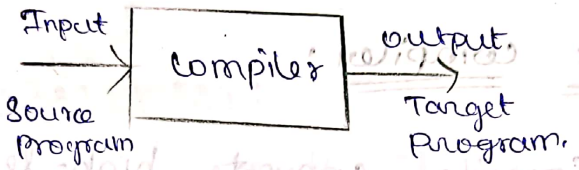
UNIT-I Introduction to compilers & Lexical Analysis

Definition:

⇒ Compiler is a program which takes one language (source program) as input and Translates it into an equivalent another language (Target program).



Translator:

⇒ Translator translates high level language to ~~to level~~ low level language. There is two types of Translators.

1. Compiler
2. Assembler

⇒ Compiler converts high level language to machine level language.

⇒ Assembler converts Assembly language program to machine level language.

compilation and Interpretation:

⇒ The compilers and the Interpreters take heiman readable code and convert into computer readable machine code.

⇒ The compilation of language, the target machine directly translates the program.

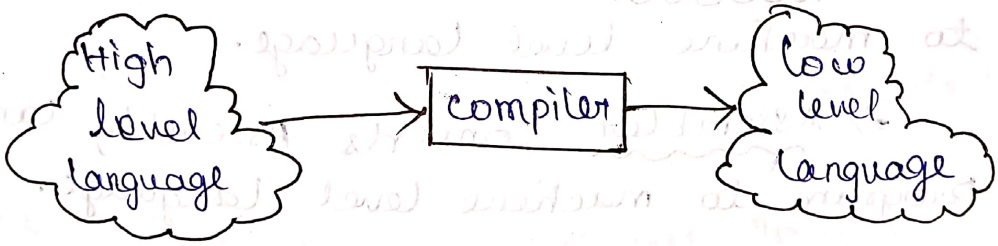
⇒ In an interpreter language the source code is not directly translated by the target machine.

Role of compiler:

1. Compiler converts high-level language into machine level language that can be easily understood by computer.

   (i) compiler converts high-level language to intermediate assembly language.

   (ii) And then assembled into machine code by an assembles.



Advantages of compiler:

* Compiled code runs faster in comparison to interpreted code.

* compilers helps in improving the security of Applications.

* As compilers give debuging tools which help in fixing errors easily.

Disadvantages of Compiler:

* the compiler can catch only syntax errors and some semantic errors

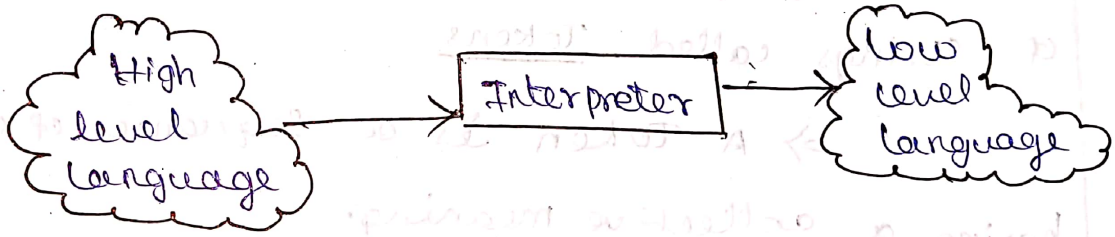
* compilation can take more time in the case of bulky code.

Role of Interpreter:

⇒ The simple role of an interpreter is to translate the material into a target language.

⇒ An interpreter works line by line on a code.

⇒ It also converts high-level language to Machine language.



Advantages of Interpreter:

* Programs written in an interpreted language are easier to debug.

* Interpreters allow the management of memory automatically which reduces memory error risks.

* Interpreted language is more flexible than a compiled language.

Disadvantages of Interpreter:

* The interpreter can run only the corresponding interpreted program.

* Interpreted code runs slower in comparison to compiled code.

Phase of compiler:

Lexical Analyzer:

⇒ It is also called as Scanning.

⇒ In these phase complete source code and the program code is broken up into group of strings called Tokens

⇒ A Token is a sequence of character having a collective meaning.

Example:

total = count + rate * 10

total → Identifier

= → Assignment Symbol

count → Identifier

\+ → Plus Sign

rate → Identifier

* → multiplication sign

10 → constant number

Note:

Blank characters used in the programming Statements are eliminated during lexical analysis phase.

Syntax Analyzer:

⇒ It is also called parsing.

