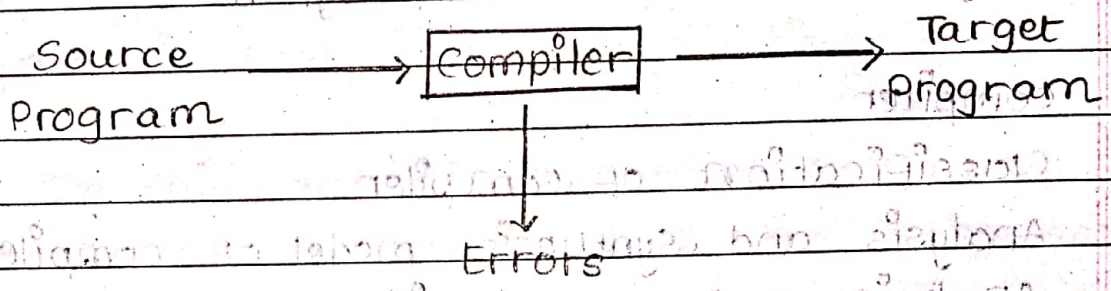
Date 1.
Page

Compiler Design

It is a program that reacts a program written in one language [source language] and translate it into an equivalent target program.



→ Classification of Compiler

1. Single Pass
2. Multi Pass
3. Load and go
4. Debugging
5. Optimized

→ Debugging is like interpreter.

→ Analysis / Synthesis model of Communication

Analysis
↓
Synthesis

→ Analysis Tools.

1. Structure Editors
2. Pritty Printers
3. Static checkers
4. Interpreters

→ It is actually command based.

→ To discuss potential bug

→ Following are additional analysis tools
ex. Text formattor silicon compiler
query interpreter
→ Output is in circuit form.
→ Output is in circuit diagram.

→ Analysis of the source program.

compiler
Classification of compiler
Analysis and synthesis model of compiler
Analysis tools of compiler

1. Linear Analysis / Lexical Analysis / Scanning

The stream of character making up the source program is to read from left to right and grouped into tokens there are sequences of character having a collective meaning.

- Token.
  Operators
  Keywords
  Datatype
  Signs

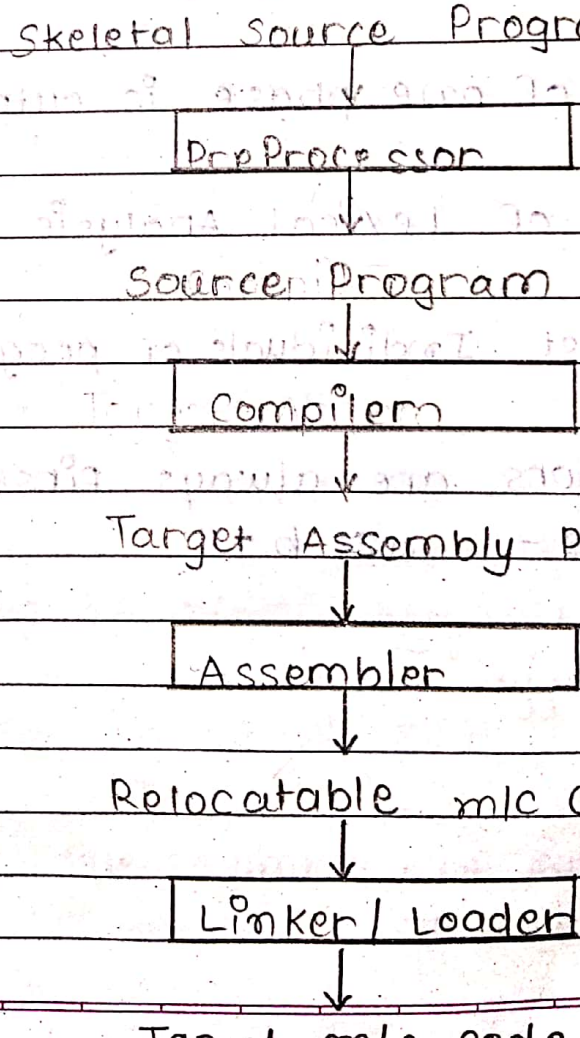
2. Hierarchical Analysis / syntax Analysis / Parsing.

All inner nodes are always operators and leaves are number or identifier.
In this phase of analysis characters or tokens are grouped hierarchically into nested collections with collective meaning.

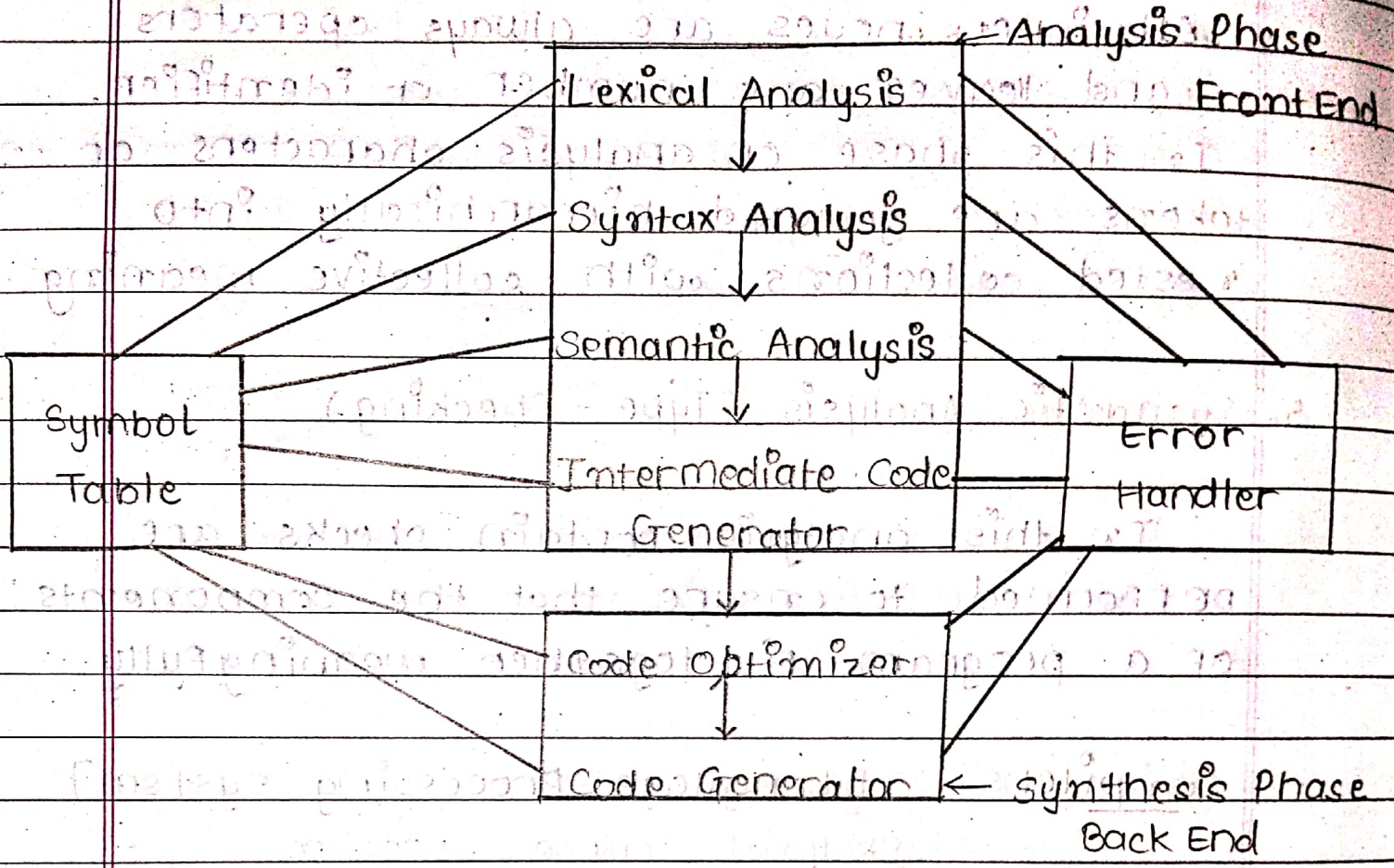
3. Symantic Analysis (Type - checking)

In this analysis certain checks are performed to ensure that the components of a program fit together meaningfully.

→ Compilers. [Language Processing System]



→ Phases of compiler



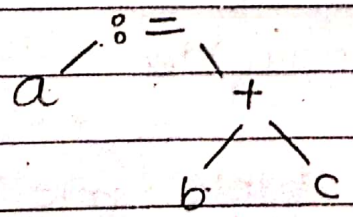
→ Input of one phase is output of other phase.

→ Output of Lexical Analysis is sequence of Tokens.

→ smallest Individuals of program are called Tokens.

→ operators are always first

Ex :- a := b + c



sum := a + b * c

t1 := b * c
t2 := a + t1
sum := t2

→ Types of Error.

1. Lexical Error :- For unknown symbol / sequence.
2. Syntax Error :- checking of syntax
3. Semantic Error :- Focus on type checking. [datatype]
4. Logical Errors :-

→ Cousins of Compiler.

1. Preprocessors
   → Macro processing
   → File inclusion
   → Rational Preprocessor
   → Language Extension
2. Assemblers
3. Two-Pass Assembly
4. Loaders and Linkers (Link-Editors).

