FLC Theory :-

Fall-23

(1) a) **Formal Language** : A set of strgs formed from a specific alphabet, governed by grammatical rules.

In prog. lang, it define the syntax and structure of lang using mathematical models. Uses in PL and Compiler desig

① **Lexical Structure** :- Describes the smallest units (tokens) like Keywords if, while and identifiers x, total etc. also Literals "3.14", "hello" etc.

② **Syntax Structure** :-
* Specifies how tokens can be combined to form valid programs.
* Defines using (CFG) context free grammars (parsed using pushdown automata)

③ **Semantics** :- Formalizes, defines meaning of syntactically correct sentences
Analysis.

b)

Formal	Informal
* Structured set of rules for symbol strg	* used by human in daily life
2. Strict, unambiguous grammar	2. Flexible, often ambiguous.
3. programming, maths, logic	3. Used for human interaction
4. Machine readable precise meaning	4. multiple interpretations
5. C, Java, HTML, Logic expressions.	5. English Bengali, French.

c) **Analysis phase:**

① Reads src prog, splits it into multiple tokens and constructs it intermediate representation of src prog.

② Checks, indicates syntax & semantic errors of a src prog.

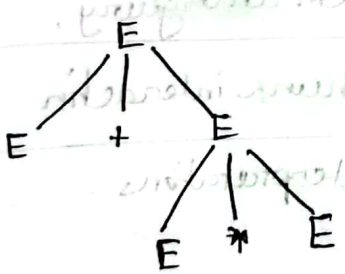
③ Collects info abt src. prog — prepares symbol table.
Symbol table will be used all over the compilation process.

④ Also called as the front end of the compiler.

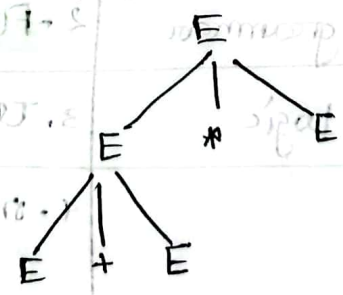
d) **Ambiguous Grammar:-** When grammar fails to provide unique structure for each string in it's Language.

~~Ex:~~
How it affects parse tree:
Suppose E+E*E
It has 2 derivations: ① E→E+E→E+E*E
② E→E*E→E+E*E

In 1, the 2nd E is replaced by E*E and in 2 the 1st E is replaced by E+E. Thus the parse tree will be,



(1)



(2)

according to (1), 1+2*3 will be 1+(2*3) = 7. and for 2 (1+2)*3 = 9. Thus, 2 or more parse tree is causing ambiguity.

(Proof: a CFG = (V, T, P, S) is ambiguous if at least one string w in T^* for which 2 diff parse trees can be found, each with root labeled S and yield w.)

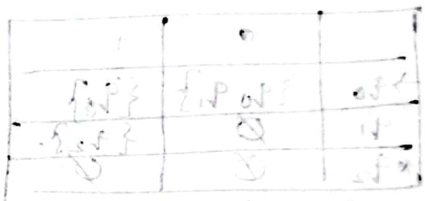
(multiplicity of derivation do not cause ambiguity:

1. $E \to E+E \to I+E \to a+E \to a+I \to a+b$.
2. $E \to E+E \to E+I \to I+I \to I+b \to a+b$.)

---

(1) $E \to E+E$
$\to I+E$
$\to a+E$
$\to a+E*E$
$\to a+I*E$
$\to a+a*E$.
$\to a+a*a$

$E \to E*E$
$\to E+E*E$
$\to I+E*E$
$\to a+E*E$
$\to a+I*E$.
$\to a+a*E$
$\to a+a*I$
$\to a+a*a$.



How to reduce:

(1) refine grammar adding rules: operator precedence
Left/right associativity.

(2) Use unambiguous grammars designing CFG that produce only one parse tree per string.

e) NFA → Nondeterministic Finite Automata is a quintuple (5-tuple). that is, a system which consists of 5 elements.

It is used to recognize regular lang that allow multiple transitions for the same symbol. and ε transitions which allow state changes without consuming input.

Transition Function of NFA: Takes as arguments a state and an input symbol and returns a subset of Q.

Example: Consider sequence of 0's and 1's with 01 in the end.

~~let it be 00~~

	0	1
→q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	∅	$\{q_2\}$
*q_2	∅	∅

Transition table.

