studocu

Compiler Design Unit-1

Compiler Design (Jawaharlal Nehru Technological University, Hyderabad)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

UNIT- I

THE STRUCTURE OF COMPILER

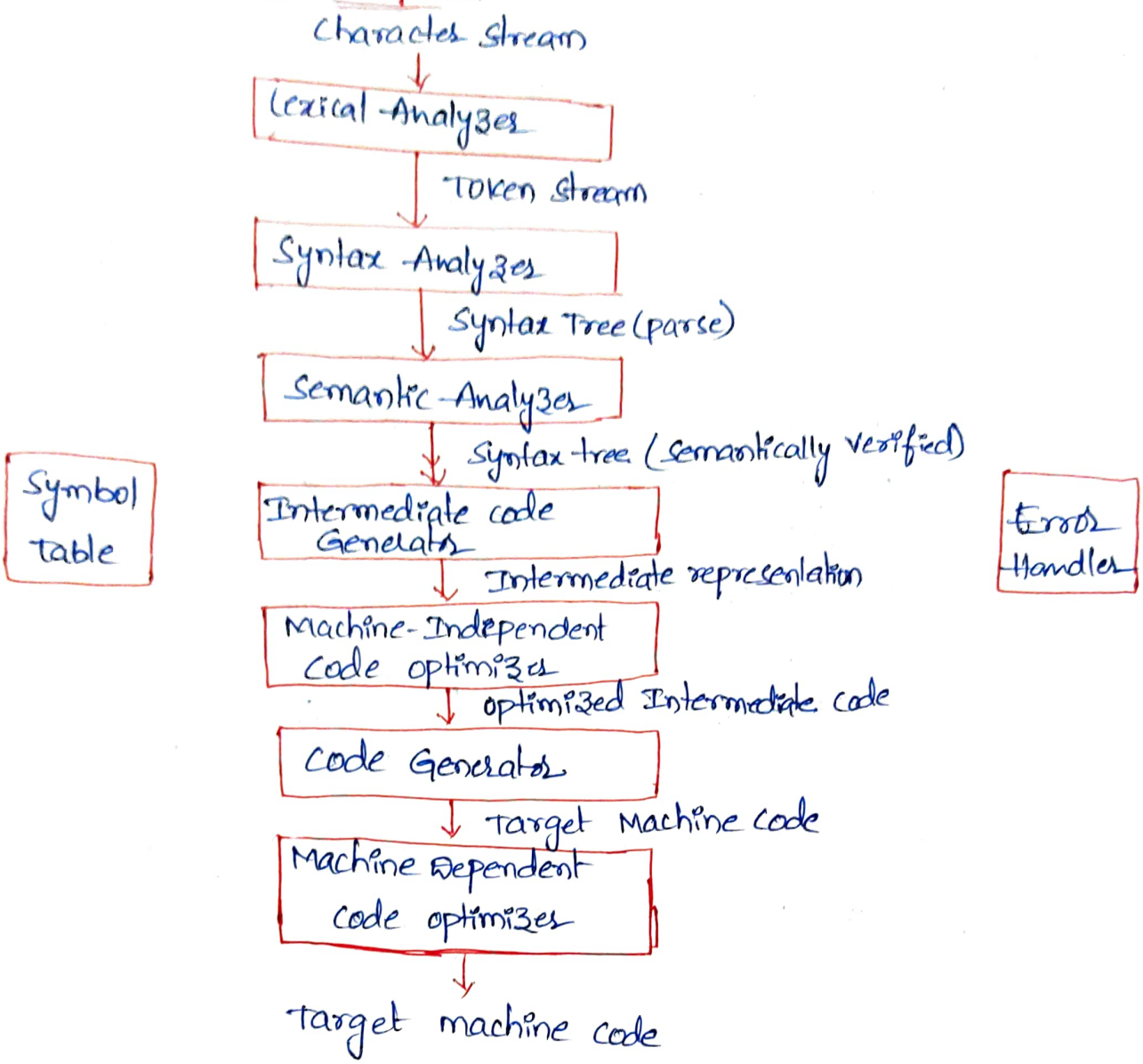


Fig: phases of the compiler

* All the phases are interact with the symbol table & Error Handler

(i) Lexical Analysis:-

↳ Lexical Analyzer is also called as scanner.

↳ LA reads the source program character by character and converts the corresponding characters into ~~lexeme~~ meaningful sequence called "lexeme."

lexeme → A sequence of characters.

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

↳ For each lexeme the lexical Analyzer produces "TOKEN" as o/p

↳ Token may be a keyword, identifier, operator, constant

↳ Token is represented as <Token-name, attribute-value>

Name of the variable
id component
2nd component
↓ points to entry in symbol table entry

Eg:- Assignment stmt a = b+c*90

lexeme	Token
a	identifier
=	Assignment operator
b	identifier
+	Addition operator
c	identifier
*	Multiplication operator
60	constant

* a=b+c*90
* <id,1>,<=>, <id,2>,<+>
<id,3> <*>, <60>
* operators & constants doesn't contain attribute value.

id1 = id2+id3*60
↓
o/p of Lexical Analyzer

↳ Tol: Symbol Table is a data structure is used to store the token information (such as what is variable name, value, & scope of the variable ---)

Syntax Analyzer:-

↳ Syntax Analyzer is also called as parser.

↳ SA will checks whether the corresponding syntax of the source program is correct or not.

↳ If the syntax is not correct, Error Handler reports error to the user.

↳ If the syntax of source program correct,

↳ Syntax Analyzer takes i/p as tokens and creates tree like intermediate representation as called the syntax tree.

↳ In the syntax tree, there are two nodes exists
(i) Interior nodes → represents operations
(ii) children node → represent arguments of the operation

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

while Evaluating the expression:-

* The operator which is having higher priority will be evaluated first followed by lowest priority operator.

$id_1 = id_2 + id_3 * 60$

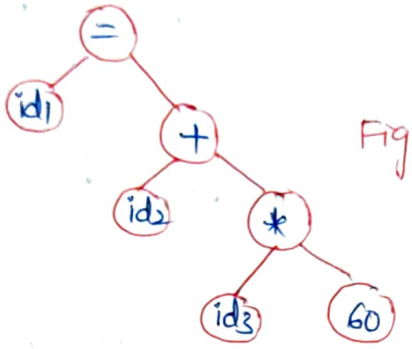


Fig:- parse tree

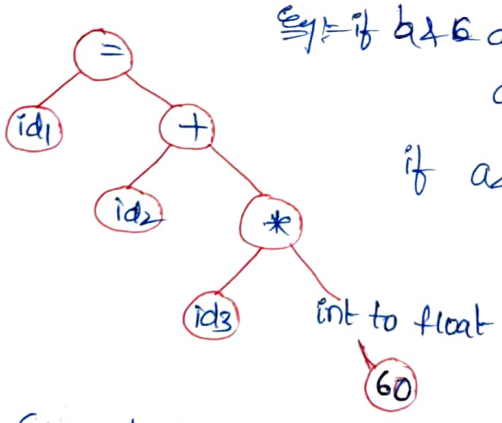
Semantic Analyzer:-

→ The parse tree is given as an i/p to semantic Analyzer.

→ Semantic Analyzer will check whether the meaning of the parse tree is correct or not (semantically verifies the parse tree)

→ For verifying the parse tree semantically, it uses a sub called type checker, it checks the data types of variable, if there any need of type conversion then it will perform type conversion.

Eg:-



Eg:- if b & c are int, 1 & 2

$a = b + c * 60 \Rightarrow 1 + 2 * 60 \Rightarrow$

if a & b are float 1.5, 2.5

$a = 1.5 + 2.5 * 60 \Rightarrow$

Fig:- Semantically verified parse tree.

Intermediate code Generator:-

↳ Semantically verified parse tree is given as an i/p to ICG. and it produces intermediate representation as o/p.

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

↳ In process of translating source program into target code compiles may construct one or more intermediate representation (such as, three address code, polish Notations)

↳ Syntax tree is also one of the Intermediate representation.

3-Address code:- 3-imp rules are

① 3-address code assignment instruction should have atmost one operator on RHS (should not have more than operator on RHS)

② Compiles must generate a temporary variables for storing result.

③ some instruction may contain fewer than 3-operands.

Eg:- $id_1 = id_2 + id_3 * 60$

$t_1 = \text{int to float}(60)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

$t_1, t_2, t_3 \rightarrow$ are temporary variables

Code optimization:-

→ Intermediate code representation is given as an i/p to code optimation

→ Code optimization is process of eliminating unnecessary statements (shorter code that executes fast, consumes less power).

$t_1 = id * 60.0$

$id_1 = id_2 + t_1$

Code Generator:-

→ It takes optimized code intermediate code from code optimizer and generate machine code, in

→ In order to produce machine code it uses some assembly languages.

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

```
LD   Reg, M                LDF   R2, id3
ST   Memory, Reg.          MULF  R2, R2, #60.0
MUL  R1, R2, R3            LDF   R1, id2
                           ADDF  R1, R2
                           STF   id1, R1
```

Error Handles:-

↳ Errors Handles interact with all the phases of the compiles

→ If any errors reports the errors in present any of the compiles phase, to the uses.

Grouping of phases into passes:-

→ In implementation

(1) Single pass compiles :=

* All the phases of compiles are grouped into one part.

(2) Two-pass compiles:-

* Also called as multi-pass compiles.

* The phases of compiles into are grouped into two-parts.

Single pass compiles:-

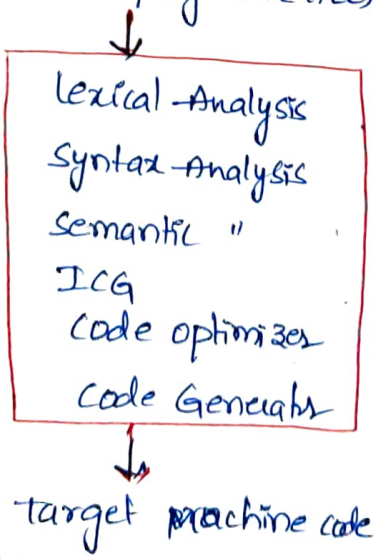


Fig:- Single pass compiles

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Two pass compiles :-

1. Analysis part (or) Front end of the compiles
2. Synthesis part (or) Backend of the compiles

1. Analysis part :-

↳ For analysing the code, we are using analysis part.