→ The grouping of phase

1. Front-End and Back-End
2. Passes
3. Reducing the number of passes.

→ Compiler construction Tools.

Following are systems for compiler writing process:

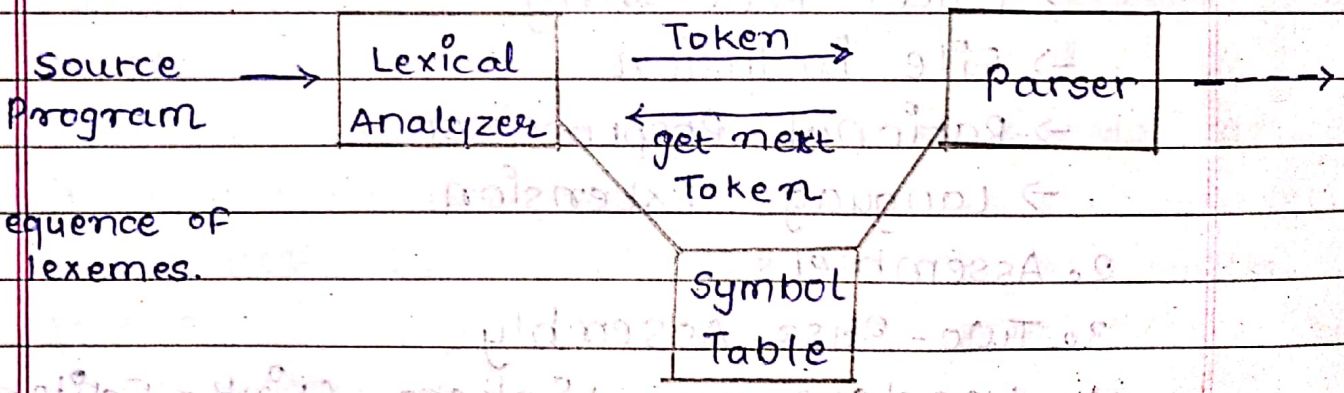
1. compiler - compilers
2. compiler Generators
3. Translator writing systems

Following are compiler construction tools:

1. Parser Generators
2. Scanner Generators
3. Syntax directed translation engines
4. Automatic code Generators
5. Data Flow Engines.

Unit-2

→ The Role of the Lexical Analyser.



→ In compiler white space includes tab, newline character, blank.

Comparison of lexemes → Lexical Analyzer

Lexical Analyzer divided into two phases:

1. Scanning - Simple task
2. Lexical Analysis - Complex Operation.

It is the first phase of compiler.
It's main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

Depending upon receiving a get next token command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Additional task is stripping out from the source program, comments and white spaces like blank, newline and tab characters.

→ Issues in lexical analysis

1. Simpler Design
2. Compiler Efficiency
3. Compiler Portability.

→ Define Following Terms :-

1. Lexeme :- It is a sequence of characters in the source program that is matched by the pattern for a token.

2. Pattern :- Set of strings is described by a rule called a pattern associated with the token.

Date
Page 8

3. Token :- (Smallest individual units of program (they are terminal symbols produced by lexical analyzer)

Token	Lexeme	Pattern
const	const	const
if	if	if
relational operator	<, <=, >, >=, =, <>	< or <= or > or >= or = or <>
num	70	sequence of digit
id	amount	letter followed by letter(s) or digit(s).

→ Attributes for Token.

When more than one pattern ~~machines~~ matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.

For example, the pattern num matches both the strings 0 and 1, but it is essential for the code generator to know what strings was actually matched.

-> Lexical Errors

Few errors not possible to detect during lexical analysis.

For eg. By mistake "if" it is entered "fi" instead "if" keyword then a lexical analyzer cannot feel whether "fi" is a misplacing of the keyword "fi".

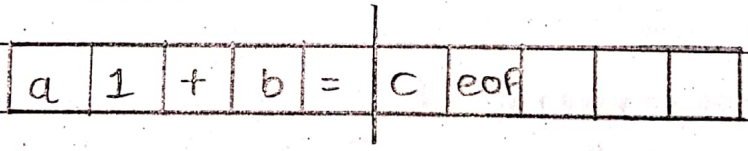
There is simplest recovery strategy called panic mode recovery.

Following are possible error recovery actions.

1. deleting an extra character.
2. Inserting a missing character.
3. Replacing an incorrect character by a correct character.
4. Transposing two adjacent characters.

-> Input Buffering

1. Buffer Pairs.




```
if forward of end of 1st half then begin
                reload second half;
                forward := forward + 1
end
else if forward at end of second half then begin
                reload first half;
                move forward to beginning of 1st half
```

```
end
else forward := forward + 1;
```

- Code to Advance forward pointer

This process is for lexical analysis and memory management.

2. Sentinels :-

a	=	b	+	eof	c	eof			eof

```
forward := forward + 1;
IF forward = eof then begin
    IF forward at end of first half then begin
        reload second half;
        forward := forward + 1;
    end
    else IF forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
    else
        terminate lexical analysis
end
```

- Lookahead code with sentinels

* Regular Expression

They are important notation for specifying patterns

Each pattern matches a set of strings, so regular expressions will serves as names for sets of strings.

* Strings and languages

The term alphabet or character class denotes any finite set of symbols.

A string over some ~~of~~ alphabet is a finite sequence of symbols, drawn from that alphabet.

In language theory the terms sentence and word are often used as synonymous from the term string.

↳ Length of string :-

eg. s = 'abc';

s = 'a';

|s|

→ empty string := string with length 0.

ϵ →

→ The term language denotes any set of strings over some fixed alphabet.

→ abstract languages can be presented using {ϕ, empty set / ϵ} (no string)

→ ex :- if x and y, are strings then concetenation of x and y return as xy or x.y.

→ $x = 'abc'$ $xy = 'abcxyz'$
$y = 'xyz'$

→ $x = 'abc'$ $xy = 'abc\epsilon'$
$y = \epsilon$
$\epsilon s = s\epsilon = s$

→ Consider eg for concetanation as a product, define $s^0 \rightarrow$ length.

$s^0 \rightarrow \epsilon$
$x^0 \rightarrow \epsilon$
and for $i > 0$ define s^i to be $s^{i-1} s$.

→ Definitions for different parts of string:
1. prefix of s:
2. suffix of s:
3. substring of s:
4. proper prefix, suffix or substring of s
eg: $s \neq x$ x = prefix, suffix, substring.

5. subsequence of s:-
eg :- $s = 'abcd'$ bd

6. operations on language :-
3 operation :- (1) Union
(2) Concetenation
(3) Closure

Union → or, Concat → and,

Closure → occurance of particular symbol.

Consider following eg. Let L be the set having symbol

$\{A, B, \ldots z, a, b, \ldots, z\}$

$D = \{a_1, \ldots, a\}$

→ New language can be created from L and D by applying the operators.

1. $L \cup D =$
2. $L \cdot D =$
3. $L^4 = LLLL \rightarrow (L^+)$
4. $L^* = L^0 \rightarrow \in$, $L^1 \rightarrow L$, $L^2 \rightarrow LL$

   $* \rightarrow 0$ or more times occurances.
5. $D^+ \rightarrow + \rightarrow 1$ or more times occurances

D^1 D^2

D DD

(0 to a) (00 to 99)

→ R.E → $L(L \cup D)^*$

↳ Definitions of operations on language :-

	operation	Definition
1.	L union M ($L \cup m$)	$L \cup M = \{s / s$ is in L or s is in $m\}$

2. Concatenation of L and m — $Lm = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } m\}$

3. Kleene closure of L written as L^* — $L^* = \sum_{i=0}^{\infty} L^i$

L^* denotes 0 or more concatenation of L

4. Positive closure of L written as L^+ — $L^+ = \sum_{i=1}^{\infty} L^i$

→ L^+ denotes 1 or more concatenation of L.

L> Regular expressions :-

They are kinds of notations which defines different rules.

eg. letter (letter | digit)*

letter ↑ Component 1 ; (letter | digit)* ↑ Component 2

→ There are rules that define the regular expression over alphabet set associated with each rule is a specification of the language denoted by the regular expression being defined.

(1) R.E → ∈ , ∈(input) , outcome {E}
(input string) (Token)

(2) If R.E. is a,
lexeme a
token a
outcome {a}

(3) Suppose r and s, are regular expression denoting languages, regular sets.

L(r) and L(s) then

3.1 → (r)|(s) → L(r) ∪ L(s)
3.2 → (r).(s) → L(r) . L(s)
3.3 → (r)* → (L(r))*
3.4 → (r) → L(r)

→ A language denoted by regular expression is said to be a regular set. (MCQ)

↳ Priority of operations:-

eg. $\Sigma = \{a, b\}$

	R.E.	Sets.
1.	a/b (1)	{a, b}
2.	(a/b)(a/b) (2.)	{{a,a}, {a,b}, {b,a}, {b,b}}
3.	a*	{∈, a, aa, aaa, ..., ∞}
4.	(a/b)*	{∈, a, b, aa, ab, bb, ba, ...∞}

5. a/a^*b — {a, b, ab, aab, ...}

↳ Algebric properties of regular expressions :-

Axiom (Formation logic)	Description
(1) $r/s = s/r$	/ is cumulative
(2) $r/(s/t) = (r/s)/t$	/ is associative
(3) $(rs)t = r(st)$	concatenation is associative
(4) $r(s/t) = rs/rt$ $(s/t)r = sr/tr$	concatenation distributes over (or) operator.
(5) $\epsilon r = r$ $r\epsilon = r$	ϵ is identity element for concatenation
(6) $r^* = (r/\epsilon)^*$	relation between * & ϵ
(7) $r^{**} = r^*$	* is idempotent.