$Q = q_0, q_1, q_2$

$\Sigma = 0, 1$

$q_0 = q_0 \in Q$

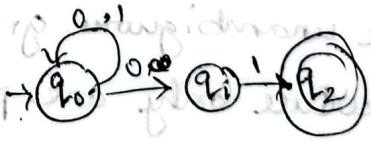
$F = q_2$

Basis: $\delta(q, \varepsilon) = \{q\}$, without reading input symbols, we are only in the state we began

Induction: suppose $w = xa$, where a is final symbol of w and x. is the rest of w. Suppose that $\hat{\delta}(q, x) = \{p_1, p_2 \dots p_k\}$.

Let, $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \dots r_m\}$.

Then, $\hat{\delta}(q, w) = \{r_1, r_2 \dots r_m\}$.



~~Example~~

So For 00101 →

① $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$

② $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$

③ $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$

④ $\hat{\delta}(q_0, 001) = \delta(q_0, 0) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

⑤ $\hat{\delta}(q_0, 001.0) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$

⑥ $\hat{\delta}(q_0, 0010) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

(Ans)

f).

Bottom Up	(vs) Top Down Parsing
① L to right input parses, leaves to root tree	① Left to right Tree → root to leaves
② Begins from input symbol	② from start symbol of grammar
③ Uses shift reduce/LR parser	③ recursive descent / LL parser
④ No backtracking.	④ backtracking (unless predictive).
⑤ Can handle LR(k) grammars	⑤ works only with LL(k) gram
⑥ more powerful & complex	⑥ simple.
⑦ often implemented automated, using tools	⑦ ~~Complex~~ easier to implemt manually.

Bottom up better (LR parsering).

1. Handles larger class of grammar, includes left recursive and ambiguous.
2. Powerful & efficient, prog lang with complex syntax.
3. Tools like automate LR parser, making it practical
4. Support complex grammar
5. Avoids backtracking.

9) Algorithm to construct predictive parsing table:

Input : Grammar G.

Output: Parsing Table M.

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

(1) For each terminal in FIRST(α), add $A \to \alpha$ to M[A, a].

(2) If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A, b]. If ϵ is in FIRST(α) and is in FOLLOW(A), add $A \to \alpha$ to M[A, $] as well.

Conditions: A grammar G is LL(1) if and only if whenever $w \to \alpha | \beta$ are two distinct productions of G.

(1) For no terminal w do both α and β derive strings beginning with.

(2) At most one of α and β can derive the empty string.

(3) If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \overset{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

Challenges:

