Left - Most Derivation

Derivation is a sequence of production rules. It is used to get the i/p string through these production rules.

We have to decide
- → which Non-Terminal to replace
- → production rule by which the non-terminal will be replaced

Left Most Derivation:- I/p scanned & replaced with the production rule from left → right.

eg $S \rightarrow S+S \mid S-S \mid a \mid b \mid c$

input a - b + c

LMD
$S \rightarrow S+S$
$S \rightarrow S-S+S$
$S \rightarrow a-S+S$
$S \rightarrow a-b+S$
$S \rightarrow a-b+c$

Right-Most Derivation:- I/p scanned & replaced with the production rule from Right → left

$S \rightarrow S+S \mid S-S \mid a \mid b \mid c$ i/p a - b + c

RMD
$S \rightarrow S-S$
$S \rightarrow S-S+S$
$S \rightarrow S-S+c$
$S \rightarrow S-b+c$
$S \rightarrow a-b+c$

Parse Tree

It is a typical depiction of how the start symbol of a grammar derives a string in the language.

OR

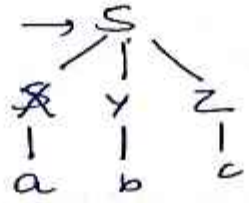
It is a graphical representation of symbol that can be terminals or non-terminals.

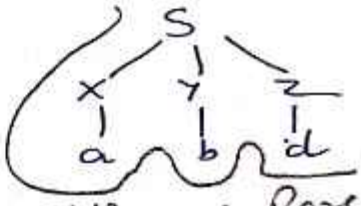
Properties

1) Root is always the start symbol.
2) All leaf nodes are terminals
3) All interior nodes → non-terminals

Eg

$S \rightarrow XYZ$
$X \rightarrow a$
$Y \rightarrow b$
$Z \rightarrow c/d$





Yield of Parse tree
a b d

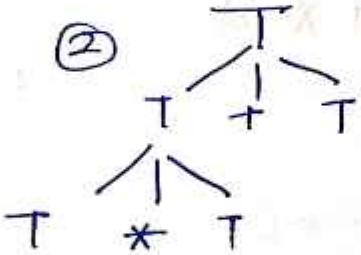
$L = \{abc, abd\}$

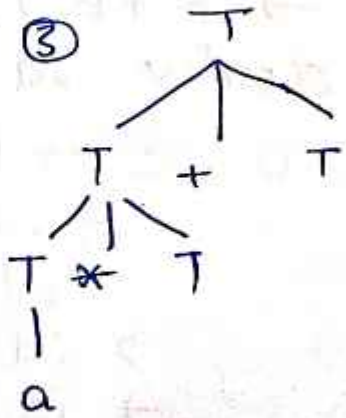
Eg

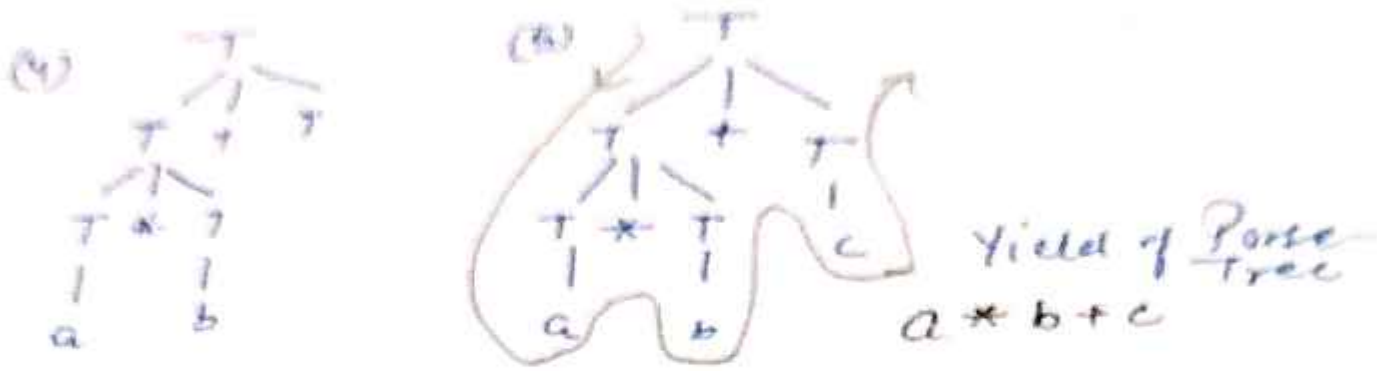
$T \rightarrow T+T \mid T*T$
$T \rightarrow a/b/c$

I/p a * b + c

① 

② 

③ 



★ Ambiguous Grammar

A context free grammar is said to be ambiguous if there exists more than one derivation trees for the given i/p string

or

more than one Left Most & Right Most derivation.