→ Regular Definition

For notational convinience we may wish to give names to regular expressions and to define regular expression using some names, if Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form.

$d_1 \rightarrow r_1$ d → definition

$d_2 \rightarrow r_2$ r → regular expressions

:

$d_n \rightarrow r_n$

eg letter → A|B|C|...|z| a|b|c|...|z|

digit → 0|1|...|9|

id → letter (letter|digit)*

eg. Regular Definitions for unsigned number

digit → 0|1|2|...|9

digits → digit digit*

opt-frac → .digits|∈

opt-Expo → (E(+|-|∈) digits|∈

⇒ num → digits opt-frac opt-Expo

⇒ Notational Shorthands.

1. One or more instance (+)

Regular Expression (r)

$r^* = (r^* | \in)$ $r^+ = r r^*$

2. Zero or one instance (?)
(∈ (+|-) ? digits)?

3. character classes
Regular Expression [a]
[abc]

eg. l(l|d)*
[A-Z a-z] [A-Z a-z 0-9]*

→ Language Denoted by Regular Expression are called Regular sets.

→ Recognition of Tokens:-

Consider the following grammar format:-

stmt → if expr then stmt
| → if expr then stmt else stmt
expr → term relop term
| → term
(relop ↑ relational operator.)
term → id
| → num

[CFG] → Context Free Grammer
⇒ stmt, expr → Non Terminal
⇒ if, then → Terminals / Direct
⇒ stmt → if expr then stmt ⇒ Production
Label in L.H.S. called start symbol.
Only Non-Terminals can act as label or start symbol.
→ Name given to regular expression called regular definition.
→ * → for closure function
indicates 0 or more occurances
? → 1 or more occurances.
→ We cannot have Regular Definition for Non-Terminals.
→ Regular expression Patterns for tokens.

Regular Expression	Related Tokens	Attribute Values
whitespace	-	-
if	if	-
then	then	-
else	else	-
id	id	ptr to table entry
num	num	ptr to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

☆ Transition Diagram

- It can be defined as stylish flowchart signs:-

(1) ○ → it is to represent state

(2) ◎ → Final state (Accepting state)

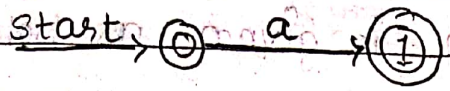
(3) ⟶ ⟵ ↑ ↓ → Directed Edges containing some label about workflow.

(4) * → represent initial state.

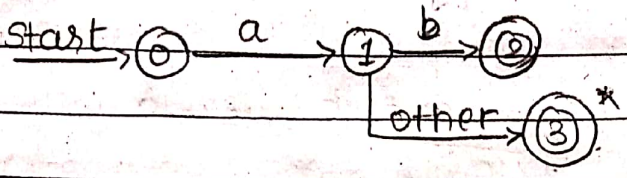
(5) other → unknown signs.

eg. Regular expressions are converted into diagrams.

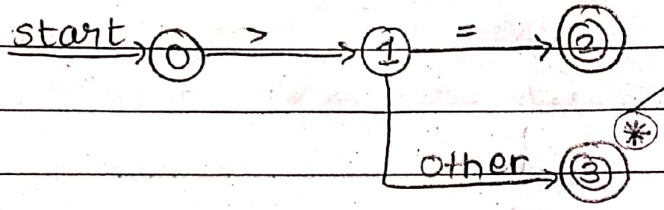
Regular Expressions — a label.

1) 

(2) R.E. abla

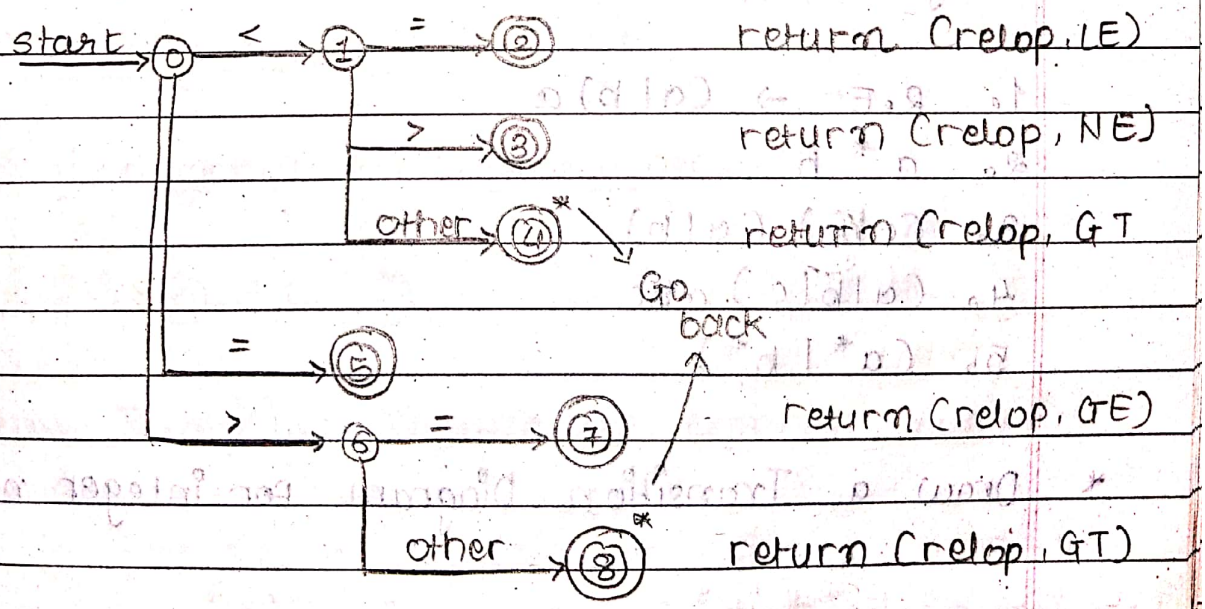


→ Transition Diagram for relop (>=).



→ Transition Diagram for relational operators.

<, <=, >, >=, =, <>



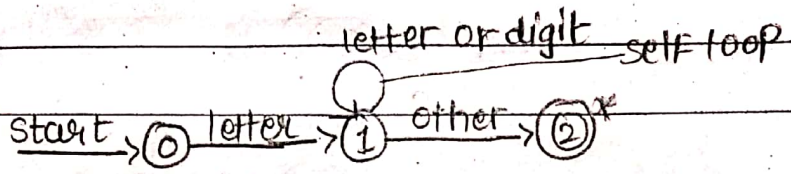
Date
Page 22

* Draw a Transition Diagram for valid identifier.

Regular Expression is given as below:-

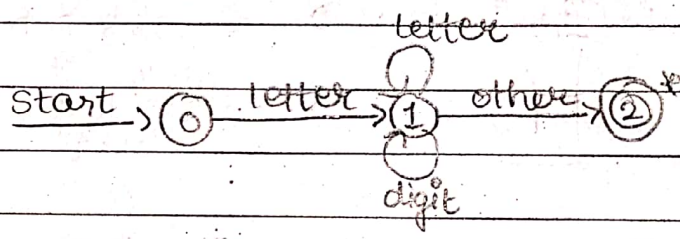
Letter (Letter | Digit)*

(1)



OR

(2)

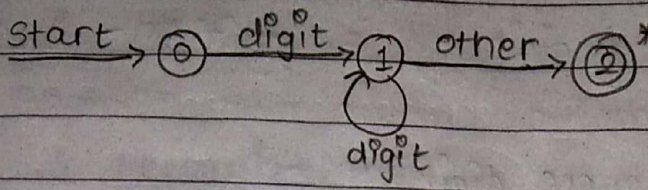


* Draw a Transition Diagrams for following regular expressions:-

1. R.E → (a | b) a
2. a* b
3. (a|b) (a|b)
4. (a|b|c) abc
5. (a* | b*)

* Draw a Transition Diagram for integer number.

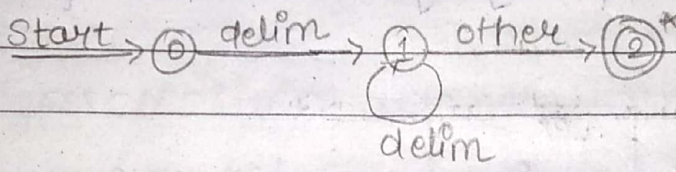
digit+ or digit.digit*



* Draw a Transition Diagram to recognize [white space. (Blank, [Tab, newline].

ws$^+$ or delim$^+$

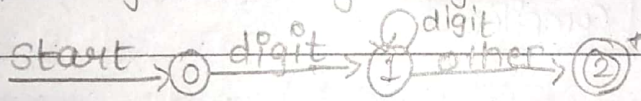
delim$^+$ or delim . delim*



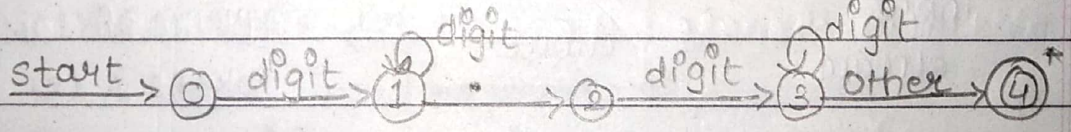
* Draw Transition Diagram For Real Numbers.

eg 10.25 , 27.1 , 123.789

digit$^+$ = digit digit* ← closure



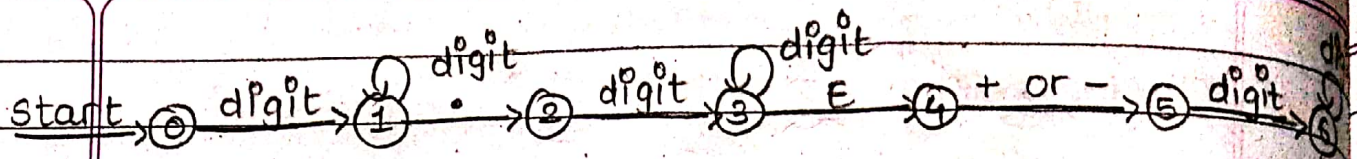
Final Diagram For Real Numbers.



