ToC- Basics

1. **Symbol**: [Lowest unit] represented by Lower case or Spl. symbols.
   * Also known as Terminals.
2. **Strings**: Seq. of symobols. Denoted by w.
   * Strings should be finite, and Seq matters.
3. **Length of string**: $|w|$ No of symbols involved in the string.
   $w_1 = abcd \therefore |w_1| = 4$ *(Never) $|w| \neq \infty$
4. **Empty String or Null String**: (ϵ) String w/ Length 0 ie $|w| = 0$.
   [* $\epsilon \rightarrow$ string, not a symbol]
5. **Concat. of strings** ($\cdot$) Concat. of 2 string will always be a string.
   $w_1 \cdot w_2 \neq w_2 \cdot w_1$ non-commutative.
   $\epsilon \cdot any = any \cdot \epsilon = any$
6. **Prefix of a String**: ie Seq of Leading Symbols.
   $w_1 = abcd \therefore Prefix(w_1) = \epsilon, a, ab, abc, abcd.$
   $w_2 = aaaa \therefore Prefix(w_2) = \epsilon, a, aa, aaa, aaaa.$
   → [A string is a prefix of itself.]
   → [ϵ is a prefix of every string.] (Trivial).
7. **Suffix of a String**: ie Seq of trailing Symbols.
   $w_1 = abcd \therefore$ Suffix = $\epsilon, d, cd, bcd, abcd.$
   → A string is a suffix of itself.
8. **Sub String**: It is a string present in the string.
   $w = abcd$
   Substrings: (ϵ) a ab abc (abcd) w
   b bc bcd
   c cd
   d
   Trivial, Trivial
   If $|w| = n$; then
   max no of Sub str $= \frac{n(n+1)}{2} + 1 = \Sigma n + 1$ → Valid, when all symbols are unique [ie distinct].
9. **Reverse of a String** (w^R): $w = abcd$; $w^R = dcba$.
   [$w \cdot w^R$, $w^R w$] → palindrome. Say, $w = \epsilon$, then $w^R = \epsilon$.
10. **Language**: Set of strings. [May be finite or infite]
    * Empty Language is possible. $L = \{\} = \emptyset \neq \{\emptyset\}$
11. **Alphabet**: (Σ) Non empty finite set of symbols on which, the Language is defined.
    [* w/o Alphabet, Language is not possible.
    * infinite Alphabet not possible]
    ⊛ Σ doesnt contain ϵ.
    ⊛ for $L = \{\}$ and $L = \{\epsilon\}$, any alphabet is valid
12. **Power of String** $[w^n \mid n \in \mathbb{Z}^+]$
    $w = abc \therefore$ $w^0 = \epsilon$; $w^1 = w = abc$
    $w^2 = w \cdot w = abc.abc$
    $w^3 = www = abcabcabc.$
13. **Concat of Language**: Just like cartesian prod. of 2 sets
    $L_1 = \{a, b\}$ $L_2 = \{1, 2\}$
    $\therefore L_1 \cdot L_2 = \{a1, a2, b1, b2\}$
    ⊛ Concat of 2 Lang is finite only if, both of them are finite.

If any of the Lang in the Concat is ∞, then Concat is ∞.

$\emptyset \cdot any = any \cdot \emptyset = \emptyset$

14. Reverse of a language $[L^R]$

L^R contains w^R for $\forall w \in L$. $L = \{ab, ba, aa\ bb, a, b\}$

$\therefore L^R = \{ba, ab, aa, bb, a, b\}$

15. Power of Language $[L^n \mid n \in \mathbb{N}]$

$L = \{a, b\}$ $L^0 = \{\in\}$, $L^1 = L = \{a, b\}$

$L^2 = L \cdot L = \{a,b\}\{a,b\} = \{aa, ab, ba, bb\}$

* Union and intersection of 2 Lang. will always be a Language

16. Complement of a Lang. $[L^c]$: L^c = Universal - L

$U^c = \emptyset$

$\emptyset^c = U$

17. Set diff : $L_1 - L_2 = L_1 \cap (L_2)^c$

18. Kleene Closure (*) :

$$a^* = \bigcup_{i=0}^{\infty} a^i \Rightarrow a^* = a^0 \cup a^1 \cup a^2 \cup a^3 \cdots \infty$$

$= \{\in, a, aa, aaa \ldots\}$

this gives a Lang as o/p.

$(a+b)^*$ = All combinations of a and b = Universal Lang for $\Sigma(a,b)$.

$\emptyset^* = \{\in\}$ Very. IMP.

* $[(a^*b^*)^* = (a+b)^*]$

* $(a^*+b^*) = \{a, aa, \ldots b, bb, \ldots, \in\}$

* $[(a^*)^* = a^*]$

19.) Positive Closure: (+)

$$a^+ = \bigcup_{i=1}^{\infty} a^i$$

$a^* = a^+ + \in$

$a^+ = a^* - \in$

$aa^* = a^+$

$\emptyset^+ = \emptyset$ and $(a^+)^+ = a^+$

$(a^*b^*)^+ = (a+b)^*$

TOC - Regular Language:

⊛ Language is regular, if accepted by FSM. (DFA | NFA | ∈ NFA).

Finite State Machine: $FA = \{Q_0, \Sigma, Q, F, \delta\}$ → Defn of FSM.

$\delta: Q \times \Sigma \to Q$ — DFA

$\delta: Q \times \Sigma \to 2^Q$ — NFA

$\delta: Q \times \{\Sigma \cup \in\} \to 2^Q$ — ∈ NFA.

⊛ A FSM may have 0 or more final state, but exactly one initial state.

Important Points:

→ if Language contains ∈, then init state will be final state.
→ if init state is final, then lang. contains ∈.
→ A Lang. may have more than one FSM for it, but min FSM is always unique.
→ FINITE Language is always regular.
⊛ FSM for finite Lang. will always have TRAP state.
→ finite Lang. FSM will never have loops.
→ if we make all states final in DFA, then it accepts Σ^*.
→ by finite automata, we cannot do, comparison or counting as it has no memory. [it only has mem in forms of state]
→ Linear Power of symbol is always Regular.
→ Mod-machines are always regular.
→ Non-Linear Power of symbol ⇒ NRL.
→ Trap not Reqd in NFA | ∈-NFA.
→ Power: DFA = NFA = ∈-NFA
→ Trap state can never be final state. And in DFA, atmost 1 TRAP.
→ if we make all state as FINAL in NFA, then it will accept Prefix(L) [ie Lang containing all the prefixes of all its strings.]

Complement of a FSM: [M^c]

Change: Final → non Final, non Final → Final ⇒ this is only valid for DFAs, and invalid for NFA.

⊛ DO NOT Reverse the arrows.

⊛ The complement of a regular lang is always regular.
⊛ The concat. of 2 Lang is always regular.

Reverse of a Machine: [M^R]

→ make initial state as final state, and all final state as non final. and init, Reverse the arrows.

⊛ Reverse of DFA could be NFA. ⊛ Rev of RL is always RL.

* if more than 1 initial state (after conversion), then make new init state w/ ∈ transition to them.