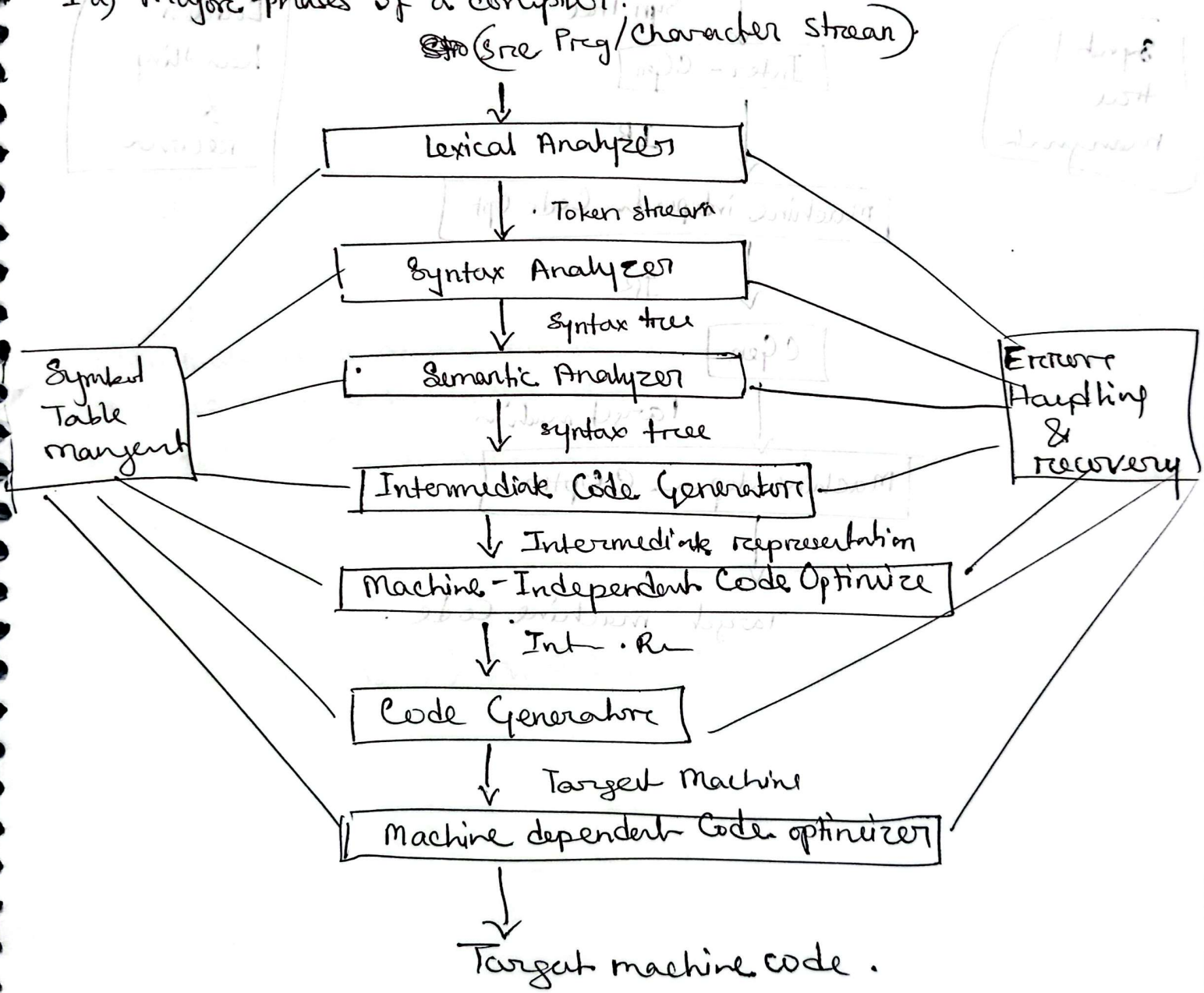
A predictive parser table M[A, a] is a 2D array where "A" is non-terminal and "a" is terminal or symbol $, the input endmarker.

Idea is: production $A \to \alpha$ is chosen if the next input symbol a is in FIRST(α). The only complication occurs

when $\alpha \overset{*}{\Rightarrow} \varepsilon$, In this case, we should again choose $A \to \alpha$. if the current input symbol is in FOLLOW(A), or if the $ on the input has been reached and $ is in FOLLOW(A).

Spring 23 -

1 a) Major phases of a compiler.



b) DFA :- Formal definition : (a quintuple (5-tuple) that is a system with 5 elements ⇝ $A = (Q, \Sigma, \delta, q_0, F)$ where

Q : finite non-empty set of states

Σ : " " " " " input-symbols/alphabets.

δ : transition f^n that takes as arguments a state and an input symbol and returns state, $\delta : Q \times \Sigma \to Q$.

q_0 : initial/start symbol, $q_0 \in Q$;

F : set of final or accepting states, $F \subseteq Q$.

How it helps to build compiler:

* used in lexical analysis phase to tokenize src code.
* reads input char stream & grps them into tokens (kw, id, num, symbol etc),

Example:

a DFA that accepts all and only strings of 0's and 1's that have sequence 01 somewhere in the strg.

So, $L = \{w \mid w$ is of form $x01y$ for some strg. x and y consisting 0's, 1's$\}$.

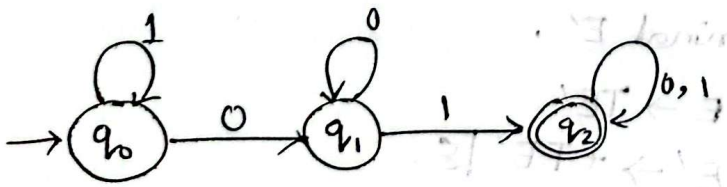
$\Sigma = \{0, 1\}$.

$Q = \{q_0, q_1, q_2\}$

$F = \{q_2\}$

$\delta = \{((q_0, 0), q_1), ((q_0, 1), q_0), ((q_1, 0), q_1), ((q_1, 1), q_2), ((q_2, 0), q_2), ((q_2, 1), q_2)\}$.

$q_0 = q_0$.