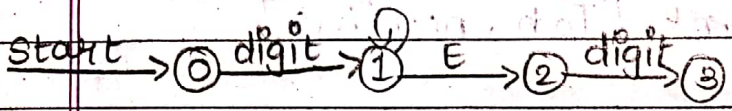
→ Draw Transition Diagram for exponent numeric value.

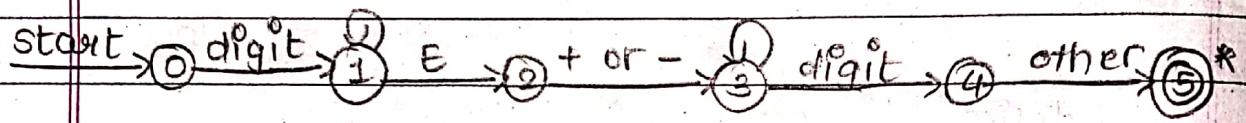
Eg. 23.25 E+5, 125.1 E-20, 78 E30, 71 E+23

93 E-235 (. => period in real no.)

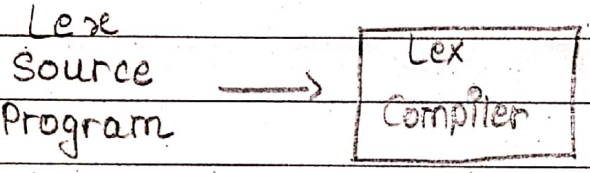
1. 

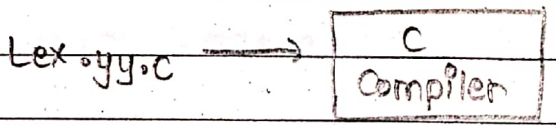
[self loop for more digits].

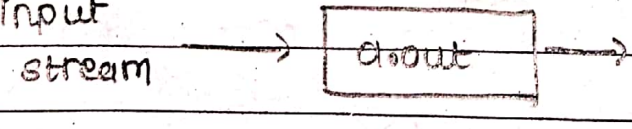
2. 

3. 

* A language for specifying lexical analyzer.







A lex program consist following 3 parts:

1. Declaration
2. Translation Rules
3. Procedures

→ Finite Automata.

A recognizer for a language is a program that takes as input string x and answers yes if x is a sentence of the language and no otherwise.

Compile a Regular expression into a recognizer by constructing a generalized transition diagram called Finite automaton.

A finite automaton can be deterministic or non-deterministic.

* Types of Finite Automata

(1) Non-Deterministic Finite Automata (NFA)
(2) Deterministic Finite Automata (DFA).

NFA :- Non-Deterministic Finite Automata.

Non-Deterministic Finite Automata is a mathematical model consist of

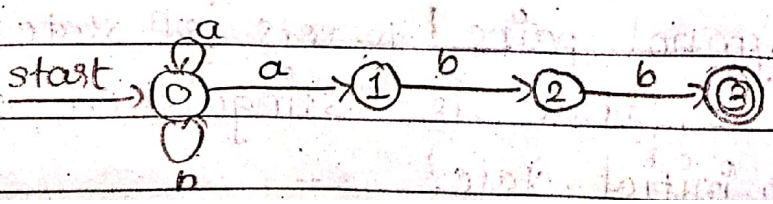
(1) A set of states S.
(2) A set of input symbols ∈ (sigma)
(3) A transition function move that maps state symbol pairs to sets of states.
(4) A state s_0, that is distinguished as the start or initial state.
(5) A set of states F distinguished as accepting or final states.

Date
Page 26

	NFA	DFA
1.	Input label ε(sign) is allowed	It is not allowed at all.
2.	NFA is a kind of mathematical model.	It is a special kind of NFA / Extension of NFA.
3.	Edges living from a particular state may have identical input symbols.	Edges living from particular state should have distinct input symbols
4.	NFA diagrams are smaller than DFA diagrams.	DFA diagrams are bigger than NFA diagrams.
5.	Backtracking is possible.	There is no chance of backtracking.
6.	NFA works less efficiently than DFA	They are faster than NFA.

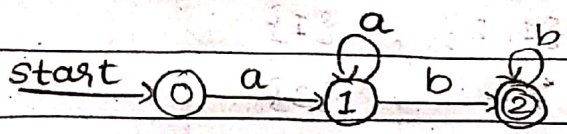
→ Draw NFA diagram for following regular expression and also prepare Transition Table for same.

1. $(a|b)^* abb$



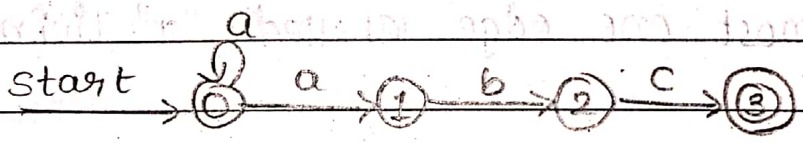
State	Input Symbols	
	a	b
0	{0,1}	{0}
1	–	{2}
2	–	{3}
3	–	–

2. $aa^*|bb^*$



State	Input Symbols	
	a	b
0	{1}	{0} –
1	{1}	–
2	–	{2}

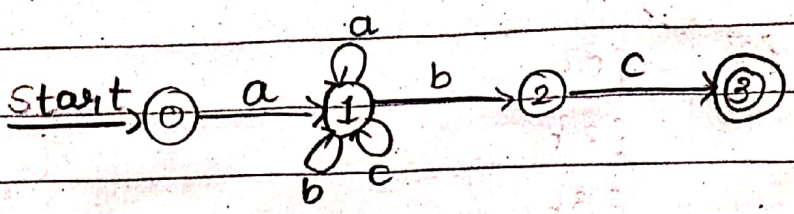
3. a^*abc



State	Input Symbols		
	a	b	c
0	{0,1}	–	–
1	–	{2}	–
2	–	–	{3}
3	–	–	–

Date
Page 28.

4. a (a|b|c)* bc



State	Input Symbols		
	a	b	c
0	{1}	–	–
1	{1}	{1, 2}	{1}
2	–	–	{3}
3	–	–	–

DFA :- Deterministic Finite Automata.

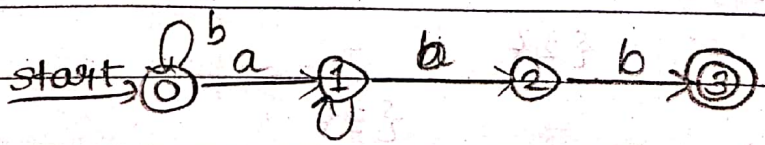
Deterministic Finite Automata, they are special case of NFA in which

(1) No state has an Epsilon (ε) transition

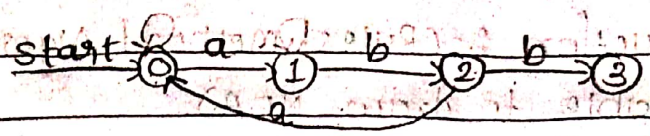
(2) For each state s and input symbol A there is atmost one edge labelled "a" living state s.

→ Draw DFA diagram for following Regular Expression

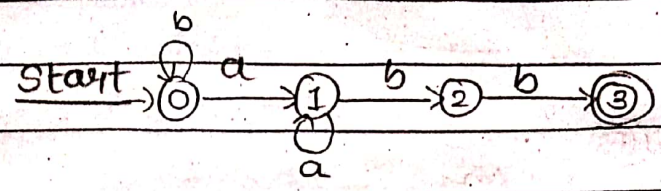
(a|b)* abb.



→ abaabb.

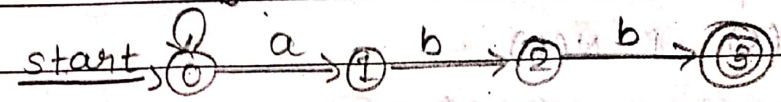


→ baabb



→ aaabb

→ bbabb



Algorithm Simulating DFA

```
S := s0;
c := nextchar;
while c ≠ eof do
    S := move(S,c);
    c := nextchar;
end;
if S is in F then
    return "yes";
else
    return "no";
```

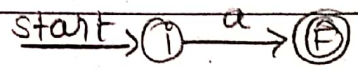
→ Thompson's Construction :- (Rules/ method/ Algorithm)
[Only Applicable to draw NFA).