NOTE

$n \leq$ #states in DFA $\leq 2^n$

• $n \rightarrow$ no of states in NFA of Equivalent Lang

Standard RL and NRL

1. $\{a^n b\}$ RL
2. $\{a^n b^m \mid n, m \geq 1\}$ RL
3. $\{a^n b^m \mid n \leq 7, m \leq 1\}$ RL
4. $\{a^n b^m \mid n = 2 \text{ or } 4\}$ RL
5. $\{a^n b^m \mid n \geq 1, m \geq 2\}$ RL
6. $\{a^n b^m \mid n \times m \geq 2\}$ RL
7. $\{a^n b^m \mid n \times m \leq 2\}$ RL
8. $\{a^n b^m \mid n+m = 6\}$ RL
9. $\{a^n b^m \mid n+m \geq 6\}$ RL
10. $\{a^n b^m \mid n+m \leq 6\}$ RL
11. $\{a^n b^m \mid n - m = 5\}$ NRL
12. $\{a^n b^m \mid n = m+5\}$ NRL
13. $\{a^n b^m \mid 100 - n = m\}$ RL
14. $\{a^n b^m \mid n = 2m\}$ NRL
15. $\{a^n b^m \mid n > m\}$ NRL
16. $\{a^n b^m \mid n \geq 2m\}$ NRL
17. $\{a^n b^m \mid n/m = 5\}$ NRL
18. $\{a^n b^m \mid n^2 = m\}$ NRL
19. $\{a^p \mid p \rightarrow \text{Prime}\}$ NRL
20. $\{a^n b^n c^m \mid n \leq 7, m \geq 2\}$ RL
21. $\{a^n b^n c^m \mid n \geq 1, m \leq 2\}$ NRL
22. $\{a^n b^n e^m \mid n \geq 1, m \geq 1\}$ NRL
23. $\{a^n b^n c^m \mid n = m\}$ NRL
24. $\{a^n b^m c^p \mid n = m+p\}$ NRL
25. $\{a^n b^m c^p \mid n = m \times p\}$ NRL
26. $\{a^n b^m c^p \mid n = 2m+p\}$ NRL
27. $\{a^n b^n c^n \mid n \geq 1\}$ NRL
28. $\{a^p \mid p \text{ is prime}, |p| \leq 30\}$ RL
29. $\{a^n b^n c^n \mid n \leq 10^7\}$ RL
30. $\{a^n a^n \mid n \geq 1\}$ RL
31. $\{w.w \mid w \in (a,b)^*\}$ NRL
32. $\{w w^R \mid w \in (a,b)^*\}$ NRL
33. $\{w (w^R)^* \mid w \in (a,b)^*\}$ RL
34. $\{w \# w^R \mid w \in (a,b)^*\}$ NRL
35. $\{w \mid w \in (a,b)^*\}$ RL
36. $\{w x w^R \mid w, x \in (a,b)^*\}$ RL
37. $\{w x w^R \mid w, x \in (a,b)^+\}$ RL
38. $\{x w w^R \mid w, x \in (a,b)^*\}$ RL
39. $\{x w w^R \mid w, x \in (a,b)^+\}$ NRL
40. $\{w w^R x \mid w, x \in (a,b)^*\}$ RL
41. $\{w w^R x \mid w, x \in (a,b)^+\}$ NRL
42. $\{x \in y \mid x, y \in (a,b)^*\}$ RL
43. $\{w x w \mid w, x \in (a,b)^+\}$ NRL
44. $\{w x w \mid w, x \in (a,b)^*\}$ RL
45. $\{w x w \mid w \in (a,b)^* \; x \in (a,b)\}$ NRL
46. $\{w x w \mid w, x \in (a,b)^+, |w| \leq 3\}$ RL
47. $\{a^n b^n c^m d^m \mid n, m \geq 1\}$ NRL
48. $\{a^n b^m c^p d^q \mid n, m, p, q \geq 1\}$ RL
49. $\{a^n b^m c^p d^q \mid n, m \geq 3, pq \leq 10\}$ RL
50. $\{a^n c b^n \mid n \geq 1\}$ NRL
51. $\{a^{2n} c b^{3m} \mid n, m \geq 1\}$ RL
52.
53. $\{awa \mid w \in (a,b)^*\}$ RL
54. $\{waw \mid w \in (a,b)^*\}$ NRL
55. $\{wcw w^R \mid w \in (a,b)^*\}$ NRL
56. $\{ab b^n \mid n \geq 1\}$ RL
57. $\{w \mid w \in (a,b)^*, |w| = \text{even}, n_a(w) = \text{odd}\}$ RL
58. $\{a^n b^n \mid n \geq 1, n \neq 99\}$ NRL
59. $\{a^n b^m \mid n < m < 2^n\}$ NRL
60. $\{a^n b^n \mid n > m > 10\}$ NRL
61. $\{a^n b^m \mid n < m < 10\}$ RL
62. $\{a^n b^{2m} \mid 3 < m < 4, n \geq 1\}$ RL
63. $\{a^m b^n c^p \mid m+n+p = 10\}$ RL
64. $\{w_1 . w_2 \mid w_1, w_2 \in (a,b)^*, |w_1| = |w_2|\}$ RL
65. $\{w \mid w \in (a,b)^* \; | n_a(w) - n_b(w) | \leq 2$ for every prefix of $w\}$ RL

Regular Expression : Mathematical formula used to rep. RL only.

* if RE for a language exists, then it's RL.
* RE for a given Lang, will generate only the strings in the given language.

RE → Unrestricted $(\Sigma, (\ ), \cdot, *, +, ^+, \cap, \cup, \sim)$
RE → Semi-Restricted $(\Sigma, (\ ), \cdot, *, +, ^+, \cap, \cup)$
RE → Restricted $(\Sigma, (\ ), \cdot, *, +, ^+)$ in syllabus

] out of syll.

Priority : $(\ )$, a^*, a^+, $\cdot$, $+$ ↓ order of evaluation.

Ardens Theorem (only for ϵ free FSM)

$R = Q + RP$
$\Rightarrow R = QP^*$

R, Q, P are R.E, where $Q \neq \epsilon$.
* it has unique soln $\Rightarrow$ $R = QP^*$.

Algebraic Laws of RE :

Commutative Law:
(a) $r_1 + r_2 = r_2 + r_1$
(b) $r_1 \cdot r_2 \neq r_2 \cdot r_1$

Associative Law:
(a) $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$
(b) $(r_1 \cdot r_2) \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$.

Distributive Law :
(a) $r_1 + (r_2 \cdot r_3) \neq (r_1 + r_2) \cdot (r_1 + r_3)$ [not distributive like this]
(b) $r_1 \cdot (r_2 + r_3) = r_1 \cdot r_2 + r_1 \cdot r_3$ [this is valid].

Identity Law :
(a) $r_1 + \phi = r_1$
(b) $r_1 \cdot \epsilon = r_1$

Idempotent Law :
(a) $r_1 + r_1 = r_1$ [valid]
(b) $r_1 \cdot r_1 \neq r_1$ [not valid]

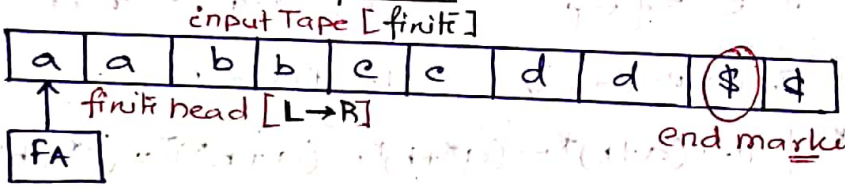
Standard Results for Reg Expr. :

1. $(r^*)^* = r^*$
2. $\phi^* = \{\epsilon\}$
3. $\phi^+ = \phi$
4. $\epsilon^* = \{\epsilon\}$
5. $\epsilon^+ = \{\epsilon\}$
6. $r^+ = r \cdot r^* = r^* \cdot r$
7. $r^* = r^+ + \epsilon$
8. $\epsilon + r = \epsilon + r \neq r$
9. $r \cdot r \neq r$
10. $\epsilon + r r^* = r^*$
11. $(a+b)^* = (a^* + b)^* = (a + b^*)^* = (a^* + b^*)^*$
$= (ab^* + ba^*)^* = ((a+b)^*)^*$
$= (a^*b^* + a^*b^*)^* = (b^*a^*)^*$
$= (a^*b^*)^*$
12. $a^* \cdot \phi = \phi$
$r \cdot \phi = \phi$.

Important Notes:

1. Union of 2 RL is always RL and union ⇒ commutative and Ass.
2. Finite union of RL is always RL.
   * infinite union of RL may or maynot be RL.
3. Finite intersection of 2 RL is always RL.
   * infinite intersection is not RL. (NRL).
4. Concat. of 2 or more RL is always RL.
5. Subset of RL may or maynot be RL. $[a^n b^n \subseteq a^n b^m]$
6. Subset of NRL may be RL.
7. RL ∪ DCFL = DCFL RL ∩ DCFL = DCFL
   RL ∪ CFL = CFL RL ∩ CFL = CFL
   RL ∪ CSL = CSL RL ∩ CSL = CSL
   RL ∪ REC = REC RL ∩ REC = REC
   RL ∪ RE = RE. RL ∩ RE = RE.
8. Kleene closure (*) and +ve closure (+) of RL is always RL.
9. Reverse of RL is always RL.
10. Every NRL has atleast 1 RL superset (Universal Lang.)
11. RL ∪ (L) = RL
    (L) need not be regular.
12. NRL ∪ NRL = (L) may or may not be RL.
13. RL ∪ (L) = NRL
    (L) always NRL.
14. RL ∩ NRL = (L) may or may not be RL.
15. NRL ∩ NRL = (L) may or may not be RL.
16. NRL ∩ (L) = RL may or may not be RL.

Mathematical Model of FSM:



NOTE

DFA + ∞ Stack = DPDA
NFA + ∞ Stack = NPDA

Power (NPDA) > Power (DPDA)

* DFA + Finite Stack = DFA
  NFA + Finite Stack = NFA

Transducers (FSM w/ op)

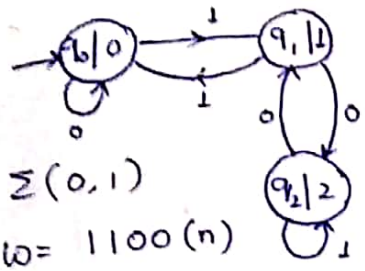
$\{q_0, \Sigma, Q, \lambda, \delta, \Delta\}$

[$\lambda \rightarrow$ o/p functions
$\Delta \rightarrow$ o/p symbols]

Moore

o/p associated w/ state

$\lambda : Q \rightarrow \Delta$



$\Sigma(0,1)$

$w = 1100\,(n)$

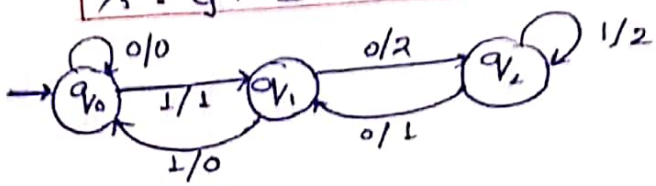
$o/p = 01000\,(n+1)$.

