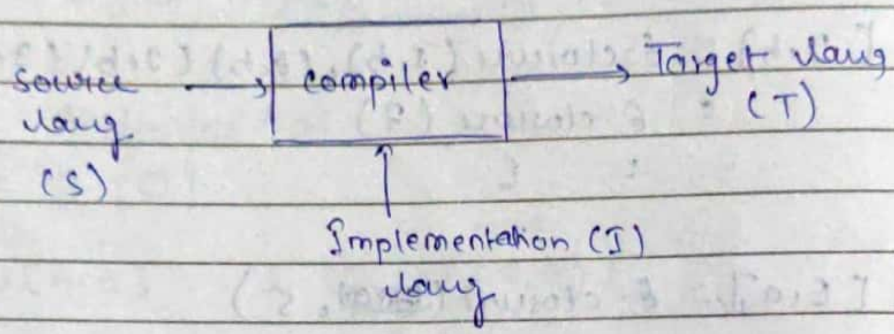
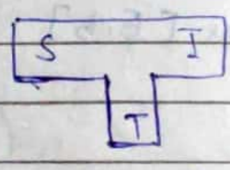
Date 18.02.19

Page No.:

Bootstrapping / Cross compiler :-



Notation:

C^{ST}_{I} or $S_I T$ or 

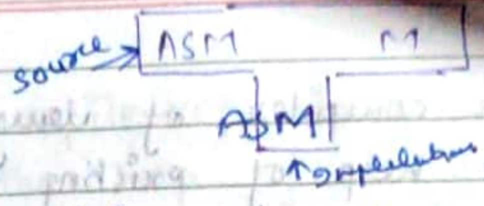
(T-Diagram)

Bootstrapping :-

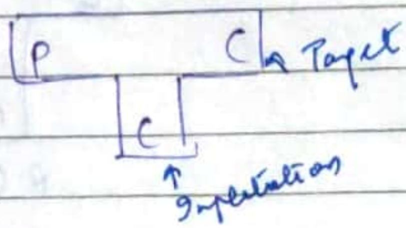
- The process by which simple language is used to translate more complicated program which in turn may handle even more complicated program and so on is known as Bootstrapping

- A compiler is characterised by three language
1) Source language (S)
2) Implementation language (I)
3) Target language (T)

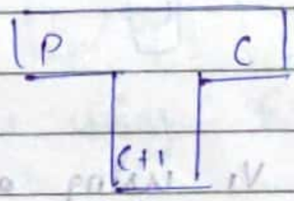
There are 3 types of compiler
1) self-host compiler - (source & implementation language are same)



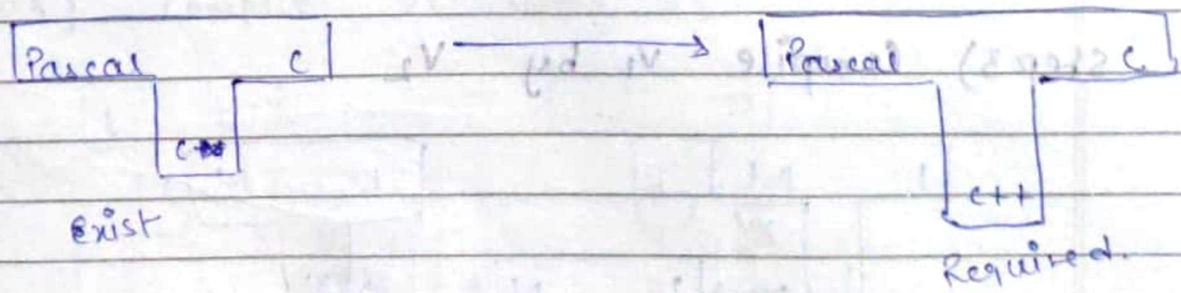
2) Native compiler (implementation language & target language are same)

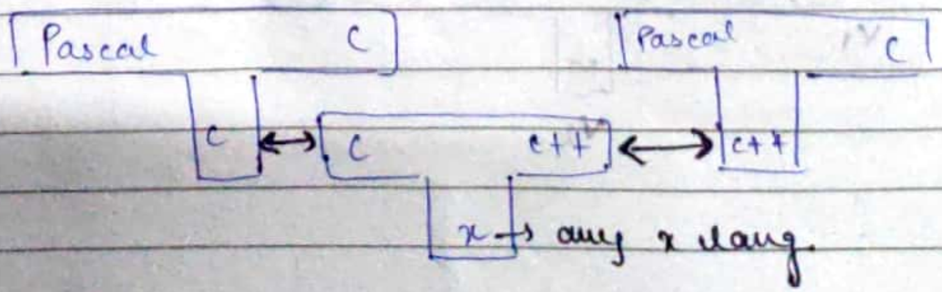


3) Cross-compiler :-
If the source, implementation & target language are different then such a compiler is called cross compiler

