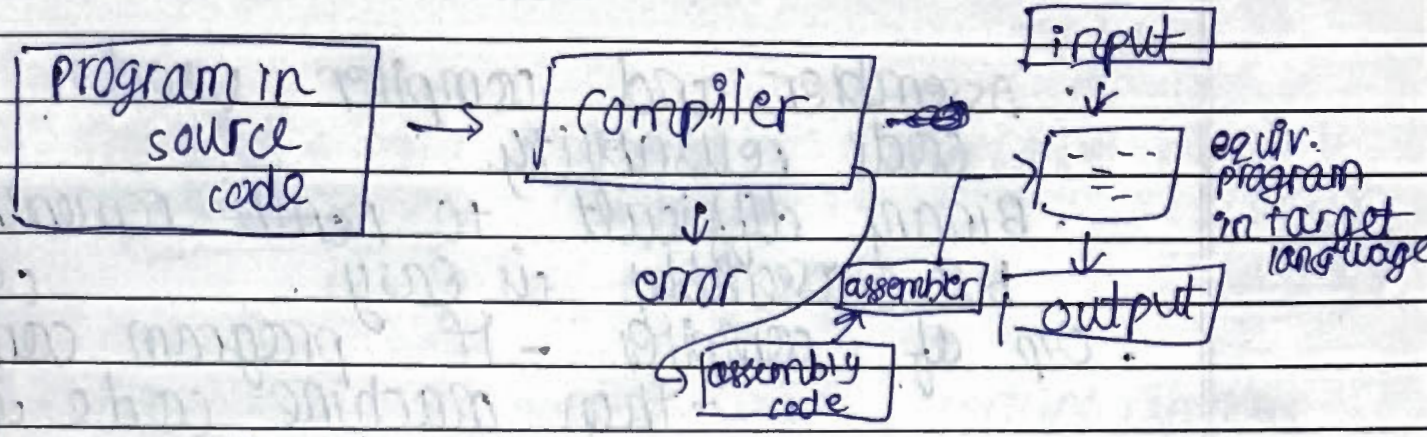
CC

Date: / /20
Page No.:

- Compiler - Translates language → target - low level computer lang.
- compiln - Transln of program written in source language into a semantically equivalent program written language. ①
  ↳ done by Translator.
  ↳ eg - binary ② assembly language
- easily understood by computer - low level language.



- compiler is a special type of translator.
- Assembly code is diff for diff. architecture.
  ↳ eg - ADD (instruction) AL BL (registers)
  ↳ still not very low for computer.

Translator → Assembler
↓ ↳ type of translator
Binary Language ↳ target - binary.

★ Python has both compiler & interpreter → Line by line in CMD.
↳ .py file

Date: / /20
Page No.:

↳ also a translator

- For compiler, target lang. can be binary or assembly

[compiler] → Assembly code

[compiler] → Binary code.

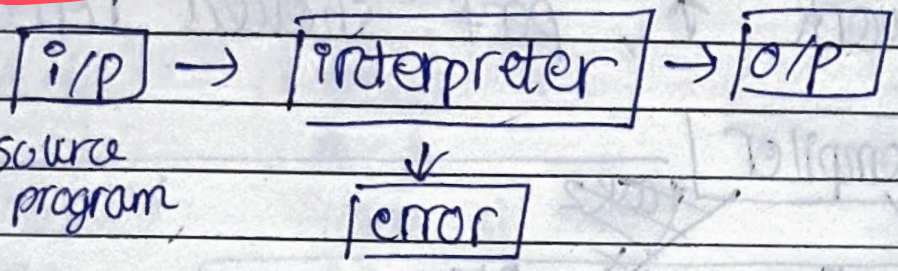
- Assembler and compiler used code reusability.
- Binary difficult to ~~remb~~ remember but assembly is easy. (syntactically)
- O/p of compiler - If program correct - then machine code is generated.

⇒ Thus, compiler translates if i/p is correct, else list of error.

→ Interpreter, another translator, doesn't generate o/p program code in binary or assembly code but translates & execute code line by line
eg- translator for BM modi. translate line by line, not all at once.
eg- ~~cc~~ command prompt

Date : / /20
Page No.:

- Note - debuggers always require interpreter



- compilers are required to improve performance. (optimizⁿ)
  ↳ req. of every code.

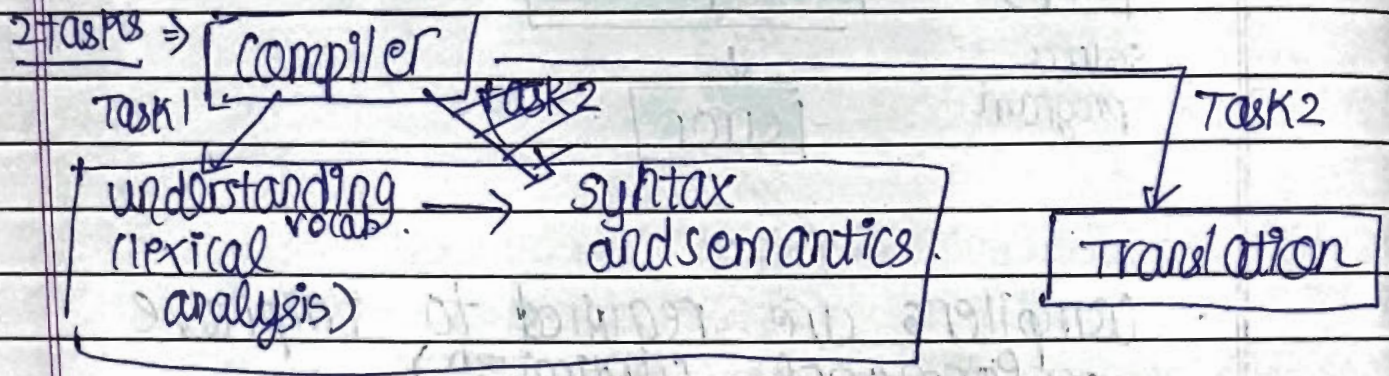
- compilers ↑ system perf. reduce system load & are needed for portability of compiled code.

* 2 main purposes - translⁿ
  - optimizⁿ of perf.

- A C exe file is machine dependant
  - may or may not run due to machine architecture.
- Performance is needed in almost all domains.

Date : / /20
Page No.:

- specific companies produce specific compilers.
- optimizn is a continuos req. Even if work ↑, perf. should remain same.



- Optimizn happens ~~after~~ after correct translm.

Date : / /20
Page No.:

Lab

Regular expressions

a|b = a or b
[ab] = a or b
[a-c] = from a to c
ac = ac
"ac" = ac
[^a-c] = except a to c
^ac = starts with ac.
ac$ = end with ac.
[a-z]+ - one or more occurences of a-z
[a-z]* - 0 or more occurences.
[a-z]? - 0 or 1 occurence.
[a-z]{2,5} = 2 to 5 occurences
[a-z]{2,} = 2 or more occurences
[a-z]/"" = look ahead character (/)

→ To make a flex.l file, ⇒ Format

%{
declar^n section
variable declar^n
funct^n declar^n
%}
%%
RE1 { Transl^n Rule }
RE2 { Transl^n Rule }
%%
user funct^n

→ [a|b] ↳ a or b or '|'
→ . = anything

Date : / /20
Page No.:

first.l

{ defn's ⇒

%{
%}
%{
%}

[_ a-z A-Z][_ a-z A-Z 0-9]* {printf ("~~comment~~ identifier");}

"if" / "else" {printf ("keyword");}

↳ longest pattern match.

⇒ first.l → [flex] → lex.yy.c

↓

[gcc]

↓

a.out

HLL = high level language
LL = Low level language

Date: 6 / /20
Page No.:

class

→ gcc produces assembly code.

HW Q: Identify assemblers & assemblers ~~twin~~ producing binary language.

Ans eg- NASM (Netwide assembler), MASM (microsoft macro assembler), GAS (GNU assemble), TASM (turbo assembler)

→

compiler	source	target
gcc	C	Binary / machine.
G++	C++	Bin. Mach.
Javac	Java	Byte code

→ source to source compiler (transpiler / trans compiler)

- Produces o/p in another lang. rather than binary (I programming lang i/p & another lang. as o/p)
eg- C++ to C transl^n.
- need - Legacy code to new code lang.
- optimiz^n
↳ parallel computat^n
- need to check which oper^n can be done parallelly
a=a+1 , b=a+1 ✗
a=a+1 , b=b+1 ✓

→ cross compiler → eg- compiler runs on PC but generates that runs on android

- used for cloud computing.
- IAB, PAS, SAS
- produce executable machine code on another ~~proof~~ platform (~~where~~ other than one where compiler is running)

→ Bootstrap compiler

- written in programming lang. that they have to compile
  - eg- C compiler written in C language

→ Decompiler

- translates exe file to high level source file which can be recompiled successfully
  - eg- source code not there, so we reverse engineer the exe.

⇒ Java is both compiled & interpreted because source code is first compiled into a binary byte code for a machine which doesn't exist (Java virtual machine).
- This byte code runs on JVM which is software based interpreter.

- compiler is faster than interpreter as CPU can execute instruct faster in Hz when everything there. 1 I/O time much slower than (CPU cycle).

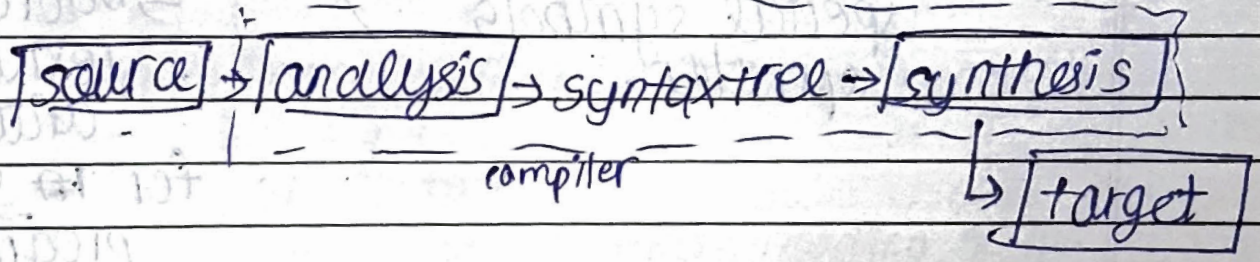
Date : / /20
Page No.: 8

Analysis-synthesis model of compil^n

- 2 parts to compil^n

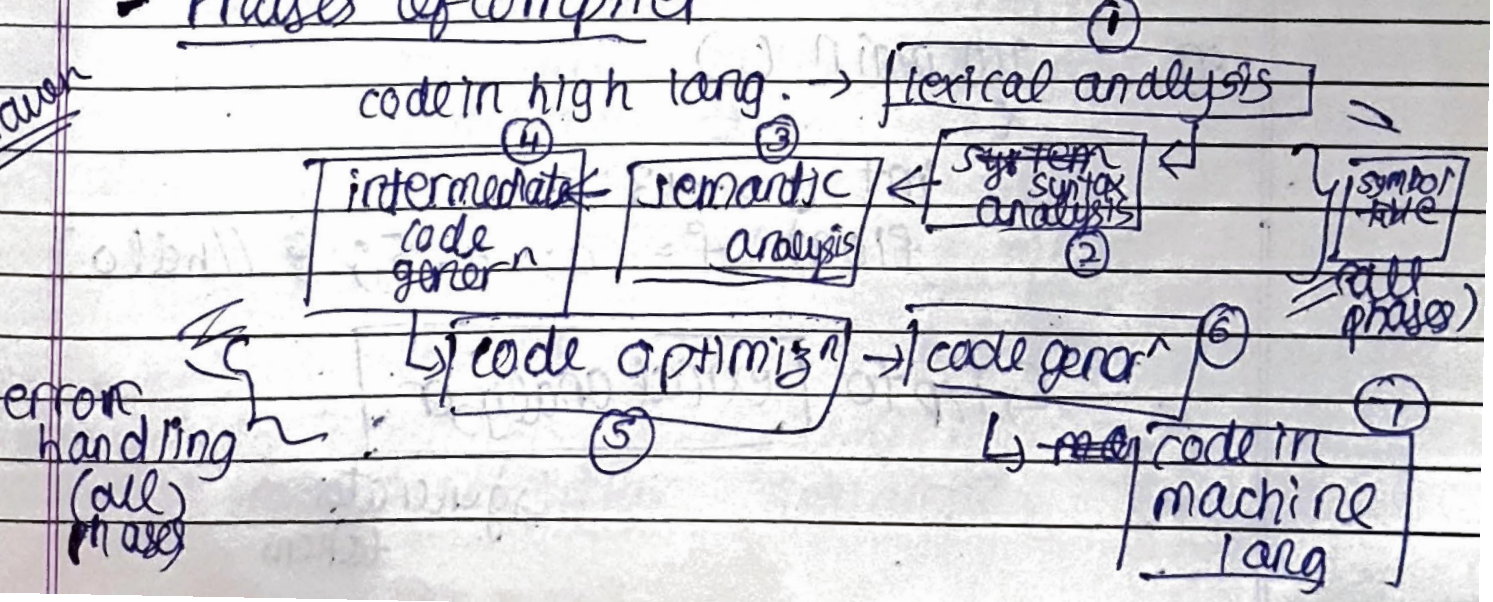
① - analysis → check if i/p program correct or not
eg - addit^n, -, *
- understand source program as per program lang. used as source program
- determines oper^n's implied by the source program which are recorded in tree structure.