TT :

	0	1
→ q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_2\}$
* q_2	$\{q_2\}$	$\{q_2\}$

⑨ Elimination of Left recursion from CFG:

Input: G with no cycles or ε-productions
Output: Equivalent grammar with LR.

Method:

1. Arrange the nonterminals in order: $A_1, A_2 \ldots A_n$.
2. For each i from 1 to n
   - (a) For each j from 1 to i-1
     - replace each production of the form $A_i \to A_j \gamma$ with productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current prod for A_j.
   - (b) Eliminate any immediate LR among A_i

Example: $A \to A\alpha \mid b$, given with LR recursive

Transformed Formula:

$A \to bA'$

$A' \to \alpha A' \mid \varepsilon$

Suppose: $E \to E+T \mid T$

Here, ① $E \to E+T$ is LR.

So, $\alpha = +T$

and nonLR is $E \to T$.

So, $\beta = T$.

② a) new nonterminal E'.

b) rewritten $E \to TE'$

$E' \to +TE' \mid \varepsilon$.

d)

Ambiguous	vs	Unambiguous
① at least one string has multiple parse tree.		① every string has one parse tree.
② allows more than 1 parse tree for same string.		② only 1 parse tree for each.
③ Multiple Leftmost / rightmost derivation possible.		③ one LM / RM derivation
④ Not good for Deterministic parser		④ Good for LL(1), LR(1) and other deterministic parsers.
⑤ ambiguity in interpretation		⑤ consistent interpretation
⑥ E → E+E		⑥ E*E
⑦ needs grammar rewriting		⑦ no need

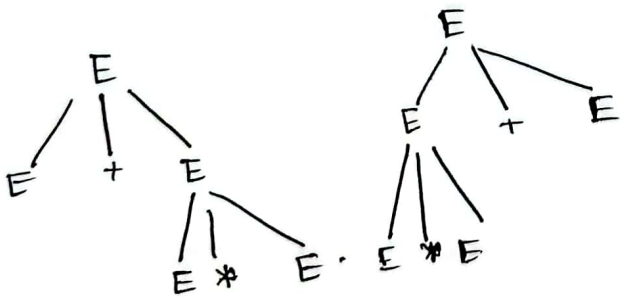
Example:

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

or, $E \to E+E*E$

so, $E \to (E+E)*E$

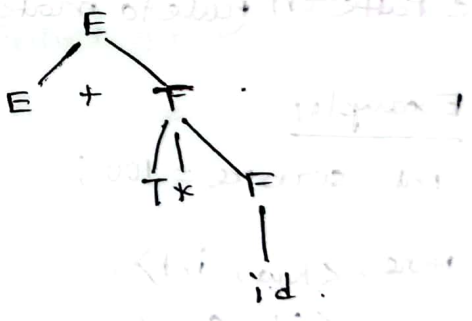
$E \to E+(E*E)$



$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$



e) **Lexical Analysis phase:** 1st phase of a compiler – also called scanning. works like a filter. Reads stream of characters of src code, groups them into lexeme and converts them into a stream of token passed to syntax analyzer (parser).

Main task:

1. Read char from src code
2. Identify lexemes (meaningful seq)
3. Produce tokens (<id name> or <number, 5>)
4. skips comments, whitespace,
5. Report errors / invalid lexemes

Terms:

1. Token (pair consisting name, attribute value).
   lexical unit: <token name, attribute value> <id, x>
2. lexeme (actual seq of characters). kw, op, id, const etc
3. Pattern (rule to match lexemes).

Example:

```
int score = 100;
```

Here, <kw, int>.
<id, score>
<op, =>
<number, 100>
<seperator, ;>

f) Errore recovery:

① Panic Mode:

* successive chars from input are ignored one at a time until a designated set of synchroniving tokens is found, sync tokens are delimiters such as ; or }

* Adj: easy to implement + no infinite loop.
* Dis: considerable amt of input skipped w/o checking for additional errors

② Statement Mode:

* performs necessary correction on remaing input so the rest of in. statement allow the parser to parse ahead.
* Corrections: (1) deletion of extra semicolons.
(2) replace comma by ";"
(3) inserting missing ";"
* needs extra care so that don't go into infinite loop.
* Dis: difficult to find situations with actual error ~~occur~~ before point of detection.

③ Error productions:

* Errores can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
common Errores known by user - that can be encountered.
* Adj: parsing appropriate errore msg can be generated and parsing can be continued.
* Dis: Difficult to maintain.

