Date ______
Page ______

# Lexical Analysis

→ The scanning/lexical analysis phase of a compiler performs the task of reading a stream of characters as an input and produce a sequence of tokens such as names, keywords, numbers etc for syntax analyzer.

→ It discards the white space and comments between the tokens and also keep tracks of line number.

→ lexical analyzer correlate error message with source program.

## Role of lexical Analyzer

→ lexical analyzer is the first phase of a compiler.

→ Its main task is to read the input characters and produce as output a sequence of tokens that parser uses for syntax analysis.

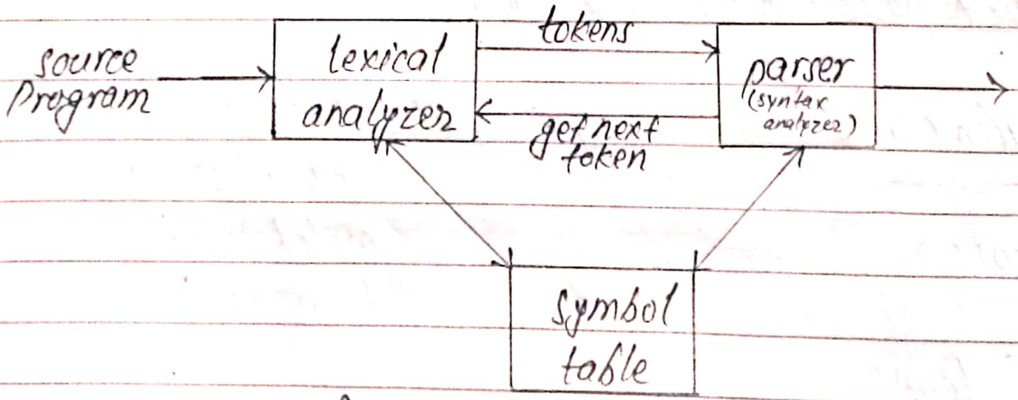


fig: lexical Analyzer

→ As in the figure, upon receiving a "get next token" command from the parser the lexical analyzer read input characters until it can identify the next token.

→ It removes the comments from the source program.

→ It keeps track of line numbers while scanning the new line characters. These line numbers are used by the error handler to print the error message.

## Tokens, Patterns, Lexemes

→ Token is a sequence of characters that can be treated as a single logical entity.
E.g. Identifiers, keywords, operators, constants etc.

→ E.g. of non-tokens:
Comments, preprocessor directive; macros, blanks, tabs, newline etc

→ Pattern: A set of string in the input for which the same token is produced as output. This set of string is described by a rule called a pattern associated with the token.
E.g. The pattern for the Pascal identifier token, id is:
id → letter (letter | digit)* i.e. letter followed by letter & digits.

→ A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
E.g. The pattern for the RELOP token contains six lexemes

→ In SQL

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

↳ != (In C)

# Input: x = x * (acc + 123)

| token | lexemes | token | lexemes | token | lexemes |
|---|---|---|---|---|---|
| identifier | x | star | * | plus | + |
| equal | = | left-paren | ( | integer | 123 |
| identifier | x | identifier | acc | right-paren | ) |

Date
Page

## Attribute of tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the attribute of the token.
→ Attributes are used to distinguish different lexemes in a token.

Some attributes:
- <id, attr> where attr is pointer to the symbol table.
- <assigop,> no attribute is needed (if there is only one assignment operator)
- <num, val> where val is the actual value of the number.

→ Token type and its attribute uniquely identifies a lexeme.

E.g. dest = source+5
Tokens: <id, pointer to symbol_table ~~for~~ entry for dest>
<assign-op,>
~~<num, integer val 5>~~
<id, pointer to symbol_table entry for source>
<add-op, >
<num, integer val 5>

E = M * C ** 2
<id, pointer to symbol-table entry for E>
<assign-op,>
<id, pointer to symbol-table entry for M>
<mult-op, >
<id, pointer to symbol-table entry for C>
<exp-op, >
<num, integer value 2>

Date
Page

## Lexical Error

→ Dusing the lexical analysis phase this type of error can be detected.

→ lexical error is a sequence of characters that doesnot match the pattern of any token. lexical phase error is found during the execution of the program.

→ lexical phase error can be:

- spelling error
- Exceeding length of identifier or numeric constant
- To remove the character that should be present
- Appearance of illegical characters.
- To replace a character with an incorrect character
- ~~Transposition of two characters~~

E.g.

```
void main()
{
    int x=10, y=20;
    char *a;
    a=&x;
    x=5xab;
}
```

In this code, 5xab is neither a number nor an identifier. so this code will show the lexical Error.