② - synthesis
- takes the tree structure & converts oper^n's in target language.



Phases of compiler

redraw



tools using analysis-synthesis model :-
- editors, printers, interpreters, text formatters etc.

Date : / /20
Page No.:

eg- int a ;
float b, c ;
a = b % c ;
binary operand ↙ ↳ semantic error
(% needs int not float).

⇒ Types of tokens

- comment
- identifier
- keywords
- constants
- special symbols
- operator

} diff. for diff lang.
token analysis ↳ is done by lexical analysis for source program

eg - int main ()
{
int a = 2+3 ;
float f = a % 4.5 ; // hello ;

↳ i/p to [lexical analyzer]
↳ generates tokens

Date : / /20
Page No.:

⇒ ~~datatype(DT)~~
⇒ ~~keyword~~, ~~ID~~() {

lexical o/p ⇒ DT ID() {
DT ID = ~~const~~ const. op. ~~ID~~ const. ; ~~Hello~~
DT ID = ~~ID~~ op. ~~ID~~ ; 3
~~const~~. ID const.
→ o/p to next phase.
newline & comments are discarded

→ · Now, syntax analysis checks this o/p.
· It only understands tokens
eg- id = id + id ✓ understands
↳ not id = a + id
↳ X

· a = b + c + d can be checked using -
⇒ E + T
expression | term
↳ binary oper. needing 2 ips

→ · Now we do semantic analysis.
· symbol table stores name of id, its datatype, etc.

eg- DT ID = const. op. const
↳ details in symbol table.

** · so 3 analysis took place
- lexical, syntax, semantic.

Date : / /20
Page No.:

→ maintained for this eg.

Symbol Table

symbol name	Data type	mem. loc^	value	Type
main				func
a	int			ID
f	float			ID

id are symbol

```
eg - int main()
{ int a = 2+3;
  float f = a/.45;
  float a;
}
```

lexical analysis

⇒ DT ID () {
DT ID = CONST. OP. CONST;
DT ID = ID OP CONST.
DT ID ;
}

↳ throws multiple datatype error as already integer

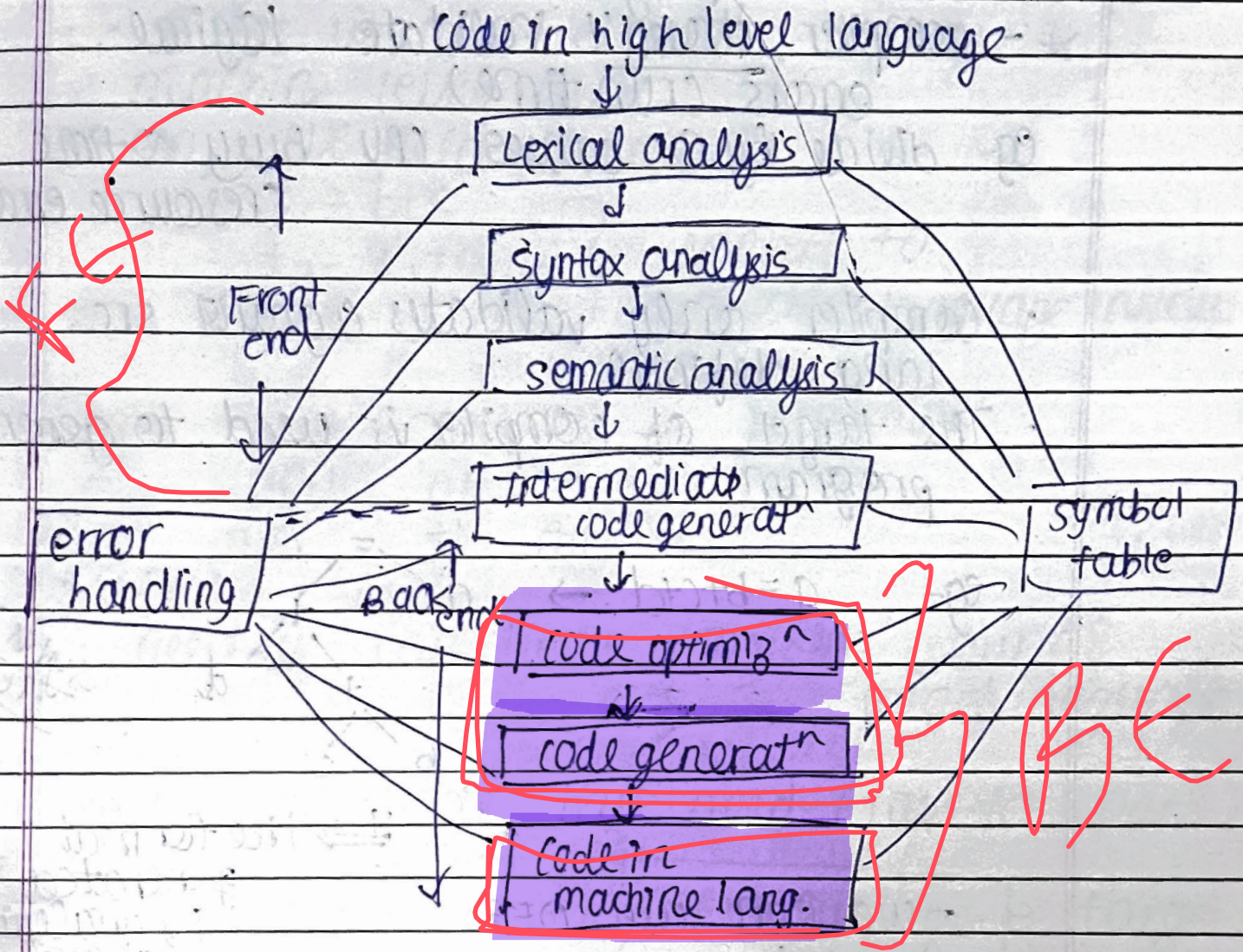
→ Now, for next phases, we can check datatypes in symbol table.

→

Date : / /20
Page No.:

redrawn

Phases of compiler



→ compiler process

src program in src lang. → analysis → format (syntax tree) → synthesis → src program translated in target lang (machine lang.)

lexical, syntax, semantics - 3 phases

Date : / /20
Page No.:

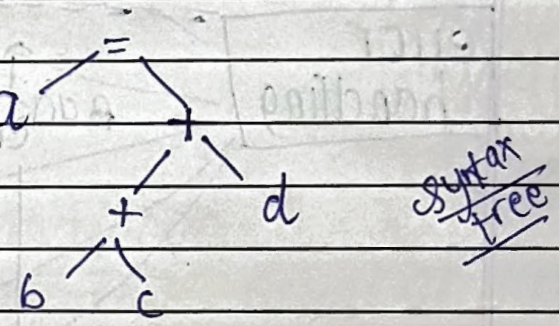
- Analysis - understand & validate source program

* - compiler doesn't validate logical errors (run time)

eg- divide by 0 causes CPU busy ∞ time (resource error)

- compiler only validates against src. lang. defin^n.
- The target of compiler is used to generate program

eg- $a = b + c + d$ →



syntax tree

⟹ the format generated by analysis phase.

- we can use DFS for this.

So first of all, $t_1 = b + c$ is processed. then $t_2 = t_1 + d$, then $a = t_2$

- Tree makes it easy to write code in any language.
- this format makes it easy to convert to target language.

synthesis

Date : / /20
Page No.:

- synthesis means gener^n
  eg- $a = t_2$ is the last step of synthesis
- But it is not enough, as we need machine level code.
  so, we need to convert -

$t_1 = b + c$
$t_2 = t_1 + d$
$a = t_2$
} convert to machine language instruct^n.

eg-
```
MOV   AH, b
ADD   AH, c
MOV   t1, AH
```
→ optimiz^n

- need of compiler- knowing source & target language.

so, we can divide this need (task to 2 parts)

src → [analysis → format → synthesis] → target
[analysis → format] : 1st need
[format → synthesis] : 2nd need.

- compiler has 2 jobs- transl^n & optimiz^n
  → making less instruct^n's.

→

Date : / /20
Page No.:

eg- a = a+1 can be done using ⇒

```
mov   AH, a
Add   AH, #1
mov   a1, AH
```

↳ optimized ⇒ mov AH, a
INC AH
} can only be implemented if machine architecture has 'INC' funct^n. (circuit available)

increment

⇒ This is machine dependant optimiz^n.

∴ 2 optimiz^n - Before machine & after machine code generated

Before machine:
- machine indep.
- use less instruct^n's.

After machine code generated:
- machine dependant.

eg- a = a+1
b = a → useless, so remove before transl^n into machine code.
c = a+2

Date : / /20
Page No.:

so, ---- [format] → [synthesis]

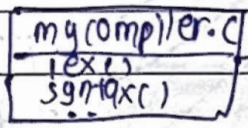
intermediate code → optimiz^n → code generat^n → op.

★ ⇒ optimiz^n is an opt^nal phase of compiler.

- phases of compiler can be seen as functions.

eg - let say 2 approaches
1) Funct^ for each analysis phase
- pipeline - ~~which~~ while lexical generates each token, it keeps on passing it to next phases.

- single pass
↳ i/p/ace where funct^ defined



· ~~phase~~ is 1 functionality ~~(sequence)~~

· clubbing all functionality in 1 phase ⇒ pass

2) multipass -

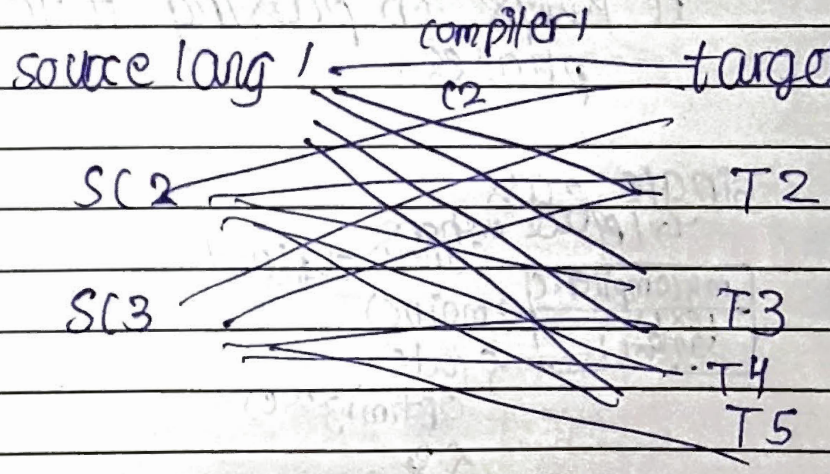
[analysis] → [format] → [synthesis]

CP1.C — CP2.C

★★ - more ~~multiple~~ time (scanning) needed for ~~multiple~~ multipass

- more RAM in single pass.
- more ROM in multipass to store.
- multipass doesn't generate code if program wrong, single pass can give intermediate o/p, as o/p of previous phase may get passed ahead.

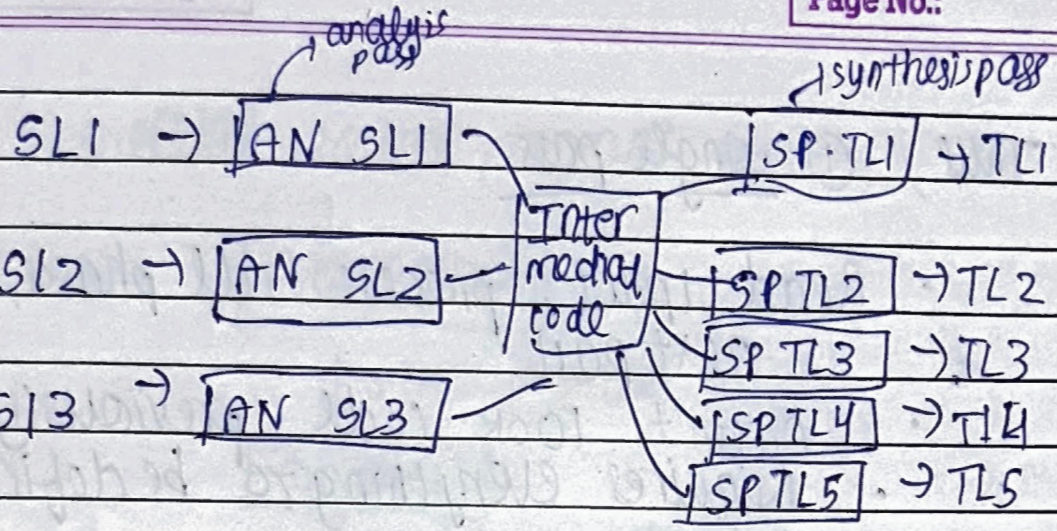
eg -



(single pass compiler) ⇒ 3X5 = 15 compilers needed.

· For every compiler, we need 1 analysis, 1 synthesis
↳ 15 total ↳ 15 total & for 15 compiler

Date: / /20
Page No.:



(multipass compiler) ⇒ 3+5 = 8 pass (reduct^ using reusability)

→ Frontend - who interacts with the user
↳ eg- how is scanf designed

→ Backend - functrality logic (not interacting with user)
eg - C is both front & backend.

⇒ Analysis pass is frontend pass
Synthesis pass is backend pass
· It is preferable to seperate frontend & backend pass.

