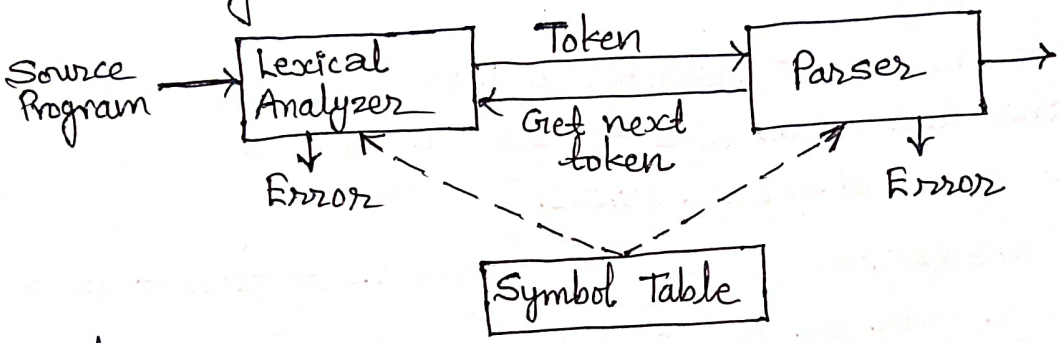
lexical analysis is done for validation of tokens

UNIT-2
Lexical Analyzer

Lexical Analysis: The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis.



Example: Example of Lexical Analysis, Tokens, Non-Tokens

Consider the following code that is fed to Lexical Analyzer.

```
#include <stdio.h>
int Largest (int x, int y)
{
   if (x>y) //This will compare two numbers
      return x;
   else
      return y;
}
```

Examples of Tokens created:

Lexeme	Token
int	Keyword
largest	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
)	Operator
{	Operator
if	Keyword

Examples of Non-Tokens:

Type	Examples
Comment	// This will compare 2 numbers.
Pre-processor directive	#include <stdio.h>

⊛. Role of Lexical Analyzer:

→ Lexical analyzer helps to identify token into the symbol table.
→ It can either work as a seperate module or as a sub-module.
→ It is responsible for eliminating comments and white spaces from the source program.
→ It reads the input character and produces output sequence of tokens that the Parser uses for syntax analysis.
→ It also generates lexical errors.
→ Lexical analyzer is used by web browsers to format and display a web page with the help of parsed data from Javascript, HTML, CSS.

⊛. Lexemes, Patterns, Tokens:

Lexemes: A lexeme is a sequence of alphanumeric characters that is matched against the pattern for token. A sequence of input characters that make up a single token is called a lexeme. A token can represent more than one lexeme.
Example: The token "String constant" may have a number of lexemes such as "bh", "sum", "area", "name" etc.

Patterns: Patterns are the rules for describing whether a given lexeme belonging to a token or not. The rule associated with each set of string is called pattern. Lexeme is matched against pattern to generate token. Regular expressions are widely used to specify patterns.

Token: Token is word, which describes the lexeme in source program. It is generated when lexeme is matched against pattern. A token is a logical building block of language. They are the sequence of characters having a collective meaning.

Example1: Example showing lexeme, token and pattern for variables.

- Lexeme: A1, Sum, Total
- Pattern: Starting with a letter and followed by letter or digit but not a keyword.
- Token: ID

Example2: Example showing lexeme, token and pattern for floating number.

- Lexeme: 123.45
- Pattern: Starting with digit followed by a digit or optional fraction and or optional exponent.
- Token: NUM

⊛Specifications of Tokens:

Regular Expression is a way to specify tokens. The regular expression represents the regular languages. The language is the set of strings and string is the set of alphabets. Thus, following terminologies are used to specify tokens:

1) **Alphabets:** The set of symbols is called alphabets. Example any finite set of symbols $\Sigma = \{0,1\}$ is a set of binary alphabets.

2) **Strings:** Any finite sequence of alphabets is called a string. Length of string is the total number of occurance of alphabets. e.g, the length of string 'Kanchanpur' is 10. A string of zero length is known as empty string and is denoted by ε (epsilon).

3) **Language:** A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them:

   1) **Union:** It is defined as the set that consists of all elements belonging to either set A or set B or both. In regular expression + symbol is used to represent union operation.

$$A = \{dog, ba, na\} \text{ and } B = \{house, ba\}$$

$$\text{then, } A \cup B = A + B = \{dog, ba, na, house\}$$

ii) Concatenation: It is the operation of joining character strings end-to-end. In regular expression dot operator (.) is used to represent concatenation operation.