→ There is no standard method to check ambiguity, we have to do it by pratice / hit & trial method.

problem with ambiguity → precedance of operators is violated / not respected.

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

derive id + id * id

L.M.D. $E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

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

RMD $E \to E+E$

$\to E+E*E$

$\to E+E*id$

$\to E+id*id$

$E \to id+id*id$

$E \to E*E$

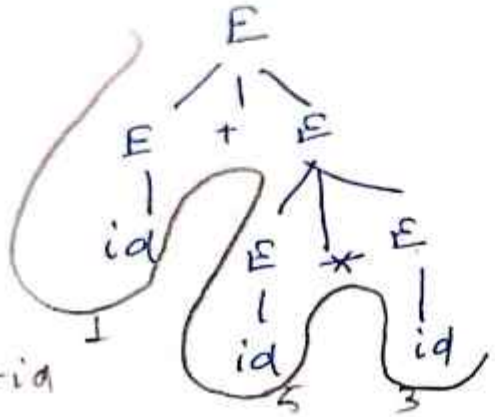
$E \to E*id$

$\to E+E*id$

$\to E+id*id$

$E \to id+id*id$

Parse Tree-1 (valid)

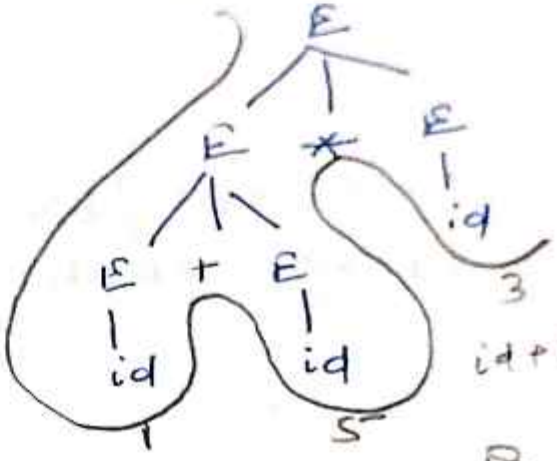


id+id*id

valid parse tree-1 bcz * precedence is more than other operator

= 16

Parse Tree-2



id+id*id

= B

∴ Since grammar is ambigious. Parser will get confused which one to generate

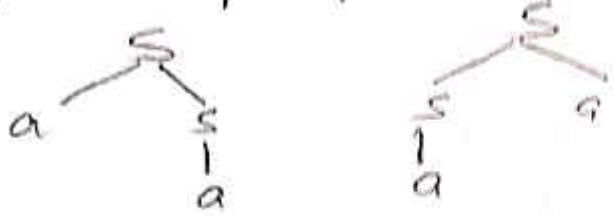
Note :- So, parsers donot allow ambiguous gra except of one.

↓

operator precedence operator

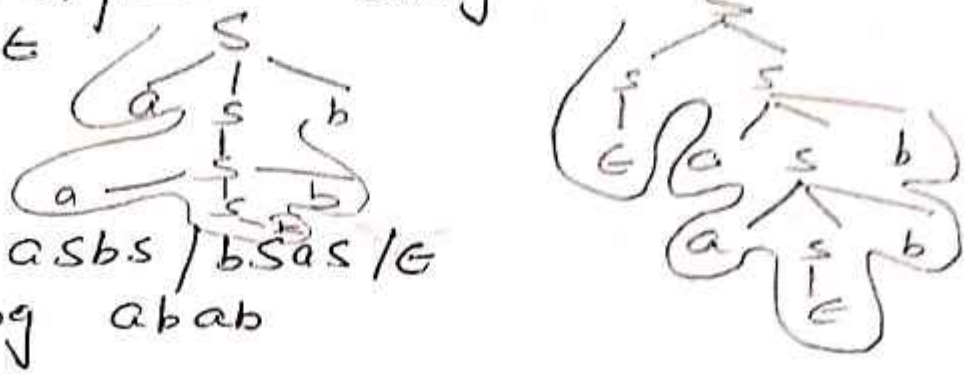
Q Check grammar is ambiguous or not?