④ Global Correction:

* parser examines the whole program, tries to find out the closest match for it which is error free.
* Adj: closest match prog has smaller no. of insertions, deletions and changes of tokens to recover from erroneous input.
* Dis: High time + space complexity → not implemented practically.

⑨ Shift-reduce: In bottom up parsing, (LR parsing), shift reduce conflict occurs when parser cannot decid whether to shift or reduce at a certain point in the input

Why arise?
when grammer allows a produn rule to be reduced and another prod rule to be shifted for the same token, leading to ambiguity in parsing process.

~~Example~~:

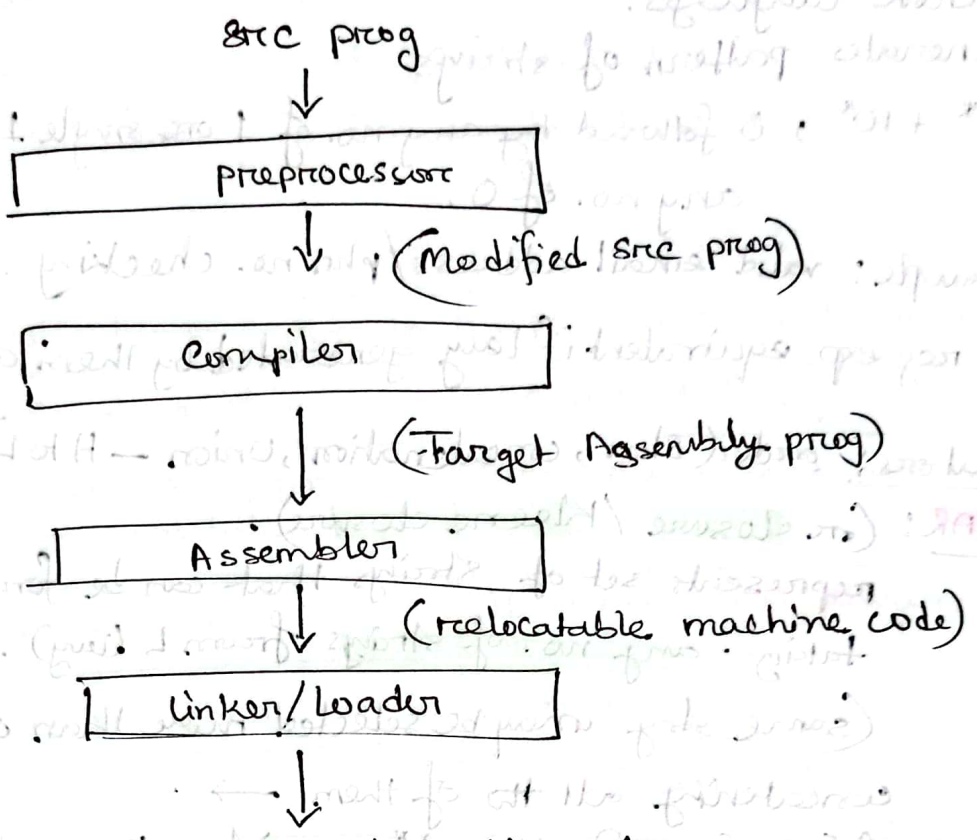
② Reduce reduce conflict: parser cannot decide btw 2 different reduction rules for the same SEQUENCE of inputs. Arises when grammer allows 2 or more prod rules to apply to the same seq. of inputs, leading to multiple possible interpretations of input.

Example:
Shift reduce: suppose E+E*E, conflicts arises when parser need to decide whether to shift * and continue or to reduce + operation

RR: S→A, A→a, S→B, B→a | A, B both matches a. Parser doesn't know who to reduce.

Fall-22

(1) a) Language processing system:



src prog – (divided into modules stored in seperate files)
preprocess করে modified version compiler-এ যায়।
compiler assembly lang-এ prog output দেয় (easy to debug by machine) → assembler-কে দেয় ← produces relocatable machine codee.
large prog – compiled in pieces — needs to be linked together with other relocatable obj files and library files.
linker resolves external memory addresses, when code in one file may refer to a location in another file, loader then puts together all executable obj files into memory to execute

b) **Regular expression:**

* represents certain sets of strs in an algebraic form
* denote languages.
* generates pattern of strings.
* $01^* + 10^*$, 0 followed by any no. of 1 or single 1 followed by any no. of 0.
* Example: valid email address/phn no. checking.
* 2 reg exp equivalent if lang generated by them are same.