↳ It breakesup the source program into pieces and imposes grammatical structure on them, and these structure is used to create intermediate representation source program.

SP → C

C

T.M.C (Assembly lag)

→ Analysis part contains the phases that are dependent on source language and independent on target program.

→ It detects the errors in source program syntactically & semantically

→ It collects the information about the sp and stored it in symbol Table

Syntesis part :-

↳ It focuses contains the phases that are dependende on Target language and independept of source program.

↳ It construct the Target program from Intermediate representation stored in the symbol table.

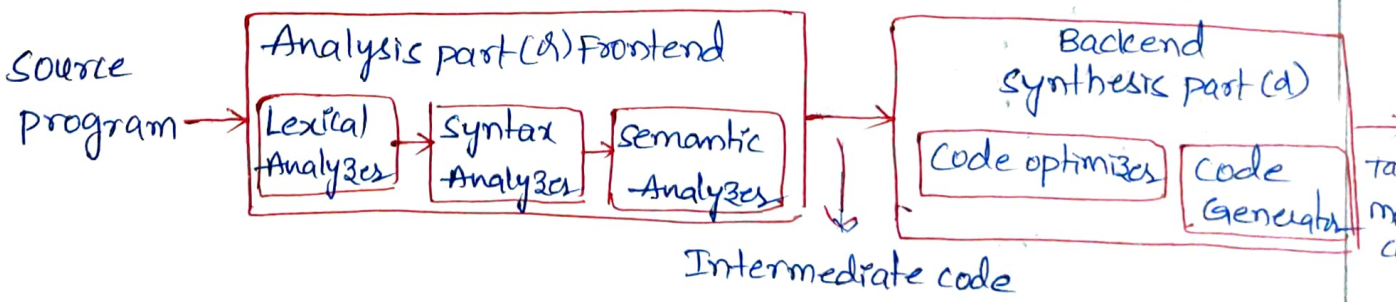


Fig :- Two-pass (or) Multi-pass compiles

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

(2) THE SCIENCE OF BUILDING A COMPILER:-

↳ A compiles must accept all the source program, that com follow the language specifications.

↳ A source program is infinite & any program can very large, consisting million lines of code.

↳ Any transformation performed by the compiles while translating source program must preserve the 'meaning of program being compiled.

→ compiles writers must not only have complete idea of the compiles they create, but all the programs that their compileres compiles.

Modeling in compiles Design and Implementation:-

→ The study of compiles is mainly a study of a how we design the right mathematical models & choose the right algorithm.

→ Some of the most Fundamental models are.

(i) Finite state machine and Regulas Expressions are useful for describing the lexical units of programs & for describing algorithms used by the compiler to recgoize those units.

(ii) Contex Free Grammar is used to describe the syntactic structure of the programming languages.

The science of code optimization:-

→ The "optimization" in compiles design refers to the attempts that a compiles makes to produce code that is more efficient than the obvious code.

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Compiles optimization must meet the following objectives.

- The optimization must be correct, that is preserve the meaning of compiled program.
- The optimization must improve the performance of many programs.
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.
- It is not guarantee that the optimizing compiles should produce completely errors free code.

Applications of compilers:=

→ Implementation of High level languages.
→ optimization for computer Architectures
→ Design of New computer Architecture
→ Program Translation
→ S/w productivity tool

Compiles Construction tools:=

→ compiles writers, any s/w developer, use s/w development tools such as editors, debuggers, version managers, profiles, test harness...
→ In addition to these tools, more specialized tools have been created to implement various phases of compiles.

1. lex tool - scanner generator
   It takes i/p as RE & produces o/p as stream of tokens.
2. parser generator := YACC tool - it generates parse tree.
3. SDD engines - collection of functions are used to traverse the parse tree & generate Intermediate code.
   ↓
   Semantic Analysis
4. Intermediate code generator generators.

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

4. Data Flow Analysis Engine - code optimization is done.

6. compiler - construction tool kit → provide an integrated set of routines for constructing various phases of compiler.

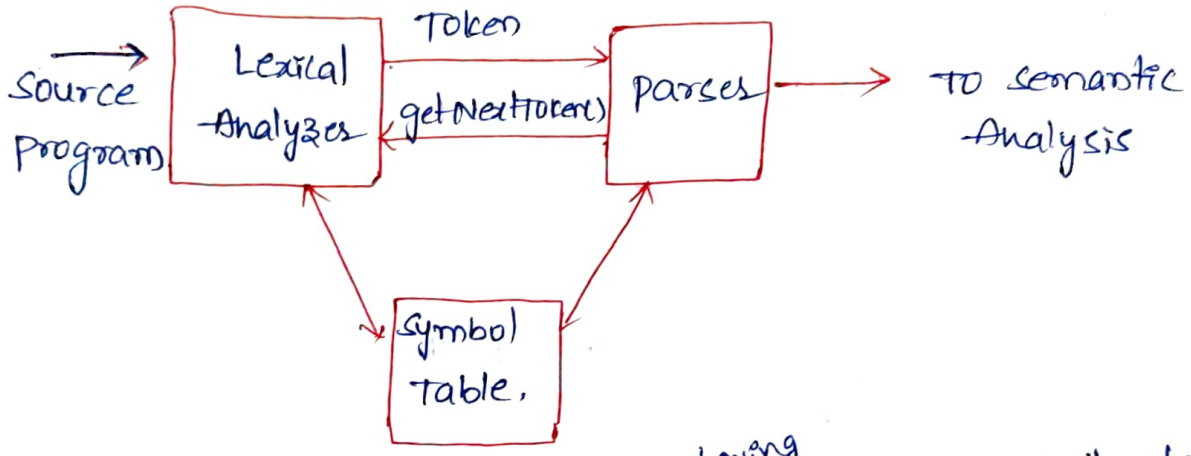
Role of the Lexical Analyzer :-

→ It is the first phase of the compiler.

→ The main task of lexical Analyzer is to read the i/p characters from the source program, and group them into lexemes and produce o/p a sequence of tokens is sent to parser for syntax analysis.

→ LA enter lexemes consisting an identifies into symbol table.

→



- An interaction is implemented by having the parser call the lexical Analyzer
- The call suggested by the NextgetToken() command, causes the lexical Analyzer to read stream of characters from its i/p until it can identify the next lexeme and produces the next token, which it returns to the parser.

This document is available on studocu
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

(1) The functions of lexical Analyzer:

① It produces a stream of tokens.

② It removes comments from the source program.

③ It also removes white spaces characters (such as blank, newline, tab spaces.

Eg:- #include <stdio.h> main() int a,

④ It counts the line numbers of source

⑤ It generates a symbol Table which stores information about identifiers

⑥ It provides errors messages with corresponding line no & column no's.

⑦ If the source program uses macro processor, the expansion of macros may be performed by the lexical-Analyzer.

Lexical Analyzer is divided into two processes.

(i) Scanning: consists of simple processes that dont require tokenization i/p, such as deletion of comments & whitespaces

(ii) Lexical-Analysis:- is more complex portion, which produces the Tokens from the o/p the scanner.

Lexical Analysis versus parsing:-

(2) Seperation of LA from SA:-

(i) TO simplify the design a compiler

(ii) TO increase the efficiency of compiler

(iii) TO enhance the portability of compiler (working on several platforms)

(i) If we combine these two phases into single phase then burden on a single phase is increased, so inorder to reduce the burden on signle phase we will seperate LA from SA. so that

LA → will do its own task

SA → will " " " " individually.

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

(2) In lexical Analyzes phase we are using Regulas expressions to recognize the tokens, RE are recognized with the help of Finite automata. → Syntax Analyzes uses CFG - pushdown automata)

* Specialized buffering techniques are used for reading i/p characters can speedup the compiles process syntactically.

(3) Tokens, Patterns, & Lexemes:-

Token:- Token is a pair which consists of two components.

<Token-name, attribute-value> → Token representation.
↳ A pointer which points to symbol table for token information.

Token name may be id, op, keyword & constant

Lexeme:-

lexeme is a sequence of meaningful characters. in source program.
→ The pattern for a token is identified by the LA as an instance of token.

E = m * c ** 2 (lexemes are matched agains the patterns)

Patterns:-

→ pattern is a rule for describing all these lexemes that can represent a particulas token in source language.

RE for id → (l/_)(l/d/_)* (* → 0 & more occuromce)
↓
alphabet & undercore l → [a-z A-Z _][a-z A-Z 0-9 _]

Token	Informal description	Sample lexeme
if	characters i, f	if
else	character e, lse	else
comparison	<or> in <=, or >=, or ==, !=	<=, !=
letter	letter followed by letter/digit	pi, score, D2
Number	any o number constant	

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Lexical Errors:

→ lexical Analyzer may not proceed if no pattern matching the prefix of remaining i/p

→ These Errors are generated during lexical analysis phase.

Commonly generated lexical errors:

1. Spelling Errors → (do → di, if → fi)
2. Un matched string — printf ("hai");
3. Appearance of illegal characters — printf ("hai"); $$ & ##
4. Exceeding the length of identifies → In C- max length of id is-32

Error Recovery Actions:

(i) Delete one character from the remaing i/p → di → Delete i, do & insert o.

(ii) Insert a missing character into the remaining i/p → printf("hai"); printf("hai")

(iii) Replace one character by another character → di, do

(iv) Transpose two adjacent characters → rp

(v) Delete successive characters from the remaining i/p until LA recognize a well-formed token (panic mode recovery)

INPUT BUFFERING:=

↳ Lexical Analyzer uses the input Buffering technique to read input characters from the source program.