1. Regular Expression = ϵ

NFA

2. Regular Expression = a

NFA

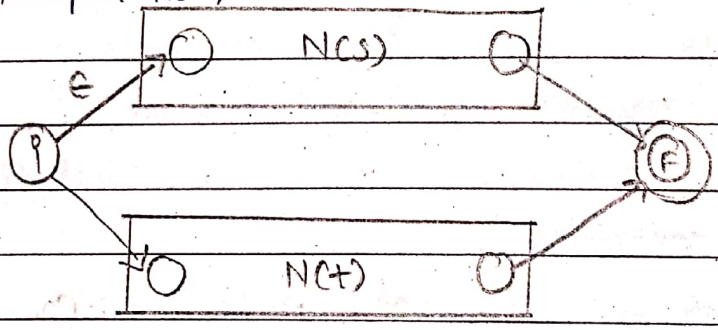


3. Regular Expression s & t

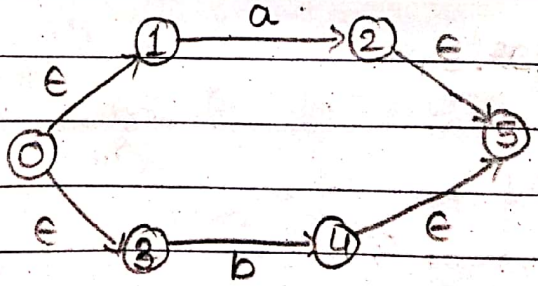
NFA : N(s) & N(t)

Union.

Union Operation structure from thompson's rule.



Ex. (a|b)



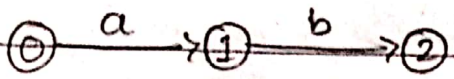
only the way for one time occurance.

Note:- For union Diamond-Shape is compulsory, and also having six states including initial state and final state.

4. S & t

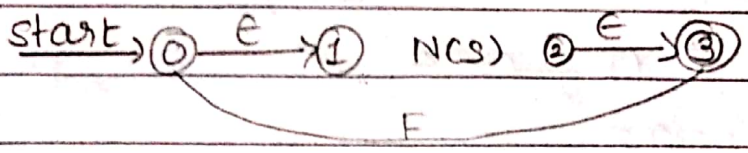
N(s) N(t)

Ex: a b



5. Closure

Consider Regular Expression, N(s)



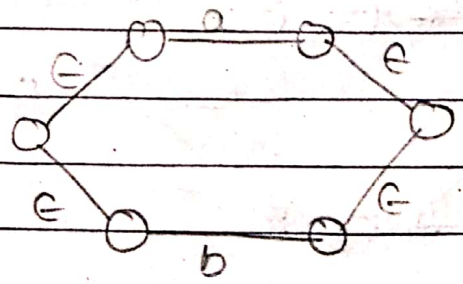
→ As per thompson construction we cannot have self-loop.

Q. Draw NFA diagram for the following given Regular Expression using Thompson Construction Rules. Also provide decomposition structure for the same.

→ $(a|b)^* abb$.