RE Operators: order(Star, concatenation, Union — H to L).

① STAR: (or closure / Kleene closure).
represents set of strings that can be formed by taking any no. of strings from L (lang).
(same str may be selected more than once)
concatenating all ths of them ⟶.
If L={0, 11} then L^* consists of 0's and 1's.
that 1's come in pairs : 011 , 01111 , 11110.
but never 01011 or 101.

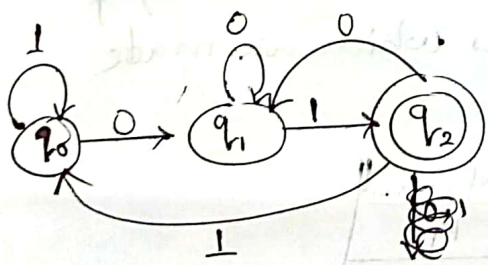
② Concatenation: lang can be formed concatenating L and/or str from L with any strg of M.
L={001, 10, 111} . M={ε, 001}.
Then LM = {001, 10, 111, 001001, 10001, 111001}.

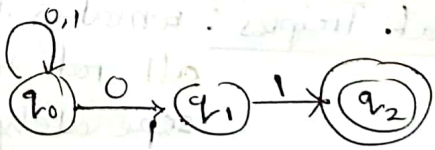
③ UNION: union of L and M.
LUM denoted, either L or M, or both.
L={001, 10, 111} . M={ε, 001}.
LUM = {ε, 10, 001, 111}.

② Finite Automaton: mathematical model of computation used to recognize pattern within ~~str~~ input strgs composed of symbols from a finite alphabet.

Consists — finite set of states, set of input symbols, transition f^n, initial state, final state set.

DFA (V8)	NFA
① Deterministic Finite Automaton	① Nondeterministic finite Automaton.
② Cannot handle empty strg.	② can handle ε.
③ single state transition for each char representation.	③ multiple possible paths.
④ Backtracking allowed.	④ Backtrack not allowed
⑤ challenging construction	⑤ simple
⑥ Requires more space	⑥ less than DFA.
⑦ input strng processing takes less time (fast	⑦ more (slow)
⑧ $\delta: Q \times \Sigma \to Q$	⑧ $\delta : Q \times \Sigma \to 2^Q$ εNFA $\to Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$
⑨ used in real compilers	⑨ converted to DFA first.
⑩ Strng ends with 01	⑩ Exple;





$Q = \{q_0\ q_1\ q_2\}$

$\Sigma = 0, 1$

$q_0 = q_0$ $\quad F = q_2$

④ 3 representations of 3 address code:

① Quadruples: 4 field namely op, arg1, arg2 and result.

example: $a = b * -c + b * -e$.

$t1 = \text{uminus } c$

$t2 = b * t1$

$t3 = \text{uminus } e$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$.

	op	arg1	arg2	result
1	uminus	c		t1
2	*	b	t1	t2
3	uminus	e		t3
4	*	b	t3	t4
5	+	t2	t4	t5
6	=	t5		a

② Triples: doesn't use extra temporary variable to represent a single operation instead when a ref to another triple's value is needed, pointer to triple is used.

3 field op, arg1 and arg2

	op	arg1	arg2
0	uminus	c	
1	*	b	(0)
2	uminus	e	
3	*	b	(2)
4	+	1	3
5	=	a	4

③ Indirect Triples: makes use of pointers to the listing of all ref to computations which are made seperately and stored.

	op	arg1	arg2
14	uminus	c	
15	*	b	14
16	uminus	e	
17	*	b	16
18	+	15	17
19	=	a	t8

	statement
0	14
1	15
2	16
3	17
4	18
5	19

c) To get a Normal form of CFG (Chomsky Normal form or Greibach Normal form) — the preliminary simplifications are:

(1) Eliminate useless symbols (variables - terminals that do not appear in any derivation of terminal string from the start symbol)

(2) " ε-productions, those of the form $A \to \varepsilon$. for some var A.

