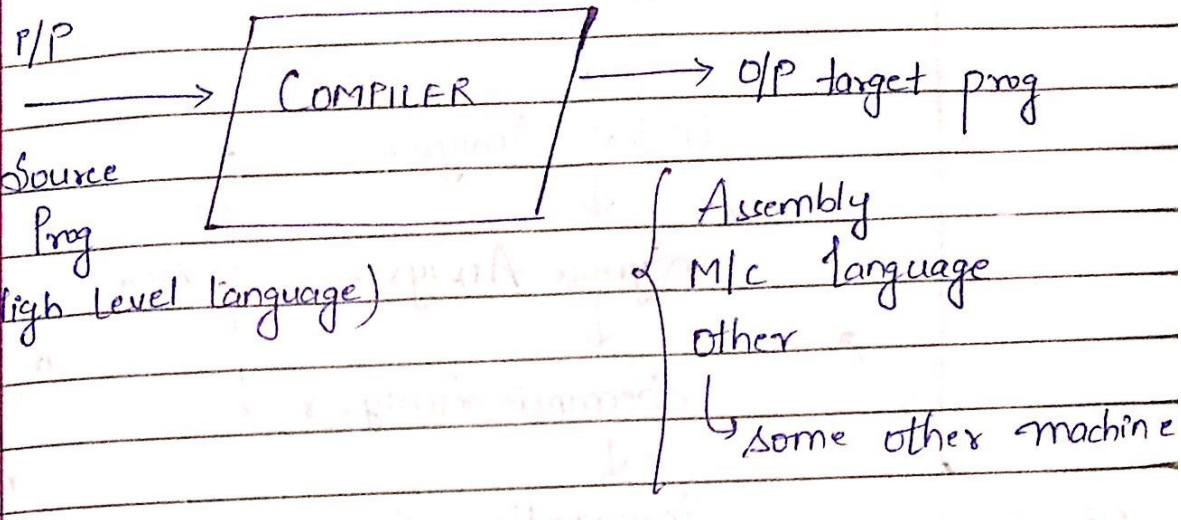
Page No.
Date : / /



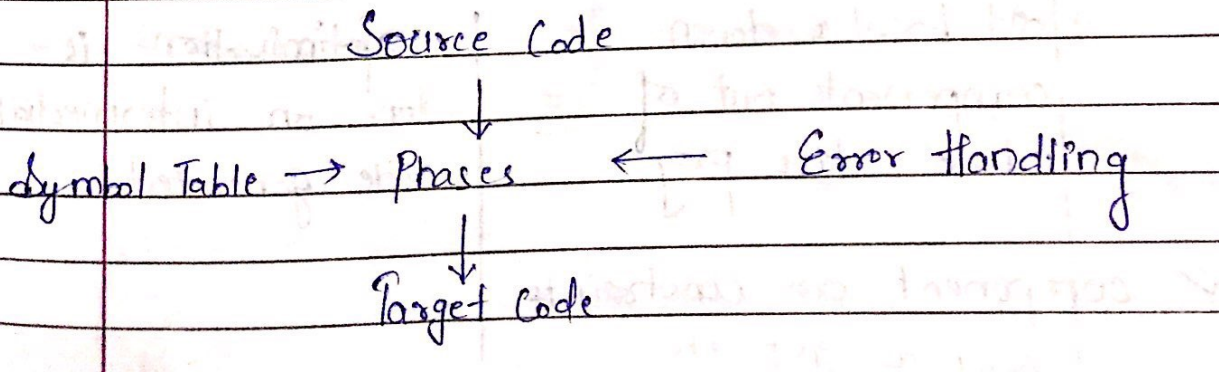
Assembly :-

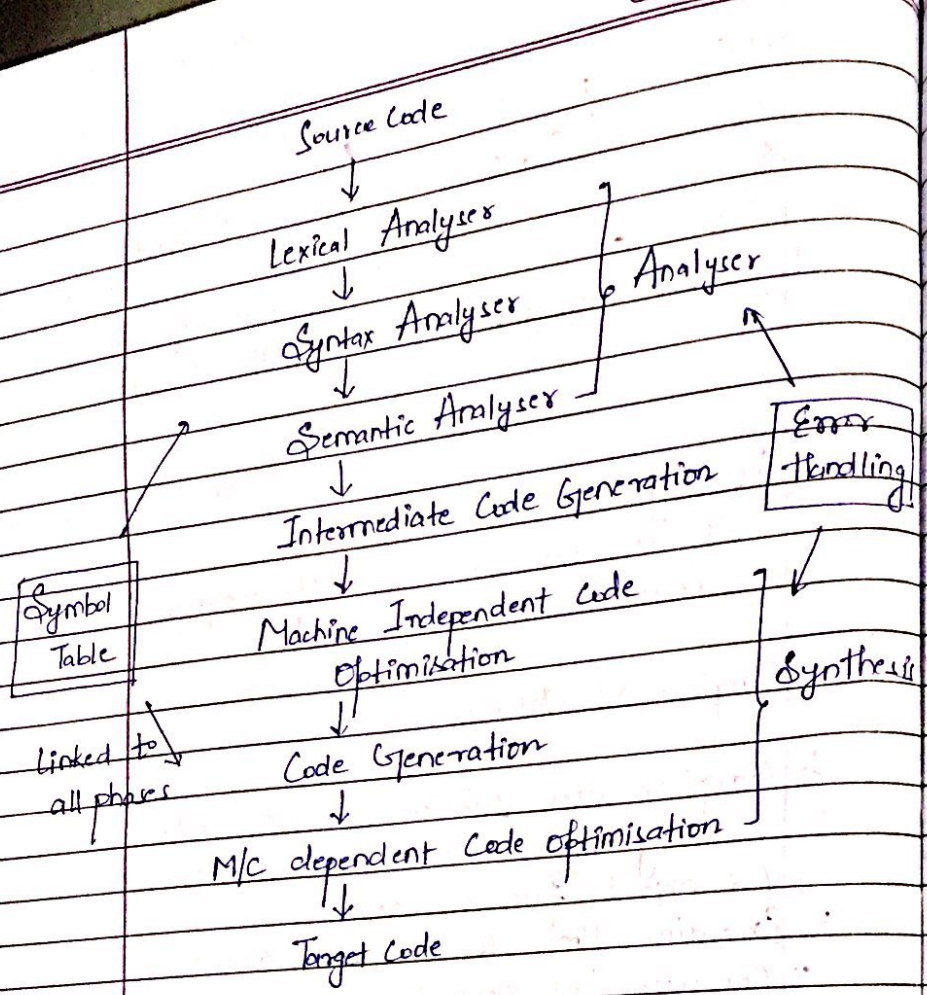
i/p → compiler → assembly language → assembler → target

