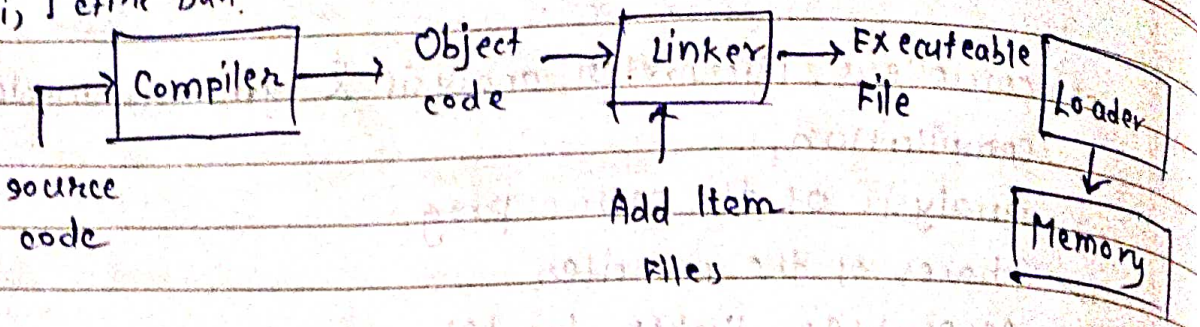
LT (Language Translation)

(Aoffrad Vaho) (B.S. Publication

- Ravi Sethi, Tetime Duh.



(1) Meanings (i) High Level Language (User Oriented Languages)
(ii) Assembly Language
(iii) Machine Level Language
(iv) Translators
(v) Linker
(vi) Loader

(2) Real time Applications of Language Translator.
→ Health-Care, Global Summit, International Affairs.

Ans.(i) → (i) High level Language :-

Hardware : Piece of Mechanical Device & its functionality are being controlled by Compatible software, & Electronics

Software :- Understand instruction in form of Electronic charge

↓

Software : Is a counter part of binary language in software
- Program.

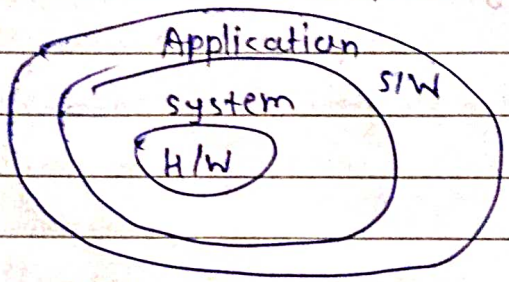
Language Translation Overview

Q) Need and component of a system.

→ Software that are related to operating system or it is an barrier (interface) b/w H/W and Application s/w.

Application software - Word,

System software - Windows, operating system, compiler, Assembler, interpreter.



convert

→ system s/w also helps to high level lang. to Machine lang.

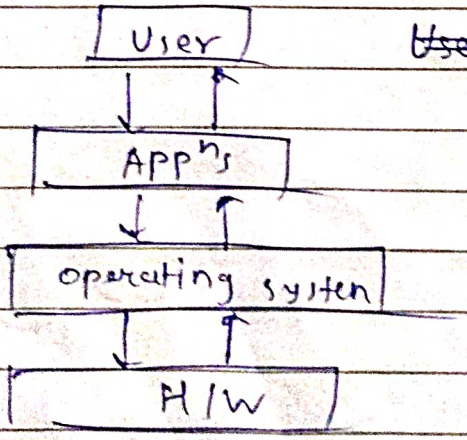
→ It is used to store database.

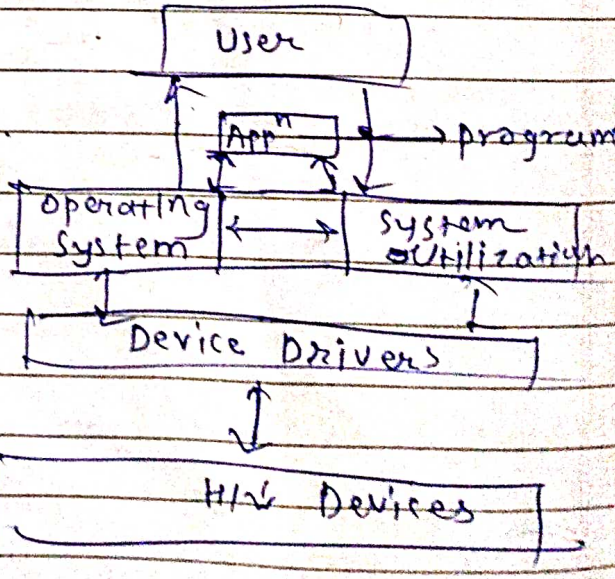
→ Manages basic f^n such as retriving files and scheduling task, etc

→ To make effective execution of each program.

→ To make effective ~~use~~ utilization of all the resources.

Component Of System S/W:

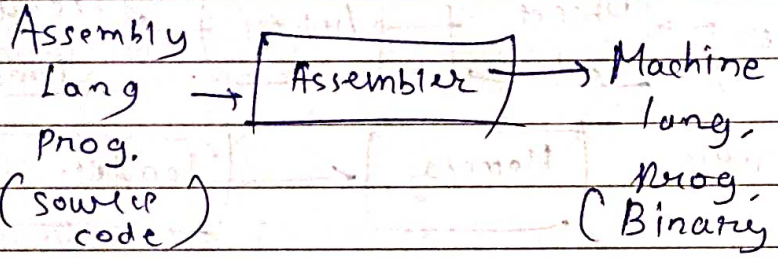




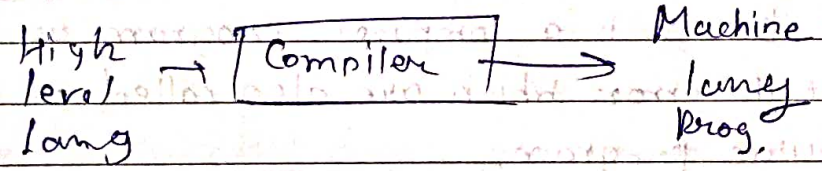
- → Assembler
- → Compiler
- → Interpreter
- → Editor
- → Linker
- → Loader

- → debuger
- → Macro
- → Operating System
- → Device drivers

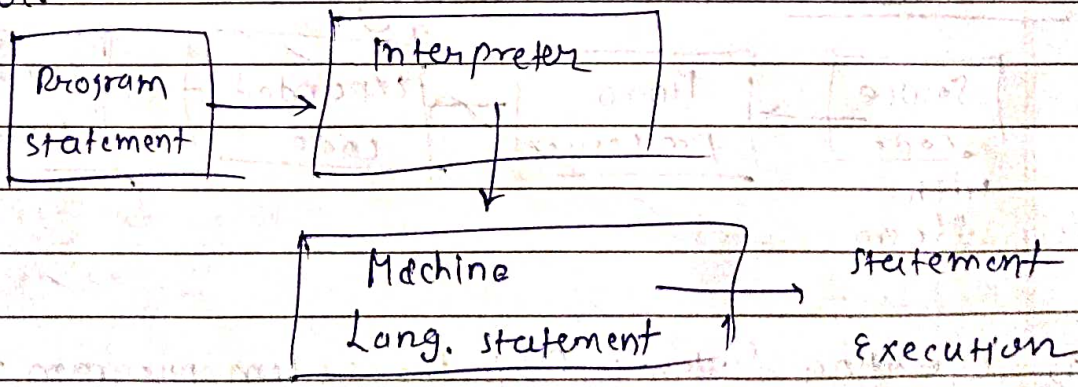
(1) Assembler -



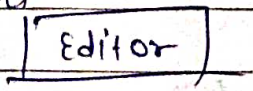
(2) Compiler :-



(3) Interpreter :-

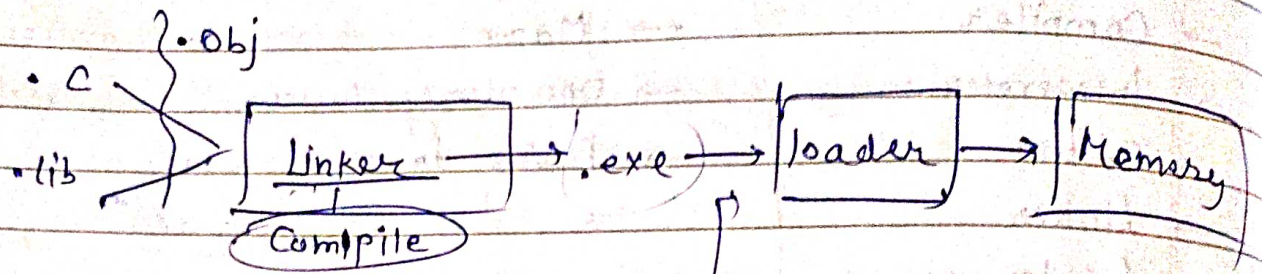


(4) Editors :- Edit a particular file, program, software as per requirement

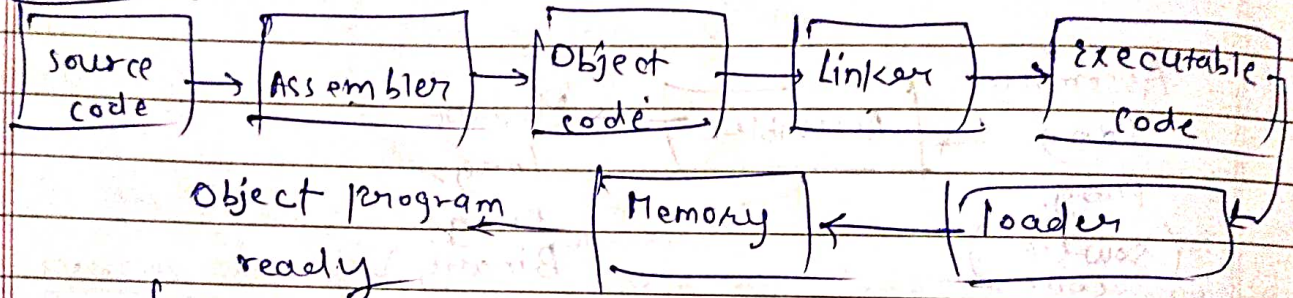


- line editor, screen editor, word processor, structure editor, etc.

(5) Linker :-

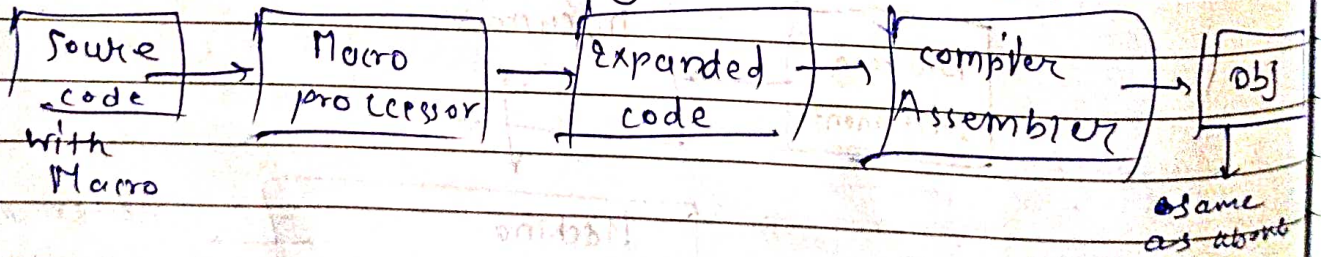


(6) Loader :-



(7) Debugger :- A Debugger is a computer program used to find out the errors which are also called as bugs in source program.

