GATE CSE NOTES

by

UseMyNotes

Introduction

HLL
↓
Preprocessor
↓ Pure HLL
Compiler
↓ Assembly language
Assembler
↓ M/C code (relocatable)
Loader/Linker
↓
Executable code / absolute M/C code

#include File inclusion
#define Macro expansion

* Single, Multipass Compiler.
  - Single: More memory, Less time. → as more memory needed for all phases
  - Multipass: Less memory, More time → for context switch

* Type info is stored by semantic analyser.

* If no declaration is there for identifier, LA saves info at symbol table.

✓ Various phases

HLL
↓
Lexical Analysis
↓ stream of tokens
Syntax Analysis
↓ Parse tree
Semantic Analysis (Type checking)
↓ Parse tree (Semantically verified)
Intermediate Code Generation
↓ Three Address Code.
Code optimization (optional)
↓
Target Code Generation
↓
Assembly

Symbol table manager — Write / Read — contains all identifiers

Error handler

- Front end of compilation (LA, Sy A, Se A, ICG)
- Back end " " (TCG)
- Code opt^n neither front nor back end.

Example

$x = a + b * c ;$ --- deleting comments.

⇓

Lexical Analyser — LEX

Pattern id' $l(l+d)^*$

↳ Need to be variable, can't be a fn name or array name

⇓

id = id + id * id ← Stream of tokens

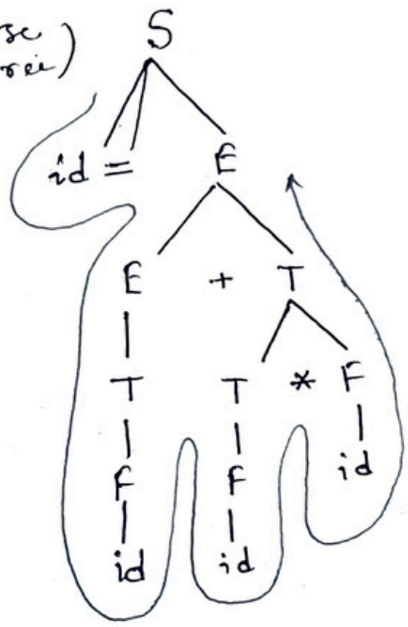
⇓

YACC — Syntax Analyser ⇐

$S \rightarrow id = E ;$
$E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow id$

CFG

(Parse tree)



yield of the parse tree and syntax analyser need to be same. (otherwise syntax error)

⇒

Semantic analyser

⇓

(Parse tree semantically verified)

⇓

Intermediate Code Generator

⇓

$t_1 = b * c ;$
$t_2 = a + t_1 ;$
$x = t_2 ;$

3-address code (max. 3 addr.)

⇓

Code optimiser

⇓

$t_1 = b * c ;$
$x = a + t_1 ;$

⇓

Target code generator

Writing code that assembler can understand

⇓

mul R_1, R_2	$a \rightarrow R_0$
add R_0, R_2	$b \rightarrow R_1$
mov R_2, x	$c \rightarrow R_2$

Till ICG, we can implement in any machine. Next 2 phases need to be changed for different machines.

Till ICG → Front end

CO — ICG → Back end

CO: neither front nor back

LANCE

* Symbol Table.

(allows us to find the record for each identifier quickly & to store or retrieve data from the record.)

Data structure created & maintained by compilers in order to store information about the occurrences of various entities such as variable names, function names, objects, classes, interfaces etc.

The information is collected by the analysis phase of a compiler & used by the synthesis phase to generate target code.

→ Usage of symbol table by phases :

Phase	Usage.
1. Lexical analysis	Creates new entries for each new identifiers.
2. Syntax analysis	Adds information regarding attributes like type, scope, dimension, line of reference & line of use.
3. Semantic analysis.	Uses the available information to check for semantics & is updated.
4. Intermediate code generation	Information in symbol table helps to add temporary variable's information.
5. Code optimization	Information in symbol table used in machine-dependent optimization by considering address & aliased variables' information.
6. Target code generator	Generates the code by using the address information of identifiers.

→ Symbol table entries.

Each entry in the symbol table is associated with attributes that support the compiler in different phases. These attributes are:

1. Name
2. Size
3. Dimension
4. Type
5. Line of declaration
6. Line of usage
7. Address

* Symbol table implemented as:

	Insertion	Search
LL	O(1)	O(n)
Hash table	O(1)	O(1)
AVL tree	O(lgn)	O(lgn)

eg.

Name	Type	Size	dim	LOD	LOU	Addr.
RAVI	char	4	1	..	..	..
AGE	int	2	0	..	..	..

- All the attributes are not of fixed size.

→ Limitation of fixing the size of symbol table:

i) If chosen small, it can't store more variables.

ii) If chosen large, lot of space wasted.

So, the size of symbol table should be dynamic to allow the increase in size at compile time.

→ Operations on the symbol table.

Dependent on whether the language is block-structure or non-block structured.

✓ Non-block structured: Contains only one instance of the variable declaration and its scope is through out the program.

For non-block structured languages the operations are:

→ Insert
→ Lookup

```
{ int i;
...
}
```

✓ Block structured: Here the variables may be redeclared if its scope is within that block.

Operations are: Insert
Lookup
Set
Reset.

```
{ int i;
...
{ int i;
...
}
}
```

G'12. Access time (Lookup time) of the symbol table is logarithmic if it is implemented by a

a) Linear list (O(n).)
✓b) Search tree (O($\log_2 n$)) → for binary (k for general - no of children max)
c) Hash table (O(1))
d) None

* LA, SyA, SeA will work at a time by synchronising with each other, so that within one pass all 3 compute their work, otherwise 3 passes needed.

• Implementation of Symbol table	Insertion time	Lookup time	Disadvantages.
(1) Linked list			- Lookup time directly proportional to table size.
(a) Ordered list			
↳ Array	O(n)	O(log n)	- Every insertion operation preceded with lookup operation. (no duplicate elems)
↳ Linked list	O(n)	O(n)	
(b) Unordered list	~~O(n)~~ O(1)	O(n)	
(2) Self organising list. (unsorted)	O(1)	O(n)	Poor performance when less frequent items are searched.
(3) Search tree	$O(\log_k n)$	$O(\log_k n)$	We have to always keep it balanced.
(4) Hash table	O(1)	O(1)	When there are too many collisions the time complexity increases to O(n).

* Lexical analyser. - Grammars.

- Lexemes → Tokens.
- Removing comments
- Removing white spaces
- If error, show (by providing row, col no.)

```
int max (x, y)
  int x, y ;
/* ..... */
{
    return (x > y ? x : y);
}
```

tokens = 25

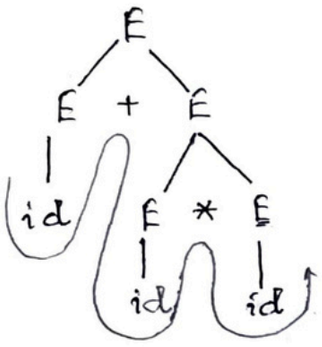
- printf (" %d asd " , & x) ;
  1 2 3 4 5 6 7 8

No. of tokens. - 8

- Grammar G = (V, T, P, S).
- LA uses DFA for tokenization. LA only phase that reads prog. char by char.

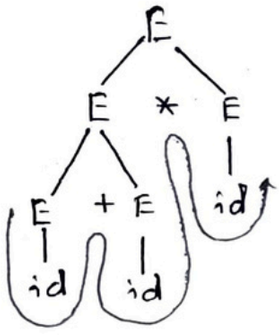
eg. $E \to E+E \mid E*E \mid id$

$E \Rightarrow id + id * id$



$2 + 3 * 4$

$= 14$



$2 + 3 * 4$

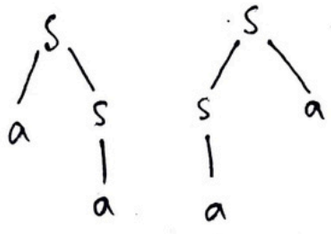
$= 20$

↓

Ambiguous. (>1 parse trees)

eg. $S \to aS \mid Sa \mid a$

$w = aa.$

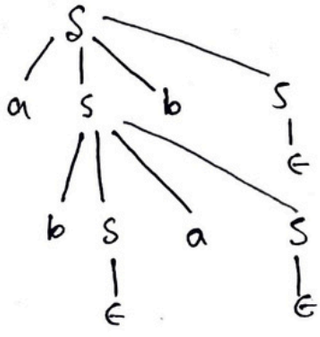


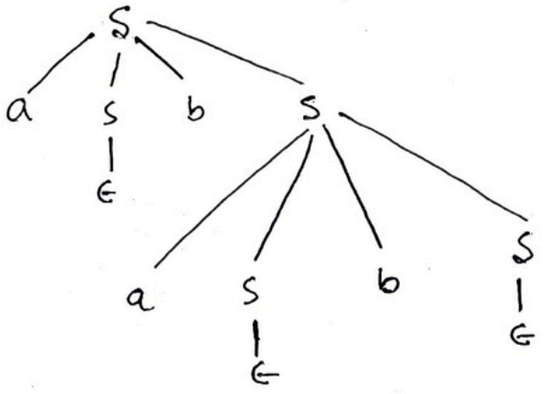
ambiguous.

~~Ambiguity problem is undecidable.~~

eg. $S \to aSbS \mid bSaS \mid \in$

$w = abab.$

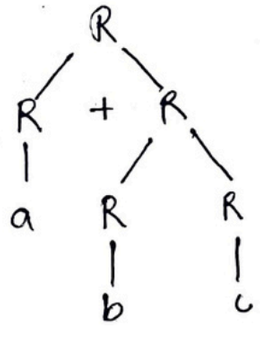


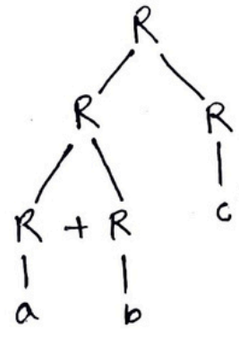


ambiguous.

eg. $R \to R+R \mid RR \mid R^* \mid a \mid b \mid c.$

$w = a+bc$





ambiguous.

→ Errors & their recovery in lexical analysis.

- Errors –
  1. Numeric literals that are too long
  2. Long identifiers
  3. Ill-formed numeric literals. | int a = $123
  4. Input characters that are not in the source language.

- Error recovery –
  1. Delete. : Unknown characters are deleted. Known as panic mode recovery.

  eg. "charr" corrected as "char" deleting 'r'

  2. Insert : An extra or missing character is inserted to form a meaningful token.

  eg. "cha" corrected as "char".

  3. Transpose. : Based on certain rules we can transpose 2 characters.

  eg. "whiel" can be corrected as "while"

  4. Replace : Based on replacing one character by another.

  eg. "chrr" can be corrected as "char" by replacing 'r' with 'a'.

→ Disambiguity rules. (Precedence, Associativity – left or right)

eg. E → E + E / E * E / id

↓

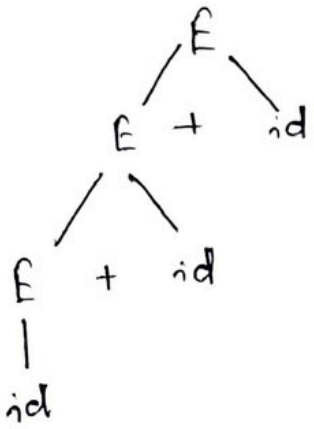
id + id + id (2 parse trees)

– Rules of associativity failed

id + id * id (2 parse trees)

– Operator precedence failed.

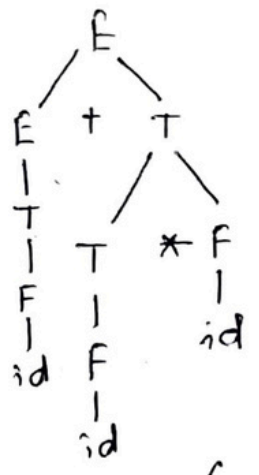
$E \rightarrow E + id \;/\; id$



✓ Left recursive →
* Left associative

$$\left\{\begin{array}{l} E \rightarrow E + T \;/\; T \\ T \rightarrow T * F \;/\; F \\ F \rightarrow id \end{array}\right.$$

Unambiguous.



Level in grammar.

+ ↘
*

(+ at higher level
* at lower level).

eg. bExp → bExp or bExp
ambiguous ↗ / bExp and bExp
/ not bExp / True / False.

$E \rightarrow E$ or F / F
$F \rightarrow F$ and G / G.
$G \rightarrow$ not G / True / False.

} Unamb

Precedence.

$\neg > \wedge > \vee$

$\neg$ unary
$\wedge, \vee$ left associative.

Higher precedence operator is evaluated at the end of the grammar.

eg. $R \rightarrow R+R \;/ RR / \; R^* / a/b/c.$ (ambiguous)

$E \rightarrow E + T / T.$
$T \rightarrow TF / F$
$F \rightarrow F^* / a/b/c.$

} Unamb.

Precedence

* > · > +.

+, · left associative

eg. G'

$A \rightarrow A \$ B / B$
$B \rightarrow B \# C / C$
$C \rightarrow C @ D / D$
$D \rightarrow d.$

Left recursive.
→ Left associative.

$:> $
:>
@ :> @.

precedence
$ <· # <· @

Rules

- Lowest precedence should be first derived.
- When left recursive / left associative form a derivation like $A \rightarrow A \$ B / B$

Imp

* eg. $E \to E * F / F + E / F$

$F \to F - F / id$

Decide the precedence & associativity rules. $\Rightarrow$

+, * same precedence

− > +, *

* left associative

+ right associative.

* > *

+ < +.

- **Recursion**

  - **Left**

    $A \to A\alpha / \beta$

    $\beta\alpha^*$ lang.

    ↓

    Leads to infinite recursion.

    So eliminate LR.

  - **Right**

    $A \to \alpha A / \beta$.

    $\alpha^*\beta$. lang.