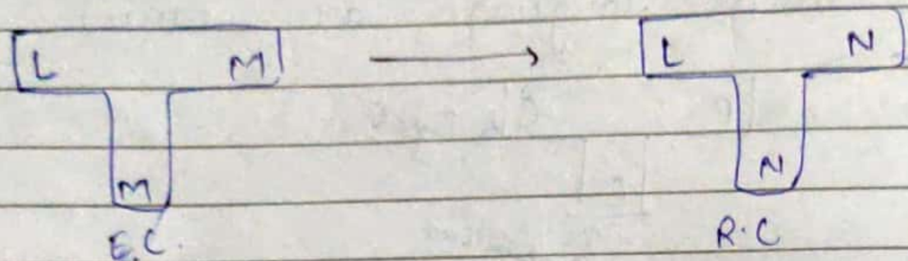


Q:- We have a pascal translator written in C language that take Pascal code as input and produce C code as output. Create a pascal translator in C++ for the same.

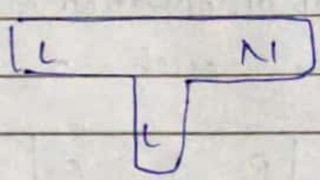




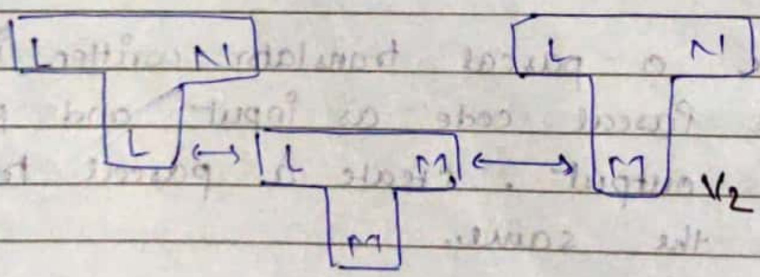
Q) Write the compiler of lang 'L' for m/c 'N' with the help of existing lang 'L' and m/c 'M' [Assume both compilers to be Native]



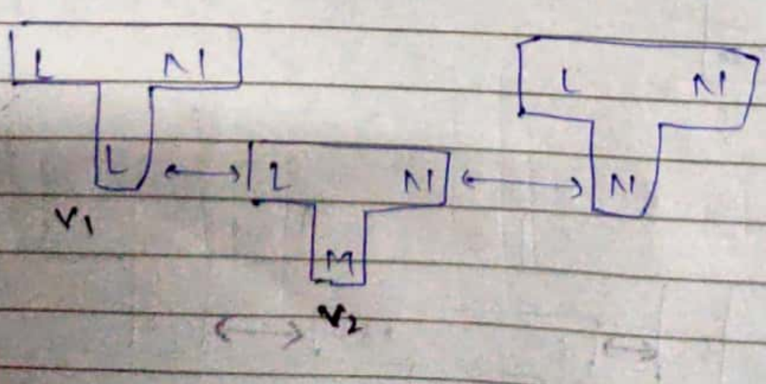
step 1) Create a self-host compiler V_1



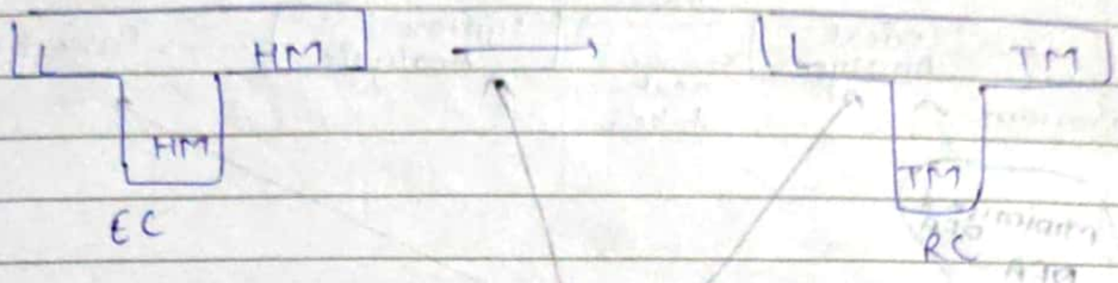
step 2) Compile V_1 using existing compiler