↳ To read i/p characters from the source program LA uses two pointers

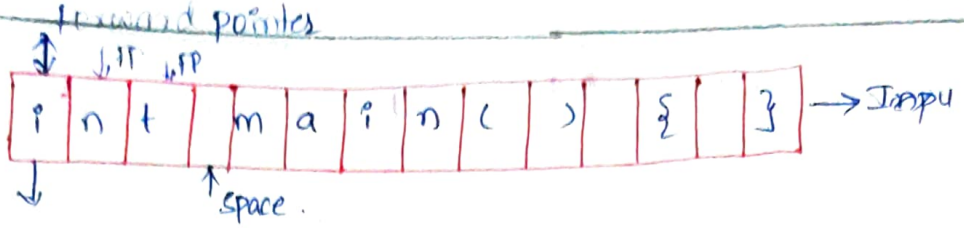
(i) lexeme Begin pointer

(ii) Forward pointer

Eg:= int main()
{
}

Program
↓ After saving stored in HDD
↓ at the compilation also resides HDP
Main memory (at the time of Execution)
↓
CPU

Downloaded by Moltapuri Himanshu (moltapurihimanshu@gmail.com)



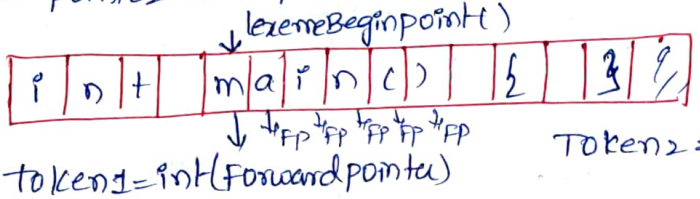
→ lexeme Begin pointer points to the Begining character of the lexeme.

→ Initially The Forward pointer is placed at begining character of lexeme.

→ first Forward pointer reads 1st character (i) and moves one position to the right to read the next character (n)....

→ whenever Forward pointer reaches a blank space then it identifies it as the end of current lexeme. so, the "int" is considered as Token.

→ The FP ignores the blank space, lexeme begin pointer and Forward pointer are placed at next character location.



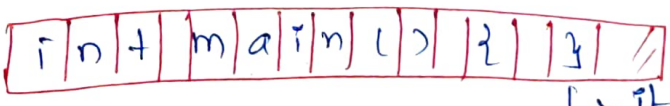
→ For each character from secondary memory (HD) one system call is required.

→ If the program size is very large, it requires more no. of systemcalls are required.

To overcome this problem, we are using Buffering Techniques.

Buffering:- Instead A Block of characters are to be read into the buffer using only 1 system call.

↳ The use of i/p buffering is instead of reading a single character from secondary memory, the block of characters are read.



↳ it requires only 1-system Call to read the prgrm

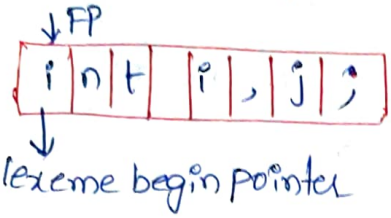
studocu This document is available on
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Buffering can be implemented in 2-ways.

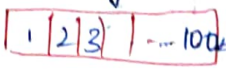
(i) One Buffer Scheme:

→ In this technique we are using only one buffer to read the input string.

→ In this technique also we are using (i) lexeme Begin pointer and (ii) Forward pointer to read the i/p characters from the buffer.



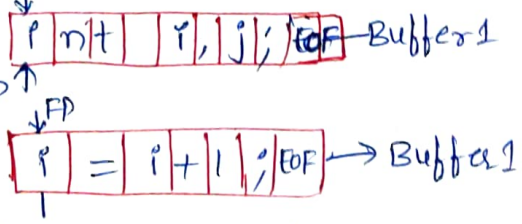
→ The problem with this approach is if the i/p string size is larger than than the capacity of the buffer, then the buffer has to be overridden inorder to store the remaining i/p string.

Eg:- i/p string = 500 Buffer size is = 100  overridden

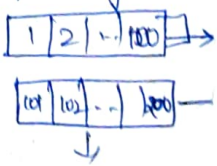
(*) This problem can be solved by using two-buffer scheme.

(ii) Two-Buffer Scheme:-

→ In this technique, it uses two Buffer to store i/p characters.



Eg:- i/p string = 500

1-100 

→ In this approach, 1st the buffer 1 is filled.

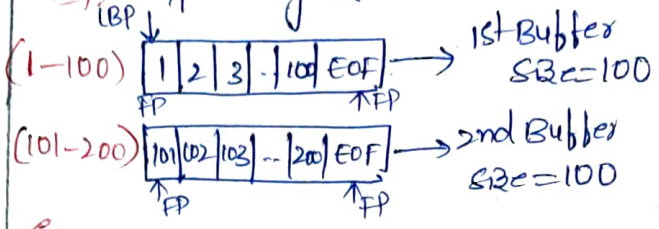
→ After the first buffer is completely filled, Then only we will fill the second buffer.

→ To determine whether the 1st buffer is completely filled or not, it uses a specail character called "Sentinal" & "EOF". (EOF → (End of the file & Sentinal character)

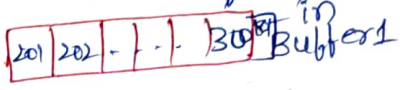
→ whenever the second buffer is completely filled, Then the remaing i/p string has to be filled in first buffer, Now the first buffer is "overridden"

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Eg:- i/p string size = 500



(After filling)




```
Eg:- switch (*forward++)
     {
     case 'eof':
       if ( forward is at end of 1st buffer)
       {
        forward= begining of 2nd buffer;
          refill 2nd buffer
       }
else if (forward is at end of 2nd buffer)
       {
        forward= beginning of 1st buffer;
        refill 1st buffer
       }
```

Recognition

RECOGITION OF TOKENS:=

Tokens are recognized Transition diagram.

(i) Recognition of identifiers
(ii) Recognition of delimeters
(iii) Recognition of Relation operators
(iv) Recognition of keywords (if, else, for...)
(v) Recognition of Numbers (int/float)

Recognition of identifiers:= The Regular definitions for tokens as follows

letter → a|b|---|z|..A|B|---|z|

digit → 0|1|---|9|

id → letter(letter|digit)*

Num → digit+ (digit+)?(E[+-]digits)?

studocu
This document is available on
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

if → if then → then else → else relop → <|<=|=|<>|>|>=

digit → [0-9] digits → digit+

delim → blank | delimeters (delimeter)*

WS → delim+ | delimeters (delimeter)*
↓
(Wordspace)

lexemes	Token Name	Attribute value
if	if	–
then	then	–
else	else	–
Any id	id	pointer to table entry
Numbes	Numbes	" " "
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

→ Tokens are recognized with the transition diagram

Transition diagram:-

→ The intermediate step in the construction of lexical analyzer, is converting of patterns into flowcharts called are called Transition diagram.

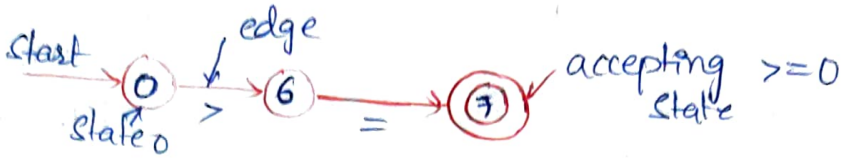
→ Transition diagram have a collection nodes or circles, called "states"

→ The states are connected by arrows, called "Edges"

→ A Double circle indicating an accepting state (a) a final state in which a token is found.

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

→ Transition diagram is also called as "Finite automata".
→ Start (or) Initial state is indicated by edge labled as "start".
→ Transition diagram always begin in the start state before any i/p symbols have been read.



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

retract (comeback by one character)