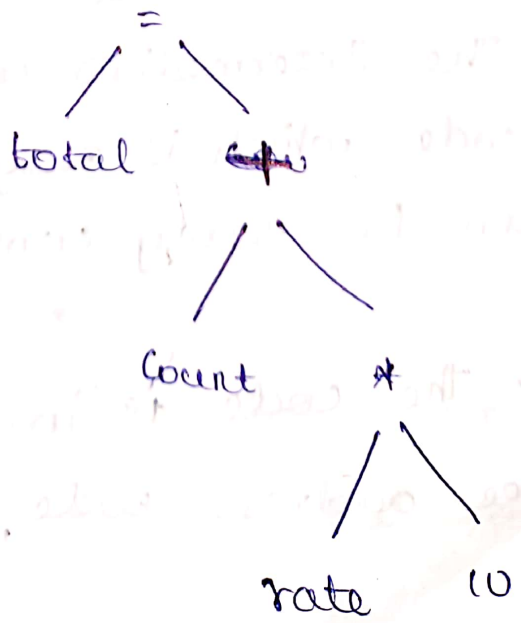
⇒ In this phase tokens generated by the lexical analyzer are grouped together to form a hierarical structure

⇒ The Syntax analyzer determines the structure of the source string.

⇒ The hierarcial structure generated in this phase called parse tree or Syntax t

Example:

total = count + rate * 10



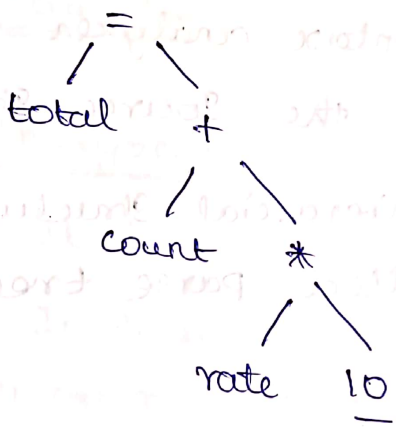
Semantic Analyzer:

⇒ The Semantic Analyzer determines the meaning of the source string. (Hierarchical structure).

⇒ It means checking the scope of operation. (mismatch kind of error)

Example:

total = count + rate * 10



(5)

Intermediate code generation:

⇒ The Intermediate code generator is a kind of code which is easy to generate and this code can be easily converted to target code.

⇒ the code is in variety of forms such as Three address code, Quadruple, Triple, Posix

Example:

total = count + rate * 10

t1 := inttofloat (10)

t2 := rate * t1

t3 := count + t2

total := t3.

Code Optimization :-

(Reduce)

⇒ This phase improves the intermediat code.

⇒ It is neccessary to have a faster executing code or less consumption of memory.

⇒ optimizing the code the over all running time of the target program can be improved.

Code Generator :-

⇒ In code generation phase the target code gets generated.

⇒ The intermediate code instructions are translated in to sequence of machine instruction. assembly code or machine

Example:

total = count + rate * 10

```
MOV rate, R1
MUL #10.0, R1
MOV count, R2
ADD R2, R1
MOV R1, total
```

Symbol Table Management:

⇒ To support these phases of compiler a symbol table is maintained.

⇒ The symbol tables stores identifiers (variables).

⇒ The symbol table stores Information about attributes of each identifier.

⇒ The symbol tables also stores information about the subroutines.

