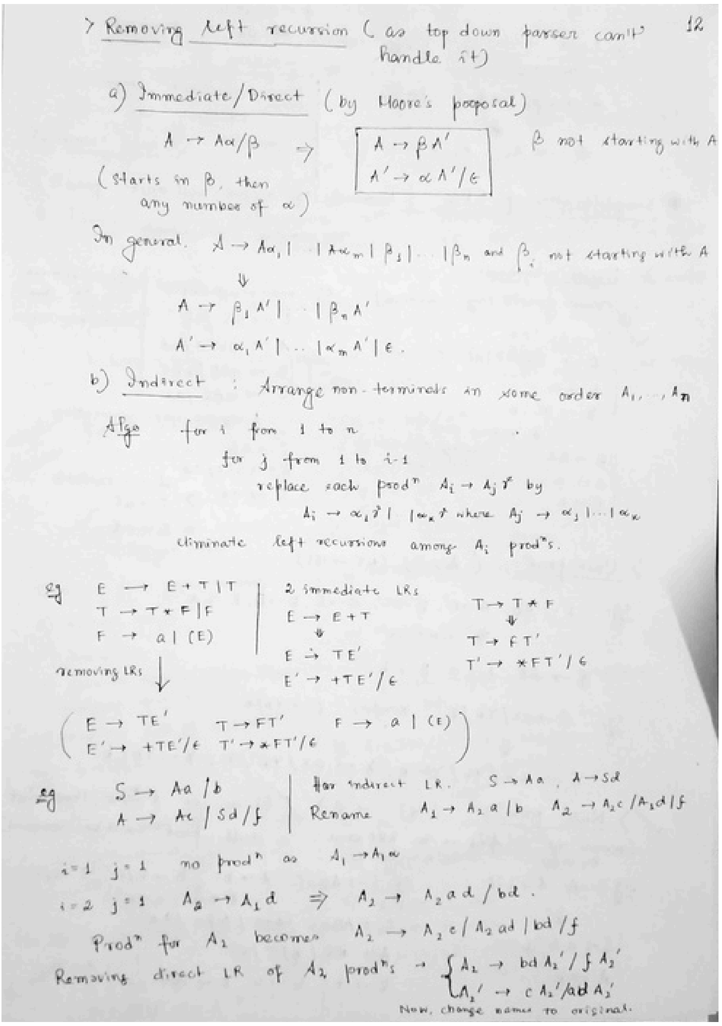
GATE CSE NOTES

by

UseMyNotes

- **Lexical errors:**Lexical error is a sequence of charactersthat does not match the pattern of any token.
- For unambiguous grammars, Leftmost derivation andRightmost derivation represent the same parse tree.
- For ambiguous grammars, Leftmost derivation and Rightmostderivation represent different parse trees.
- Concatenating the leaves of a parse tree from theleft produces a string of terminals. This string ofterminals is called the**yield of a parse tree**.

- **Eliminating Left Recursion:**https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/



> Removing left recursion (as top down parser can't handle it) 12

a) Immediate/Direct (by Moore's proposal)

$A \to A\alpha / \beta \Rightarrow \boxed{A \to \beta A' \quad A' \to \alpha A' / \epsilon}$ β not starting with A

(starts in β, then any number of α)

In general, $A \to A\alpha_1 | \ldots | A\alpha_m | \beta_1 | \ldots | \beta_n$ and β_i not starting with A

$\Downarrow$

$A \to \beta_1 A' | \ldots | \beta_n A'$

$A' \to \alpha_1 A' | \ldots | \alpha_m A' | \epsilon$.

b) Indirect : Arrange non-terminals in some order $A_1, \ldots, A_n$

Algo for i from 1 to n
 for j from 1 to i-1
 replace each prod^n $A_i \to A_j \gamma$ by
 $A_i \to \alpha_1 \gamma | \ldots | \alpha_k \gamma$ where $A_j \to \alpha_1 | \ldots | \alpha_k$
 eliminate left recursions among A_i prod^{n}s.