Date : / /20
Page No.:

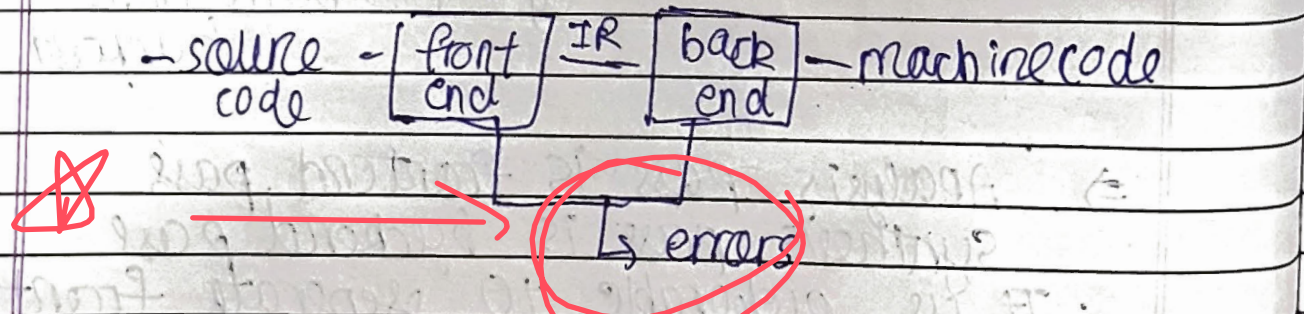
Thus, ① single pass

- Read 1part, process all phases, read next part
- doesn't look code previously processed
- Requires everything to be defined before
- Require large memory.

② multipass

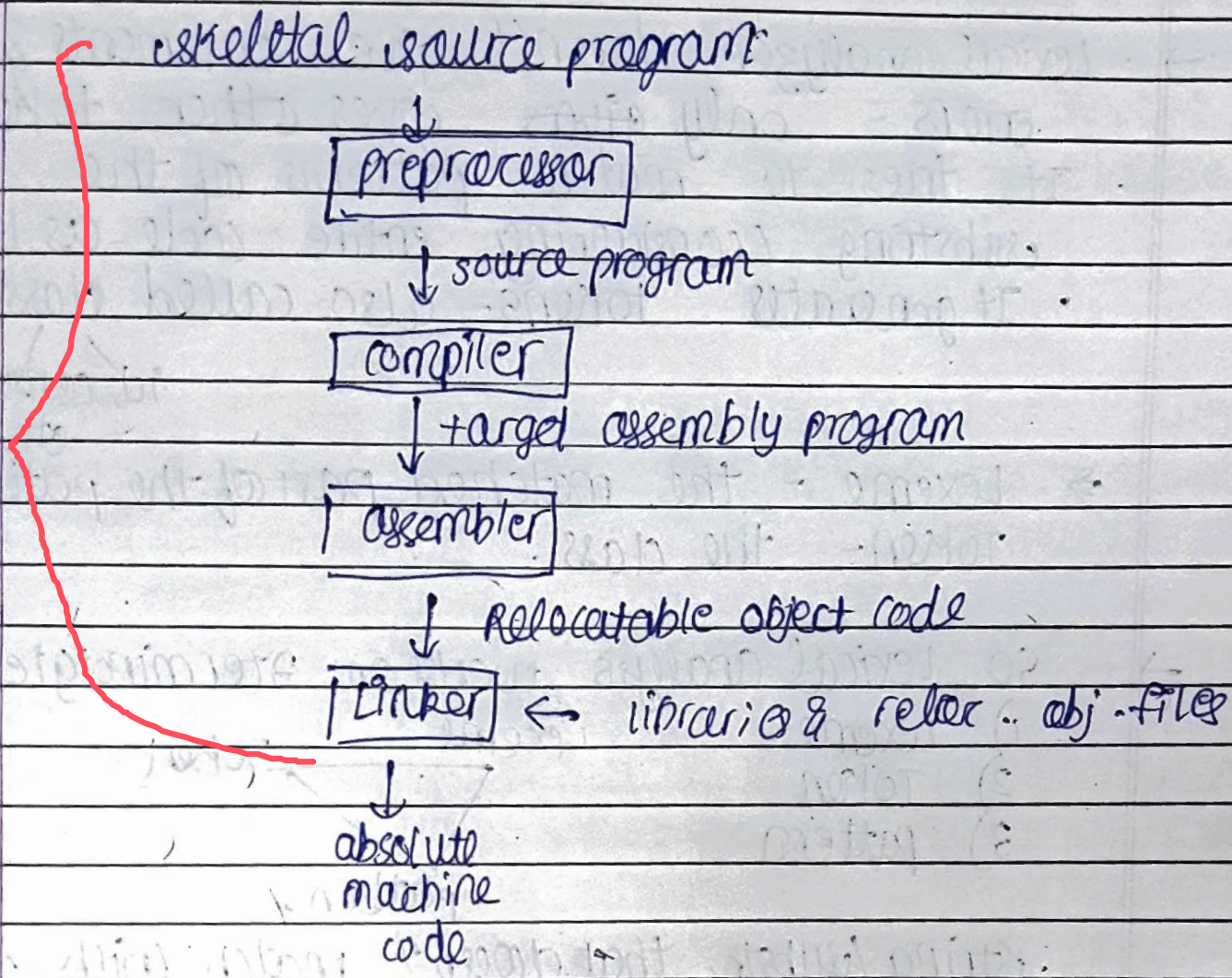
- every pass results new represent & i/p to next pass
- compiler - frontend - backend

frontend → analysis (machine indep.)
backend → synthesis (machine dependent)



- similar to compilers - preprocessors, assembler, linker (cousins of compiler)

Date : / /20
Page No.:



→ applic^s of compiler techniques

- Lexical analyzer - text editors, IRS, pattern recognit^
- syntax analyzer - SQL query processing, Kmap for circuit design
- ~~Sem~~ syntax + semantic analyzer - equat^ solver
- NLP systems

means read from here.

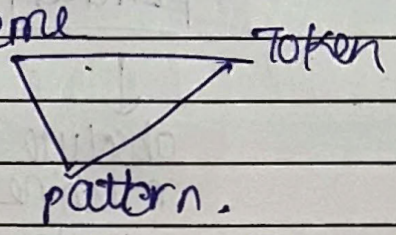
clang, MSVC, ICC - compiler that directly generate machine ~~assembly~~ code.

Date : / /20
Page No.:

→ · Lexical analyzer doesn't give comments & errors - only ~~gives~~ gives other tokens.
- It tries to match patterns of the substring (considering entire code as 1 string)
· It generates tokens - also called classes
id, const., operator etc.

⇒ Lexeme - the matched part of the pattern.
Token - the class.

→ So, Lexical analysis works on 3 terminolgies -
1) lexeme
2) Token
3) pattern



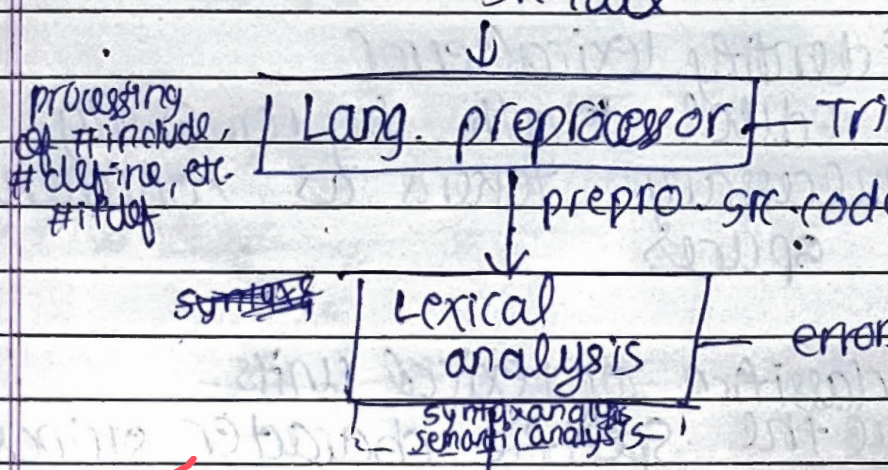
· string / substr. that doesn't match with any pattern - lexical error

* · Lexical analyzer role
- scan src. program
- translates char stream to token stream
- eliminate unnecessary tokens (eg - comments, spaces)
- enter symbols along with loc^n, type into symbol table (call symbols) but doesn't classify them.
- lexical error identificn

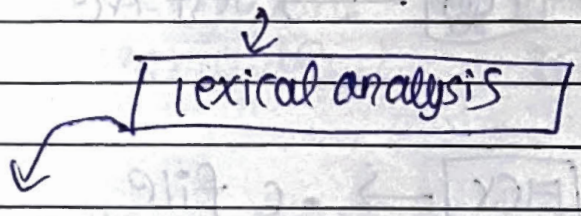
⇒ symbol is a label
Identifiers.

Date: / /20
Page No.:

- Lexical analyzer works only at 1 token at a time
- To classify token we need previous & next token.



eg - " area = 3.14 * radius * radius ; @ /* eg */ if..



⇒ area = token (id) - symbol table ✓
radius = id - ✓
↳ token val → token attribute like lexeme

o/p - <id, "area"> <assign> <num>
<multiply> <id, "radius">
<mul> <id, "radius"> <;>
@ ⇒ error as invalid

token class

comment ignored.

so, area = 3.14 * radius /* hello */ 3.14

Lexical analyzer

<id, "area"> <assign> <num, 3.14> <mul> <id, "radius"> <comment> <mul> <num, 3.14>

symbol table

sym.	token (class)	data type	offset
area	id	-	-
radius	id		

remove comment

Syntax analyzer

Date : / /20

Page No.:

- If we execute this as pass, then we get o/p of intermed. successful tokens.
- If taken as 1 phase, then it doesn't give o/p.

Rules - identify lexical error
- reduce length by removing unnecessary tokens as comments, spaces

- A token is classifier of lexical units.
- Lexemes are the specific character strings that make up a token.
- ~~Patterns a~~

⇒

test.c (src. program) → [Compiler] → test.exe

abc.l → [Flex] → .c file

eg. float const. Regex

↳ [0-9]+"."[0-9]+

↳ 4.25 ✓
.25 ✓
4.X

Date : / /20
Page No.:

eg → preprocessor statements

"#" "<include>"

^"#".*

→ multiline comment

"/*" .* "*/"

"/*" .* "\n" .* "*/"

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

eg- if as a keyword ⇒ if()
if as a variable ⇒ int if;

⇒ "if" / "C" {keyword}.
"if" {ID}

→

Date : / /20
Page No.:

→ Rules to solve matching conflicts

- Longest matching ~~pattern~~ token wins.
- Ties in length resolved by priorities
  - token specific order often defines priority.

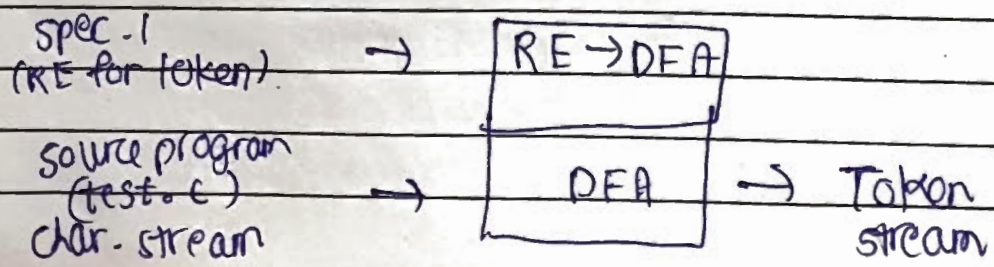
* → Design of Lexical analyzer generator

1) Approach to operate & recognize token using DFA.
2) Input buffering
3) optimiz^n using sentinel.

Lab

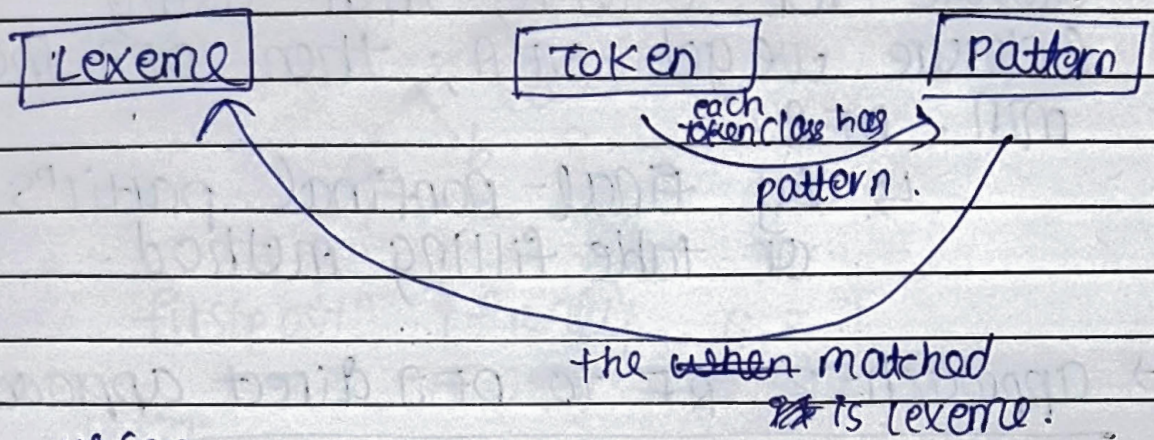
- The string token is stored in yytext.
- To store an id in symbol table, we can add ~~an~~ + use a self defined function and

→ Lexical analyzer gener. (Lex/Flex) working :-



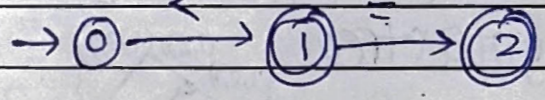
Date : / /20
Page No.:

class



we can
- use Finite automata to check regex match in a loop

eg. check <=



automⁿ	space (worstcase)	Time (worstcase)
NFA	O(|r|)	O(|r| * |x|)
DFA	O(2^|r|)	O(|x|)

Regex to DFA

approach 1 ⇒ Regex → NFA (thompson's) → DFA → min. DFA

approach 2 ⇒ (direct) Regex → DFA → min. DFA

Date : / /20
Page No.:

→ For appr. 1, we can use thompson construct^n ~~closure~~ for ε NFA, then using ε closure we get NFA, then DFA then min. DFA.