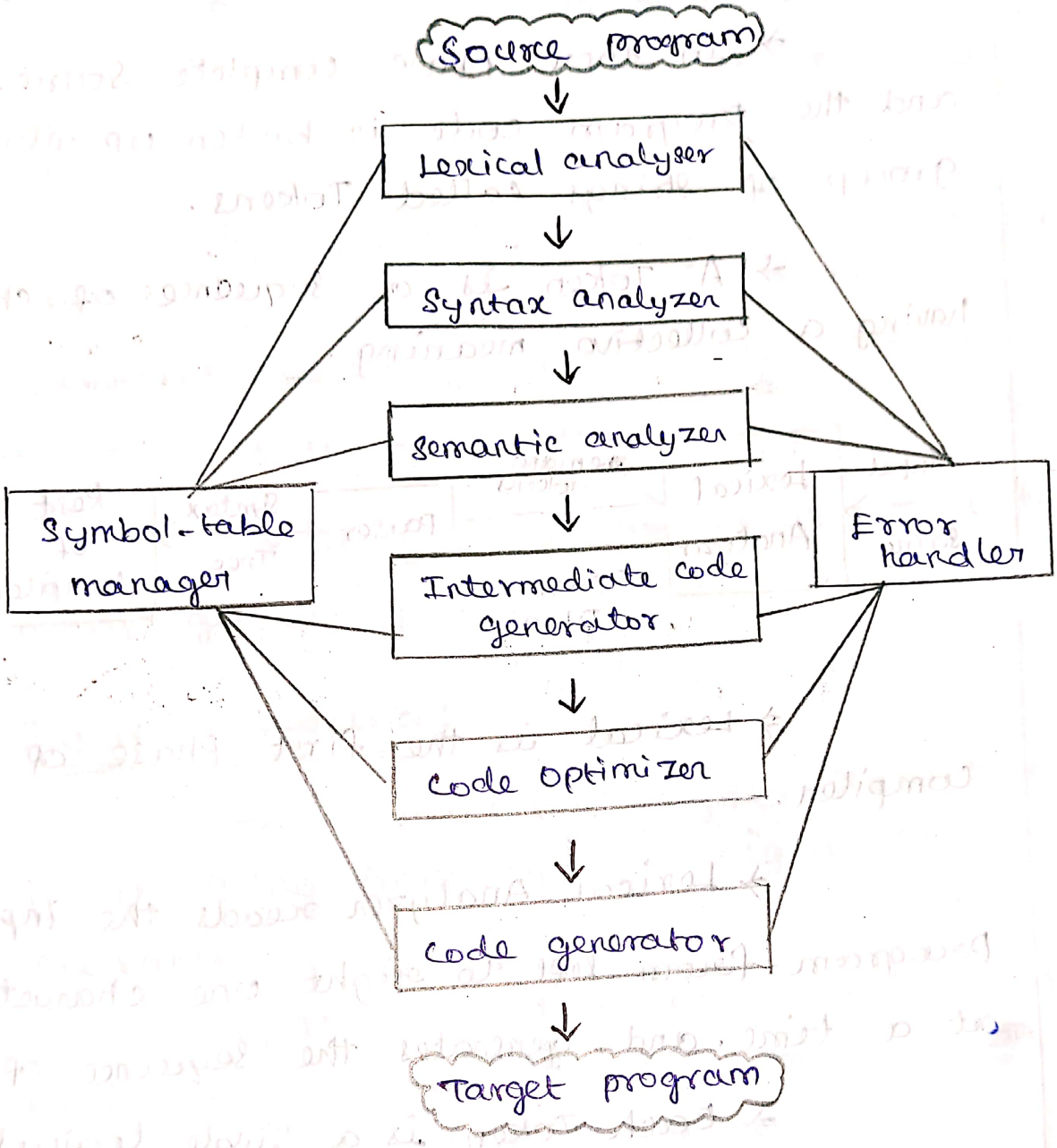
Error detection and handling:-

⇒ Each phase detects error.

⇒ Errors are reported in the form of message.

⇒ Large number of errors can be detected in Syntax phase. This errors are popularly called as syntax errors.

⇒ During semantics phase type mismatch kind of error is usually detected.

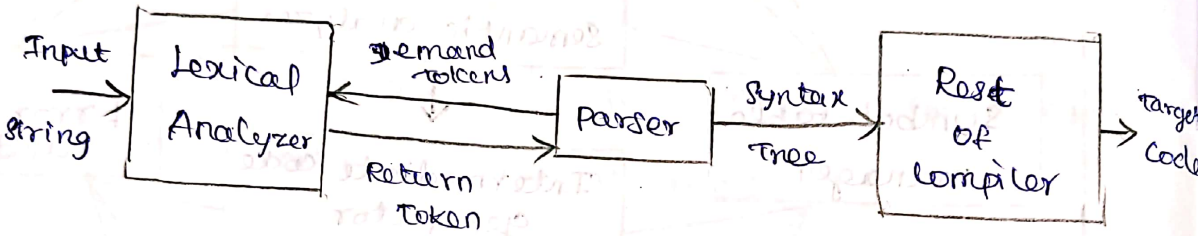


Lexical Analyzer and its Role :-

⇒ It is also called as scanning

⇒ In these phase complete source code and the program code is broken up into group of strings called Tokens.

⇒ A Token is a sequence of characters having a collective meaning.



⇒ Lexical is the first phase of the compiler.

⇒ Lexical Analyzer reads the input program from left to right one character at a time, and generates the sequence of tokens.

⇒ Each token is a single logical cohesive unit such as identifier, key-words, operators and punctuation marks.

⇒ Then the parser to determine the Syntax of the source program can use these Tokens.