Compiler v/s Interpreter

1. Compiler uses sub codes while compiling.
2. Memory req. is more than interpreter (∵ object file is created)
3. Execution time is less than interpreter
∵ Compiler has, execution as well as compilation time & interpreter has one execution time only

PHASES OF COMPILATION





Compiler consist of 2 main phases

Compiler

Analysis Phase	Synthesis Phase
→ It breaks down components out of whole HLL prog	→ optimisation is done on intermediate code generated.
w component are constraints such as int, etc. identifier, etc.	

✓ Then analysis is performed.

✓ intermediate code is generated

position := initial + rate * 60

Analysis Phase

(1). Lexical Analysis (Linear analysis / Scanning)

```
void main ()
{
    float initial, rate, position;
    position = initial + rate * 60;
}
```

• Linear analysis :- bcz it reads each char from left to right i.e. linear scan from left to right.

→ It then converts into some meaningful token. Token are acc. to diff^n languages.

→ Symbol table contains all keyword from where token is checked during token generation.

→ A meaningful set of char read by lexical analyser is a lexeme.
Then, this lexeme is matched with existing pattern in present language & then check it in symbol table to confirm if it was a keyword.

Date : / /

Eg. void matches variable name pattern but also present in Symbol Table.

∴ It is a keyword.

→ It only validates lexeme with tokens of source language.

→ lexeme is current set checked.

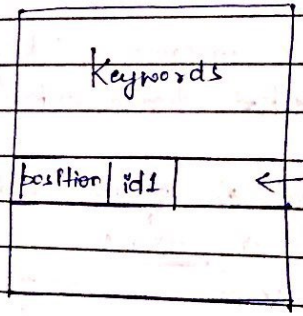
→ token is stored constraints.

Eg. void → lexeme	; → lexeme
Keyword → token	Semicolon → token

→ As soon as a new identifier is found, (→ variable name)
it is entered in symbol table.

↳ if, encountered next time, ptr is linked to that ptr stored in ST.

Symbol Table :-



Page No.
Date : / /

O/P of Lexical Analysis

<position, id1, ptr> → of ST

< t, plus sign >

↳ Errors it will not report :-

main ()

validate ← if not present then it is not a problem in this phase.

↳ It only checks stored tokens.

(2). SYNTAX ANALYZER

also, Hierarchial Analysis / Parsing → o/p tree is traversed.

Src code → LA → token → Syntax Analyser → Parse Tree (if able to make a parse tree successfully then it is a valid sentence.)

LA ↔ FA/RE

Syntax Analyser ↔ Grammar (CFG)

Grammar (CFG):
- all rules related to a particular language used.
- ✓ It helps in building meaningful sentences.

Expression

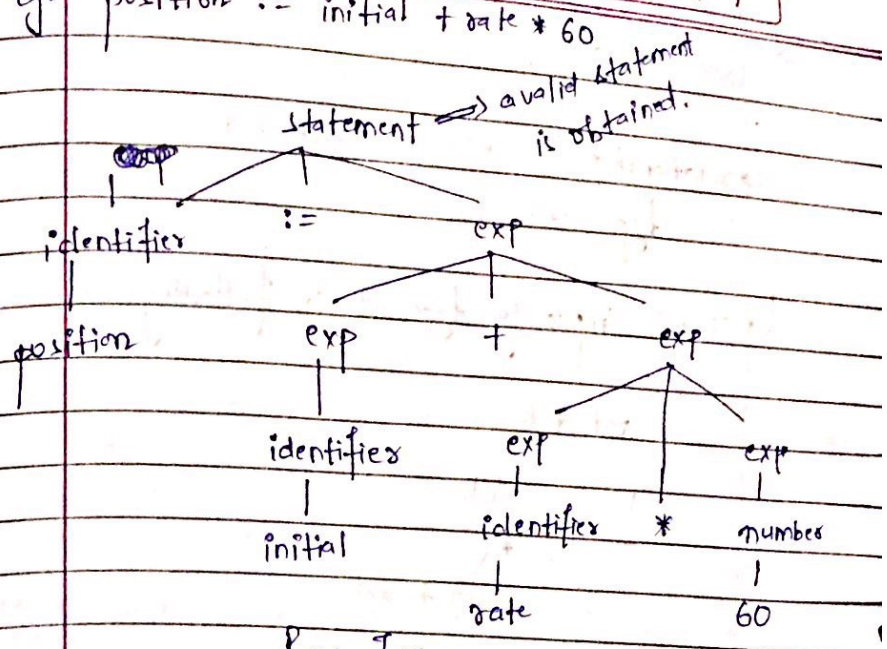
1. Any identifier is an expression
2. Any number "
3. If exp_1 & exp_2 are expressions then,