$\Delta(0,1,2)$.

Mealy

o/p associated w/ transition.

$\lambda : Q \times \Sigma \rightarrow \Delta$



$\Sigma(0,1)$

$w = 1100\,(n)$

$o/p = 1000\,(n)$.

(*) These 2 machines are inter-convertible.

NOTE

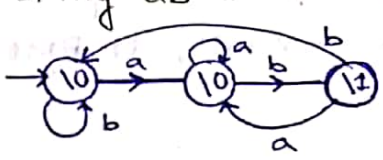
1. [Moore $\longrightarrow$ Mealy] $\rightarrow$ no change in no. of states.
2. [Mealy $\longrightarrow$ Moore] $\rightarrow$ no. of states may increase.

(*) In mealy m/c if $|Q| = m$, and $|\Delta| = n$, the

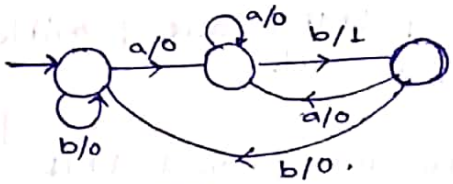
No of states in Moore $= (m \times n) \Rightarrow$ max possible.

Q> Construct a Mealy and Moore m/c to count occurance of substring 'ab' in the input w.

Soln

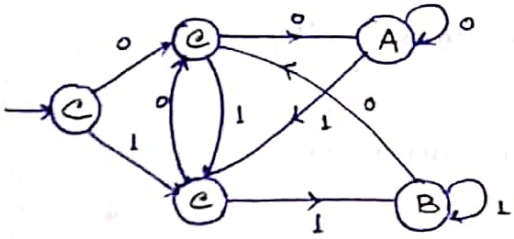


MOORE



MEALY

Q> Moore and mealy for: give o/p A when string ends with 00, B when ends with 11, C otherwise.



ToC - Grammar.

→ Set of rules by which, we represent a Language.
→ if there exists a Language, then there exist a grammar for it.
→ Language w/o grammar is not possible.
* 1 Lang. may have more than 1 valid grammar.
* Grammar w/o Language is possible [futile grammar]
* A Grammar can generate atmost 1 Language.

$G = \langle V, T, S, P \rangle$ 4 Tuple.

- V: Variables
- T: Terminals
- S: Start Variable (Unique)
- P: Production Rule.

Derivation: To derive a string from the grammar.
- LMD: Left most symbol resolved first
- RMD: Right most symbol resolved first.

* Graphical Rep: Parse Tree, abstract syntax Tree, derivation Tree.

1. If there exists LMD for a string, then there will exist RMD for it and vice versa.
2. Parse Trees of LMD and RMD may not be same.
3. We may have more than 1 LMD/RMD for a string.
4. If N no of LMD's are possible, then exactly N RMDs are also possible.

Ambiguous grammar: [USELESS]
If there exists, more than 1 LMD or RMD for any given string in the grammer, then it is ambiguous.
↳ Parse Tree of LMD or RMD may be diff.
↳ Grammar is ambiguous if we find atleast 1 string for which more than LMD/RMD exists.
↳ No algorithm exists to check ambiguity.
[undecidable]
↳ There are some grammars from which we can remove ambiguity.
↳ There are some grammar from which we can never remove ambiguity] inherently ambiguous.

NOTE

Grammar (G) → Is Lang Ambiguous L(G)
- Yes → inherently ambiguous.
- No → Can be converted.

RE, REC, CSL, CFL } may or may not be ambiguous.

DCFL, RL } unambiguous

Type 3 : Regular Grammar :

Left Linear

$V \to VT^*$
or
$V \to V$
or
$V \to T^*$

- ⊛ atmost one Variable on RHS of production.
- ⊛ Any Regular grammar can be converted to an unambiguous grammar.

Right Linear

$V \to T^*V$
$V \to V$
$V \to T^*$

Type 2 : Context free Grammar : (CFG.)

- * single variable defines anything.
- * Every regular Grammar is CFG.

$V \to (V+T)^*$
$[V \to \text{any}]$

NOTE
1. To check if 2 grammars are equivalent or not is undecidable.
2. To check if 2 Regular grammar are equivalent or not is decidable.

Type 1 : Context Sensitive :

$\alpha \to \beta$

1. $\alpha \in (V+T)^+$
2. $\beta \in (V+T)^+$
3. $|LHS| \leq |RHS|$

} Restrictions

Type 0 : Unrestricted Grammar :

$\alpha \to \beta$

i) $\alpha \in (V+T)^+$
ii) $\beta \in (V+T)^*$

No restrictions.

* Turing M/c for $L = \{\in\}$ is not possible
* Context Sensitive Grammar doesn't Produce $\in$.

CNF : Chomsky Normal Form :

The productions are of type $V \to VV$ or $V \to T$.

* In CNF, $\in$ is not allowed
* Algorithm for converting : (CFG → CNF).
  1. Remove $\in$-Production (unless inherent)
  2. Remove unit Production
  3. Simplify the grammar : (a) Remove useless Variable
     (b) Remove unreachable Var.

⊛ No of Steps in Derivation : $2N-1$

GNF : Greibach Normal Form :

Production of the form : $V \to TV^*$

$\in$-production not allowed in GNF.

- ⊛ exactly one terminal followed by any no. of variables.
- ⊛ $\in$ is allowed only in the start variable.
- ⊛ Any grammar can be converted to a GNF (CFG)

Steps to convert : (CFG → GNF)

1. Remove Left Recursion
2. Remove ∈-Production
3. Remove unit Production
4. Simplify the grammar.

* If Length of string is N, then we need N steps for derivation from GNF.

Removal of Left Recursion:

$A \to A\alpha \mid \beta$ (single)

$\Rightarrow \begin{bmatrix} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{bmatrix}$

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \cdots A\alpha_n \mid \beta_1 \mid \beta_2 \cdots \mid \beta_n$ (Multiple)

$\Rightarrow \begin{bmatrix} A \to \beta_1 A' \mid \beta_2 A' \cdots \beta_n A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots \mid \alpha_n A' \end{bmatrix}$

* Left Recursion can be removed from every CFG.
* Indirect Left Recursion is also possible.

Removal of ∈ - production:

$S \to AB$, $A \to aA \mid \epsilon$, $B \to bB \mid b$ $\Rightarrow \begin{bmatrix} S \to AB \mid B \\ A \to aA \mid a \\ B \to bB \mid b. \end{bmatrix}$

$S \to AB$, $A \to aA \mid \epsilon$, $B \to bB \mid \epsilon$ $\Rightarrow \begin{bmatrix} S \to AB \mid B \mid A \mid \epsilon \\ A \to aA \mid a \\ B \to bB \mid b. \end{bmatrix}$

* if Lang. of grammar contains ∈, then can't remove.

Removal of unit Production:

$[V \to V]$ this is unit production.

$E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid id$ $\Rightarrow$ $E \to E+T \mid T*F \mid (E) \mid id$, $T \to T*F \mid (E) \mid id$, $F \to (E) \mid id$ $\Rightarrow$ $E \to E+T \mid T*F \mid (E) \mid id$, $T \to T*F \mid (E) \mid id$, $F \to (E) \mid id$.

Simplification of Grammar:

(a) Useless Variable : Variables that don't have terminating points are useless.

$\begin{bmatrix} S \to AB \mid AC \\ A \to a \\ B \to BC \mid BA \\ C \to b \end{bmatrix} \Rightarrow \begin{bmatrix} S \to AC \\ A \to a \\ C \to b \end{bmatrix}$

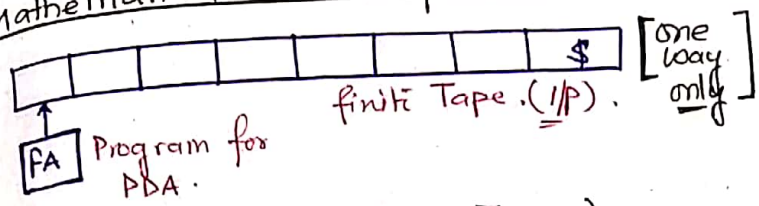
(b) Unreachable variable: If the variable doesn't lie in the Tree of start Var.

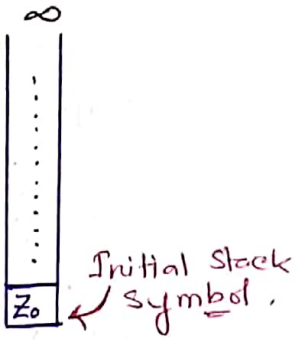
$\begin{bmatrix} S \to aB \mid ba \\ A \to bAA \mid aS \mid a \\ B \to aBB \mid bS \mid b \end{bmatrix} = \begin{bmatrix} S \to aB \mid ba \\ B \to aBB \mid bS \mid b \end{bmatrix}$

ToC - CFL

* CFL is more powerful than DCFL.