i) $S \rightarrow aS \mid Sa \mid a$ String aa



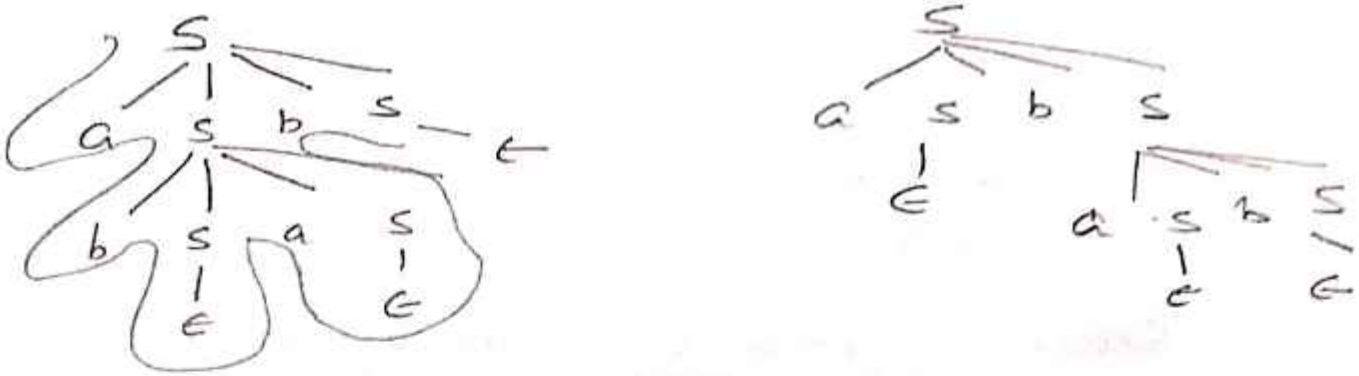
ii) $S \rightarrow aSb \mid SS$ String aabb

$S \rightarrow \epsilon$



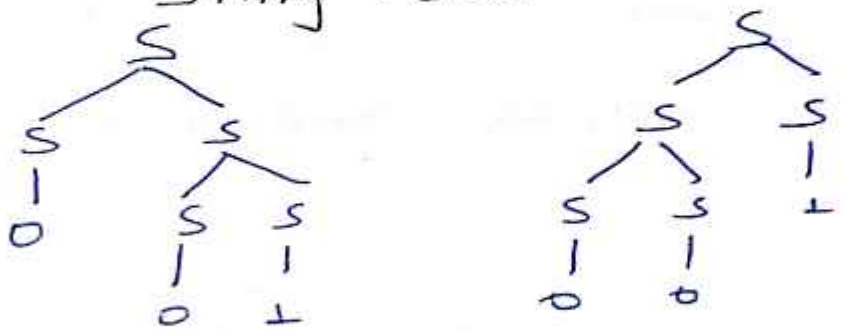
iii) $S \rightarrow aSbS \mid bSaS \mid \epsilon$

String abab



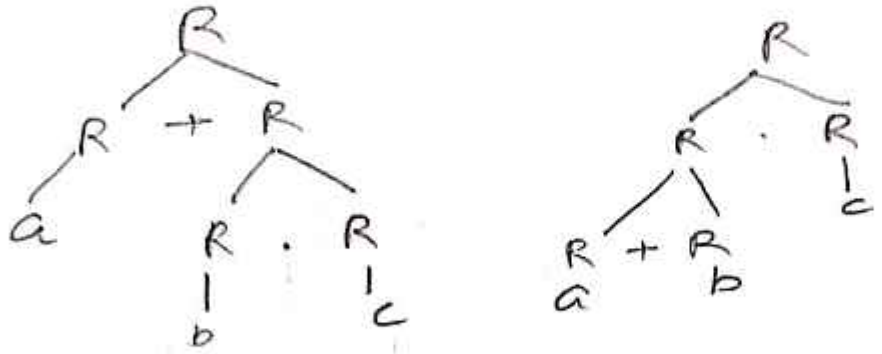
iv) $S \rightarrow SS \mid 0 \mid 1$

String 001



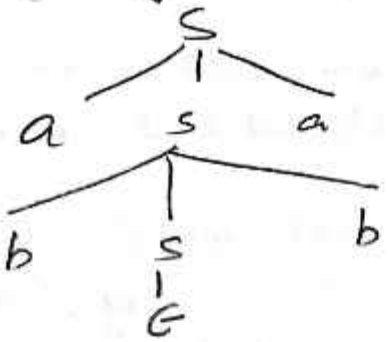
v) $R \longrightarrow R+R \mid R-R$ String a + bc

$R \rightarrow a \mid b \mid c$



(VI) $S \longrightarrow a\,Sa \mid b\,S\,b \mid \in$

String - a b b a



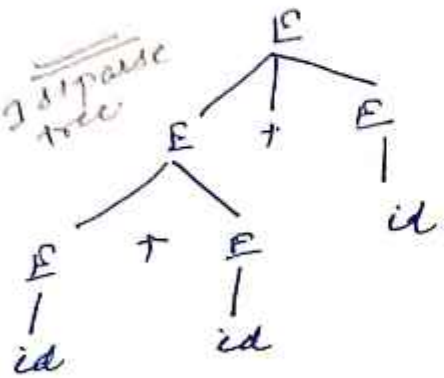
∴ This grammar unambiguous

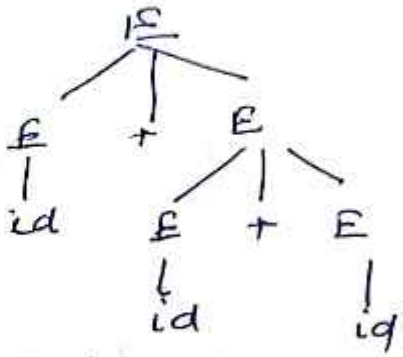
Converting Ambiguous to Unambiguous

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

$E \to id$

1st parse tree





id + id + id (the grammar failed b/c rules of associativity failed)

↓

It has 2 operators on either side of it. Which operator should I associate this with?