Rule:

$A \to A\alpha / \beta$ LR

$\Updownarrow$

$\begin{cases} A \to \beta A' \\ A' \to \epsilon / \alpha A' \end{cases}$ RR

eg. $\underset{A}{E} \to \underset{A}{E} + T / \underset{\beta}{T}$ (E: A, α = $+T$; T: β)

$\begin{cases} E \to TE' \\ E' \to \epsilon / +TE' \end{cases}$

eg. $S \to (L) / x$

$L \to L, S / S$

$\bar{A} \quad A\alpha \quad \beta$.

$\begin{cases} S \to (L) / x \\ L \to SL' \\ L' \to \epsilon / ,SL' \end{cases}$

eg. $\underset{A}{S} \to \underset{A}{S}0\underset{\alpha}{S}1\underset{\beta}{S} / \underset{\beta}{01}$

$\begin{cases} S \to 01S' \\ S' \to \epsilon / 0S1SS' \end{cases}$

eg. $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \cdots$

$/\beta_1 / \beta_2 / \beta_3 / \cdots$

$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' / \cdots$

$A' \to \epsilon / \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \cdots$

• Grammar

Ambig. (X) → Unambig. (precedence, associativity)

Grammar

LR (X) → RR

$A \to A\alpha/\beta$ ⇓ $A \to \beta A'$, $A' \to \epsilon/\alpha A'$

Grammar

Non-deterministic

Deterministic. (generate new derivation having Common prefix)

$A \to \alpha\beta_1/\alpha\beta_2/\alpha\beta_3 \to A \to \alpha A'$, $A' \to \beta_1/\beta_2/\beta_3$

Left factoring (to get rid of non-determinism). or common prefixes

eg. $S \to iEtS/iEtSeS/a$ Ambiguous (iEtiEtSeS)

$E \to b.$

⇓ Left factoring

$S \to iEtSS'/a$

$S' \to \epsilon/eS$ ① Deterministic.

$E \to b.$

Eliminating nondeterminism does not affect ambiguity.

eg. $S \to aSSbS/aSaSb/abb/b.$

Max common prefix aS. or a

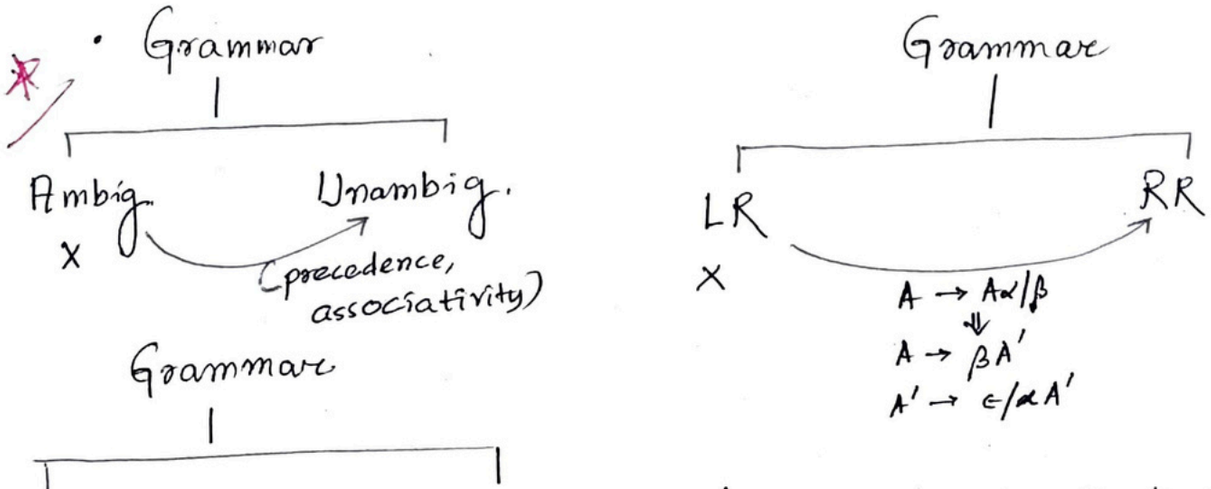
$S \to aS'/b.$

$S' \to SSbS/SaSb/bb$

$S' \to SS''/S''$

$S'' \to SbS/aSb.$

Eliminating common prefix



Left factoring (to get rid of non-determinism). or common prefixes

eg. $S \rightarrow iEtS / iEtSeS / a$ Ambiguous (iEtiEtSeS)

$E \rightarrow b.$

Left factoring

$S \rightarrow iEtSS' / a$

$S' \rightarrow \epsilon / eS$

$E \rightarrow b.$

Deterministic.

Eliminating nondeterminism does not affect ambiguity.

eg. $S \rightarrow aSSbS / aSaSb / abb / b.$

Max common prefix aS. or a

$S \rightarrow aS' / b.$

$S' \rightarrow SSbS / SaSb / bb$

$S' \rightarrow SS'' / S''$

$S'' \rightarrow SbS / aSb.$

Eliminating common prefix

eg. Eliminating common prefix.

$S \rightarrow \underline{bS}Saas / \underline{bS}Sasb / \underline{bS}b / a$

$\Downarrow$

$S \rightarrow bss' / a$

$s' \rightarrow \underline{S}aas / \underline{S}asb / b.$

$s' \rightarrow sas'' / b$

$s'' \rightarrow as / sb$

~~Parsers~~ Parsers

* Parsers (Amb. x)

- Top down parsers (TDP)
  - TDP with full backtracking
    - Brute-force method
  - TDP without backtracking (LRX, NDX)
    - Recursive descent
    - Non-recursive descent
      - LL(1) ← (LRec^{n} x, Amb x, Non-det^{m} x)
- Bottom up parsers (BUP) (SR parsers: Shift-reduce)
  - Operator precedence parser (Amb.)
  - LR parsers
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1)

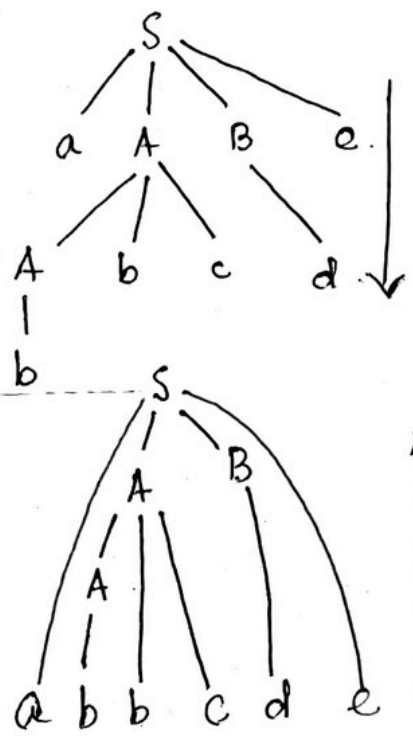
(Amb. x)

eg. $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d.$

$w = abbcde$



TD — Left most derivation

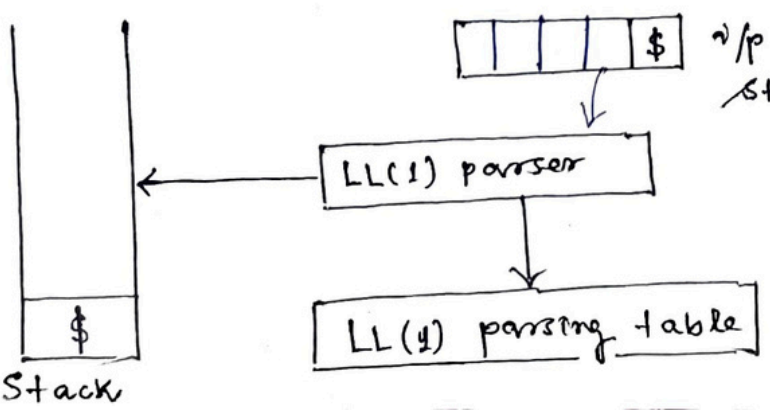
Which prod^{n} to choose for the variable?

∵ BU — Reverse right-most derivation

When to reduce a symbol?

* LL(1) parsers.

- Scan from left to right — L
- Using left most derivation — L
- 1 - No of lookaheads. — 1



(used to indicate end of i/p & it contains the string to be parsed followed by $)

Stack is used to contain a sequence of grammar symbols with a $ at the bottom.

- **first()** — Symbol that will be there at first of every derivation from grammar. (Only terminal)

eg. $S \rightarrow aABCD$
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$.

first (S) = a
first (A) = b/e
first (B) = c.

- **follow()** — Which is the terminal that can follow a variable in the process of derivation?

eg. ✓ follow (S) = $.

$S \rightarrow ABCD$.
$A \rightarrow b/\epsilon$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$.

follow (B) = d
Follow (A) = c
follow (C) = e
follow (D) = $

Follow(S) = ~~follow~~ first (A).

— **first and follow**

First and Follow sets are needed so that the parser can properly apply the needed prod^n rule at the correct position.

✓ first(α) is a set of terminal symbols that begin in strings derived from α.

eg. $A \rightarrow abc/def/gh$
first (A) = {a, d, g}

Rules for calculating first().

1. For a production $X \rightarrow \epsilon$,
first (X) = {ϵ}

2. For any terminal 'a',
first (a) = {a}

3. For a production $X \longrightarrow Y_1 Y_2 Y_3$

- First(X)

If $\epsilon \notin$ First (Y_1), then First(X) = First (Y_1)

If $\epsilon \in$ First (Y_1), then First(X) = { First (Y_1) $- \epsilon$ } $\cup$ First $(Y_2 Y_3)$

- First $(Y_2 Y_3)$

If $\epsilon \notin$ First (Y_2), then First $(Y_2 Y_3)$ = First (Y_2)

If $\epsilon \in$ First (Y_2), then

First $(Y_2 Y_3)$ = { First $(Y_2) - \epsilon$ } $\cup$ First (Y_3).

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules for calculating follow()

1. For the start symbol S, place $ in Follow(S).

2. For any production $A \longrightarrow \alpha B$,

Follow (B) = follow (A).

$X \rightarrow tAc$
$A \rightarrow \alpha B$
$X \rightarrow t\alpha Bc$

3. For any production rule $A \rightarrow \alpha B \beta$.

- If $\epsilon \notin$ First (β), then

follow (B) = First (β)

- If $\epsilon \in$ First (β), then

Follow (B) = { First $(\beta) - \epsilon$ } $\cup$ (follow (A)).

NB.

1. ϵ may appear in the first function of a nonterminal. ϵ will never appear in the follow function of a nonterminal.

2. Before calculating the first & follow, eliminate Left recursion from the grammar, if present.

3. We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

e.g.

	First	Follow
$S \rightarrow ABCDE$	{a, b, c}	{$}
$A \rightarrow a / \epsilon$	{a, ε}	{b, c}
$B \rightarrow b / \epsilon$	{b, ε}	{c}
$C \rightarrow c$	{c}	{d, e, $}
$D \rightarrow d / \epsilon$	{d, ε}	{e, $}
$E \rightarrow e / \epsilon$	{e, ε}	{$}

eg.

	First	Follow.
$S \rightarrow Bb / Cd$	{a, b, c, d}	{$}
$B \rightarrow aB / \epsilon$	{a, ε}	{b}
$C \rightarrow cC / \epsilon$	{c, ε}	{d}

eg.

Imp

	First	Follow *
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' / \epsilon$	{+, ε}	{$,)}
$T \rightarrow FT'$	{id, (}	{$,), +}
$T' \rightarrow *FT' / \epsilon$	{*, ε}	{$,), +}
$F \rightarrow id / (E)$	{id, (}	{*, +,), $}

eg.

Imp

	First *	Follow *
$S \rightarrow ACB / CbB / Ba$	{d, g, h, ε, b, a}	{$}
$A \rightarrow da / BC$	{d, g, h, ε}	{h, g, $}
$B \rightarrow g / \epsilon$	{g, ε}	{$, a, h, g}
$C \rightarrow h / \epsilon$	{h, ε}	{g, b, $, h}.

eg.

	First	Follow.
$S \to aABb$.	$\{a\}$	$\{\$\}$
$A \to c / \epsilon$	$\{c, \epsilon\}$	$\{d, b\}$
$B \to d / \epsilon$	$\{d, \epsilon\}$	$\{b\}$

eg.

	First	* Follow
$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{g, f, h\}$
$C \to bC / \epsilon$	$\{b, \epsilon\}$.	$\{g, f, h\}$
$D \to EF$	$\{g, f, \epsilon\}$	$\{h\}$
$E \to g / \epsilon$	$\{g, \epsilon\}$	$\{f, h\}$
$F \to f / \epsilon$.	$\{f, \epsilon\}$	$\{h\}$

eg.

$S \to A$
$A \to aB / Ad$
$B \to b$
$C \to g$

Grammar is left recursive.

After eliminating left recursion,

$S \to A$
$A \to aBA'$
$A' \to dA' / \epsilon$
$B \to b$
$C \to g$

	First	Follow
S	$\{a\}$	$\{\$\}$
A	$\{a\}$	$\{\$\}$ i.e. Follow (S)
A'	$\{d, \epsilon\}$	$\{\$\}$ i.e. Follow (A)
B	$\{b\}$	$\{First(A') - \epsilon\} \cup Follow(A) = \{d, \$\}$
C	$\{g\}$	NA

e.g.

	First	Follow
$S \to (L) / a$	$\{(, a\}$	* $\{\$, , ,)\}$
$L \to SL'$	$\{(, a\}$	$\{)\}$
$L' \to , SL' / \epsilon$.	$\{, , \epsilon\}$	* $\{)\}$

$\{\$\} \cup \{First(L') - \epsilon\} \cup$ Follow (L) $\cup$ Follow(L')

Follow(L)

eg. $S \rightarrow AaAb / BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$.

$First(A) = \{\epsilon\}$ $\quad First(B) = \{\epsilon\}$.

$First(S) = \{First(A) - \epsilon\} \cup First(a) \cup \{First(B) - \epsilon\} \cup First(b) = \{a, b\}$

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

$Follow(A) = First(a) \cup First(b) = \{a, b\}$

$Follow(B) = First(b) \cup First(a) = \{a, b\}$

✓ Eliminating left recursion

eg.

