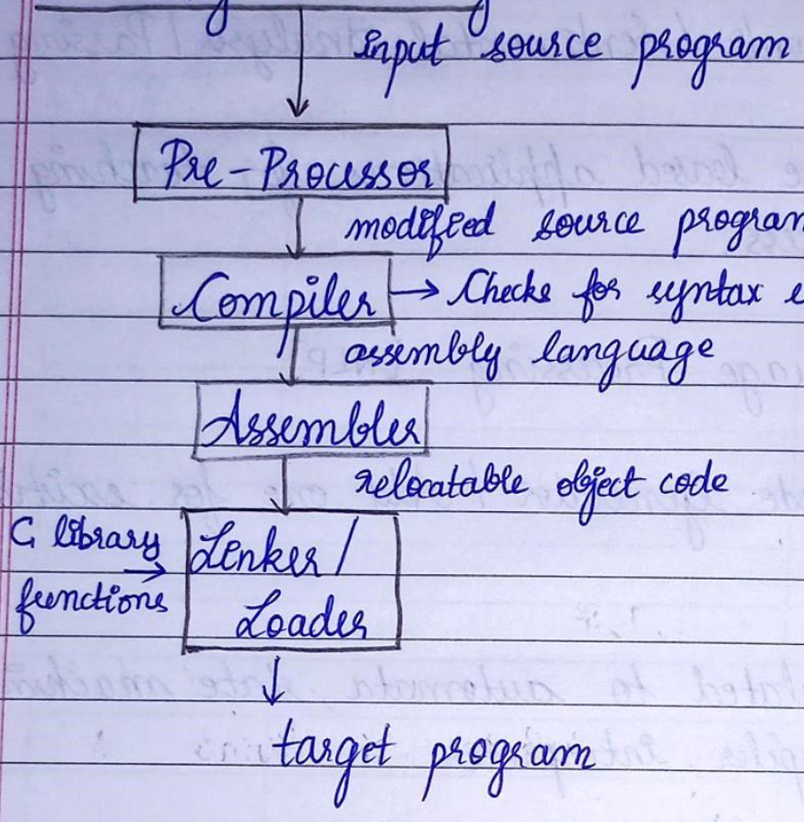
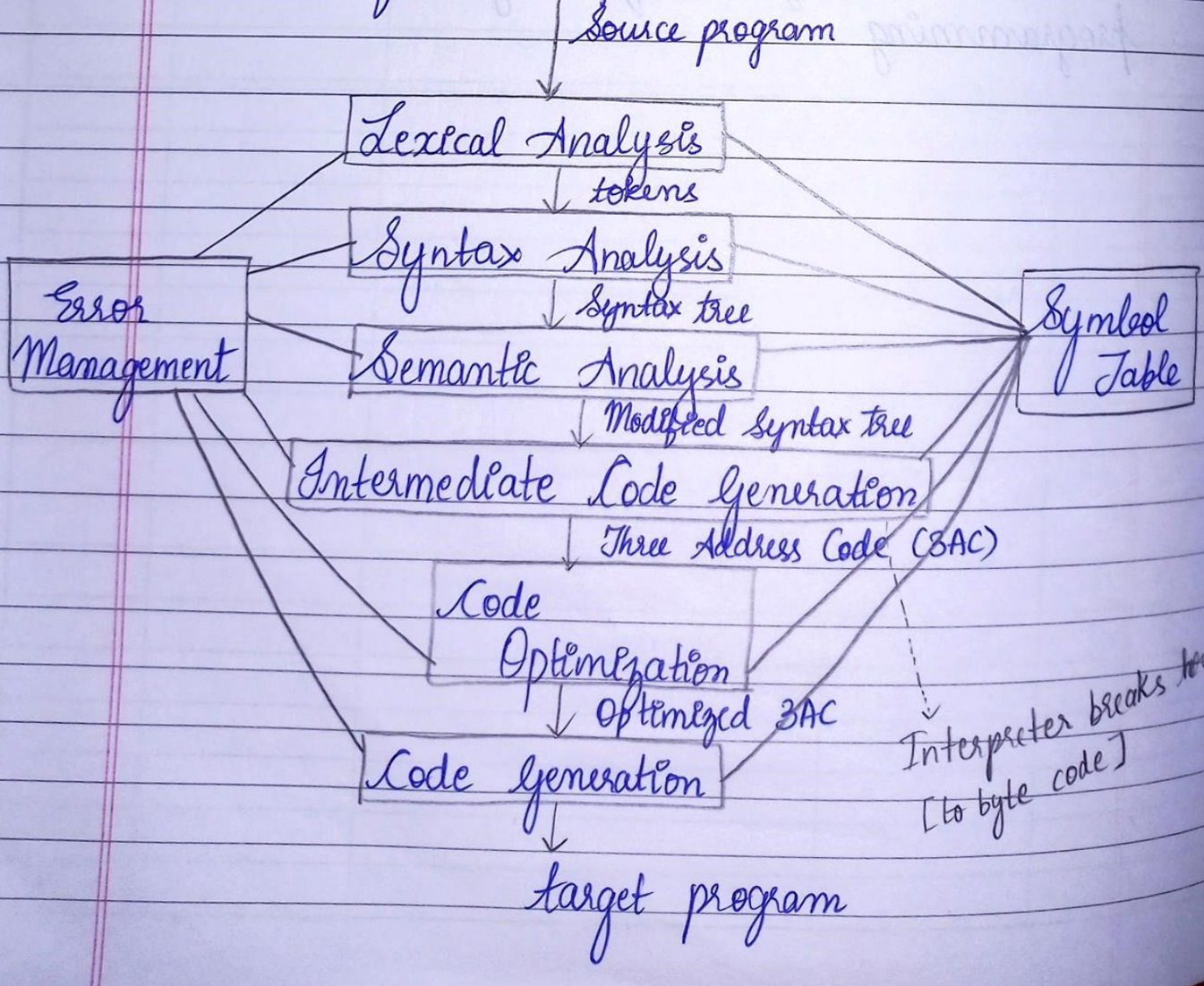
9.10.2023