If we associate with left operator → left associativity

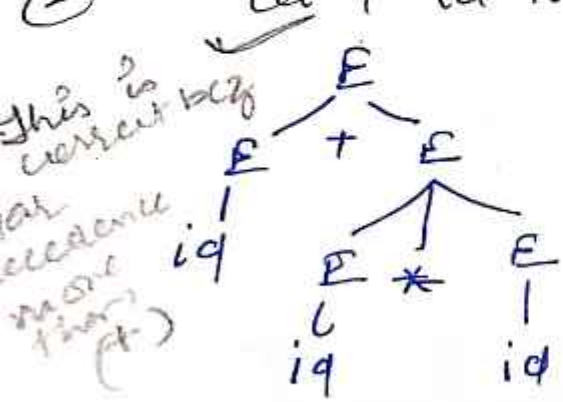
(id + id) + id

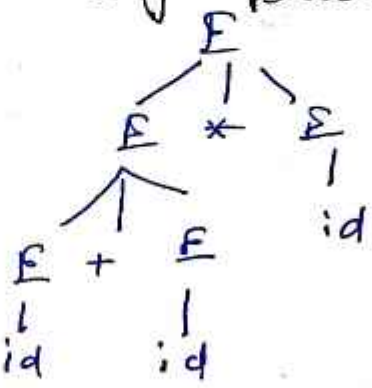
for Right associativity id + (id + id)

+ → Left associative

So 1st parse tree correct

(2) id + id * id (grammar failed? b/c precedence is not taken care) (This is wrong)

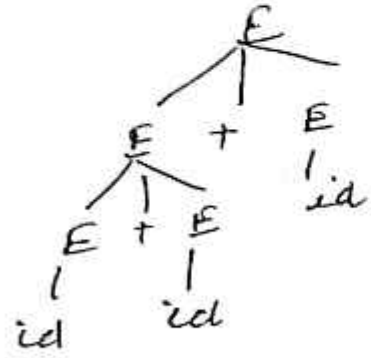




So, if we take care of (i) associativity & (ii) precedence then grammar can be unambiguous.

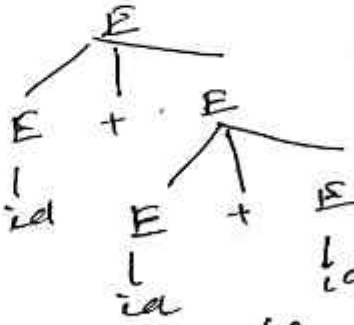
Ex (i) $E \rightarrow E+E / E*E / id$

id + id + id



(id + id) + id

left associative

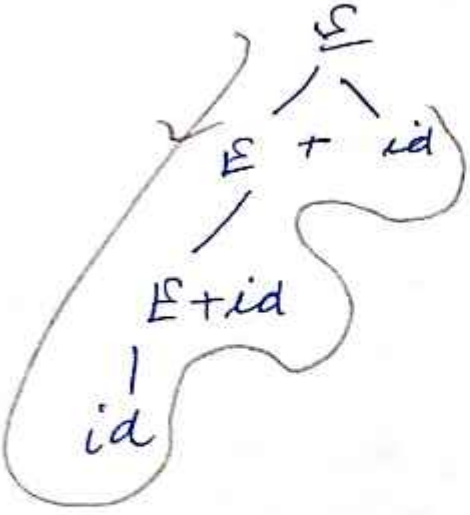


id + (id + id)

Right associative

To achieve left associativity, we have to grow in left direction only.

$E \rightarrow E+id / id$



We are growing only in left direction Because grammar is defined as Left Recursive.

So, there is no possibility of getting different parse tree.

For precedence problem

We should take care that highest precedence operator should be at the least level.