$exp_1 + exp_2$ } are exp's.
$exp_1 * exp_2$ }
(exp_1) }

Statement

1. identifier$_1$ = exp_2
2. while (exp) do stm$_2$

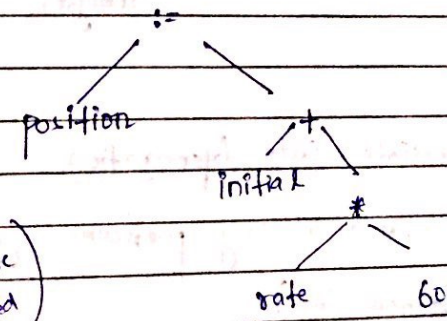
eg. position := initial + rate * 60



Parse Tree

⇒ a valid statement is obtained.

Compressed representation of Parse Tree is Syntax Tree.



Syntax Tree

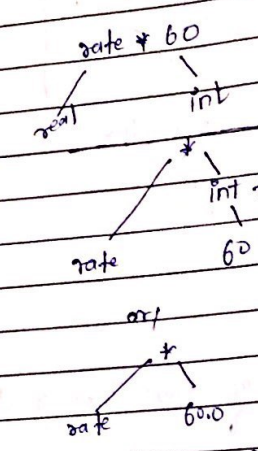
* level should be differentiated

Date : / /

③. Semantic Analysis

↳ relationship b/w variables is checked
i.e. type checking.

↳ it will implicitly typecast in following way



(annotated Parse Tree) → o/p of Semantic analyser.

④. Intermediate Code Generation

↳ a code, not in any particular language

(3-Address Code)

↓

maximum 3 addresses are allowed.

Eg. LHS := = + =

① ② ③

Page No.
Date : / /

We use temporary variables to ensure 3-address code

t1 := int to real (60)
t2 := rate * t1
t3 := initial + t2
position := t3

use tmp variables freely to make computation as easy as possible.

⑤. MACHINE INDEPENDENT CODE OPTIMISATION

↳ This optimisation works on all machines.

t1 := rate * 60.0
position := initial + t1

Date : / /

Book – Ch-3

⟹ LEXICAL ANALYZER

It also keeps new line characters which helps in returning the line no. containing error.

Eg. fi (a = 6)
- fi — identifier
- (— open bkt
- a — identifier
- = — no equal
-) — closing bkt

error handling — not matching to any pattern ⇓ * panic mode ⇓ it will throw them.

Eg. abc@@# cdf
- abc — valid, longest match
- @@# — not allowed in identifier

(abc) lexeme → identifier.

@@# → thrown away

cdf → again identifier

Recovery mode
1. " panic mode recovery
2. " deleting an extra char
3. " Inserting a missing char
4. " replacing incorrect char by a correct char
5. " transposing adjacent char

Page No. Date : / /

Eg. 123.

matching no. but not a complete no.

∴ To continue it takes inserting a missing char recovery mode.

Eg. 123,48

→ locally it will replace it with . to validate it.

(4)th recovery mode.

Eg. E 9 -4

exponent

→ locally handeled & replaced by 9 E -4

(5)th recovery mode..

Terms related to Automata.
1. Symbol → any char 0,1,a,b,# --etc.
2. (Σ) Alphabet → set of symbols
3. String → finite seq of symbol
4. Language → set of strings generated by a regular expression.
5. (abc) prefix (n+1) if n symbols including a, ε, ab, abc

Scanned with CamScanner

Date: / /

6. Suffix (n+1) c, bc, abc, ε

7. proper prefix ⇒ abc i.e. w of string not included. (Hence, w = abc)

8. proper suffix ⇒ n/g.

precedence: * , • — concatenation, + — union

ϕ → null set containing • {}

ε → empty string.

Regular definition

regular expressions are assigned a name which is used, while writing R.E. containing these exp^n.

Eg. / ⇒ same as +

letter only, digit only, with first letter (id=)

R.E. = (a|b|c|--|z|A|B|--|Z)(a|b|c|--|z|A|B|--|Z|0|1--9)*

converting to definition.

letter → A|B|C--|Z|a|b|--|z

digit → 0|1|2|--|9

Now,

Page No.
Date: / /

identifier → letter (letter | digit)*

Eg. (number)

exponent, fraction,

digit → 0|1|--|9

sign → +|-

digits → digit digit* or (digit+)