$E \rightarrow E + T / T$

$T \rightarrow T X F / F$

$F \rightarrow (E) / id$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow X FT' / \epsilon$

$F \rightarrow (E) / id$

$First(E) = First(T) = First(F) = \{(, id\}$

$First(F) = \{(, id\}$ $\quad First(E') = \{+, \epsilon\}$

$First(T') = \{X, \epsilon\}$

$First(T) = First(F) = \{(, id\}$

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

$Follow(E') = Follow(E) = \{\$,)\}$

$Follow(T) = \{First(E') - \epsilon\} \cup Follow(E) \cup Follow(E')$

$= \{+, \$,)\}$

$Follow(T') = Follow(T) = \{+, \$,)\}$

$Follow(F) = \{First(T') - \epsilon\} \cup Follow(T) \cup Follow(T')$

$= \{X, +, \$,)\}$.

→ Construction of LL(1) parsing table.

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE' / \epsilon$	{+, ϵ}	{$,)}
$T \to FT'$	{id, (}	{+,), $}
$T' \to *FT' / \epsilon$	{*, ϵ}	{+,), $}
$F \to id / (E)$	{id, (}	{*, +,), $}

put under follow set

Rule:

All null prod^{n}s are put under follow set of the symbol & remaining prod^{n}s lie under first set of the symbol.

LL(1) parsing table –

	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	$F \to id$			$F \to (E)$		

Blank entries → error entry.

$S \to (S) / \epsilon$

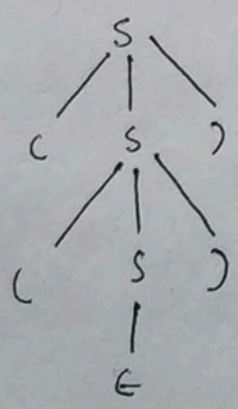
	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

$\omega = (())\$$

Stack

$	S	)	S	(	)	S	(

In the case of more than one entry in a cell, grammar may not be feasible for parsing.



- Any grammar that is left recursive / non-deterministic can't be used for LL(1) parsing.

$S \to AaAb / BbBa$

$A \to \epsilon \qquad B \to \epsilon$

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

Every entry has exactly 1 grammar. So, we can construct parsing table from it.

LL(1) parsing example.

(1) $S \rightarrow aABb$

$A \rightarrow c / \epsilon$

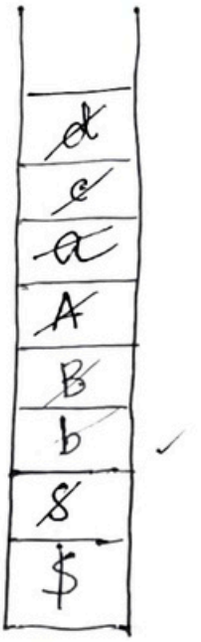
$B \rightarrow d / \epsilon$.

$w = acdb$

LL(1) table

	a	b	c	d	$
S	$S \rightarrow aABb$				
A		$A \rightarrow \epsilon$	$A \rightarrow c$	$A \rightarrow \epsilon$	
B		$B \rightarrow \epsilon$		$B \rightarrow d$	

Stack	Input	Moves
$\$S$	$acdb\$$	$S \rightarrow aABb$
$\$bBAc$	$cdb\$$	matched
$\$bBA$	$cdb\$$	$A \rightarrow c$.
$\$bBc$	$cdb\$$	-
$\$bB$	$db\$$	$B \rightarrow d$
$\$bd$	$db\$$	-
$\$b$	$b\$$	-
$\$$	$\$$	accepted.



(2) $S \rightarrow (S) \mid \epsilon$.

	(	)	$
S	$S \rightarrow (S)$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

$w = (())$.

Stack	Input	Moves
$\$S$	$(())\$$	$S \rightarrow (S)$.
$\$)S($	$(())\$$	-
$\$)S$	$())\$$	$S \rightarrow (S)$
$\$))S($	$(())\$$	-
$\$))S$	$))\$$	$S \rightarrow \epsilon$
$\$))($	$))\$$	-
$\$$	$\$$	acc. ✓

→ Check whether the grammars are LL(1) or not :

1. $S \to aSbS / bSaS / \epsilon$

	a	b	$
S	$S \to aSbS$, $S \to \epsilon$	$S \to bSaS$, $S \to \epsilon$	

2 entries ⇒ Not LL(1)

2. $S \to aABb$
$A \to c / \epsilon$ — c, b, d
$B \to d / \epsilon$ — d, b

LL(1)

3. $S \to A / a$ — a
$A \to a$ — a

Not LL(1) $\{a\} \cap \{a\}$

	a
S	$S \to A$, $S \to a$
A	$A \to a$

Ambiguous.

4. $S \to aB / \epsilon$ — a, $
$B \to bC / \epsilon$ — b, $
$C \to cS / \epsilon$ — c, $

LL(1)

5. $S \to AB$
$A \to a / \epsilon$ — a, $, b
$B \to b / \epsilon$ — b, $

LL(1)

6. $S \to aSA / \epsilon$ — a, c, $
$A \to c / \epsilon$ — c, ~~$~~ c

Not LL(1)

	a	c	$
S	$S \to aSA$	$S \to \epsilon$	$S \to \epsilon$
A		$A \to c$, $A \to \epsilon$	

7. $S \to A$ — Need not to check. One choice
$A \to Bb / Cd$ — a, b, c, d
$B \to aB / \epsilon$ — a, b
$C \to cC / \epsilon$ — c, d

LL(1)

8. $S \rightarrow aAa/\epsilon$ $\underline{a}$ $\$$ $\underline{a}$ Not LL(1)

$A \rightarrow abS/\epsilon$.

9. $S \rightarrow iEtSS'/a$ i, a

$S' \rightarrow eS/\epsilon$ $\underline{e}, e$ Not LL(1).

$E \rightarrow b$

* Recursive Descent Parser. 1 fun^n for every variable

$E \rightarrow iE'$

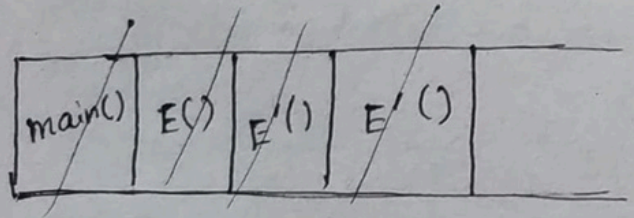
$E' \rightarrow +iE'/\epsilon$.

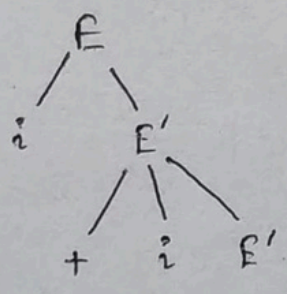
```
match (char t){
    if (l==t)
        l= getchar();
    else
        printf ("error");
}
main (){
    E();
    if (l== '$')
        printf ("Parsing success");
}
```

```
E(){
    if (l== 'i'){          //l = lookahead
        match ('i');          = getchar()
        E'();
    }
}
E'(){
    if (l== '+'){
        match ('+');
        match ('i');
        E'();
    }
    else return;
}
```

Stack by OS used.
No special stack.

eg. $i+i\$$





$\epsilon \rightarrow$ neither var nor terminal

* Operator Precedence Parser.

- Operator grammar.

$E \to E+E \mid E*E \mid id$ ✓

- No prodⁿ in RHS is $\in$.
- 2 variables shouldn't be adjacent to each other.

$E \to EAE \mid id$
$A \to + \mid *$
} ✗ Not a operator grammar.

$S \to SAS \mid a$
$A \to bSb \mid b$
} ✗. $\Rightarrow$ $S \to SbSbS \mid SbS \mid a$, $A \to bSb \mid b$ } ✓

- Operator precedence parser can also work on ambiguous grammars. (using operator relⁿ table).

- Operator relation table -

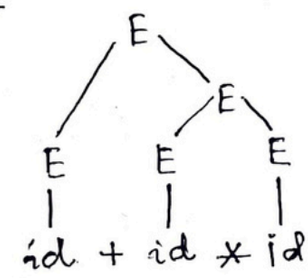
$E \to E+E \mid E*E \mid id$

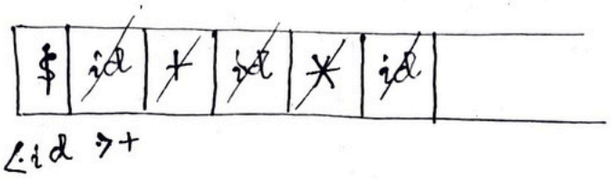
$+ \lessdot *$

left \ right	id	+	*	$
id	=	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	-

$ has least precedence
id has highest precedence

eg. id + id * id $

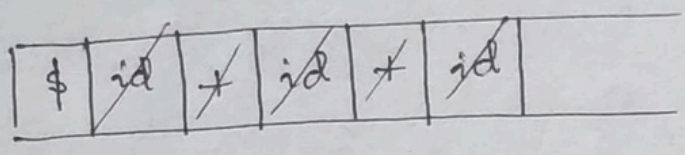


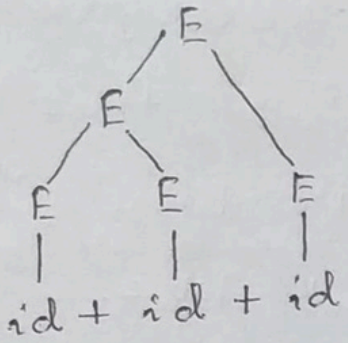


⋖id ⋗+

⋖ push ⋗ pop ⇝ reduce

eg. id + id + id $



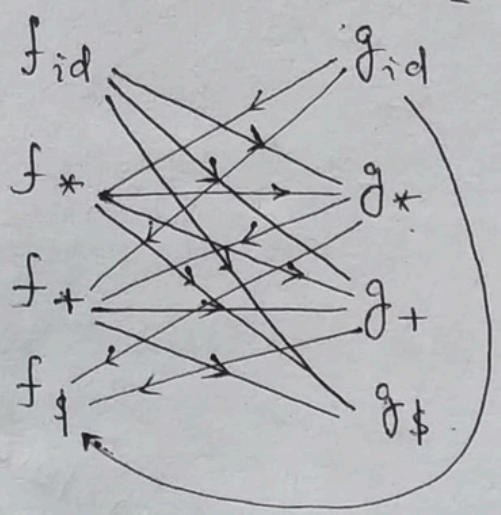


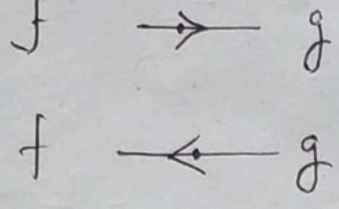
- Size of operator rel^n table $O(n^2)$

n = # operators.

This is a disadvantage.

So, we go for operator fun^n table.





If graph has no cycle, then only proceed.

Longest path starting from node.

$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$ 4

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$ 5

Operator fun^n table $\Rightarrow O(2n) \rightarrow O(n)$ size.

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

lengths of longest path as entry

→ Less size of table.

eg. $\frac{f*}{4} > \frac{g+}{1}$

$* > +$

eg. $\frac{f+}{2} > \frac{g+}{1}$

left + higher precedence

- One disadvantage of fun^n table is even though there are blank entries in operator rel^n table, there is no blank entry in fun^n table.

Error detecting capability of fun^n table is less. (even when no comparison, fun^n table has entry).

eg. $P \to SR/S$

$R \to bSR / bS$ X

$S \to wbS / w$

$w \to L * w / L$

$L \to id$

Not operator grammar

$P \to SbSR / SbS / S$

$P \to SbP / SbS / S$

$S \to wbS / w$

$w \to L * w / L$

$L \to id$

right recursive

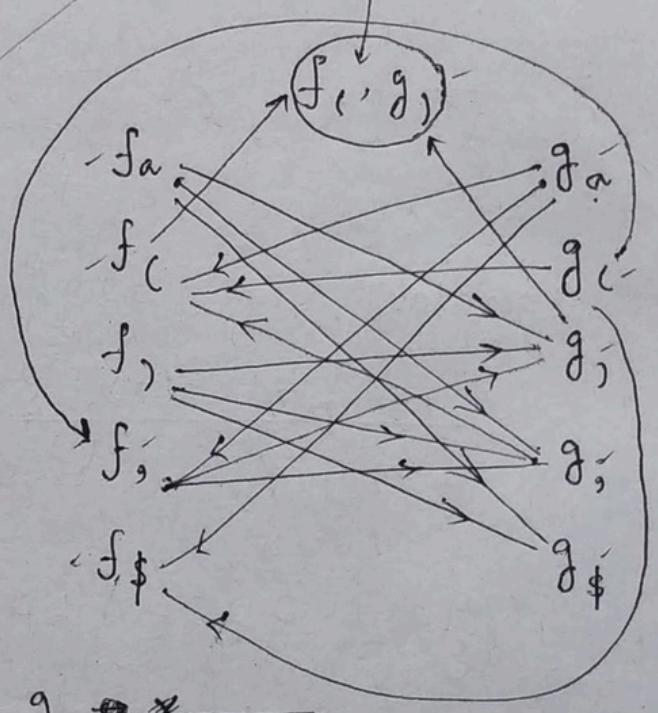
op. rel^n table

	id	*	b	$
id	-	$\dot{>}$	$\dot{>}$	$\dot{>}$
*	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$
b	$\dot{<}$	$\dot{<}$	$\dot{<}$	$\dot{>}$
$	$\dot{<}$	$\dot{<}$	$\dot{<}$	-

eg.

	a	(	)	,	$
a			$\dot{>}$	$\dot{>}$	$\dot{>}$
(	$\dot{<}$	$\dot{<}$	$\doteq$	$\dot{<}$	
)			$\dot{>}$	$\dot{>}$	$\dot{>}$
,	$\dot{<}$	$\dot{<}$	$\dot{>}$	$\dot{>}$	
$	$\dot{<}$	$\dot{<}$			

(and) have same precedence, so merge $f_($ & $g_)$ & make this transition



$f_a \to g_, \to f_(, g_) \to g_a \to f_, \to g_,$ ✗ $\to f_(, g_)$

$f_a \to g_\$$ ✗

$g_a \to f_, \to g_, \to f_(\to f_(, g_)$ ✓

$g_(\to f_, \to g_, \to f_(\to f_(, g_)$ ✓

$f_(\to f_(, g_)$

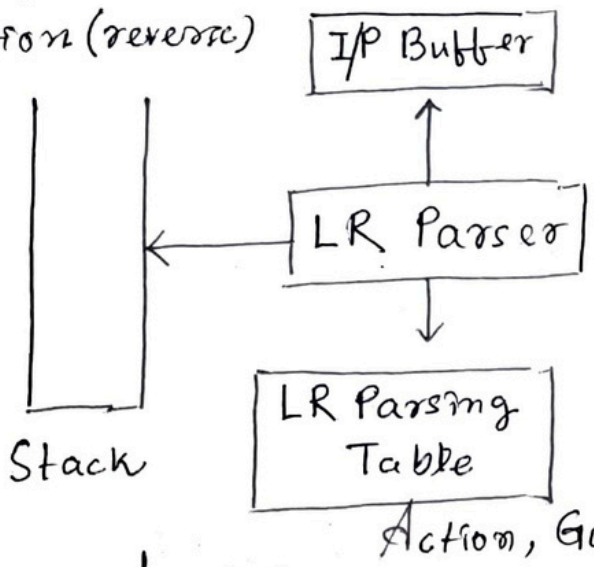
$g_) \to f_(, g_)$

$f_1 \to g_1 \to f_0$ ~~$\to f_0 g_0$~~ Thus,

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0

* LR Parsers. Left-right scanning
Right most derivation (reverse)

- LR(0)
- SLR(1) Simple LR
- LALR(1) Look Ahead LR
- CLR(1) Canonical LR.