$A.B = \{doghouse, dogba, bahouse, baba, nahouse, naba\}$

iii) Kleen closure: The Kleene closure Σ^* gives the set of all possible strings of all possible lengths over Σ including $\{\Phi\}$.

iv) Positive Closure: The set Σ^+ is the infinite set of all possible strings of all possible lengths over Σ excluding $\{\Phi\}$.

Regular Expressions: Regular expressions are the algebraic expressions that are used to describe tokens of a programming language. It uses three regular operations called union, concatenation and star.

Examples: Given the alphabet $A=\{0,1\}$.

1) $1(1+0)^*0$ denotes the language of all strings that begins with a '1' and ends with a '0'.

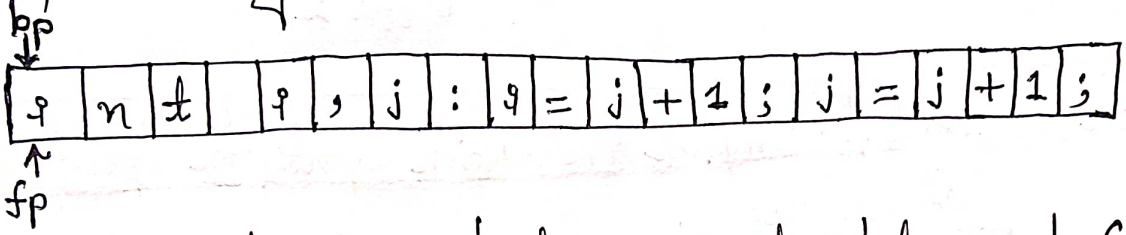
2) $(01)^* + (10)^*$ denotes the set of all strings that describe alternating 1s and 0s. [Note: Practice R.E for identifier in C, one dimensional array, two dimensional array & floating numbers. (6, 10, 11, 12 no. in kec book under RE topic)]

Recognition of Tokens: A recognizer for a language is a program that takes a string w, and answers "Yes" if w is a sentence of that language, otherwise "NO." The tokens that are specified using RE are recognized by using transition diagram or finite automata (FA). Starting from the start state we follow the transition defined. If the transition leads to the accepting state, then the token is matched and hence the lexeme is returned, otherwise other transition diagrams are tried out until we process all the transition diagram or the failure is detected. Recognizer of tokens takes the language L and the string s as input and try to verify whether $s \in L$ or not. There are two types of Finite Automata.

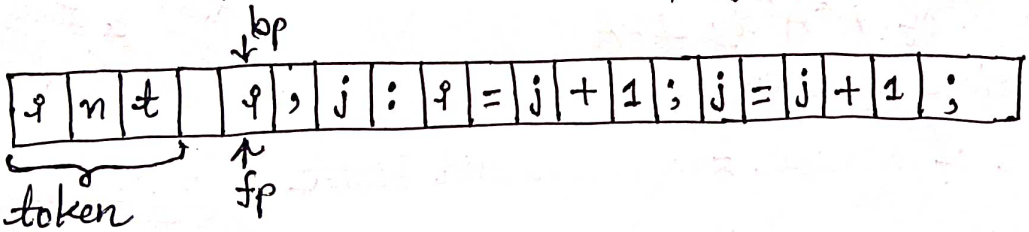
1). Deterministic Finite Automata (DFA).

2). Non Deterministic Finite Automata (NFA).

⊛. **Input Buffering:** Reading character by character from secondry storage is slow process and time consuming as well so, we use buffer technique to eliminate this problem and increase efficiency. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers, begin ptr (bp) and forward ptr (fp) to keep track of the pointer of input scanned. Initially both the pointers point to the first character of the input string as shown below;



The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as fp encounters a blank space the lexeme 'int' is identified. Then both bp and fp are set at next token as shown below and this process will be repeated for the whole program.

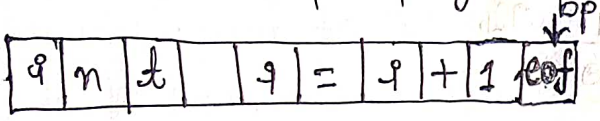


One Buffer Scheme: In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundry, to scan rest of the lexeme the buffer has to be refilled.

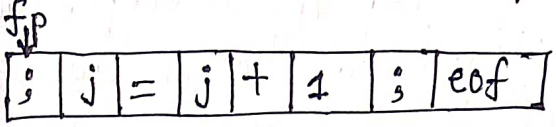
Two Buffer Scheme: In this scheme, two buffers are used to store input string and they are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves forward in search of

end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundry of first buffer end of buffer character (eof) should be placed at the end of first buffer, similarly for second buffer also. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.