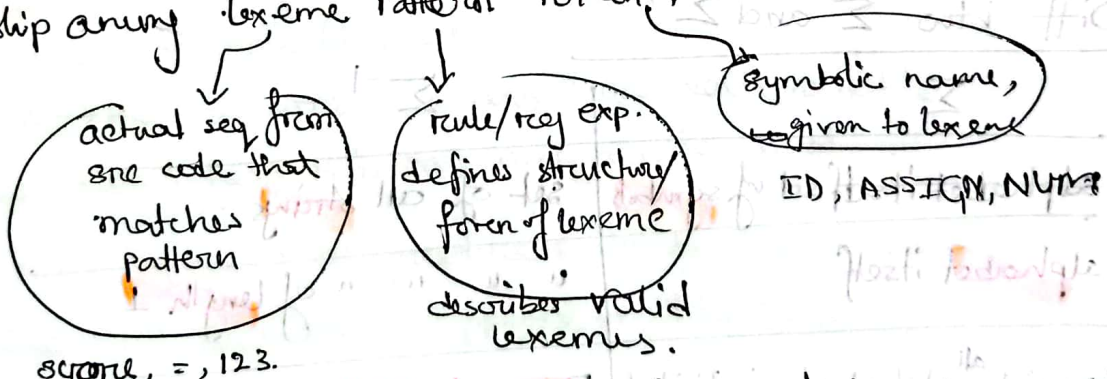
(3) " unit productions, $A \to B$.

d) Syntax Directed Translation : Grammars with embedded semantic actions — specify how to translate strings of the Lang into other forms, such as intermediate code or machine code.

Attribute grammar : formal way to define attributes of the prod of a grammar, specifying how to compute the values of attributes based on the structure of the parse tree.

Spring-22

(1a) Relationship among lexeme Pattern Token :



lexeme is identified by lexical analyzer — (stream of char to read from src code), that matches lexeme pattern)

generates token for score = 123.;

<ID, score> <ASSIGN> <num, 123> <sep, ;>

<op, =>

(2) a) power of Alphabet :-

If Σ is an alphabet, the set of all strings of a certain length from that alphabet can be expressed by using exponential notation ↴

Σ^k : set of strings of length k, and each of whose symbol is in Σ

Σ^0 : ε ; the only string of length 0 is ε.

Example :

$\Sigma = \{a, b, c\}$.

$\Sigma^1 = \{a, b, c\}$

$\Sigma^2 = \{aa\ ab\ ac\ ba\ bb\ bc\ ca\ cb\ cc\}$

$\Sigma^3 = \{aaa\ aab\ aba\ abc\ \text{———}\ ccc\}$.

set of All possible ~~string~~ lengths ~~one~~ of all possible ~~to~~ strings over an alphabet Σ is conventionally denoted by Σ^* (Kleene star).

Example : $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \text{----}\}$.

set of non empty strings, $\Sigma \longrightarrow \Sigma^+$ (Kleene Closure/Plus).

Diff btw Σ and Σ^{-1}

Σ	Σ^{-1}
~~alphabet itself~~ set of symbols	set of all strings.
alphabet itself	" " " " of length 1
consists all alphabets individually from Σ^{-1}	Belongs to Σ.
$\Sigma = \{a\ b\}$.	$\Sigma^1 = \{"a", "b"\}$
Letters	Words of length 1

③ a)

Syntax Analyzer	vs	Semantic Analyzer
① checks grammatical structure		① checks code meaningful and logical
② Focuses on structure		② meaning
③ takes tokens from lex A. as input		③ Takes parse tree as input from syn.A.
④ output : parse tree.		④ out : annotated / symbol tree.
⑤ checks balanced parenthesis, etc.		⑤ checks undeclared variables etc.
⑥ Errors example: if () { ↓ syntax error		⑥ assigning string to an int
⑦ Based on CFG.		⑦ Lang semantics + type rules.
⑧ int x = ; → error		⑧ int x = "hello";
⑨ structurally valid code		⑨ logically meaningful code

④ a)

Conditions for ~~CF~~ CFG to be in CNF :

① No ε productions, unit productions, useless symbols.

② All prod. has 2 simple forms :

1. $A \to BC$, where A, B and C are each variable.
2. $A \to a$, where A is var and a is terminal.

③ $S \to \varepsilon$, start symbol may produce ε only if ε is part of language and S doesn't appear on RHS of any production.

(5) a) Components of CFG:

Quadruple (4 tuple) — 4 elements: $G = (V, T, P, S)$

V - finite nonempty set of vars or non terminal symbols
Each var represents a lang.