⇒ Possible error recovery actions are:

- Panic mode recovery - deleting successive characters ~~un~~ until a well formed token is formed.
- Inserting a missing character.
- Replacing a missing character by a correct character.
- Transposing two adjacent character.
- Deleting an extraneous character.

# #General approaches to the implementation of a lexical analyzer

There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator like lex or flex, to produce lexical analyzer from a regular-expression based specification. The generator provides routines for reading & buffering the input. (easiest to implement; least efficient).

2. Write the lexical analyzer in high level programming language like C, using the I/O facilities of that language to read and buffering the input. (Intermediate in ease, efficiency).

3. Write the lexical analyzer in assembly language and explicitly manage the input and buffering. (Hardest to implement, but most efficient)

## Lookahead and Buffering

→ Many times, a scanner has to look ahead several ~~times~~ characters from the current character in order to recognize the token.

For e.g. 'int' is keyword in C, while the term 'inp' may be a variable name. When the character 'i' is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.

→ In order to efficiently move back and forth in the input stream, input buffering is used.

## Input Buffering

→ lexical analysis needs to look ahead several characters before a match can be announced.

→ we have two buffer input scheme that is useful when look ahead is necessary

- Buffer Pair
- sentinels

### Buffer Pair (2N Buffering)

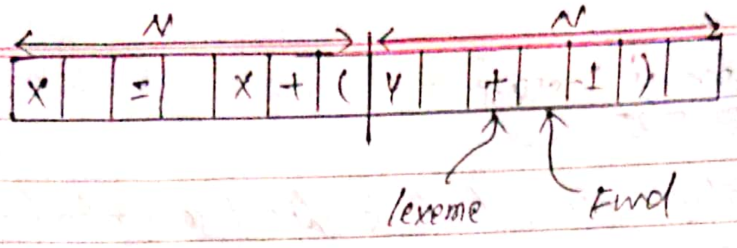
In this technique buffer is divided into two halves with N-characters each, where N is number of characters on the disk block like 1024 or 4096. Rather than reading character by character from file we ~~red N~~ read N input character at once If there are fewer than N character in input eof marker is placed. There are two pointers : lexeme pointer and forward pointer.

- lexeme pointer points to the start of the lexeme while forward pointer scans the i/p buffer for lexeme.
- when forward pointer reaches the end of one half, second half is loaded and forward pointer points to the begining of the next half.

Pseudo code :

{fwd = forward pointer

```
if (fwd at end of first half)
    reload second half;
    set fwd to point to begining of second half;
else if (fwd at end of second half)
    reload first half;
    set fwd to point to begining of first half;
else
    fwd++;
```



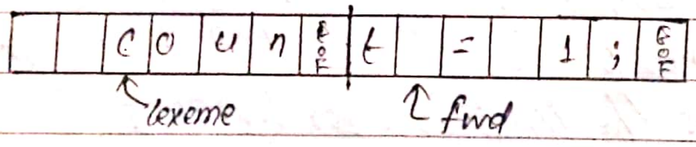
## • Sentinels

→ While using buffer pair technique, we have to check each time fwd is moved that it doesn't crosses the buffer half and when it reaches end of buffer, the other one needs to be loaded.

→ We can solve this problem by introducing a sentinel character at the end of both halves of buffer.

→ This sentinel can be any character which is not a part of source program. EOF is usually preferred as it will also indicate end of source program.

→ It signals the need for some special action (fill other buffer-half, or terminate processing).



Pseudo code:

```
fwd ++
if (*fwd == EOF)
{
    if (fwd at end of first half)
        ...
    else if (fwd at end of second half)
        ...
    else
        terminate processing
}
```

Date
Page

## Specification of Tokens

There are 3 specifications of tokens:
1) strings
2) Language
3) Regular expression

### strings and Language:

→ An alphabet Σ is a finite set of symbols (characters)
e.g. {0, 1} is the binary alphabet

→ A string 's' is a finite sequence of symbols from Σ.
- |s| denotes the length of string s.
- ϵ denotes the empty string, thus |ϵ| = 0

→ A language is a specific set of strings over some fixed alphab
Σ.
- ϕ → the empty ~~set~~ language.
- {ϵ} → language consisting of only empty string.