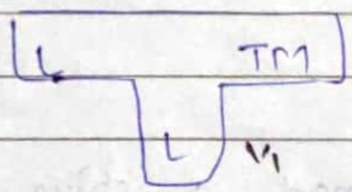
step 3) Compile V_1 by V_2



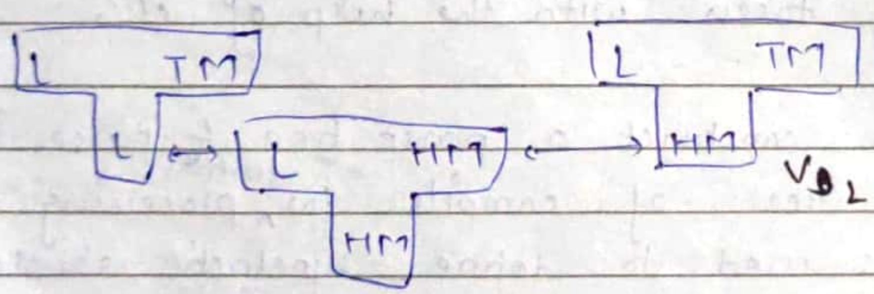
Q:- Write a compiler of language L for m/c TM with the help of existing compiler for language L & m/c HM.



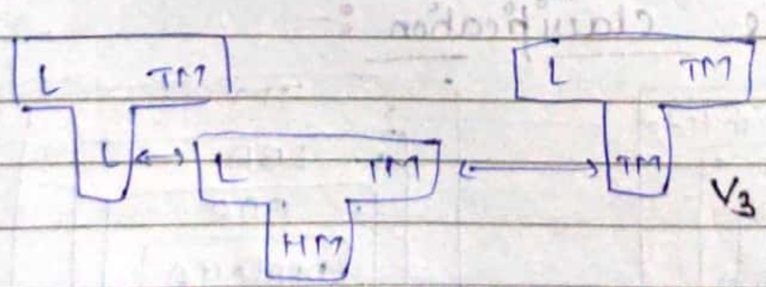
step1) Create a self host compiler V_1



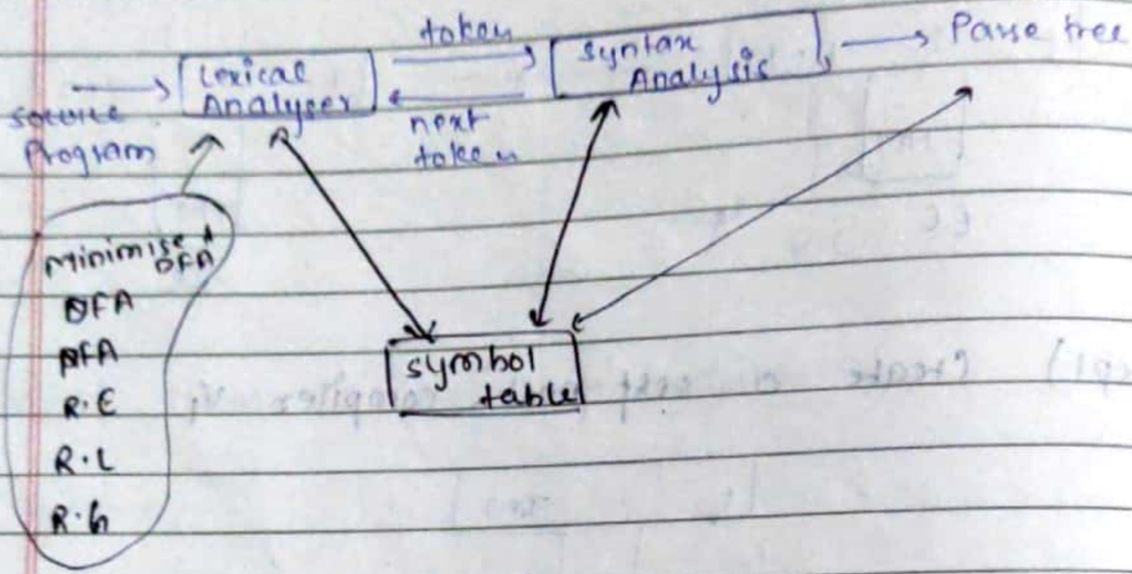
step 2) compile V_1 using E.C.