eg $E \to E+T | T$, $T \to T*F | F$, $F \to a | (E)$ | 2 immediate LRs: $E \to E+T \Downarrow E \to TE'$, $E' \to +TE'/\epsilon$; $T \to T*F \Downarrow T \to FT'$, $T' \to *FT'/\epsilon$

removing LRs $\downarrow$

$\left(E \to TE' \quad E' \to +TE'/\epsilon \quad T \to FT' \quad T' \to *FT'/\epsilon \quad F \to a | (E) \right)$

eg $S \to Aa / b$, $A \to Ac / Sd / f$ | Has indirect LR. $S \to Aa$, $A \to Sd$. Rename $A_1 \to A_2 a / b$, $A_2 \to A_2 c / A_1 d / f$

$i=1$ $j=1$ no prod^n as $A_1 \to A_1 \alpha$

$i=2$ $j=1$ $A_2 \to A_1 d \Rightarrow A_2 \to A_2 ad / bd$.

Prod^n for A_2 becomes $A_2 \to A_2 c / A_2 ad / bd / f$

Removing direct LR of A_2 prod^{n}s $\to \begin{cases} A_2 \to bd A_2' / f A_2' \\ A_2' \to c A_2' / ad A_2' \end{cases}$

Now, change names to original.

- The grammar which is either left recursive or right recursive is always unambiguous.
- The grammar which is both left recursive and right recursive is always ambiguous.
- Left recursive grammar is not suitable for Top down parsers. This is because it makes the parser enterinto an infinite loop. To avoid this situation, it is convertedinto its equivalent right recursive grammar. This is done by eliminating left recursion from the left recursivegrammar.
- The conversion of left recursive grammar into right recursive grammar and vice-versa is decidable.
- **Ambiguity:**For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.For unambiguous grammar, leftmost derivation and rightmost derivation represents thesame parse tree.
- **Checking ambiguity:**More than one parse tree or derivation tree or syntax tree or leftmost derivationor rightmost derivation for at least one string. It's undecidable.
- **Removing ambiguity:**By implementing Precedence and Associativity. > The higher the levelof the production(More towards start symbol of production),the lower the priority(precedence) of the operator. Higher precedence operator evaluated at the end of the grammar. > If the operator is left associative, induce leftrecursion in its production. If the operator is right associative,induce right recursion in its production.

https://www.gatevidyalay.com/grammar-ambiguity-ambiguous-grammar/
https://www.gatevidyalay.com/removing-ambiguity-grammar-ambiguity/

- For ambiguous grammars,
  - More than one leftmost derivation and more than onerightmost derivation exist for at least one string.
  - Leftmost derivation and rightmost derivation representsdifferent parse trees.
- For unambiguous grammars,
  - A unique leftmost derivation and a unique rightmostderivation exist for all the strings.
  - Leftmost derivation and rightmost derivation representsthe same parse tree.
- There may exist derivations for a string which areneither leftmost nor rightmost.
- Leftmost derivation and rightmost derivation of astring may be exactly the same. In fact, there mayexist a grammar in which leftmost derivation and rightmostderivation is exactly the same for all the strings.
- For a given parse tree, we may have its leftmost derivationexactly the same as the rightmost derivation.
- If for all the strings of a grammar, leftmost derivationis exactly the same as rightmost derivation, thenthat grammar may be ambiguous or unambiguous.
- **Left factoring(eliminating non-determinism):**Leftfactoring is a process by which the grammar with common prefixes is transformed to make it useful for Topdown parsers. >>We make one production for each common