### Operations on strings

1. Prefix of s: A string obtained by removing zero or more trailing symbols of string s. E.g. ban is a prefix of banana.
2. suffix of s: A string formed by deleting zero or more of the leading symbols of s. E.g. nana is a suffix of banana.
3. substring of s: A string obtained by deleting a prefix and a suffix from s. e.g. nan is a substring of banana.
4. Proper prefix, suffix, or substring of s: Any non-empty string x that is a prefix, suffix, or substring of s that s<>x.

5. Subsequence of s : Any string formed by deleting zero or more not necessarily contiguous symbols from s. E.g. baaa is a subsequence of banana.

## Operations on Languages

let L and M be two language then

1. Union : $L \cup M = \{s \mid s$ is in $L$ or $s$ is in $M\}$
2. Concatenation : $LM = \{st \mid s$ is in $L$ and $t$ is in $M\}$
3. Kleene closure of L : $L^*$ = "zero or more Concatenation of " L.
4. Positive closure of L : $L^+$ = "one or more Concatenation of" L.

## Regular Expressions :

The regular expression over alphabet specifies a language according to the following rules:

1. $\epsilon$ is a regular expression that denotes $\{\epsilon\}$, i.e. the set containing the empty string.
2. $a \in \Sigma$ is a regular expression denoting $\{a\}$.
3. If $r$ and $s$ are regular expressions denoting languages $L(r)$ and $L(s)$ respectively, then
   a) $(r)|(s)$ is a R.E denoting the language $L(r) \cup L(s)$.
   b) $(r)(s)$ is a R.E. denoting the language $L(r)L(s)$.
   c) $(r)^*$ is a R.E denoting the language $(L(r))^*$.
   d) $(r)$ is a R.E denoting the language $L(r)$.

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

## Properties of Regular Expression

For regular expression r, s & t

1. $r|s = s|r$ ( | is commutative)
2. $r|(s|t) = (r|s)|t$ ( | is associative)
3. $(rs)t = r(st)$ (Concatenation is associative)
4. $r(s|t) = rs|rt$ (Concatenation distributes over | )
5. $\epsilon r = r\epsilon = r$ ($\epsilon$ is the identity element for concatenation)
6. $r^* = (r|\epsilon)^*$ (Relation bet$^n$ * and $\epsilon$)
7. $r^{**} = r^*$ (* is idempotent)

## Regular Definition :

- If $\Sigma$ is an alphabet of basic symbols, then a regular definition is a sequence of definition of the form

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

where, $d_i$ is a distinct name and $r_i$ is a regular expression over symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Basic symbols previously defined names

e.g. In C the RE for identifiers can be written using the regular definition as

letter → A|B|...|Z|a|b|...|z|_

digit → 0|1|2|...|9

identifier → letter ( letter | digit )*

## Notational shorthands:

- This shorthand is used in certain constructs that occur frequently in regular expression.
- The following shorthands are often used:

$r^+ = rr^*$

r? $r? = r \mid \epsilon$ (zero or ~~more~~ one occurrences)

$[a\text{-}z] = a \mid b \mid c \mid \dots \mid z$

E.g.

digit → [0-9]

num → $\text{digit}^+ (.\text{digit}^+)? (E(+\mid-)?\text{digit}^+)?$

## Recognition of tokens

- A recognizer for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- Recognition of token implies implementing a regular expression recognizer. That entails the implementation of finite automaton.
- ✓ The tokens that are specified using RE are recognized by using finite automata.
- Recognizers 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)

# Deterministic Finite Automaton (DFA)

→ FA is deterministic, if there is exactly one transition for each (state, input) pair.

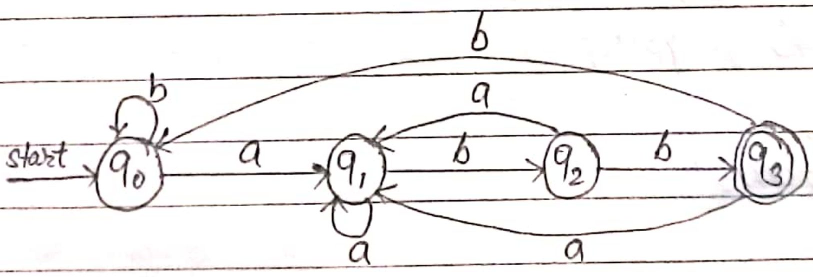
→ It is faster recognizer but it ~~make~~ may take more space.