(8) Macro :- Code Reusability



(9) Operating System :- OS is the system program that enable users to communicate with computer H/W

→ It helps other program to run and control them.

Types of OS :- (1) Single User
(2) Multitasking
(3) Multiprocessor.
(4) Distributed OS
(5) Network OS

Ex. Linux, windows, Unix.

(iv) Device Drivers :- It is a System Program. Used to controll No. of devices which are attached to the computer.

→ Device Drivers tells OS that How the devices will work or Certain commands which are generated by the User.

Ex. Mouse D^r, keyboards D^r,
Wifi D^r, Bluetooth D^r.

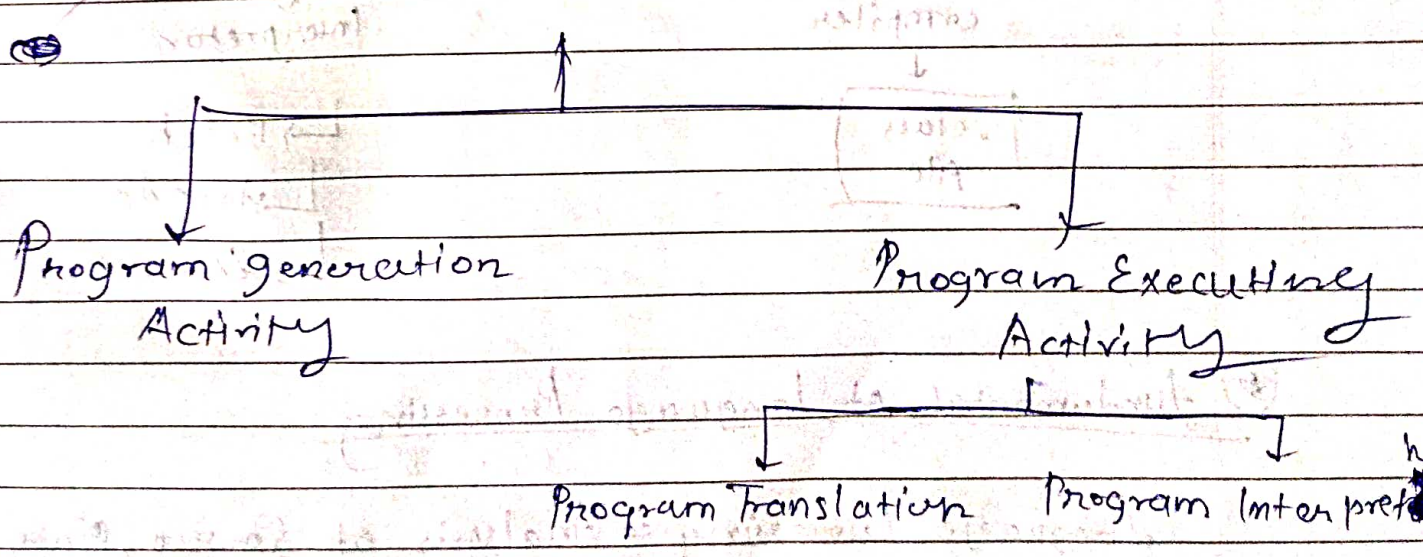
Language Processing Activities

→ Defination:

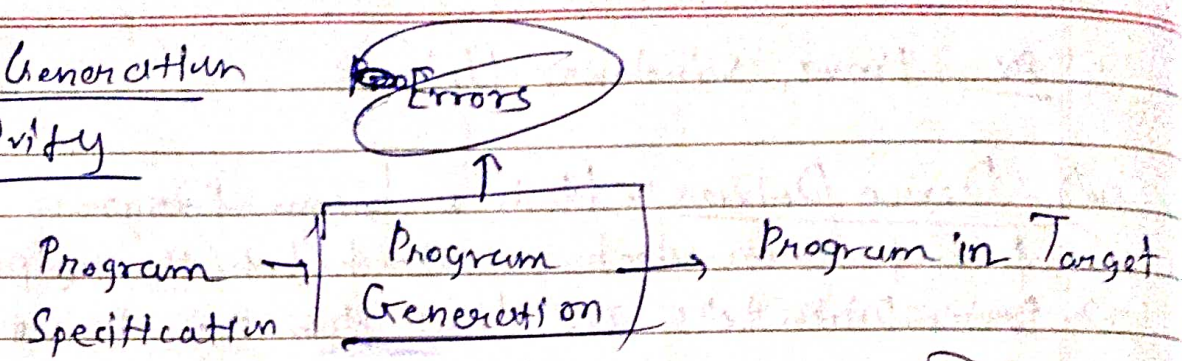
Language Processor :- Convert Source code into machine code.

Language Processing :- Activity which is carried Out by Language Processor.

Two Types of Activity are there :



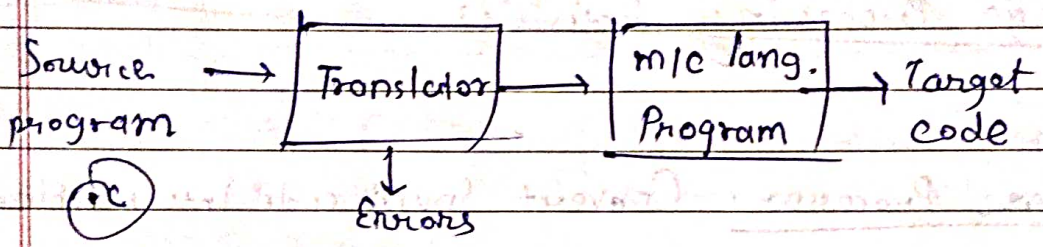
Program Generation Activity

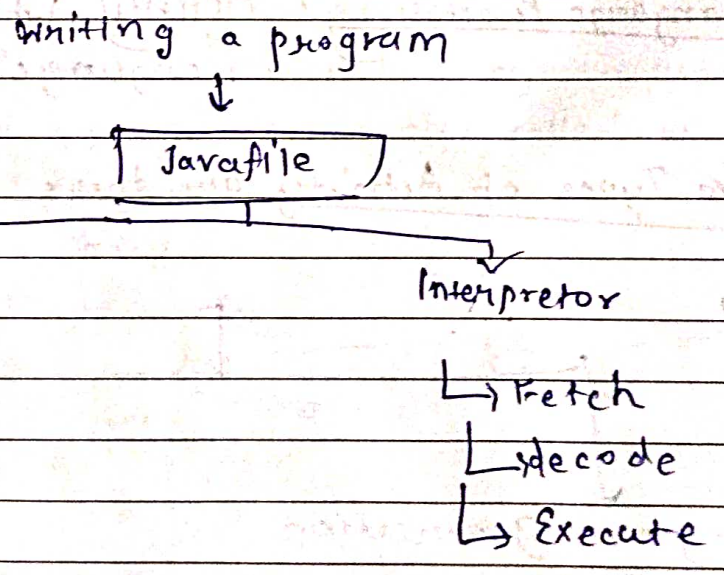


2+2

4

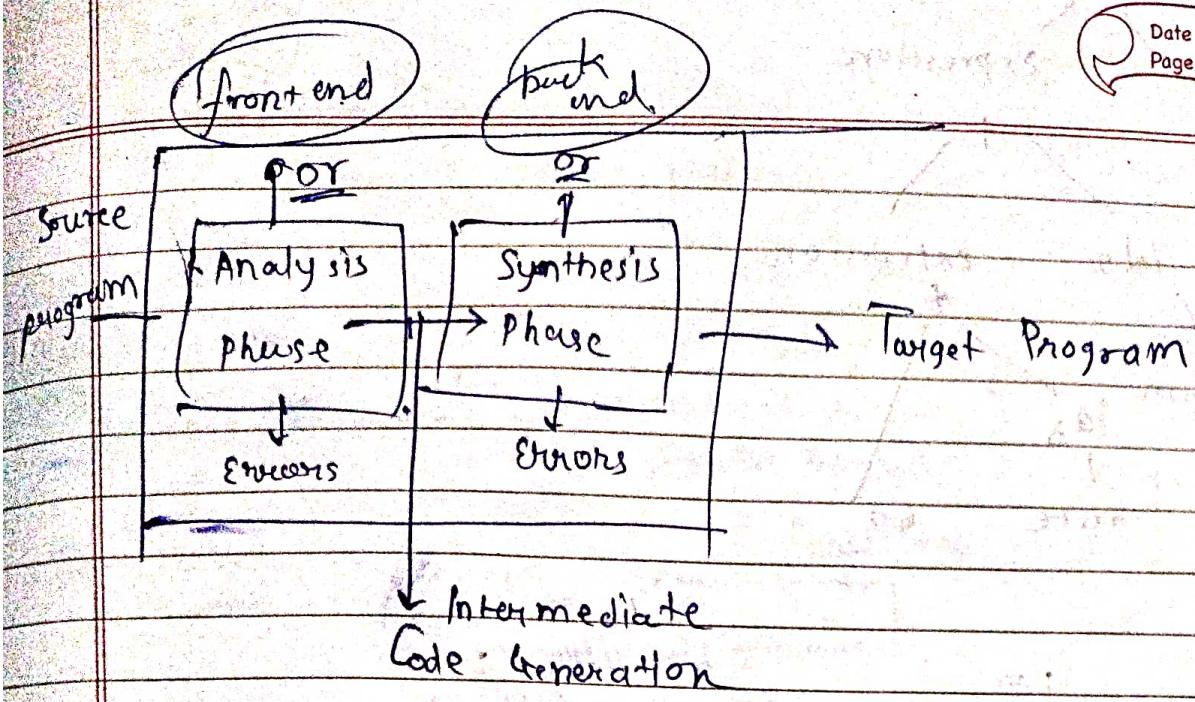
Program Translation





⑤ Fundamental of Language Processing

Language Processing = Analysis of Source Prog.
+ Synthesis of target Prog.



Analysis Phase :-

(Lexical Analysis)	(Syntax Analysis)	(Type checking) Will doing
linear Analysis :- In which the stream of characters making a the source program is Read from left to Right and Grouped into tokens, that are sequence of characters having a collective meaning.	**Hierarchical Analysis** In which characters or tokens are groped heirarchical into nested collections with collective meaning.	**Symentic Analysis** In which certain checks are performed to ensure that the components of a program fit together meaningfully.

(1) linear :-

initial := position + rate * 60

initial ——— identifier

:= ——— Assignment symbols

position ——— id

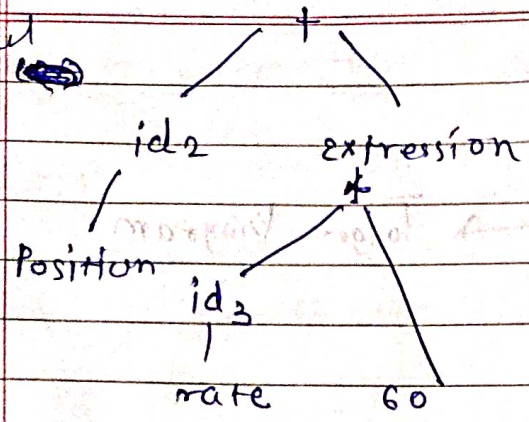
\+ ——— plus sign

rate ——— id

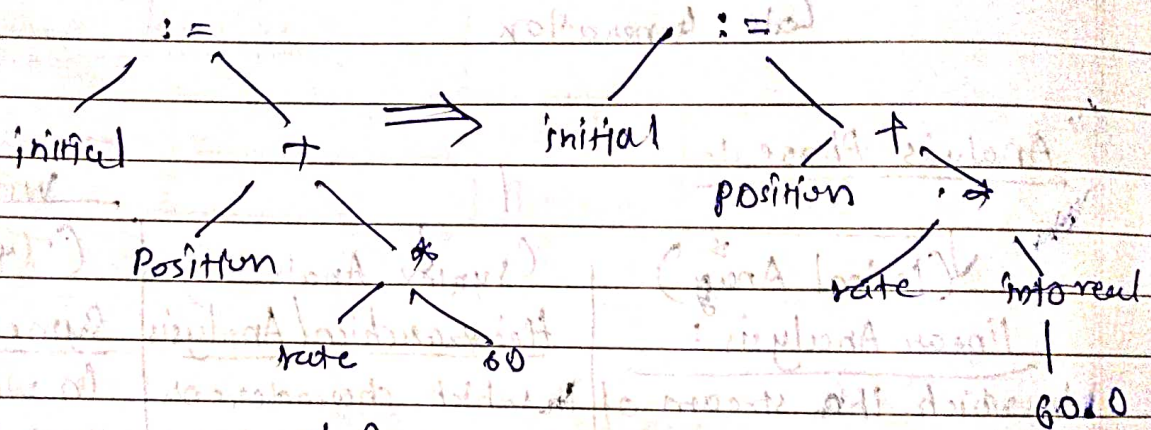
* ——— Mul.