step 3) compile V_1 using V_2

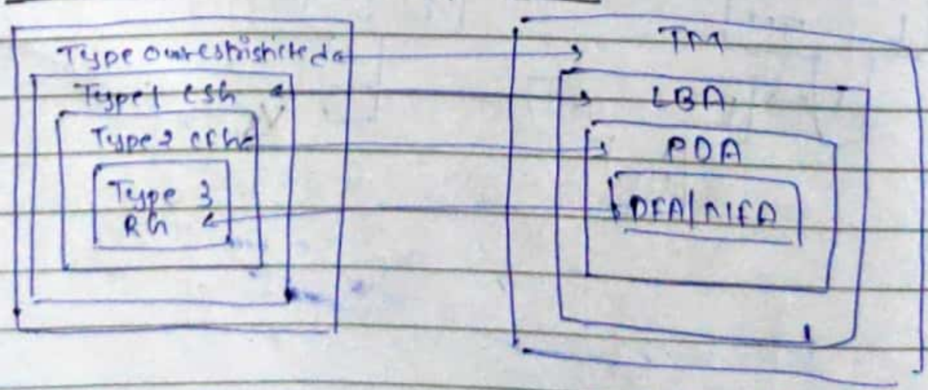


Syntax Analysis



- The parser obtained a string of token from the lexical analyser and verify that string of token with the help of CFG
- The parser construct a parse-tree & passes it to the rest of compiler for further processing
- A CFG is used to define syntactic structure of a programming language

CHOMSKY's Classification :-



A grammer G is defined as V, Σ, P, S

V → set of variable / Non-terminal
Σ → input variables
P → production
S → start symbol

- A grammer is said to be as regular, if it is right linear or left linear.

<u>Type 3</u>
$V \to VT$
$V \to TV$
$V \to T^*$

<u>Type 2</u>
$V \to (V \cup T)^*$

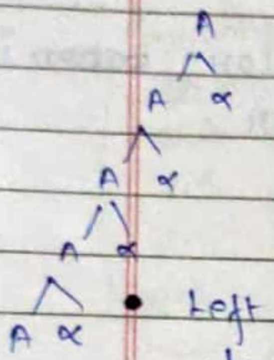
<u>Type 1</u>
$\alpha \to \beta$
$|\alpha| \leq |\beta|$
$\alpha, \beta \in (V \cup T)^+$

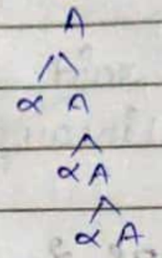
<u>Type 0</u>
$\alpha \to \beta$
$\alpha \in (V \cup T)^+$
$\beta \in (V \cup T)^*$

- Grammer is ambiguous or unambiguous
- If we construct more than one parse-tree of a given grammer then grammer is said to ambiguous.
- If we construct two lmd & rmd for a given string then the grammar is said to be ambiguous.
- If we are ~~not~~ able to construct single lmd & rmd & parse tree then grammar is said to be as unambiguous.

Grammar is said to be left recursive & right recursive.

Left recursive	Right recursive
$A \to A\alpha \mid B$	$A \to \alpha A \mid B$

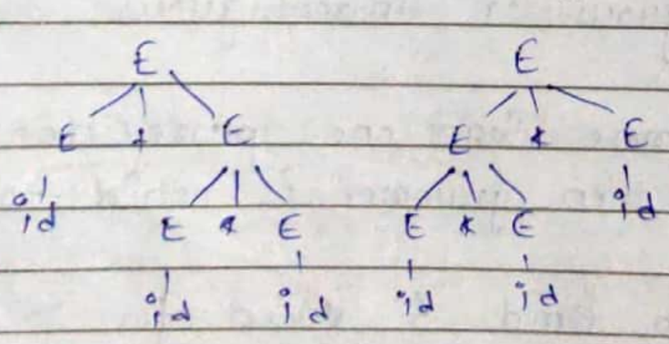




- Left recursive grammar may got to a infinite loop that's why we introduce the concept of left recursion.

Q) Consider a grammer $E \to E+E \mid E*E \mid id$ the grammar produces string of $id+id*id$

1) Construct a parse tree for a given string
2) Construct lmd & rmd for a given string
3) Find grammar is ambiguous or unambiguous.



∴ two parse tree are generated

∴ grammar is ambiguous.