→ A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where,

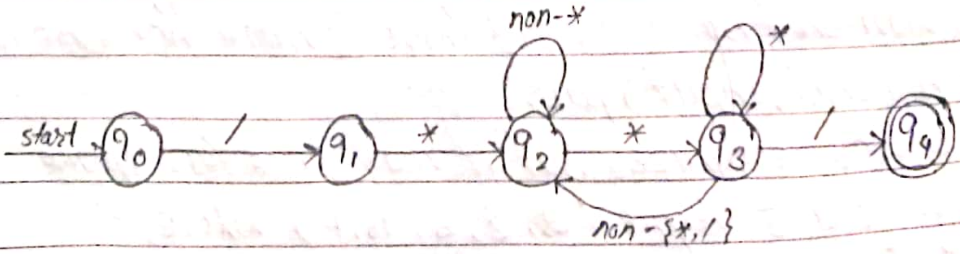
- $Q$ is a finite set of states.
- $\Sigma$ is a finite set of input alphabets
- $\delta$ is a transition function that maps $Q \times \Sigma \to Q$
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of final states.

E.g.

- DFA for R.E. $(a+b)^*abb$:



- DFA to match c-style comments:



Note: Transition not showing from a state with any symbol is going to non-accepting (trapping state).

→ A state transition from one state to another on the path is called a move.

## Implementing DFA: Algorithm

- Algorithm to simulate a DFA(D), with start state $q_0$, that returns "yes" if the input string is accepted else return "no".

```
DFAsim (D, q0)
{
    q = q0;
    c = getchar();
    while (c != EOF)
    {
        q = move(q, c);      // transition function
        c = getchar();
    }
    if (q is in F)
        return 'Yes';
    else
        return 'No';
}
```

## ※ Non-Deterministic Finite Automaton (NFA)

→ FA is non-deterministic, if there is more than one transition for each (state, input) pair.

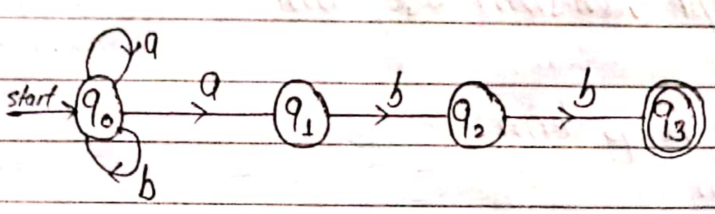
→ It is slower recognizer but it may take less space.

→ An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where,

- $Q$ is a finite set of states
- $\Sigma$ is a finite set of input alphabet
- $\delta$ is a transition function that maps $Q \times \Sigma \rightarrow 2^Q$
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of final states.

E.g.

- NFA for R.E. $(a+b)^*abb$.



→ ε-NFA :

→ ε-transitions are allowed in NFAs.

→ In other words, we can move from one state to another one without consuming any symbol.

E.g.

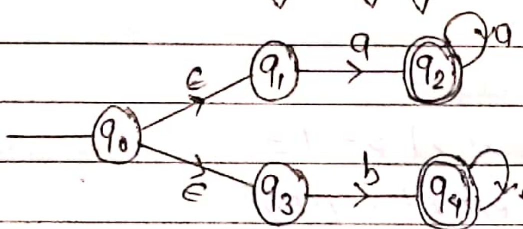


fig: state machine with ε-moves that is equivalent to the regular expression $aa^*+bb^*$.

✓ Implementing NFA:

```
q = ε-closure ({q0});
c = getchar();
while ( c != eof )
{
    q = ε-closure (move(q,c));
    c = getchar();
}
if (q∩F ≠ φ) then
    return "Yes";
else
    return "No";
```