Mathematical Model of PDA :





7tuples : $(Q, q_0, F, \Sigma, \delta, \Gamma, Z_0)$

$\Gamma \rightarrow$ Stack Alphabet. $F \rightarrow$ Final state. (can be empty.)

$Z_0 \rightarrow$ Init. Stack symbol.

DPDA : $\delta : Q \times \{\Sigma \cup \epsilon\} \times \Gamma \longrightarrow Q \times$ push/pop/skip

NPDA : $\delta : Q \times \{\Sigma \cup \epsilon\} \times \Gamma \longrightarrow 2^Q \times$ push/pop/skip.

Acceptance by PDA → (a) empty stack (b) final state.] inter convertible, exceptions exist.

Instantenious Description of PDA :

$\delta(q_0, a, Z_0) \longmapsto (q, \text{push/pop/skip.})$.

Push : $\delta(q_0, a, Z_0) \vdash (q_1, AZ_0)$

$\delta(q_1, a, A) \vdash (q_1, AA)$

$\delta(q_1, b, A) \vdash (q_2, BA)$

pop : $\delta(q_1, b, A) \vdash (q_2, \epsilon)$

$\delta(q_2, b, B) \vdash (q_3, \epsilon)$

skip : $\delta(q_3, c, \underline{B}) \vdash (q_3, \underline{B})$

* We can push more than one element at a time, and also pop more than one element at a time.

CFL ∪ CFL = CFL

DCFL ∪ CFL = CFL

DCFL ∪ DCFL = CFL

DCFL ∩ DCFL = CSL

$DCFL^C = DCFL$ $CFL^C = CSL$

$CSL^C = CSL$ (may be CFL)

Set diff properties :

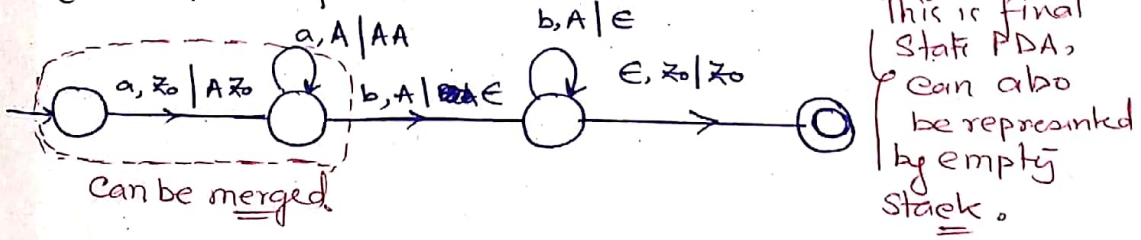
DCFL − DCFL = CSL	DCFL − R = DCFL
R − DCFL = DCFL	CFL − DCFL = CFL
DCFL − CFL = CSL	CFL − CFL = CSL

Important Points :

* By PDA we can't do more than 1 comparison.
* We can't recognize non Linear power. of PDA.
* String matching not possible [ie $L = \{ww\}$ not CFL]
  ↳ but reverse matching is possible. ie $\{ww^R\}$.

Some Important PDA's :

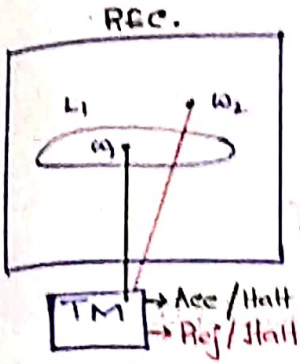
1. $L = \{a^n b^n \mid n \geq 1\}$



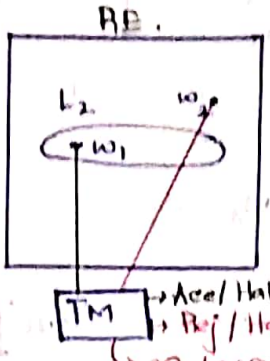
ToC- Closure Properties

OPERATION.	RL	DCFL	CFL	CSL	REC	R.E
1. Union (U)	ok	X	ok	ok	ok	ok
2. Intersection ($\cap$)	ok	X	X	ok	ok	ok
3. Complement (^{c})	ok	ok	X	ok	ok	X
4. Concat. (.)	ok	X	ok	ok	ok	ok
5. Set diff. (−)	ok	X	X	ok	ok	X
6. Kleene closure (*)	ok	X	ok	ok	ok	ok
7. +ve closure (+)	ok	X	ok	ok	ok	ok
8. Reversal (L^R)	ok	X	ok	ok	ok	ok
9. $L \cap RL$	ok	ok	ok	ok	ok	ok
10. $L - RL$	ok	ok	ok	ok	ok	ok
11. $RL - L$	ok	ok	X	ok	ok	X
12. $L \cup RL$	ok	ok	ok	ok	ok	ok
13. Prefix (L) or Init(L)	ok	ok	ok	ok	ok	ok
14. Subset.	X	X	X	X	X	X
15. ∞ Union	X	X	X	X	X	X
16. ∞ Intersection.	X	X	X	X	X	X

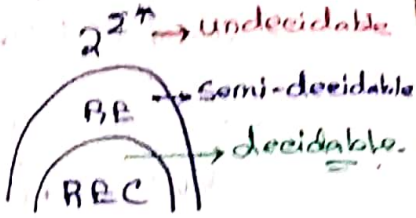
ToC - REC and RE



[Turing decidable]



[Turing Recognizable]

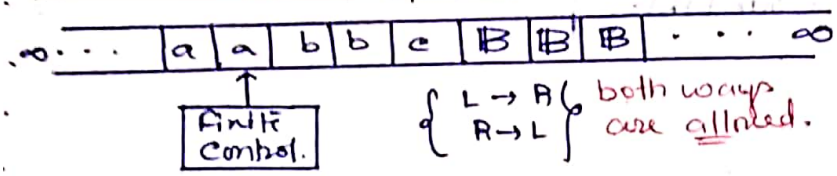


REC ∪ RE = RE
REC ∪ REC = REC
RE ∪ RE = RE

$REC^c = REC$.
$RE^c = 2^{\Sigma^*}$

$(2^{\Sigma^*})^c = RE$ → only theoretically possible, practically we can't know.

Mathematical Model of T.M:



7 Tuple : $\{q_0, Q, \Sigma, B, \Gamma, F, \delta\}$

Γ → Tape Alphabet B → Blank Symbol.

- FA + ≥2 Stack = TM
- FA + ∞ Tape but no mem. = FA
- DFA + both dir = 2 DFA = FA
- FA + Queue = TM
- FA + 2 counter = TM
- PDA + Stack = TM

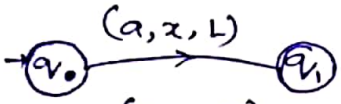
DTM $\delta : Q \times \Gamma \to Q \times \Gamma \times (L/R)$
NTM $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times (L/R)}$

(*) TM can behave as : → [Transducers, Acceptors, Enumerators]

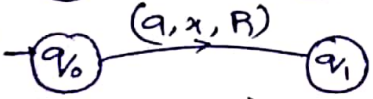
*Instantaneous description :

$\delta(q_0, a) \vdash (q_1, x, L/R)$ ⇒ q_0 state reads a, goes to state q_1, and writes x in place of 'a', and moves Left or right.

Ex : (a)



(b)



Standard Well Known Problems :

1. State Entry Problem : [Given a T.M, a state $q \in Q$ and $w \in \Sigma^+$, decide if the state q is ever entered ?] Undecidable.
2. Halting Problem : [Given description of TM and i/p : $w \in \Sigma^+$, does the m/c started w/ 'w' as i/p hlt ?] Undecidable.
3. Blank Tape Halting Problem : [Given a T.M, if the m/c halts or not when started w/ Blank Tape ?] Undecidable.

NOTE The Language which contains encoded T.M as a string (L_V) is recursive Language, and so we can make Halting TM (HTM) for that.

∴ L_V is R.E.C.

Universal Turing M/c

Its a T.M, where provided i/p itself is a T.M.

$$U = \{ L = \{ \langle TM, w \rangle \} \}$$

* The set of all strings of the form $\langle M, w \rangle$ accepted by UTM is called Universal Language.

$$L_U = \{ \langle TM_1, w \rangle, \langle TM_2, w \rangle, \ldots \ldots \}$$

Countability:

[Finite no of interval b/w 2 number / item] ⇒ Countable.

$N = \{1, 2, \ldots \infty\}$ Countable Even $= \{2, 4, \ldots \infty\}$ Countable

Prime $= \{2, 3, 5, 7 \ldots \infty\}$, countable.

$R = \{ -\infty \ldots -0.1 \ldots 0.2 \ldots \infty \}$ uncountable.

$$\Sigma^* = (a+b)^* = \text{Countable.}$$

NOTE

- ⇒ Subset of countable set is always countable.
- ⇒ Set of all TM's are countable.
- ⇒ Set of all RE's are countable.
- ⇒ Union of 2 or more countable set is always countable.
- ⇒ Power set of ∞ countable set = uncountable.

$P(\infty \text{ countable}) = \text{uncountable}$
$P(\text{uncountable}) = \text{uncountable.}$

$2^{\Sigma^*} = \text{uncountable}$

- ⇒ Cartesian Product of 2 countable set is always countable.
- ⇒ Complement of countable, may or may not be countable
- ⇒ Complement of uncountable may or may not be countable.
- ⇒ Intersection may or may not be countable.