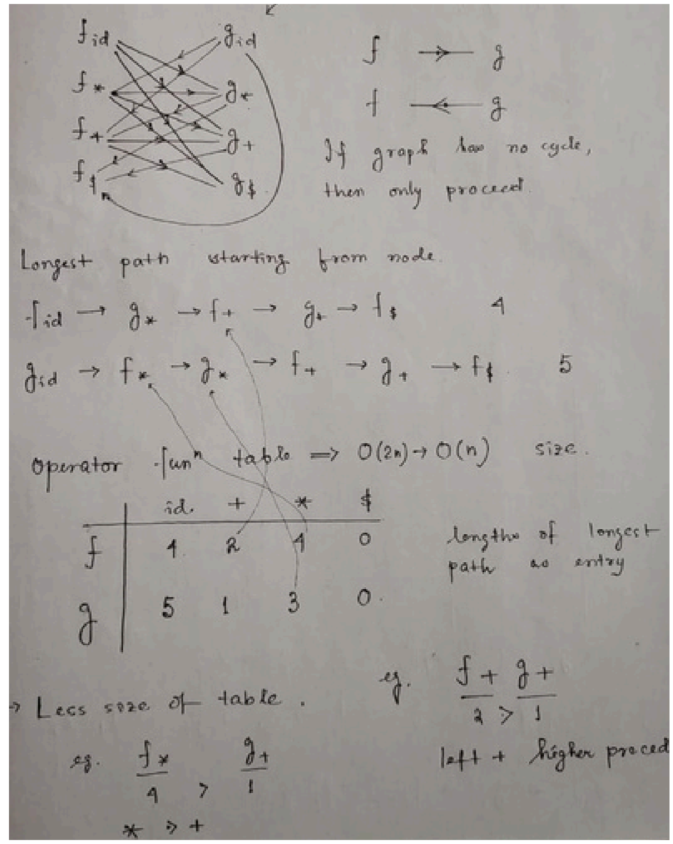
prefixes. The common prefix may be a terminal or a non-terminal or a combination of both. Rest of the derivation is added by new productions. The presence or absence of left recursion does notimpact left factoring and

- ambiguity anyhow. The presence or absence of left factoring does notimpact left recursion and ambiguity anyhow.
- The presence or absence of ambiguity does not impactleft recursion and left factoring anyhow. **Parsers**:
-
-
    - Top down parsers
        - w/ full backtracking (brute force)
        - w/o backtracking
            - Recursive descent parser(Goes to infinite loop ifgrammar is not free of left recursion)
            - Non-recursive descent parser or LL(1) parser - PredictiveParser(No ambiguity, No Left Recursion, No nondeterminism)
    - Bottom up parsers
        - Operator precedence parser
        - LR Parsers (LR grammar can never be ambiguous)
            - LR(0)
            - SLR(1)
            - LALR(1)
            - CLR(1)
- **First and Follow calculation**(eliminate left recursionbefore calculating,*follow*never contains eps)
- **LL(1) Parsing**
    - Using LMD, left → right scanning, #lookahead = 1
    - Constructing LL(1) parsing table [Table size = #V* (#T + 1), 1 for $]
        - All null productions put under*follow*set of thesymbol, rest under the*first*set.
    - In case of >1 entries in a cell of table, we can'tparse using LL(1) parsing.
    - LL(1) parsing examples (using stack, put the productionin reverse into the stack, match top of the stack with input symbol, if matched pop, then look for theappropriate production by seeing the top and the input string symbol)
    - Checking whether grammar is LL(1) or not(>2 entries=> not LL(1)).
    - If a grammar contains Left recursion/Common prefixes/Ambiguity,it can't be LL(1).

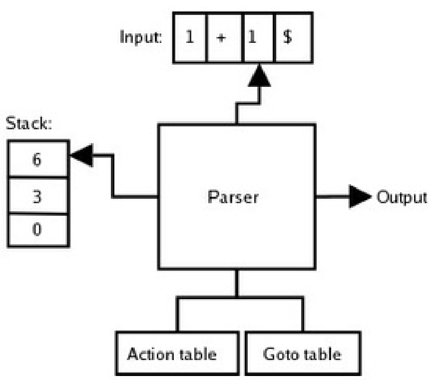
    - LL grammar is a subset of CFG with some restrictionson it.

- **Recursivedescentparsing**
    - Uses stack
    - If contains left recursion will go to infinite loop
    - If contains common prefixes may generate parsing error
    - Recursive descent is a top-down parsing techniquethat constructs the parse tree from the top and the input is read from left to right. It uses proceduresfor every terminal and non-terminal entity. Thisparsing technique recursively parses the input to make a parsetree, which may or may not require back-tracking. But the grammar associated with it (if not left factored)cannot avoid back-tracking. A form of recursive-descent parsing that does not require anyback-tracking is known as predictive parsing.
    - This parsing technique is regarded recursive as ituses context-free grammar which is recursive in nature.
    - For Recursive Descent Parser, we are going to writeone recursive function/procedure for every variable. https://www.geeksforgeeks.org/recursive-descent-parser/
- **Operatorprecedenceparsing**
    - Operator grammar should not have 2 variables adjacentto each other and null productions(❌).
    - Can also work on ambiguous grammar. Only parser thatworks on ambiguous grammar.