# #Grammar

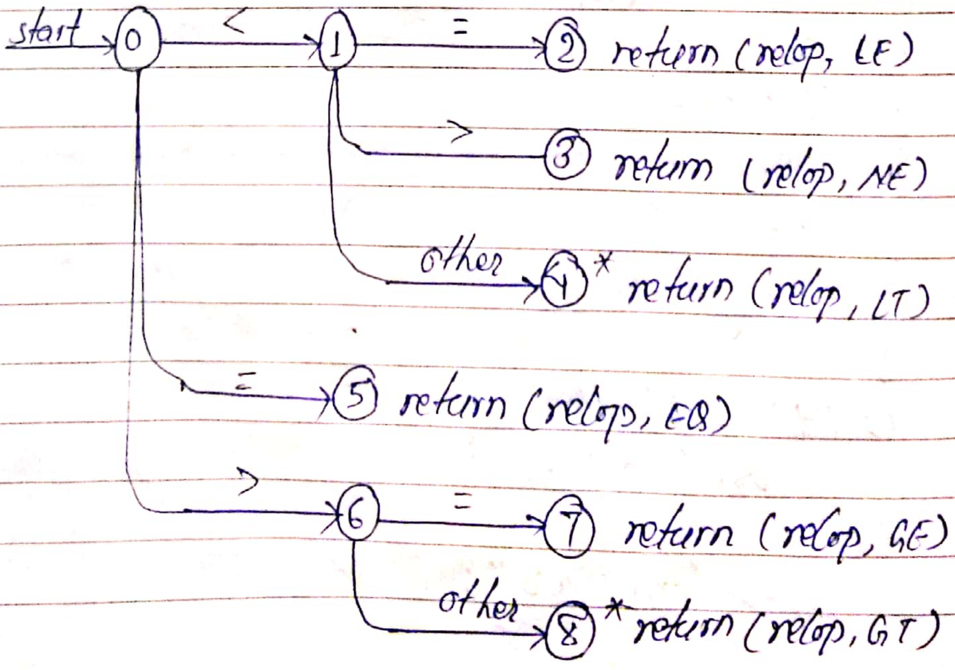
stmt → if expr then stmt
 | if expr then stmt else stmt
 | ε
expr → term relop term
~~term~~ | term
term → id
 | num

Regular definitions: for above grammar:

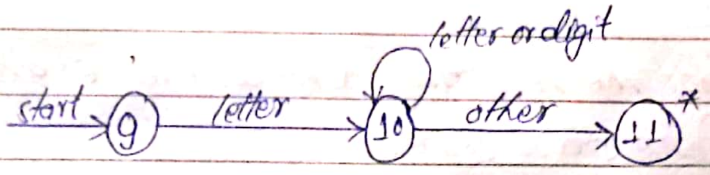
if → if
then → then
else → else
relop → < | <= | <> | > | >= | =
$id \rightarrow letter\ (letter \mid digit)^*$
$num \rightarrow digit^+ (.digit^+)? (E(+ \mid -)? digit^+)?$

Transition diagram:

relop → < | <= | <> | > | >= | =



id → letter (letter / digit)*



return (gettoken(),
install-id())

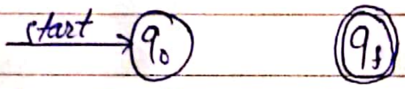
## R.E to NFA / Thomson's construction

→ It guarantees that the resulting NFA will have exactly one final state, and one start state.

Input : RE, r, over alphabet Σ
Output : ε-NFA accepting L(r)

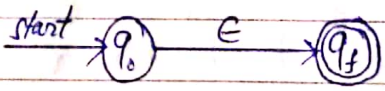
The method consists of following steps:

i) For φ, we construct the NFA as

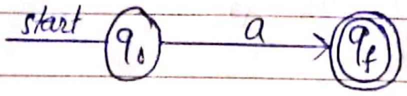


where $q_0$, is an initial state & $q_f$ is a final state.

ii) For ε, we construct the NFA as

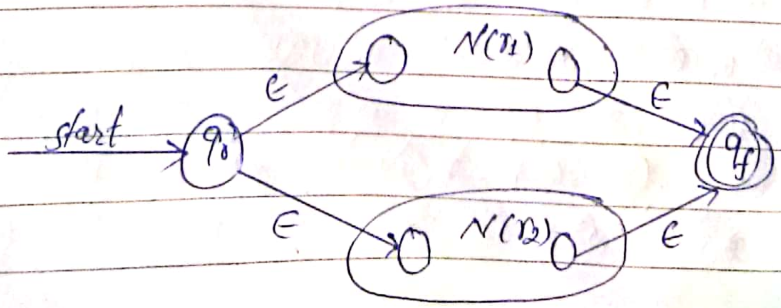


iii) For every a ∈ Σ we construct the NFA as

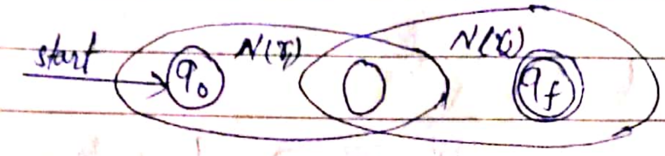


iv) If $N(r_1)$ and $N(r_2)$ are NFAs for R.E $r_1$ and $r_2$

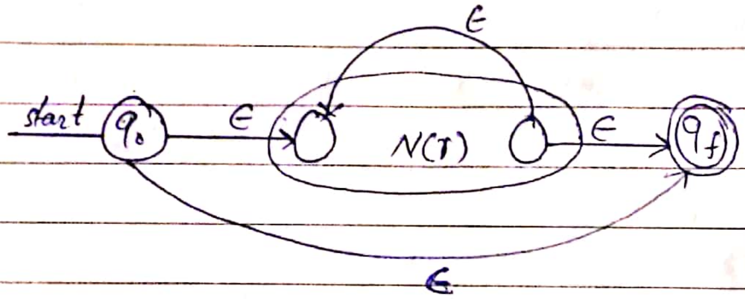
a) For R.E. $r_1 | r_2$ (i.e. $r_1 + r_2$) we construct the NFA as



b) For the R.E $r_1 r_2$, we construct the NFA as.