Computational Complexity:

* We cannot reduce an undecidable problem to a decidable in polynomial time.

1. $P_1 \leq_P P_2$ — P_1: undecidable; P_2: this has to be undecidable.
2. $P_1 \leq_P P_2$ — P_1: decidable; P_2: decidable
3. $P_1 \leq_P P_2$ — P_1: decidable; P_2: need not be decidable
4. $P_1 \leq_P P_2$ — P_1: may or may not be decidable; P_2: undecidable.

ToC- decidability

Problem :	RL	DCFL	CFL	CSL	REC	RE
1 $\omega \in L$	D	D	D	D	D	X
2 $L = \emptyset$	D	D	D	X	X	X
3 $L = \Sigma^*$	D	X	X	X	X	X
4 $L_1 = L_2$	D	X	X	X	X	X
5 $L_1 \subseteq L_2$	D	X	X	X	X	X
6 $L_1 \cap L_2$	D	X	X	X	X	X
7 L is finite	D	D	D	X	X	X
8 Complement is of same type.	D	D	X	D	D	X
9 If L is RL..	D	D	X	X	X	X

* $\{a^n b^n c^n d^n \ldots z^n \ldots\}$ is CSL, but its complement is always CFL.

CD - Introduction

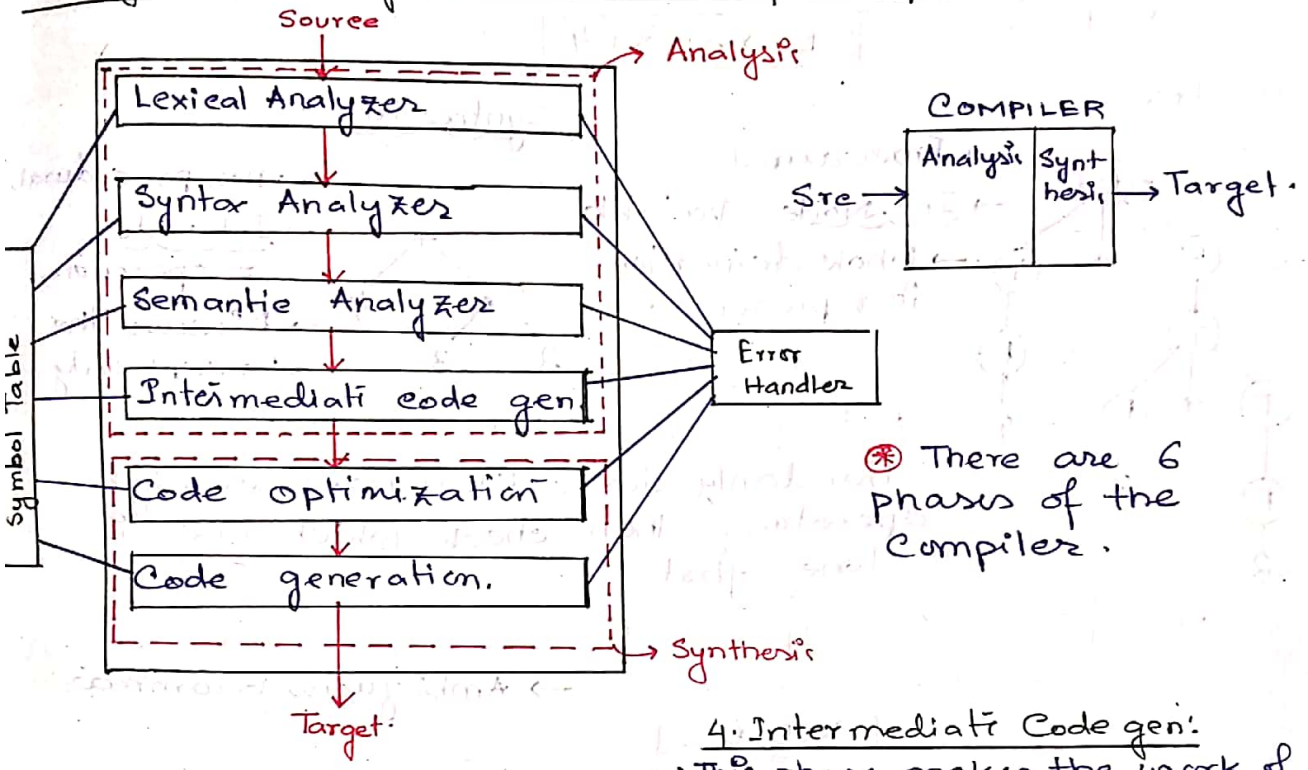
Defn: Convert high level language to low level language.

(any P.L.) Source → Translator → (any P.L.) Target

* High Level Language can perform more than one operation in a single statement.

* Low Level Lang can perform atmost one operation in a stmt.

Analysis and synthesis model of Compiler:



COMPILER

* There are 6 phases of the Compiler.

1. Lexical Analyzer:

Program of DFA, it checks for spelling mistakes of program.

* Divides source code into Stream of Tokens

2. Syntax Analyzer:

Checks grammatical errors of the program. (Parser)

* Parser is a DPDA.

3. Semantic Analyzer:

Checks for meaning of the program.

[Eg: Type miss match, Stack Overflow.]

* w/ Error handler, compiler can still work.

4. Intermediate Code gen:

→ This phase, makes the work of next 2 phases much easier.

→ Enforces reuseability and portability.

5. Code Optimization:

1.) Loop invariant construct
2.) Common sub expr elimination
3.) Strength Reduction
4.) function inlining.
5.) Dead code elemination.

6. Symbol Table:

1) Data about data (meta data)
2) Data Structure used by Compiler and shared by all the phase.

* w/o Symbol Table compiler cannot work.

CD- Grammar

⇒ In compiler we only use : Type-2(CFG) and Type-3(RG)
⇒ Compiler = Program of Grammar
⇒ Compiler = Membership Algorithm.

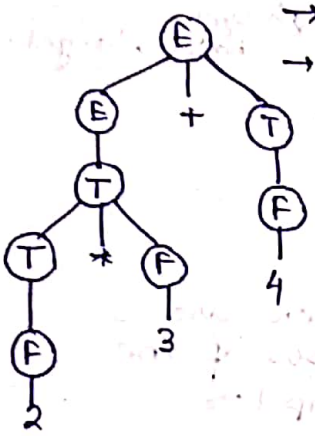
* Every programming Language is CSG. (CSL).

Parse Tree and Syntax Tree :

G: $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

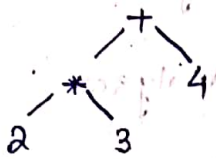
$E \to E+T \to T+T \to T*F+T \to F*F+T$
$\to 2*F+T \to 2*3+T \to 2*3+F$
$E \to 2*3+4$

Parse Tree :



→ Procedural
→ Int. Node = Variable
→ Whole derivation is represented.

Syntax Tree :



→ non procedural
→ Int. node = operator.
→ Discuss the concept only.

* To check to priority / Associativity :
Randomly derive till u have enough operators, then check which one is done first.

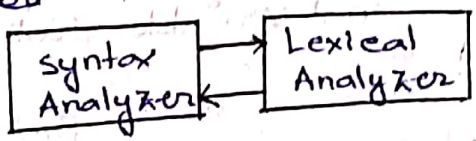
* If priority of 2 operators are same and both Left and Right associative, → Ambiguous Grammar [USELESS]

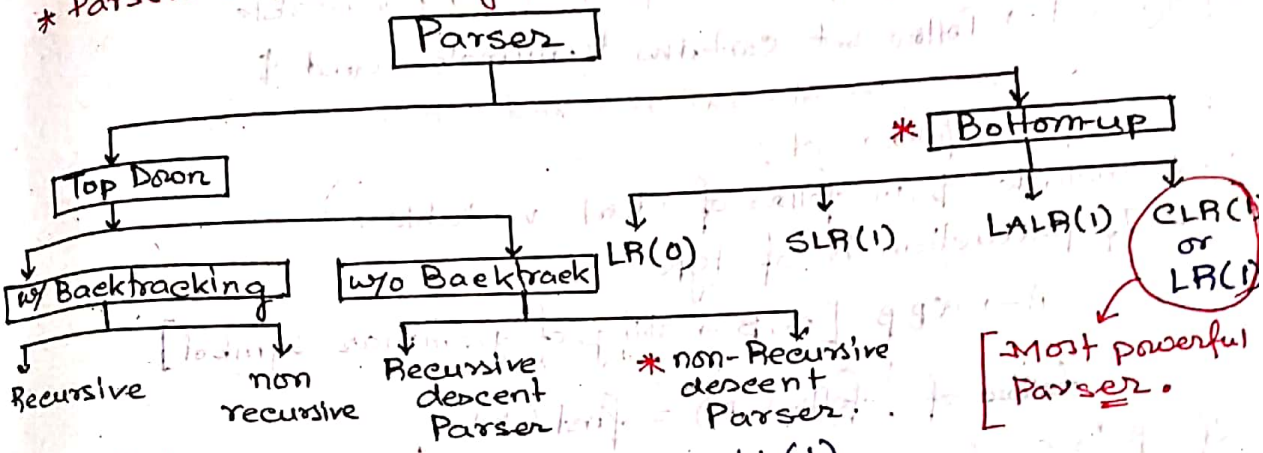
Types of item in Bottom Up : [CD- Parser]

item
- reduced ($X \to \alpha.$) dot(.) at the end.
- unreduced
  - State ($X \to \alpha . t \beta \mid t \in$ non-terminals)
    ↳ Variable after(.).
  - Shift ($X \to \alpha . t \beta \mid t \in$ terminals)
    ↳ Terminal after dot(.).