op_fraction → ε|. digits

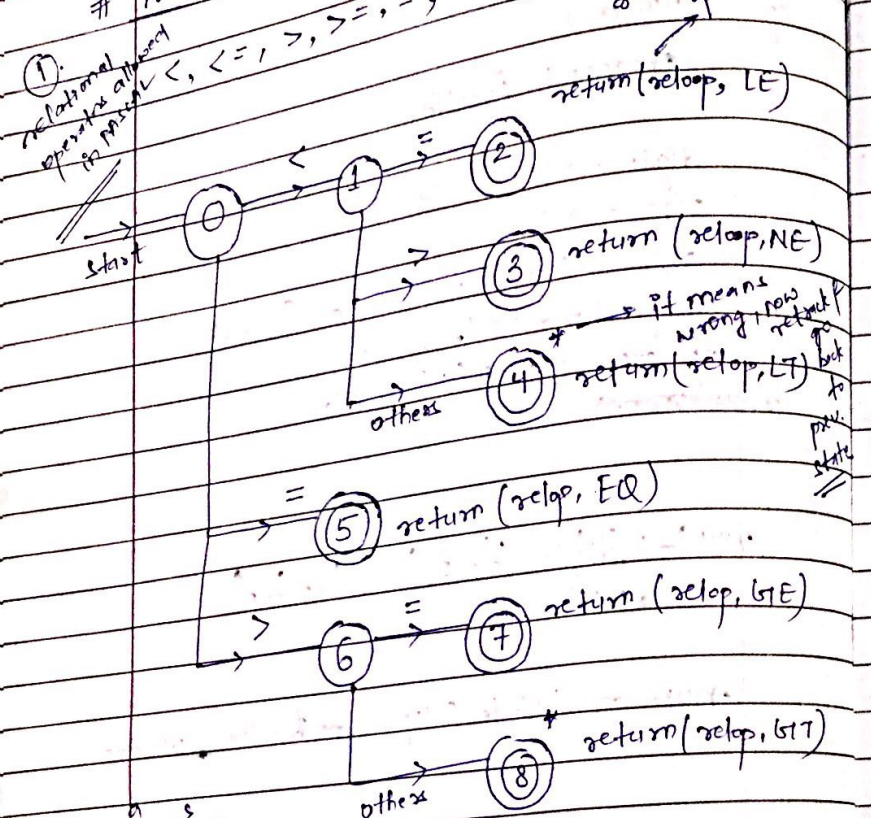
op_exp → ε|E(+|-|ε) digit

num → digits /op_fraction /op_exp
(+|-|ε)

num → (+|-|ε)(digits . op_fraction . op_exp)

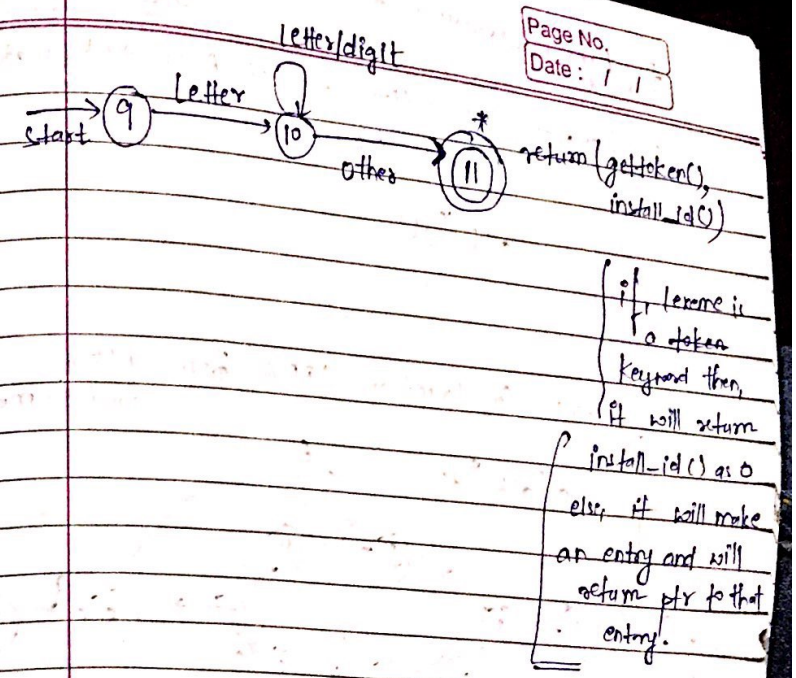
Transitional diagram

1. relational operators allowed in prog. <, <=, >, >=, =, <> (not equal to)

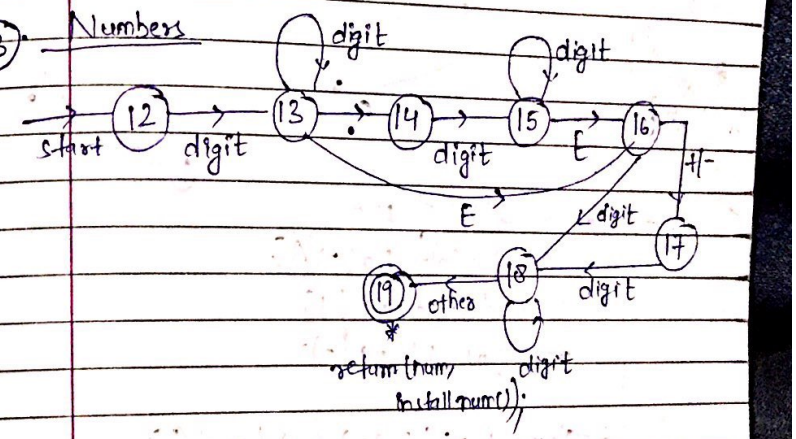


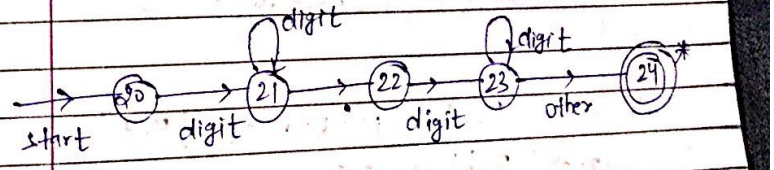
Every time a lexeme matches by starting at state 6.

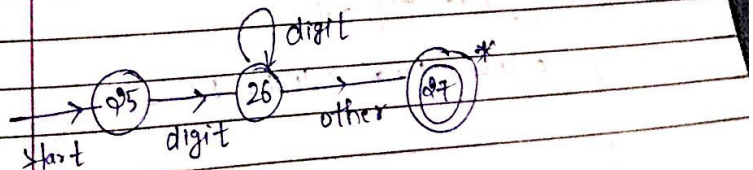
2. Identifier → starting with alphabets only, followed by a numeral or alphabets.