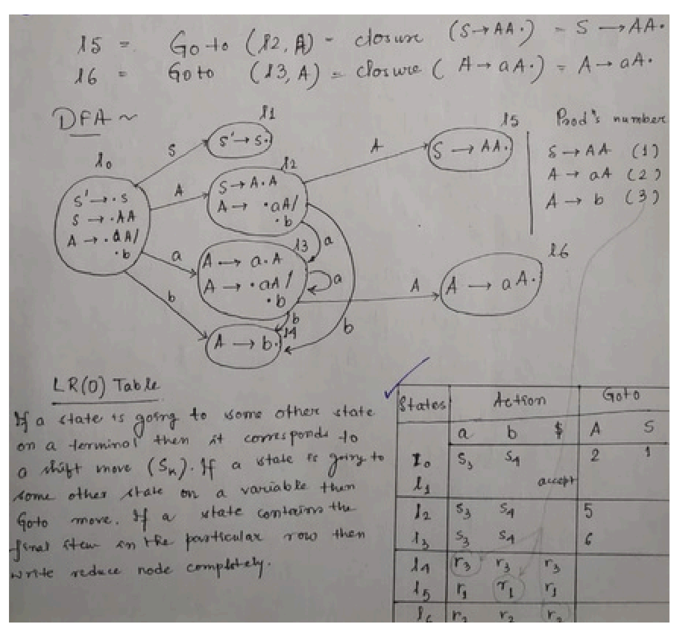
- Uses operator relation table(having associativity and precedence description)
- Size of operator relation table = n2[n = #operators]
  So, we have operator function table. Size = 2n [O(n)]
    - Building Operator function table(from operator relation graph): >Proceed only if graph is acyclic. >Length of longest path from the node.
    - Disadvantage of the function table is there’s no blank entries here(as in operator relation table). So, error detecting capability of the function table is less.



- **LR Parsers**
  - Left to right scanning
  - Using right most derivation
  - Canonical collection of LR(0) items: LR(0), SLR(1) table
  - Canonical collection of LR(1) items: LALR(1), CLR(1)
  - Ambiguous grammar can never be accepted by LR parsers.
  - In LR(0), we place the*reduce*node in the entire row of the table(action part).
  - Grammar + (S’ → S) = Augmented grammar



- **LR(0) Parser**
  - Uses Canonical collection of LR(0) items: Add augmented production S’ → S and put “.” at first positionof every production’s RHS. (“.” signifies what it has seenso far in the string)
  - CFG → Check ambiguity → Remove ambiguity if possible → Add augment production and make canonicalcollection of LR(0) items → Draw DFA → Construct LR(0) parsingtable
  - Making the DFA: Put all productions in the initialstate. Then, compute*Goto*and the closure of that production,thereby, giving birth of new states.
  - Making the LR(0) table:
    - If a state is going to some other state on seeing a terminal, it’s a action*shift move*(specified with the state number where it’s going in DFA).
    - If a state is going to some other state on seeing some variable, then it's a*goto movo*(specified with the state number where it’s going in DFA).
    - If a state contains some final item(like A → Aa.) then in the*action*part of the table put *reduce move*(reduce using specified production)throughout the row.