Grammatical Errors are ~~Found~~ checked w/ the help of parsers.

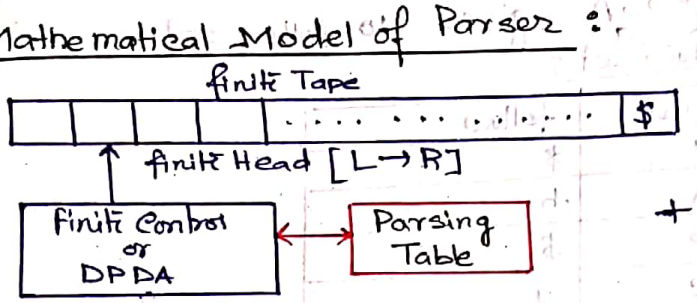
* Parsers are basically DPDA's.





* All of these parsers are Table driven.

Mathematical Model of Parser :



⇒ Parser = DPDA + Parse Table.

* Parsers generate Parse Tree, for a given string by the given grammar

Top down Parser (LL(1)) :

* It uses LMD and is equivalent to DFS in Graph.

Algorithm to construct Parsing Table :

1. Remove Left Recursion. if any.
2. Left factor. [Remove common ~~Suffixes~~ prefix.]
3. Find 1st and Follow set
4. Construct the Table.

* If we increase the look ahead symbol:
→ Strength of parser ↑
→ Complexity of parser ↑

* [Due to Common Prefix: Backtrack
Due to Left Recursion: ∞ Rec.]

Removal of Common Prefix : (Left factor).

1. $S \to a \mid ab \mid aA$

⇒ $S \to aY$
$Y \to \epsilon \mid b \mid A$.

2. $A \to abA \mid aA \mid b$.

⇒ $A \to aX \mid b$
$X \to bA \mid A$

* indirect common prefix

$A \to aX \mid b \;\|\; X \to bA \mid aX \mid b$

⇒ $A \to aX \mid b$
$X \to aX \mid bY$
$Y \to A \mid \epsilon$

First and Follow:

- **First Set** → extream Left terminal from which the string of that variable starts.
  - ✱ it never contains variables, but may contain 'ϵ'.
  - ✱ We can always find the first of any variable
- **Follow set** → Follow set contains terminals and $.
  It can never contain variable and "ϵ".

How to find follow set?

1. Include $ in follow of start variable.
2. If production is of type →

   $A \rightarrow \alpha B \beta$ [α, β → strings of grammar symbol.]

   ~~follow of~~ $\boxed{\text{follow}(B) = \text{first}(\beta)}$

If, $\beta \xrightarrow{*} \epsilon$, ie $A \rightarrow \alpha B$, then $\boxed{\text{follow}(B) = \text{follow}(A)}$

✱ Productions Like: (A) → aA gives No follow set.

Examples of first and follow Set:

1. $S \rightarrow AB \mid cD$
   $A \rightarrow aA \mid a$
   $B \rightarrow bB \mid b$
   $C \rightarrow eC \mid e$
   $D \rightarrow dD \mid d$

	First	follow
S	a, c	$
A	a	b
B	b	$
C	c	d
D	d	$

Entry into Table : Top down :

1. No of Rows = No of unique variables in Grammar
2. No of Columns = [Terminals + $]
3. For a Variable (Row) fill the Col (terminal) if it is there in its first by the production req'd.
4. If ϵ is in first put V → ϵ under $ and its follow.

✱ if any cell has multiple items, then it not possible to have LL(1) Parser. Since that will be ambiguous.

✱ [In Top down we do: derivation
In Bottom-Up we do: Reduction]

Q) Construct LL(1) Parsing Table for the given grammar:

$E \to E+T \mid T$; $T \to T*F \mid F$; $F \to (E) \mid id$; } G_0

⇒ Removing Left Recursion:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$ } G_1

	first	follow
E	(, id	$,)
E'	+, ϵ	$,)
T	(, id	+, $,)
T'	*, ϵ	+, $,)
F	(, id	*, +, $,)

② left factoring not Reqd.

Construction of Table: [LL(1)]

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

④ Since for G_1, Table constructed w/ no multiple entries, hence successfully completed. Hence G_1 is LL(1).

Q) Construct LL(1) Parsing Table for the following grammar:

$S \to L=R \mid R$; $L \to *R \mid id$; $R \to L$. } G_0

Left Factoring:

$S \to L=R \mid L$
$L \to *R \mid id$
$R \to L$

⇒

$S \to LX$
$X \Rightarrow =R \mid \epsilon$
$L \to *R \mid id$
$R \to L$ } G_1

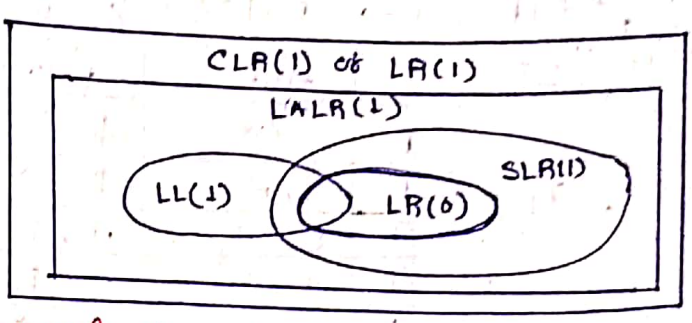
	first	follow
S	*, id	$
X	=, ϵ	$
L	*, id	$, =
R	*, id	$, =

Construction of Table:

	*	=	id	$
S	$S \to LX$	error	$S \to LX$	error
L	$L \to *R$	error	$L \to id$	error
R	$R \to L$	error	$R \to L$	error
X	error	$X \to =R$	error	$X \to \epsilon$

⑤ G_1 is a LL(1) Grammar

Hierarchy of Parsers : [for ε-free Grammar]



* For ε-producing grammars, every LL(1) may not be LALR(1).

NOTE We can't construct any ~~grammar~~ parser for ambiguous grammar. except: operator precedence parser possible for some ambiguous grammar.

* there are some unambiguous grammar, for which there are no parsers.

Eg: G: $S \to aSa \mid bSb \mid a \mid b$ | unambiguous but no parser.

$L(G) = w(a+b)w^R$ (odd palindrome).

⇒ Every RG is not LL(1) as it may be ambiguous, or Recursive or Common Prefix.

⇒ Parsers exists only for the grammar if its Lang. is DCFL.

⊛ There are some grammar whose Lang is DCFL but no parser is possible for it.

Operator Precedence Grammar:

Format: 1. No 2 or more variable side by side
2. No ε production.

Eg $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid id$ — O.G.

or $E \to E+E$, $E \to E \times E$, $E \to a \mid b$ — O.G.

$S \to aSa \mid bSb \mid a \mid b$ — O.G.

or, $S \to AB$, $A \to aA \mid \epsilon$, $B \to bB \mid \epsilon$ — Not O.G.

Checking LL(1) w/o table:

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$, then →

$\text{first}(\alpha_1) \cap \text{first}(\alpha_2) = \emptyset$
$\text{first}(\alpha_1) \cap \text{first}(\alpha_3) = \emptyset$
$\text{first}(\alpha_2) \cap \text{first}(\alpha_3) = \emptyset$

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \epsilon$.

$\text{first}(\alpha_1) \cap \text{first}(\alpha_2) = \emptyset$
$\text{first}(\alpha_1) \cap \text{first}(\alpha_3) = \emptyset$
$\text{first}(\alpha_2) \cap \text{first}(\alpha_3) = \emptyset$
$\text{follow}(A) \cap \text{first}(\alpha_1) = \emptyset$
$\text{follow}(A) \cap \text{first}(\alpha_2) = \emptyset$
$\text{follow}(A) \cap \text{first}(\alpha_3) = \emptyset$

Bottom-UP Parsers:

→ It uses RMD in reverse.
and has no problem w/ :
(a) Left Recursion
(b) Common Prefix.

SLR(1), LR(0) } LR(0) items
CLR(1), LALR(1) } LR(1) items

LR(1) = LR(0) + 1 lookahead.

→ No Parser possible for ambiguous grammar.
→ There are some unambiguous grammar for which there are no Parser.
→ The Language of the grammar must be DCFL.

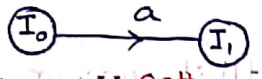
Basic Algorithm for Construction:

→ Augment the grammar and expand it. And give numbers to it.
→ Construct LR(0) or LR(1) item set.
→ From that fill the entries in the Table accordingly.