60 ——— Constant.

(2) Expression → Assignment Statement



(3)



temp1 = inttoreal (60)
id3
temp2 = rate + temp1
id2
temp3 = position + temp2
initial = temp3
id1

(2) Synthesic Phase :-

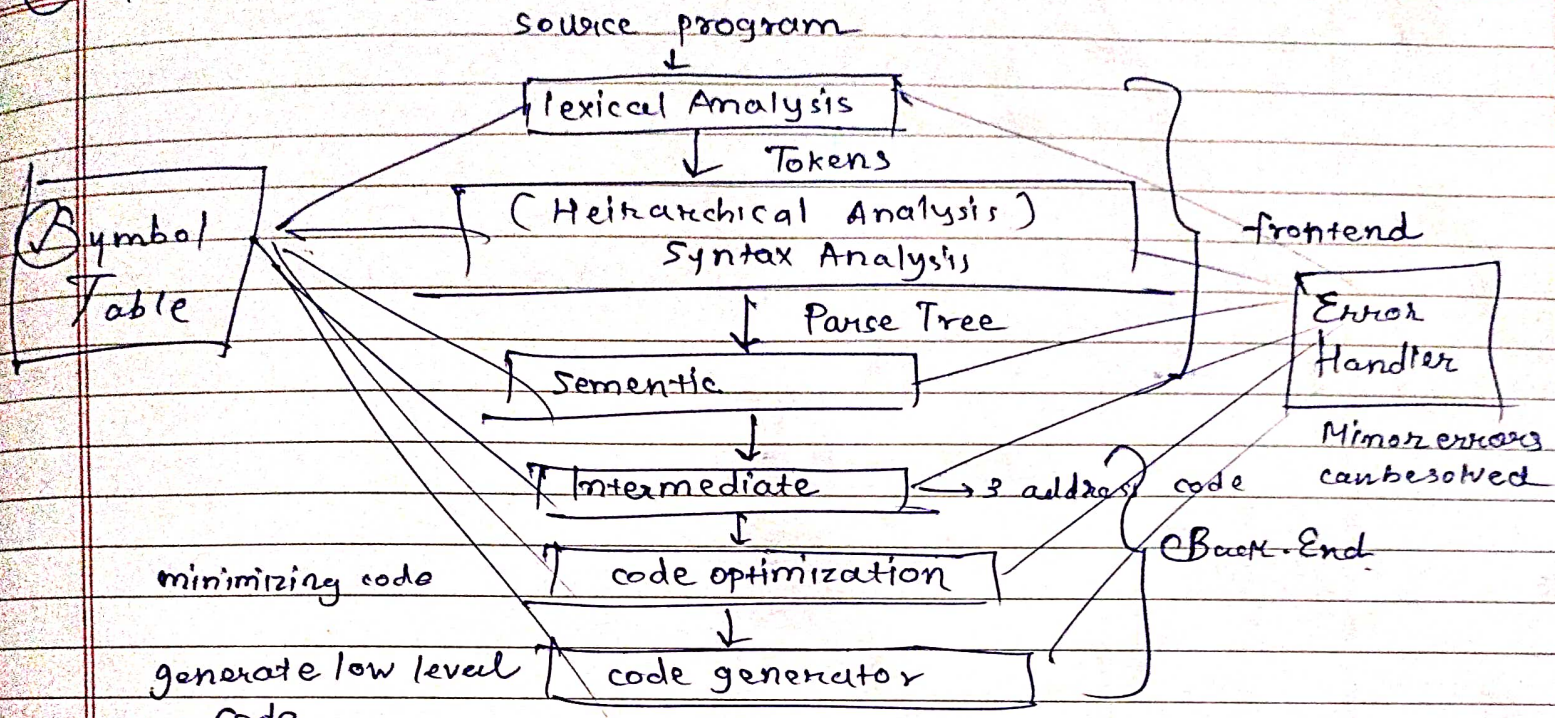
Code Optimizer :

temp1 = id3 * 60.0
id1 = id2 + temp1

Code Generation

MOVF id3, R1
MULF #60.0, R1
MOVF id2, R2
ADDF R2, R1
MOVF id1, R2

Phases of Compiler



Total = a/b + c * 60

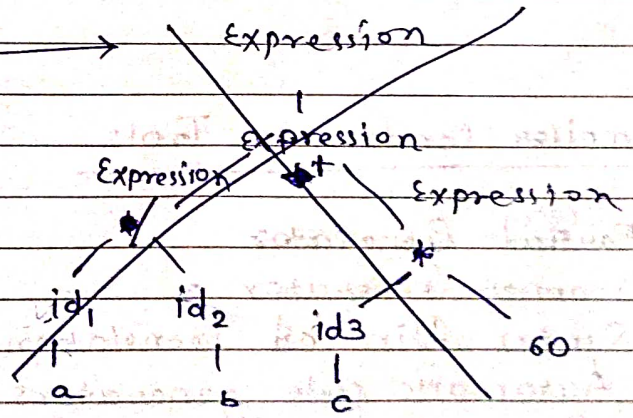
lexical Analysis

→ Tokens:-

total — id1
=
a — id2
/
b — id3
+
c — id4
*
60

Symbol table

1	total
2	a
3	b
4	c

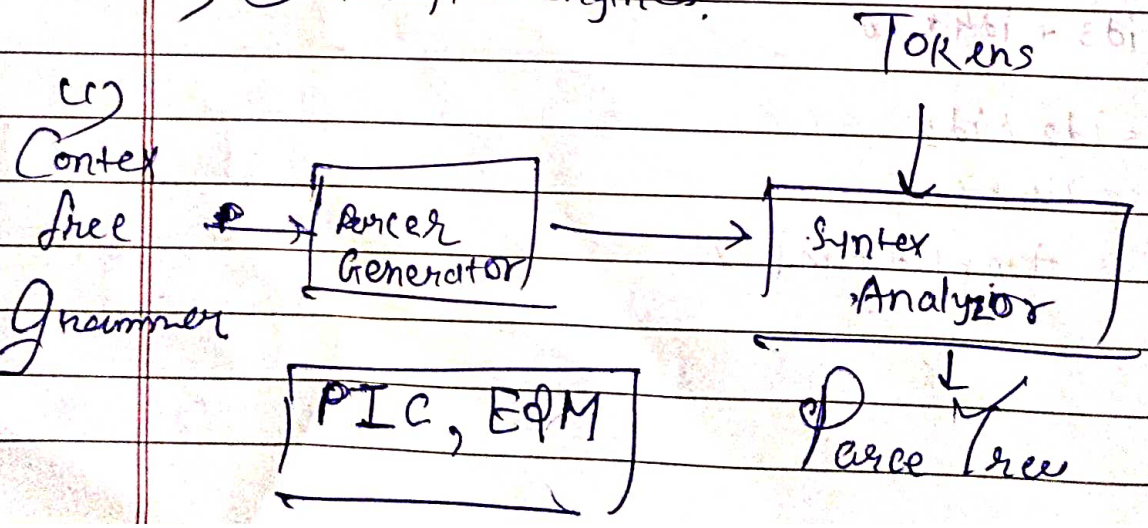


id1 = id2 / id3 + id4 * 60

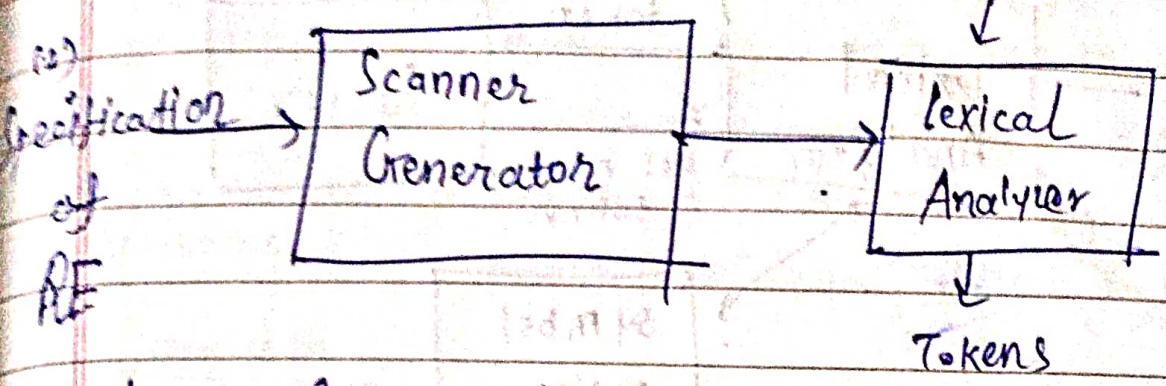
→ temp1 = id2 / id3
temp2 = id4 * 60
temp3 = temp1 + temp2

(*) Compiler Construction Tools

1) ~~Parser~~ Generator
2) Scanner Generator
3) Syntax Directed Translation Engines
4) Automatic code generators
5) Data-flow Engines.

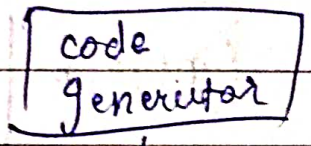


(2)



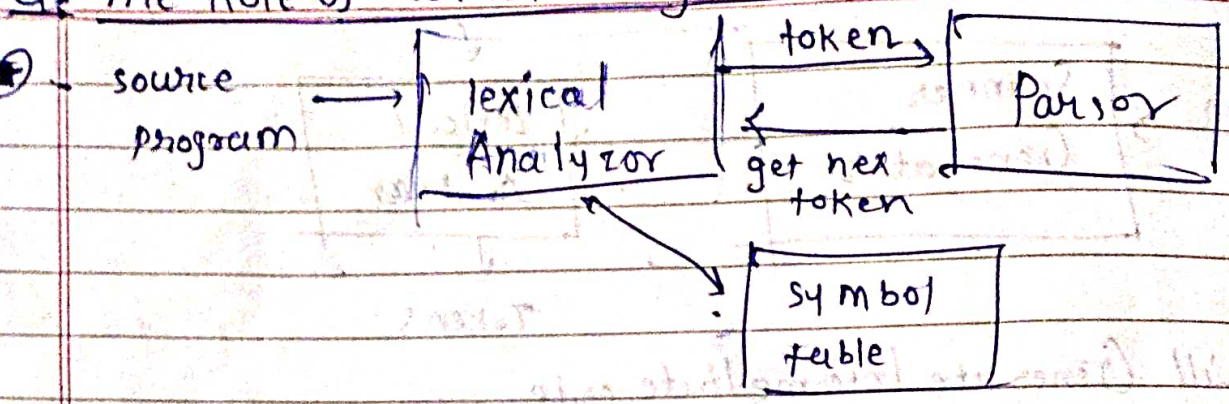
(3) It will Generate Intermediate code.