Buffer 1



Buffer 2.

⊛ Read Following basic topics from 4th sem note of TOC (Unit-2 and Unit-3):

1) Finite Automata.
2) Deterministic Finite Automata (DFA).
3) Non Deterministic Finite Automata (NFA).
4) Epsilon NFA (ε-NFA).
5) Minimization of DFA. (state partition method) (same TOC में पढ़ा है)
6) Equivalence of Regular Expression and Finite Automata.
7) Reduction of Regular Expression to ε-NFA.
8) Converting ε-NFA to its equivalent DFA.

May not be asked in exam but these topics are necessary topics to understand for upcomming topics. These are basics.

⊛ Design of a Lexical Analyzer: [Imp]

Among these two algorithms one is asked in exam. Here if we see NFA then, it is always ε-NFA

Lexical Analyzer can be designed using following two algorithms:

Algorithm1: Regular Expression → NFA → DFA (two steps: first RE to NFA, then NFA to DFA).

Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA).

Same

#Algorithm1:

This consists of following two steps:

i.e, Reduction of Regular Expression to ε-NFA that we read in TOC Unit-3

1) Regular Expression to NFA (Thomson's Construction):

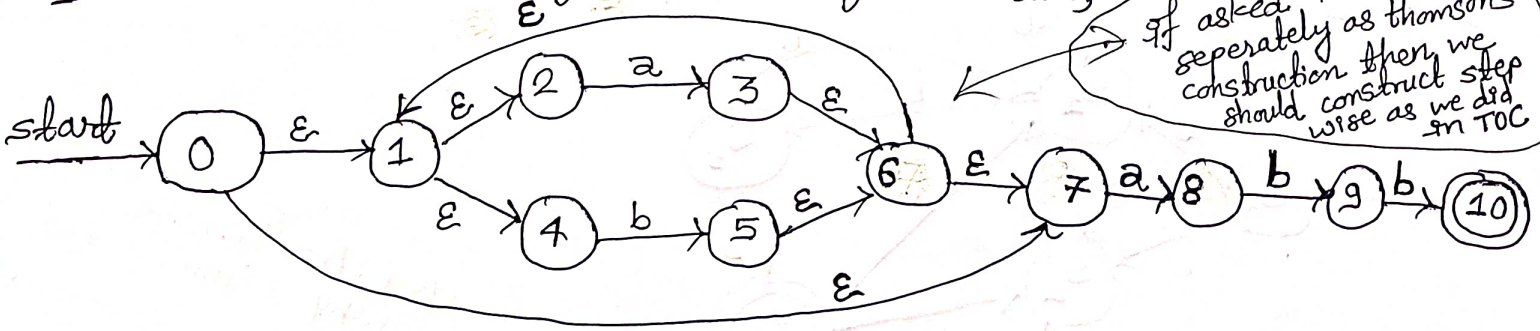
i.e, Converting ε-NFA to its equivalent DFA that we read in TOC Unit-2

2) Conversion from NFA to DFA (Subset Construction Algorithm):

Example: For Regular Expression $(a+b)^*abb$, first convert this RE to NFA, then convert resulting NFA to DFA.

Solution: The NFA of given RE is given below;

If asked in exam seperately as thomson's construction then we should construct step wise as we did in TOC



Now we convert above NFA to DFA as,

The starting state of DFA = S_0 = ε-closure (S_0)

(starting state ट start state बाट ε लिएर कुन कुन state सम्म पुगेको छ त्यसको list)

$\Rightarrow S_0 = \{0,1,2,4,7\}$

Mark S_0, (Since we have two input symbols a, & b so we check for each input)

($S_0 = 0,1,2,4,7$ so S_0 मा a आउदा 3, र 8 मा गएको छ)

For a: ε-closure(δ(S_0, a)) = ε-closure (3,8) = {1,2,3,4,6,7,8} → S_1

For b: ε-closure(δ(S_0, b)) = ε-closure (5) = {1,2,4,5,6,7} → S_2

(कुनै नयाँ set आउदा नयाँ state ले denote गर्ने like here S_1)

(just denote गरेको यदि DFA बनाउदा सजिलो होस भनेर)

(previous step मा आएको नयाँ set check गर्ने next step मा)