↳ by final-nonfinal partit^{n}s or table filling method.

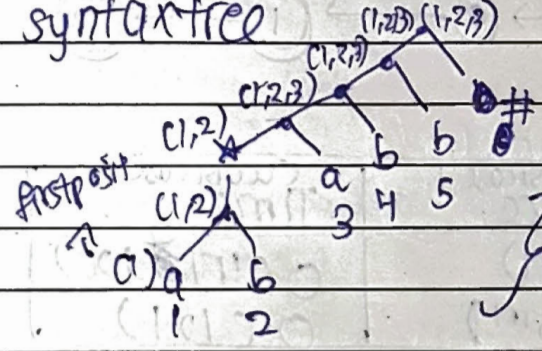
→ approach-2 RE to DFA direct approach

steps

1) append RE r with # to make accepting state

eg - r = (a/b)*abb → (a/b)*abb#

2) syntax tree



eg- class (a/b)*a → firstposit^n can be 1,2,3.

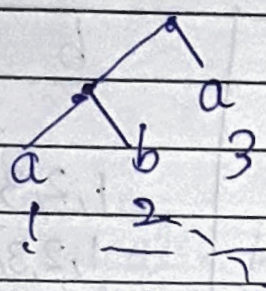
1 2 3 firstpos(1,2,3)

· Now every leaf ~~note~~ node gets a number.

Date : / /20
Page No.:

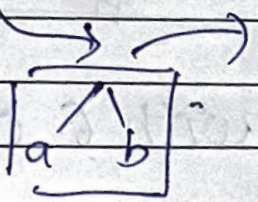
3) For every node calculate firstposit^.

eg-



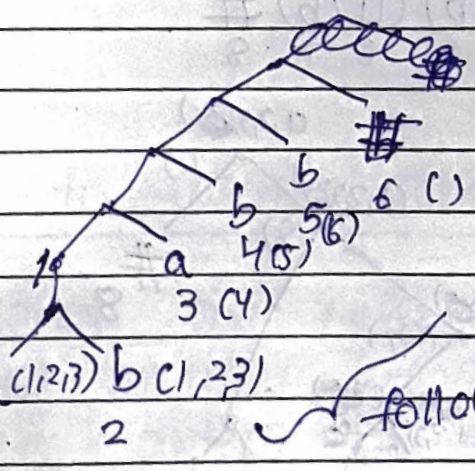
firstposit^ of only, a = 1
if only b = 2

if



firstposit^ = (1,2)

- In our main example, we see the posit^s.
- similarly, followposit^ is calculated.



→

Date : / /20
Page No.:

∴ we now make a transitⁿ table :

State	a	b
A = {1,2,3}	1,2,3,4	1,2,3
B = {1,2,3,4}	B	1,2,3,5
C = {1,2,3,5}	B	1,2,3,6
D = {1,2,3,6}	B	A

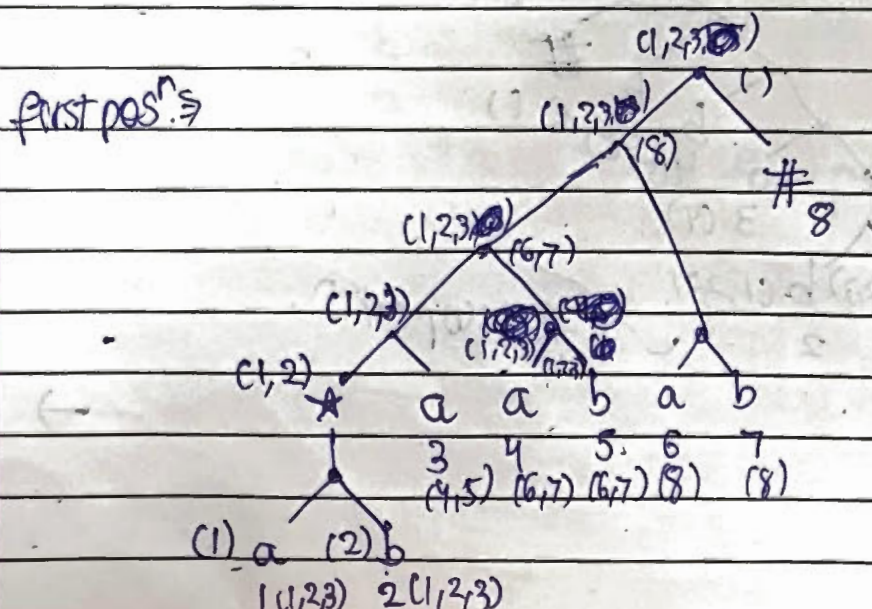
⇒ states having # are final state.

(state name ↑)

- # ⇒ 6 → states with 6 are final states
- Now we can minimize.

⇒ eg - $(a/b)^* a(a/b)(a/b)$

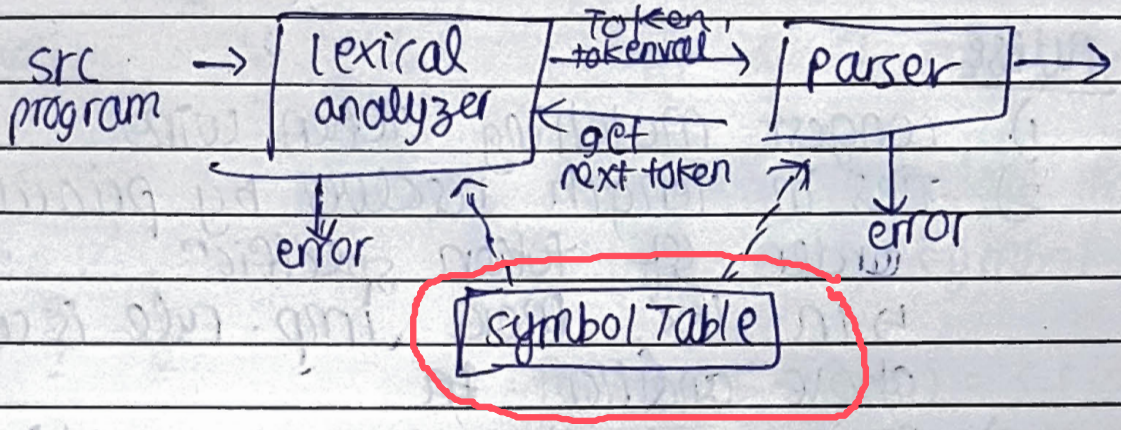
⇒ $(a/b)^* a(a/b)(a/b)\#$
1 2 3 4 5 6 7 8



Date : / /20
Page No.:

self Notes

→ lexical analysis



- Token is a classific^n of lexical units
  - ↳ a class
- Lexemes - specific char string that make up a token.
  - ↳ the matched part of token becomes a lexeme

→ How to describe Token

- Patterns are rules to describe set of lexemes belonging to a token
  - eg - pattern for id :- " letter followed by digits"

→

Date : / /20
Page No.:

- To resolve matching conflicts like:-

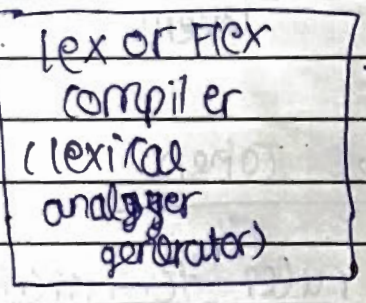
eg- elsex = 0;

① else, x, =, 0, ;
② elsex, =, 0, ;

Rules

1) Longest matching token wins
2) Ties in length resolves by priority order of token specific
   ↳ in lex, more imp. rule is write above another
3) REgex + priorities + longest matching token rule = definit^n of Lexer

→ Lex working



lex.yy.c → C compiler → a.out

i/pstream → a.out → seq. of tokens.
(Lexical analyzer)

A A A

- for | or (1) firstpos = firstpos(L) ∪ firstpos(R)

Note ⇒ Firstpos → first pos of left child (if not nullable) ← for concat, if nullable ↓ yes firstpos(L) ∪ firstpos(R)

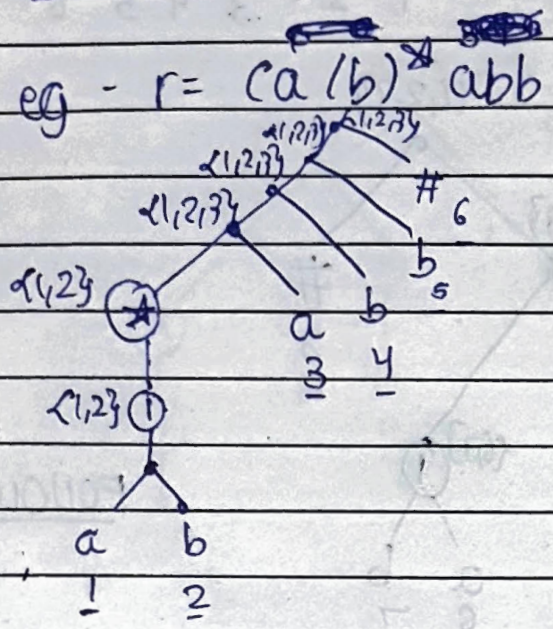
. for *, firstpos = first pos of child

Date : / /20

Page No.:

eg - RE to DFA

eg - r = (a|b)* abb ⇒ (a|b)* . a . b . b . #
1 2 3 4 5 6



. The {...} are firstpos

. Now calculate followpos of each symbol.

nullable - if it can generate null

followpos - fp(1) = {1,2,3}
fp(2) = {1,2,3}
fp(3) = {4}
fp(4) = {5}
fp(5) = {6}
fp(6) = ∅

Now table → take firstpos of root as I state

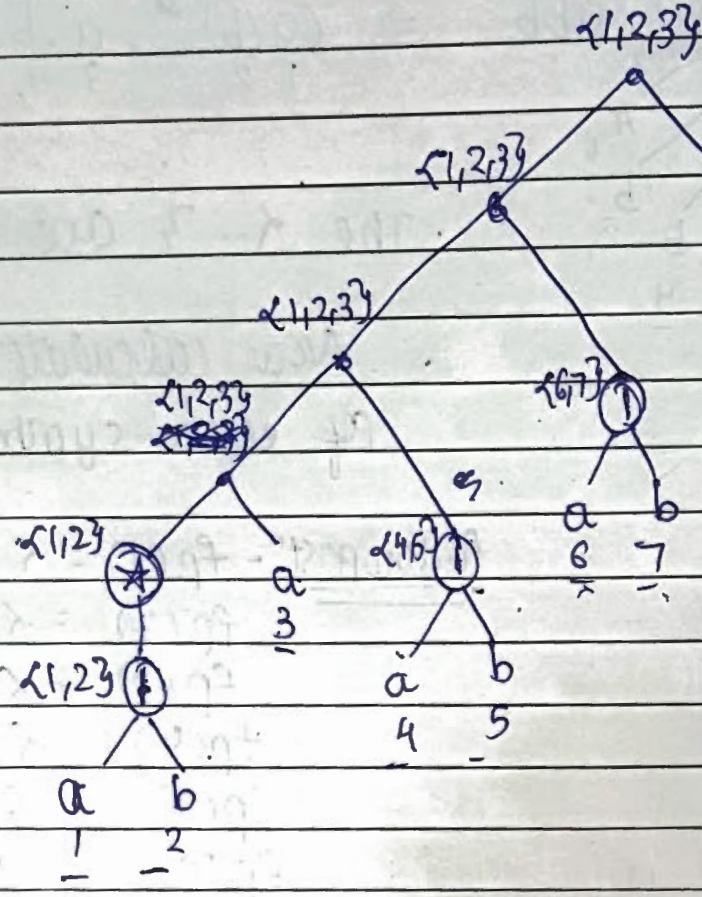
← symbols →		
state	a	b
A = {1,2,3}	{1,2,3,4}	{1,2,3} → A
B = {1,2,3,4}	B B	{1,2,3,5} → C
C = {1,2,3,5}	B	{1,2,3,6} → D
(D) = {1,2,3,6}	B	A

→ Now, we use the fp of each no in { } as a value
→ herein A = {1,2,3} 1 & 3 have same symbol so union them
→ states having # (6) are final
. so, D is final