Date __/__/__

LMD

$E \to E + E$	$E \to E * E$	~~$E \to E * E$~~
$\to id + E$	$\to E + E * E$	~~$E \to id * E$~~
$\to id + E * E$	$\to id + E * E$	$\to E$ ~~id~~ $+ E + E$
$\to id + id * E$	$\to id + id * E$	$\to$ ~~id~~ $E + id * E$
$\to id + id * id$	$\to id + id * E$	$\to$ ~~id~~ $id + id * id$

rmd

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

How to make unambiguous grammar

To make unambiguous grammar we must take care of

1) precedance
2) associativity

Precedance

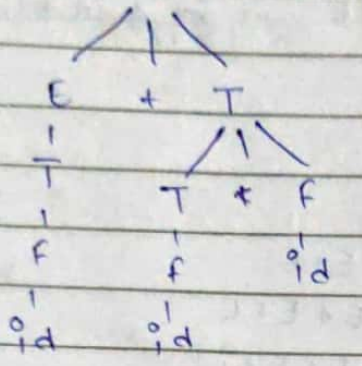
$\uparrow > * > +$

Associativity

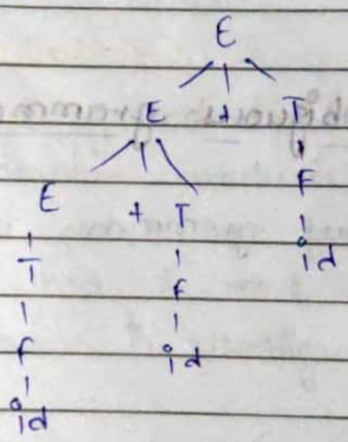
$+$	(L to R)	$(2 + 3 + 5)$
$*$	(L to R)	$(2 * 3 * 5)$
$\uparrow$	(R to L)	$(2 \uparrow 3 \uparrow 5$

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

$E \Rightarrow E*T$ id+id+id



id+id+id



L $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to x \uparrow F \mid x$ (right associativity)
$x \to id$

Date __/__/__

Q:- ~~[illegible]~~
Given grammar is unambiguous. find following operator associativity & precedance of

$, #, @

$A \to A\$B \mid B$
$B \to B\#C \mid C$
$C \to C@D \mid D$
$D \to d$

precedance $@ > \# > \$$
associativity
@ (L to R)
# (L to R)
$ (L to R)

Q:- find the associativity & precedance of the
$E \to E*E \mid F+E \mid F$
$F \to F-F \mid id$

precedance :- $- > +, *$
associativity :-
* (L to R)
+ (R to L)
- (L to R or R to L)

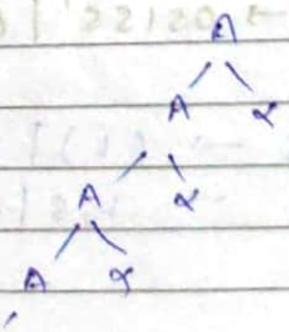
20/02/19

Left recursion :-

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

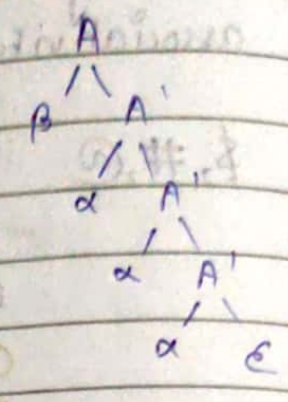
$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

string : $\beta, \beta\alpha, \beta\alpha\alpha, \ldots$



infinite loop.

Removal of left recursion



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

$\longrightarrow \quad A \to \beta A'$

$\quad\quad A' \to \alpha A' \mid \epsilon$

$\Rightarrow A : \{ \beta, \beta\alpha, \beta\alpha\alpha \ldots \}$

(Here pointer can easily read the next character.

Q: Remove left recursion from given grammar

$E \to E+T \mid T$

$T \to T*f \mid f$

$f \to id$

$\underset{A}{E} \to \underset{A}{E}\underset{\alpha}{+T} \mid \underset{\beta}{T}$ $\quad\quad$ $\underset{A}{T} \to \underset{A}{T}\underset{\alpha}{*f} \mid \underset{\beta}{F}$

$\Rightarrow E \to TE'$ $\quad\quad$ $\Rightarrow T \to FT'$

$E' \to +TE' \mid \epsilon$ $\quad\quad$ $T' \to *FT' \mid \epsilon$

$F \to id$

Q: $\underset{A}{S} \to \underset{A}{S}\underset{\alpha}{0S1S} \mid \underset{\beta}{01}$