# Unit-I

## * Learning Processing




## * Structure of a Compiler



* Grammars
↳ checking 'C' expressions for arithmetic operations

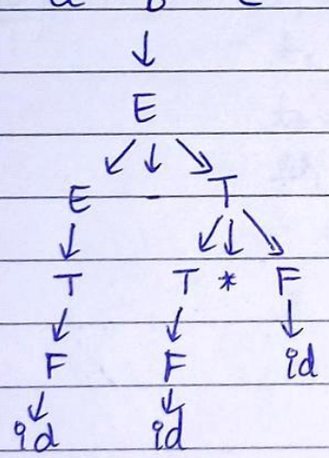
* Types :-

1) Context Free Grammar
2) Context Sensitive Grammars

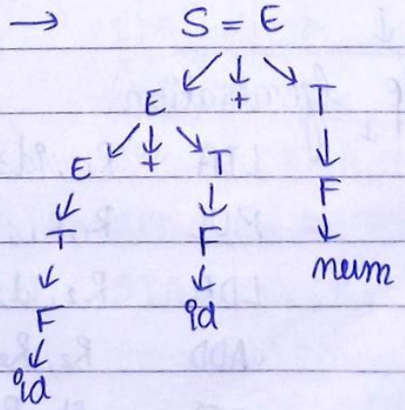
(Statement) $S = E \mid id$
(Expression) $E \rightarrow E+T \mid E-T \mid T$
(Term) $T \rightarrow T * F \mid T/F \mid T\%F \mid F$
(Factor) $F \rightarrow (E) \mid id \mid num$

Non-terminals :- E, T, F
Terminals / Tokens :- (, ), id, num, +, -, *, /, %
* Tokens - has its own definition in Symbol table (user can't define)

Eg: a - b * c




a = b + c + 10 →



* ab = b + c * 10

↓

Lexical Analysis

↓

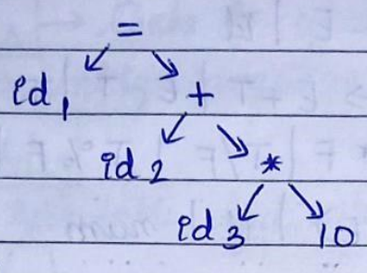
<id,1> <=> <id,2> <+> <id,3> <*> <10>

↓

Syntax Analysis

↓

Syntax tree:




↓

Semantic Analysis

↓

Same syntax tree

↓

Intermediate Code Generation

↓

Three Address Code (3AC)

$t_1 = id_3 * 10$

$t_2 = id_2 + t_1$

$id_1 = t_2$

↓

Code Optimization

↓

$t_1 = id_3 * 10$

$id_1 = id_2 + t_1$

↓

Code Generation

↓

Assembly language code

```
LDA   R1, id3
MUL   R1, R1, #10
LDA   R2, id2
ADD   R2, R2, R1
STA   id1, R2
```

* Translate HLL a = b + c + 10 to Assembly language [a, b, c float]

a = b + c + 10

↓

Lexical Analysis

↓

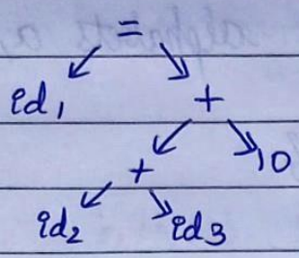
<id,1> <=> <id,2> <+> <id,3> <+> <10>

↓

Syntax Analysis

↓

Syntax tree:

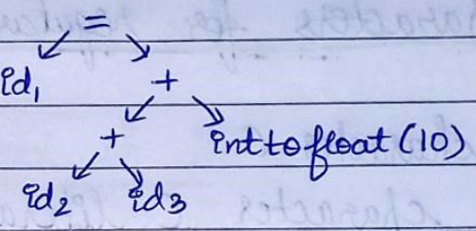



↓

Semantic Analysis

↓

Modified Syntax tree:




↓

Intermediate Code Generation

↓

3AC:

$t_1 = id_2 + id_3$

$t_2$ = int to float (10)

$t_3 = t_1 + t_2$

$id_1 = t_3$

↓

Code Optimization

↓

$t_1 = id_2 + id_3$

$id_1 = t_1 + 10.0$

↓

Code Generation

↓

Assembly language code

```
LDA  R1, id2
LDA  R2, id3
ADD  R2, R2, R1
ADD  R3, R2, #10.0
STA  id1, R3
```

* Precedence : Closure > Concat. > Union
* Operator : \+, Char : +

## * Regular Expression

Union, Concatenation, Closure ⇒ Positive (+) - atleast 1 occurence of ch
(L|D) (LD) ↘ Kleene (*) - occurence can be null