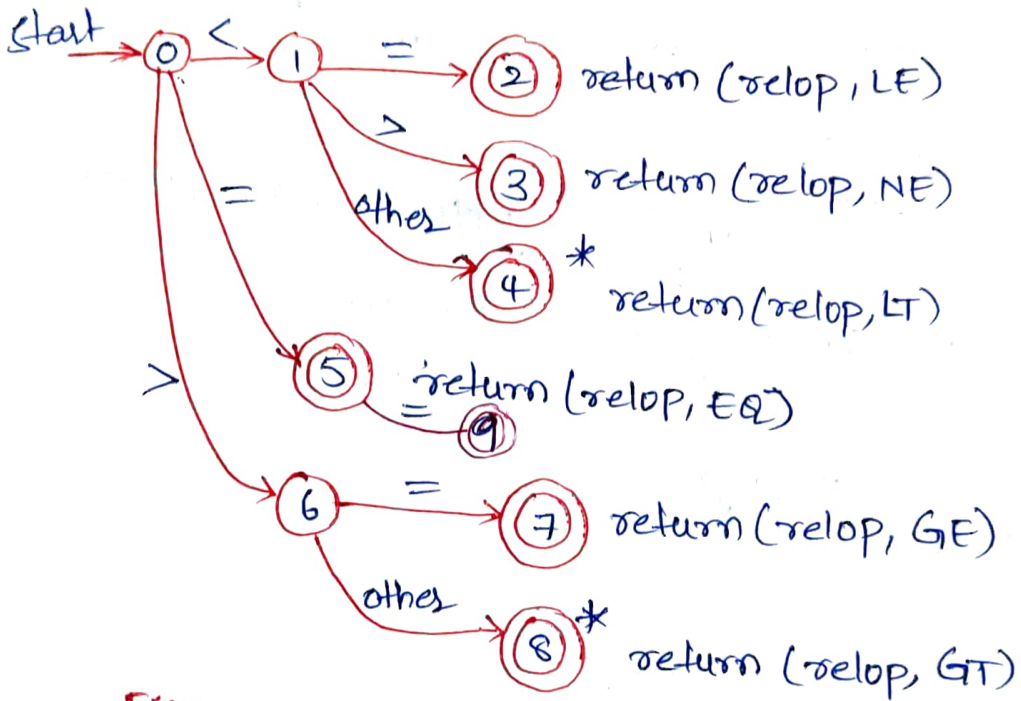


Fig: Transition diagram for relop

Recognition of delimeters:-

RE → ws = delimeter (delimeter)*

Fig1: Transition diagram for delimeters

Recognition of keywords:-

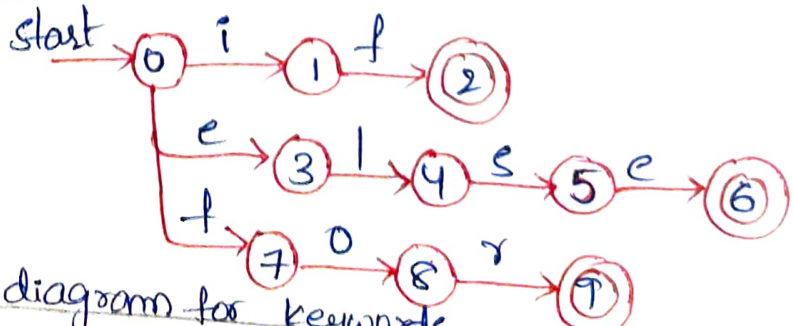
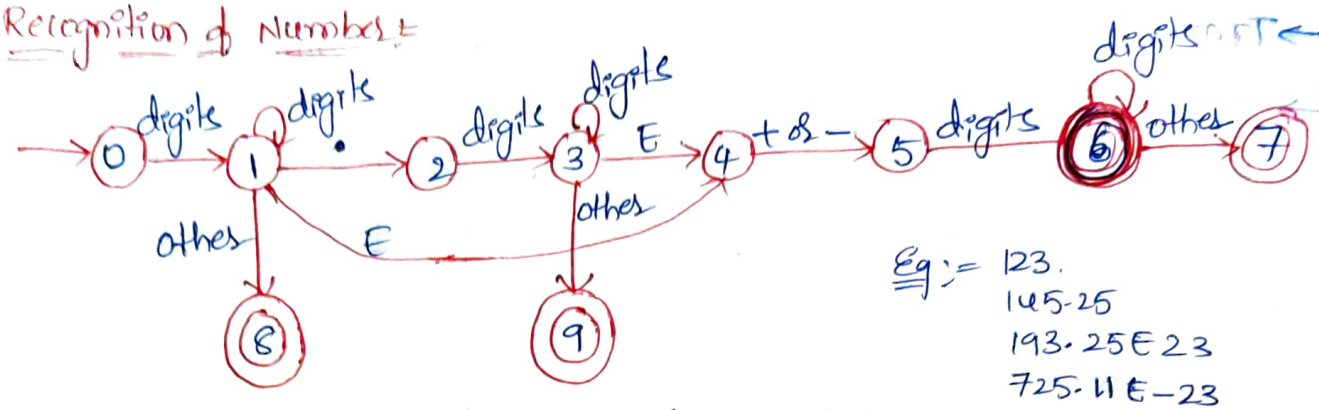


Fig: Transition diagram for keywords

studocu
This document is available on
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Recognition of Numbers:-



Eg:- 123.
145-25
193.25E23
725.11E-23

Fig: Transition diagram for Numbers

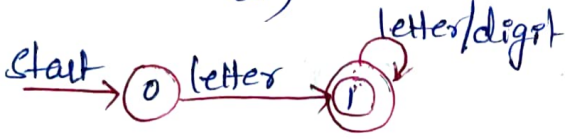
[? → Represents 0 & more one occurance]

RE: digits+(.digits+)?(E[+ -]?digits+)?

Recognition of Identifiers:- letter(letter/digit)*

Fig: Transition diagram for identifiers

(d)



Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

LEXICAL ANALYZER GENERATOR LEX

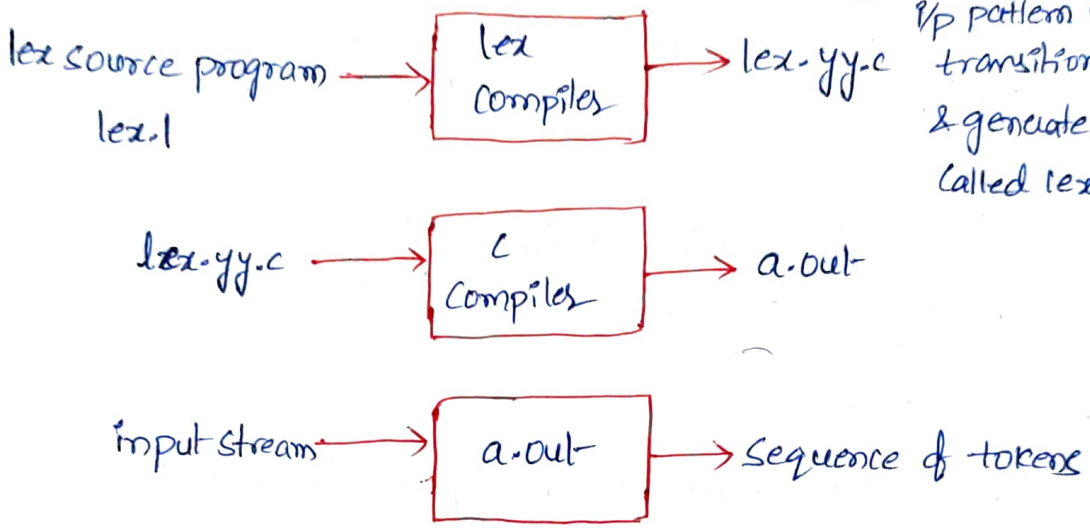
A language for specifying lexical Analyzer (A)

↳ Lex is a tool or a language which is used to generate the the lexical analyzer

→ Lexical analyzer specifies a Regular Expressions and these RE are used to represent the pattern for tokens.

Creating lexical Analyzer with

[lex compiles, transforms i/p pattern into transition diagram & generate a code, called lex.yy.c]



→ lex compiles compiles lex.l file and produces a file lex.yy.c

→ lex.yy.c program is given as i/p C compiles, it compile this program binary and produces a binary (A) object file called a.out

→ Then we need give input to a.out and produces tokens.

Structure of lex program:-

Lex program contains 3-section

(i) Declarations

(ii) Translation Rules

(iii) Auxilliary Functions

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Declarations
%%
Translation rules
%%
Auxilliary functions.

(i) Declarations:-

→ Declaration section is mainly useful inorder to declare C variables and constants.

Syntax:
```
%{
    variables, constants
%}
```

Eg:-
```
%{
    int a, b;
    float count=0;
%}
digit [0-9]
letter [a-zA-Z]
```
} RE

→ Declaration section is also used to define the Regular Expressions

(ii) Translation Rules:-

→ Translation Rules are used to specify the pattern rules.

→ Translation Rules are defined in blw %% & %%

Syntax:-
```
%%
    Translation rules
%%
```

→ We can specify the rule in the form of pattern followed by Actions

Pattern ↓ (RE) {Action} ↓ (is a C-language statement)

Eg:-
```
%%
pattern1   {Action1}
pattern2   {Action2}
   ||
%%
```

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

(iii) Auxiliary functions:- → used to define the function.

→ All the functions that are needed are defined in this section.

Eg:- lex program to recognize identifiers, keywords, Relational operators & Numbers.

lex.l :-

```
%{
    /* definition of lex program for recognize Tokens
       LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUMBERS, RELOP */
%}
/* regular definitions */
letter [a-zA-Z]
digit [0-9]
id {letter} ({letter}/{digit})*
delim [\t\n]
ws {delim}+
number {digit}+(.{digit}+)?(E[+-]?{digit}+)?
%%
{id}
{ws} { /* no action and no return */ }
if {return (IF);}
then {return (THEN);}
else {return (ELSE);}
{id} {yylval = (int) installID(); return (ID);}
{number} {yylval = (int) installNum(); return NUMBER);}
"<" {yylval = LT; return (RELOP);}
```

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

```
"<="    {yylval = LE; return(RELOP);}
"="     {yylval = EQ; return(RELOP);}
"<>"    {yylval = NE; return(RELOP);}
">"     {yylval = GT; return(RELOP);}
">="    {yylval = GE; return(RELOP);}
%%
int installID() {/* function to install the lexeme, whose first character
                    is pointed to by yytext, and whose length is yyleng,
                    into the symbol table and return a pointer thereto */
int installNum() {/* similar to InstallID, but puts numerical constant
                     into a seperate table */
```

Finite Automata:-

→ Finite Automata automata

FINITE AUTOMATA:-

→ Finite automata are recognizers; They simply say "yes" or "NO" about each possible string.

→ For the given language we construct a finite automata, FA is used to check that for the given string is present in the language (or) Not.

→ Finite automata is an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, state and set of transitions from state to state that occurs on i/p symbols from alphabet Σ

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

There are 2-types of Finite Automata.

① NFA (Non-Deterministic Finite automata)

A NFA consists of 5-tuples NFA = $(Q, \Sigma, \delta, q_0, F)$

Q (or) $S \Rightarrow$ A Finite set of states

Σ = A set of i/p symbols (ε is never a member of Σ)

q_0 = start (or) initial state

F = set of Final states (Accepting state)

δ = Transition Function = $Q \times \Sigma = 2^Q$

→ NFA can have null states (ε-transitions)

→ In NFA on single i/p it is going to multiple states or null states.

→ We can represent NFA or DFA by transition graph.

This graph is very much like to transition diagram Except.

(a) A same symbol can label edges from one state to several different states, and

(b) An edge may be labeled by ε, the Empty string

Eg: = Construct a NFA for RE $(a|b)^*abb$

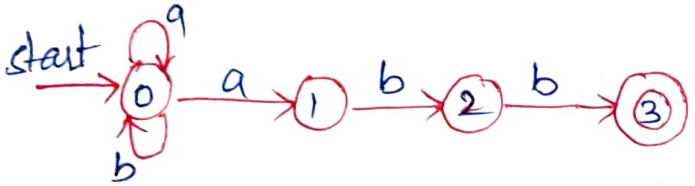


Fig: NFA for string $(a|b)^*abb$

→ It is easy to construct NFA than DFA for a given language

→ Every NFA is not DFA, but each NFA can be translated into DFA

→ In NFA, there exist many paths for specific i/p from the current state to the next state

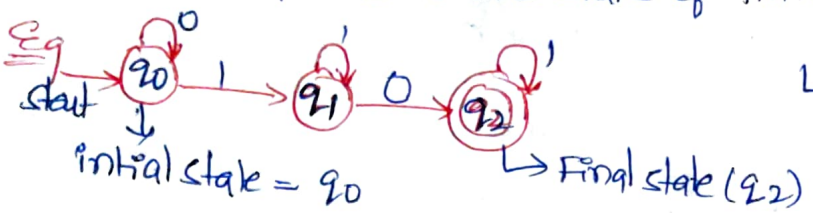
This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Transition Graph:= (a) Transition Diagram

It is a directed graph associated with vertices of the graph corresponds to the state of finite automata.