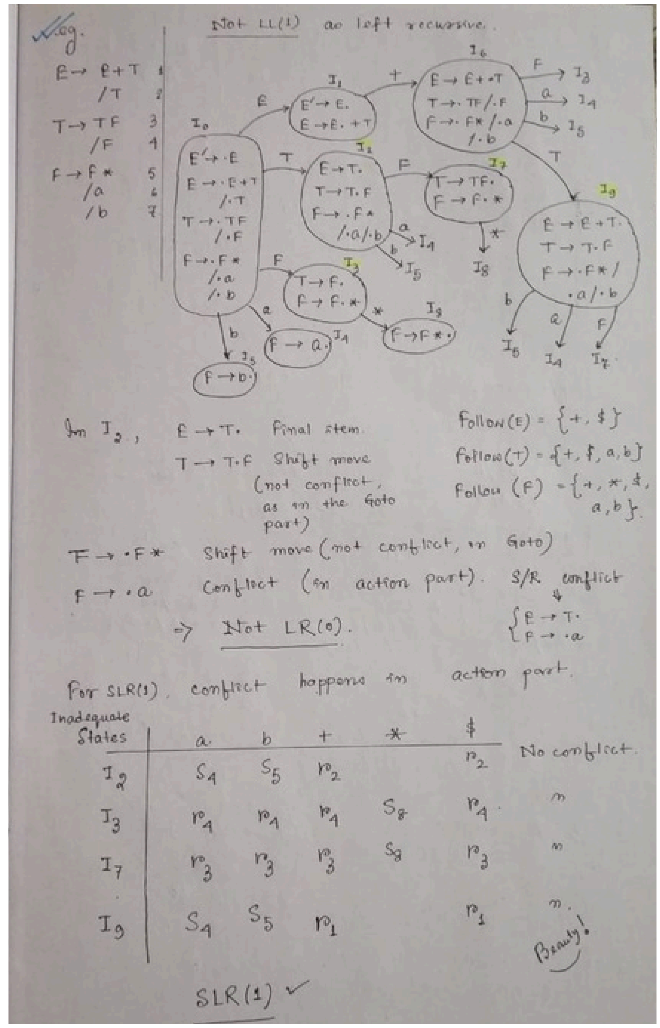


- **SLR(1) Parser**
  - **Also uses canonical collection of LR(0) items.**

- Only difference with LR(0) is here we put *reduce move* only in the *follow* of LHS.
- Makings DFA(same as LR(0)), SLR(1) table: Same as LR(0) except the rule that when a state contains a final item(as A → ab.), fill with *reduce* entry only in the *follow* of A.
- **Conflicts:**
  - R/R conflict: State containing 2 *reduce* items.
  - S/R conflict: State containing both *shift* and *reduce* moves.
  - Conflicts can happen for LR(0), SLR(1).
  - If conflict is there, we say it's not LR(0), or SLR(1), in which case it applies.
  - If grammar is LR(0), it's SLR(1).
  - No relation of a grammar of being LL(1) and [LR(0) or SLR(1)].
  - Number of **inadequate states** is nothing but the number of states which have S/R or R/R conflicts.

A → a.
B → b.
For SLR(1), it's an R/R conflict only if *follow(A)*, *follow(B)* sets intersect.

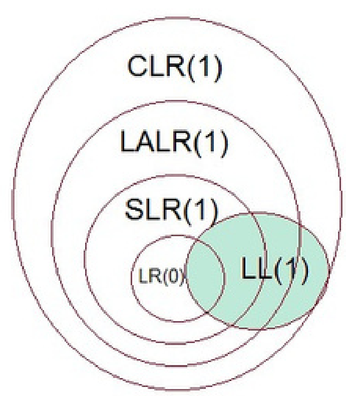
- **CLR(1) Parser (Canonical LR) or LR(1)**
  - **LR(1) item:** Collection of LR(0) item along with a lookahead symbol. If A → ∝.Bβ, a is in closure of
  - some state and B →γ is a production rule of the grammar, then B → γ, b will be in the closure of that state also for each terminal b in *first*(βa) Lookahead is used to determine where we place the final item. For
  - the augment production, lookahead is always the $. Placement of *shift* nodes is the same as LR(0) or SLR(1), only difference is for the placement of
  - *reduce* nodes. While making the CLR(1) table, we place the *reduce* only in the lookahead symbols. Example: Next page



-

- **LALR(1) Parser (Lookahead LR)**
  - The LR(1) items(states in the DFA) having the same productions but different lookahead are combined to form a single set of items. If there are no states with
  - same productions and different lookahead in CLR(1), then LALR(1) parsing table is identical to CLR(1). If a grammar is CLR(1) it may or may not be LALR(1).
  - After combining items it may give rise to conflicts. We just reduce the size of the parsing table in LALR(1) compared to CLR(1), but the conflicts can't go away. If a grammar is not CLR(1) it cannot be LALR(1). Example: next page
  -
  -

- **Some pointers:**
  - LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ CLR(1)
  - LL(1) ⊂ LR(1)
  - Every ϵ free LL(1) grammar is SLR(1) grammar, thereby LALR(1) and CLR(1) too.
  - (#states(SLR(1)) = #states(LALR(1))) <= #states(CLR(1))
  - CLR(1) parser most powerful among the bottom-up parsers.
  - Based on power, OPP < LL(1) < LR(0) < SLR(1) <= LALR(1)<= CLR(1)
  - If a LL(1) grammar's each variable can derive a nonnull string then it is LALR(1) i.e. an LL(1) grammar with symbols that have both empty and non-empty derivations is also an LALR(1) grammar.