Language - A set of finite symbols consisting of alphabets

Eg: L is a set of alphabets a,b,c

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

B is binary language $B = \{0,1\}$

## * Lex characters for regular expressions

C - character C
\c - character C literaly
"s" - string s
. - any character other than newline
^ - beginning of line
$ - end of line
[s] - any one of the character in s
[^s] - any characters not s
r* - 0 or more occurence
r+ - 1 or mor occurence
r? - 0/1 occurence
r{m,n} - b/n m and n occurences
$r_1 r_2$ - $r_1$ followed by $r_2$
$r_1 | r_2$ - $r_1$ or $r_2$
(r) - same as r
$r_1 / r_2$ - $r_1$ when followed by $r_2$
[a-zA-Z] - character class

* Let $L = \{a, b\}$

1) $\underbrace{(a \cup b)(a \cup b)}_{c}$

$= \{aa, ab, ba, bb\}$

2) $a^*$

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

3) $(a|b)^*$

$= \{\varepsilon_0, a, aa, aaa, \ldots, b, bb, bbb. \}$

4) $a|a^*b$

$= \{a, b, ab, aab, aaab \ldots\}$

* Represent the statement in RE

1) All strings of lowercase letters that contain five vowels in order.

conso $\rightarrow [b\text{-}df\text{-}hj\text{-}np\text{-}tv\text{-}z]$

Ans: str $\rightarrow$ {conso}* a {conso}* e {conso}* i {conso}* o {conso}* u {conso}*

2) Any string of $\Sigma = \{a, b\}$ where it starts with 'a' and ends with 'a'

$= a(a^*|b^*)^* a$ or $a(a^*b^* | b^*a^*)^* a$

~~$a a^* b^*$~~

$a(a|b)^* a$

(S|s)(

17.10.2023

* 1) Write regular expression for SQL keywords:-

SELECT (Case insensitive) Eg: SeLEct, SelccT etc

* 2) Comments consisting of string surrounded by /* and */.

* 3) Describe the language description for the following R.Es

a) a*b a*b a*b a* = $\varepsilon_0, a, aa, aaa \ldots b\varepsilon_0, a, aa \ldots b\varepsilon_0, a, aa\, b\varepsilon_0, a, aa$

b) (a|b)* a (a|b)(a|b)

c) ^[^aeiou]*$

Answers

1) SELECT

(S|s)(E|e)(L|l)(E|e)(C|c)(T|t)
[Ss][Ee][Ll][Ee][Cc][Tt]

2)

"/*"[(^("/*"|"*/"))(.|"\n")]* "*/"

3)

a) a*b a*b a*b a*
A language L={a,b} with string of any length containing 3 b's in order

b) (a|b)* a (a|b)(a|b)

A string with characters a & b such that last 3rd character is always a

Write RE to identify arithmetic, log, relational op. in C

c) ^[^aeiou]*$

A line consisting of all characters except a, e, i, o, u.

* R.E to identify arithmetic, log. & relational op. in C.

[-+*/%=<>!&|^]|==|!=|<=|>=

16.10.2023

# Tutorial - 1

I. Write character class for the following set of characters.

1) The first 10 letters (upto j) in either uppercase or lowercase.

2) Lowercase consonants

3) Characters that appear at the end of English sentence (Eg: Exclamation point)

II. Write a regular expression that matches string "|

III. Translate the foll. assignment statement:-

Eg: $n = a*a + 2*a*b + b*b$
(Assume all variables are int)

## Answers

I

1) [a-jA-J]

2) [b-df-hj-np-tv-z]

3) [.?!;]