$S \to 01S'$

$S' \to 0S1SS' \mid \epsilon$

Q: $S \to (L) \mid x$

$L \to L,S \mid S$

$\longrightarrow$ $S \to (L) \mid x$

$L \to SL'$

$L' \to ,SL' \mid \epsilon$

Date __/__/____

**

Q:- $S \to Aa|b$
$A \to Ac|Sd|\epsilon$

$\hookrightarrow A \to SdA'|\epsilon A'$
$A' \to cA'|\epsilon$

$S \to SdA'a|\epsilon A'a|b$
$\to S \to \epsilon A'aS'|bS'$
$S' \to dA'aS'|\epsilon$

Ans:- $\boxed{A \to SdA'|\epsilon A'}$ remove (of no use)

$$\boxed{\begin{array}{l} A' \to cA'|\epsilon \\ S \to \epsilon A'aS'|bS' \\ S' \to dA'aS'|\epsilon \end{array}}$$ Ans

Q:- $S \to Aa|b$
$A \to Ac|Aad|bd|\epsilon$

Ans:- $S \to Aa|b$
$A \to bdA'|\epsilon A'$
$A' \to cA'|adA'|\epsilon$

Q:- $A \to \overset{A\ \alpha}{ABd}|\overset{A\ \alpha}{Aa}|\overset{\beta}{a}$
$B \to Be|b$

$\alpha A|\alpha\beta_1|\alpha\beta$

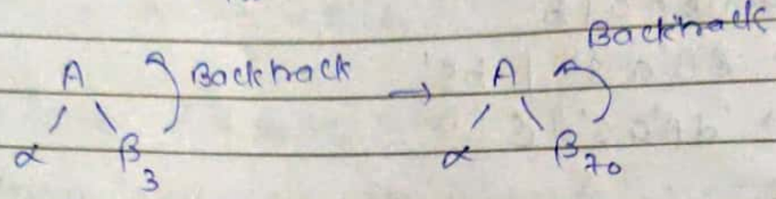
$A \to \alpha A'$
$A' \to \beta_1|\beta$

$A \to aA'$
$A' \to BdA'|aA'|\epsilon$
$B \to bB'$
$B' \to eB'|\epsilon$

Left factoring :-

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \cdots | \alpha\beta_n$

for $\alpha\beta_{100}$



Removal process

$$\boxed{\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 | \beta_2 | \cdots | \beta_n \end{aligned}}$$

If grammar having problem of ~~reg~~ backtracking then it is left factoring and for ~~for~~ removal of left factoring grammar should be of the above form.

Q. Consider a grammar & apply left factoring.

$S \rightarrow \underbrace{iets}_{\alpha,\ \beta_1 = \epsilon} | \underbrace{iets}_{\alpha}\underbrace{es}_{\beta_2} | a | m$

$S \rightarrow ietsS' | a | m$

$S' \rightarrow \epsilon | es$

Date __/__/__

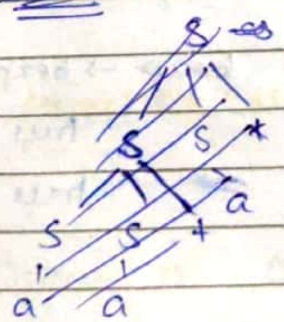
Q: $A \rightarrow aAB \mid aA \mid a$
$B \rightarrow bB \mid b$

$\boxed{A \rightarrow aAA' \mid a}$ left factoring
$A' \rightarrow B \mid \epsilon$
$B \rightarrow bB'$
$B' \rightarrow B \mid \epsilon$

$\Longrightarrow$ $A \rightarrow aA''$
$A'' \rightarrow AA' \mid \epsilon$
$A' \rightarrow B \mid \epsilon$
$B \rightarrow bB'$
$B' \rightarrow B \mid \epsilon$

Q:- Consider the CFG
$S \rightarrow SS+ \mid SS* \mid a$
and string $aa+a*$
1) Give lmd of string
2) Give a rmd of string
3) check grammar is ambiguous or unambiguous for a given string.

