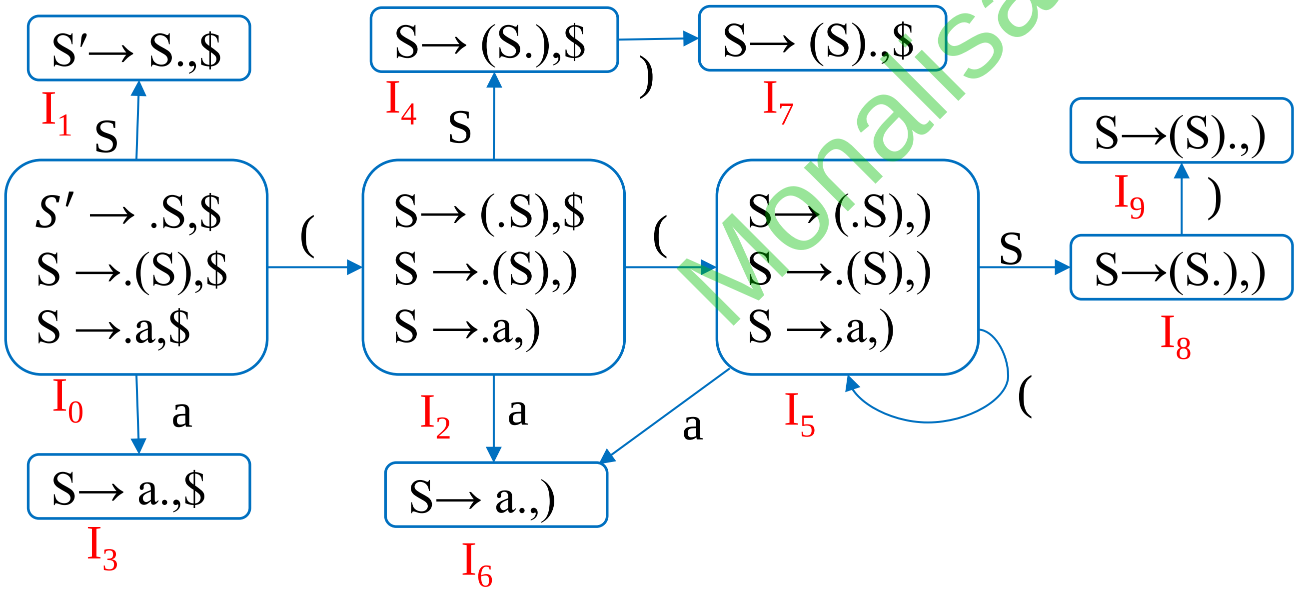
- LL(k) ≤LR(k)
- Set of all LL(0) CFG ⊂ Set of all LL(1) CFG ⊂ Set of all LL(2) CFG…
- Set of all LR(0) CFG ⊂ Set of all LR(1) CFG ⊂ Set of all LR(2) CFG…
- Set of all LL(k) CFG ⊂ Set of all LR(k) CFG.
- CLR(1) is more powerful efficient among all the parser.
- But it is very costly hence LL(1) & LALR(1) widely used in the real time compiler construction.
- If one LL(1) grammar having no null production then its also SLR(1)
- Every LL(1) grammar is LALR(1),hence LR(1) as LALR(1) ⊂CLR(1)
- Power :LR(0)< SLR(1) <LALR(1) <CLR(1)
- Easy to implement: LR(0)>SLR(1)>LALR(1)>CLR(1)
- Grammar : Set of LR(0) CFG ⊂SLR(1) class ⊂LALR(1) class ⊂CLR(1) Class
- Language : Set of LR(0) Language ⊂SLR(1) class ⊂LALR(1) class ⊂CLR(1) Class
- Parser : LR(0)< SLR(1) <LALR(1)<CLR(1)
- Table Size :LR(0)=SLR(1)=LALR(1)≤CLR(1)
- Reduce entry in table: LR(0)>SLR(1)=LALR(1)>CLR(1)
- Number of state in Automata :LR(0)=SLR(1)=LALR(1)≤CLR(1)

Relation between all parser



- GATE CS 2003,Q57:Consider the grammar shown below.
- S→C C
- C→c C|d
- This grammar is
- (A)LL(1) (B)SLR(1) but not LL(1)
- (C)LALR(1) but not SLR(1) (D)LR(I) but not LALR(1)
- Ans: (A)LL(1)
- GATE CS 2008,Q55:An LALR(1) parser for a grammar G can have shift-reduce (S-R) conflicts if and only if
- (A)The SLR(1) parser for G has S-R conflicts
- (B)The LR(1) parser for G has S-R conflicts
- (C)The LR(0) parser for G has S-R conflicts
- (D)The LALR(1) parser for G has reduce-reduce conflicts
- Ans :(B)

- GATE CS 2005, Q60: Consider the grammar:
- $S \rightarrow (S) | a$
- Let the number of states in SLR (1), LR(1) and LALR(1) parsers for the grammar be n_1, n_2 and n_3 respectively. The following relationship holds good:
- (A) $n_1 < n_2 < n_3$ (B) $n_1 = n_3 < n_2$ (C) $n_1 = n_2 = n_3$ (D) $n_1 \geq n_3 \geq n_2$
- Number of state in Automata : LR(0)=SLR(1)=LALR(1)≤CLR(1)
- $n_1 = n_3 \leq n_2$



- $n_2 = 10$
- $I_2 \cup I_5 = I_{25}$
- $I_3 \cup I_6 = I_{36}$
- $I_4 \cup I_8 = I_{48}$
- $I_7 \cup I_9 = I_{79}$
- $n_3 = 6, n_1 = 6$
- Ans: (B) $n_1 = n_3 < n_2$