II

III

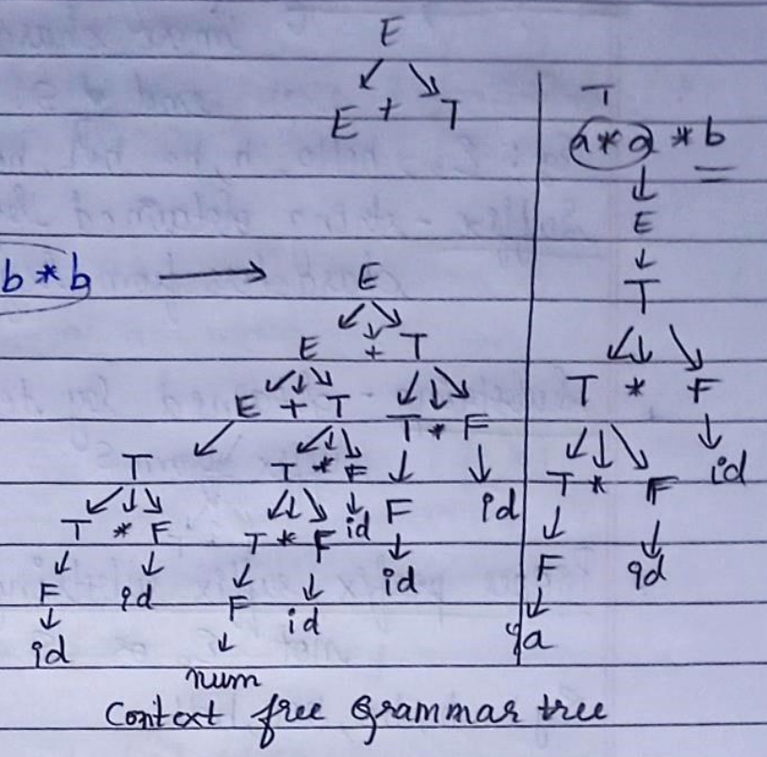
Eqn = a*a + 2a*b + b*b

↓

**Lexical Analysis**

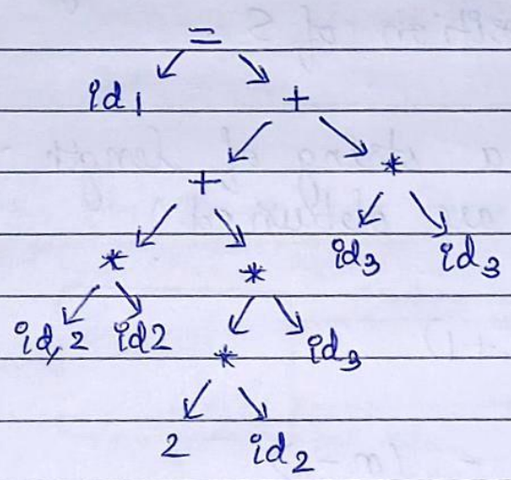
↓

<id,1><=><id,2><*><id,2><+>
<2><*><id,2><*>
<id,3><+><id,3><*>
<id,3>




↓

**Semantic Analysis** → ALL:




↓

**Intermediate Code Generation**

↓

$t_1 = id_2 * id_2$
$t_2 = 2 * id_2$
$t_3 = t_2 * id_3$
$t_4 = t_1 + t_3$
$t_5 = id_3 * id_3$
$t_6 = t_4 + t_5$
$id_1 = t_6$

→ **Code Optimization**

↓

$t_1 = id_2 * id_2$
$t_2 = 2 * id_2$
$t_3 = t_2 * id_3$ →
$t_4 = t_1 + t_3$
$t_5 = id_3 * id_3$
$id_1 = t_4 + t_5$

* Prefix of string - string obtained by removing 0 or more characters (symbols) from the end of S.

Eg: $\varepsilon$, hello, h, he, hel, hell

* Suffix - string obtained by removing 0 or more characters from beginning of S.

* Substring - obtained by deleting any prefix and any suffix from S

* Proper prefix, suffix substring - not $\varepsilon$ or S

Eg: h, he, hel, hell

* Subsequence - any S formed by deleting 0 or more not necessarily consecutive position of S.

---

Q: * If there is a string of length "n", How many of the foll. are obtained?

a) Prefix - $(n+1)$

b) Proper Prefix - $(n-1)$

c) Substring - $\frac{n(n+1)}{2} + 1$

$\downarrow$ $3\left(\frac{4}{2}\right) + 1 = 7$

3 -> a, b, c, ab, bc, ac, abc

* != or <>

* Digits → [0-9]+

Num → digits (\. digits)? (E[+-]? digits)?  } Regular Definition
(floating point value)

↳ Regular Expression

letter → [A-Za-z_]

id → letter (letter | digits)*  } Regular Definition

↳ R.E for identifier

relational operators :- relop → < | <= | <> | > | >= | =

arithmetic operators :- arithop → [+ \- * / % =]

(\- → "-" characters literally)

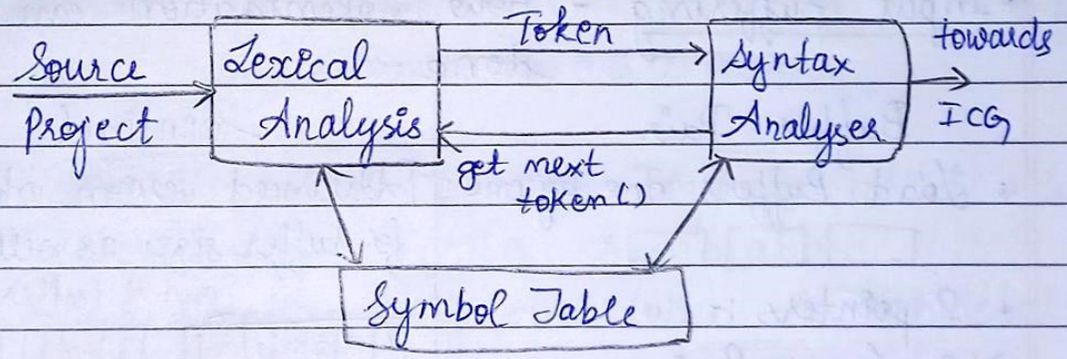
or

+ | - | * | / | %

## * Finite Automata (Deterministic)

* Transition Diagram

## * Interaction b/n Lexes and Passes




Tasks:

1) Tokenization
2) Stripping comments, tab spaces and white spaces.
3) Keep tracking of line nos.
4) Correlating error messages.

RE → pattern

RE → name ] Token

* Terms:

1) Pattern - Description of the form that lexemes of a token may take.

2) Lexeme - Sequence of character in a source program that matches pattern for a token.

3) Token - pair consisting of token names and attribute values.

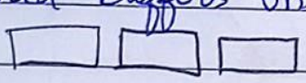
* Lexical Analyser Error Recovery Strategy can do the foll: (Lexical Error Handler)
  1) Delete any extra character
  2) Insert any one character
  3) Swap adjacent characters

```
if(x==0)
{
x=1;
}
```