. Now we can minimize this DFA

Date : / /20
Page No.:

eg - $(a/b)^* a(a/b)(a/b) \Rightarrow (a/b)^*.a.(a/b).(a/b).\#$
1 2 3 4 5 6 7 8



Followpos's (FP)

fp(1) = {1,2,3}
fp(2) = {1,2,3}
fp(3) = {4,5}
fp(4) = {6,7}
fp(5) = {6,7}
fp(6) = {8}
fp(7) = {8}
fp(8) = ∅

Table

State	← symbols → a	b
A = {1,2,3}	{1,2,3,4,5}	{1,2,3}
B = {1,2,3,4,5}	{1,2,3,4,5,6}	{1,2,3,6,7}
C = {1,2,3,4,5,6,7}	{1,2,3,4,5,6,7,8}	{1,2,3,6,7,8}
D = {1,2,3,6,7}	{1,2,3,4,5,8}	{1,2,3,6}
E = {1,2,3,4,5,6,7,8}	{1,2,3,4,5,6,8}	{1,2,3,6,7,8}
F = {1,2,3,6,7,8}	{1,2,3,4,5,8}	{1,2,3,6}
G = {1,2,3,4,5,8}	{1,2,3,4,5,6,8}	{1,2,3,6,7}
H = {1,2,3,8}	{1,2,3,4,5}	{1,2,3}

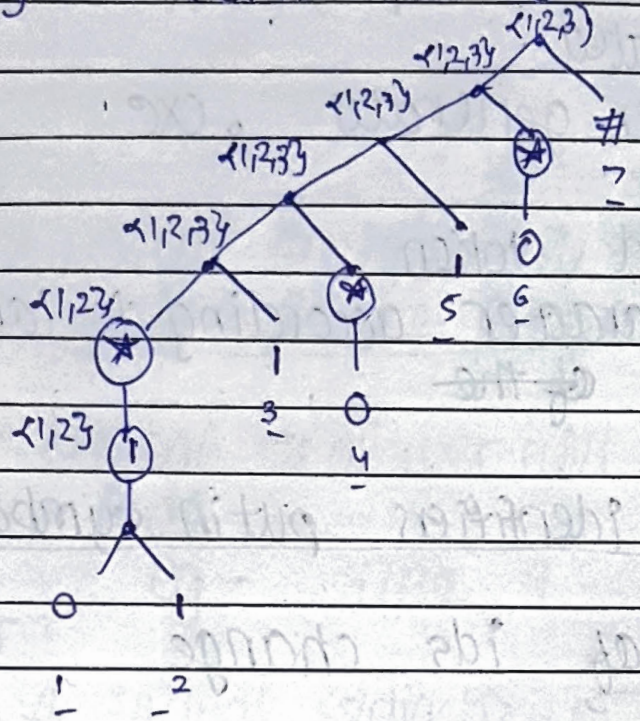
} now complete.

State having 8 (#) are final

Date : / /20
Page No.:

eg - $(0|1)^*10^*10^* \rightarrow (0|1)^*10^*10^*\#$

1 2 3 4 5 6 7



Follow pos^n

$fp(1) = \{1,2,3\}$
$fp(2) = \{1,2,3\}$
$fp(3) = \{4,5\}$
$fp(4) = \{6, \{4,5\}\}$
$fp(5) = \{6,7\}$
$fp(6) = \{6,7\}$
$fp(7) = \emptyset$

Table

	0	1
$A = \{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3,4,5\}$
$B = \{1,2,3,4,5\}$	$\{1,2,3,4,5\}$	$\{1,2,3,6,7\}$
$C = \{1,2,3,6,7\}$	$\{1,2,3,6,7\}$	$\{1,2,3,4,5\}$

Final

Date : / /20
Page No.:

Class - · When we compile a C program, .obj file is generated.
· compiler never generates .exe

- lexeme = a valid token
↳ collect of character according to long ~~of the~~

→ · Why are only identifiers put in symbol table
· only value of ids change

eg - int a,b,c;;

↳ <KW1, int> → int
<id1, ptr. to symbol table> → a
<id2, ptr.>
<id3, ptr>

ptr →	id1	type	mem-loc^n	scope

~~lexical error~~ → itn a;

↳ not a lexical error
itn will be identified as an identifier

Date : / /20
Page No.:

- Similarly, fi (a7b) ⇒ not a lexical errors
{
}

→ also called scanner - reading scanning L to R

=* Lexical analyzer is the only phase reading full source program

↳ • Lexical analyzer will attach the line no. with code
eg - Line 9 - invalid token.

Causes of Error

include <stdio.h> → to use inbuilt funcn.
define PI 3.14 → inline / lambda
void main()
{
printf();
scanf();
}

• # include allows funcn's to be used.
• Lambda / inline / macro - replaces the value
↳ eg - PI 3.14
↳ replace it (replace PI by 3.14)
↳ speed up process
↳ this is expansn of source program
• This is role of preprocessor.

☆ stdio - only has func^n name & ;, not the code.

Date : / /20
Page No.:

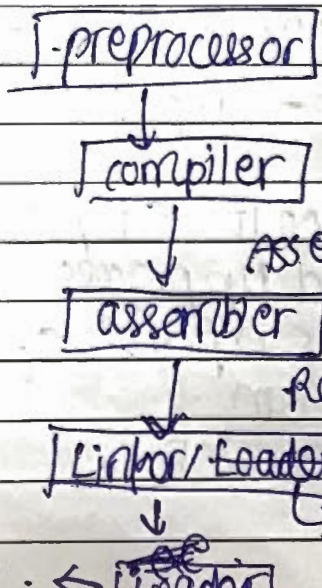
: Now compiler will get this expanded program.

- Compiler generates assembly language.
  - ↳ eg- ADD __ } JVM can run anywhere
    STA __ }
    BUN __
- Then assembler generates relocatable machine / object code.
- Then, Linker (or Loader) links the func^n declared

(eg- scanf & printf), & then also

scanf ↳ keyboard
printf ↳ moniter

links the relocatable obj. code of our program, & obj. code of monitor - printer keyboard.



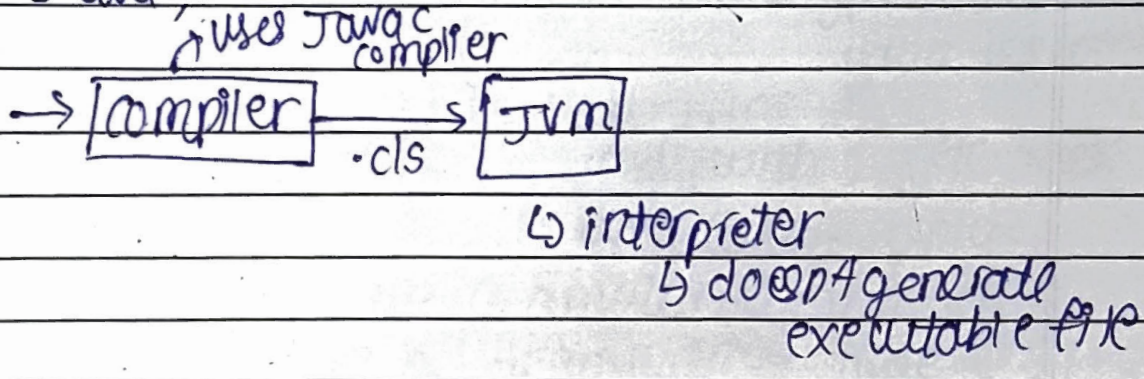
Date : / /20
Page No.:

* Java & python use both compiler & interpreter - so, they don't have .exe file.

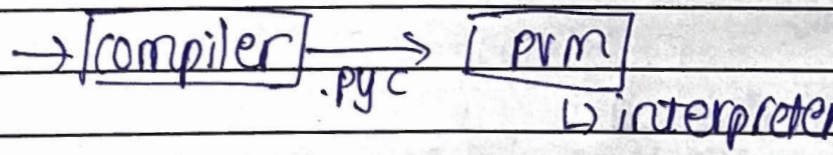
→ Thus, compiler can't function without linker, assembler, loader, preprocessor. So, they are called cousins of compiler.

* If we compile & run, ~~entire~~ all the phases are run
If we only run, then only Linker & Loader ~~will~~ run

→ In Java



→ In Python,



- So, Java & python use both compiler & interpreter.

Date : / /20
Page No.:

Lab

Practical-2

- For a grammar, check if string present or not.

eg - $S \rightarrow 0S1 / 0A / 0A1$
$A \rightarrow 011$

0011 ⇒ ✓

- S has 3 fun^{n}s.
- Take 3 variable grammar.

class

Date : / /20
Page No.:

Class → Syntax analyzer

eg- void main()
{ int a,b,c;
int int;
}

→ not a lexical error.

→ smallest unit of programing = word
↳ verified by lexical analyzer.

⇒ verify word to check program

- once verified, words generate sentences.
- syntax analyzer checks syntactic struc. of the statement.
- So, in C, we have ⇒ datatype variable;
↳ this is syntax
every line ends with ; etc. these are rules of syntax.

⇒ these rules are grammar of the lang.

- we have 4 types of grammar -- type 0, 1, 2, 3
↙ free/unrestricted ↓ context sensitive ↓ context free ↓ regular grammar

- Here, we only use context free grammar
↳ used by syntax analyzer.

Date : / /20
Page No.:

- So, syntax analyzer works with CFG as CFG works with grammar which can be verified.

⇒ G = (V, T, P, S)
- V: variables
- T: terminals (Σ = {a, b})
- P: production rule
- S: start symbol

- The grammar uses process of derivation

★ Deriv^n is process of deriving a string from starting symbol.

- Parsetree - A graphical represent of deriv^n process using tree structure
  - ↳ root node at start (start sym.)
  - ↳ intermediate vertex - nonterminals
  - ↳ leaf node - terminals.

S → DL;
↳ product

S ⇒ DL
↳ derivation

★ · use '|' for seperat^n
⇒ D → int | float

→

Date : / /20
Page No.:

- $S \rightarrow DL;$

Grammar ⇒ $D \rightarrow int \mid float$

$L \rightarrow L, L \mid R$

$R \rightarrow a \mid b \ldots$

Deriv^n ⇒

- $S \Rightarrow DL;$
- $\Rightarrow int\ L, L;$
- $\Rightarrow int\ a, L, L;$
- $\Rightarrow int\ a, b, c;$

sentential form

Deriv^n

sentence

→ sentence - It is a collect^n of terminal symbols during the deriv^n process.

So, if $S \overset{*}{\Rightarrow} G$ meaning if using (* means 0 or more) 0 or more steps if from S we generate a string G only of ~~variables~~ terminals, then we get a sentence.

→ sentential form - Deriv^n steps that may contain nonterminals is known as sentential form.

So, $S \Rightarrow DL;$

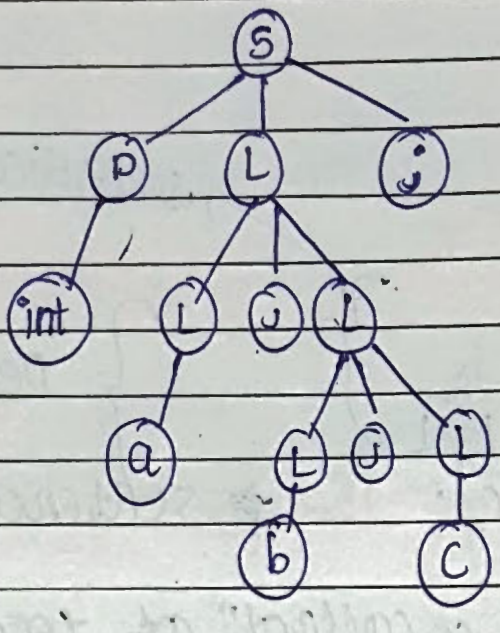
$\Rightarrow int\ L, L;$

$\Rightarrow int\ a, L, L;$

} sentential ~~sentinal~~ form.

→

Date : / /20
Page No.:



⇒ This is a parse tree

- now, if ⇒ int a,b,c & no semicolon, 1st product" itself won't ~~execute~~ execute
- If [a,a;] then DL;
  ↳ not present.

3 errors gen.

⇒ 1st type of error generated is called misspelled keyword error

or

⇒ If no ; ~~or~~ ; then we get missing / invalid punctuat" marks.