Mark S_1,

For a: ε-closure (δ(S_1, a)) = ε-closure (3,8) = {1,2,3,4,6,7,8} → S_1

For b: ε-closure(δ(S_1, b)) = ε-closure (5,9) = {1,2,4,5,6,7,9} → S_3

Mark S_2,

For a: ε-closure(δ(S_2, a)) = ε-closure (3,8) = {1,2,3,4,5,7,8} → S_1

For b: ε-closure(δ(S_2, b)) = ε-closure (5) = {1,2,4,5,6,7} → S_2

Mark S_3,

For a: ε-closure(δ (S_3, a)) = ε-closure (3,8) = {1,2,3,4,6,7,8} → S_1

For b: ε-closure(δ(S_3, b)) = ε-closure (5,10) = {1,2,4,5,6,7,10} → S_4

Mark S_4,

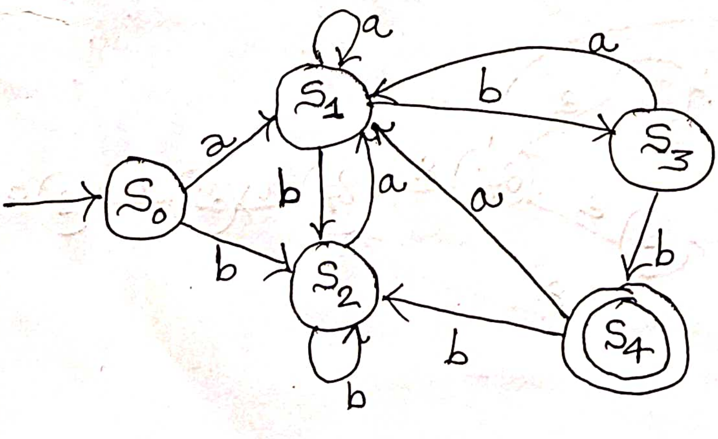
For a: ε-closure (δ(S_4, a)) = ε-closure (3,8) = {1,2,3,4,6,7,8} → S_1

For b: ε-closure(δ (S_4, b)) = ε-closure (5) = {1,2,4,5,6,7} → S_2.

(No new states here S_1, S_2 are already checked so we stop here. We stop when there are no any new states to check)

S_0 is the starting state. S_4 is an accepting state of DFA since final state of NFA is 10 i.e, states containing 10 are member of final state of DFA. (Since S_4 = {1,2,4,5,6,7,10}).

Now, constructing DFA using above information:



Note: Practice more examples from kec book

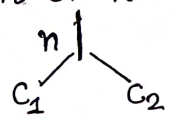
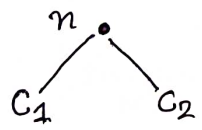
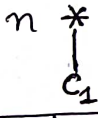
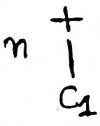
#Algorithm 2 (Conversion from RE to DFA Directly):

Conversion steps:

for exam mostly numericals are asked theory for understanding only

1. Augment the given regular expression by concatenating it with special symbol #.
2. Create the syntax tree for this augmented regular expression.
3. Then each alphabet symbol (including #) with be numbered as position numbers.
4. Compute functions nullable, firstpos, lastpos, and followpos.
5. Finally construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n), and followpos(i) from the syntax tree.

Rules for calculating nullable, firstpos and lastpos:

Node n	nullable (n)	firstpos (n)	lastpos (n)
A leaf labelled ε	True	ϕ	ϕ
A leaf with position i	False	{i} (position of leaf node)	{i}
An or node 	Nullable (c_1) or Nullable (c_2).	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
A concatenation node 	Nullable (c_1) and Nullable (c_2).	If (Nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2). else firstpos(c_1)	If(Nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2). else lastpos(c_2).
A star node 	True	firstpos(c_1)	lastpos(c_1)
A +ve closure node 	False	firstpos(c_1)	lastpos(c_1)

Now we calculate followpos for each i we numbered. Now we can construct DFA with starting state S_1 = firstpos(root) from syntax tree and marking each state for each input until no new unmarked states occur.

Example: Convert the regular expression $(a|b)^*a$ into equivalent DFA by direct method.