LR(0), SLR(1) } Canonical collection of LR(0) items.

LALR(1), CLR(1) } Canonical collection of LR(1) items.

0, 1
No. of input symbols of the look ahead used to make no. of parsing decision.

eg. G
$S \to AA.$
$A \to aA/b.$
grammar

Add one prod^n $S' \to S$ | Augmented grammar.

An LR(0) item is a production G with dot at some position on the right side of the production.
LR(0) items are useful to indicate that how much of the i/p has been scanned up to a given point in the process of parsing. In LR(0), we place the reduce node in the entire row.

Inserting • symbol at the first position for every production in G.

Canonical collection of LR(0) items.
- $S' \to \cdot S$
- $S \to \cdot AA$
- $A \to \cdot aA$
- $A \to \cdot b$

- $S \to \cdot AA$ seen nothing
- $S \to A \cdot A$ seen A
- $S \to AA \cdot$ seen everything

Think like • is passing through string.

When we have seen everything on the RHS of a prod^n reduce it.

Add augment prod^n to the I0 state & compute the closure.

I0 = closure ($S' \rightarrow \cdot S$)

$S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$

Add all prod^{n}s starting with S in to I0 state because $\cdot$ is followed by the non-terminal. So, the I0 state becomes

I0 = $S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$

Add all prod^{n}s starting with A in modified I0 state because $\cdot$ is followed by the non-terminal. So, the I0 state becomes -

I0 = $S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$.

Now, what the $\cdot$ can see next $\rightarrow$ S, A, a, b.

(1) I1 = Goto (I_0, S) = closure ($S' \rightarrow S \cdot$)
= $S' \rightarrow S \cdot$

Prod^n is reduced, so close the state.

(1) I1 = $S' \rightarrow S \cdot$

(2) I2 = Goto (I0, A) = closure ($S \rightarrow A \cdot A$)
= $S \rightarrow A \cdot A$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$.

from I2 { Goto (I2, a) = closure (A → a·A) = (same as I3)
Goto (I2, b) = closure (A → b·) = (same as I4)

③ I3 = Goto (I0, a) = closure (A → a·A).

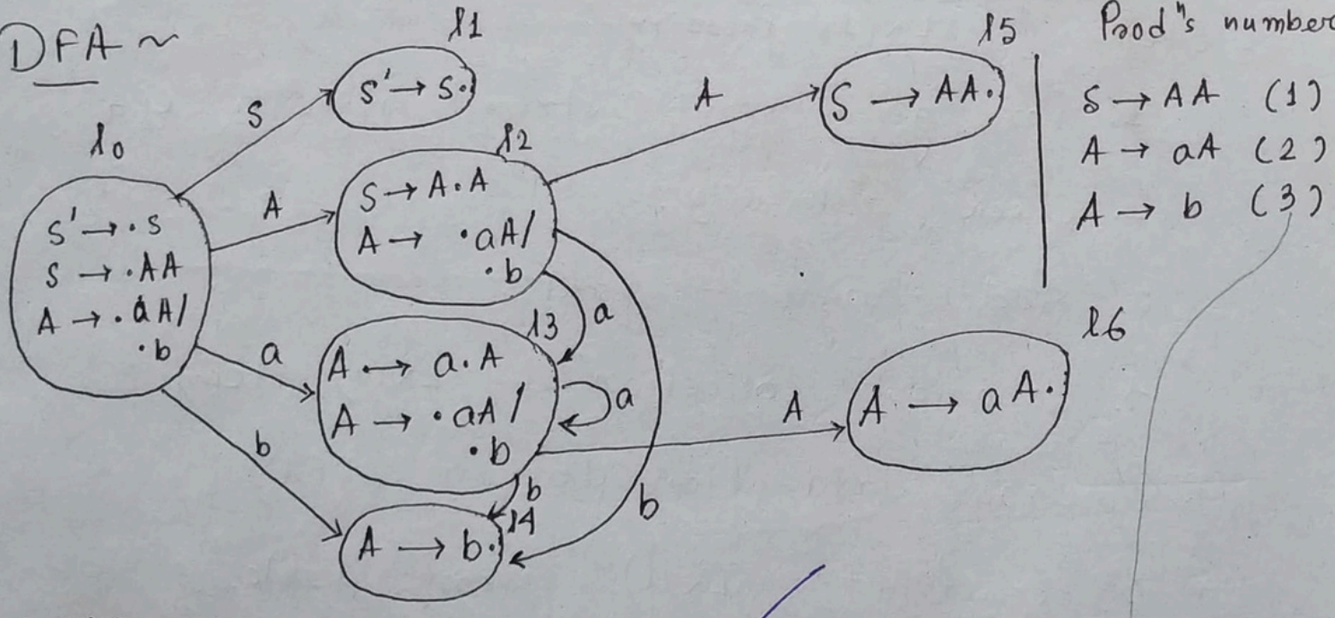
Add prod^n's starting with A in I3.

A → a·A
A → ·aA
A → ·b

Goto (I3, a) = closure (A → a·A) = (same as I3)
Goto (I3, b) = closure (A → b·) = (same as I4)

④ I4 = Goto (I0, b) = closure (A → b·) = A → b·
I5 = Goto (I2, A) = closure (S → AA·) = S → AA·
I6 = Goto (I3, A) = closure (A → aA·) = A → aA·

DFA ~



Prod^n's number

S → AA (1)
A → aA (2)
A → b (3)

LR(0) Table

If a state is going to some other state on a terminal then it corresponds to a shift move (S_K). If a state is going to some other state on a variable then Goto move. If a state contains the final item in the particular row then write reduce node completely.

States	Action			Goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

LR Parsing Example: $w = aabb$.

Stack	Input.
0	aabb $
0a3	abb $
0a3a3	bb $
0a3a3b4	b $
0a3a3b4 (reduce)	b $
0a3a3A6 (reduce. $I_6 \xrightarrow{b} r_2$ reduce using 2nd production)	b $.
0a3A6 ($I_6 \xrightarrow{b} r_2$)	b $.
0A2	b $.
0A2b4 ($I_4 \xrightarrow{\$} r_3$)	$.
0A2A5	$
~~0A2A~~	
0S1	$

Accepted.

Stack has 0 state (I_0) initially. Input ends with $.

See the top of stack (right most of stack) & the leftmost symbol of input.

Suppose, at first (0, a) $I_0 \xrightarrow{a} S_3$ (see table) Then, put the input symbol along with shifted state on top of stack. $\Rightarrow$ 0a3 And eliminate the symbol from input.

Case – St. IP

...4 b$

$I_4 \xrightarrow{b} r_3$

Says to reduce with production no. 3, i.e. $A \to b$.

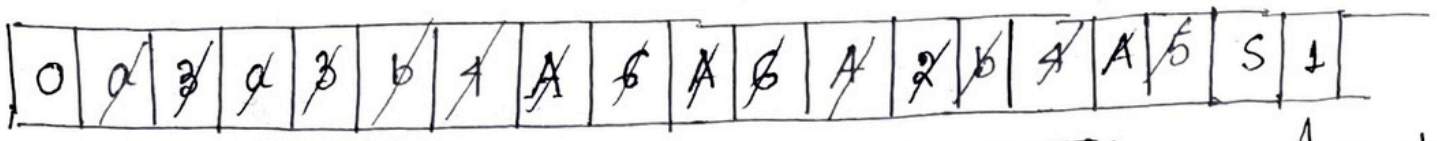
Put A on top of stack. Don't remove b from input yet.

* After putting A, on stack, see last state and based on the table put the goto state. $I_3 \xrightarrow{A} I_6$

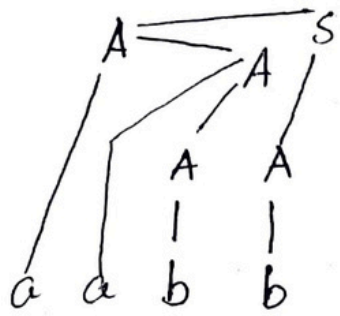
Example – parsing using the LR(0) table.

w = aabb.

a a b b $
↑ ↑ ↑ ↑ ↑



Stack



Accept.

(Example – VC. Joseph p104. Elements of CD).

- SLR(1). Difference in the parsing table with LR(0). To construct SLR(1) table, we use canonical collection of LR(0) items.

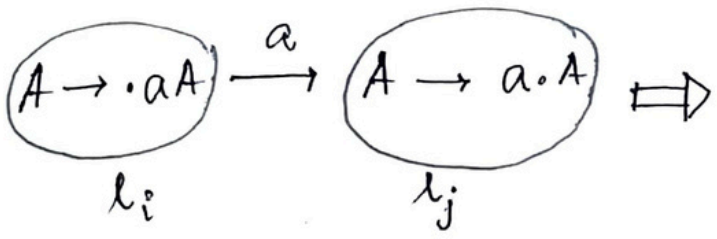
(In the SLR(1) parsing, we place the reduce move only in the follow of left hand side.)

Steps in SLR(1) passing:

1. For given input string, write a CFG.
2. Check ambiguity.
3. Add augment prodⁿ.
4. Canonical collection of LR(0) items.
5. Draw data flow diagram (DFA)
6. Construct SLR(1) parsing table.

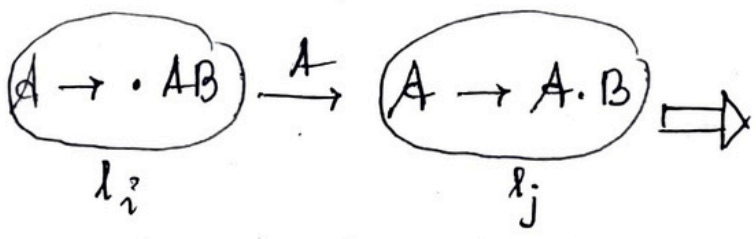
Table construction:

a) If a state (I_i) is going to some other state (I_j) on a terminal - then it corresponds to a shift move in the action part.



State	Action		Goto
	a	$	A
l_i	S_j		
l_j			

b) If a state (l_i) is going to some other state (l_j) on a variable then it corresponds to go to move in the Goto part.

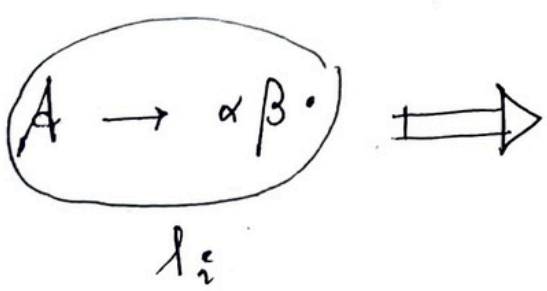


State	Action		Goto
	a	$	A
l_i			j
l_j			

c) If a state (l_i) contains the final item like $A \rightarrow ab.$ which has no transitions to the next state then the prod^n is known as reduce prod^n. For all terminals x on FOLLOW(A), write the reduce entry along with their prod^n numbers.

eg. $S \rightarrow .Aa$ 1
$A \rightarrow a\beta.$ 2

Follow (S) = {$}
Follow (A) = {a}.



State	Action			Goto	
	a	b	$	S	A
l_i	r_2				

eg. Previous one

$S \rightarrow AA$ (1)
$A \rightarrow aA$ (2)
$A \rightarrow b$. (3)

$S' \rightarrow S$

I4 $A \rightarrow b.$
I5 $S \rightarrow AA.$
I6 $A \rightarrow aA.$

Follow (A) = {a, b, $}.
Follow (S) = {$}

~~I0~~

States	Action			Goto	
	a	b	$	A	S
I0	S_3	S_4		2	1
I1			acc.		
I2	S_3	S_4		5	
I3	S_3	S_4		6	
I4	r_3	r_3	r_3		
I5			r_1		
I6	r_2	r_2	r_2		

→ If a state contains 2 reduce items, it's a reduce/reduce (r/r) conflict.

I5

$A \rightarrow a.$
$B \rightarrow b.$

r/r conflict.