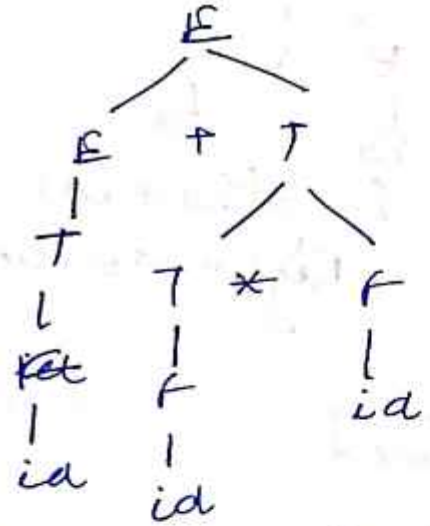
$E \rightarrow E + E \mid E * E \mid id$

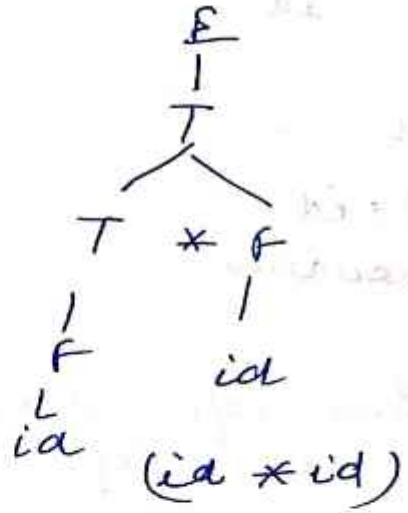
generate
id + id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$





(id + (id * id))

(id * id)

Right associative

Like we have to group in rightwards

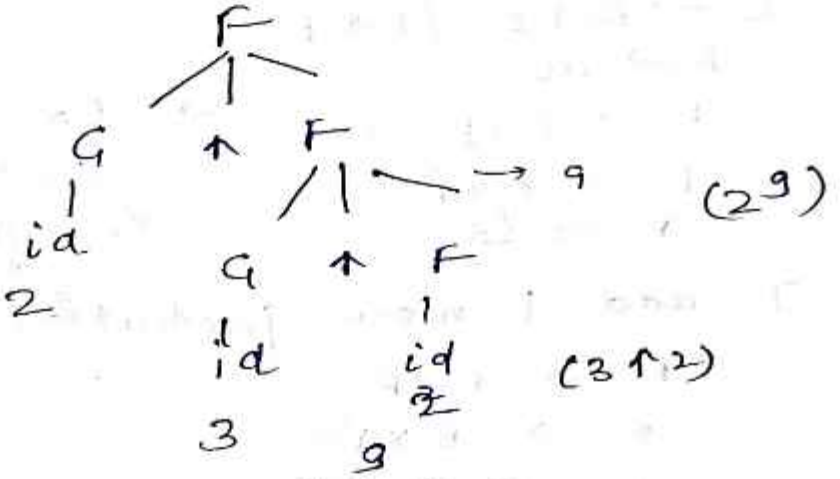
Eg $(2 \uparrow (3 \uparrow 2))$ means $2^{(3^2)} \rightarrow 2^9$

(first solved this, after second solved)

using Right recursion

$F \rightarrow G \uparrow F \;/\; G$ (Right associativity)

$G \rightarrow id$



Q

$bExp \rightarrow bExp$ OR $bExp$
/ $bExp$ AND $bExp$
/ NOT $bExp$
/ True
/ false

True & false are terminal

First we have to convert in unambiguous grammar

OR < AND < NOT → precedence of NOT is higher than AND, OR

$E \rightarrow E$ OR F / F
$F \rightarrow F$ AND G / G
$G \rightarrow$ NOT G
$G \rightarrow$ / True / false

OR } left
AND } associative
NOT } any
it is not associative

Q. $A \to A \$ B / B$
$B \to B \# C / C$
$C \to C @ D / D$
$D \to d$

"Tell associativity & precedence."

Associativity we find through recursion i.e. left Associative

Starting symbol se jitna jyada operator aayega uska utna precedence jayada hoga.

$\$ < \# < @$

Q. $E \to E + E / E * F$
$F \to id$
$E \to E + f$
$E \to E * f$
$F \to id$

+ & * → same level
∴ Same precedence & left associative

If I add 1 more production

$E \to E + f$
$E \to E * f$
$F \to f - id$

Hence (−) is having more precedence than + & *.

Transition Diagrams

Transition diagrams are usually used for analyzing a particular pattern comparing it & generating the tokens in the lexical analyzer.

Recognition of Tokens

The regular definitions for token are as follows:

Keywords ← if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
identifier ← id → letter (letter | digit)*
num → digit+ (.digit+)? (E(+|-)? digit+)?
delim → blank | tab | newline
ws → delim.
(whitespace)

→ Lexical analysis uses transition diagram to keep track of information about characters that are seen as the forward pointer scans the i/p. (what is read)