~~lmd~~ :



~~rmd~~ : S

∴ We have one lmd & rmd
∴ grammar is unambiguous

lmd: $S \rightarrow SS*$
$S \rightarrow SS+S*$
$aS+S*$
$aa+S*$
$aa+a*$

rmd: $S - SS*$
$S \rightarrow Sa*$
$\rightarrow S \rightarrow SS+a*$
$S \rightarrow SS+a*$
$S \rightarrow Sa+a*$
$S \rightarrow aa+a*$

"In order to succeed, your desire for success should be greater than your fear of failure." – Narendra Modi

Date ___ / ___ / ___ Page No.: ___

Q:- G = {{bexp}, {and, or, not, true, false}}

bexp → bexp or bexp | bexp and bexp |

bexp → true | false

1) whether grammar is ambiguous or not

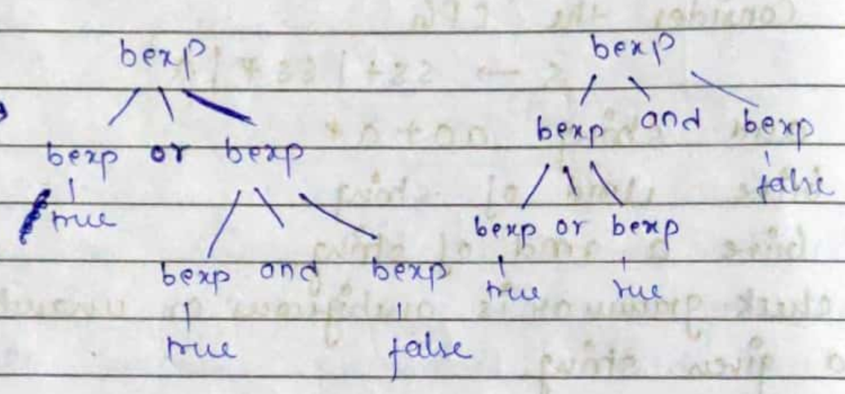
Construct lmd & rmd

If grammar is ambiguous then convert it into unambiguous

eg :- true or true and false

~~lmd~~

~~bexp~~ →



lmd

bexp → bexp and bexp

→ bexp or bexp and bexp

→ true or bexp and bexp

→ true or true and bexp

→ true or true and false

bexp → bexp or bexp

true or bexp

true or bexp an

rmd →

bexp → bexp ~~and~~ or bexp

→ bexp or bexp and bexp

→ bexp or bexp and false

→ bexp or true and false

→ true or true and false

Date __/__/____

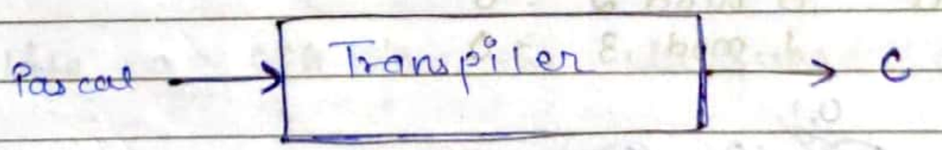
unambiguous grammar

bexp → bexp or bexp' | bexp'
bexp' → bexp' and bexp'' | bexp''
bexp'' → true | false.

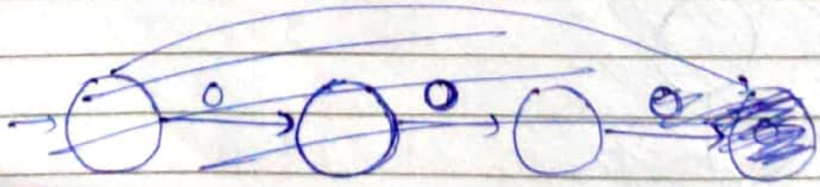
Trans-Compiler | Transpiler | Source-to-source compiler

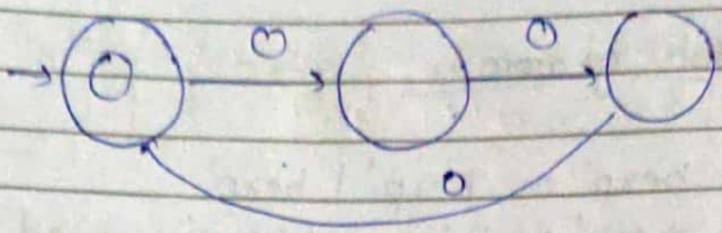
- A source-to-source compiler | transpiler is a type of compiler that takes the source code of a program written in one language as input and produces the equivalent source code in another language.
- A source-to-source compiler may perform a translation of a program at roughly the same level, like from Pascal to C, while a traditional compiler translate from a language like C to assembly or Java to byte code.

eg:- suppose Python 2 is converted to Python 3.