⇒ As the lexical Analyzer scans the source program to recognize the tokens it is called as Scanner.

Functions of Lexical Analyzer :-

⇒ It produce stream of tokens.

⇒ It eliminates the blank and commands.

⇒ It Generates symbol table which stores the information about Identifier, constants ~~and~~ encountered in the input.

⇒ It keeps track of line numbers.

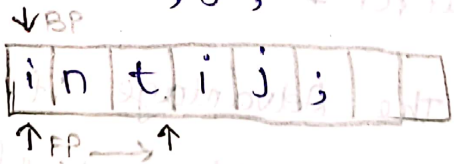
⇒ It reports the error encounted while generating the tokens.

Input Buffering :-

⇒ The Lexical Analyzer scans the input string from left to right one character at a time.

⇒ It uses two pointer such as Begin pointer and forward pointer. to keep track of the portion of the input scanner.

eg: int i, j ;



Key points.

* ~~BP~~ Begin pointer
* FP (Forward p)
* Blank symbol
* EOF

⇒ The both pointers move a head to search for the lexeme (Tokens)

⇒ The forward pointer encounters a blank space or wide space.

Types ⇒ There are two buffer.

1. One Buffer Scheme
2. Two Buffer Scheme.

One Buffer Scheme:

⇒ one buffer is used to store input string.

⇒ The problem in one buffer scheme reads the input at the end of the boundary makes the string over write in the buffer.

⇒ It makes we move to the two buffer scheme.

Two Buffer scheme:

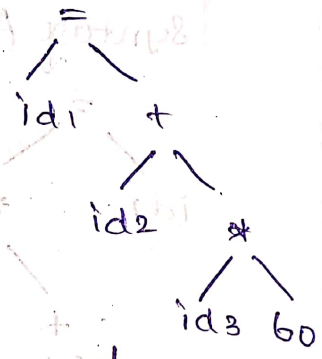
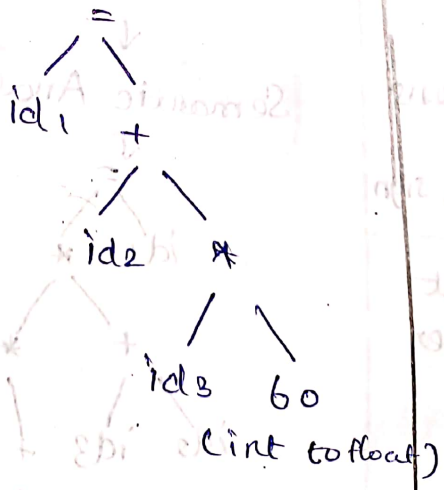
⇒ In the scheme separate the buffers into two parts. (Bf 1, Bf 2)

⇒ The First Buffer is filled completely and the string is moved to second buffer. If the string is very long rewrite ~~the~~ one again in buffer 1.

⇒ the Advantage of the two buffer Scheme it Stores alternatively.

⇒ The eof (end of file) is also called Sentinal.

① Show how an input a=b+c*60 get processed in compiler. Show the output at each stage of compiler. Also show the contents of symbol table.

Storage	Input processing in compiler	Output
a ..... b ..... c ..... d ..... a → Identifier = → equal sign/ assingment opr b → Identifier c → Identifier * → multiple sign + → Plus sign 60 → constant/ Integer	Input code a = b + c * 60 ↓ Lexical Analyzer ↓ id1 = id2 + id3 * 60 ↓ Syntax Analyzer ↓  ↓ Semantic Analyzer ↓  ↓ Intermediate code	Input code Token generation Parse Tree Semantic Tree

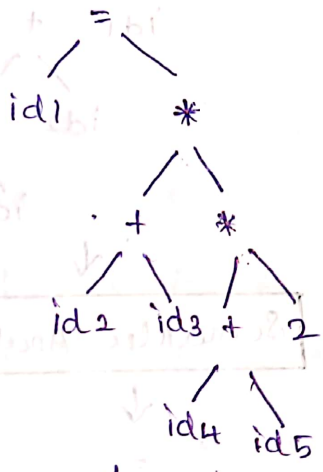
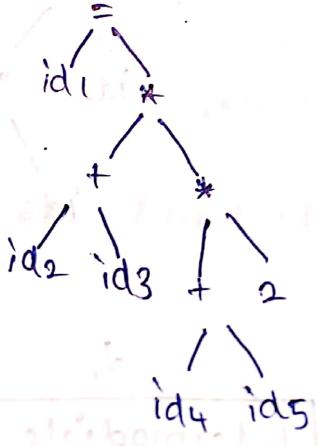
	↓ t1 := int to float (60) t2 := id3 * t1 t3 := id2 + t2 id1 := t3 ↓ Code optimizer ↓ t1 : id3 * 60.0 id1 := id2 + t1 ↓ Code generation ↓ MOV F id3, R2 MUL F #60.0, R2 MOV F id2, R1 Add F R2, R1 MOV F R1, id1	Intermediate code generation Code optimizer machine code