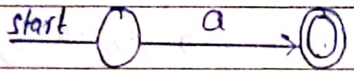
vi) For the R.E. $r^*$, we construct the NFA as



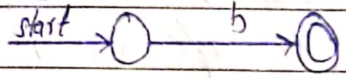
Examples

① $(a+b)^*a$ to NFA.

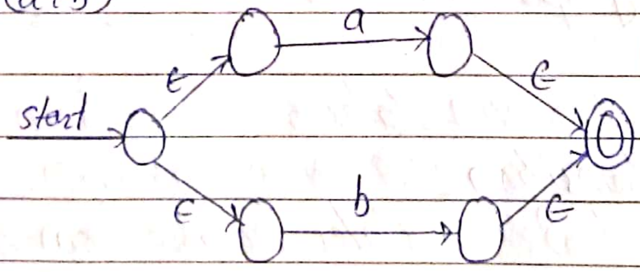
For a



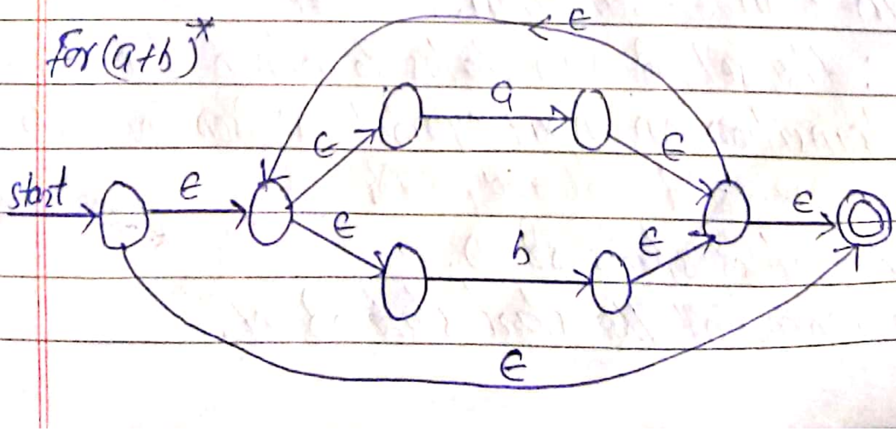
For b



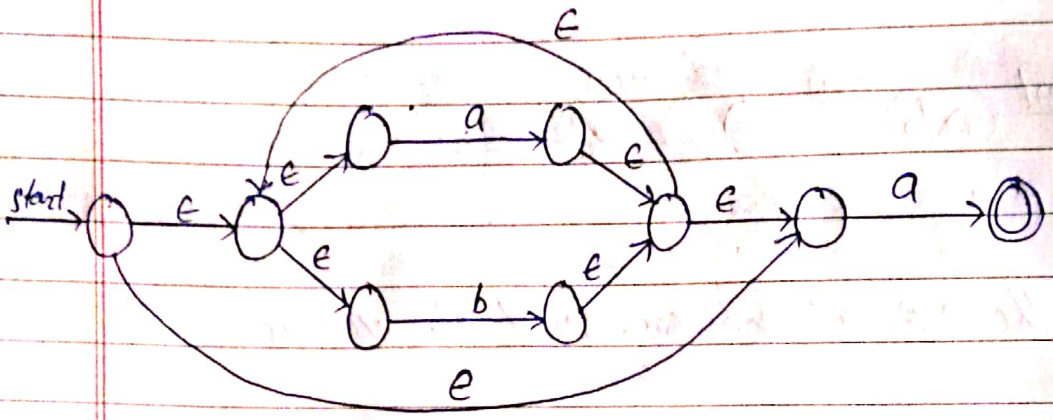
For (a+b)



For $(a+b)^*$



For $(a+b)^*a$



See Book for other Eg.

## NFA to DFA (subset constuction)

The subset construction algorithm converts an NFA into a DFA.