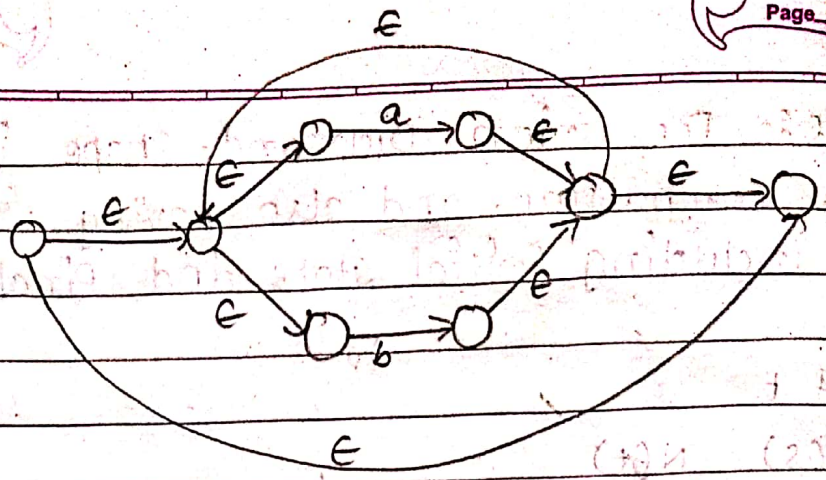
r_1 = 

r_2 = 

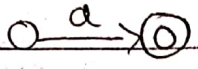
r_3 = 

$r_4 = (r_3)$

r_5 =

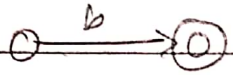


r_6 =

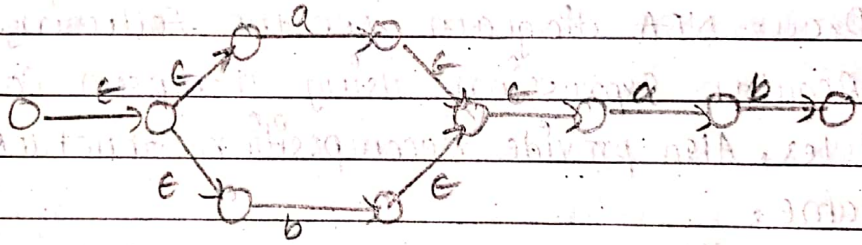


r_7 =

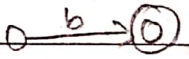
r_8 =



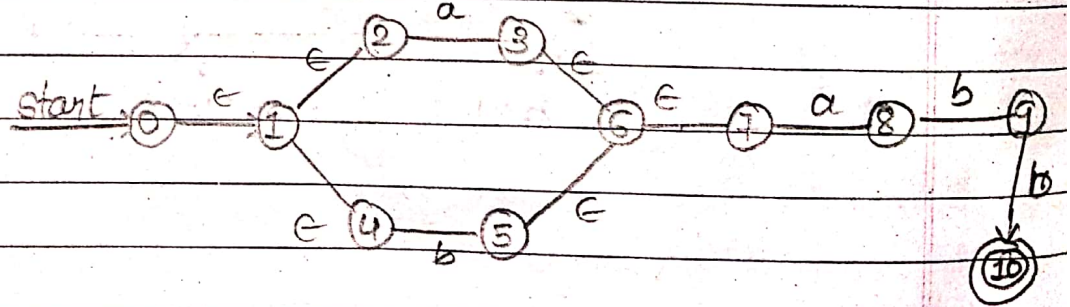
r_9 =



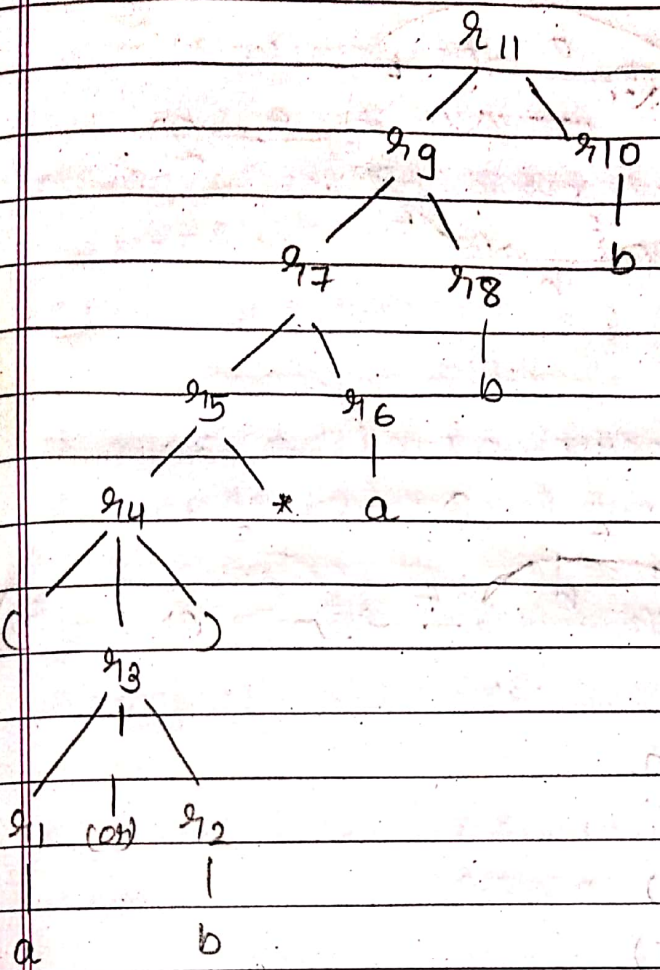
r_{10} =



r_{11} =

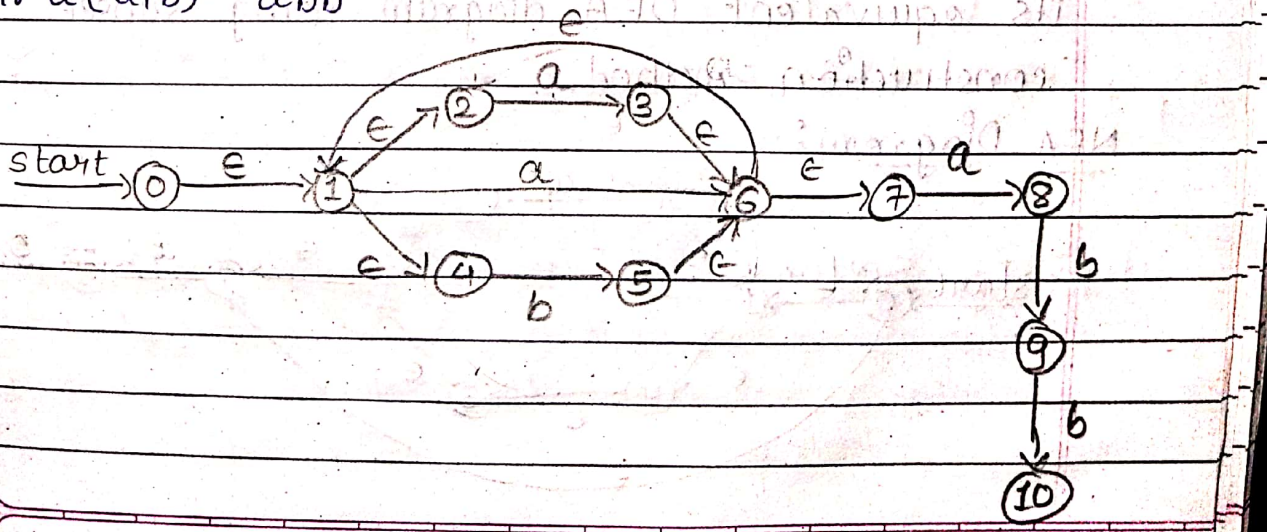


Decomposition Diagram

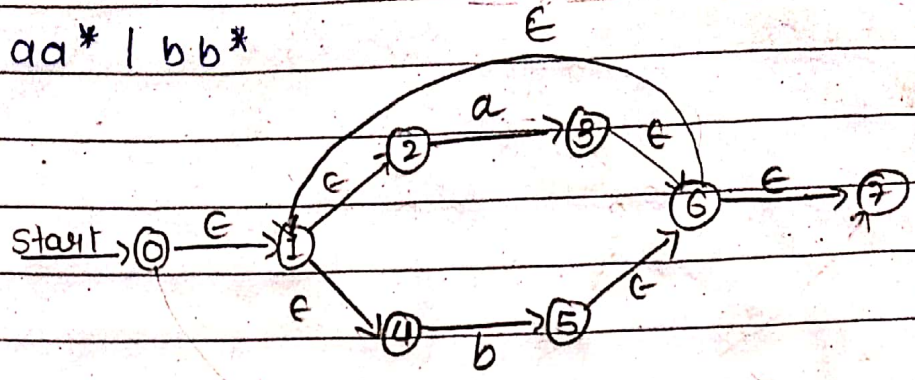


Q. Draw NFA diagram using Thompson's construction rules for the following Regular Expression.

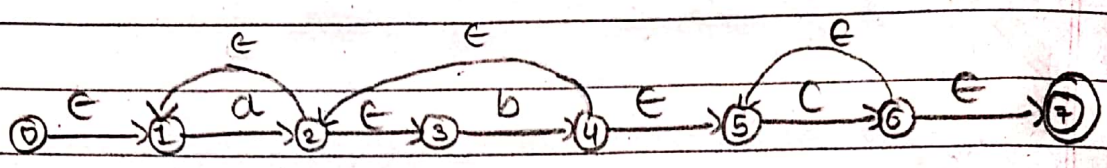
1. a(a|b)* abb



2. $aa^* \mid bb^*$



3. $a^* a b^* b c^* c$

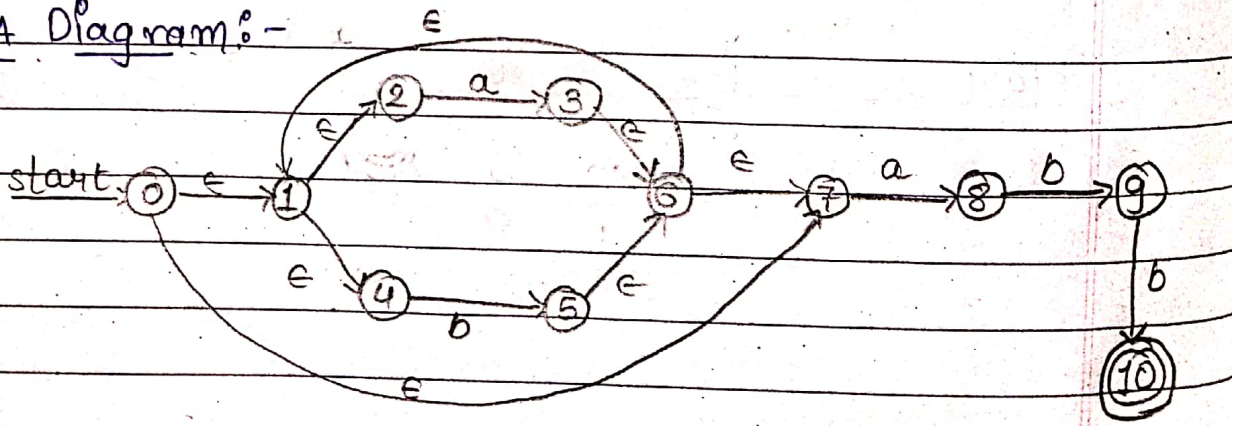


→ Subset Construction.

1. ϵ - closure (s)
2. ϵ - closure (T)
3. move (T, a)

Q. Draw NFA diagram using Thompson's construction rules for the regular expression. Convert it into its equivalent DFA diagram using subset construction method.

NFA Diagram:-



ε - closure (0)

A = {0, 1, 7, 2, 4}

mov (A, a) = {3, 8}

ε(mov(A, a)) = {3, 8, 6, 7, 1, 2, 4} = B

A = {5}

ε(mov(A, b)) = {5, 6, 7, 1, 2, 4} = C

ε(mov(B, a)) = {3, 8}

ε(mov(B, b)) = {5, 9, 6, 7, 1, 2, 4} = D

ε(mov(C, a)) = {3, 8} = {3, 8, 6, 7, 1, 2, 4} = B

ε(mov(C, b)) = {5}

ε(mov(D, a)) = {3, 8} = B

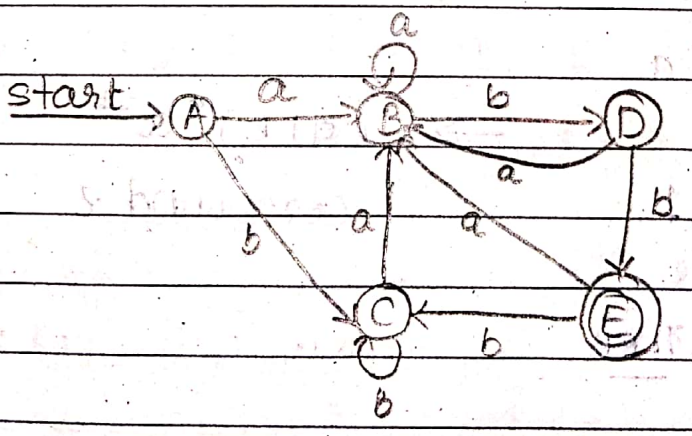
ε(mov(D, b)) = {10, 5} = E

E = {5, 10, 6, 1, 7, 2, 4}

ε(mov(E, a)) = {8, 3} = B

ε(mov(E, b)) = {5} = C

DFA :-



Algorithm

initially, ε-closure(S_0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates
do begin
    mark T;
    For each input symbol a do begin
        U := ε-closure (move (T, a));
        IF U is not in Dstates then
            add U as an unmarked state to Dstates
        Dtran [T, a] := U
    end
end

- Regular expression to DFA :-

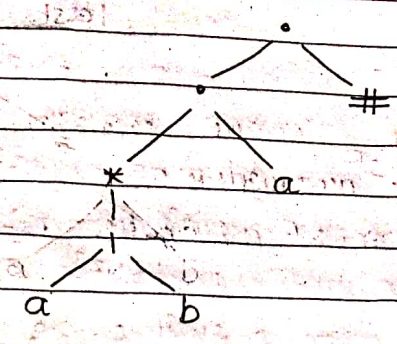
Conversion of Regular expression to DFA

Cast without using NFA

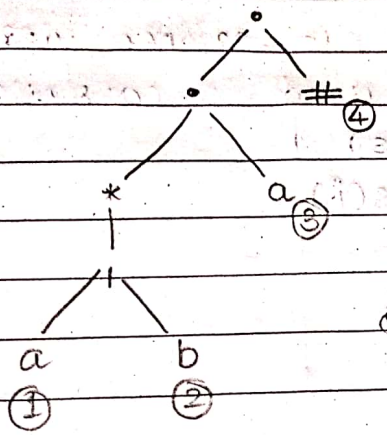
-> R.E. (a|b)*a

step(1) (a|b)* a → (a|b)* a# Argumented
compound 1 compound 2

step:-(2) Syntax Tree

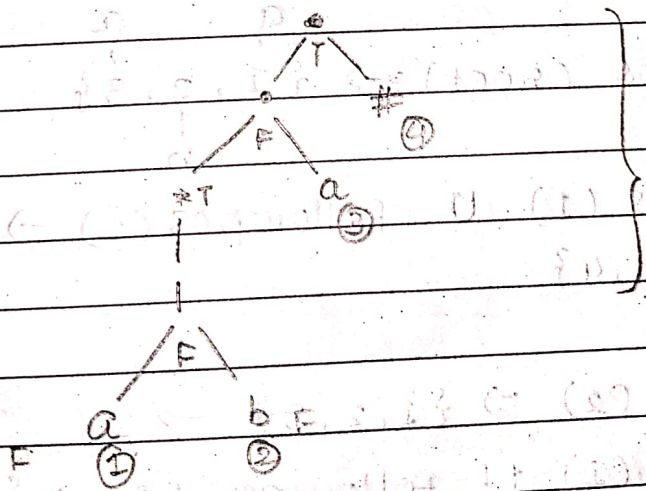


Step 2(3) Numbering of Nodes. (except ∈)