Eg



↳ The states present in btw inital & Final state are called intermediate state

↳ inpute $\Sigma = \{0, 1\}$

Transition Table:=

→ NFA is also represented by Transition table, whose rows corresponds to state, columns corresponds to i/p symbols.

State	i/p a	b	ε
0	{0,1}	{0}	ϕ
1	ϕ	{2}	ϕ
2	ϕ	{3}	ϕ
3	ϕ	{ϕ}	ϕ

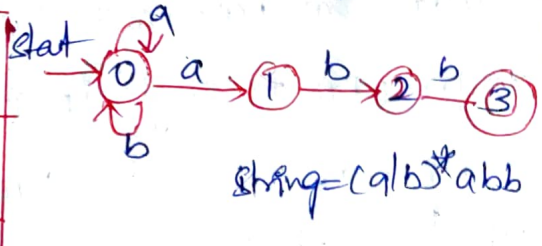


Fig: Transition Table.

Transition Function:=

→ The mapping (δ) Transition function is denoted by δ

→ two parameters are passed to transition function.

(i) current state & (ii) i/p symbol

→ Transition function (q0) returns a state i.e next state. (a)

Eg := $\delta(q_0, a) = q_1$ (a) $(q_0, 1) = q_1$

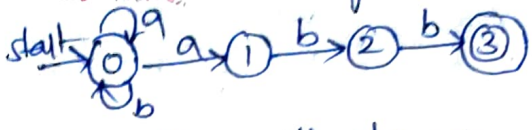
↓ current state ↓ i/p symbol ↓ next state.

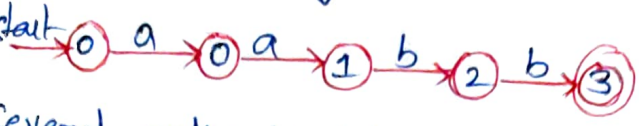
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

• Acceptance of Input string by Automata:-

The path labeled by aabb

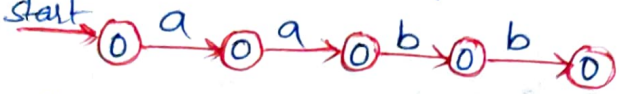
Fig:- NFA for $(a|b)^*abb$



(i) 

(In this path string is accepted)

Several paths labeled by the same string may lead to the different states

(ii) 

(in this path string is not accepted)

but always choose the path which is starts from initial state to final state, so the first path is considered.

DFA: (Deterministic FA):-

→ DFA is a special case of FA where;

(i) There are no moves on input ε, and

(ii) For each states and input symbol a, there is exactly one edge out of S labeled a.

DFA = $(Q, \Sigma, \delta, q_0, F)$

$\delta = Q \times \Sigma = Q$

Q = A set of Finite state

Σ = a set of i/p symbols

q_0 = initial (or) start state

F = Final state

δ = Transition function

Construction of DFA:-

Eg#:- String ending with a substring.

Step1:- Decide the min no. of states required to construct DFA

Substring length = n no of states = n+1

Step2: Decide the string for which you will construct DFA.

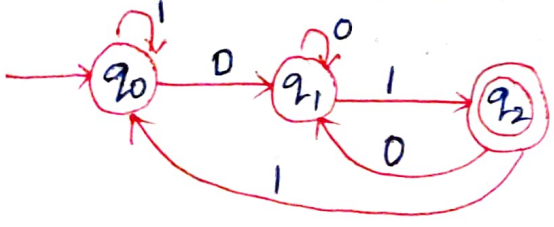
Step3: construct the DFA for above decided string (always prefer to go with existing path)

Step4: After construction of DFA, send the left over possible combinations to the the starting state.

Eg1:- String ending with "01" over $\Sigma = \{0, 1\}$

Regular Expression = $(0|1)^*01$

min no of = 3 substring → 01, 001, 0101



[NOTE:- Always prefer to go with the existing path, create a new path only when can't find a path to go with]

This document is available on studocu
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

(ii) String ending with {abb} over $\Sigma = \{a, b\}$

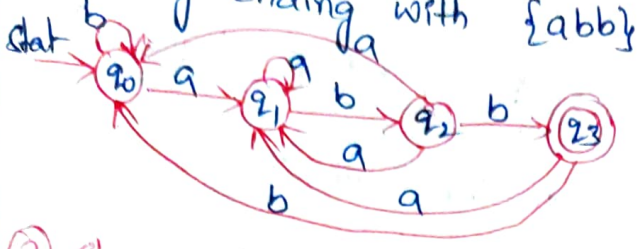


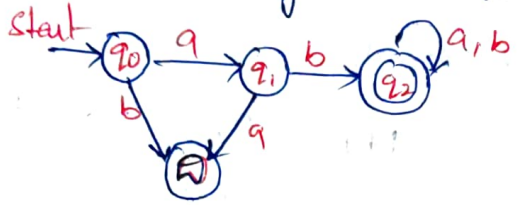
abb
aabb
ababb
abbabb

(2) String starting with substring.

→ No of states = n+2

→ In this left over possible combinations are connected to dead state

Eg 1: String starting with "ab" over $\Sigma = \{a, b\}$

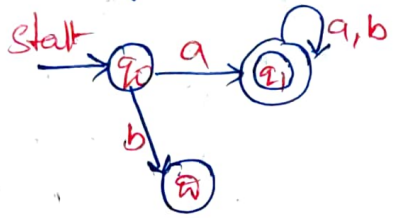


RE = ab(a|b)*

substring ab
aba
abab

no. of states = 2+2
= 4

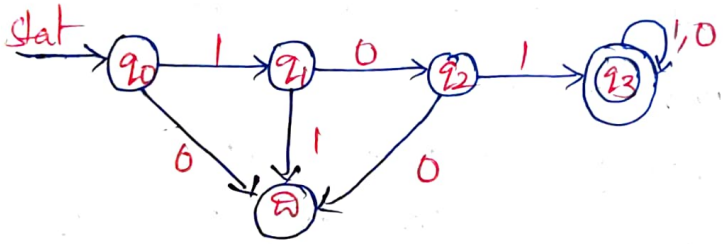
Eg 2: String start with "a" over $\Sigma = \{a, b\}$ RE = a(a|b)*



substring a
a
bb

no. of states = length of substring + 2
= 1+2
= 3

Eg 3:- String start with "101" over i/p $\Sigma = \{0, 1\}$, RE = 101(0|1)*

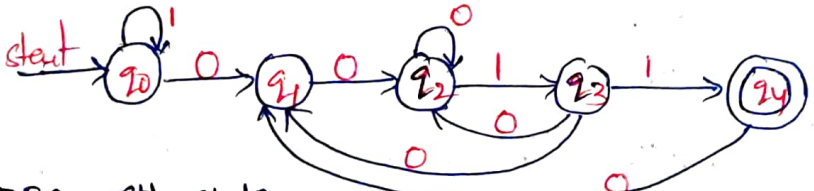


substring = 101
1011
10110
10101

no. of states = 3+2
= 5

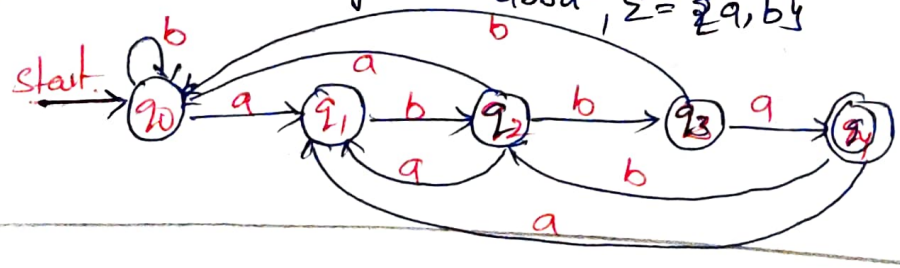
DFA Eg:- Construct an DFA that should start with "0011" over i/p $\Sigma = \{0, 1\}$

RE = 0011(0|1)*



0011
00011
000011
001011
0011011

Eg :- DFA with starting with "abba", $\Sigma = \{a, b\}$



Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

From Regular Expression to Automata:=

Conversion of an NFA to DFA:=

Eg:= (1) subset construction method:=

Eg:= The string end with "ab" over I/P alphabet $\Sigma = (a, b)$

Fig: NFA

Step1:= We have to write all the subset of states

$Q = \{q_0, q_1, q_2\}$

subset = $\{\phi, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_1, q_2, q_3\}\}$

If the given NFA contain n elements, the no. of possible subsets are $= 2^n$

Here $n = 3$, subsets $= 2^3 = 8$

Step2:= transition table for NFA.

	Input	
States	a	b
ϕ	ϕ	ϕ
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
q_1	ϕ	q_2
* q_2	ϕ	ϕ
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
* $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
* $\{q_1, q_2\}$	ϕ	$\{q_2\}$
* $\{q_0, q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

This document is available on studocu

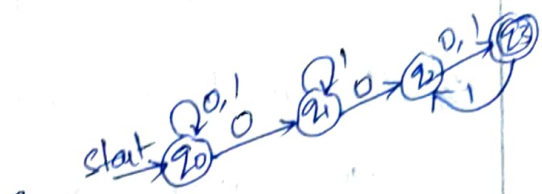
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

→ Now, we are going to eliminate the unwanted states.