T - finite nonempty set of terminal symbols, $V \cap T = \phi$

P - " " " " productions or grammar rules of form,

$A \rightarrow \alpha$, where
[head] $A \in V$
[Body] $\alpha \in (V \cup T)^*$

[prod symbol] "$\rightarrow$" means 'could take the value'
'can be replaced with';

S - One of the variables represents the lang being defined,
it is called the start symbol, $S \in V$.

Example:

G: $S \rightarrow Abb$
$A \rightarrow \varepsilon$
$A \rightarrow aA$
$A \rightarrow bA$

$S \rightarrow Abb$
$A \rightarrow \varepsilon \mid aA \mid bA$

Importance of CFG in compilers:

1. provides formal way to define the grammatical structure.
2. Parser uses CFG to check whether src code is valid.
building parse/syntax tree.
3. CFG helps tools to automatically generate parsers
4. Detects syntax errors / provide helpful feedback,
attempts error recovery using CFG.

(6) a)

~~CFL drive from CF~~

Left factoring :- grammar transformation technique used to remove common prefixes from productions.

Helps make a grammar suitable for predictive parsing (LL(1)) where parser uses one lookahead symbol to decide which production to use.

Example:

$A \to \cdot ES$
$A \to ES \mid S$; | Both starts with S symbol.

left factoring:

$A \to ESA'$
$A' \to S \mid \varepsilon$. | lookahead after S is A'.

Role of lookahead Symbol :-

1. Next input token parser sees.
2. LL(1) uses it to select the correct production rule w/o backtrack
3. Works efficiently only if the grammar is left factored.

Non left recursive. —— making it deterministic

Top down doesn't work on some CFG:

(1) Left recursion : falls into infinite recurn.

$A \to A\alpha \mid \beta$

solution $A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$.

(2) Ambiguity : a lot of parse tree.

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

(3) Common prefixes (needs LF)

multiple prod with same prefix.

$A \to$ if E then S

$A \to$ if E then S else S.

solution.

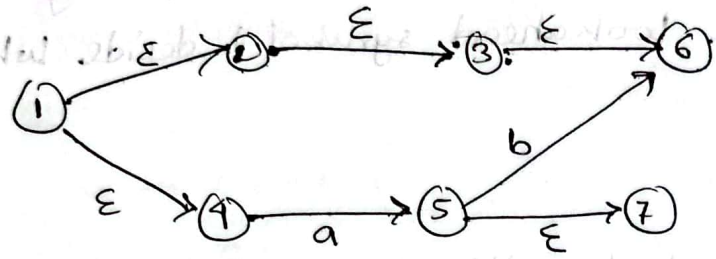
$A \to$ if E then SA'

$A' \to$ ~~if S~~ else $S \mid \varepsilon$.

⑥ b) ~~Espi~~ Epsilon closure of an ENFA:

$A = (Q, \Sigma, \delta, q_0, F)$.

all the states that can be reached from a particular state only by seeing the ε symbol.



EClose (3) = {3, 6}.
EClose (2) = {2, 3, 6}.
EClose (1) = {1, 2, 3, 4, 6}.

3 fields + next state + ~~FCOS~~ Eclose of next state if it has ε in transition

---

⑧ Relationship Formal Lang – ~~Compiler~~ Compilers:

① FL mathematical foundation for specifying syntax of prog lang.
② recognize valid program structures using FL
③ Translates src code to intermediate / machine code using FL

④ LA of computer uses regular languages (recognized by DFA)
⑤ ~~SP~~ Syn A " " CFG (handled by CFGs and PDAs).

⑧ Importance of DFA – NFA

DFA	NFA
① used in LA	① used in LA
② Faster + easier to implement	② easier to construct for RE.

used to convert RExp to build a LA that can recognize kw, ID, num etc.

(*) Finite Automaton	(vs)	Pushdown Automaton
① No memory beyond current state		① Has stack of memory
② Accepts Reg B Languages		② CFL
③ less powerful		③ more power — can match nested pattern
④ LA		④ Syntax Analysis (parsing)
⑤ Example pattern ID, num, OP		⑤ matching parantheis, nested if-else.

(*) · Importat of DFA minimization :

~~① Efficiency less.~~

① Efficient
② less memory.
③ computes faster (in LA).
④ smaller DFA — easy to understad + implement
⑤ Remove redundancy — eliminates duplicate / equivalent states — keeps essential ones
⑥ Unique · up to renaming states.
⑦ Helps comparing automata and proving correctness.