- If a LL(1) grammar has all the variables producing only null strings, then it may or may not be LALR(1).
- Every LL(k) grammar is also an LR(k) grammar.
- https://gateoverflow.in/blog/6215/parsing-notes

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

$I0 = \text{closure}(S' \to \cdot S)$
$= S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b.$

I1

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

I2

$I2 = \text{Goto}(I0, A) = \text{closure}(S \to A\cdot A, \$)$
$= S \to A\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

I3

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

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

I4

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

I5

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

I6

$I6 = \text{Goto}(I2, a) = \text{closure}(A \to a\cdot A, \$)$
$= A \to a\cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

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

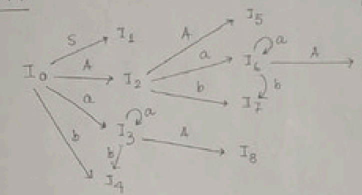
$I7 = \text{Goto}(I2, b) = \text{closure}(A \to b\cdot, \$)$
$= A \to b\cdot, \$$

$I8 = \text{Goto}(I3, A) = \text{closure}(A \to aA\cdot, a/b)$
$= A \to aA\cdot, a/b$

$I9 = \text{Goto}(I6, A) = \text{closure}(A \to aA\cdot, \$)$
$= A \to aA\cdot, \$$

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		

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

- Placement of shift nodes in CLR(1) parsing table is same as the SLR(1) parsing table. Only difference is in the place 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 $S \to AA$ $A \to aA/b$.

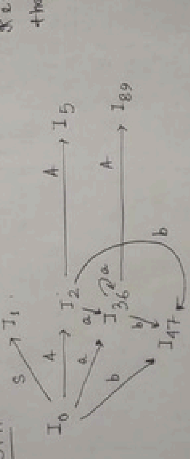
I_3 and I_6 are same in their LR(0) items but differ in their lookahead. So, we combine.

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

Same for I_4, I_7 | Same for I_8, I_9

$I_{47} = \{A \to b\cdot, a/b/\$\}$ | $I_{89} = \{A \to aA\cdot, a/b/\$\}$

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		

- Any string derivable fromthe start symbol is a**sententialform**— it becomes a sentenceif it contains only terminals. A sentential form that occurs in the leftmost derivationof some sentence is called**left-sentential form**. A sentential
- form that occurs in the rightmost derivationof some sentence is called**right-sentential form**. A**handle**of a right
- sentential form ‘γ’ (γ=αδβ) isa production E→δ and a position in γ where δ canbe found and substituted by E to get
- the previous step in theright most derivation of γ — previous and not “next”because we are doing rightmost derivation in REVERSE. **Viable prefixes**are the prefixes of right sententialforms that do not extend beyond the end of its handle,i.e., a viable prefix either has no handle or just one possiblehandle on the extreme RIGHT which
- can be reduced. **Viable prefixes**can be recognized by using a FINITEAUTOMATA. Using this FINITE AUTOMATA and a stack we get the power of a Push Down Automata and thatis how we can parse context-free
- languages. In LR(1) we see the input from left till we get ahandle. After this, we see one more lookahead symboland determine the parser action. i.e., the parser hasMORE information to decide its action than in LL(1)and
- this makes it more powerful than LL(1). More powerful means,any grammar that can be parsed by LL(1) can alsobe parsed by LR(1). This holds in general case too –for any k, LL(k) set of grammars is a PROPER subsetof LR(k) set of grammars.

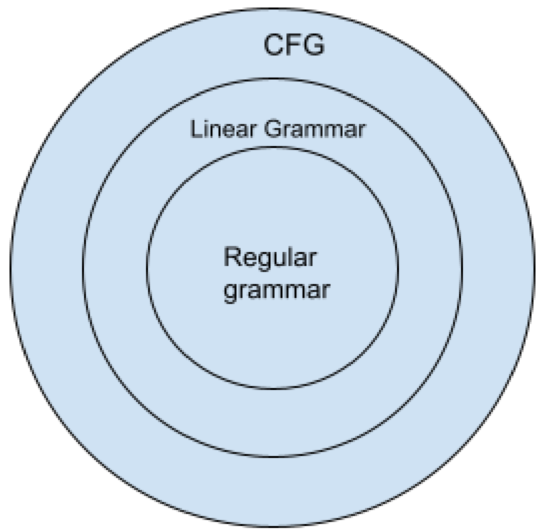
Just revising on the previous part we have the following relation. Here $Lang$ denote the set of languages defined by the given set of grammars.