Step3:= construction of DFA transition table

states	input a	input b
→ q_0	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
* $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Eg2:=



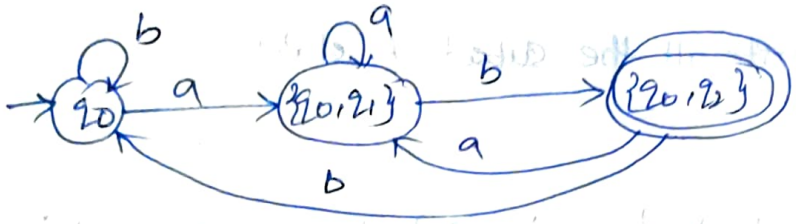


Fig: DFA

(2) ε-NFA to DFA:=

RE:= $(a+b)^* a$ $\Sigma = \{a, b\}$

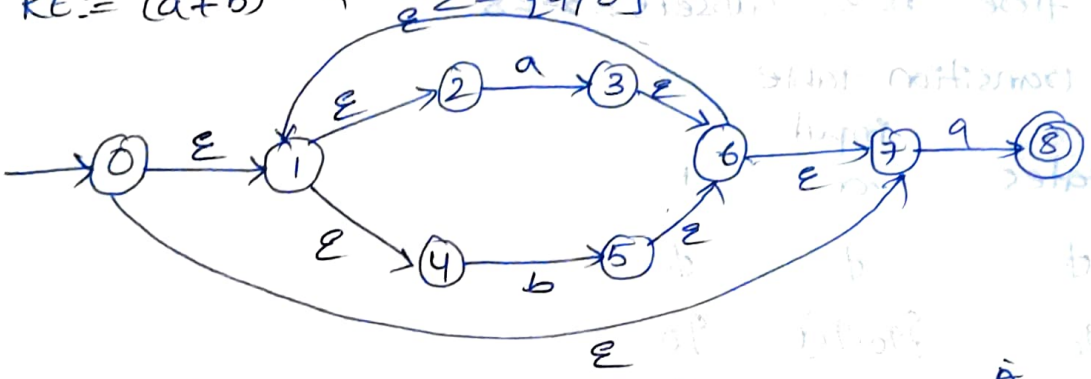


Fig:= NFA for $(a+b)^* a$ ε-closure = All ε-path of the given state is called as ε-closure)

A = ε-closure(0) = {0, 1, 2, 4, 7}

$\delta(A, a)$ = ε-closure($\delta(0,1,2,4,7), a$)

= ε-closure($\delta(0,a) \cup \delta(1,a) \cup \delta(2,a) \cup \delta(4,a) \cup \delta(7,a)$)

= ε-closure(3, 8) = {3, 6, 1, 7, 2, 4, 8} = B

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

$\delta(A,b)$ = ε-closure $(\delta(0,1,2,4,7),b)$

= ε-closure $(\delta(0,b) \cup \delta(1,b) \cup \delta(2,b) \cup \delta(4,b) \cup \delta(7,b))$

= ε-closure (5)

= ε-closure $\{5,6,7,1,2,4\} = C$

$\delta(B,a)$ = ε-closure $(\delta(3,6,1,7,2,4,8),a))$

= ε-closure $(\delta(3,a) \cup \delta(6,a) \cup \delta(1,a) \cup \delta(7,a) \cup \delta(2,a) \cup \delta(4,a) \cup \delta(8,a))$

= ε-closure $(8,3) = B$ ✓ Final state

$\delta(B,b)$ = ε-closure $(\delta(3,6,1,7,2,4,8),b))$

= ε-closure $(\delta(3,b) \cup \delta(6,b) \cup \delta(1,b) \cup \delta(7,b) \cup \delta(2,b) \cup \delta(4,b) \cup \delta(8,b))$

= ε-closure $(5) = (5,6,1,2,4,7) = C$

$\delta(C,a)$ = ε-closure $(\delta(5,6,7,1,2,4),a)$

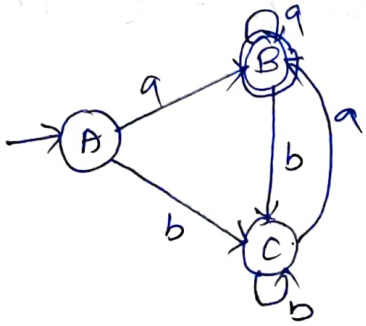
= ε-closure $(\delta(5,a) \cup \delta(6,a) \cup \delta(7,a) \cup \delta(1,a) \cup \delta(2,a) \cup \delta(4,a))$

= ε-closure $(8,3) = B$

$\delta(C,b)$ = ε-closure $(\delta(5,6,7,2,4),b))$

= ε-closure $(\delta(5,b) \cup \delta(6,b) \cup \delta(7,b) \cup \delta(2,b) \cup \delta(4,b))$

= ε-closure $(5) = C$



DFA Transition Table

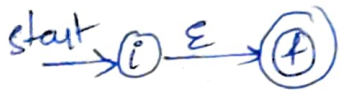
	a	b
A	B	C
B	B	C
C	B	C

Fig:- DFA

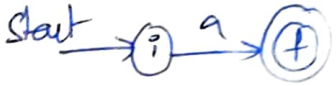
This document is available on studocu
Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Construction of an NFA from Regular Expression :-

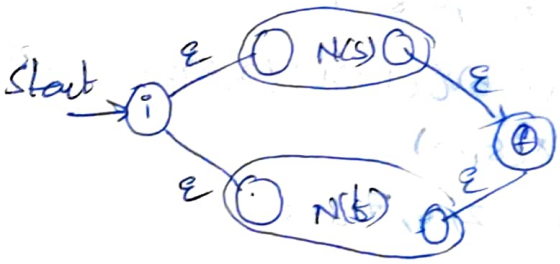
(1) for expression ε construct NFA (Thompson construction method)



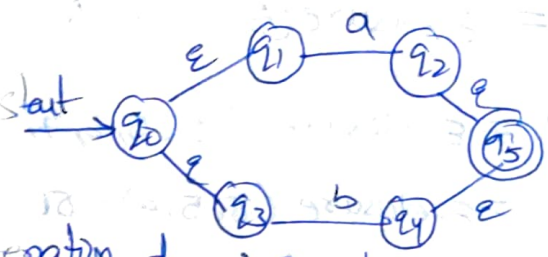
(2) for any subexpression a in Σ, construct NFA



(3) NFA for the union of 2 regular expression

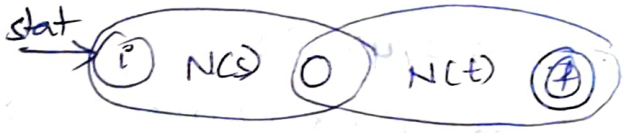


Eg :- (a+b) = (a|b)

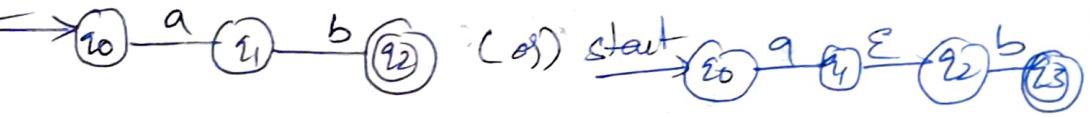


(4) NFA for the concatenation of 2 Regular expression

"ab"