3. Numbers







Page No.
Date : / /

Regular Expression to NFA

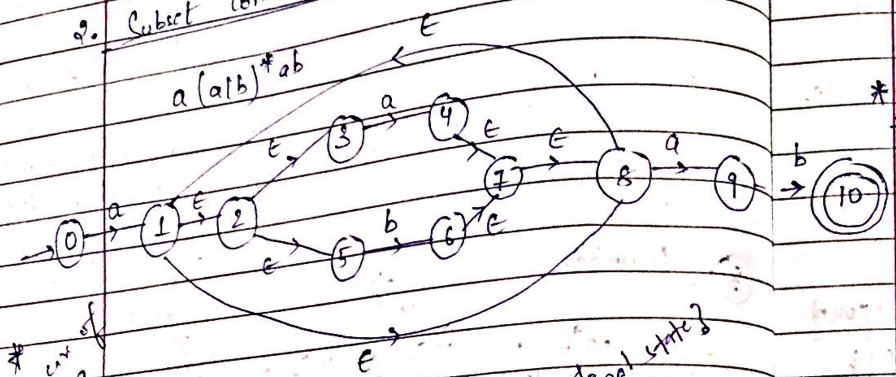
⇒ ε-NFA should be made

1. Thompson Construction
RE → NFA

$(a|b)^* abb$

2. Subset Construction (DFA from NFA)
with ε-moves

$a(a|b)^* ab$



* Take care of numbering

⑩ final state}

1. → initial state of DFA
ε- clos (0) → {0} → (A)
mov ({0}, a) → {1}
mov ({0}, b) → (-)

ε-closure (1) → {1, 2, 3, 5, 8} → (B)

2. mov {B, a} → {{1,2,3,5,8}, a} → {4, 9}
mov {B, b} → {6,}

ε-closure {4,9} → {4, 7, 8, 1, 2, 3}

* if directly mentioned then draw NFA w/o ε.

Page No.
Date : / /

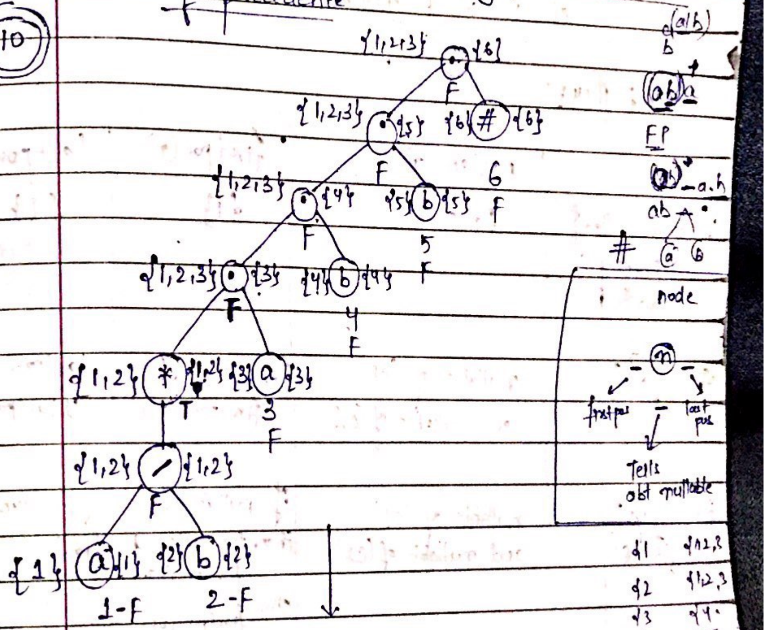
3. Regular Expression to DFA

$(a|b)^* abb\#$

In NFA :-
⇒ States with no ε-moves are called Imp. States.
⇒ States with ε-moves are called non-Imp states.

Ⓢ follow this dir^n.
if parenthesis comes, make it separate

* Step I : Make Tree by keeping in mind order of precedence



* Step II : Assign unique pst. to leaf nodes.

Every node (interior or exterior) is a expression.

Step 3: Calculate following fn for the tree

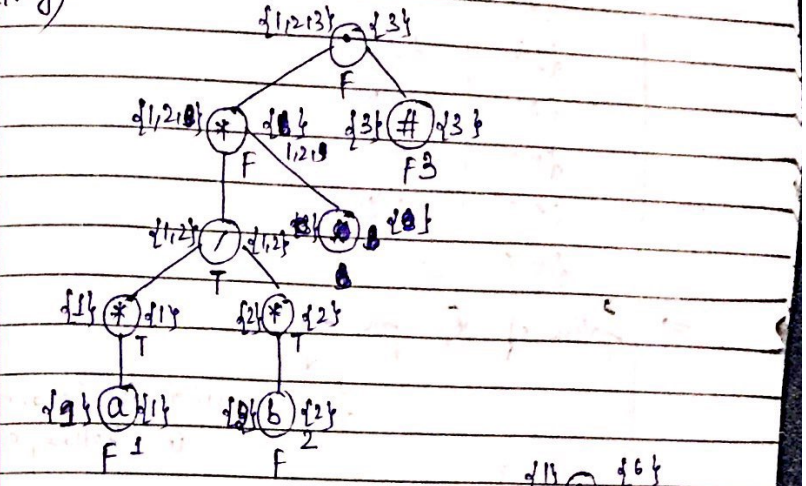
1. Nullable → calculated for each and every node
2. firstpos → calculated for each node (first symbol in generated string)
3. lastpos → " (last symbol in a generated string)
4. follow → for some nodes only. it belongs to ⊙ concatenation node?
   (*) → every node

→ if a language generates 'ε' along with any o/p it is called nullable.