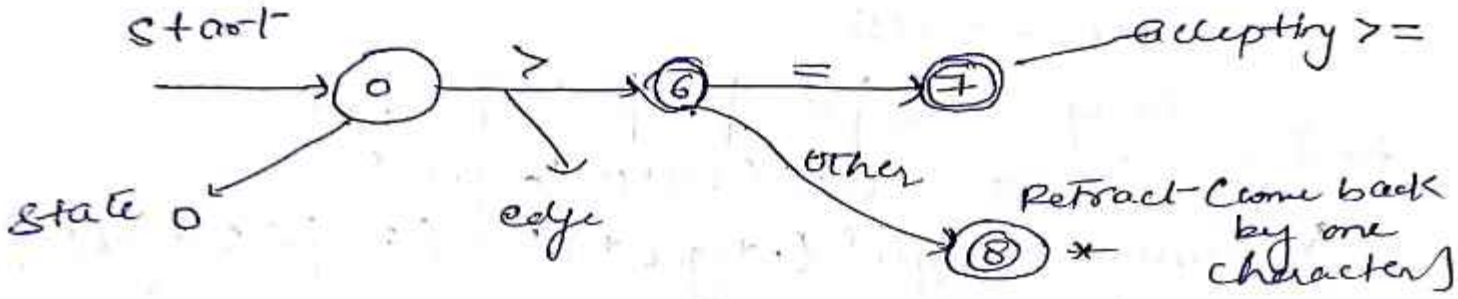
Transition diagrams are also called finite automata

→ Positions in a transition diagram are drawn as circles & are called states.

→ The states are connected by arrows, called edges.

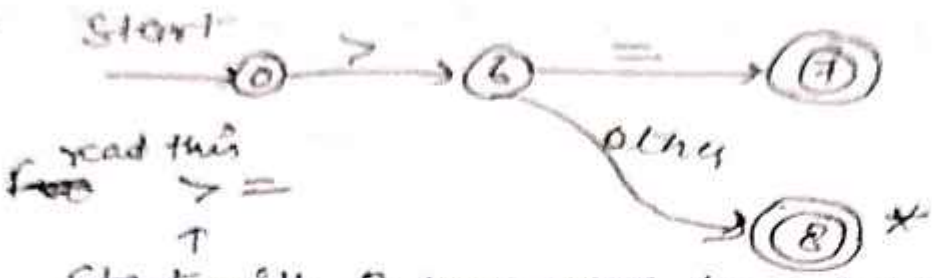
→ A double circle indicated an accepting state, a state in which a token is found.

→ * indicates that i/p retraction must take place.



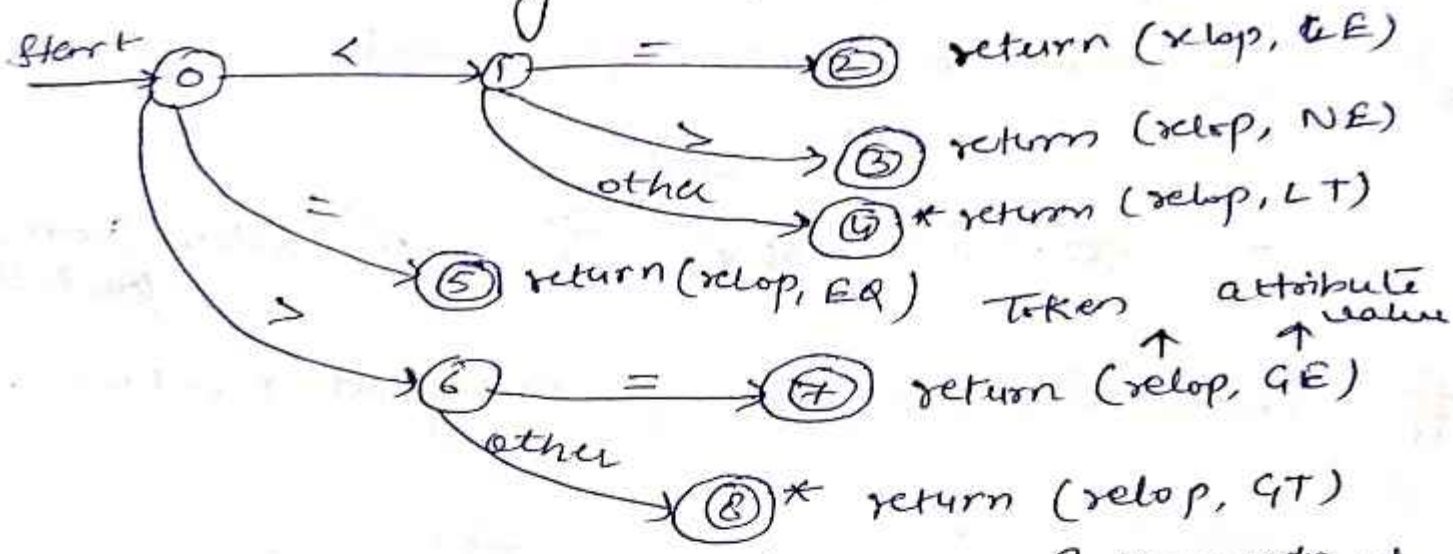
Attribute Values

Regular Expression	Token	Attribute Value
ws	—	—
if	if	—
then	then	—
else	else	—
id	id	ptr to table entry (Symbol table)
num	num	"
<	relop	LT
<=	"	LE
=	"	EQ
<>	"	NE
>	"	GT
>=	"	GE