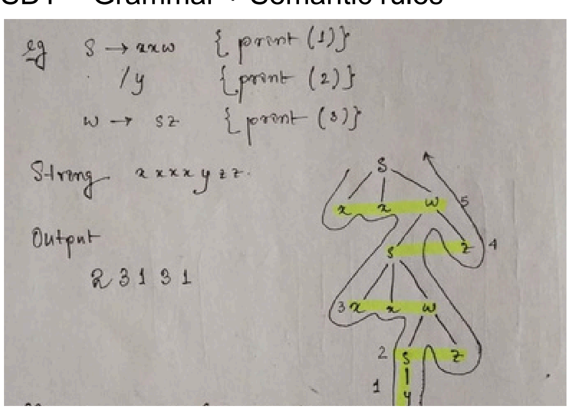
1. $LL(0) \subset LL(1) \subset LL(2) \dots LL(k) \subset LL(k+1) \dots$
2. $Lang(LL(0)) \subset Lang(LL(1)) \subset Lang(LL(2)) \dots Lang(LL(k)) \subset Lang(LL(k+1)) \dots$
3. $LR(0) \subset LR(1) \subset LR(2) \dots LR(k) \subset LR(k+1) \dots$
4. $Lang(LR(0)) \subset Lang(LR(1)) = Lang(LR(2)) \dots Lang(LR(k)) = Lang(LR(k+1)) \dots$ ($Lang(LR(0)$ is DCFL with prefix property and $Lang(LR(k)); k \geq 1$ is DCFL (with or without prefix property))
5. $Lang(LR(0)\$) = Lang(LR(1)) = Lang(LR(2)) \dots Lang(LR(k)) = Lang(LR(k+1)) \dots$ ($Lang(LR(0)\$$ is DCFL as when we add an end delimiter no string becomes a prefix of another)

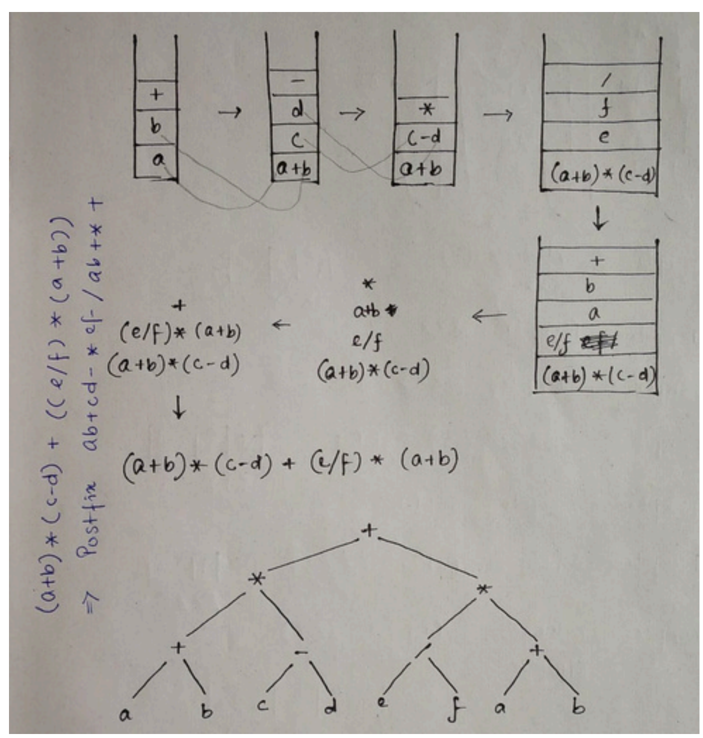
- Every LL(1) is LR(1)
- If a grammar is LL(k) then it is LL(k+1) too.
- Every LL(1) is~~LALR(1)~~
- Every LL(1) is~~LR(0)~~
- LL(k) is subset of LR(k) (LL(2) is subset of LR(2))
- LR(1) table ≅ size LL(1) table size
- #states (LR(0) = SLR(1) = LALR(1)) <= CLR(1)
- Reading terminal on LR parser DFA, if we go out ofthat state then*Shift*move is put in the table.
- LR Parsers conflict
    - S/R
    - R/R