* Input Buffering - How Tokenization can be done

Buffer Pair

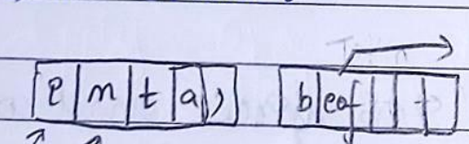
* Load Buffers one by one

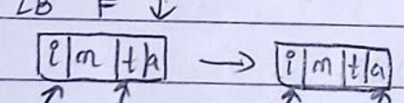


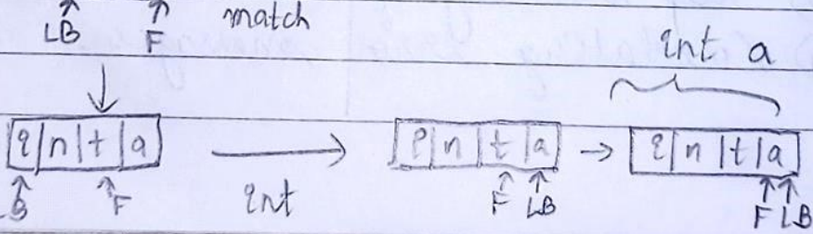
* 2 pointers :-
1) LB - Lexeme Begin
2) F - Forward

Sentinel

[Advanced version which checks for buffer size as well]

token name  attr. value

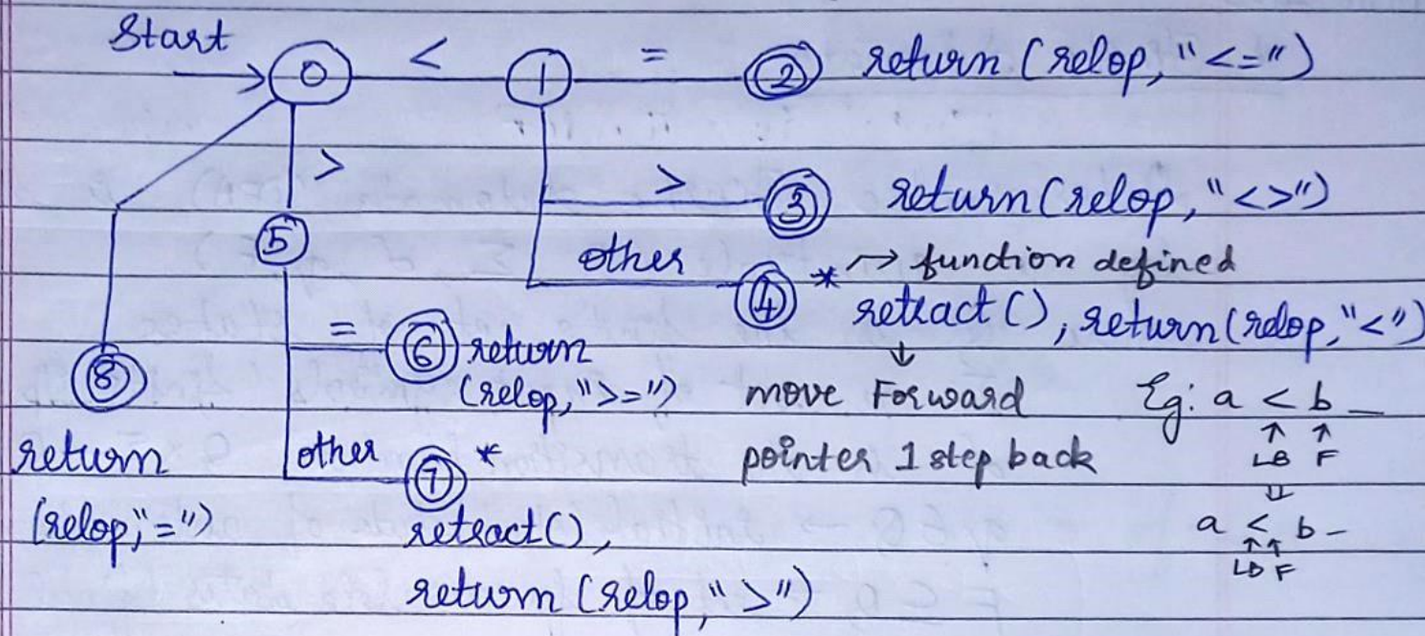
* Defined on R.E { float → "float"
  int → "int"
  id → l(l|d)*
  d → [0-9]+

* Some scripting languages → panic-mode recovery (errors in all lines after an error in particular line)

---

* relop → <= | <> | < | > | >= | =

Start state, Accepting state




This is called Transition Diagram which is Deterministic Finite Automata.
[can identify only one token at each state]

---

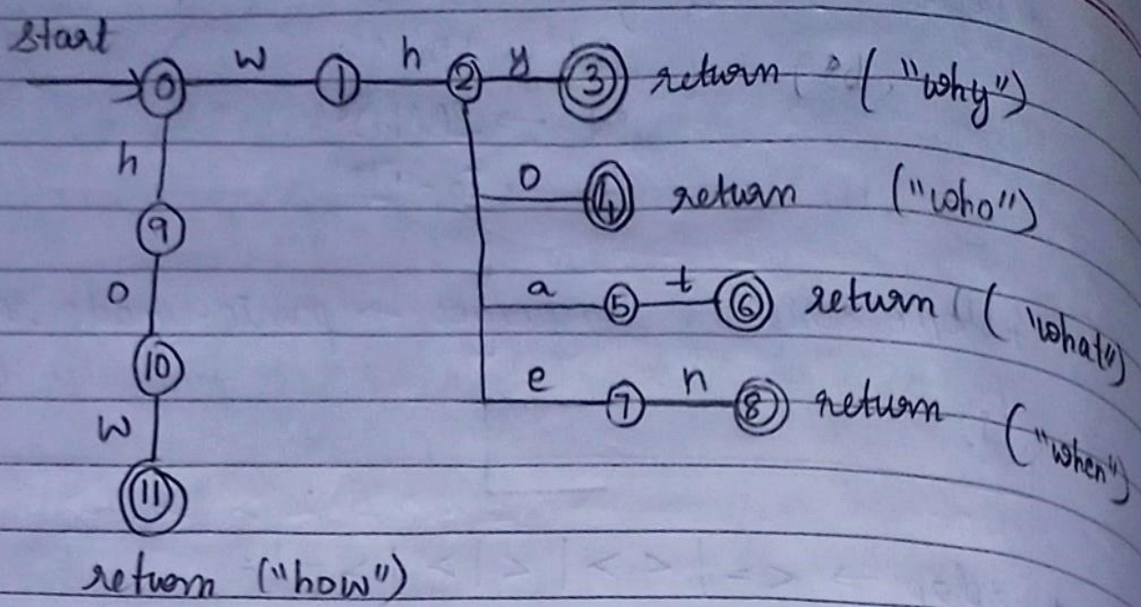
* Draw Transition diagram to accept why, who, what, when, how.