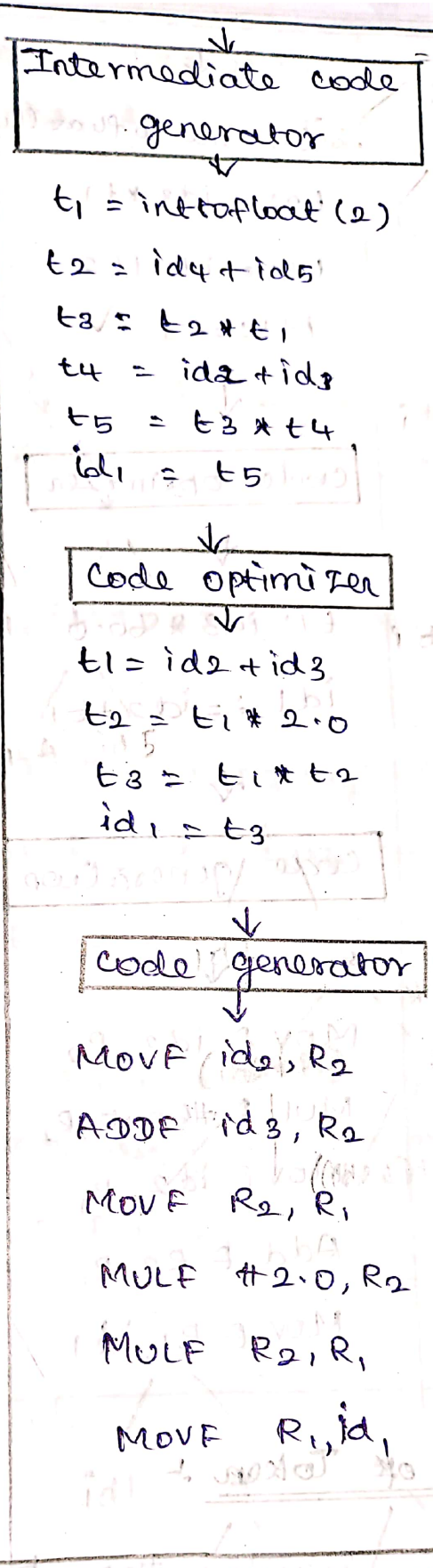
Specification of Token :-

⇒ To Specify the tokens regular expression are used. Regular expression are mathematical symbolisms which describe the set of strings of specific language.

H.W

Show how an input a = (b+c) * (b+c) * 2 get processed in compiler. Show the output at each stage of compiler. Also show the contents of symbol table.

Symbol	Input processing in compiler	Output
	Input code	Input code
a → Identifier	a = (b+c) * (b+c) * 2	
(→ Paranthesis	Lexical Analyzer	
b → Identifier	id1 = (id2+id3) * (id2+ id3) * 2	Token generation
+ → Plus sign		
c → Identifier	Syntax Analyzer	
) → Paranthesis		
* → multiple sign		Parse Tree
(→ paranthesis		
b → Identifier		
+ → Plus sign		
c → Identifier		
) → Paranthesis	Semantic Analyzer	
* → mutiple sign		Semantic Tree
2 → constant number		


Intermediate code generator
t1 = inttofloat (2)
t2 = id4 + id5
t3 = t2 * t1
t4 = id2 + id3
t5 = t3 * t4
id1 = t5
Code optimizer
t1 = id2 + id3
t2 = t1 * 2.0
t3 = t1 * t2
id1 = t3
Code generator
MOVF id2, R2
ADDF id3, R2
MOVF R2, R1
MULF #2.0, R2
MULF R2, R1
MOVF R1, id1
Intermediate code generation
code Optimizer
Machine code

Problems:

(1) Write a regular expression for a language containing the strings of length two over $\Sigma = \{0,1\}$

Soln:-

$$RE = (0+1)(0+1)$$

(2) write a RE for a language containing the strings of which end with "abb" over $\Sigma = \{a,b\}$