Types of Entries:

① Shift Entries:

Transitions on terminals.



a ∈ Terminal
I0, I1 : item sets.

entry: ... action ...

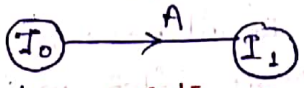
	a
I0	S1

* Shift entries are same for all Bottom-up Parser.

② State entry:

Transition on non-terminal. (Variables)

A → Variable.



entry <... goto ...>

	A
I0	1

* Same for all Bottom up Parser.

③ Reduce Entry:

Done for each separate production in the item set of type: $i > X \to \alpha .$

where [i → Prod. No
X → Producing Var.
α → Grammar String]

In:

(a) LR(0) Parser: Put R_i in every cell of the set in action table (ALL)

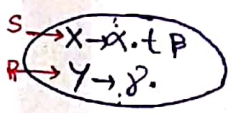
(b) SLR(1) Parser: Put R_i only in the Follow(X) form the Grammar. (Follow(X))

(c) LALR(1) and CLR(1): Put Ri only in the lookaheads of the production (Lookaheads)

Conflicts:

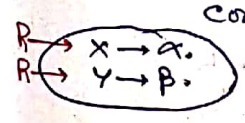
LR(0) Parser:

SR: Shift Reduce Conflict



then SR Conflict

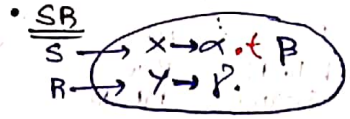
RR: Reduce Reduce conflict.



then RR.

SLR(1) Parser

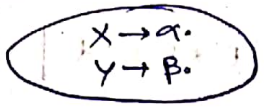
• SR



$\not\in$ then SR conflict

't' ∈ Follow(Y)

• RR



$\not\in$ then RR

Follow(X) ∩ Follow(Y) ≠ ∅

LALR(1) and CLR(1) : Same as SLR(1), but instead of use the provided Lookahead.

SR

$X \to \alpha . t \beta, L_1$
$Y \to \gamma ., L_2$
$\$$ ~~$t \in$ Follow~~
$t \in L_2$
} then SR conflict.

RR

$X \to \alpha ., L_1$
$Y \to \beta ., L_2$
$L_1 \cap L_2 \neq \phi$
} then RR conflict.

Inadequate State: A state having ANY conflict is called a conflicting state or 'inadequate state'.

NOTE The state $S' \to S.$ or $S' \to S., \$$ is accepted state, and, this is not a reduction.

* The only difference b/w CLR(1) and LALR(1) is that, the states w/ the similar items, but diff Lookaheads are merged together to Reduce space.

$$\#\text{States in LR(0)} = \#\text{State in SLR(1)} = \#\text{States in LALR(1)} \leq \#\text{State in CLR(1)}$$

Important Points:

1. If CLR(1) doesn't have any conflict, then conflict may or may not arise after merging in LALR(1)
2. If LALR(1) has SR-conflict, then we can conclude that CLR(1) also has SR-conflicts.
3. LALR(1) has SR- conflict if and only if CLR(1) also has SR.

* We can construct parser for every ~~CLR(1)~~ unambiguous regular grammar : [CLR(1) Parser].

- **L** ← Left to Right Scan of i/p
- **R** → using reverse RMD
- **(0)** → No Lookahead

- **S** Simple
- **L** L to R scan
- **R** using Rev RMD
- **(1)** Lookahead

- **L** Look
- **A** ahead
- **L** L to R scan
- **R** Rev RMD
- **(1)** Lookahead.

- **C** Canonical
- **L** L to R scan.
- **R** Rev RMD
- **(1)** Lookahead.

* **Very Important Point**:

LALR(1) Parser can Parse non LALR(1) grammar, which only has SR- conflict by favoring shift over reduce.

Eg: $E \to E+E \mid E*E \mid id$ | $2+3*5 \Rightarrow E+E . *5$ (shift)

CD - Lexical Analysis

char stream → Lexical A. → Token stream

→ Token Generation
→ Scanner
→ Linear Phase
→ Token Recognizer

• defn: Scan the whole program, char. by char and produces the corresponding token.

* also produces/reports the Lexical Lexical Errors (if any).

• Function of Lexical Analyzer :
i) Scans all the char. of the program.
ii) Token Recognizer.
iii) Ignores the comment & Spaces
iv) Maximal Munch Rule [Longest Prefix match]

NOTE The Lexical Analyzer uses, the Regular Expression.
- prioritization of Rules.]
- Longest Prefix match]

• Lexeme → Smallest unit of program or Logic.
• Token → internal representation of Lexeme.

Lexemes → LA → Tokens

• Types of Token :
① Identifier
② Keywords
③ operators
④ Literals/constants.
⑤ Special Symbol.

* all identifier will have entry in symbol Table, LA, gives entries into the symbol Table.

• Token Separation
1. Spaces
2. punctuation

• Implementation
→ LEX tool ⇒ Lex.yy.c

[Regular Expression → DFA → Lexical Analyzer]

Find No. of Tokens :

① `void main ( ) { printf ("gate"); }` {11 Tokens}

② `int x, *p; x = 10; p = &x; x++;` [18 Tokens]

③ `in/*prog*/t x;` ↳ `in t x;` [4 Tokens]

④ `int x; x=y; x==y;` [11 Tokens]

⑤ `int 1x23;` ↳ ? [Lexical Error]

⑥ `char ch = 'A';` [5 Token]

⑦ `char ch = 'A;` ? Lexical Error.

⑧ `char *p = "gate";` [6 Tokens]

⑨ `char *p = "gati;` ? Error.

⑩ `int x = 10; /*comment x=x+1;` ... ? ERROR.

⑪ `int x = 10; comment */ x = x+1;` [14 Tokens]

CD-Syntax Directed Translation.

CFG + Translation ⇒ Syntax + Translation] SDT.

SDT: CFG + Translation → Meaning / Semantic.

Application of SDT:
1. Used to perform Semantic Analysis
2. Produce Parse Tree
3. Produce intermediate Rep.
4. Evaluate an expression
5. convert infix to prefix or postfix.

Example:
$S \to S_1 S_2$ [S.count = S_1.count + S_2.count]
$S \to (S_1)$ [S.count = S_1.count + 1]
$S \to \epsilon$ [S.count = 0]

(*) count is an attribute for non-Terminal.

Attributes: ① Inherited Attribute ② Synthesized Attribute.

$S \to AaB$

S is parent to A, a, B
A is child to S
A is Left Sibling to a, B.

Inherited Attribute: (RHS)

$S \to A\,B$ { A.x = f(B.x | S.x) }

The computation at any node de (non-terminal) depends on parent or sibling(s).

* In Above example, x is inherited attribute.

Synthesized Attribute: (LHS)

$S \to A\,B$ { S.x = f(A.x | B.x) } x is synthesized Attribute.

The computation at any node (non-terminal) depends on children.

Identifying Attribute Type: → (*) Always check every Translation.

① $D \to T\,L$; { L.Type = T.Type } inherited.
$L \to L_1$, id ; { L_1.Type = L.Type } inherited.
$L \to$ id
$T \to$ integer { T.type = int } synthesized.

Type is neither inherited nor Synthesized.

② $E \to E_1 + T$ { E.val = E_1.val + T.val } Synth.
$E \to T$ { E.val = T.val } Synth.
$T \to T_1 - F$ { T.val = T_1.val - F.val } Synth
$F \to$ id { F.val = id.val } Synth

Val is synthesized attribute.

③ $S \to AB$ { A.a = B.x ; S.y = A.x } x is inherited | y is synth
$A \to a$ { A.y = a } y is synth
$B \to b$ { B.y = a.y } y is synth

x ⇒ inherited
y ⇒ Synthesized.

④ $D \to TL$ { L.in = T.type } inherited (in)
$T \to$ int { T.type = int } Synth (type).
$L \to$ id { AddType (id.entry, L.in) }

in ⇒ inherited
type ⇒ synthesized

Syntax Directed Definations (SDDs): (Attribute Grammar)

① L-Attributed Grammar:

→ Attribute is synthesized or restricted inherited.
  - Parent
  - Left Sibling

  ] only.

→ Translation can be appended anywhere in RHS of production

Eg: $S \to AB \{ A.x = S.x + 2 \}$
or, $S \to AB \{ B.x = f(A.x \mid S.x) \}$
or, $S \to AB \{ S.x = f(A.x \mid B.x) \}$

→ Evaluation: In Order (Topological)

② S-attributed Grammar:

→ Attribute is synthesized only

→ The Translation is placed only at the end of production.

Eg: $S \to AB \{ S.x = f(Ax \mid Bx) \}$

→ Evaluation: Rev. RMD (Bottom Up Parsing)

Identify SDD:

① $E \to E_1 + E_2 \{ E.type = if(E_1.type == int \ \&\& \ E.type == int) \ then \ int \ else \ type\text{-}error. \}$ Synth.
$E \to id \{ E.type = Lookup(id.entry) \}$ Synth.