(4) optimizing code



(5) To make code as fast as possible

Q. The Role of Lexical Analysor



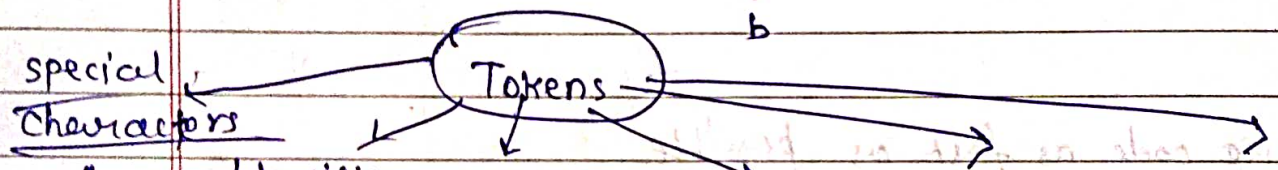
Tokenization :-

lexemes (x) → Tokens

int a, b

int a , b

Q. Difference b/w lexims & Tokens ?




```
1   int main() {
2     int a=20, b=30;
3     if (a<b)
4       return (b);
5     else
6       return (a);
7   }
```

printf ("i = %d, &i = %x", i, &i);

↓ Long string

33 = tokens

→ Eliminats comments & white space (tabs, blank space, newline)

Give Error Messages

(1) ~~Existing length~~ → Exceeding length of identifier. int a = 2139124908492;

(2) Unmatched Strings → (/* ... */) x = "one ;

(3) Illegal characters (am#sh) (int # = 30)

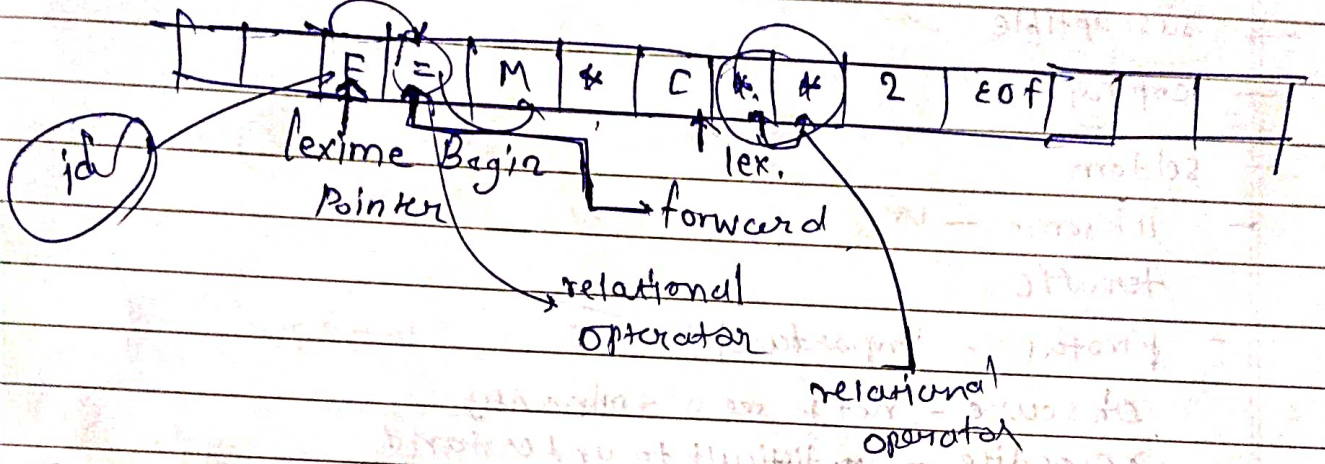
Apperance of

printf ("6th →"); $ (More on next 2)

2. Sentinal

Buffers :- It's stores the data for short amount of time.

E = M * C * * 2



Recognization of Tokens :-

(1) Recognization of Identifiers

(2) " of Delimiters

(3) " of Relational Operator → >= <= != ==

(4) " of Keywords

(5) " of Numbers

(1) Identifiers A to z, a to z, digit 0 to 9

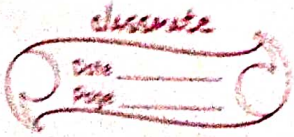
id → letter (letter / digit)*

(2) Delimiters new line tab or space \n, \t, " "

delim → blank | tab | newline

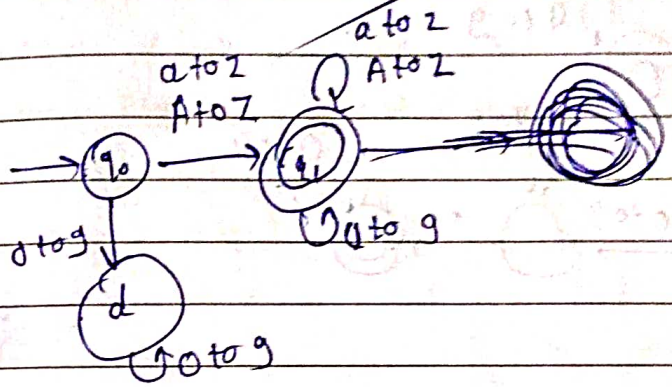
ws → (delim)+

The Tokens and associated attribute values for the statement E = M * C * * 2 , <'id, pointer to symbol table entry for E>

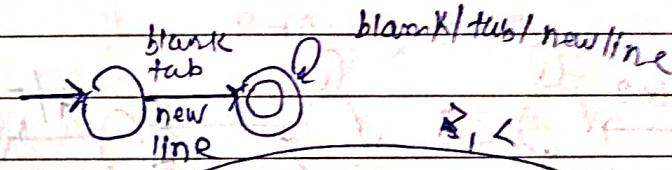


< assign_op , assign operator >
< id , " " >
< mult_op , * >
< id , " " >
< ~~relop~~ Exp_op , >
< num ,

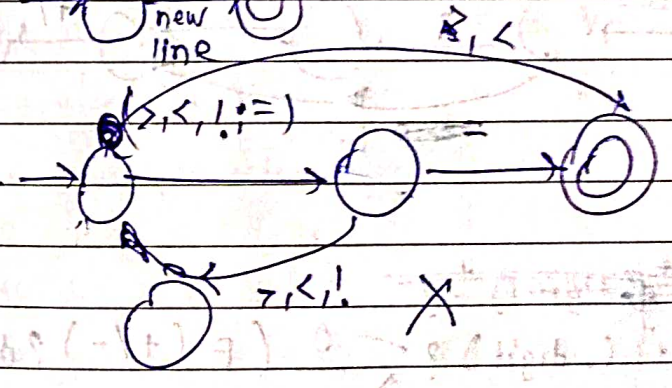
(1)

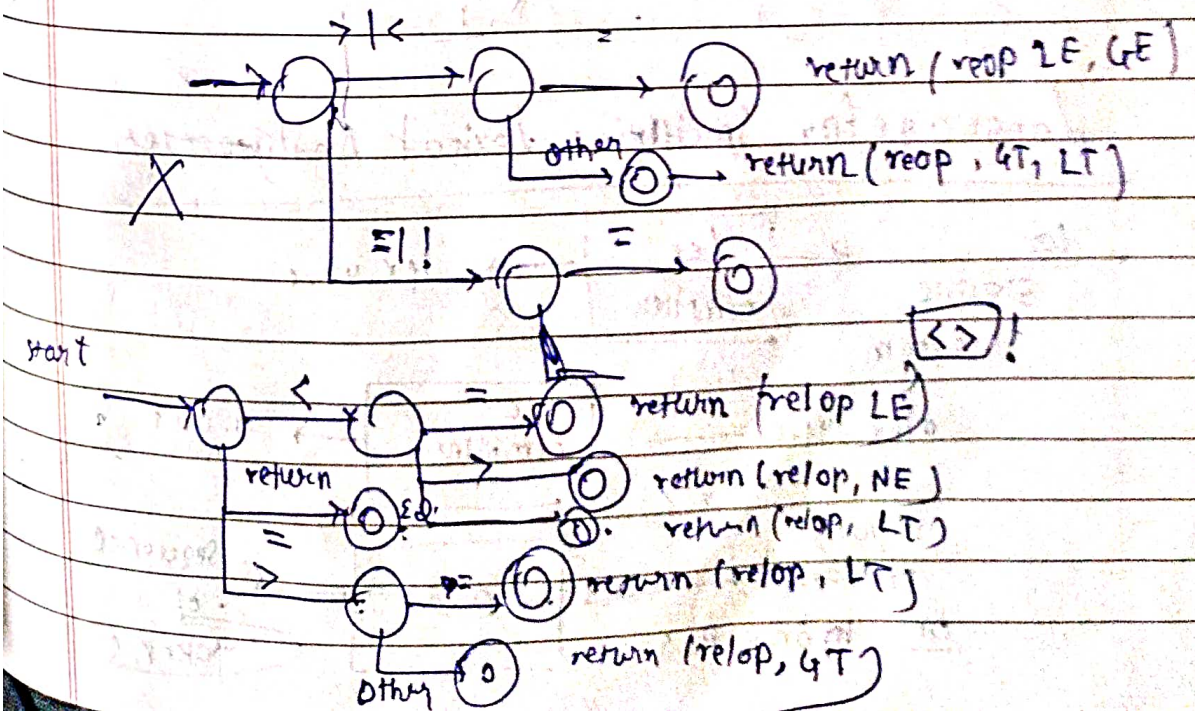


(2)

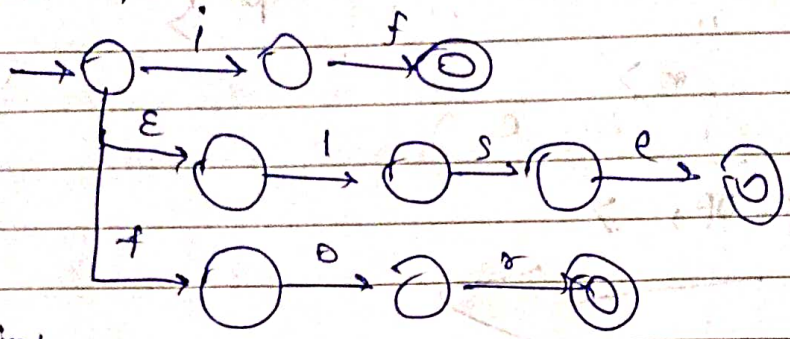


(3)