Soln:-

$$RE = (a+b)abb$$

(3) Writing RE for the language containing all the strings with any number a's and b's

$$RE = (a+b)^*$$

(4) write a RE to represent the Valid date format in mm-dd-yyyy format.

$$RE = [1-12]\ [1-31]\ [0-9]\ [0-9]\ [0-9]\ [0-$$

Convert the given NDFA with ε to NDFA w/o ε

Rule of conversion

$$\delta'(q, a) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q, \varepsilon), a))$$

where, $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-closure}(q_0)$

ε-closure $(q_0) = \{q_0, q_1, q_2\}$

$(q_1) = \{q_1, q_2\}$

$(q_2) = \{q_2\}$

$\hat{\delta}(q_0, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0, \varepsilon), 0))$

$= \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), 0)$

$= \varepsilon\text{-closure}(\delta\{q_0, 0\}, \{q_1, 0\}, \{q_2, 0\})$

$= \varepsilon\text{-closure}(\delta\{q_1\} \cup \phi \cup \phi)$

$= \varepsilon\text{-closure}(\delta(q_0))$

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

$\hat{\delta}(q_0, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), 1)$

$= \varepsilon\text{-closure}(\delta\{q_0, 1\}, \{q_1, 1\}, \{q_2, 1\})$

$= \varepsilon\text{-closure}(\delta\{\phi\} \cup \{q_1\} \cup \phi)$

$= \varepsilon\text{-closure}(\delta\{q_1\})$

$= \{q_1, q_2\}$

$\hat{\delta}(q_0, 2) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0, \varepsilon), 2))$

$= \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), 2)$

$= \varepsilon\text{-closure}(\delta(\{q_0, 2\}, \{q_1, 2\}\{q_2, 2\}$

$= \varepsilon\text{-closure}(\delta\{\phi\} \cup \{\phi\} \cup \{q_2\})$

$= \varepsilon\text{-closure}(\delta\{q_2\})$

$= \{q_2\}$

$\hat{\delta}(q_1, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_1, \varepsilon), 0))$

$= \varepsilon\text{-closure}(\delta(q_1, q_2), 0))$

$= \varepsilon\text{-closure}(\delta\{q_1, 0\}\{q_2, 0$

$= \varepsilon\text{-closure}(\delta(\phi \cup \phi \cup \phi))$

$= \varepsilon\text{-closure}(\delta\{\phi\}$

$= \phi$,,

$\hat{\delta}(q_1, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_1, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_1, q_2), 1))$

$= \varepsilon\text{-closure}(\delta\{q_1, 1\}, \{q_2, 1\})$

$= \varepsilon\text{-closure}(\delta\{q_1\})$

$= \{q_1, q_2\}$

$\hat{\delta}(q_1, 2) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_1, \varepsilon), 2))$

$= \varepsilon\text{-closure}(\delta(q_1, q_2), 2))$

$= \varepsilon\text{-closure}(\delta\{q_1, 2\}; \{q_2, 2\}$

$= \varepsilon\text{-closure}(\delta\{\phi\} \cup \{q_2\})$

$= \varepsilon\text{-closure}(\delta\{q_2\}$

$\hat{\delta}(q_2, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_2, \varepsilon), 0))$

$= \varepsilon\text{-closure}(\delta(q_2, 0))$

$= \varepsilon\text{-closure}(\delta(\phi))$

$= \{\phi\}$

$\hat{\delta}(q_2, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_2, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_2, 1))$

$= \varepsilon\text{-closure}(\delta\{\phi\})$

$= \{\phi\}$

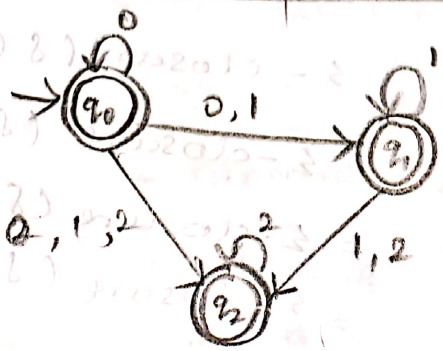
$\hat{\delta}(q_2, 2) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_2, \varepsilon), 2))$

$= \varepsilon\text{-closure}(\delta(q_2, 2))$

$= \varepsilon\text{-closure}(\delta(q_2))$

$= \{q_2\}$

State / Input	0	1	2
→ q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$



Convert the given NFA into its equivalent DFA

(OR)

Conversion of NFA with ε to DFA

ε-closure $(q_0) = \{q_0, q_1, q_2\} \Rightarrow A$

$(q_1) = \{q_1, q_2\} \Rightarrow B$

$(q_2) = \{q_2\} \Rightarrow C$

$\delta(A, 0) = \varepsilon$-closure $\{(q_0, q_1, q_2), 0\}$

$= \varepsilon$-closure $(\{q_0, 0\}, \{q_1, 0\} \{q_2, 0\})$

$= \varepsilon$-closure $(\{q_0\} \cup \phi \cup \phi)$

$= \varepsilon$-closure (q_0)

$\delta(A, 0) = \{q_0, q_1, q_2\} \Rightarrow A$ ||.