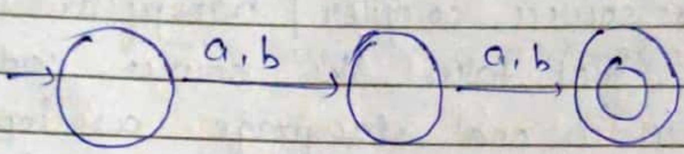
Q:- Draw a DFA that accepts string 0 mod 3 = 0



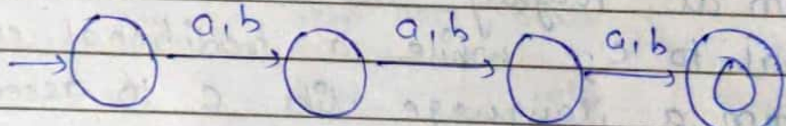


Q:- Draw DFA of a string consisting of a,b of length exactly 2.

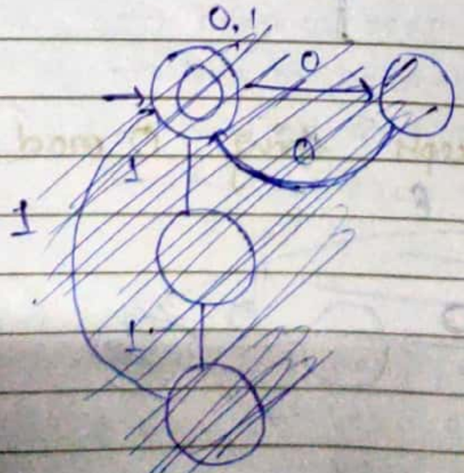
(a+b) (a+b)



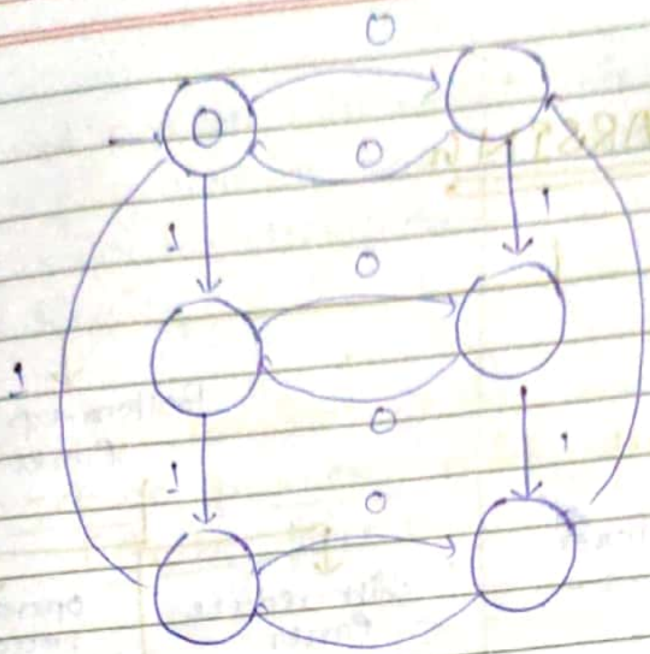
Q: Draw DFA that accept string of length exactly 3. (a+b)(a+b)(a+b)



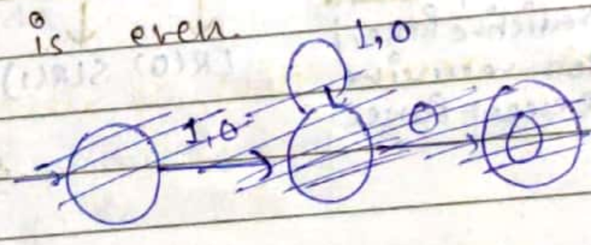
Q: Draw a DFA that accept string of 0 & 1 where 0 mod 2 = 0
1 mod 3 = 0

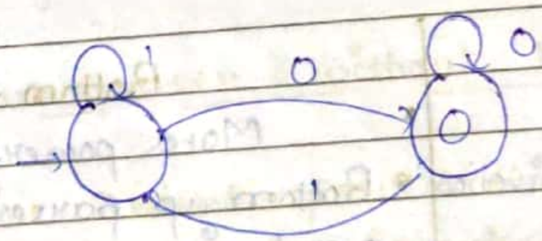


Date __/__/____



Q: Construct a DFA that accepts any binary no which is even.





Q:- Write a CFG for the language $a^n b^n$ $n \geq 1$

aabb

~~S → a aaabbb ab~~

$S \rightarrow aSb \mid ab$

Scanned with CamScanner