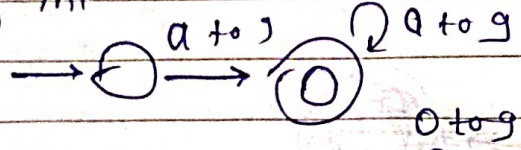




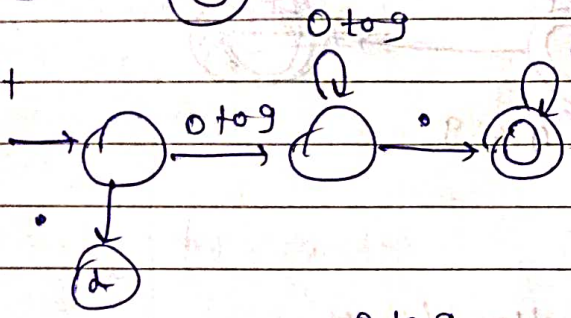
(4) if, else, for, while,



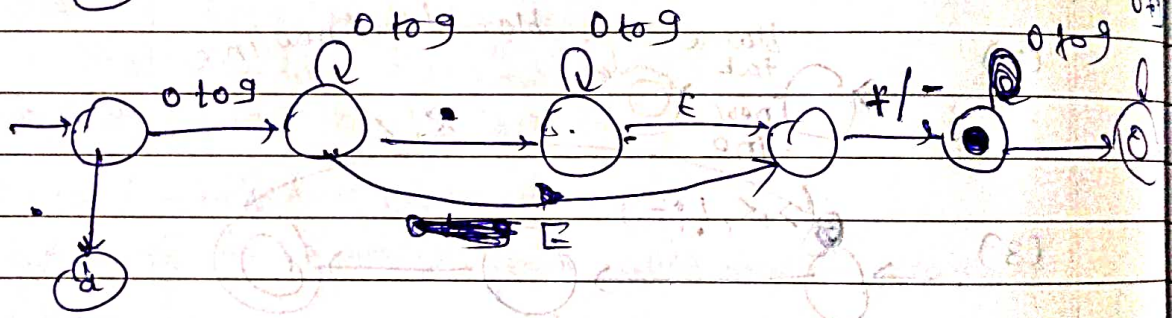
or) 1) int



2) float



3)

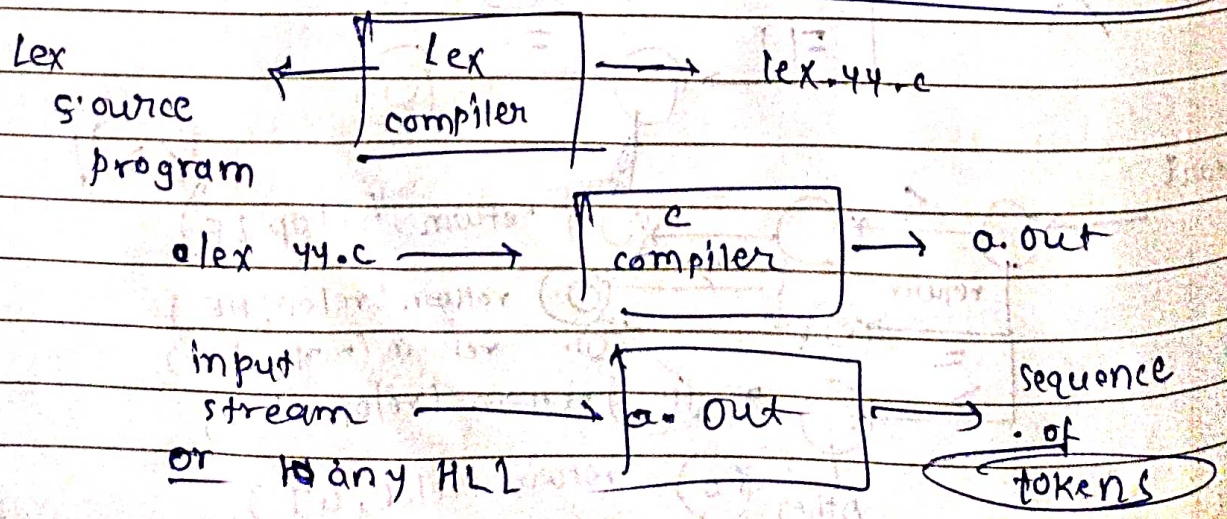


~~num → digit + digit . ?~~

num → digit$^+$ (. digit)? ← 0 (E (+/-)? digit$^+$)?

1

Language for Specifying Lexical Analyzer



Lex specification

abc [a-z] [a-z]* [^n;0-2]+	^a → start with a ab$ → end [0-9]+ yy text

yywrap() - It return 1
yylex()

→ yytext : It is a pointer to a input stream, (executed)
→ yylex() - It called by lex or where input is executed.
→ yylex ⇒ Iterated input stream and generate token to the RegEx

remaining

declarations →
%{
%}

%%
transition rules : Pattern actions
%%
auxiliary Procedures, → main part

→ delim
→ delim

%{
#include <stdio.h> → header
/* */ → declaration
%}
%%
"hii" { printf("How are you"); }
.* { printf("wrong string"); }
%%

```
main()
{
  printf("enter i/p");
  yylex();   ⟶ generate tokens
}
int yywrap()
{ return 1; }
```

(*) find lexems, tokens and patterns of the following Question:

int x = 5;

lexems	Tokens	Patterns
int	keyword	keyword
x	identifier	letters (letter / digit)
=	operator	=, <, >
5	constant	constant, number
;	separator	separator

(*) find out lexecal Errors :-

(i) Int x; ⟶ This is not a lexical Error

(ii) int x,y; no lexical Error

(iii) float m = 5..7; lexical Error

(iv) int 1sal; lexical error

Error (Continued)

(4) lexecal Error: spelling error or identifier Rule not satisfied)

int 6a = 75;

(5) Replacing a character with an incorrect character.

int x = 1$2;

int a= b;

(vi) int A = #%2; lexical Error

(vii) String s = " sucess; "

(viii) #include <stdio.h>

```
main()
{
  int x=5, y=6;
  char *a;
  a = &x;
  x = 56)cab;
  printf("%d", x);
```

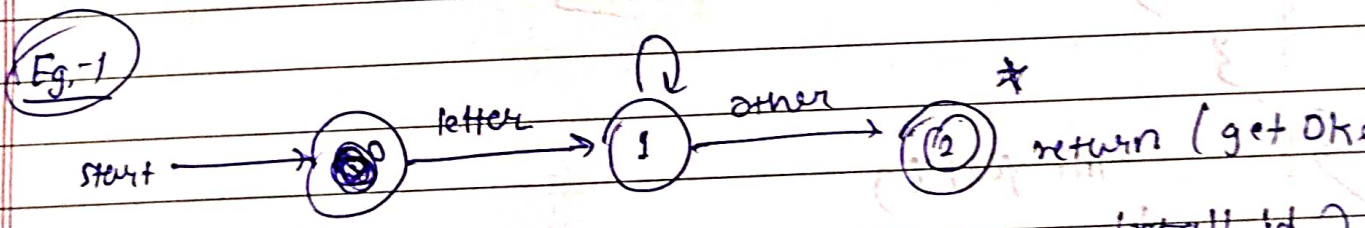
(viii) a) int a b;

b) in t a b;

c) int 1 ab 2;

d) All

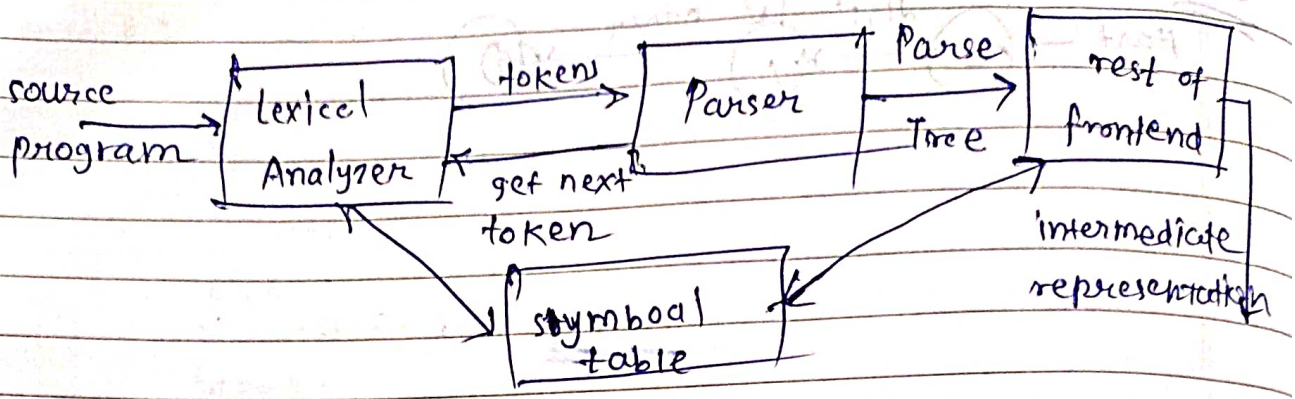
Eg.-1




```
int state = 0;
token nexttoken()
{
  while(1)
  {
  switch(state)
  {
    case 0:
      c = nextchar();
      if (isletter())
        state = 1;
      else state = fail();
      break;
```

```
case 1 :
        c = nextchar ();
        if (isletter () || is digit ())
            state = 1;
        else
            state = 2;
        break;
case 2 :
        retract (1); // retract is a loop ahead pointer 1
        install id ();   character, it is used b'cause
                         delim is not a part of indentifier
// is used to access the buffer & it is called as
   attribute value
        return (getoken ()); // to obtain the tokens
    }
}
int fail ()
{
    switch (start)
    {
        case 0:
            start = 0;
            break;
        default:
            /* compiler error */
    }
    return start;
}
```

The Role of the Parser



Contex-free Grammer :- Contex free grammar is four tuples

$$G = (V, T, P, S)$$

↓ Variables

V = finite set of symbols called as non terminal or variable
- it may be in upper case
- usually S is used for starting symbol, some italic names are also allowed.

Ex. {expr stmt

$$S \rightarrow 0A \mid 1$$
$$A \rightarrow 0B \mid 0$$
$$B \rightarrow \varepsilon$$

T = set of symbol that are called as Terminals.
1) lowerCase letters,
2) Operator Symbols +, -
3) punctuation Symbols like {}, ,
4) digits 0 to 9
5) Bold phased String (id, if)

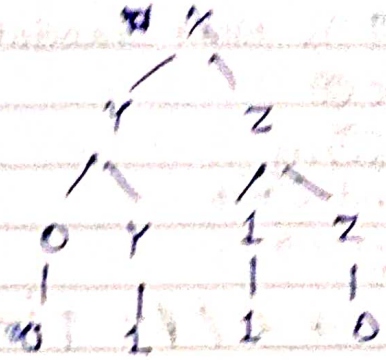
Such symbols are called as terminals.

P = is a set of production

S = is a member of V, start sym.

X → YZ
Y → 0Y / 1
Z → 1Z / 0

w = 0110



(*) Ambiguity : A Grammer that produce more than one parse tree for same sentence is said to be Ambiguous Grammer.