⇒ operator missing error can also be generated

Note- only lexical & syntax analyze phase can update symbol table

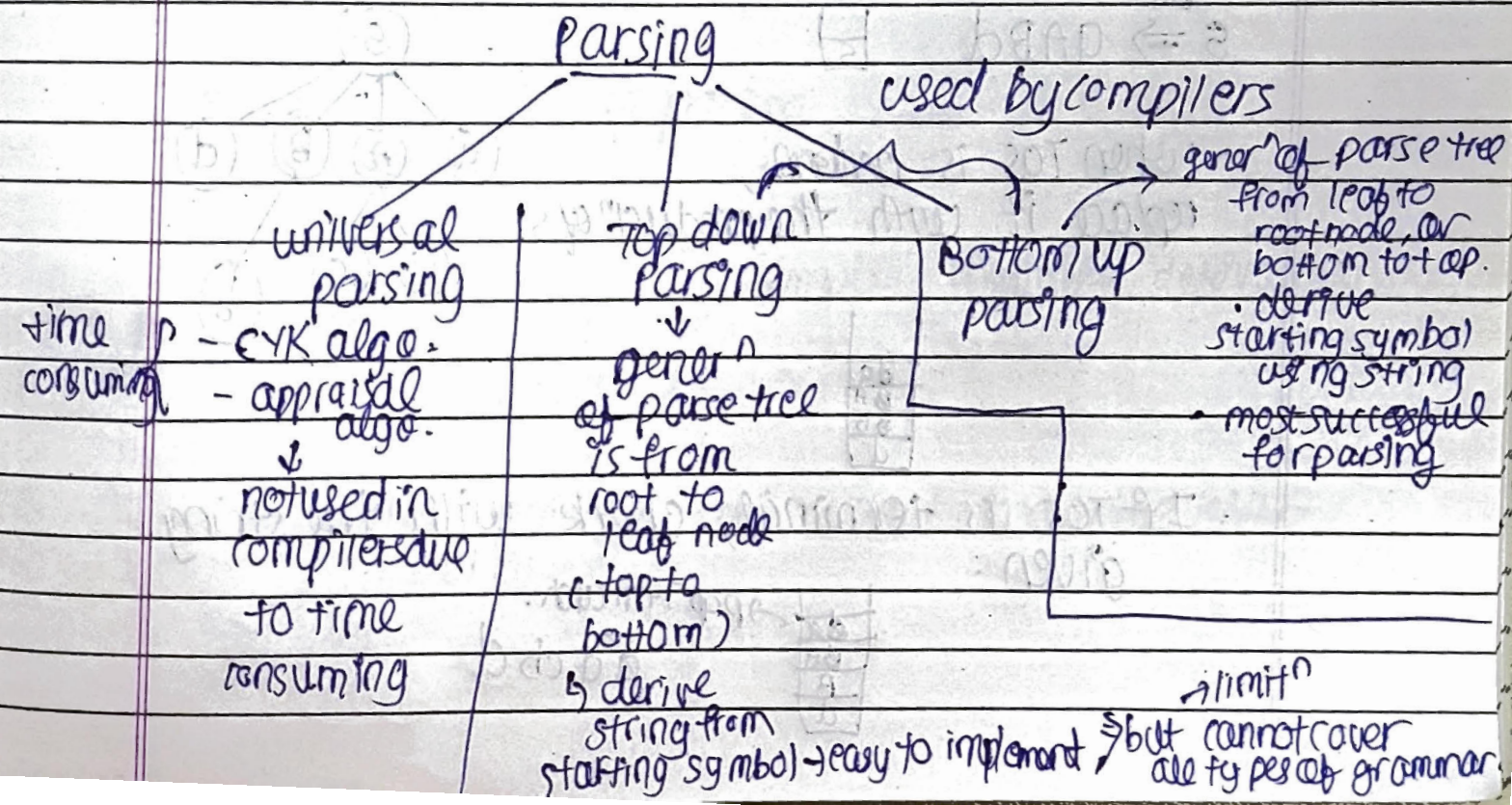
Date: / /20
Page No.:

- if ⇒ int int ;
Lexical analyzer sends them to syntax analyzer, but syntax error - invalid identifier.

→ - so, if ⇒ int a, b, c
lexical enters ⇒ a, b, & c in symbol table
syntax enters ⇒ int datatype of a, b, c

- syntax analyzer also scans data LtoR.

→ Parsing - deriving the string using the parse tree.
- there are 3 types of parsing.



Date : / /20
Page No.:

- Left most deriv^n = The left most variable is derived first
- Right most = vice versa.

→ Top down parsing uses left most deriv^n, & bottom up parsing uses right most deriv^n in reverse.

Use of stack →

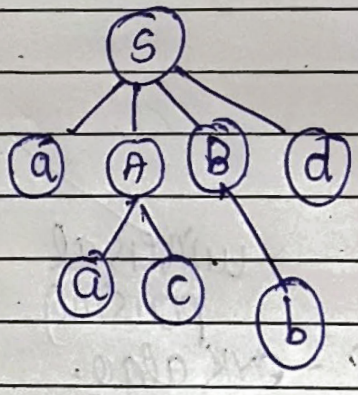
$S \rightarrow aABd$
$A \rightarrow ac$
$B \rightarrow b$

string = aacbd

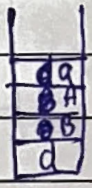
→ For top down parsing,

push S in stack

$S \overset{lm}{\Rightarrow} aABd$



- When TOS is nonterm., replace it with the product of S



- If TOS is terminal, check with the string given.

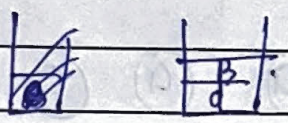
→ pop if match.

aa.cbd

Date : / /20
Page No.:

now, as TOS = A ⇒ nonterm., ~~expand it~~.
replace it with producer = ac
$A \overset{lm}{\Rightarrow} ac$

acbd



replace B by b

$B \overset{Lm}{\Rightarrow} b$

bdc

⇒ stack & string over.
· So, string is a member

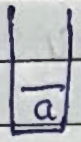
⇒ For bottom up parsing,
aaabd

$S \overset{Rm}{\Rightarrow} aABd$
$\overset{m}{\Rightarrow} aAbd$
$\overset{m}{\Rightarrow} aacbd$
} simple Rightmost derivation

- While parsing, we want to derive start symbol of string

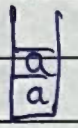
→

Date : / /20
Page No.:

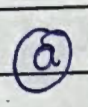


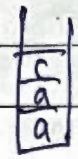
· a not there on right side of reduct's

· If we get a match on right side, reduce it it is called a handle

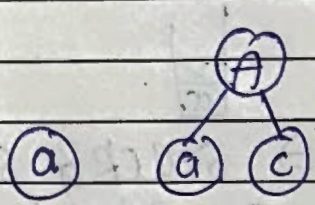


· aa not there on right side of product's

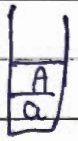




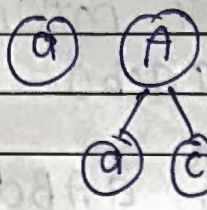
· ac present of right side.
· we are reducing so =>



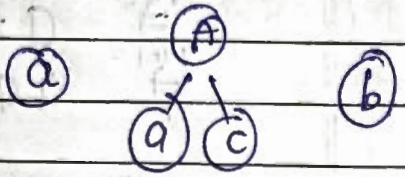
→ handle



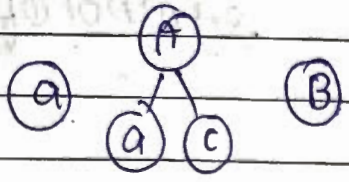
· aA not on right side



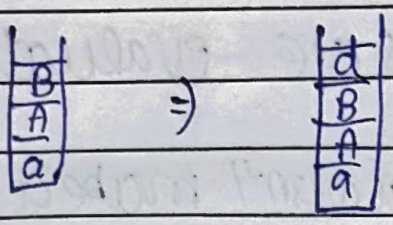
· aAb
↳ b on right side



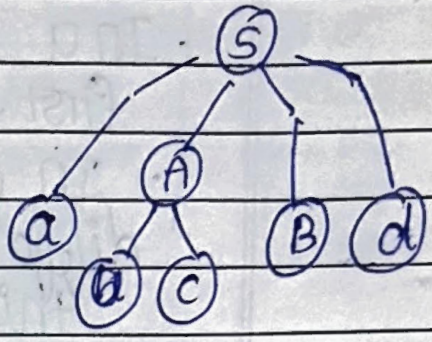
↳



Date : / /20
Page No.:



aABd present on right side



S ⇒ we reduced entire thing by S.

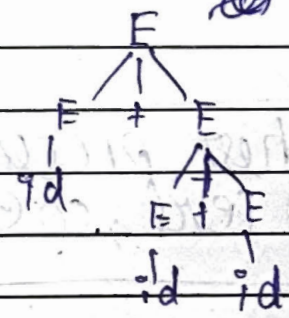
we used this as ⇒ reverse of rightmost derivn - as S was derived in the end,

L) eg- in ~~left~~ rightmost B was derived first, but here B was derived last.

class ⇒ E → E+E|id

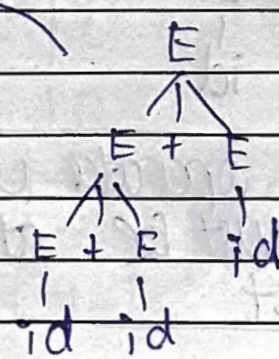
derive id+id+id ⇒ eg- 2+3+5

- Here associativity is the problem



correct

or



This is correct as '+' is left associative

- id+id+id

solve first

E ⇒ E+E
E ⇒ id + E+E
E ⇒ id+id+id

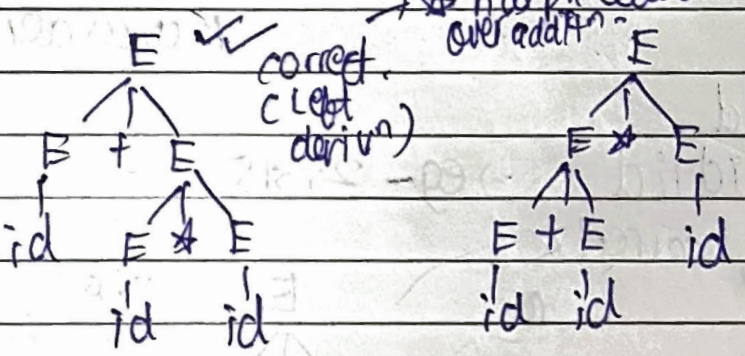
E ⇒ E+E
E ⇒ E+E+id
E ⇒ id+id+id

Date: / /20
Page No.:

- In a tree, the leaf nodes are evaluated first.
- In addition this doesn't make a difference, but if 2 power 3 powers this makes diff.. ~~gramma~~ answers for both trees.
- This grammar is ambiguous
  ↳ more than 1 tree.

eg. $E \rightarrow E+E \mid E*E \mid d$
id+id*id

} here, precedence is not taken care of



** · So, operator with highest precedence must be at lowest level, so evaluated first

→ How to make an ambiguous gramman unambiguous

· Steps to make unambiguous grammar

So, if we take care of associativity & precedence, we can make the gramma

- If we make grammar unambiguous, if operator is left associative, introduce a new nonterminal on Right side, if right asso., then new terminal on Left side.

Date : / /20
Page No.:

→ write first as lowest precedence.

→ ! E → E+T | T
T → T*F | F
F → id

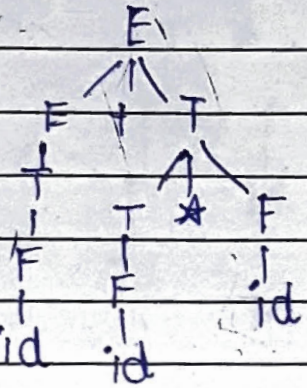
- we can take care of precedence by diff levels

we can't write T*T as again ambiguous.

id has highest precedence, hence @ lowest level

eg- id+id*id.

→ E is first as we want left side to be evaluated first.



left associative
a+b+c
↳ evaluate 1st-
a-b-c
↳ evaluate first.

Q- E → E+E | E*E | E-E | E/E | E↑E | id.

E → E↑A | A
A → A/B | B
B → C* | C

E → E+T | E-T | T
T → T*C | C
C → C/D | D
D → D↑E | E
E → id

E → E+B | E-B | B
B → B*C | C
C → C/D | D
D → D↑M | M
M → (E) | N
N → id

Q - above Q with | (E)

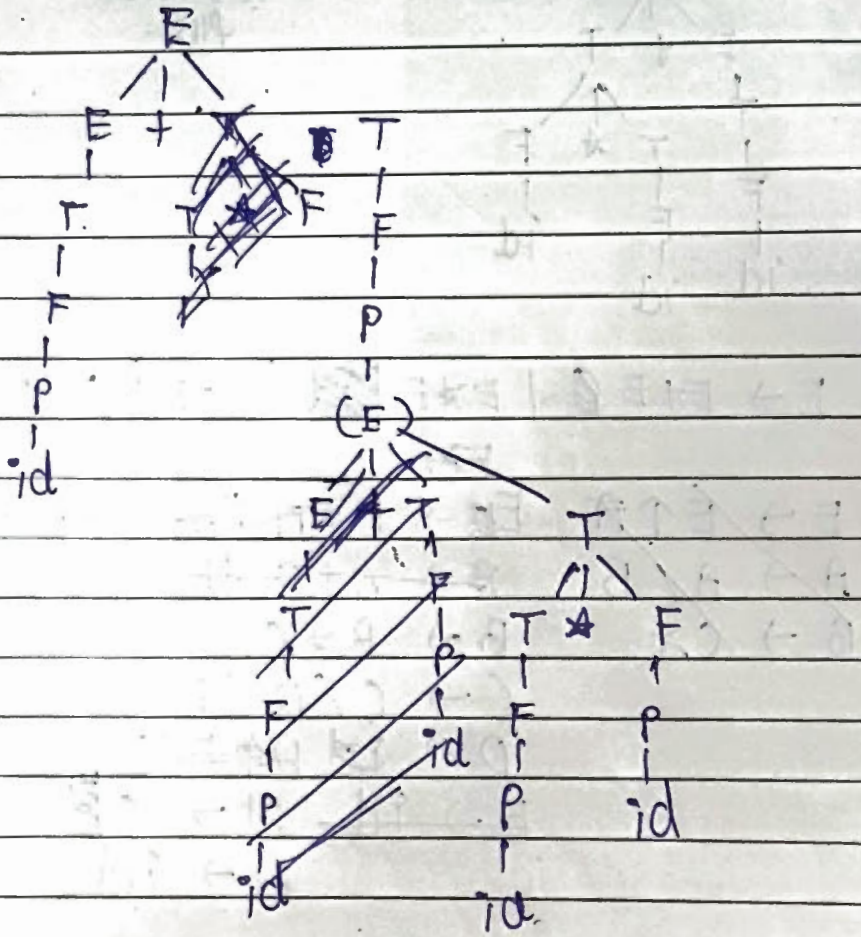
↓
brackets.