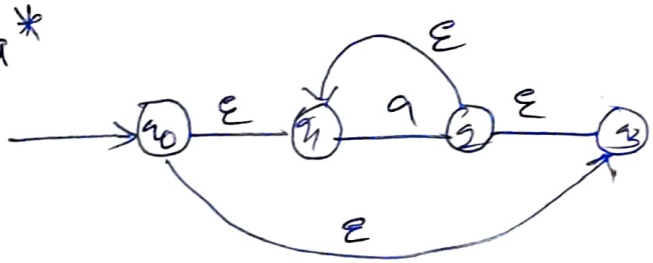


Eg :-



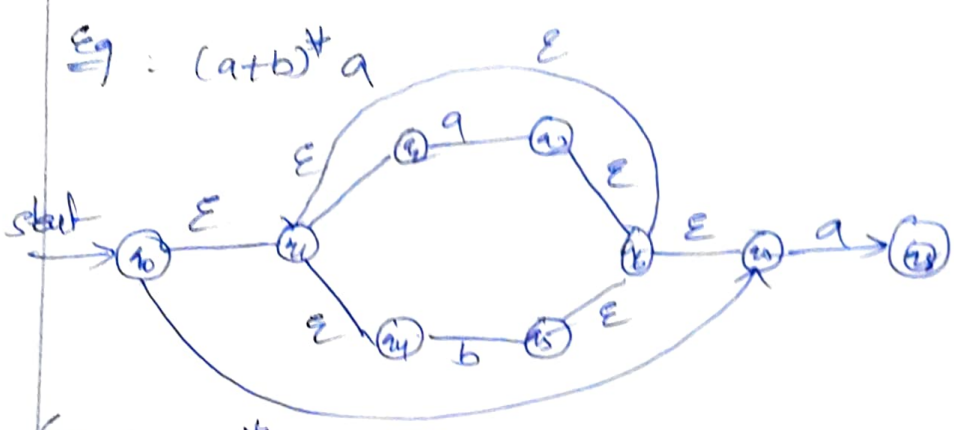
(5) NFA for the closure of RE

a^*

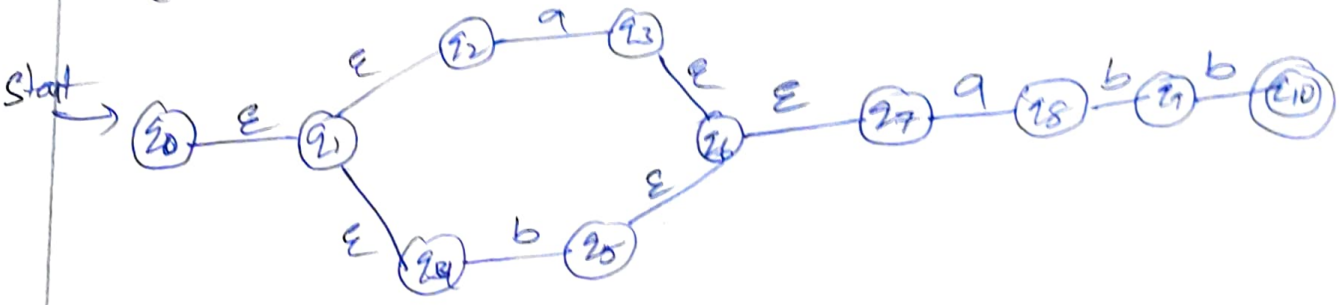


Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

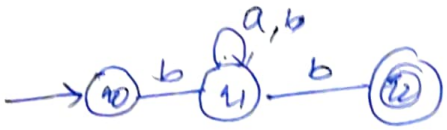
Eg : $(a+b)^* a$



Eg $(a+b)^* abb$



Eg :- (1) ba^*b

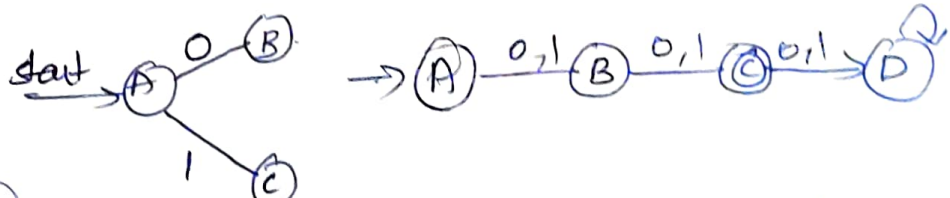


Examples for construction of DFA :-

(1) L_1 = set of all strings start with 0 Eg : 001

(2) construct the DFA that accepts of all string over {0,1} of length 2

$\Sigma = \{0,1\}$ $L = \{00, 01, 10, 11\}$



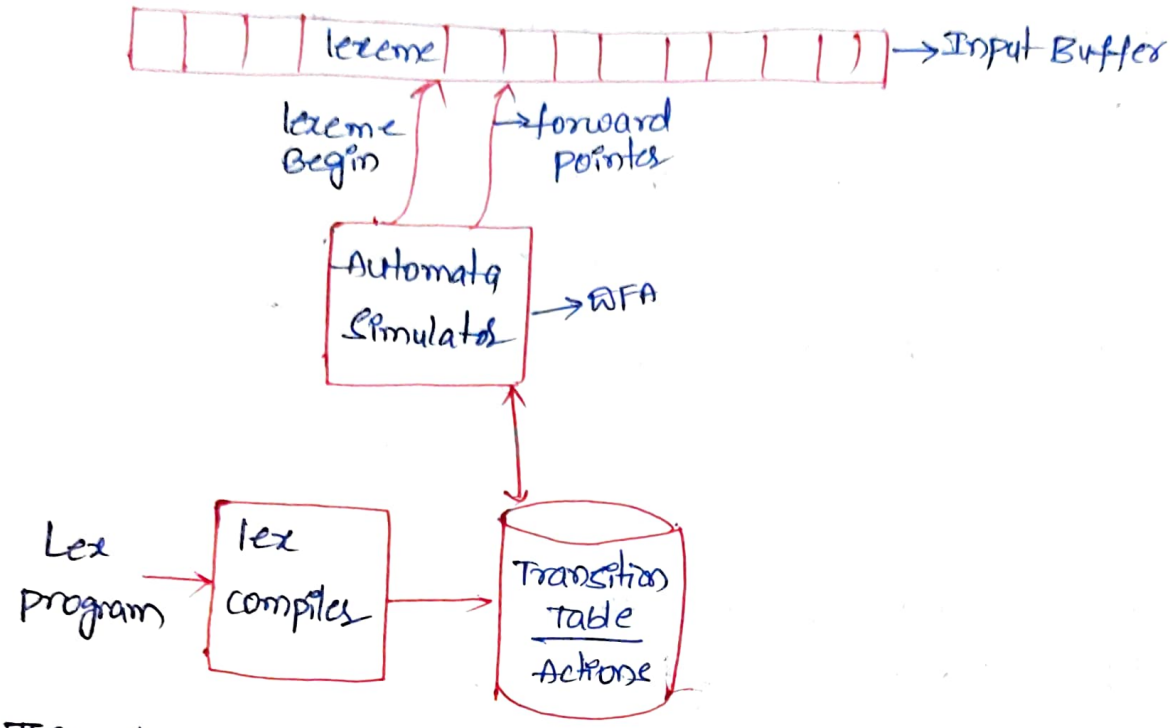
(3) Construct a DFA that accepts any string over {a,b} that doesn't contain the string aabb in it $\Sigma = \{a, b\}$

1st const DFA that accepts aabb

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

The Structure of the Generated Analyzer:-



→ This structure describes how an Finite automata simulator is working for the lexical Analyzer

→ lex program simulates an automata.

→ lex program consists of all RE definition and Translation Rules.

→ Once a lex program is compiled the Regular Expressions are converted into the Finite automata.

→ The translation rules are turned into transition table and actions which are used by the a Finite automata simulator. to recognize the tokens.

→ Finite automata simulator checks any i/p strings that are stored i/p buffer, is matching with any pattern, if it matches then it will produce corresponding Token as an o/p.

Automata simulator uses the following components

(i) Transition Table for automata.

(ii) Function are passed directly through lex to o/p.

(iii) The action from i/p, which appear as fragments of code to be invoked at the appropriate time by automata simulator

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

To construct an automata:=

1. Convert the Regular Expression patterns in the lex program to the NFA.
2. We need a single automata that will recognize lexemes matching any of the patterns in the program.
3. We combine all the NFA into one by introducing a new start with ϵ-transition.

Eg:= a { action A_1 for pattern P_1 }

abb { action A_2 for pattern P_2 }

a^*b^+ { action A_3 for pattern P_3 }

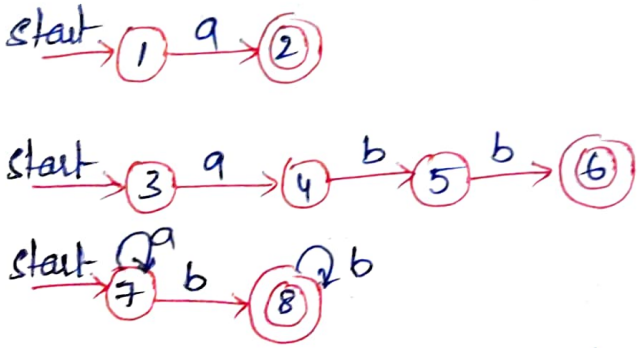
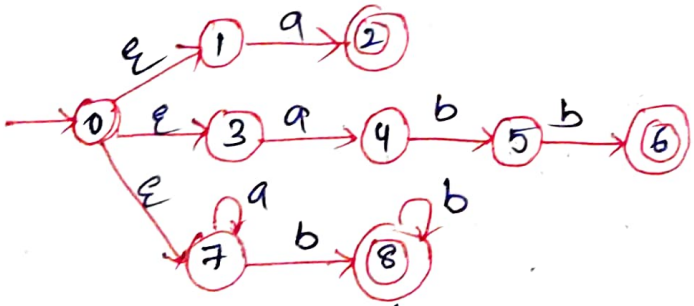


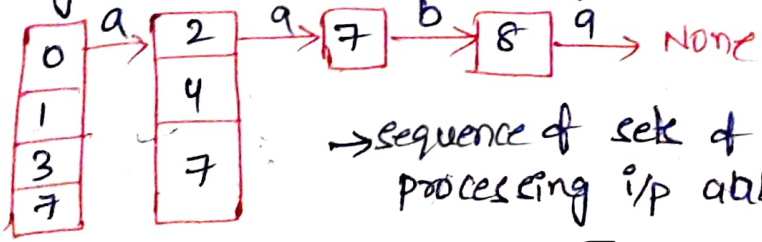
Fig: NFA's for a, abb, and a^*b^+



→ converted this NFA to DFA

Fig:= Combined NFA

I/p string abbq pattern matching Based on NFA:=

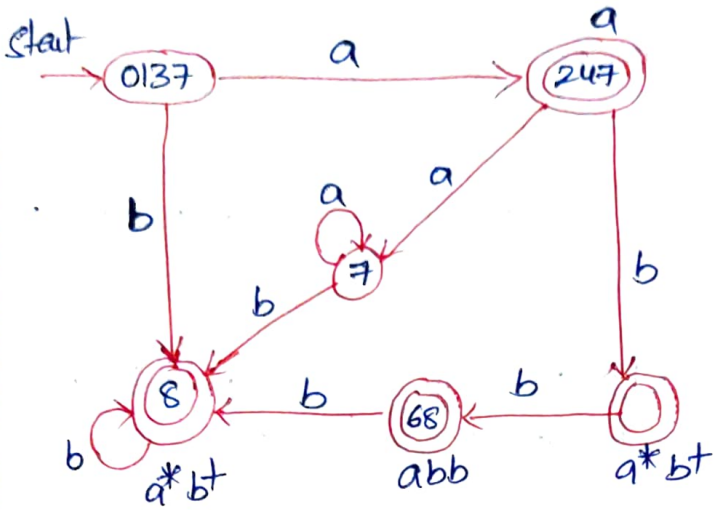


→ Sequence of sets of states entered when processing i/p abbq

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

DFA for the above NFA



Eg:- Transition graph for DFA handling the patterns a, abb, & a*b+

OPTIMIZATION OF DFA-BASED PATTERN MATCHERS:-

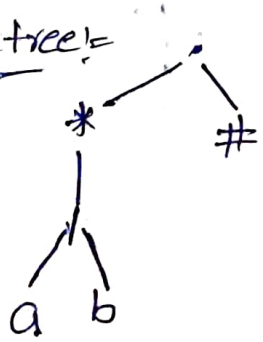
- Convert the Regular Expression into DFA without creating NFA.

* Introduce Augumented RE.
* Construct a syntax tree for RE.
* Number the leaf Nodes.
* Traverse the syntax Tree to construct functions Nullable,
* Firstpos and lastpos for each node.
* Compute Followpos of each node.
* Converting to DFA.