S → Sa / aS / a

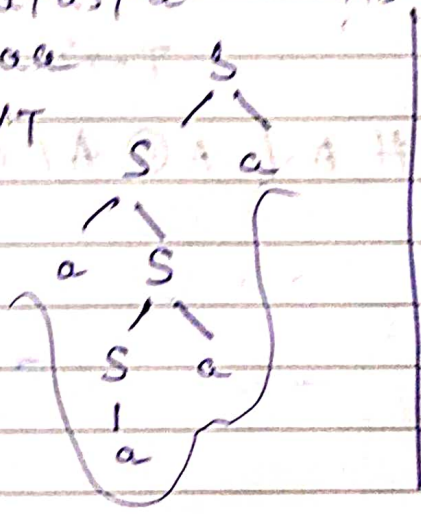
w = aaaa

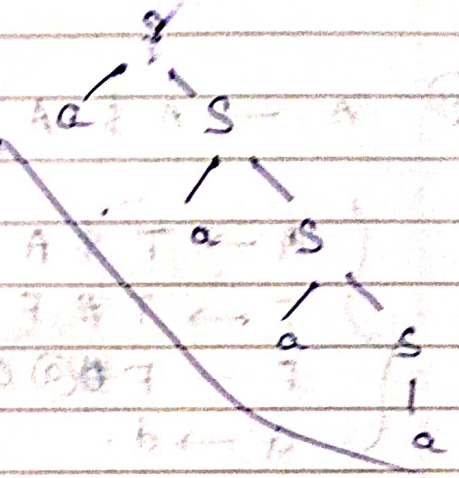
S → Sa / a / T

T → aS

LMD

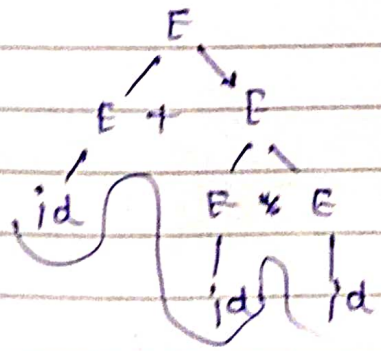
RMD → Left Most Derivative tree

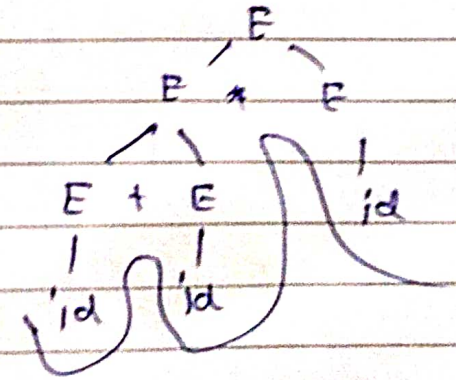




(*) E → E+E / E*E / id

w = id1 id * id

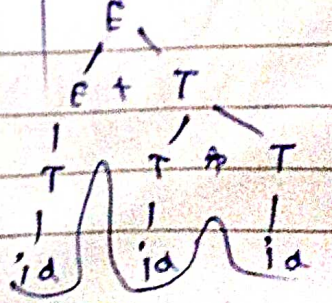




E → E+T / T

T → T*F / F

F → id



Rules for Removing Ambiguity :-

(1) $ and ↑ is right accociated. Remaining all are left accociati

(2) $ < # < @

priority

(3) or < and < not

(4) − < + < * < / < ↑ or $

(5) + < * < *

(6)

(7) E → E$E / id → E → T$E | T

T → id

(8) A → A $ A / A # A / A @ A / d.

A → T $ A / T

T → T # F / F

F → F @ G / G

G → d.

S → d

eg: bExp → bExp or bExp | bExp and bExp | Not bExp | True | False

bExp → bExp or T | T

T → T and F | F

F → Not S | S

S → True | False

eg: R → R+R | RR | R* | a | b | c

E → E+T | T

T → TF | F

F → F* | a | b | c

R → R+T | T

T → TF | F

F → a* | a

a → a | b | c

* Left Recursion

$A \rightarrow A\alpha \mid \beta$

Algorithm :

input: Will take some Grammar G.
output: Equivalent grammar with no left rec.

Method : if (we have left recursive pair of production $A \rightarrow A\alpha \mid \beta$
Where β does not begin with A
then
we can eliminate left recursion by replacing this pair of production with
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

→ Two types of left Recursion
- Direct left Recursion
- Indirect left Recursion.

(i) Direct left Recursion.

Eg $A \rightarrow ABd \mid Aa \mid a$ abd
$B \rightarrow Be \mid b$

$B' \rightarrow eB' \mid \varepsilon$

$A \rightarrow aA'$

$A' \rightarrow aA' \mid \varepsilon \mid BdA'$

Eg $A \Rightarrow Aa B \mid x$

$B \Rightarrow Bcb \mid y$

$C \rightarrow Cd \mid \varepsilon$

$A \rightarrow xA'$

$A' \rightarrow aBA' \mid \varepsilon$

$B \rightarrow yB'$

$B' \rightarrow cbB' \mid \varepsilon$

$C \rightarrow C'$

$C' \rightarrow cC' \mid \varepsilon$

$A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

Indirect left Recursion :-

Q. $S \to Aa / b$

$A \to Ac / Sd / \varepsilon$

$\Rightarrow A \to Ac / Aad / bd / \varepsilon$

$A \to bdA' / A'$

$A' \to cA' / adA' / \varepsilon$

Left Factoring :-

$A \to aAb / aAc / aAd / \varepsilon$

Algo. for removing :

Input :- will take grammer G.

Output :- Equivelent non left factor Grammar.

Method :- for each non terminal A find the prefix α, common to two or more of its alternatives. Replace one of the A productions

$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_4 / \gamma$

where, γ represents all alternatives that do not begin with α.

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 / \beta_3 / \beta_4$

Repidetly apply that transformation until those two alternat for a common prefix.

$A \to abc / abd / acd / ad / d$

(*) $S \to iEts / iEtses / a$

$E \to b$

$S \to iEtsA' / a \qquad E \to b$

$A' \to es / \varepsilon$

(i) $A \to xByA \mid xByAzA \mid a$

$B \to b$

$A \to xByAM \mid a \quad B \to b$

$M \to zaA \mid \varepsilon$

(ii) $A \to aAb \mid aA \mid a$

$A \to aAA' \mid a$

$A' \to b \mid \varepsilon$

$A \to aA'$

$A' \to Ab \mid A \mid \varepsilon$

$A' \to AA'' \mid \varepsilon$

$A'' \to b \mid \varepsilon$

(iii) $A \to ad \mid a \mid ab \mid abc \mid b$

$A \to aB \mid b$

$B \to d \mid b \mid bc \mid \varepsilon$

$B \to bD \mid d \mid \varepsilon$

$D \to c \mid \varepsilon$

(iv)

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

$E \to TE'$

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

$T \to FT'$

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

$F \to id$

Parcer

Top down parser — Bottom up parser

Brute force method — Recursive Decent Parcer — Non-reursive Decent parser

(1) Top-Down Parcer :- Generates parse tree for the given input stream with the help of grammer productions by expanding the non-terminals in this it will start from start symbol and ends on the terminals.

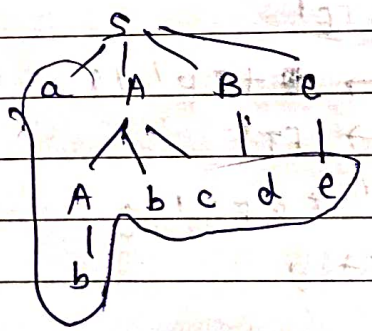
→ It uses left-Most Derivation

$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

string - abbcde



$S \rightarrow aABe$

$\rightarrow aAbcBe$

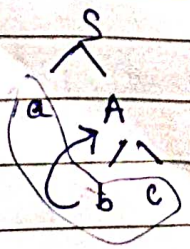
$\rightarrow abbcBe$

$\rightarrow abbcde$

→ at every point we have to decide "what" is the next production we should use.

(a) Brute force Parce Tree :-

(i)



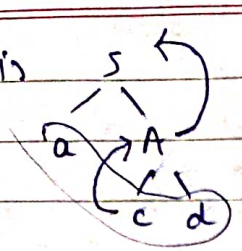
$S \rightarrow aA / bB$

$A \rightarrow bc / cd$

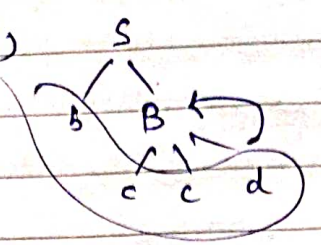
$B \rightarrow ccd / ddc$

string - bddc

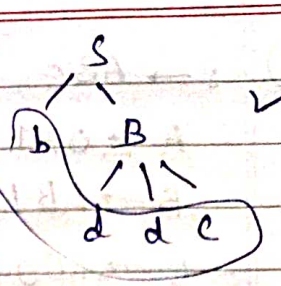
(ii)



(iii)



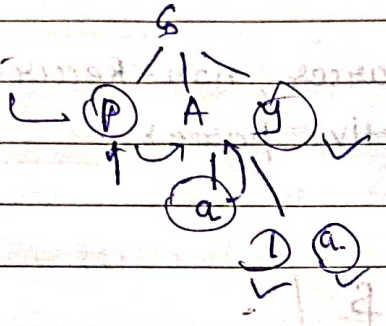
(iv)



⇒ T.C. is $O(2^n)$ → We have to generate more no. of parse trees

(b) Recursive Desent Parser :-

match fn :-



$S \rightarrow pAy$

$A \rightarrow a / La$

string :- play

⊛ $E \rightarrow iE'$

$E' \rightarrow +iE' / \epsilon$

(i) E()

(ii) E'()

(iii) match (token t)

$S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

→

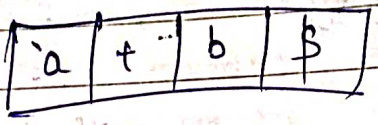
```
E(){
    if (lookahead == 'i')
    {
        match('i');
        E'();
    }
    if (lookahead == '$')
    {
        print ("success");
    }
    else
        return;
}
```

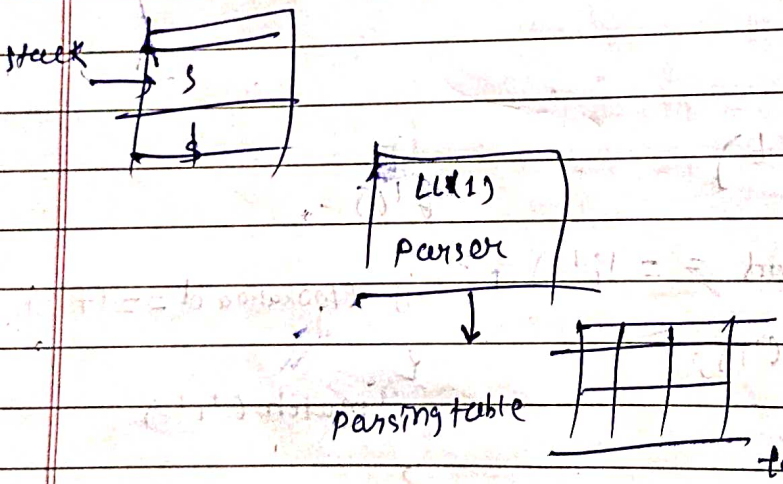
```
E'()
{
    if (lookahead == '+')
    {
        match ('+')
    }
    if (lookahead == 'i')
    {
        match (i)
    }
    E'();
    else return;
}
```

(A) $S \rightarrow ABC$
$A \rightarrow 0A1 / \varepsilon$
$B \rightarrow 1BL / \varepsilon$
$C \rightarrow 1C0 / \varepsilon$

(A) $E \rightarrow E+i / i \rightarrow E \rightarrow iE'$
$E' \rightarrow +iE' / \varepsilon$

(A) LL(1) parser, perdictive parser, non-Recursive predictive parser, Table driven perdictive parser.