→

Date : / /20
Page No.:

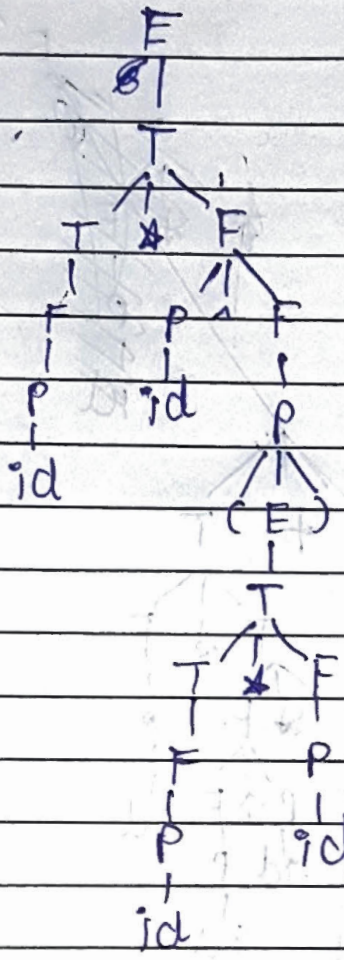
Q, $E \to E+T \mid T \mid E-T$
$T \to T*F \mid T/F \mid F$
$F \to P \uparrow F \mid P$
$P \to (E) \mid id$

string = id + (id * id)



Date : / /20
Page No.:

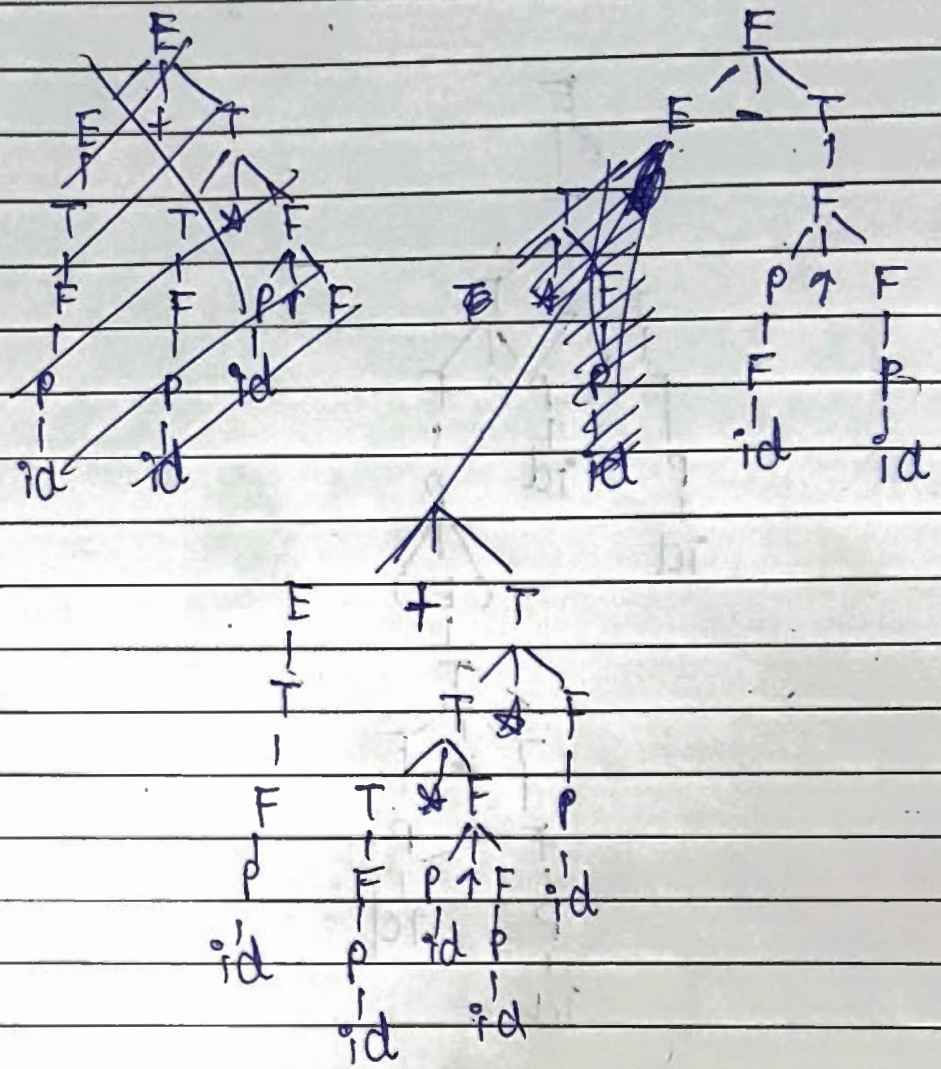
string ⇒ id*id^(id*id)



Date : / /20
Page No.:

Trick
** ↓ Traverse till right, then while returning, split based on lowest precedence operator (check till end on return)

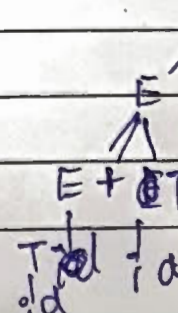
eg- id + id * id ↑ id * id - id ↑ id.



→ self notes

1+2+3 ⇒ left associative

so, we grow tree on left side ∴ left recursive grammar



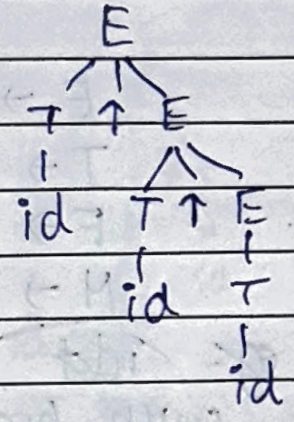
E → E + T | T

T → id

Date : / /20
Page No.:

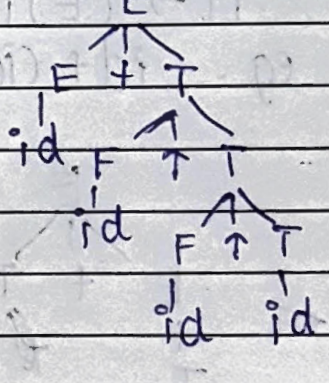
⇒ $2 \uparrow 3 \uparrow 2$ - power is right associative
- $2^{3^2} = 2^9$ ↳ right recursive grammar

$E \rightarrow T \uparrow E \mid T$
$T \rightarrow id$



⇒ $2+3 \uparrow 2 \uparrow 4$

$E \rightarrow E+T \mid T$ → left recurs^
$T \rightarrow F \uparrow T \mid F$ → right recurs^
$F \rightarrow id$



eg - $A \rightarrow A \$ B \mid B$
$B \rightarrow C @ B \mid C$
$C \rightarrow C \# D \mid E$
$E \rightarrow id$

⇒ here, associativity of
- A = left, B = right,
C = left

precedence
⇒ # > @ > $

Date : / /20
Page No.:

class eg. resolved

eg. $E \to E+E \mid E*E \mid E-E \mid E/E \mid E \uparrow E \mid id$.

⇒ $E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

$F \to M \uparrow F \mid M$

$M \to id$

eg. ~~id+id~~

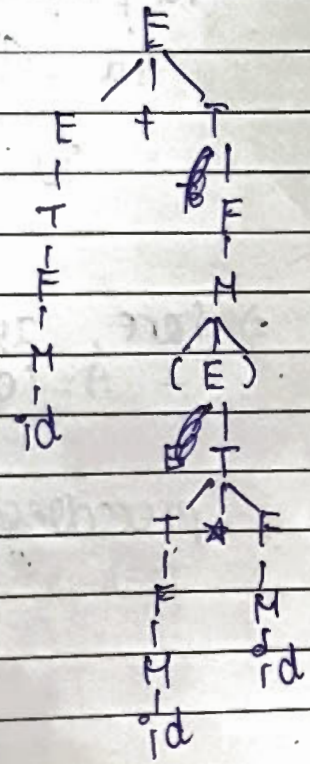
with brackets,

! same

$F \to M \uparrow F \mid M$

$M \to (E) \mid id$

eg. $id + (id * id)$

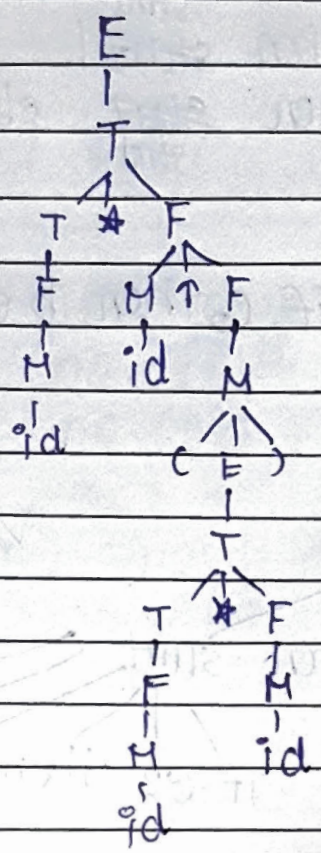


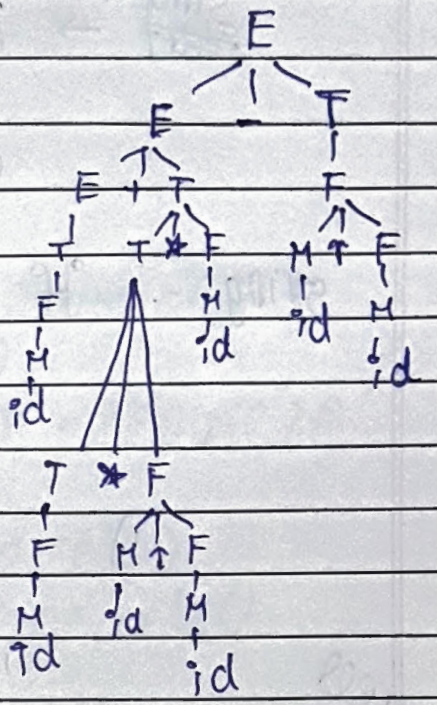
Date : / /20

Page No.:

split based on lowest precedence from left to right

eg - (id*id^(id*id)) ; id+id*id↑id*id-id↑id





- we also have inherited ambiguous grammar → eg - $L \to a^n b^n c^m d^m \mid a^m b^n c^n d^n$
- ambiguous → convers^n → unambiguous.
  - ↳ not removable → inherited ambiguous grammar

$a^n b^n c^n d^n$ inherited ambiguous grammar

Date: / /20
Page No.:

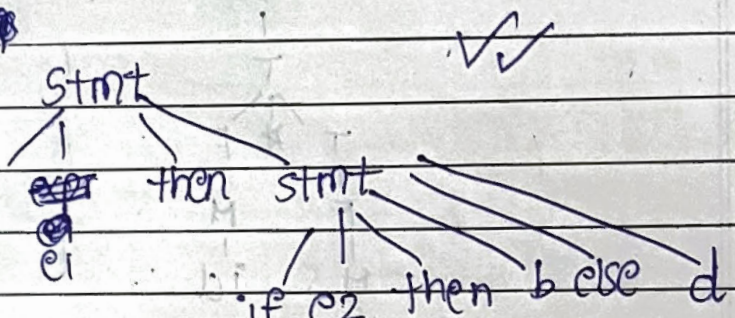
Ques → Dangling else problem

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

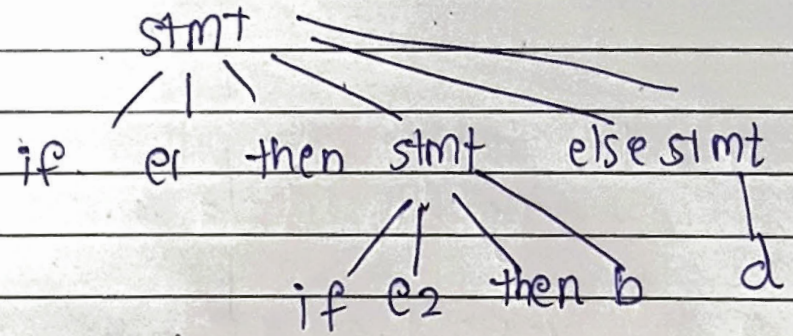
string:- if e_1 then if e_2 then b else d

2 trees

1)



2)



⇒ 1st is correct, as if is always attached with nearest if.