If some column in the action part (a terminal for a state) contains both shift and reduce moves, it's a shift-reduce (s/r) conflict.

$A \rightarrow b.$
$X \rightarrow X.a$

s/r conflict

I6

Transition on a reduce by $A \rightarrow b$.

→ To avoid some s/r & r/r conflicts, SLR(1) maintains Follow(LHS).

→ $A \to a.$
$B \to b.$

It's an r/r conflict only if Follow(A) and Follow(B) sets intersect, otherwise it's not a conflict.

$$\text{Follow}(A) \cap \text{Follow}(B) \neq \phi \Rightarrow \text{Conflict.}$$

---

eg.

$S \to dA / aB$

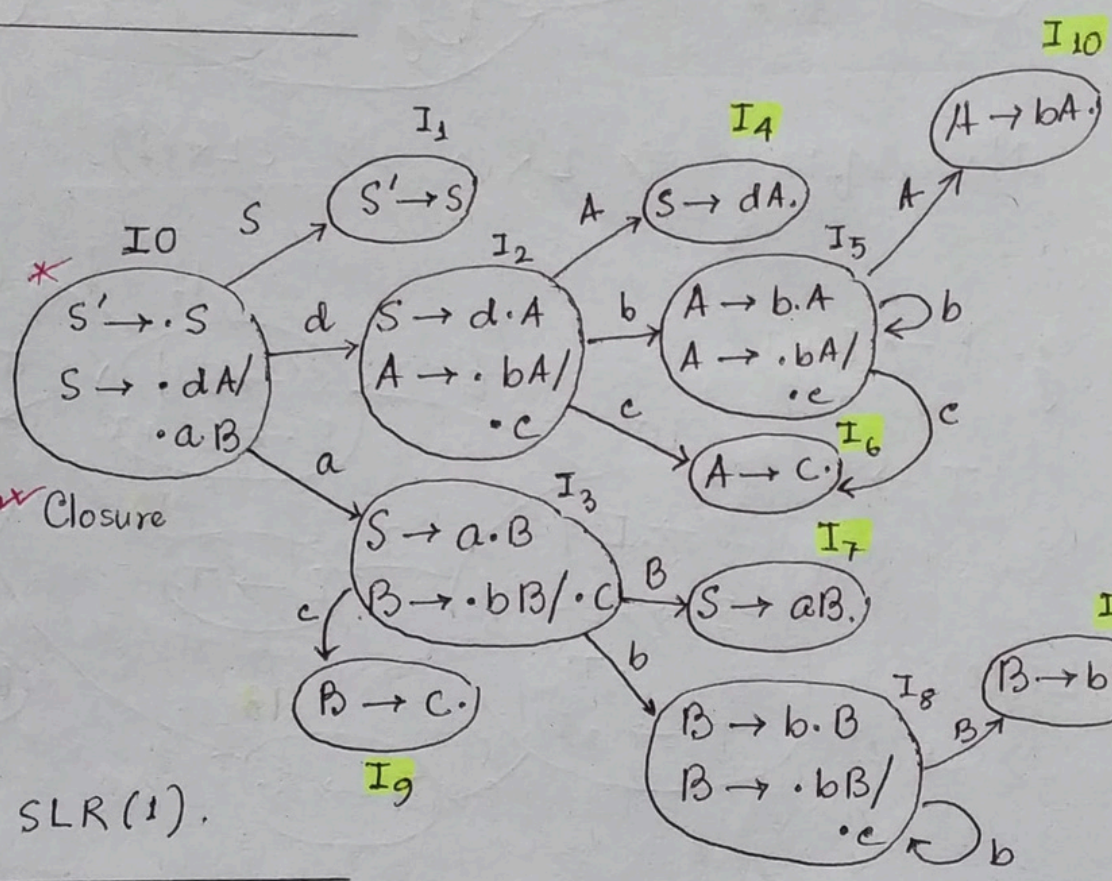
$A \to bA / c$

$B \to bB / c$

LL(1) ✓

No conflicts

$\Rightarrow$ LR(0) $\Rightarrow$ SLR(1).



---

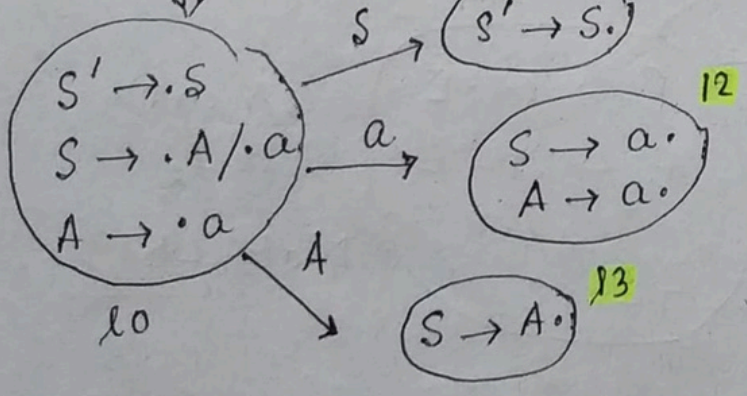
eg.

$S \to A / a$

$A \to a$

Ambiguous.

Not LL(1).



Not LR(0) (12 r/r conflict)

follow(S) = {$}

∩

follow(A) = {$}

≠

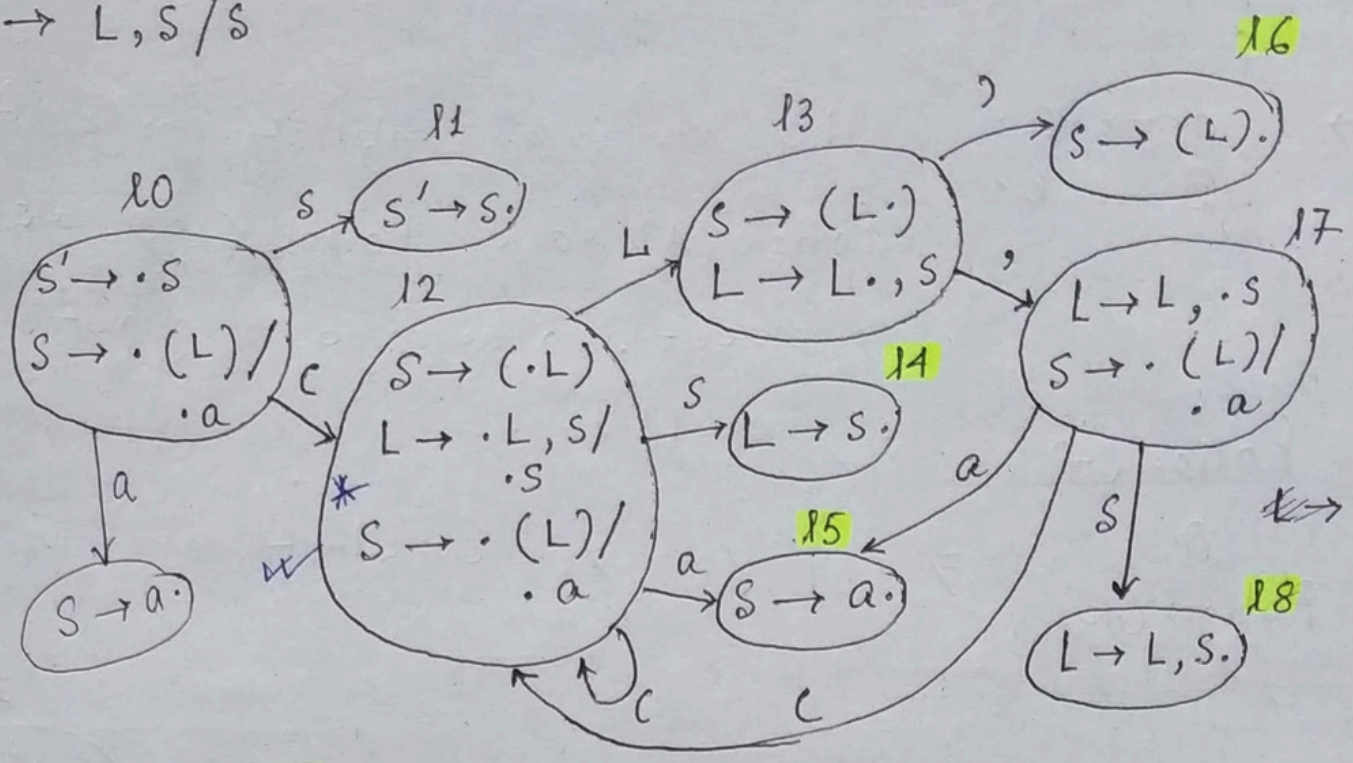
ϕ.

$\Rightarrow$ Not SLR(1).

eg.

$S \to (L) / a$ Not LL(1) as left recursive.

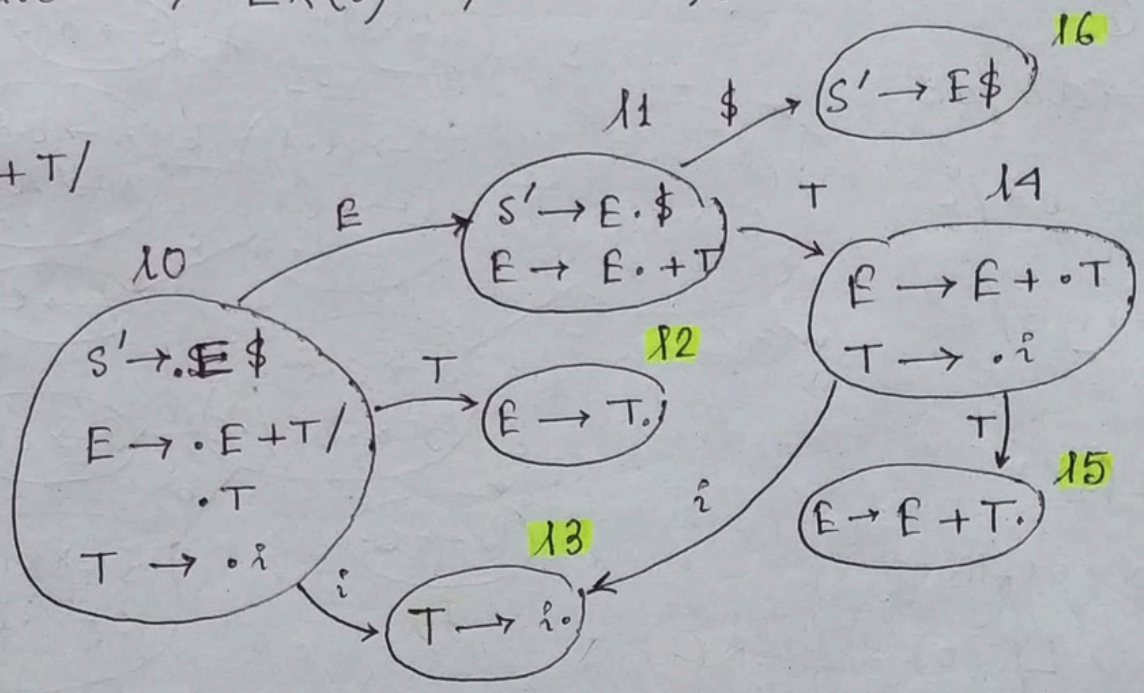
$L \to L, S / S$



No conflict ⇒ LR(0) ⇒ SLR(1).

eg. $E \to E+T /$
T
$T \to i$

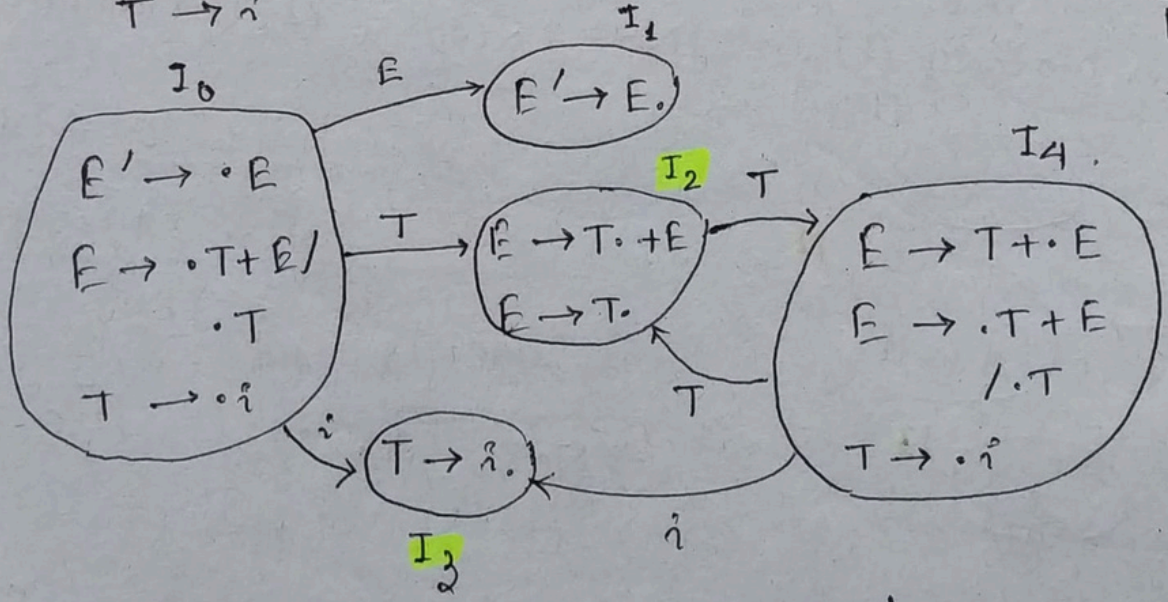
LR(0)
⇒ SLR(1)



eg. $E \to T+E / T$ Not LL(1) as double entry

$T \to i$

	+	i
E		$\{E \to T+E$ $E \to T$
T		$T \to i$



s/r conflict in I_2

Not LR(0)