→ LL(1) → In the input symbol means it uses only 1 i/p sym
→ It uses left most derivation for input stream.
↓ input is scanned from left to right

→ Data structure Uses 3 things: Stack, Parsing Table, I/P buffer
↓ ↓ ↓ stores the input token

→ steps of LL(1): (1) Remove Left Recursion
(2) Find first and follow,

(3) Construct parse table
(4) Stack implementation
(5) Parse Tree generation

① $E \to E+E \mid E*E \mid id$

eg $E \to id\,E'$
$E \to +EE' \mid *EE' \mid \varepsilon$
} (1) Remove ambiguity (2) left Recursion

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

first (Rules/Steps) :-
(1) If any terminal are there put it directly
(2) If non-terminal are there then go to the non-terminal and repeat the step 1.
(3) If Null are there

	first
$E \to TE'$	{c, id}
$E' \to +TE' \mid \varepsilon$	{+, ε}
$T \to FT'$	{c, id}
$T' \to *FT' \mid \varepsilon$	{*, ε}
$F \to (E) \mid id$	{c, id}

follow (Rules/Steps) :-
(1) first production $ is compulsory.
(2) Null never be there.

	follow
E	{$,)}
E'	{$,)}
T	{), +, $}
T'	{), +, $}
F	{), *, +, $}

②

	first	follow
$S \to aBDh$	{a}	{$}
$B \to Cc$	{c}	{g, f, h}
$C \to bC \mid \varepsilon$	{b, c, ε}	{g, f, h}
$D \to EF$	{g, f, ε}	{h}
$E \to g \mid \varepsilon$	{g, ε}	{f, h}
$F \to f \mid \varepsilon$	{f, ε}	{h}

③

	first	follow
$S \to ABC \mid bB \mid BC$	{d, g, h, ε, b}	{$}
$A \to da \mid BC$	{d, g, h, ε}	{$, g, h}
$B \to g \mid \varepsilon$	{g, ε}	{h, $, a}
$C \to h \mid \varepsilon$	{h, ε}	{$, b, g, h}

$m = 2$ | $80 - 32 +$

~~First~~

~~$S \to (L) / a$~~	~~$(, a)$~~
~~$L \to L, S / S$~~	~~$,$~~

	first	follow
$S \to (L) / a$	$\{ (, a \}$	$\{ \$, ,,), \}$
$L \to SL'$	$\{ (, a \}$	$\{) \}$
$L' \to , SL' / \varepsilon$	$\{ , , \varepsilon \}$	$\{) \}$

	first	follow
$S \to ABC$	$\{ a, b, d, e, f, \varepsilon \}$	$\{ \$ \}$
$A \to a / b / \varepsilon$	$\{ a, b, \varepsilon \}$	$\{ \$ \}$
$B \to c / d / \varepsilon$	$\{ c, d, \varepsilon \}$	$\{ \$ \}$
$C \to e / f / \varepsilon$	$\{ e, f, \varepsilon \}$	$\{ \$ \}$

~~$S \to aBDh$~~ ~~$B \to cC$~~

Rules (Cont...) :

(3) If no. of non terminals are there at RHS then we don't req. any thing in follow.

(4) B ← NT → First → ε → futher no

not ε → done

T → add into follow

End

Nothing → left side → Termind → foll

(5) If follow of itself is there then no need to require any further writing of follow.

(3) Constructing a parse table

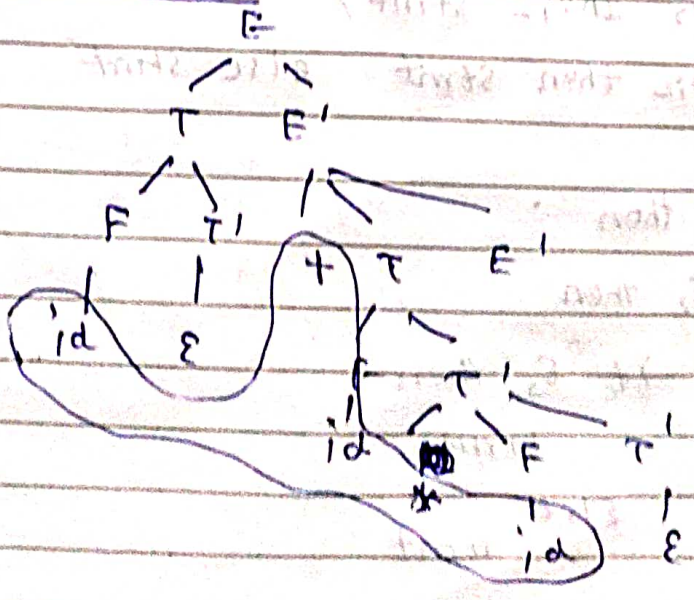
	t	id	*	c	d	$
E		$E \to TE'$		$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T		$T \to ET'$		$T \to ET'$		
T'	$T' \to \varepsilon$		$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F		$F \to id$		$F \to (E)$		

string : id + id * id

(4) Stack Implementation :

Stacks	Input Buffer	Implementation
$ E	id + id * id $	
$ E' T	id + id * id $	E → TE'
$ E' T' F	id + id * id $	T → FT'
$ E' T' id	id + id * id $	F → id
$ E' T'	+ id * id $	matched
$ E'	+ id * id $	T' → ε
$ E' T +	+ id * id $	E' → +TE'
$ E' T	id * id $	matched
$ E' T' F	id * id $	T → FT'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	matched
$ E' T' F *	* id $	T' → *FT'
$ E' T' F	id $	matched
$ E' T' id	id $	F → id
$ E' T'	$	matched
$ E'	$	E' → ε, T' → ε
$	$	

(5) Parse Tree Generation :-



(2)

	First	Follow
S → iEtSS' / a	{ i, a }	{ $, e }
S' → eS / ε	{ e, ε }	{ $, e }
E → b	{ b }	{ t }

(3)

	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS, S' → ε			S' → ε
E		E → b				

this will not accepted by LL(1) parser.

(*) S → (L) / a
L → L, S / S

String :- (a, a)

	(	)	,	a	$
S	S → (L)			S → a	
L	L → SL'			L → SL'	
L'		L' → ε	L' → , SL'		

(*) Dangling Else problem :-

stmt = iEts | iEtSeS / other

→ if Expr then stmt /
if Expr then stmt else stmt

if E_1 then
  if E_2 then
    if E_3 then
      stmt
    else
      stmt

stmt → matched stmt / unmached stmt

matched stmt =

if expr then matched stmt

else matched stmt

if expr then stmt

Expr then mathed stmt

else unmatched stmt

Unmatched stmt = if Expt then stmt

E1

if Expt then

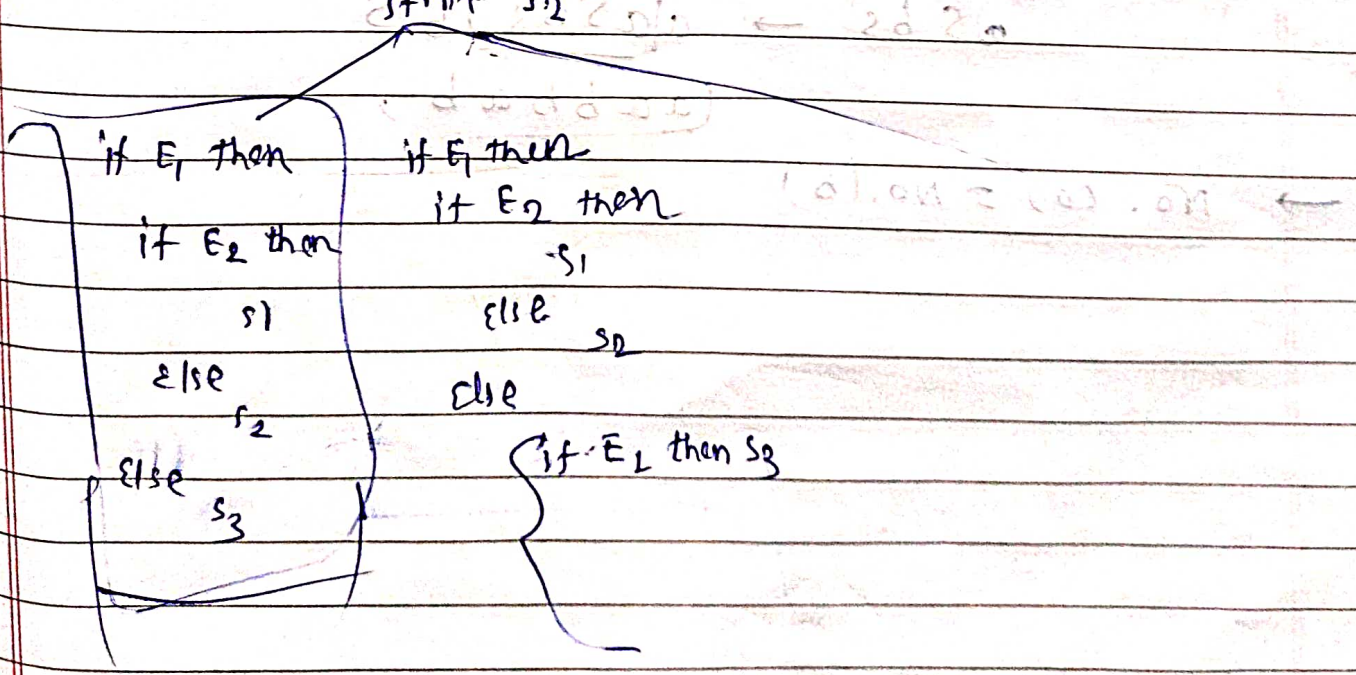
E2

if expr then

stmt S1

Else

stmt S2



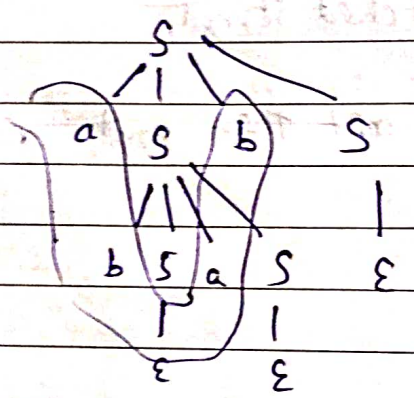
~~stmt~~ → if expn then stmt / matched_stmt

matched_stmt → if expn then matched_stmt

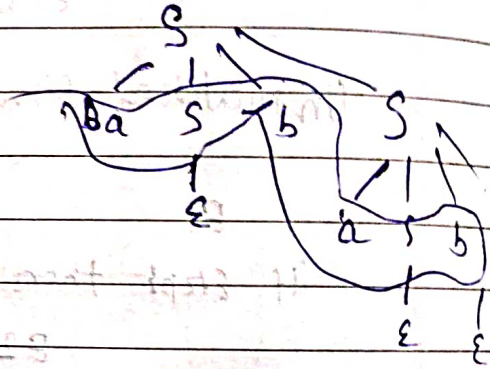
else stmt / other

S → aSbS / bSaS / ε

(5) Show that ambiguous by constructing two diff. left most derivations for the sentence abab construct the corresponding right most derivation for abab, what lang. does this grammar generate?



LMD



RMD

aSbS → a[aSbS]bS

aabbab,

→ No. (a) = No. (b)

(2x+1)

Date
Page

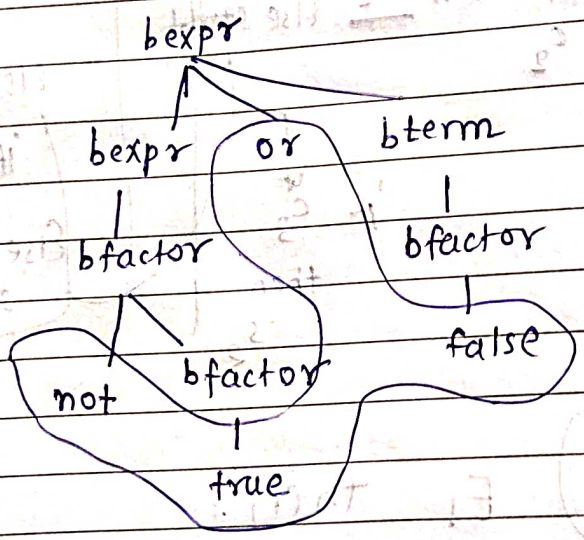
(6) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

→ (1) Construct a purse tree for a sentence :-
not true or false

(2) Show that this grammer generates all boolean Expressi

(3) Is this grammer ambiguous? Why ?