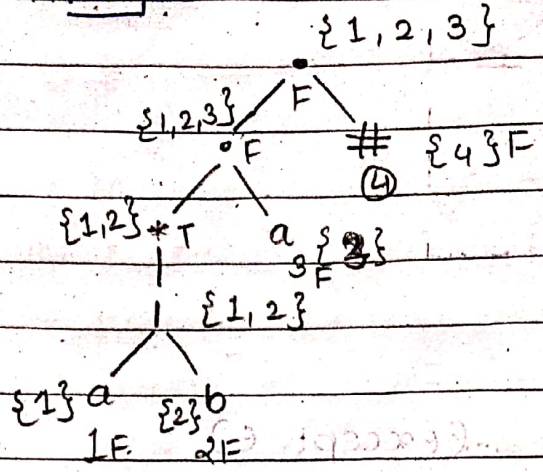
each symbol is numbered

Step 2(4) Firstpos, Lastpos, nullable

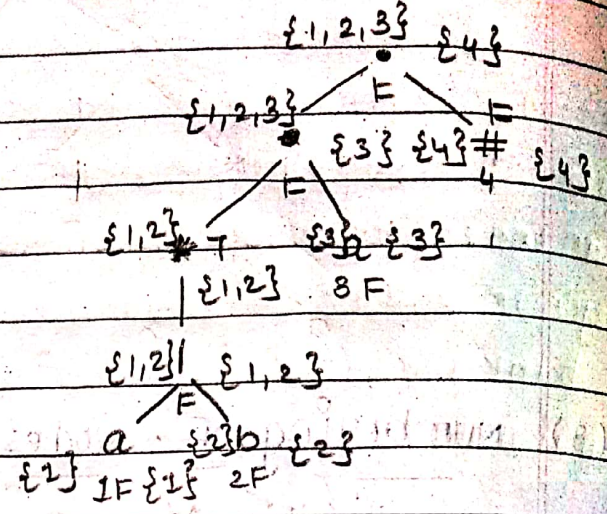


Nullable

Firstpos



Lastpos



Step :- 5 Followpos

(Firstpos) Node	(Lastpos) Followpos(i)
1	{1,2,3}
2	{1,2,3}
3	{4}

Step :- 6 State Generation

S_1 = Firstpos (root) => {1, 2, 3} (1 → a, 2 → b, 3 → a)

- a : followpos (1) ∪ followpos (2) → {1,2,3}
  S_1 = {1,2,3,4}

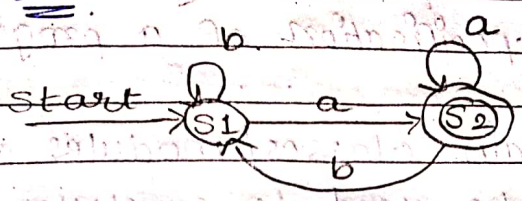
- b : followpos (2) → {1,2,3} → S_1

- a : followpos (1) ∪ followpos (3) → {1,2,3}
- b : followpos (2) → {1,2,3} → S_1

Step :- 7 T.T. (Transition Table)

State	Input symbol	
	a	b
S_1	S_2	S_1
S_2	S_1	S_1

Step:-8 DFA :-



Unit:-3

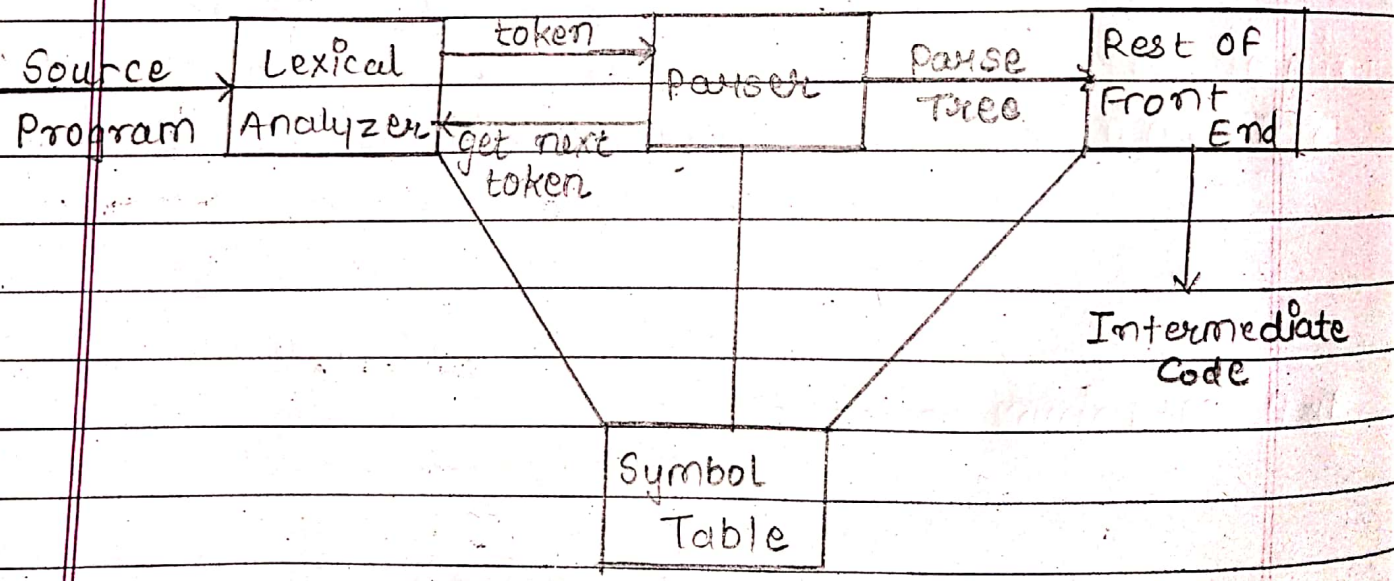
Syntax Analysis

Every programming language has rule that prescribe the syntactic structure of well formed programs. Following are same advantages of syntax analysis:

(1) Grammar gives a precise, easy to understand and syntactic specification of a programming language.

(2) There are certain classes/modules of grammar available can be used to construct an efficient parser.

=> The Role of the Parser:-



The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

It is also to get outcome in the form of syntax errors &.

Types of Parser:-

1. Universal Parsing Method
2. Top-down Parser
3. Bottom-up Parser

Types of error

1. Lexical , 2. Syntactic, 3. Semantic, 4. Logical

- Goals of Error Handler

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

- Error Recovery strategies.

1. Panic Mode
2. Phrase Level Recovery
3. Error Productions
4. Global Corrections.

- Context- Free Grammar [CFG]

Many Programming languages constructs have an inherently recursive structure that can be defined by context free grammar.

eg. if s1 and s2 are statements and E is expressions

=) if E then s1 else s2.

CFG :- stmt → if expr then stmt else stmt
T NT T NT T NT

- Components of CFG

1. Terminal

Terminals are the basic symbols from which strings are formed.

The word Token is a synonym for Terminal when we are talking about grammar for programming language.

2. Non-Terminal

They are syntactic variables that denote sets of strings.

3. Start-Symbol

One Non-Terminal is distinguished as the start symbol and the set of strings, it denotes is the languages defined by the grammar.

[Always find on left hand side of Regular Expression]

4. Production

The production of grammar specify the manner in which the terminals & non-terminal can be combined to form strings.
Each production consist of a non-terminal followed by an arrow, followed by string of non-terminals and terminals.

Example:-

Notational Conventions

1. Terminals
   → small case letters [a, b, c, ...]
   → digits
   → operators
   → punctuation marks / separators
   → bold face strings

2. Non-Terminals
   → uppercase letter [A, B, C, ...]
   → lowercase *italic* string
   → S

3. Uppercase characters like U, V, W, X, Y, Z
   [Late in series and can act as terminals and non-terminals]

4. Lowercase characters like u, v, w, x, y, z
   [Late in series and can act only as terminals]

5. Greek letter [Arbitrary strings]
α, β, γ - here, T/NT/TNT production rule.

6. $A \to \alpha_1$, $A \to \alpha_2$, $A \to \alpha_3$, ... , $A \to \alpha_n$

7. whatever symbol available on L.H.S. of arrow sign represents start symbols.

* Derivations [Providing alternative information]

Here, production is treated as rewriting rule in which the non-terminal on the left is replaced by the string on the right side of the production.

eg. $E \Rightarrow -E$

E "derives" $-E$

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

→ In abstract string $\alpha A \beta \Rightarrow \alpha \gamma \beta$

if $A \to \gamma$ is a production

if $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \ldots \Rightarrow \alpha_n$

we say $\alpha_1 \Rightarrow \alpha_n$

1. $\Rightarrow$ derives in one step
2. $\overset{*}{\Rightarrow}$ derives in 0 or more steps
3. $\overset{+}{\Rightarrow}$ derives in 1 or more steps

Purpose of derivations

1. $\alpha \overset{*}{\Rightarrow} \alpha$ for any string α.
2. $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$

→ Define sentence.