Date : / /20
Page No.:

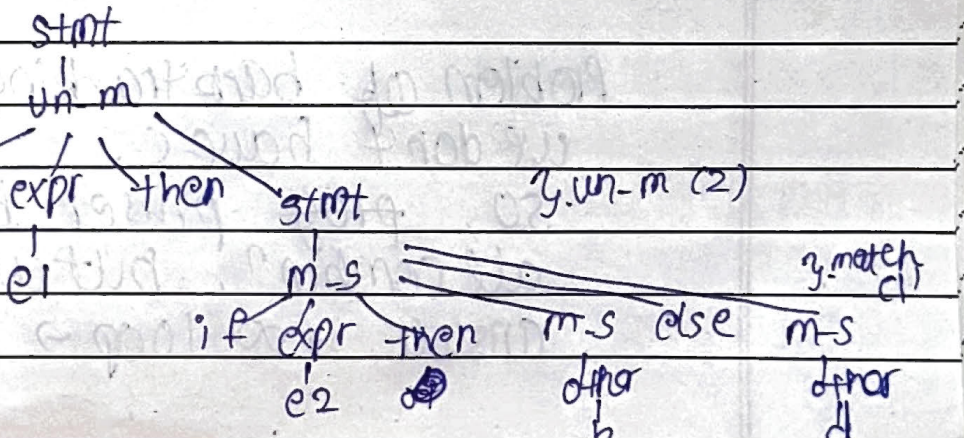
- This is dangling else problem, as we don't know which else is attached with which if.
- The grammar is ambiguous.

solution to the problem

; There are 2 types of if statements
⇒ no.(if) = no.(else)
⇒ no.(if) ≠ no.(else) → simple 'if'
↳ m-s
match stmt. ⇒ no.(if) = no.(else)
unmath stmt ⇒ no.(if) ≠ no.(else)
↳ un-s

⇒ stmt → m_s | un_s
m_s → if expr then m_s else m_s | other. (equal if else)
un_s → if expr then stmt.
if expr then m_s else u_s | other

eg- if e_1 then if e_2 then b else d.

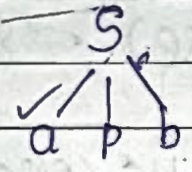


Date : / /20
Page No.:

→ **Recursive Descent Parsing** → simplest form of top down parsing

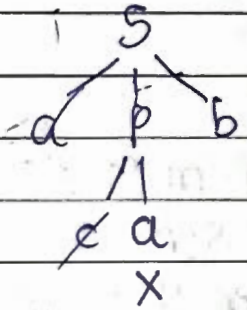
eg ⇒ S → aPb
P → c | ca
string ⇒ acb.

start from starting symbol ⇒



acb

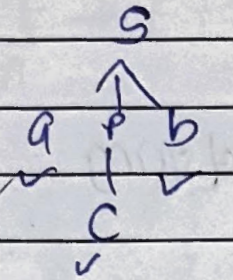
if we choose ⇒ 'a'



a c b

- now as we do not get a match, we do backtracking.
- Problem of backtracking ⇒ let P = ca|cb|cc... we don't have c.
- so, parser backtracks & tries all combin?, but we don't get answer. → ∞ loop → parser crash.

Date : / /20
Page No.:



acb

after backtracking.

- There are 2 more problems.
  1) Left recursn
     ⇒ S → Sa
     ⇒ ∞ loop
  2) Left factoring

P → ca | cb | cc

same prefix 'c'.

so which to choose?

- Because of problems with ambiguity, leftrecursn & left factoring, we don't use Recursive descent parsing.

⇒ - To solve leftrecursn,
problem ⇒ A → Aα / β,

ans. ⇒ A → βA'
A' → αA' | ∈

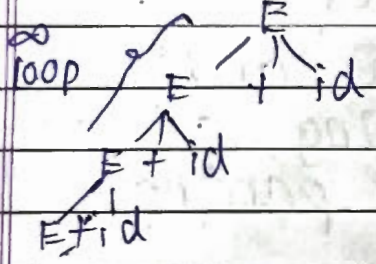
Date : / /20
Page No.:

eg - $E \to t + E'$
$E' \to +E \mid \epsilon$

let string ⇒ t+t $ → end of string

→ **Recursions**

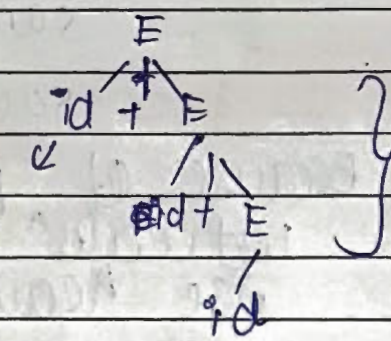
Recursion
- Left
  $E \Rightarrow E+id \mid id$
  Left terminal or left side
  eg - id+id+id



loop

here parser doesn't know how many id's are generated

- Right
  $E \to id + E \mid id$
  eg id+id+id



we now id done

} right recursion not a problem.

Date : / /20
Page No.:

eg- $E \rightarrow E+t \mid t$
$\quad\quad A \quad\; A\ \alpha \quad \beta$

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

$\therefore \quad E \rightarrow tE'$
$E' \rightarrow +tE' \mid \epsilon$

*class → Non deterministic grammar

- Here the parser gets confused when looking at the productions.

$S \rightarrow aAr$
$A \rightarrow b \mid bc$

more than 1 option - non determinism
↳ confusion for parser.

- So, we need to apply left factoring
- Left factoring - grammar technique to make prodn suitable for top down parsing.

Step - If there is a grammar with prodn $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, then we can rewrite as :-

$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

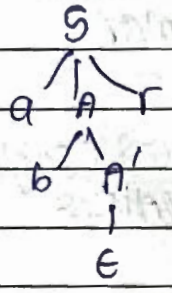
→ Keep common prefix & introduce new terminal.

Date : / /20
Page No.:

∴ If $S \to aAr$
$A \to b|bc$

⇒ $S \to aAr$
$A \to bA'$
$A' \to c|\epsilon$

now to get abr,



abr ⇒ parsed ✓✓

· For top down parsing, we need 3 condit's -
1) grammar - unambiguous
2) grammar - no left recursion ~~& no nondet~~
3) grammar - no nondeterminism (left factoring problem)

→ Predictive parsing

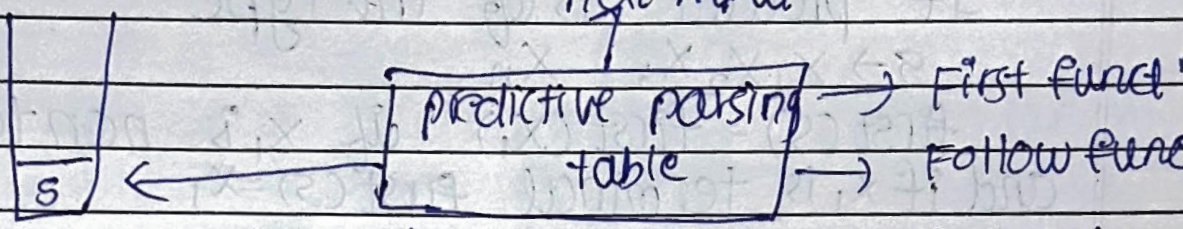
· It is parsing technique in which the select^n of the produc^n is based on the input symbol.

Date : / /20
Page No.:

$S \rightarrow a \mid b$

parser predicts which is suitable / correct production.

- we have a stack & a predictive parsing table.



- This method uses TOS & the next input.
- This predictive parsing is of 2 types
  - 1) Recursive predictive parsing → X not used.
  - 2) Non recursive predictive parsing
    - ↳ maintaining the stack implicitly. (we saw in top down parsing)
    - ↓ we will only study this.
- we use 2 func^n's to create this table
  - :- 1) First function
  - 2) Follow function

→

Date: / /20
Page No.:

1) First funct^n

- If the product^n is of the type $S \to a$ then
  first(S) = a → nonterm. → terminal
  and first(a) = a

(if we expand S, we get a first)

- If product^n is of the type
  $S \to X_1 X_2 X_3 \dots X_n$
  first(S) = first(X_1) if X_1 is nonterminal.
  and if X_1 is terminal first(S) = X_1

eg. $S \to ABC$ first(S) = first(A) = a
$A \to a$

→ and first(S) = { first(X_1) ∪ first(X_2) if $X_1 \Rightarrow \epsilon$ (if X_1 is nullable) }

eg. $S \to ABC$
$A \to a \mid \epsilon$
⇒ abc or bc ∴ first(S) = a or b
↳ if A = ϵ

here first(A) = a or ϵ

2) Follow funct^n

- If product^n is of the type $S \to \alpha$ and S is the starting symbol,

Date : / /20
Page No.:

follow(S) = { $ }

- If production is of the type $S \to \alpha B \beta$

  follow(B) = first(β)

  we look on right side

- If β is nullable, ($B \Rightarrow \epsilon$)

  Follow(B) = follow(S) ($S \to \alpha B \beta$)

⇒ eg - $S \to aAB \mid ASB$

$B \to b \mid \epsilon \mid S$

follow(S) = { $, b }

follow(B) = follow(S) = { $, b }

follow(A) = { b, $ }

- If production of the type -

$S \to \alpha B$

follow(B) = follow(S)

→ follow(A) = { b, $ }

Date : / /20
Page No.:

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

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

→

	First	Follow
E	{(, id}	{), $}
E'	{+, ε}	= follow(E) = {$,)}
T	{(, id}	⇒ first(E') = {ε, +} {+, $,)}
T'	{*, ε}	= follow(T) = {+, $,)}
F	{(, id}	= {*, ε, +, $,)}

→

Date : / /20
Page No.:

eg-9 -

$S \rightarrow aSA \mid SB$
$A \rightarrow aH \mid \epsilon$
$B \rightarrow bcd \mid \epsilon$
$H \rightarrow h \mid \epsilon \mid b$

$S \rightarrow aSA \mid aSAS'$
$S' \rightarrow BS' \mid \epsilon$
$A \rightarrow aH \mid \epsilon$
$B \rightarrow bcd \mid \epsilon$
$H \rightarrow h \mid \epsilon \mid b$

	First	follow
S	{a}	{$, a, b}
S'	{ε, b}	{$, a, b}
A	{a, ε}	{$, a, b}
B	{b, ε}	{$, a, b}
H	{h, b, ε}	{$, a, b}

→

Date : / /20
Page No.:

class 9 resolve

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$
} left recurs^

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

	First	Follow
E	{(, id} ✓	{$,)} ✓
E'	{+, ε} ✓	{$,)} ✓
T	{(, id} ✓	{+, $,)} ✓
T'	{*, ε} ✓	{+, $,)} ✓
F	{(, id} ✓	{*, +, $,)} ✓

Q- $S \to aSA \mid SB$
$A \to aH \mid \epsilon$
$B \to bcd \mid \epsilon$
$H \to h \mid \epsilon \mid b$
} left recurs^.

⇒ $S \to aSAS'$
$S' \to BS'$
$A \to aH \mid \epsilon$
$B \to bcd \mid \epsilon$
$H \to h \mid \epsilon \mid b$

Date: / /20
Page No.:

	First	Follow
S	{a} ✓	{a, b, $} ✓
S'	{b, ε} ✓	{a, b, $} ✓
A	{a, ε} ✓	{b, a, $} ✓
B	{b, ε} ✓	{b, a, $} ✓
H	{h, ε, b} ✓	{b, a, $} ✓

class → • Now to use predictive parsing after calc. first & follow, we list down nonterm. row wise & terminals usually in order of increasing order of precidence) in cols:

→ predictive parsing table

	+	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→id	

in production

2) • If we get ε, we put it in follow of that term. → nonterm. → ε

1) • we put the non-term.'s product^n in its ~~follow~~, first

↳ eg - if we want to write for S, we check first(S) → ↳ then we write product's.

If we get ε in the nonterm.'s product, eg - E' → +TE' | ε ↳ put E' → ε in follow symbols of E'

Date: / /20
Page No.:

- Now we can parse a string using the table. (as on right side)

stack	input (end marker $)	action
$E	id+id*id$	
→ $E'T' (match now see entry of T & id)	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
now T' & + $E'T'id (pop)	id+id*id$	F→id
$E'	+id*id$	T'→ϵ
$E'T+	*id*id$	E'→+TE'
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F→id
$E'T'F*	*id$	T'→*FT'
$E'T'id	id$	F→id
$E'	$	T'→ϵ
$	$	E'→ϵ

→ parsed ↳ accepted

- If we ~~get~~ do not get a ~~prodn~~ production pair, then it is a error.
- Then error handler will recover.
  ↳ panic mode error recovery.

→ Here, by looking at stack & i/p symbol, parser can predict what could be the next production.

Date: / /20
Page No.:

Q- $S \to A$
$A \to aB \mid Ad$ } left recursion
$B \to bBC \mid f$
$C \to g$

string ⇒ a bbfgg d.

⇒ $S \to A$
$A \to aBA'$
$A' \to dA' \mid \epsilon$
$B \to bBC \mid f$
$C \to g$

	first	follow
S	{a}	{$}
A	{a}	{$}
A'	{d, ε}	{$}
B	{b, f}	{g, d, $}
C	{g}	{g, d, $}

	a	b	d	f	g	$
S	S→A					
A	A→aBA'					
A'			A'→dA'			A'→ε
B		B→bBC		B→f		
C					C→g	

Date : / /20
Page No.:

Stack	Input	action
$ S	abbfggd $	-
$ A	abbfggd $	S → A
$ A'BA	abbfggd $	A → a B A'
$ A'CBb	bbfggd $	B → bBC
$ A'CCBb	bfggd $	B → bBC
$ A'CCf	fggd $	B → f
$ A'Cg	ggd $	C → g
$ A'g	gd $	C → g
$ A'd	d $	A' → dA'
$	$	A' ∈

string accepted