→ In this algorithm we use the symbol N to represent an NFA and D for representing DFA.

→ This algorithm constructs a transition table Dtran for D.

We use the following operations: (s represent an NFA state and T a set of NFA states)

- ε-closure(s) : the set of NFA states rechable from NFA state s on ε-transition. i.e. ε-closure(s) = $\{s\} \cup \{t \mid s \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} t\}$
- ε-closure(T) : the set of NFA states rechable from NFA state s in T on ε-transition. i.e. $\cup_{s \in T}$ ε-closure(s).
- move (T, a) : the set of NFA states to which there is a transition on input symbol 'a' from NFA states s in T. i.e. $\{t \mid s \xrightarrow{a} t \text{ and } s \in T\}$

→ Dstates is the set of states of D.

→ We use $s_0$ to represent the start state of N.

## Computation of $\epsilon$-closure :

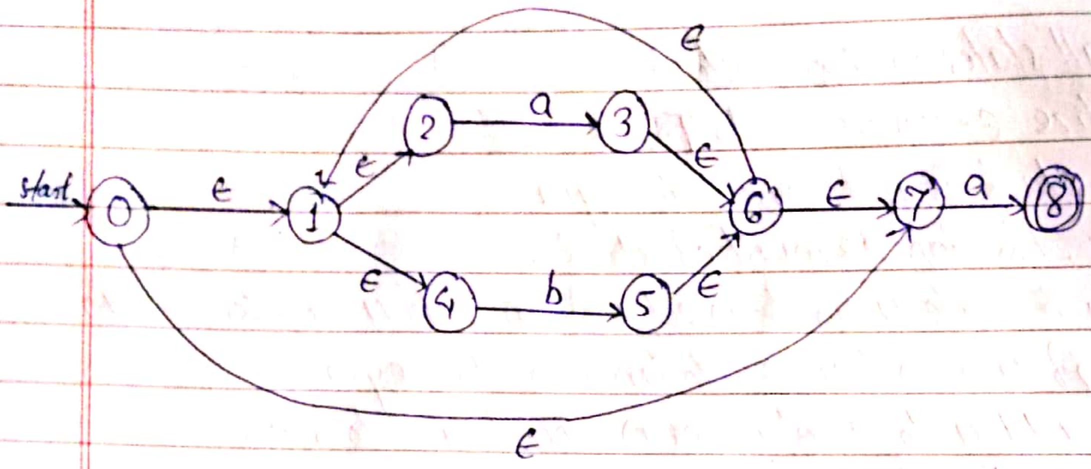
```
Push all states in T onto stack;
Initialize $\epsilon$-closure(T) to T;
 while stack is not empty do begin
  Pop t, the top element of stack;
  for each state u with an edge from t to u labeled $\epsilon$ do
        if u is not in $\epsilon$-closure(T) do begin
        add u to $\epsilon$-closure(T); push u onto stack;
        end
end
```

## ~ Subset construction Algorithm:

```
Put $\epsilon$-closure($s_0$) as an unmarked states into Dstates.
   while there is an unmarked state T in Dstates do.
     mark T;
     for each input symbol $a \in \Sigma$ do
          U = $\epsilon$-closure(move(T,a))
          if U is not in Dstates then
                add U as an unmarked state to Dstates
          end if
             Dtran[T,a] = U
     end do
   end do
```

→ The start state of DFA is $\epsilon$closure($s_0$)

## Example



ε-closure → Init state → all states

Sol<sup></sup>

The initial state of the NFA is 0.
Therefore, the initial state of the DFA is,

$A = \epsilon\text{-closure}(\{0\}) = \{0, 1, 2, 4, 7\}$

Here

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

Now,

$Dtran[A, a] = \epsilon\text{-closure}(move(A, a))$ // Dtran is a transition table for DFA.

$= \epsilon\text{-closure}(\{3, 8\})$

$= \{1, 2, 3, 4, 6, 7, 8\}$

$= B$ (say)

$Dtran[A, b] = \epsilon\text{-closure}(move(A, b))$

$= \epsilon\text{-closure}(\{5\})$

$= \{1, 2, 4, 5, 6, 7\}$

$= C$ (say)

$Dtran[B, a] = \epsilon\text{-closure}(move(B, a))$

$= \epsilon\text{-closure}(\{3, 8\})$

$= \{1, 2, 3, 4, 6, 7, 8\}$

$= B$

$\text{Dtran}[B,b] = \epsilon\text{-closure}(\text{move}(B,b))$
$= \epsilon\text{-closure}(\{5\})$
$= \{1,2,4,5,6,7\}$
$= C$