$S \overset{*}{\Rightarrow} w$ ~~must be terminals~~.

→ Define Sentencial form

$S \overset{*}{\Rightarrow} \alpha$ can be T/NT/TNT Both

A sentence is a sentencial form with no non-terminals.

Derivation

eg. $E \to E + E$

$E \to E * E$

$E \to (E)$

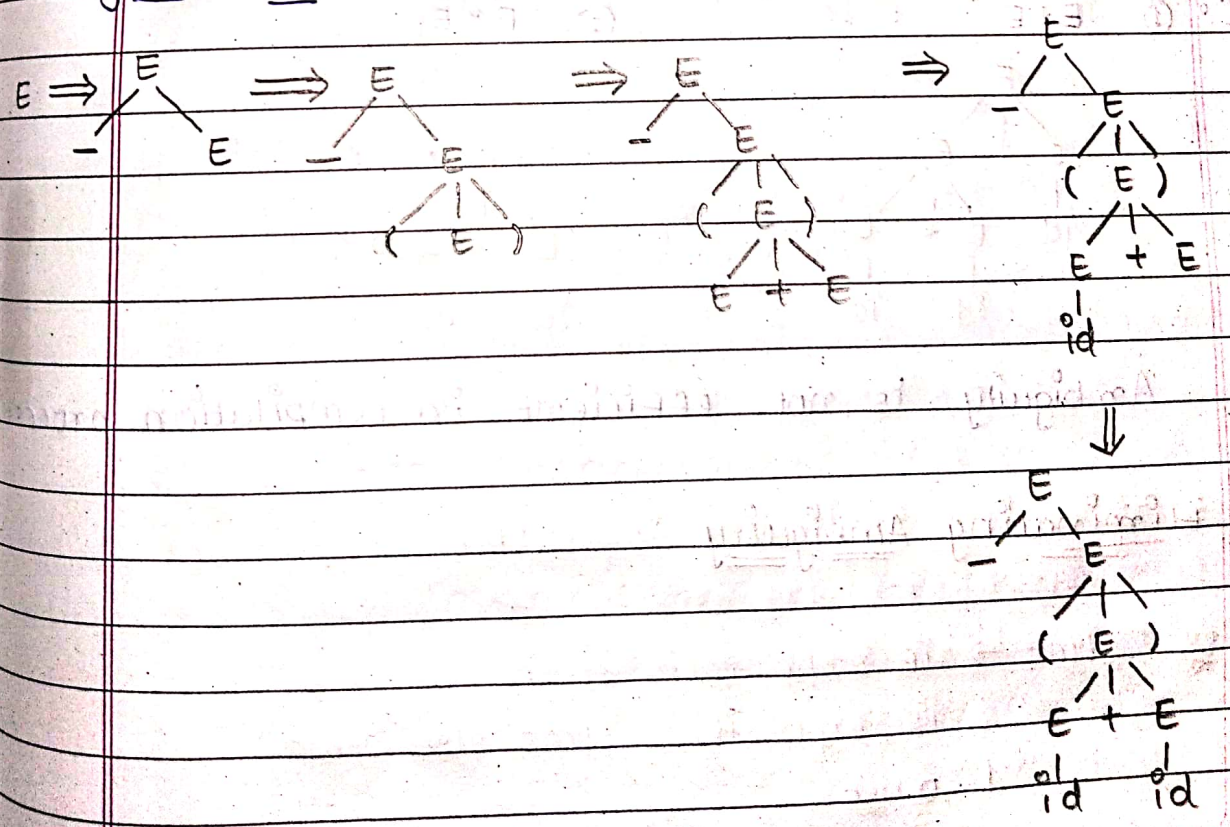
$E \to -E$

$E \to id$

Input :- $-(id+id)$

$E \overset{lm}{\Rightarrow} -E \overset{lm}{\Rightarrow} -(E) \overset{lm}{\Rightarrow} -(E+E) \overset{lm}{\Rightarrow} -(id+E) \overset{lm}{\Rightarrow} -(id+id)$

Syntax Tree



Input :- id + id * id

Above input may have double or two possibilities

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

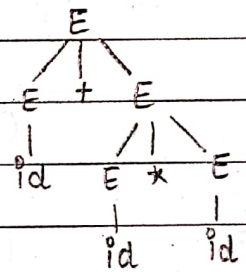
$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

Ambiguity :- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

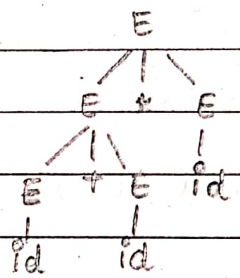
OR

An ambiguous grammar is one that produces more than one left most or more than one right most derivation for the same sentence.

Ex :- ① E + E



② E * E.

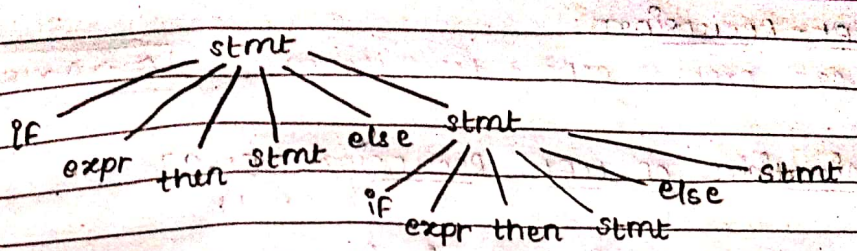


Ambiguity is not efficient in compilation process.

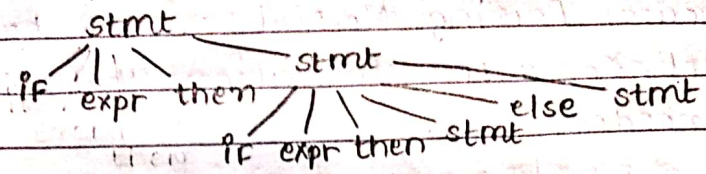
* **Eliminating Ambiguity**

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

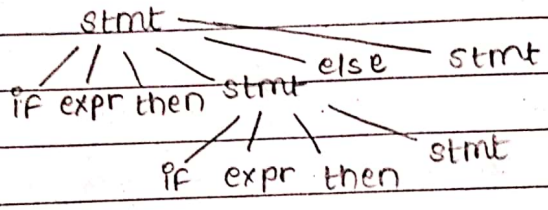
(i)



(ii)



(iii)



→ stmt → matched_stmt
| unmatched_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

Date
Page 48

- Left Recursion.

eg. expr → expr + term | term

$A \to A\alpha | \beta$

* Elimination of Left Recursion [L.R]

A Grammar is L.R. if it has a non-terminal, A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Top-Down Parsing Method cannot handle L.R Grammars. So, a transformation that eliminates L.R. is needed.

eg. $E \to E+T | T$ $\quad A \to A\alpha | \beta$

$T \to T*F | F$ $\quad A \to A\alpha | \beta$

$F \to (E) | id$

(i) $E \to TE'$

$E' \to +TE' | \epsilon$

(ii) $T \to FT'$

$T' \to *FT' | \epsilon$

(iii) $F \to (E)$

$F \to id$

$\Rightarrow A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$

$A \to \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$

$A' \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$

Algo. Eliminating Left Recursion

1. Arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$
2. for $i := 1$ to n do begin
   for $j := 1$ to $i-1$ do begin
   replace each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
   where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions; end
   eliminate the immediate left recursion among the A_i-productions
   end

→ Eliminate L.R. in the following given grammar.

$S \to Aa \mid b$ —①

$A \to Ac \mid Sd \mid \epsilon$ —②

[1] $S \to Aa \mid b$

using eq. —②.

$S \to Sda \mid b$ —③

[2] $A \to Ac \mid Sd \mid \epsilon$

using eq. —①

$A \to Ac \mid Aa \mid bd \mid \epsilon$ —④

$S \rightarrow Sda|b$ —③

$A \rightarrow Ac|Aa|bd|\epsilon$ —④

$A \rightarrow A\alpha|\beta$ —⑤ $\Rightarrow A \Rightarrow \beta A'$

$\Rightarrow A' \Rightarrow \alpha A'|\epsilon$

→ Left Factoring.

It is a grammar transformation that is used for producing a grammar suitable for predictive passing.

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

Algorithm Left Factoring

I/P : G (Grammar)

For each N.T. A find the longest prefix α common to two or more of its alternative.

If $\alpha \neq \epsilon$, that is, there is a non-trivial common prefix replace all the capital A productio $A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$, where γ represents all alternatives that do not begin with α by

$A \rightarrow \alpha A'|\gamma$ —①

$A' \rightarrow \beta_1|\beta_2|\ldots|\beta_n$ —②

Here, A' is a new N.T. Repeatedly apply this transformation until no two alternatives for a N.T. have a common prefix.

→ $S \to iEtS \mid iEtSeS \mid a$ —①

$E \to b$ —②

For $A \to \alpha A'$

$A \to iEtSA' \mid a$

$A' \to \epsilon \mid eS$

$E \to b$

$A \to \alpha\beta_1 \mid \alpha\beta_2$

⇓

$A \to \alpha A' \mid \gamma$

$A' \to \beta_1 \mid \beta_2$