RULES

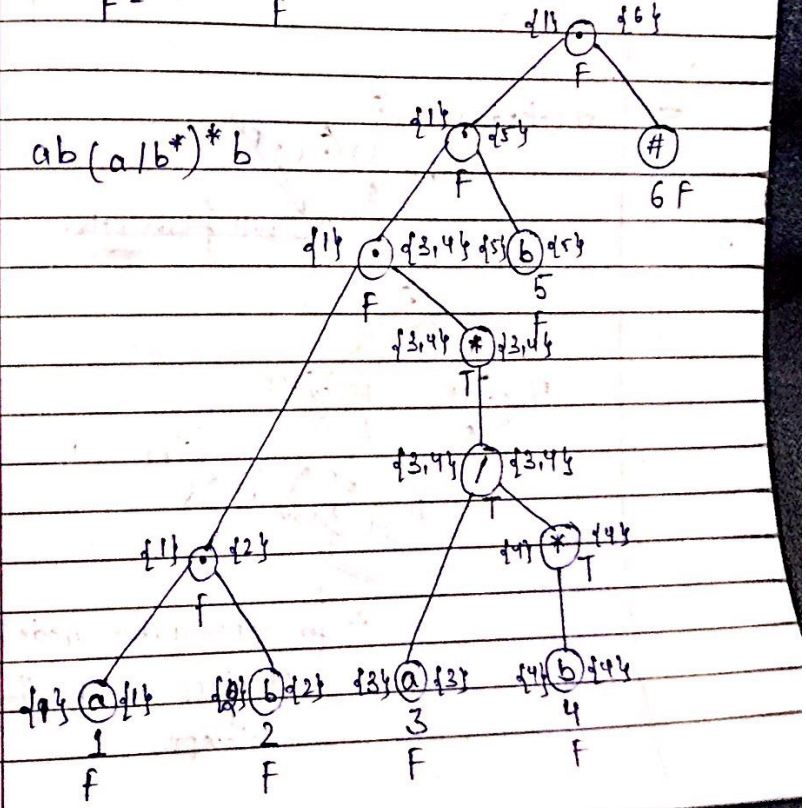
	Nullable(n)	firstpos(n)	lastpos(n)
1. n is a leaf (labeled with i)	false	{i}	{i}
2. n: / with children c1, c2	nullable of (c1) or, nullable of (c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
3. n: ⊙ with children c1, c2	nullable of (c1) and nullable of (c2)	if, nullable(c1) then, firstpos(c1) ∪ firstpos(c2); else, firstpos(c1)	if, nullable(c2) then lastpos(c1) ∪ lastpos(c2); else lastpos(c2)
4. n: * with child c	True	firstpos(c)	lastpos(c)

Q. Eg. $(a^*/b^*)^*.\#$



H.W.

Q. $ab(a/b^*)^*b$



Date : / /

→ follow is calculated for all nodes but they are only calculated for ⊙ and (*) nodes.

find out positions

• (a+b)*abb#

pos	follow
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4}
{4}	{5}
{5}	{6}.
{6}	-

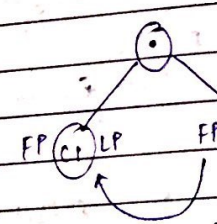
⇒ follow of (*) pos.

first position is followed up in follow pos.

⇒ concatenation, ⊙

LHS ⊙ RHS

This will follow LHS.



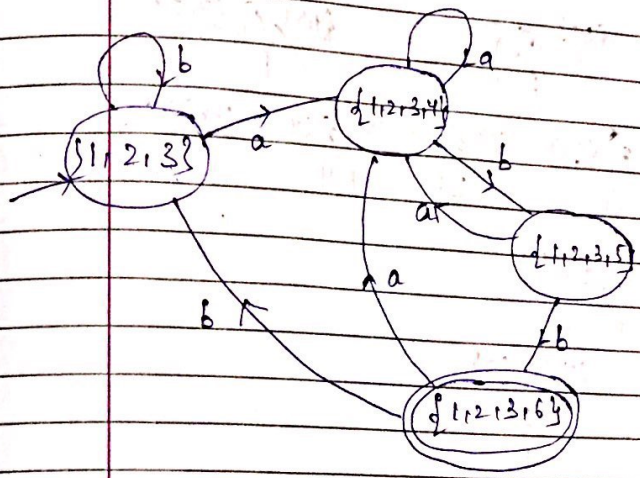
Here, FP of c2 will follow LP of c1

∴ for concatenation node, LP of c2 will be the follow pos.

Rules :-

Page No.
Date : / /

1. → Start state of DFA, consist of all (FP) of root node.
2. To find out transition, we find that transition variable on pos included in current node or state. Then, union of all the follow pos is taken.