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

r_2 under $

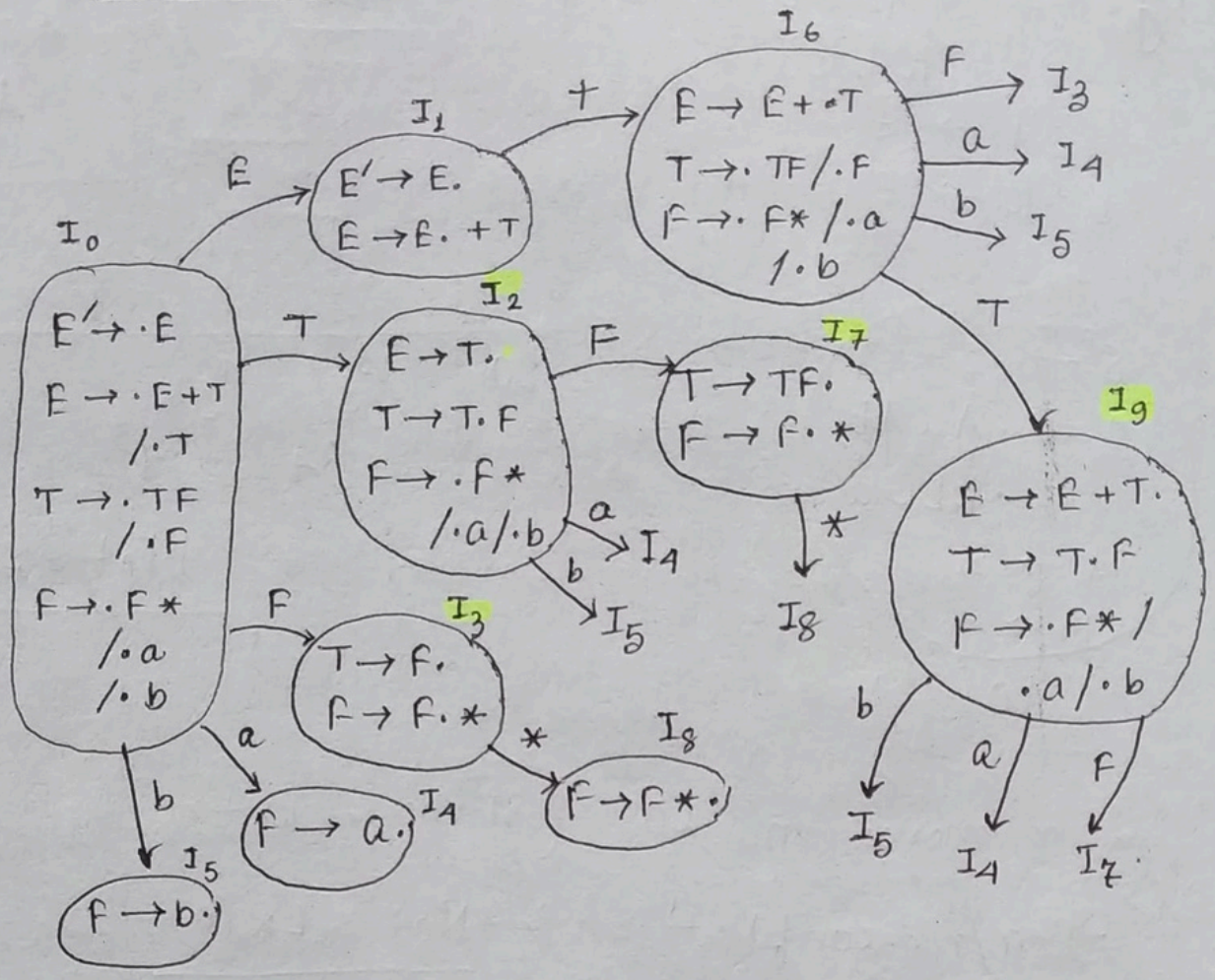
No s/r conflict.

SLR(1) ✓

eg.

Not LL(1) as left recursive.

$E \to E+T$ 1
$\quad /T$ 2
$T \to TF$ 3
$\quad /F$ 4
$F \to F*$ 5
$\quad /a$ 6
$\quad /b$ 7



In I_2, $E \to T.$ Final item.

$T \to T.F$ Shift move (not conflict, as in the Goto part)

$F \to .F*$ Shift move (not conflict, in Goto)

$F \to .a$ Conflict (in action part). S/R conflict

$\Rightarrow$ Not LR(0).

$\begin{cases} E \to T. \\ F \to .a \end{cases}$

Follow(E) = {+, $}

Follow(T) = {+, $, a, b}

Follow(F) = {+, *, $, a, b}.

For SLR(1), conflict happens in action part.

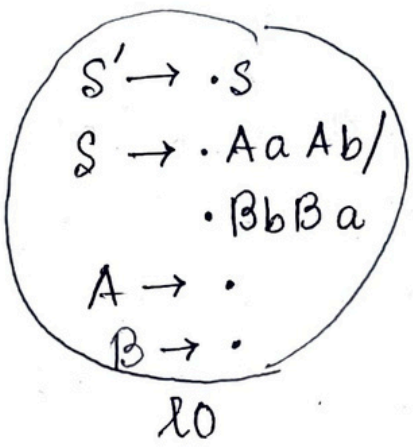
Inadequate States	a	b	+	*	$	
I_2	S_4	S_5	r_2		r_2	No conflict.
I_3	r_4	r_4	r_4	S_8	r_4	"
I_7	r_3	r_3	r_3	S_8	r_3	"
I_9	S_4	S_5	r_1		r_1	". Beauty!

SLR(1) ✓

eg. $S \to AaAb / 1$
$\quad BbBa \quad 2$
$A \to \epsilon \quad 3$
$B \to \epsilon \quad 4$

G is LL(1) ✓

$A \to \cdot\epsilon$ }
$A \to \epsilon\cdot$ }
$A \to \cdot$



	a	b	$
I0	r_3/r_4	r_3/r_4	

conflict

Not SLR(1)

2 r moves in I0 ($A \to \cdot$, $B \to \cdot$)

$\Rightarrow$ r/r conflict $\Rightarrow$ Not LR(0).

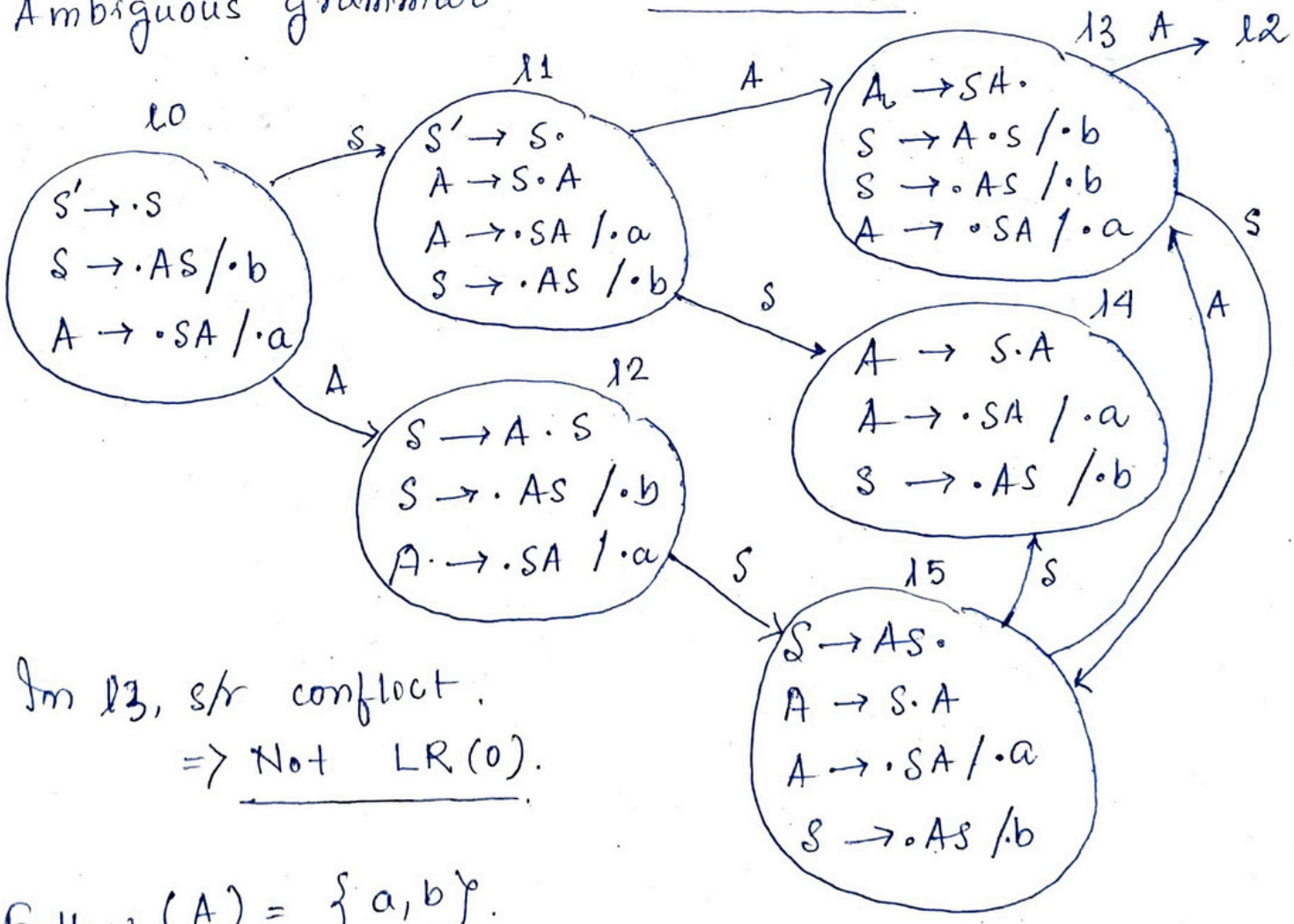
eg. $S \to As / b$
$A \to SA / a.$

Ambiguous grammar

	a	b
S	$S \to AS$	$S \to AS$ / $S \to b$

two ~~two~~ entries.

Not LL(1)



In I3, s/r conflict.
$\Rightarrow$ Not LR(0).

Follow(A) = {a, b}.

In I3, $A \to SA\cdot$ on a, b } conflict.
$S \to \cdot b$ on b.

Not SLR(1).

eg. Not LL(1) — 1 and 3 prod^{n}s in same cell (S, d)

$S \to Aa$ 1
$/ bAC$ 2
$/ dc$ 3
$/ bda$ 1
$A \to d$ 5

I_0:
$S' \to .S$
$S \to .Aa$
$/ .bAc$
$/ .dc$
$/ .bda$
$A \to .d$

$\xrightarrow{d}$

I_1:
$S \to d.c$
$A \to d.$

$c \in Follow(A)$

	a	b	c	d	$
I_1	r		s/r		

Not LR(0). conflict in I_1.

Not SLR(1).

14

- **CLR(1) Parsing**

CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR(1) items to build the CLR(1) parsing table. CLR(1) parsing table produces more number of states as compared to the SLR(1) parsing. In the CLR(1), we place the reduce node only in the lookahead symbols.

LR(1) item

LR(1) item is a collection of LR(0) items & a look-ahead symbol.

LR(1) item = LR(0) item + lookahead ✓

Lookahead is used to determine that where we place the final item. The lookahead always add $ symbol for the augment production.

LR item : $A \to \alpha.\beta$, a Lookahead 'a' has no effect where β is not $\in$. But an item of the form $[A \to \alpha., a]$ calls for a reduction by $A \to \alpha$ only if next symbol (r/p) is a.

Set of such 'a's will be a subset of follow(A), but it could be a proper subset (less possibility of conflict).

LR(1) item :

If $A \longrightarrow \alpha \cdot B\beta, a$ is in closure(I) and $B \longrightarrow \gamma$ is a production rule of G ; then $B \longrightarrow \cdot\gamma, b$ will be in the closure(I) for each terminal b in first (βa).

$A \longrightarrow \cdot\, B\underline{C, a}$ $\qquad C \longrightarrow \underline{+P}$

$B \longrightarrow \cdot\gamma, +$ ___ First (Ca)

eg. $S \to AA$ (1)
$A \to aA/b$
(2) (3)

$$\begin{cases} S' \to \cdot S, \$ \\ S \to \cdot AA, \$ \\ A \to \cdot aA, a/b \\ A \to \cdot b, a/b \end{cases}$$

I0

I0 = closure $(S' \to \cdot S)$

$= S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b.$

I1 I1 = Goto (I0, S) = closure $(S' \to S\cdot, \$)$

$= S' \to S\cdot, \$$

I2 I2 = Goto (I0, A) = closure $(S \to A\cdot A, \$)$

$= S \to A\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

I3 I3 = Goto (I0, a) = closure $(A \to a\cdot A, a/b)$

$= A \to a\cdot A, a/b$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b$

Goto (I3, a) = closure $(A \to a\cdot A, a/b)$ = (same as I3)

Goto (I3, b) = closure $(A \to b\cdot, a/b)$ = (same as I4)

I4 I4 = Goto (I0, b) = closure $(A \to b\cdot, a/b)$

$= A \to b\cdot, a/b$

I5 I5 = Goto (I2, A) = closure $(S \to AA\cdot, \$)$

$= S \to AA\cdot, \$$

I6 I6 = Goto (I2, a) = closure $(A \to a\cdot A, \$)$

$= A \to a\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

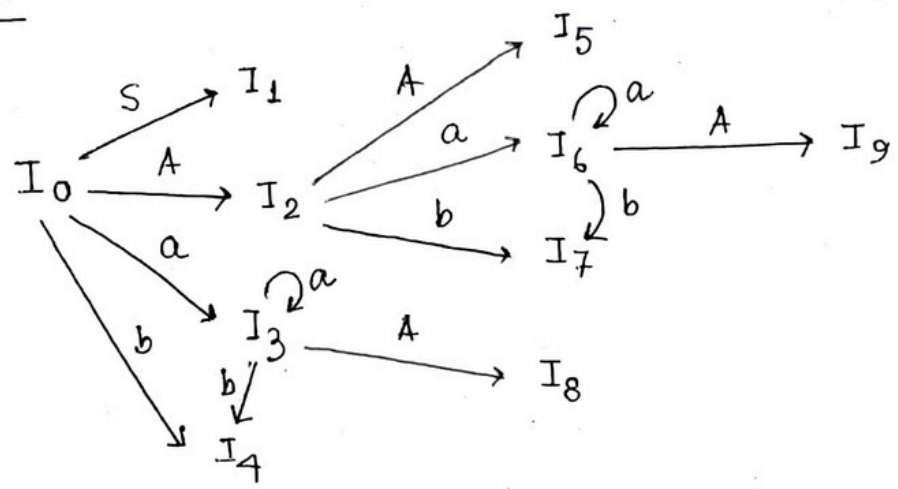
Goto (I6, a) = closure $(A \to a\cdot A, \$)$ = (same as I6)

Goto (I6, b) = closure $(A \to b\cdot, \$)$ = (same as I7)

- I7 = Goto (I2, b) = closure (A → b·, $)
  = A → b·, $
- I8 = Goto (I3, A) = closure (A → aA·, a/b)
  = A → aA·, a/b
- I9 = Goto (I6, A) = closure (A → aA·, $)
  = A → aA·, $

DFA

Data flow diagram.