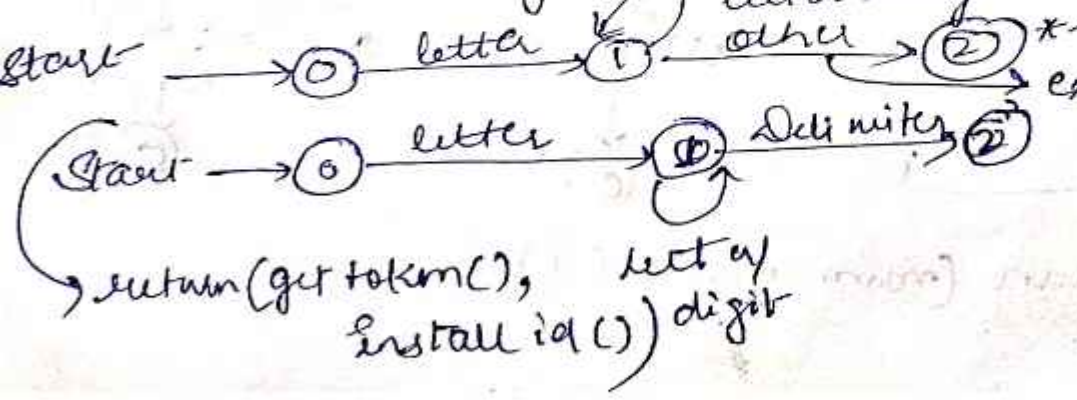


Start with 0 now move to 6 by greater than sign after it coming to state 6 now we are reading next character that is = to 2 then it to coming to next step(7) & it is accepting

Transition Diagram for Relational operators



Transition diagrams for identifiers & Keywords (combination of letter & digit)



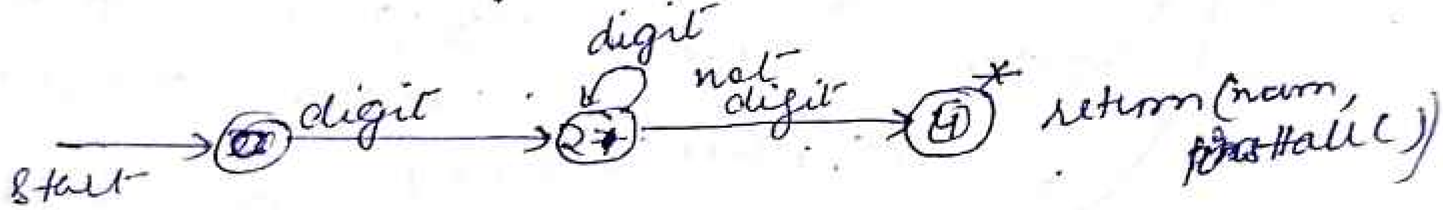
gettoken() :- It searches the identifiers in the symbol table & if it finds them references are passed as attributes with token id.

Lexical analyser: identifiers in the symbol table if it is found then find the address / attribute value then vo value to parser token ke sath pass karega.

install id() :- Otherwise install id() is invoked which first enters the identifier's name, the symbol table & then passes it reference. (pass address)

Keywords (if, then, else)

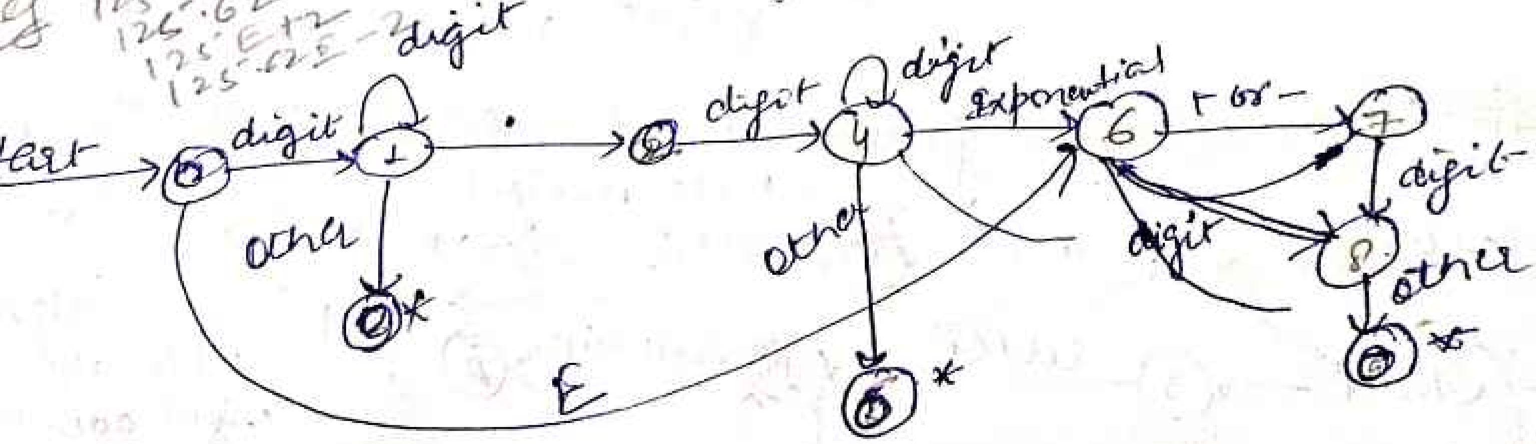
Transition diagram for ~~number~~ constants