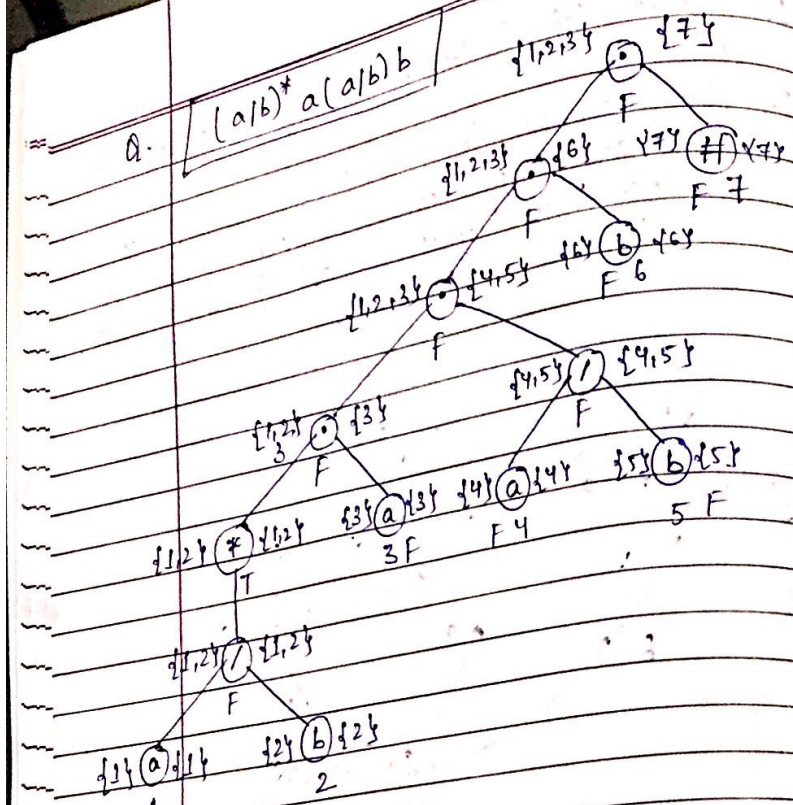
3. Accepting state ⇒ # pos.

Here, # pos = (6).

Now, all states including # pos will have final state.

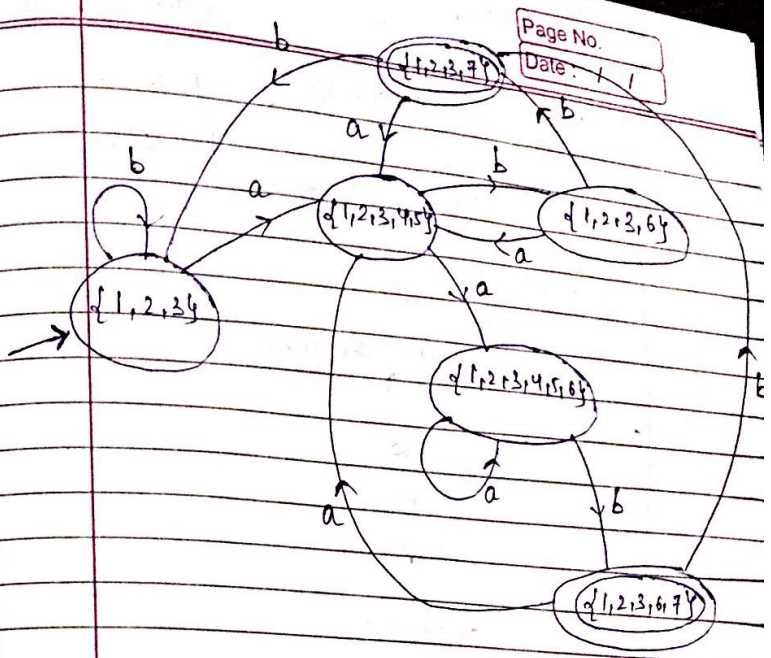
Page No.
Date : / /

Q. $(a|b)^* a(a|b)b$



follow up

pos	follow
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4,5}
{4}	{6}
{5}	{6}
{6}	{7}
{7}	-



Date : / /

SYNTAX ANALYSIS

CFG, $G = \{ V, T, S, P \}$ — P: production, S: start symbol, V: non-terminals, T: terminals

$L(G) = \{ \quad \}$ — set of terminals

G w

$S \xrightarrow{*} w$

derivation

derivation

Leftmost Rightmost

✓ Graphically derivation is Parse tree.

✓ Ambiguity ⇒ Leftmost or Rightmost 2 or more parse tree for same string.

$E \rightarrow E*E / E+E / (E) / -E / id$

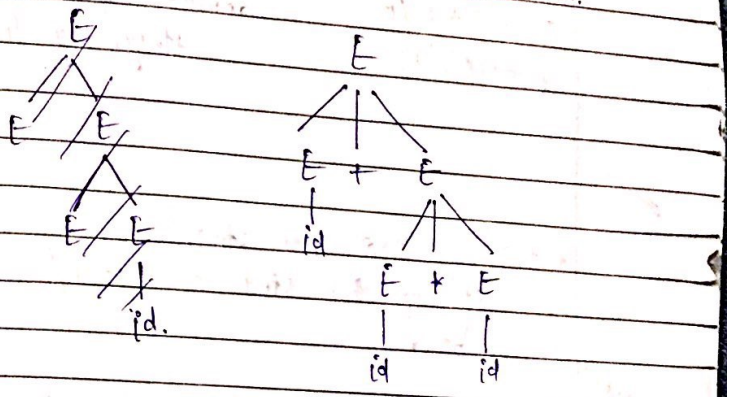
string → (id + id * id)

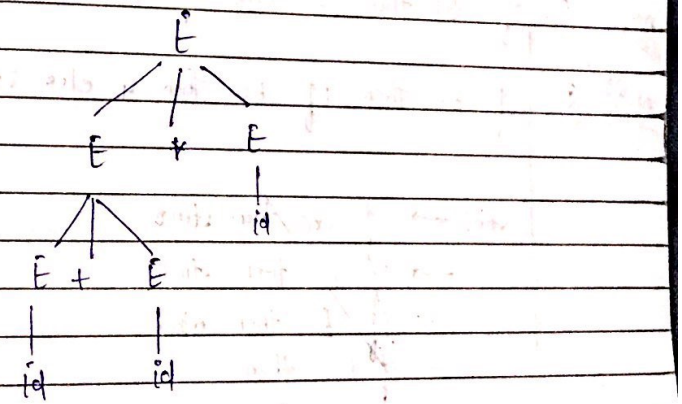
two right most

Page No.
Date : / /

$E \rightarrow E+E \rightarrow E+E*E \rightarrow E+E*id \rightarrow E+id*id$
$\rightarrow id + id*id.$

$E \rightarrow E*E \rightarrow E*id \rightarrow E+E*id \rightarrow E+id*id$
$\rightarrow id + id*id.$





⇒ Some parsers can take ambiguous trees but they need to have some disambiguous rules to handle it. → Hectic task.