CLR(1) parsing table.

States	Action a	Action b	Action $	Goto S	Goto A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
✓ I_4	R_3	R_3			
✓ I_5			R_1		
I_6	S_6	S_7			9
✓ I_7			R_3		
✓ I_8	R_2	R_2			
✓ I_9			R_2		

action (I_4, a) = R_3
action (I_4, b) = R_3
action (I_5, $) = R_1
action (I_6, a) = S_6
action (I_7, $) = R_3
action (I_8, a) = R_2
action (I_8, b) = R_3
action (I_9, $) = R_2.

- Placement of shift nodes in CLR(1) parsing table is same as the SLR(1) parsing table. Only difference is in the placement of reduce node.

- **LALR(1) Parsing**

LALR refers to lookahead LR.

The LR(1) items which have same prod^{n}s but different lookahead are combined to form a single set of items. LALR(1) and CLR(1) parsing are same, only difference in the parsing table.

eg. Previous. $S \rightarrow AA \quad A \rightarrow aA / b.$

I_3 and I_6 are same in their LR(0) items but differ in their lookahead. So, we combine.

$$I_{36} = \{ A \rightarrow a \cdot A, \ a/b/\$ \\ A \rightarrow \cdot aA, \ a/b/\$ \\ A \rightarrow \cdot b, \ a/b/\$ \}$$

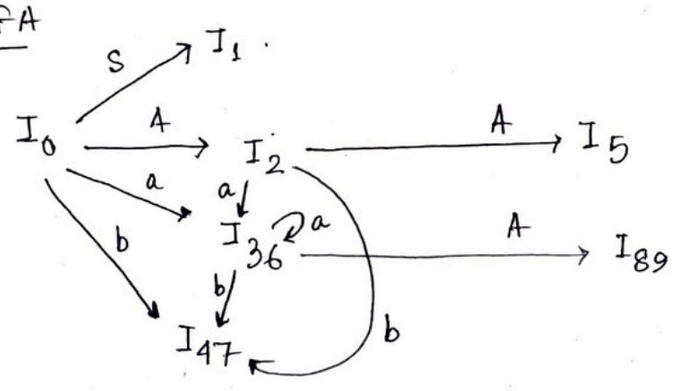
Same for I_4, I_7

$I_{47} = \{ A \rightarrow b \cdot, a/b/\$ \}$

Same for I_8, I_9

$I_{89} = \{ A \rightarrow aA \cdot, a/b/\$ \}$

DFA



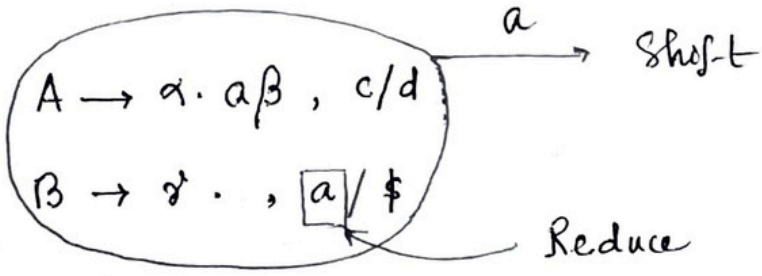
Reduced no. of states than CLR(1).

LALR(1) parsing table.

States	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		

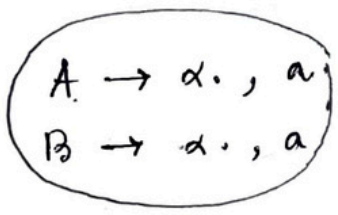
- Conflicts in CLR(1) and LALR(1).

LR(1) items



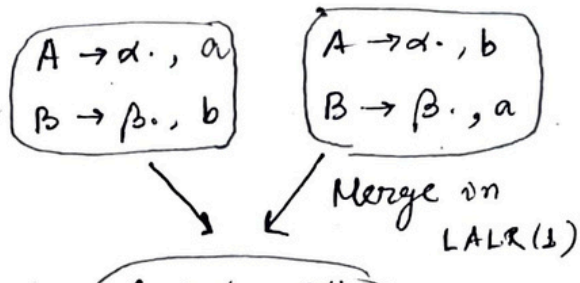
If grammar is not CLR(1) then it is not LALR(1). Because, we reduce the size of the table but not the conflicts in LALR(1) parser.

S/R conflict



R/R conflict

If grammar is CLR(1), then it may or may not be LALR(1).



eg. $S \rightarrow AaAb \,/\, BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$.

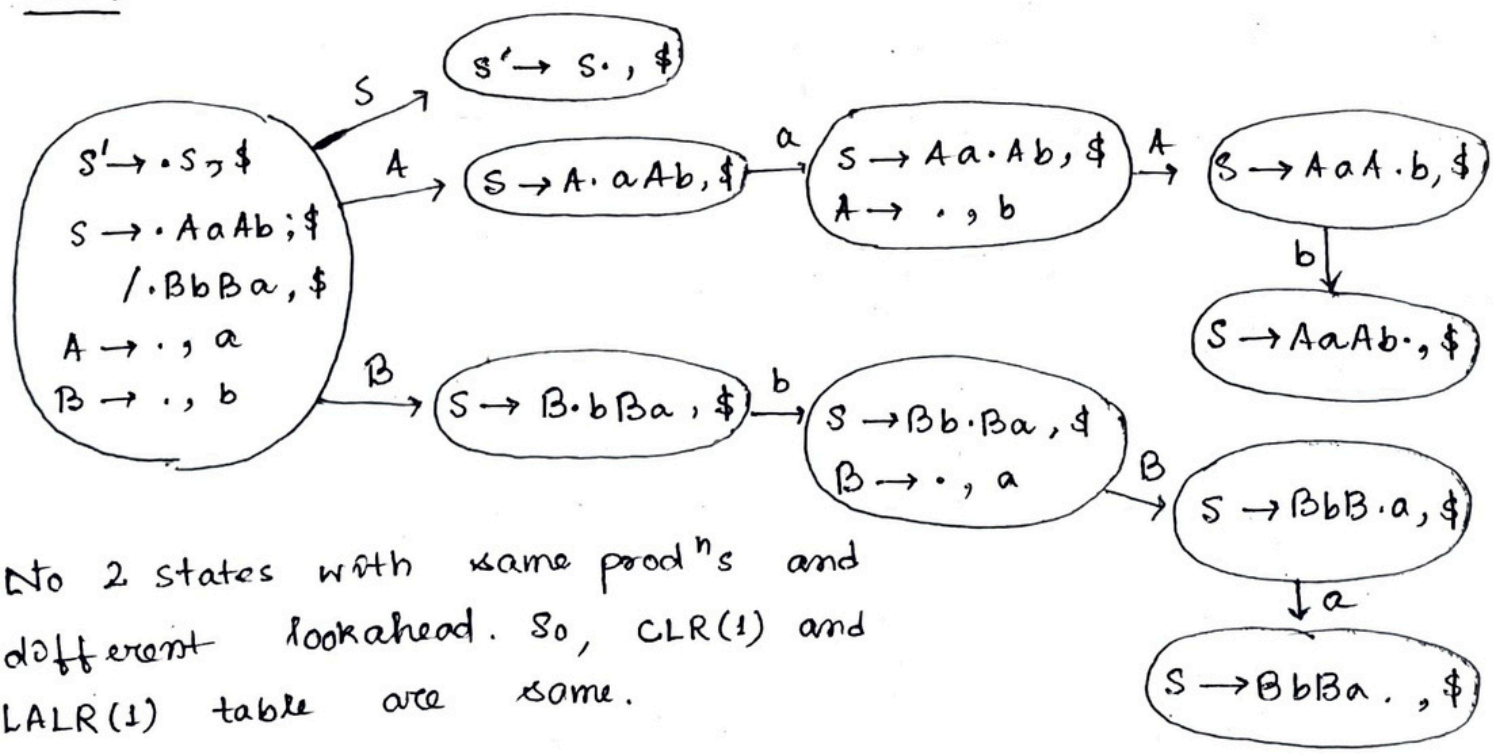
LL(1) ✓
LR(0) X
SLR(1) X
CLR(1) ✓
LALR(1) ✓

LR(0)

$S' \rightarrow \cdot S$
$S \rightarrow \cdot AaAb \,/\, \cdot BbBa$
$A \rightarrow \cdot$ placed under follow(A) = {a,b}
$B \rightarrow \cdot$ placed in follow(B) = {a,b}

Not SLR(1)

LR(1)

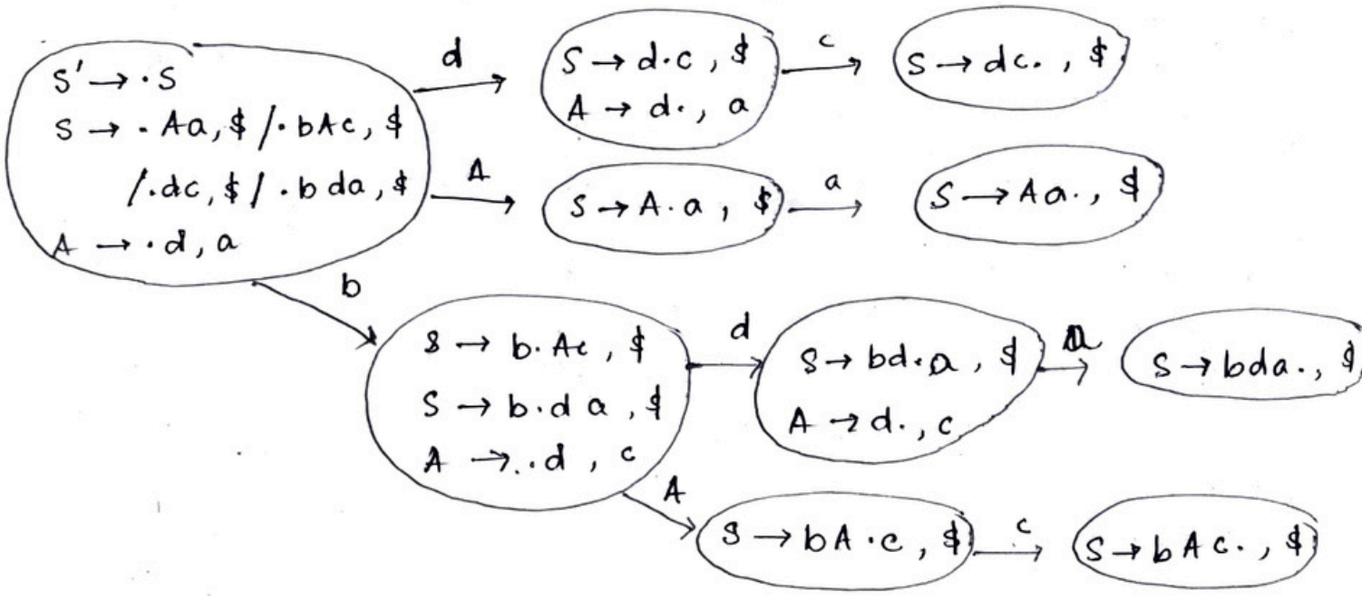


No 2 states with same prod^{n}s and different lookahead. So, CLR(1) and LALR(1) table are same.

No conflicts.