Transition diagram for unsigned numbers

digit (0 - 9)

eg 123
125.62
125E+2
125.62E-2



return (num, install())

★ Transition diagram for whitespace

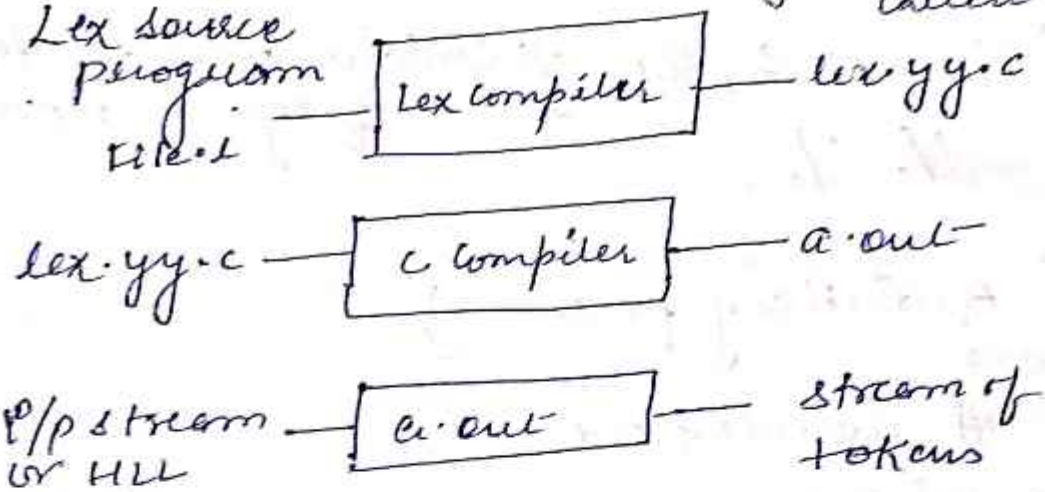
(Blank + tab + newline)

delim

Lex tool in compiler design

Lex is a tool/computer program which generates lexical analyzer. It is used with YACC parser generator.

Lex Compiler — It will transform the I/P patterns into a transition diagram & generates code in a file called lex.yy.c



1) Source code is in lex language with filename .l extension.
It is given to lex compiler, which is lex tool, & produces lex.yy.c - a C program as O/P

(2) C compiler runs this c code i.e. lex.yy.c program & produces as o/p [a.out] i.e. lexical analyzer

(3) [a.out] transforms as i/p stream to stream of tokens.

LEX file format

A lex program is separated into 3 sections by %% delimeters.

{ declarations } // declaration of variables, including files, libraries.
%% %%

{ Transition rules } // Contains rules in form of regular expression
%% %%

{ Auxilliary functions }

eg
```
%{ #include<stdio.h>
   int c=0;
%} ---> (declaration)
%%
pattern {action}  (Rules)
%%
main() -> function
{ }
```

Rules

abc - match abc

[a-z] - match small a to z

[a-z]* - match all the strings in small letters with null or more than 1 character

[a-z]+ - 1 or more character

[A-Za-z]+ - match all the character 1 atleast 1 or more character

^a - means a should come at start

a$ - means a at end

[0-9]+ - 1 or more digits

yywrap() - called by lex tool when i/p is exhausted return 1 if i/p is finished else 0.

yylex() - reads the i/p stream & generate tokens a/c to the regular expressions.

yytext - pointer to the i/p string.

```
%{ #include <stdio.h>
   declaration
%}
%%                                  → action (will be in C lang syntax)
"hi"  {printf ("By");}              → Rules.
except hi x  {printf ("worry");}
any other word
%%
main()
{printf("enter i/p");
 yylex();   // Tokens i/p & generate the patterns in
            tokens d/c to pattern rules in section
}
int yywrap()   end of i/p.
 {return 1;}
```