T.T.O →

Eg:

$\delta(q, a) \rightarrow q'$

state, input, new state




19.10.2023

## * Finite Automata

Deterministic Finite Automata (DFA) is denoted by a quintuple $(Q, \Sigma, \delta, q, F)$

where $Q$ is the finite set of states

$\Sigma$ is set of input symbols (finite i/p alphabet)

$\delta$ is the transition function $Q \times \Sigma \rightarrow Q$

$q \in Q$ → initial (start) state of automation

$F \subseteq Q$ → set of favourable states (final state)

* Deterministic means every move of a Finite Automata is completely determined by the input and current state

Eg: R.E = $(a|b)^*$

Apply DFA on RE,

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

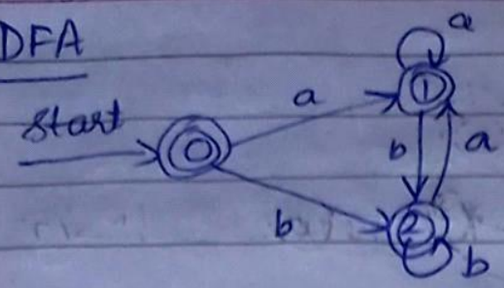
$(a|b)^* = \{\varepsilon, a, b, aa\ldots, bb\ldots, ab\ldots, ba\ldots\}$

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

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

$q = q_0$

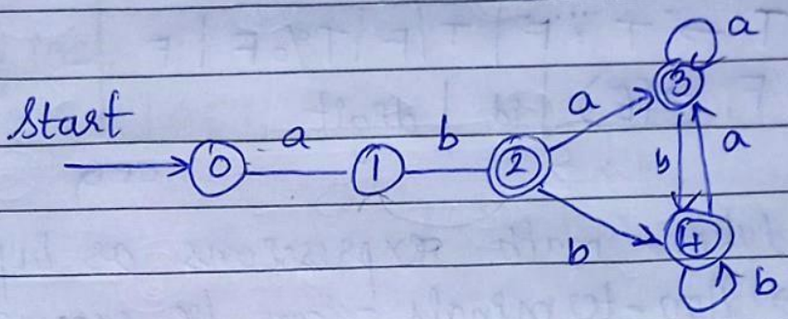
DFA




$\delta(q_0, a) = q_1$
$\delta(q_0, b) = q_2$
$\delta(q_1, a) = q_1$
$\delta(q_1, b) = q_2$
$\delta(q_2, a) = q_1$
$\delta(q_2, b) = q_2$

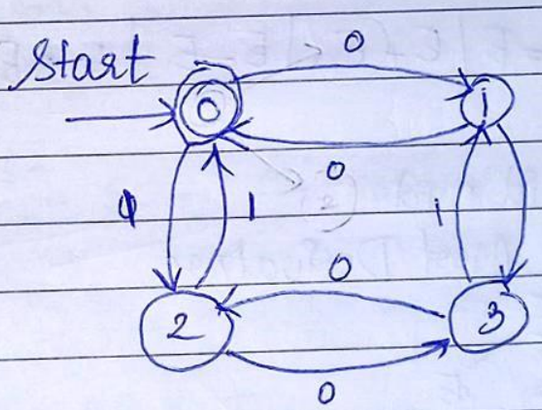
* RE = ab(a|b)* = ab, aba, abb

DFA



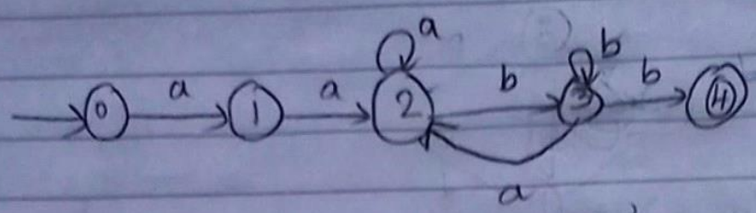

* RE Construct DFA to accept even no. of 0s and 1s.