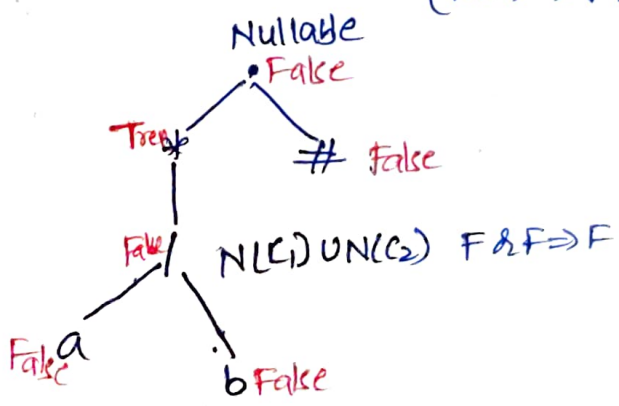
Eg:- $(a|b)^*$

Step1:- (Augumented grammar) $(a|b)^* \cdot \#$

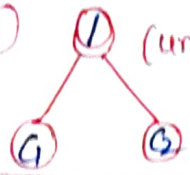
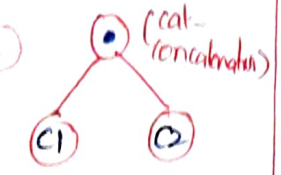
Step2:-
Syntax tree:-



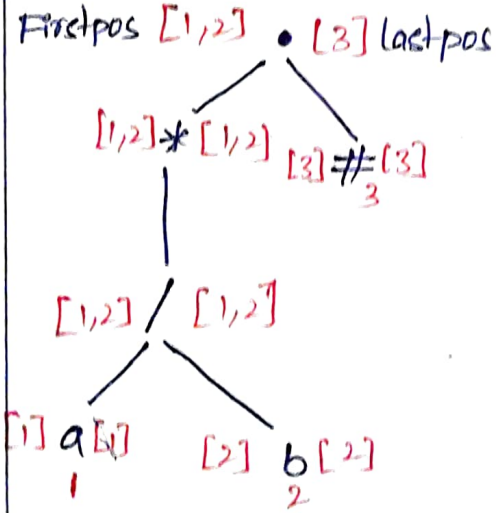
(AND → T AND F ⇒ F)



Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

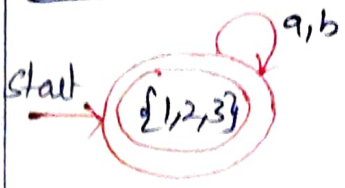
Node	Nullable	Firstpos(n)	Lastpos(n)
(i) N → leaf ε	True	ϕ	ϕ
(ii) N → leaf (i)	False	{i}	{i}
(iii)  N (star-Node)	True	Firstpos(c1)	Lastpos(c1)
(4)  (union)	Nullable(c1) or Nullable(c2)	Firstpos(c1) ∪ Firstpos(c2)	Lastpos(c1) ∪ Lastpos(c2)
(5)  (cat-concatenation)	Nullable(c1) and Nullable(c2)	if (nullable(c1)) is True Firstpos(c1) ∪ Firstpos(c2) else Firstpos(c1)	if (Nullable(c2)) is True Lastpos(c1) ∪ Lastpos(c2) else Lastpos(c2)

Firstpos and lastpost computations:=



Node	followpos
1 → a	{1,2,3}
2 → b	{1,2,3}
3 → #	

DFA:=



DFA = (Q, Σ, δ, q0, F)

q0 → (1,2,3) Σ = (a,b) F → 1,2,3

δ → (([{1,2,3}, a)), ([{1,2,3}, b))

({1,2,3}, {1,2,3})

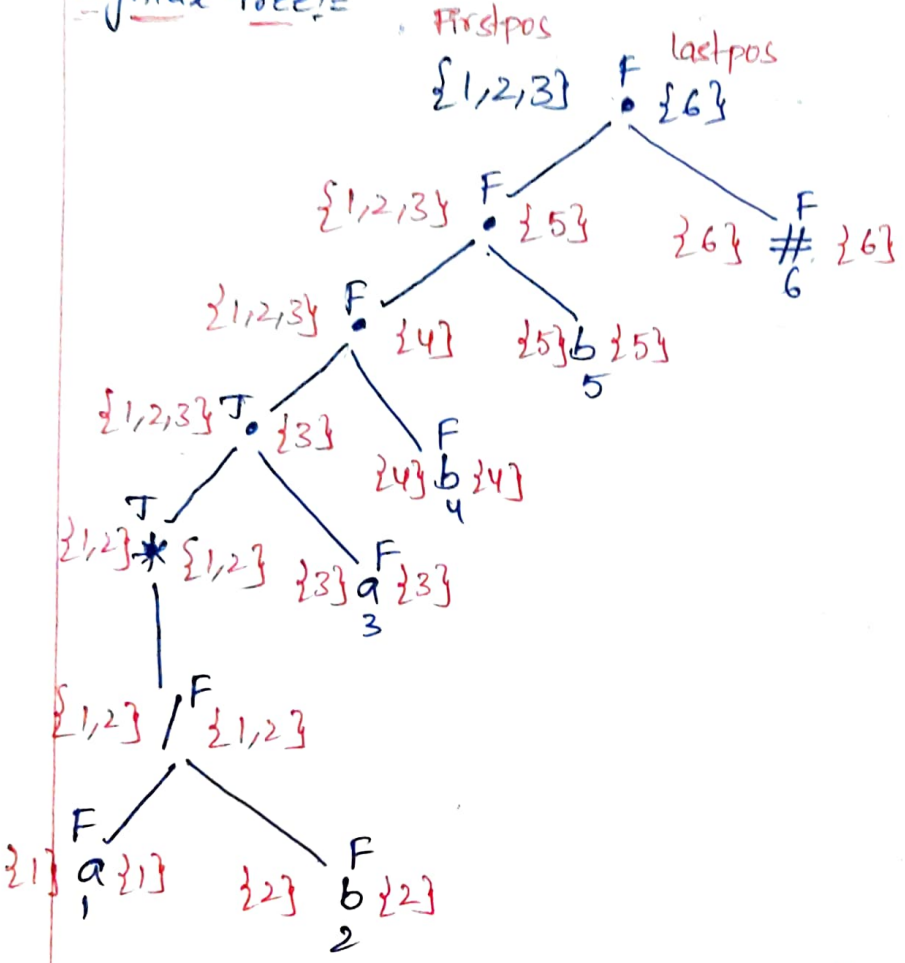
This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

Eg2:- RE → $(a|b)^*.abb$

Augumented Grammar $(a|b)^*.a.b.b\#$

Syntax Tree:-



→ For Finding Followpos star (*) and cancatenation (.) nodes will be considered.

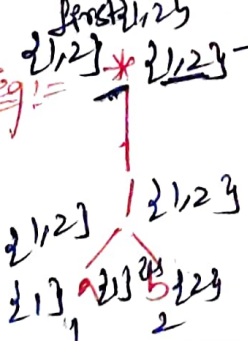
Star Node (*) → if (n is star-node)

for (each i in lastpos(n))

followpos(i) = followpos(i) ∪ first(n);

→ If n - is a star node and i="i" is a position in lastpos(n), then all position in firstpos(n) are in followpos(i)

Eg:-



Node	followpos()
1	{1,2}
2	{1,2}

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

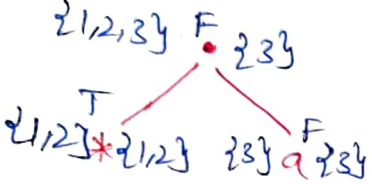
(ii) cat Node (·):- if (n==$c_1 \cdot c_2$ (concat_node))

for (each i in lastpos(c_1))

followpos(i) = followpos(i) ∪ firstpos(c_2)

→ If n is cat-node with left child c_1 & Right child c_2, then for every position "i" in lastpos(c_1), all position in firstpos(c_2) are in followpos(i)

followpos(i).



Node	followpos
c_1 lastpos {1	{1,2,3}
2	{1,2,3}

Node	followpos
1 → a	1,2,3
2 → b	1,2,3
3 → a	4
4 → b	5
5 → b	6
6 → #	-

Initial state → 1st Root Node first position {1,2,3} ⇒ A

$\Sigma = (a,b)$

1) $\delta(A,a) \rightarrow \delta(\{1,2,3\},a) \Rightarrow \{1,3\}$ = Followpos(1) ∪ Followpos(3) = {1,2,3,4} (B)

2) $\delta(A,b) \rightarrow \delta(\{1,2,3\},b) = \{2\}$ = Followpos(2) = {1,2,3} ⇒ A

3) $\delta(B,a) \rightarrow \delta(\{1,2,3,4\},a) \Rightarrow \{1,3\}$ = Followpos(1) ∪ Followpos(3) = {1,2,3,4} ⇒ B

4) $\delta(B,b) = \delta(\{1,2,3,4\},b) \Rightarrow \{2,4\}$ = Followpos(2) ∪ Follows(4) = {1,2,3,5} ⇒ C

5) $\delta(C,a) = \delta(\{1,2,3,5\},a) = \{1,3\}$ = Followpos(1) ∪ Followpos(3) = {1,2,3,4} ≈ B

6) $\delta(C,b) = \delta(\{1,2,3,5\},b) = \{2,5\}$ = Followpos(2) ∪ Followpos(5) = {1,2,3,6} = D

7) $\delta(D,a) = \delta(\{1,2,3,6\},a) = \{1,3\}$ = Followpos(1) ∪ Followpos(3) = {1,2,3,4} = B

8) $\delta(D,b) = \delta(\{1,2,3,6\},b) = \{2\}$ = Followpos(2) = {1,2,3} = A

This document is available on studocu

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)

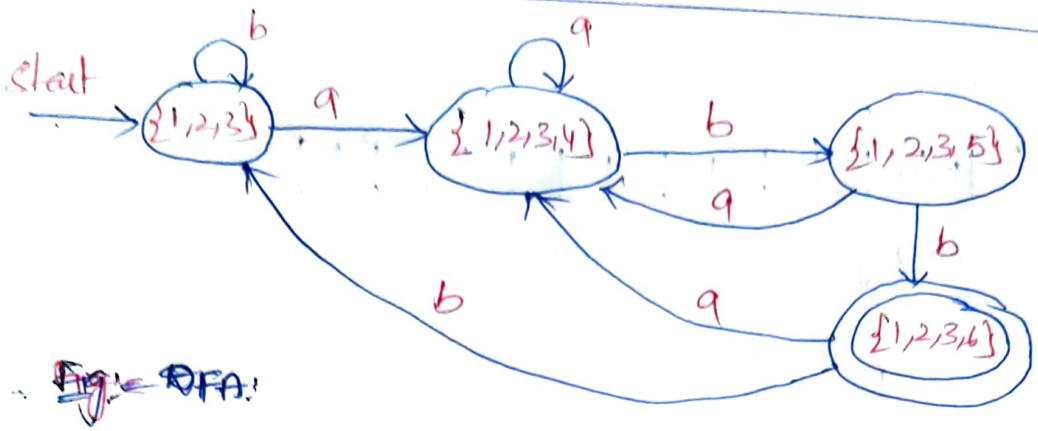


Fig:- DFA.

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

Fig:- DFA Transition Table

Downloaded by Molkapuri Himanshu (molkapurihimanshu@gmail.com)