  No S/S conflict as it is DFA, not NFA. If NFA wasthere then S?S conflicts could be there.
- **Inadequate state**: State having conflicts.
    - Inadequate state must have a minimum of 2 productions.
    - At least 1*reduce*move is there.
- In LR(0), SLR(1), LALR(1)
    - Number of states same
    - Shift entries same
    - Goto part same
    - *Reduce*entries may change
    - Error entries may change
- If CLR(1) doesn't have an R/R conflict then LALR(1)may have an R/R conflict.
- If CLR(1) doesn’t have S/R conflict then LALR(1) alsodoes not have S/R conflict.
- If LALR(1) have S/R conflict, then CLR(1) also haveS/R conflict.

- **YACC:**
  - LALR(1) parser generator: Create LALR(1) parsing table.
  - If having conflict YACC resolves them too.
    - S/R => S   Resolving S/R conflict
    - r3/r4 => r3   Resolving R/R conflict
- Every Regular grammar need not be LL(1).
- Every Regular language is LR(1).
- Every regular grammar is linear, but every lineargrammar need not be regular.
- Every LR grammar is unambiguous, but every unambiguous grammar is not LR grammar.
- Follow(A) = RFollow(A) [RFollow: Follow in right sententialform]
- Follow(A) and LFollow(A) need not be the same. [LFollow: Follow in left sentential form]
- For Unambiguous grammar
  - LMDT and RMDT should be same for some string production
  - LMD and RMD need not be same though
- **Viable prefixes:**The set of prefixes of right sententialforms that appear on the stack of Shift/Reduce parser(or LR parser)are called Viable prefixes.
- TDP (Uses LMD: Problem in multiple productions)
  - w/ backtracking: RDP
  - w/o backtracking: LL(1) / Predictive parser / NRDP
- BUP (No backtracking for any parser, Uses reverseof RMD: Problem in finding handles)
  - LR
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1)
  - OPP



- **Syntax directed translation**
  - Tointerleavesemanticand syntax analysis
  - SDT = Grammar + Semantic rules
  - 
  - Syntax tree: Compact parse tree
    - Constructionexample: ------------------->
  - **Attributes**
    - **Synthesized:** Computed from the valuesoftheattributes of the children nodes



- **Inherited:** Frombothsibling and parent nodes
- **Types of SDT:**
  - **S-attributed definition:** Uses only synthesized attributes
    - Evaluated inbottom-up parsing
    - Semantic actions placed in rightmost place of RHS
  - **L-attributed definition:**Uses both synthesized andinherited attributes with restriction(inherit from parent or left sibling only)
    - Evaluated by depth first order, left to right evaluation
    - Semantic actions placed anywhere in RHS.
  - If a definition is S-attributed then it is L-attributedalso. Not vice versa.

- **Static(fixed) storage allocation**
  - Tomake memoryinthestatic area, it should be madein compile time
  - 1-time memory created
  - Recursion not allowed
  - Dynamic data structure not allowed
- **Stack storage allocation(**will grow and shrink in runtime**)**
  - Recursion allowed
  - Dynamic DSnotallowed
- **Heap memory allocation(**will grow and shrink in runtime**)**
  - Recursion allowed
  - Dynamic DSallowed
- **Activation record contains:**
  - Localvariables: hold the data that is local to theexecution of the procedure.
  - Temporary values: stores the values that arise inthe evaluation of an expression.
  - Machine status: holds the information about the statusof the machine just before the function call.
  - Access link (optional): refers to non-local data heldin other activation records.
  - Control link (optional): points to activation recordof caller.
  - Return value: used by the called procedure to returna value to calling procedure
  - Actual parameters
  - Activation record does not contain global variables.

Lexical phase error can be:

- Spelling error.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.
- To remove the character that should be present.
- To replace a character with an incorrect character.
- Transposition of two characters.

https://drive.google.com/file/d/13hYZJVHThUcqsNCCq3SNI2RcURDO0m_P/view