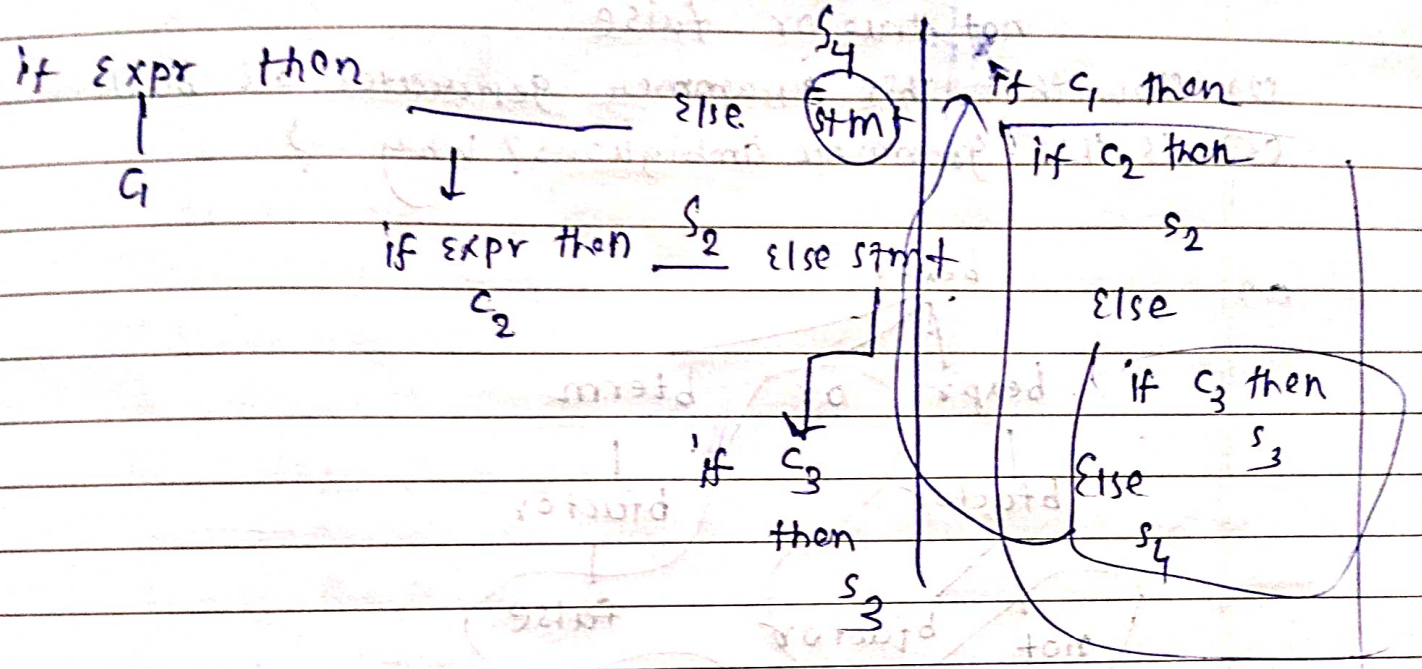
a)

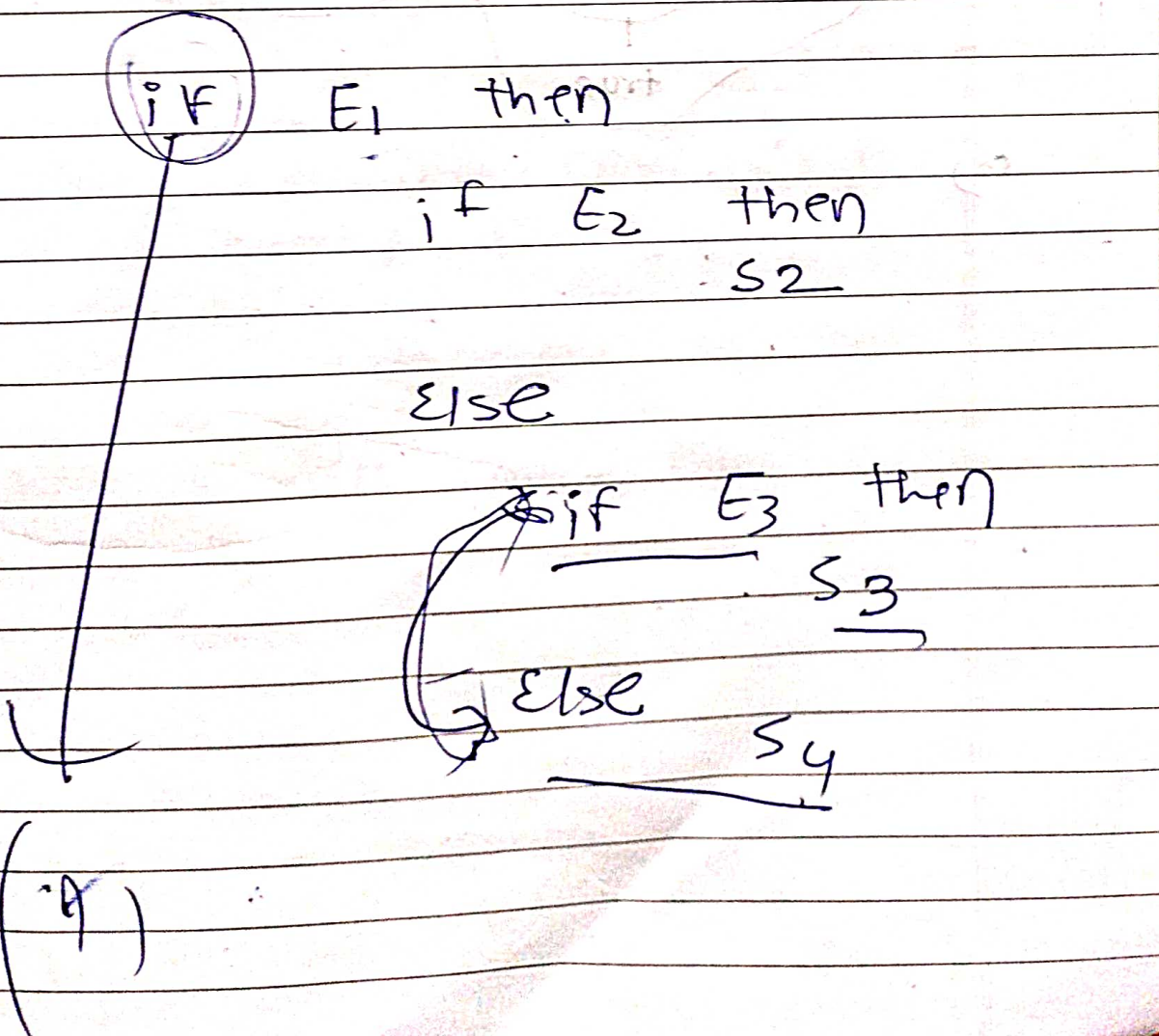


c2)

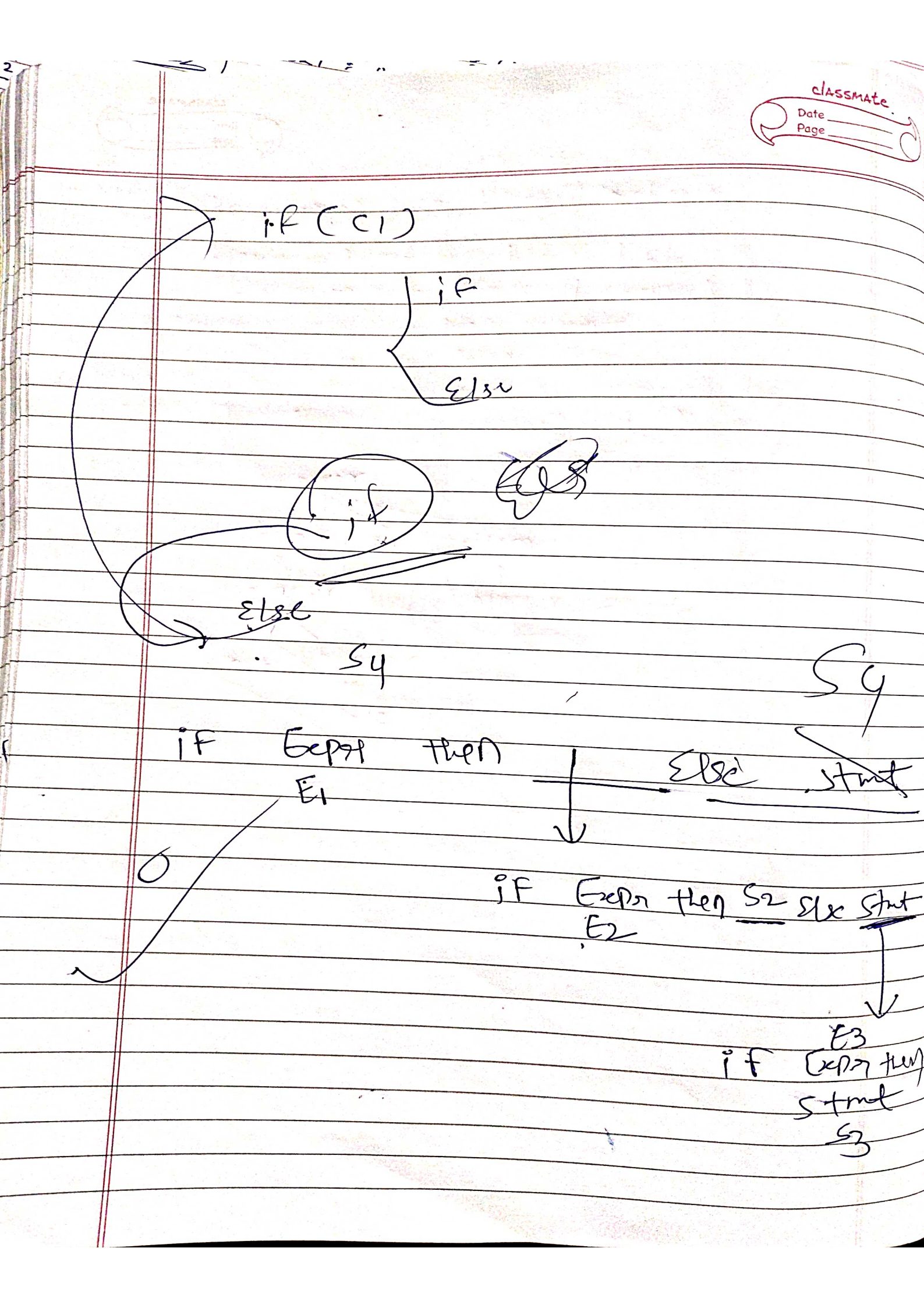
stmt → if expr then then stmt / matched stmt

matched stmt → if expr then matched_stmt
else stmt / other





(9)


if (c1)
if
Else
if
Else
S4
if Expn then
E1
Else stmt
S4
if Expn then S2 Else stmt
E2
E3
if Expn then
stmt
S3

Topics (LT)

- compilers, Interpreter, analysis & synthesis model of compilation,
- analysis of the source prog.
- ✓ phases of the compiler
- assembler, linker, loader
- ✓ compiler construction tools
- Role of the lexical analyser
- Tokens, Patterns, lexems
- Input Buffering, sentinals
- Regular Expression
- Recognization tokens,
- Transition diagrams
- Language for specifying Lexical Analyzer or lex
- Role of the Parser
- Syntax Error handling
- lexical " "
- Context free grammer
- Parse Tree and derivations (left most d., Right most d.)
- Reg. Ex. Vs Context free grammer
- Eliminating Ambiguity
- Left Recursion, left factoring
- Top down parsing
- C Recursive parser, LL(1) parser, Recursive