$\delta(A, 1) = \varepsilon$-closure $\{(q_0, q_1, q_2), 1\}$

$= \varepsilon$-closure $(\{q_0, 1\}, \{q_1, 1\}, \{q_2, 1\})$

$= \varepsilon$-closure $(\{\phi_0\} \cup \{q_1\} \cup \{\phi\}$

$= \varepsilon$-closure (q_1)

$\delta(A, 1) = \{q_1, q_2\} \Rightarrow B$

$\delta(A, 2) = \varepsilon$-closure $\{(q_0, q_1, q_2), 2\}$

$= \varepsilon$-closure $\{(q_0, 2), (q_1, 2), (q_2, 2$

$= \varepsilon$-closure $\{(q_2, 2)\}$

$= \varepsilon$-closure (q_2)

$= \{q_2\} = C$

$\hat{\delta}(B, 0) = \varepsilon$-closure $\{(q_1, q_2), 0)\}$

$= \varepsilon$-closure $\{(q_1, 0), (q_2, 0)\}$

$= \varepsilon$-closure $\{\{\phi\} \cup \phi\}$

$= \phi$

$\hat{\delta}(B,1) = \varepsilon\text{-closure}(\delta(q_1,q_2),1)$

$= \varepsilon\text{-closure}\{(q_1,1),(q_2,1)\}$

$= \varepsilon\text{-closure}\{(q_1) \cup \phi\}$

$= \{q_1, q_2\} \Rightarrow B$

$\hat{\delta}(B,2) = \varepsilon\text{-closure}(\delta(q_1,q_2),2)$

$= \varepsilon\text{-closure}(\{q_1,2\} \cup \{q_2,2\})$

$= \varepsilon\text{-closure}\{q_2\}$

$= \{q_2\} \rightarrow C$

$\hat{\delta}(C,0) = \varepsilon\text{-closure}(q_2,0)$

$= \varepsilon\text{-closure}\{\phi\}$

$= \{\phi\}$

$\hat{\delta}(C,1) = \varepsilon\text{-closure}(\delta(q_2),1)$

$= \varepsilon\text{-closure}(q_2,1)$

$= \varepsilon\text{-closure}\{(\phi)\}$

$= \{\phi\}$

$\hat{\delta}(C,2) = \varepsilon\text{-closure}\{(q_2),2)\}$

$= \varepsilon\text{-closure}\{(q,2)\}$

$= \varepsilon\text{-closure}(q_2)$

$= \{q_2\} \rightarrow C$

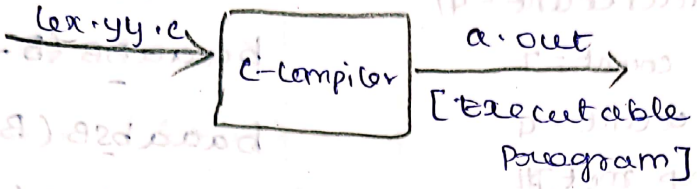
Start / Input	0	1	2
A	A	B	C
B	ϕ	B	C
C	ϕ	ϕ	C

Convert NFA to DFA

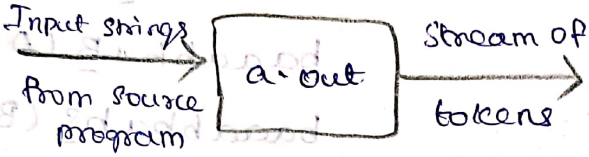
Refer . Toc . Note

M = {(P, q, r, s), {0, 1}, δ, P, {q, s}}

Lex :-



x.l → C prog → Output → Token



⇒ For efficient design of compiler various tools have been built for convertion lexical analyzer using the special perfect location for regular expression.

⇒ The lex specification file created using (.l)

⇒ The x.l is given to lex compiler to produce lex.yy.c (lexical analyzer).