∴ type is synthesized, hence S-Attribute and also L-attributed Grammar

* Every S-attributed Grammar is also L-attributed Grammar.
* For L-attibuted Evaluation, use the In-order of annotated Parse Tree.
* For S-attributed, Reverse of RMD is used.
  → Find RMD order
  → Consider its Reverse.

CD - Intermediate Representation

Representations

Linear Form
- postfix Code
- 3-Addr Code (3AC)
- static - single assignment (SSA code).

non-Linear form.
- Syntax Tree
- Directed Acyclic graph (DAG)
- Cmbol Flow Graph

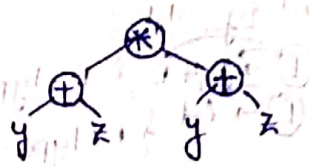
Example expression : $(y+z) * (y+z)$.

Post-fix → $yz+\ yz+\ *$

SSA → $t_1 = y+z$, $t_2 = t_1 \times t_1$ ⇒ t_1 and t_2 cannot be reassigned

3AC → $t_1 = y+z$, $t_2 = t_1 * t_1$

Syntax Tree →



DAG →

⇒ Reuse the already existing common subexpression.

3-Address Code: Code in which, atmost 3 addresses. [Including LHS].

⊛ address → name, Variable, Constant.

$x = y + z * y - a$

(2) (3)

⇒ $t_1 = z * y$, $t_2 = y + t_1$, $t_3 = t_2 - a$ — Equivalent 3-addr Code.

$x = t_3$.

Representation of 3AC

Triple not.

0	*	z	y
1	+	y	(0)
2	-	(1)	a
3	=	(2)	

Quadruple not.

0	*	z	y	t_1
1	+	y	t_1	t_2
2	-	t_2	a	t_3
3	=	t_3		x

indirect Triple not.

6	(0)
7	(1)
8	(2)
9	(3)

~~Ex~~ Indirect Notation → pointers to the rows of Triple.

Triple Notation
- → Space efficient
- → time inefficient.

Quadruple
- → Space ineffi.
- → time effi.

⊛ 3AC done using operator precidence.

Find min no of variable read in equivalent 3AC:

① $x = y - t$, $y = x * v$, $x = y + w$, $y = t - z$, $y = x * y$.

7 variables.

$x * y \Rightarrow x * (t-z) \Rightarrow (y+w) * (t-z)$

$\Rightarrow ((x * v) + w) * (t - x)$

$\Rightarrow (((u-t) * v) + w) * (t-z)$.

$u = u - t$, $v = u * v$, $w = v + w$, $v = t - z$, $z = w * v$ ⇒ 5 variables only

⇒ Evaluating the expression:

$a \Rightarrow e+b \Rightarrow d-b+b \Rightarrow b+c-b+b$

$\Rightarrow b+a+d-b+b$

$\Rightarrow b+b+c+d-b+b$

$\Rightarrow b+b+c+d$

$a = b+c$
$c = a+d$
$d = b+c$
$c = d-b$
$a = e+b$

$t_1 = b+b$
$t_2 = t_1+c$
$a = t_1+d$

Minimum 3AC.

[wrong, even t_1 is not used]

∴ Minimum:

$b = b+b$
$c = b+c$
$a = c+d$

⇒ only 3 variable.

[Most optimal.]

Static Single Assignment code (SSA code):

Every variable (addr) in the code has single assignment.

[single meaning] + 3AC.

① $x = u-t$ ⇒ [u, t, v, w, z] are already assigned.

(16) $y = x*v$ So we can't use them.

$x = y+w$
$y = t-z$
$y = x*y$

Find SSA?

⇒ Equivalent SSA code:

$x = v-t$
$y = x*v$
$p = y+w$
$q = t-z$
$r = p*q$

SSA code.

in use: x, y, p, q, r (additional)

∴ Total Variables ⇒ 10.

② (Static 2017)

$p = a-b$
$q = p*c$
$p = u*v$
$q = p+q$

⇒ [a, b, c, u, v] are already assigned.

⇒ Equivalent SSA code:

$p = a-b$
$q = p*c$
$p_1 = u*v$
$q_2 = p_1+q$

SSA code.

in use: p, q p_1, q_2

∴ Total Variable = 9.

Control Flow Graphs:

- Basic Blocks: Seq. of 3-addr code, in which control enters from 1st stmt and exits from last.

  (*) Basic Blocks can never contain Jump statements in b/w.

  Find Leaders to identify basic blocks.

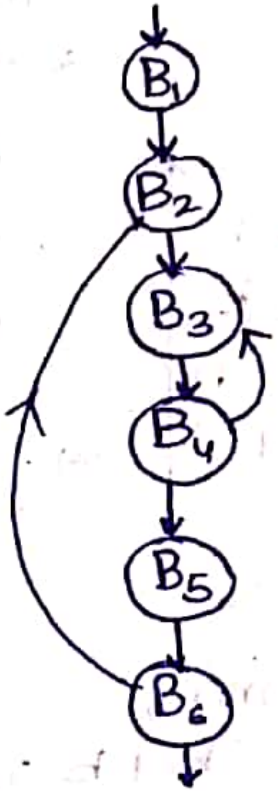
  → 1st 3AC is Leader
  → Target of Jumps are Leaders
  → Statement Just below Jump are Leaders
  → Jump is itself a Leader.

Example:

1. $i = 1$ — L — B_1
2. $j = 1$ — L — B_2
3. $t_1 = 5 * i$ — L
4. $t_2 = t_1 + 5$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = 1$
8. $j = j + 1$

B_3

9. if $j \leq 5$ goto 3 — L — B_4
10. $i = i + 1$ — L — B_5
11. if $i < 5$ goto 2 — L — B_6



[6 vertices
7 Edges]

CD - Code Optimization

→ Saves Space / time. (Basic Objective)

optimization

- High Level code optimization
- Intermediate code optimization
- Assembly code optimization (M/c dependent.)

(High Level code optimization + Intermediate code optimization) → M/c independent

Flow Analysis

- Control Flow Graph
- Data Flow Graph.

Optimization

- Local (Intra Procedural) Basic Blocks
- Global (Inter Procedural) Whole Program.

optimization Methods :

1. Constant Folding
2. Copy propagation
3. Strength Reduction
4. Dead Code elimination
5. Common Sub expr elimination
6. Loop optimization
   - Code Motion
   - Induction Variable elimination
   - Loop Merging
   - Loop unrolling

① Constant Folding :

$x = 2*3+y \Rightarrow x = 6+y$ (folding)

$x = 2+y*3$] Can't fold the constants.

② Copy Propagation :

i) Variable Propagation:

$x = y;$ ⟹ $z = y+2;$

$z = x+2;$

ii) Constant Propagation:

$x = 3$

$z = x+a;$ ⟹ $z = 3+a;$

③ Strength Reduction :

Replace expensive statement / instruction with cheaper one.

$x = 2*y$ (Costly) ⟹ $x = y+y;$ (Cheap)

$x = 2*y$ ⟹ $x = y << 1;$ (Much cheaper.)

$x = y/8;$ ⟹ $x = y >> 3;$

④ Dead Code Elimination :

```
x = 2;
if (x > 2)          // FALSE  → dead code
    printf("code");
else                // can be removed
    printf("optimization");
```

⟹

```
x = 2;
printf("optimization");
```

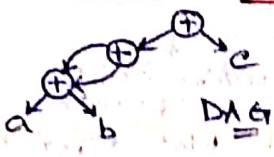
* Hence, it's a code, that never execute, during execution. We can always delete such code.

Common Sub expression elimination.

DAG is used to eliminate common sub expression.

Eg: $x = (a+b) + (a+b) + c;$ ⟹

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



DAG. [6 Nodes, 6 edges]

⑥ Loop Optimization :

(i) Code Motion - Freq Reduction :

Move the loop invariant code outside of Loop.

```
for(i=0; i<n; i++){
    x = 10;      invariant
    y = y+i;
}
```

⟹

```
x=10;
for(i=0; i<n; i++){
    y = y+i;
}
```

(ii) Induction Variable elimination :

```
i1=0;
i2=0;
for(i=0; i<n; i++){
    A[i1++] = B[i2++];
}
```

⟹

```
for(i=0; i<n; i++){
    A[i] = B[i];
}
```

only 1 induction variable : i

i, i_1, i_2 : 3 induction Variables

(iii) Loop Merging / Combining : (Loop Jamming)

```
3n+2  for(i=0; i<n; i++)
          A[i] = i+1;
3n+2  for(j=0; j<n; j++)
          B[j] = j-1;
```

6n+4

⟹

```
for(i=0; i<n; i++){
    A[i] = i+1
    B[i] = i-1
}
```

4n+2

Reduced.

(iv) Loop unrolling :

```
① for(i=0; i<3; i++)
      printf("CD");
```

⟹

```
printf("CD");
printf("CD");
printf("CD");
```

3×3+2 = 11 statements

3 statements.

```
② for(i=0; i<2n; i++){
      printf("CD");
  }
```

⟹

```
for(i=0; i<n; i++){
    printf("CD");
    printf("CD");
}
```

(2×3n+2) = 6n+2

(4n+2)