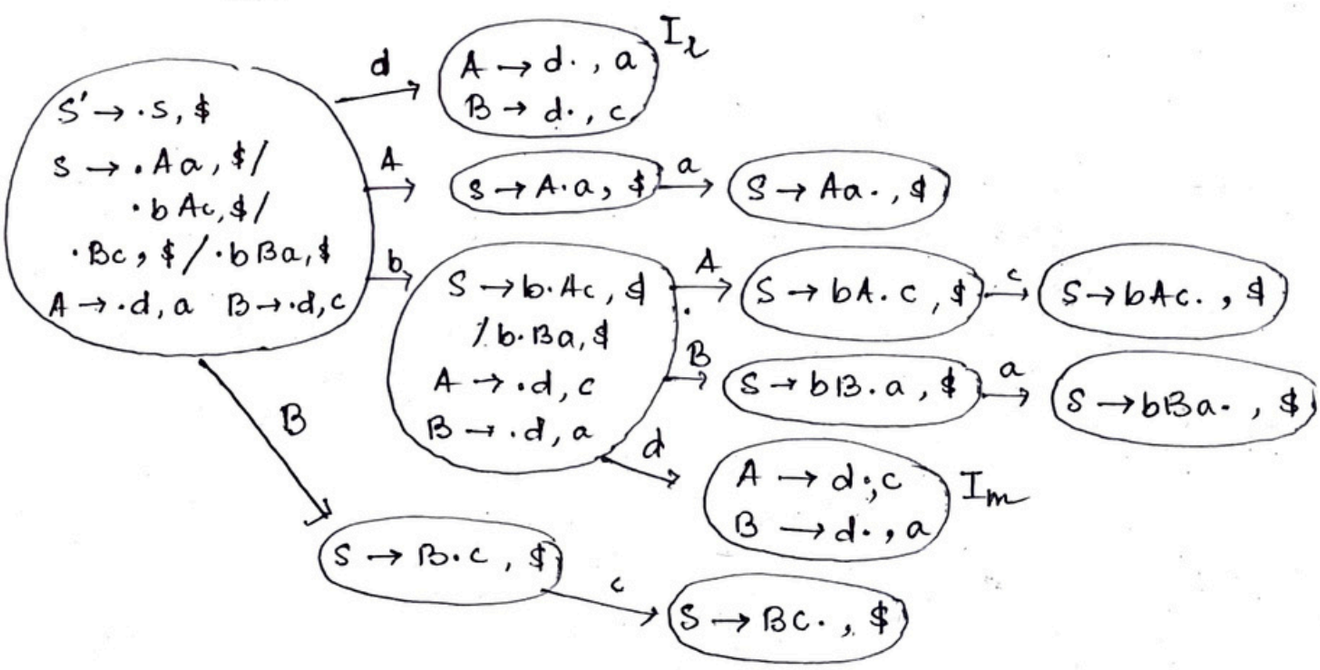
eg $S \to Aa / bAc / dc / bda$
$A \to d$

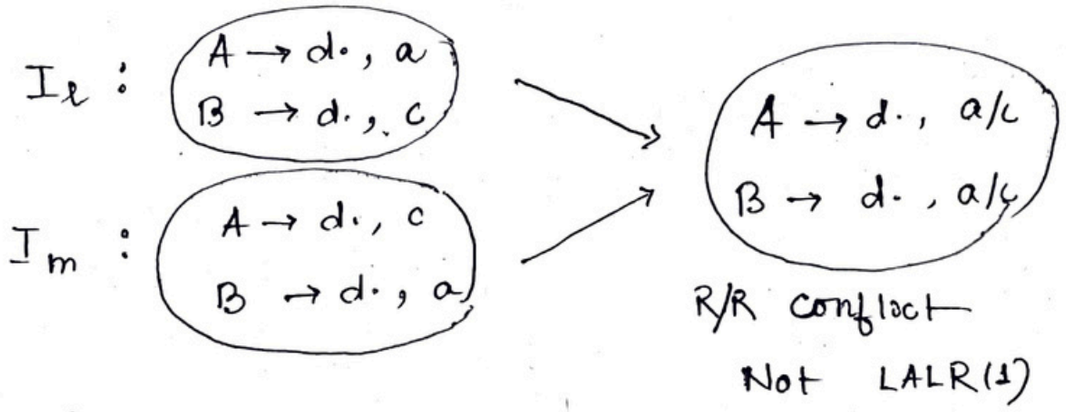
LL(1) ✗	CLR(1) ✓
LR(0) ✗	LALR(1) ✓
SLR(1) ✗	



eg. $S \to Aa / bAc / Bc / bBa$
$A \to d \quad B \to d$

LL(1) ✗	SLR(1) ✗
LR(0) ✗	LALR(1) ✗
CLR(1) ✓	

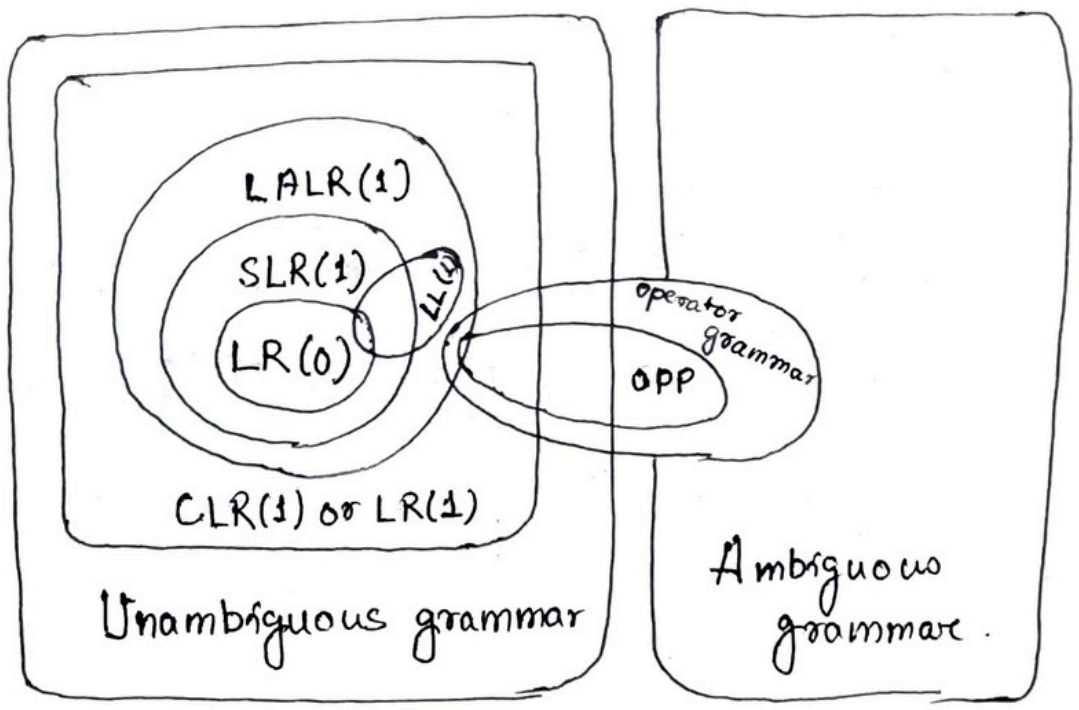


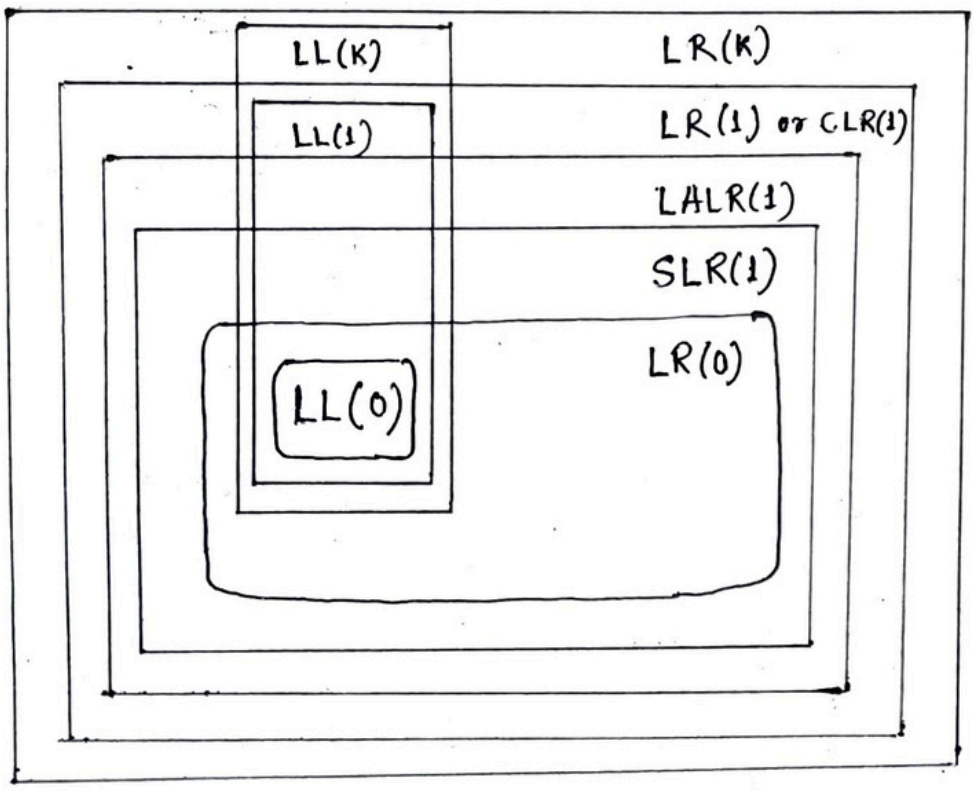


R/R conflict

Not LALR(1)

* Comparison of the parsers.





- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ CLR(1)
- LL(1) ⊂ LR(1) or CLR(1)
- If, for a grammar G, LALR(1) can be constructed then CLR(1) parser can also be constructed.
- If CLR(1) grammar parser can be constructed for a grammar, then we may or may not construct LALR(1) parser.
- No. of entries in the LALR(1) parser ≤ no. of entries in the CLR(1) parser.

- No. of entries in the LALR(1) parsing table is equal to the no. of entries in the SLR(1) parsing table.

- #states (SLR(1)) = #states (LALR(1))
  $\leq$ # states (CLR(1))

- CLR(1) parsers are more powerful, efficient than any other parser.

eg $S \to A$
$A \to AB/\epsilon$
$B \to aB/b$

Closure ($S' \to \cdot S, \$$).

Closure ($S \to \cdot A, \$$)
$S' \to \cdot S, \$$
$S \to \cdot A, \$$
$A \to \cdot AB, \$$
$\quad /\cdot, \$$

$\Rightarrow$

$S' \to \cdot S, \$$
$S \to \cdot A, \$$
$A \to \cdot AB, \$/a/b$
$\quad /\cdot, \$/a/b$

$A \to \cdot AB, a/b$
$\quad /\cdot, a/b$

→ We have to add them too because there can be new lookaheads for the prodn $A \to \cdot AB, \$$ $/\cdot, \$$. So, we add new lookaheads a, b as these are

First ($B\$$) = First (B).

NOTE

LL(k): For a grammar to be LL(k), we must be able to recognise the use of a production by seeing only the first k symbols of what its RHS derives.

LR(k): For a grammar to be LR(k), we must be able to recognise the use of a production by having seen all of what is derived from its RHS with k more symbols of lookahead.

eg. $E \to E+T/T \quad T \to T*F/F \quad F \to i$

$E' \to \cdot E, \$$

$E \to \cdot E+T, \$$ — first $(+T\$) = +$

$/ \cdot T, \$$

$E \to \cdot E+T, +$

$/ \cdot T, +$

$T \to \cdot T*F, \$/+$ — first $(*F\$+) = *$

$/ \cdot F, \$/+$

$T \to \cdot T*F, *$

$/ \cdot F, *$

$F \to \cdot i, \$/+/*$

$\Rightarrow$

$E' \to \cdot E, \$$

$E \to \cdot E+T, \$/+$

$/ \cdot T, \$/+$

$T \to \cdot T*F, \$/+/*$

$/ \cdot F, \$/+/*$

$F \to \cdot i, \$/+/*$

. E → $E' \to E\cdot, \$$; $E \to E\cdot+T, \$/+$

T → $E \to T\cdot, \$/+$; $T \to T\cdot*F, \$/+/*$

F → $T \to F\cdot, \$/+/*$

i → $F \to i\cdot, \$/+/*$

\+ → $E \to E+\cdot T, \$/+$; $T \to \cdot T*F, \$/+/*$; $/ \cdot F, \$/+/*$; $F \to \cdot i, \$/+/*$

T → $E \to E+T\cdot, \$/+$; $T \to T\cdot*F, \$/+/*$

OPP $<$ LL(1) $<$ LR(0) $<$ SLR(1) $\leq$ LALR $\leq$ CLR(1).

+(CD folder)
photo

Q. Consider SLR(1) and LALR(1) tables for a CFG. Then

(a) Goto of both tables may be different

✓(b) Shift entries are identical in both.

✓(c) Reduce entries in tables may be different

✓(d) Error entries in tables may be different.

Q. $S \to CC$
$C \to cC \mid d$

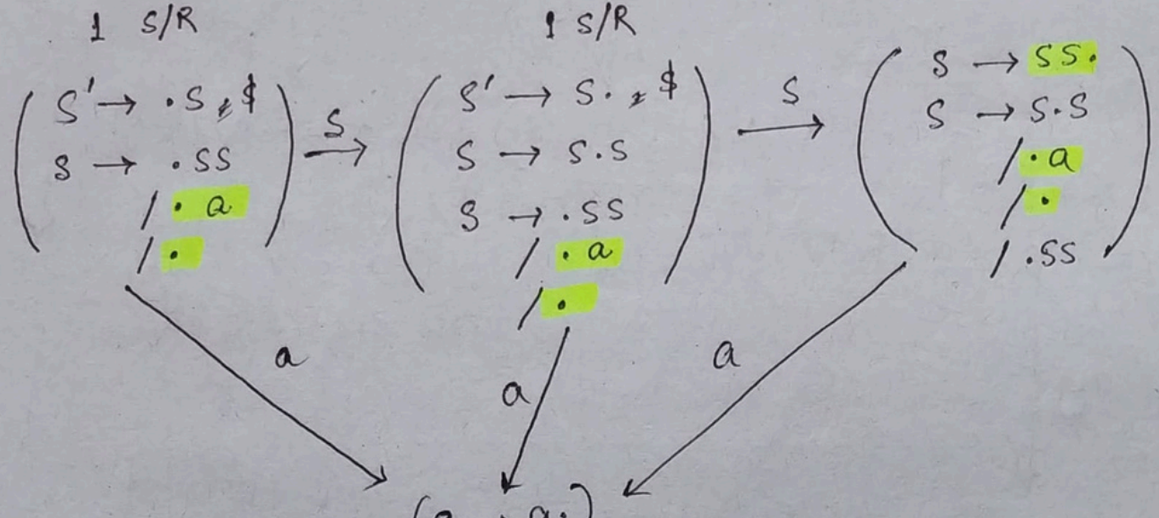
✓a) LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) CLR(1) but not LALR(1)

Q Find # SR, RR conflicts in the DFA with LR(0) items.

$S \to \cdot SS$
$\mid a$
$\mid \epsilon$

or # inadequate states

1 S/R 1 RR



$(S \to a.)$

3 S/R & 1 R/R conflict

4 inadequate states.