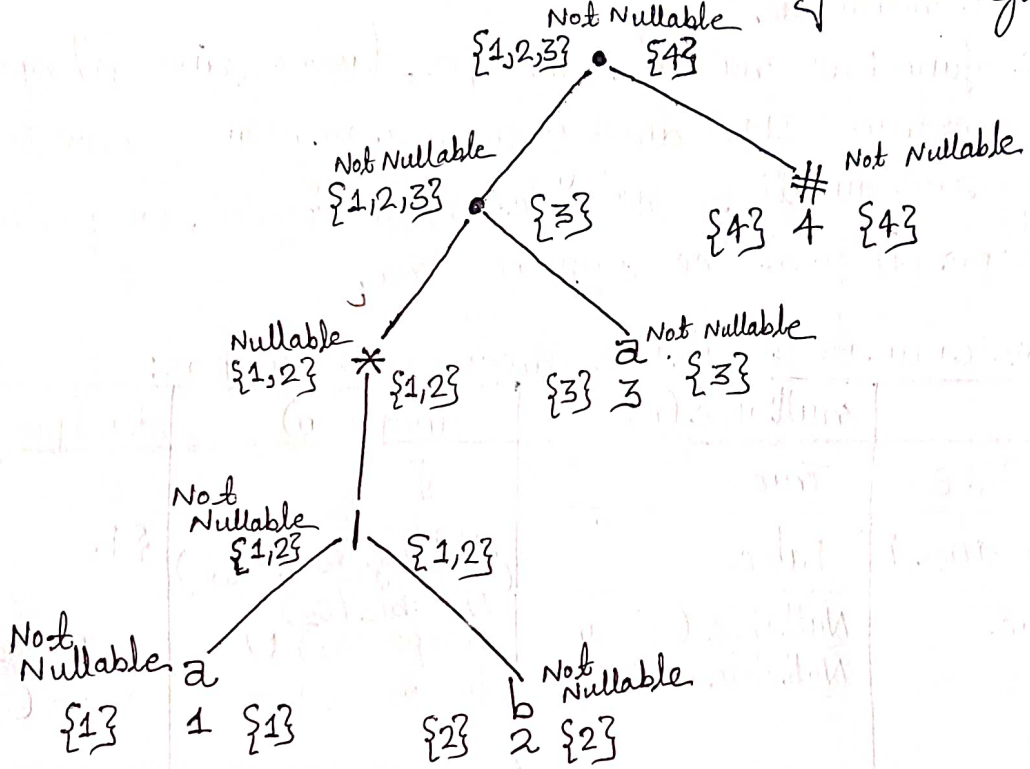
Solution:

Step1: At first we augment the given regular expression as,

$(a|b)^*.a.\#$ ← last मा # थपे augment हुन्छ। अनि operator हरु product form मा हुने भएकाले dot operator हुन्छ।

1 2 3 4 ← alphabets including # numbered

Step2: Now we construct syntax tree of augmented regular expression as,



Step3: Now we compute followpos as,

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {∅} ← empty

Step4: Now we start with starting state as,

S_1 = firstpos(root node of syntax tree) = {1,2,3}

Since we are marking for a for S_1 = {1,2,3}. Now look at step1 where where a occurs in 1,2,3. Here a occurs in 1 & 3 so Union of 1 & 3

Mark S_1,

For a: followpos(1) ∪ followpos(3) = {1,2,3,4} → S_2 ← marked as new state using variable or state name S_2

In step 1 between 1,2,3 b occurs at 2 only

for b: followpos(2) = {1,2,3} → S_1

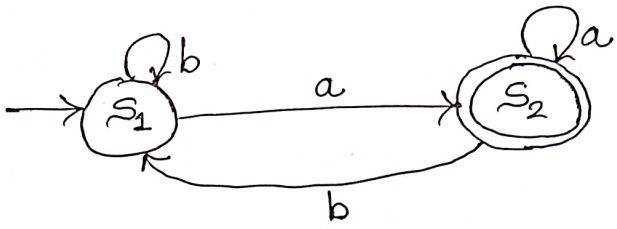
Mark S_2,

For a: followpos(1) ∪ followpos(3) = {1,2,3,4} → S_2

for b: followpos(2) = {1,2,3} → S_1.

No new states occur so we stop marking here. Accepting state means state containing position of # (i.e, 4). So, here S_2 = {1,2,3,4} is accepting state.

Now, based on the above information, resulting DFA of given regular expression is as follows:



Practice these questions also if any confusion or want to try to match answer then refer ref book for solution page no 37.

Q1. Convert the regular expression $(a|\varepsilon)bc^*$ into equivalent DFA by direct method.

Q2. Convert the regular expression $ba(a+b)^*ab$ into equivalent DFA by direct method.

⊛ Flex: An Introduction

possibility of asking in short note for 2.5 marks

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Flex Specification: A flex specification consists of three parts:

Regular definitions, C declarations in %{ %}

%%

Translation rules

This is syntax. %% are opening and closing tags.

%%

User defined auxiliary procedures

The translation rules are of the form:

P_1	{action 1}
P_2	{action 2}
. . .	. . . . .
P_n	{action n}

In all parts of the specification comments of the form /* comment text */ are permitted.