Date : / /

↓

To handle this, we try to convert ambiguous grammar into unambiguous grammar.

Unambiguous grammar for last eg:-

$E \rightarrow E+E/T$
$T \rightarrow T*F/F$
$F \rightarrow (E)/id$

→ only one left most derivation for a particular string. e.g. Right most

stmt → if exp then stmt /
if exp then stmt else stmt /
other.

non-ambiguous

ambiguous

① if E_1 then (S_1 else if E_2 then S_2 else S_3.

② if E_1 then if E_2 then S_1 else S_2.

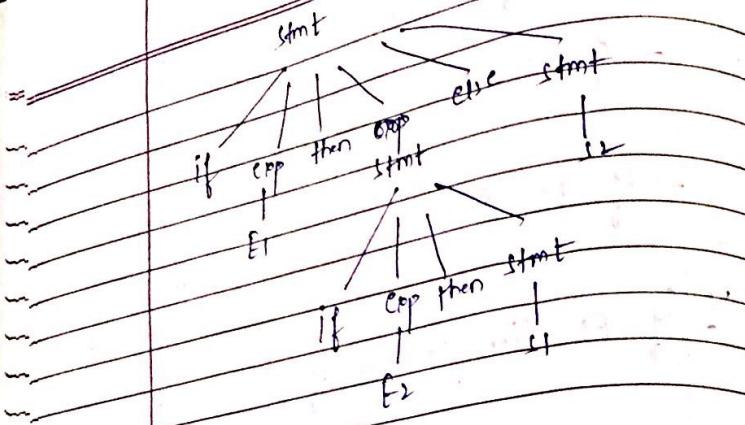
stmt → if exp then stmt
→ if E_1 then stmt
→ if E_1 then other
→ if E_1 then

stmt → if exp then stmt else stmt
→ if E_1 then stmt else (stmt)
→ if E_1 then other else stmt
→ if E_1 then S_1 else stmt
→ if E_1 then S_1 else if exp then stmt else stmt

Page No.
Date : / /

→ if E_1 then S_1 else if E_2 then stmt else stmt
→ if E_1 then S_1 else if E_2 then other else stmt
→ if E_1 then S_1 else if E_2 then S_2 else other
→ if E_1 then S_1 else if E_2 then S_2 else S_3

②



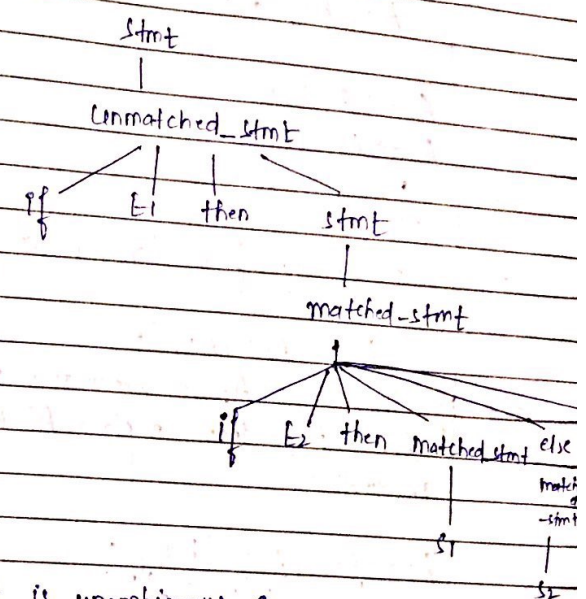
* acc. to PASCAL, ~~last~~ every else matches to its near most if, which is coming incorrect in ~~last~~ and derivation.

stmt → un_matched_stmt / matched_stmt

matched_stmt → if expr then matched_stmt else matched_stmt / other

unmatched_stmt → if expr then stmt / if expr then matched_stmt else unmatched_stmt

Eg. if E1 then if E2 then S1 else S2.



✓ Now, it is unambiguous grammar.

LEFT RECURSION (A → Aα/β)

* Top-Down Parser do not accept left Recursion.
* Bottom-up Parser can accept Left Recursion.

A → βA'
A' → αA'/ε

Left-Recursion

Immediate (Clearly visible)

Non-Immediate (after substitution, it becomes visible.

Page No.
Date : / /

Eg. $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

$A_1 \rightarrow A_2 a \mid b$
$A_2 \rightarrow A_2 c \mid A_1 d \mid \epsilon$

↓

→ Start removing from lower most production.

$A_2 \rightarrow A_2 c \mid A_1 d \mid \epsilon$

** If, there is A_j in production of A_i such that $i < j$ then replace A_i with its productions.

$A_2 \rightarrow \underbrace{A_2 c}_{A\alpha_1} \mid \underbrace{A_2 ad}_{A\alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{\epsilon}_{\beta_2}$

$A_2 \rightarrow bd A_2' \mid A_2'$

$A_2' \rightarrow c A_2' \mid ad A_2' \mid \epsilon$

Now,

$A_1 \rightarrow A_2 a \mid b$
$A_1 \rightarrow bd A_2' a \mid A_2' a \mid b$

✓

$A_1 \rightarrow bd A_2' a \mid A_2' a \mid b$
$A_2 \rightarrow bd A_2' \mid A_2'$
$A_2' \rightarrow c A_2' \mid ad A_2' \mid \epsilon$