DFA




0011
0110
1111

* RE = aa (a|b)* bb
1 2 3 4 → Min




26.10.2023

## * Context Free Grammars

$G = (V, T, P, S)$ → Start symbol
- V → non-terminals → can be expanded further : E, T, F
- T → terminals - tokens : +, -, *, id
- P → set of grammar symbol - production

| | |
|---|---|
| | S → id = E \| E |
| Exp. | E → E+T \| E-T \| T |
| Term | T → T * F \| T/F \| T%F \| F |
| factor | F → (E) \| id \| digit |

→ 1 production set

CFG

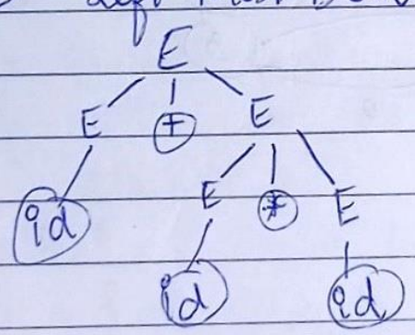
* It takes math expressions as input.
* V → Non-terminals - can be expanded further : E, T, F
* T → terminals - tokens : +, -, *, id

* Gramanas :

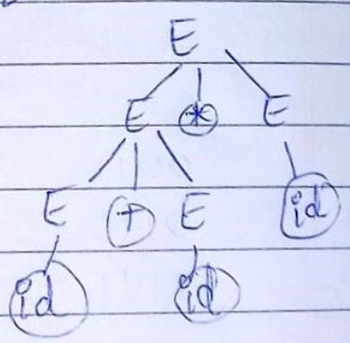
E → id = E | E+E | E-E | E*E | E/E | E%E | (E) | id | digit

Q: id + id * id

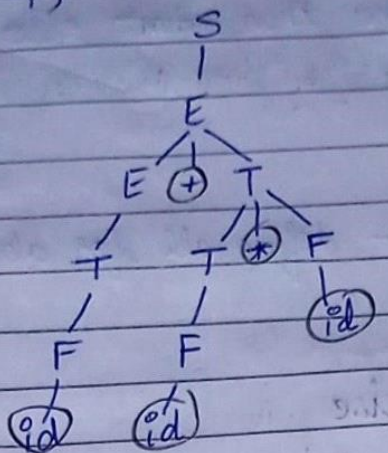
LMD - Left Most Derivative




LMD



* String :- id + id * id
Using CFG,




* Design a ∈ CFG to accept strings ...
0011, 000111, 00001111...

$S \rightarrow 00E11$
$E \rightarrow \varepsilon_0 \mid 0E1$

* Consider CFG $S \rightarrow ss+ \mid ss* \mid a$ for string aa+aa*+
a) Give LMD
b) Give RMD
c) Draw Parse tree
d) Is Grammar ambiguous? Justify

a) LMD

$S \xrightarrow{lm} ss+ \xrightarrow{lm} ss+s+ \xrightarrow{lm} as+s+ \xrightarrow{lm} aa+s+ \xrightarrow{lm} aa+ss*+ \xrightarrow{lm} aa+as*+ \xrightarrow{lm} aa+aa*+$

* Given CFG:

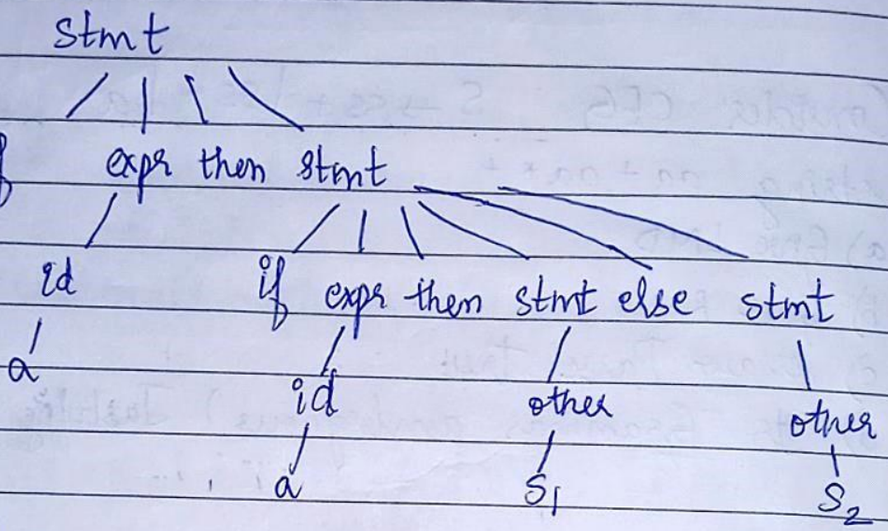
P:
- stmt → if expr then stmt
- | if expr then stmt else stmt
- | other
- expr → id

a) $G_1 = V, T, P, S$ – define all for above CFG

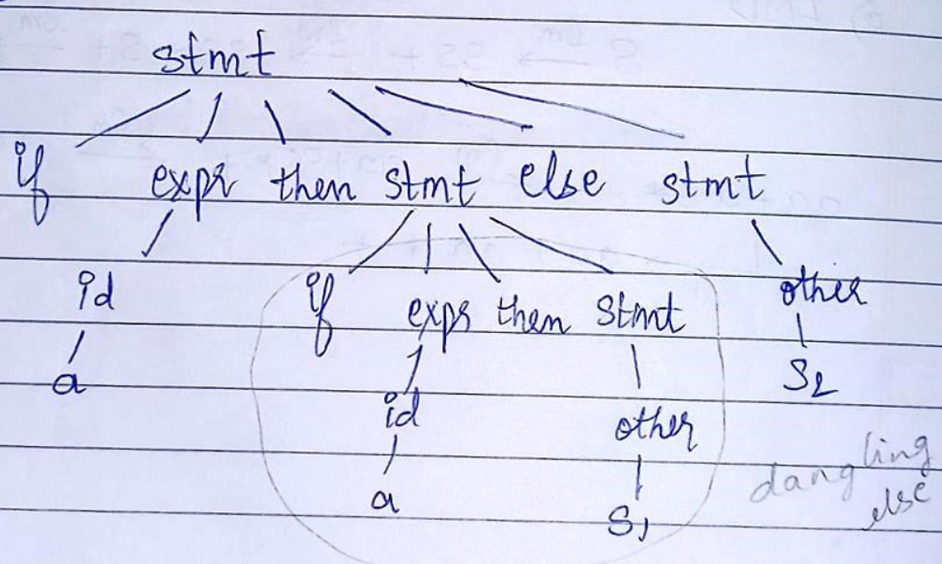
V = stmt, expr
T = id, if, then, else
P = whole CFG
S = stmt