⇒ The lex.yy.c contains the tokens in the form of table.

⇒ It produce object program a.out using C-compiler.

⇒ Finally, some input stream is given to a.out the sequence of token get generated.

(X) 2m

Structure of LEX :

1. Declaration Section (const, var)
2. Rule Section (action perform)
3. procedure Section (Procedure defined)

Lex Program :-

```
%{
int char_int=0, word_int=
Line_int=0; %}
word [^\t\n]+
%%
{word} { word_int++; char_int
+= yyleng; }
\n { char_int++; Line_int++;
{char_int++; }
%%
main()
{
yylex();
Printf("\n The character count= %d",
printf("\n The word count= %d", word_int);
Printf("\n The line count= %d", Line_int);
Print f("\n");
}
int yywrap()
{ return 1; }
```

Output:

```
[root@aap root]# lex linecount.l
[root@aap root]# gcc lex.yy.c -ll
[root@app root]#./a.out
hello Bhakti!!
You are
a newcomer
here
The character count = 41
The word count = 7
The line count = 4
[root@aap root]#
```

UNIT-2

SYNTAX ANALYSIS

CFG :

⇒ context free grammar consists of set of variables, terminals and use of rules

⇒ The context free grammar can be finally defined as a set denoted by $G = \{V, T, P, S\}$

V - non-terminal

T - Terminal

P - production rule

S - Start symbol

1. Let G be the Grammar

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

For the string baaabbabb find the left most right most derivation and parse tree.

Left most

S

bA (S → bA)

baS (A → aS)

baaB (S → aB)

baaaBB (B → aBB)

baaabSB (B → bS)

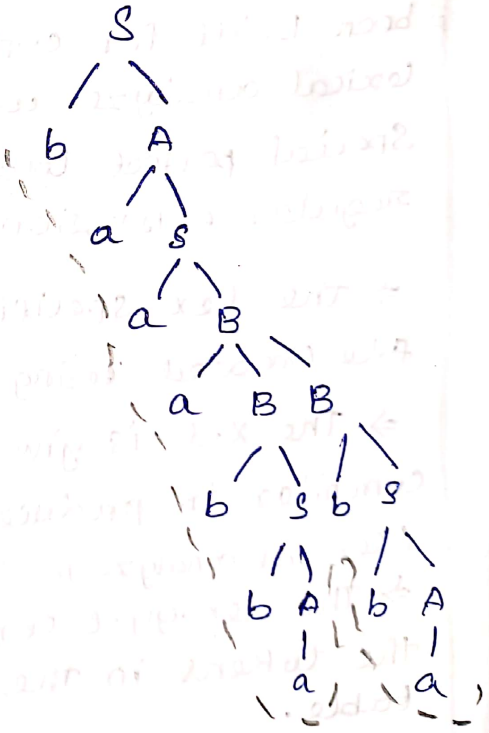
baaabbAB (S → bA)

baaabbaB (A → a)

baaabbabS (B → bS)

baaabbabbA (S → bA)

baaabbabba (A → a)



S (S → bA)

b A (A → aS)

b a S (S → aB)

b a a B (B → aBB)

b a a a B B (B → bS)

b a a a B b S (S → bA)

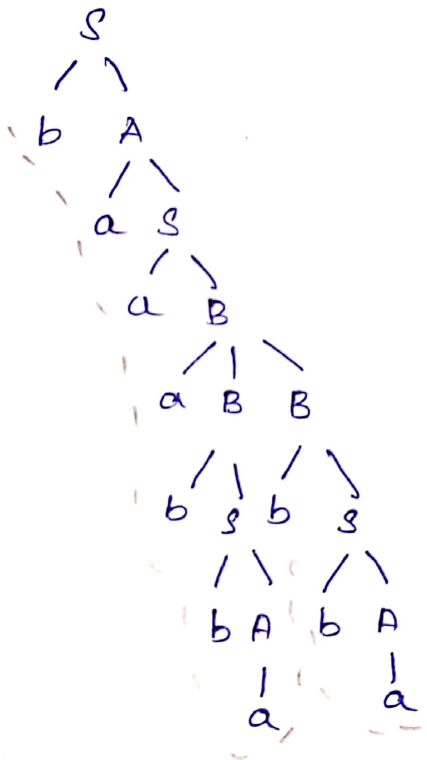
b a a a B b b A (A → a)

b a a a B b b a

b a a a b S b b a (S → bA)

b a a a b b A b b a (A → a)

b a a a b b a b b a



∴ Hence String proved.