b) i/p :- if a then if a then $S_1$ else $S_2$
↳ dangling else

LMD - 1




LMD - 2



* TDP, BUP
* Non-Recursive → Predictive

30.10.2023

## * Removing ambiguity from dangling else grammar:

stmt → if expr then stmt
| if expr then stmt else stmt
| other
expr → id

Can be converted to stmt → matched_stmt
⇒ | open stmt
matched-stmt → if expr then matched_stmt else matched-stmt
| other
open_stmt → if expr then stmt
| if expr then matched_stmt else open_stmt

expr → id

## * Eliminating ambiguity for TDP - Top Down Parser:

* S → SS* | SS+ | a

* Not suitable for NR-PP (TDP): A → Aα | β
A → αβ1 | αβ2
as backtracking is not allowed

* For TDP,
If left recursion is there ⇒ S → Ss* | Ss+ | a (LM), then eliminate LR
* If left factoring is there ⇒ S → 0S1 | 01 ⇒ do LF
A → αβ1 | αβ2
S → 0S1 | 01

⇓

A → αA' }
A' → β1 | β2 }
S → 0S'
S' → S1 | 1

* Eliminate LR (Left recursion)

$E \to E+T \mid E-T \mid T$ — ①
$T \to T*F \mid T/F \mid T\%F \mid F$ — ②
$F \to (E) \mid id \mid num$ — ③

eliminate LR ⇒

$E \to TE'$
$E' \to +TE' \mid -TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid /FT' \mid \%FT' \mid \varepsilon$
$F \to (E) \mid id \mid num$

To check for LR, see if the below format exists,

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

$E \to E+T \mid E-T \mid T$ — ①

$A \to A\alpha \mid \beta \Rightarrow \begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{cases}$

eliminate LR

Similarly for ①,

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

Illy for ②,

$T \to FT'$
$T' \to *FT' \mid /FT' \mid \%FT' \mid \varepsilon$

③ ⇒ $F \to (E) \mid id \mid num$

* Left Recursion

Direct LR
$A \to A\alpha \mid \beta$

Indirect LR
$S \to Aa \mid b$
$A \to Sa \mid a$

* Input :- Grammar G
o/p :- G without LR
Method :- Apply algorithm to G
Arrange the non-terminals in some order $A_1, A_2$
for (each i from 1 to n){
for (each j from 1 to i-1) {
replace each production of the form
$A_i \to A_j \gamma$ by the production
bring it to this format → $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \delta_k\gamma$ where (substitute)
$A_j \to \delta_1 \mid \delta_2 \mid \ldots \delta_k$ $A_j$ production
}
eliminate immediate LR from $A_i$ production

$G : S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon_0$

iteration
$i = 2$
$j = 1$

$A_1 = S$ $\quad i=1 \to 2^n$ $\quad A_i \to A_j\gamma$
$A_2 = A$ $\quad j = 1 \to 0$ $\quad A_2 \to A_1\gamma$
$A \to S\gamma$

① $S \to Aa \mid b$
② $A \to bdA' \mid A'$
③ $A \to cA' \mid adA' \mid \varepsilon_0$

$A \to Ac \mid Aad \mid bd \mid \varepsilon_0$
$A\alpha_1$ $A\alpha_2$ $\beta_1$ $\beta_2$

* $G : S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon_0$

1st iteration : $i = 1$ $\quad S \to Aa \mid b$
$j = 1 \to 0$ $\quad A_j$

2nd : $i = 2$ $\quad S \to Aa \mid b$
$j = 1$ $\quad A \to Ac \mid Sd \mid \varepsilon_0$
$A_i$ $\quad \overrightarrow{A_j}$

$A \to Ac \mid Aad \mid bd \mid \varepsilon_0$
$A\alpha_1$ $A\alpha_2$ $\beta_1$ $\beta_2$

$\Downarrow$

$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \varepsilon_0$

* Check whether the G are suitable for top-down parsers & if not, make it suitable.

1) $S \to S+S \mid SS \mid (S) \mid S* \mid a$

2) $S \to (L) \mid a$
$L \to L,S \mid S$

1) $\overset{A}{S} \to \overset{A}{S}+S \mid \overset{A}{S}S \mid (S) \mid \overset{A}{S}* \mid a$
($A\alpha_1$, $A\alpha_2$, $\beta_1$, $A\alpha_3$, $\beta_2$)

$S \to aS' \mid (S)S'$
$S' \to +SS' \mid SS' \mid *S' \mid \varepsilon$

2) $S \to (L) \mid a$
$L \to L,S \mid S$

$S \to (L) \mid a$
$\overset{A}{L} \to \overset{A}{L},\overset{\alpha}{S} \mid (L) \mid a$
($\beta_1$, $\beta_2$)

$\Downarrow$

$L \to (L)L' \mid aL'$
$L' \to ,SL' \mid \varepsilon$