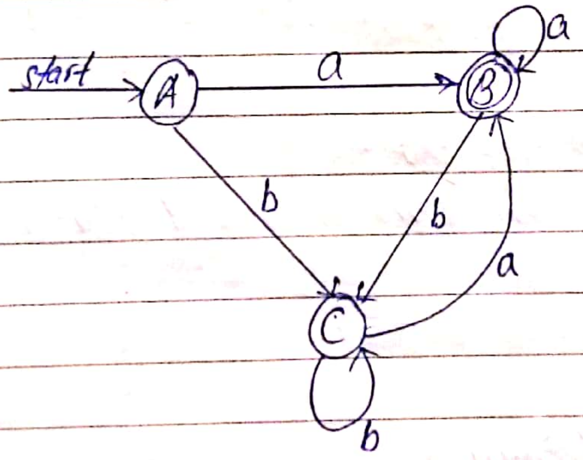
OR
~~for Dtran~~ to Calculate Dtran we Can use table

|  | a | b |
|---|---|---|
| { } |  |  |
| { } |  |  |
| { } |  |  |

$\text{Dtran}[C,a] = \epsilon\text{-closure}(\text{move}(C,a))$
$= \epsilon\text{-closure}(\{3,8\})$
$= \{1,2,3,4,6,7,8\}$
$= B$

$\text{Dtran}[C,b] = \epsilon\text{-closure}(\text{move}(C,b))$
$= \epsilon\text{-closure}(\{5\})$
$= \{1,2,3,4,5,6,7\}$
$= C$

Now the equivalent DFA is



$(Q, \Sigma, \delta, q_0, F) \rightarrow \epsilon\text{-NFA}$
$(Q', \Sigma, \delta^*, q_0', F') \rightarrow \text{DFA}$ → ENFA to DFA
$\delta^*$ is defined as:
$\delta^*(q,a) = \epsilon\text{-closure}(\delta(q,a))$

## E-NFA to NFA

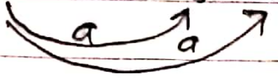
$(Q, \Sigma, \delta, q_0, F) \rightarrow$ E-NFA

$(Q, \Sigma, \delta^*, q_0, F) \rightarrow$ NFA

$\delta^*(q,a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q),a))$

{Initial state→Same as of NFA

$\delta^*(q_0,a) = \{q_3, q_4\}$ }



~~R.E~~

## Conversion of R.E Directly into DFA

- Syntax tree based reduction to DFA / using followposition base reduction

**Important states:**

The states in E-NFA is an important state if it has no null transition.

**Augmented R.E:**

E-NFA created from RE has exactly one accepting state and accepting state is not important state since there is no transition so by adding special symbol # on the RE at the rightmost position, we can make the accepting state as an important state that has transition on #.

The RE (r)# is called the augmented regular expression of the regular expression r.

## Procedure:

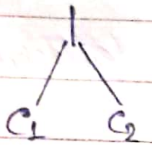
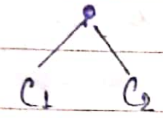
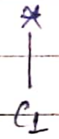
1. Augment the given regular expression by concatenating it with special symbol # i.e. r → (r)#
2. Create the syntax tree of this augmented regular expression. In this tree, all operators will be inner nodes and all the alphabet, symbols including # will be leaves.
3. Numbered each leaves.
4. Traverse the tree to construct nullable, firstpos, lastpos and followpos.
5. Finally construct the DFA from the followpos.

To evaluate followpos, we need three functions to defined the nodes of the syntax tree.

- firstpos(n): The set of the position of the first symbol of strings generated by the sub-expression rooted by n.
- lastpos(n): The set of the position of the last symbol of strings generated by the sub-expression rooted by n.
- nullable(n): true if the empty string is a member of strings generated by the sub-expression rooted by n, false otherwise.

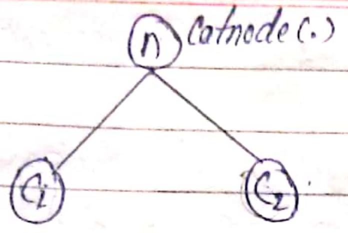
Rules for creating nullable, firstpos & lastpos

| node n | nullable(n) | firstpos(n) | lastpos(n) |
|---|---|---|---|
| 1. leaf labeled $\epsilon$ | true | $\phi$ | $\phi$ |
| 2. leaf labeled with position $i$ | False | $\{i\}$ | $\{i\}$ |

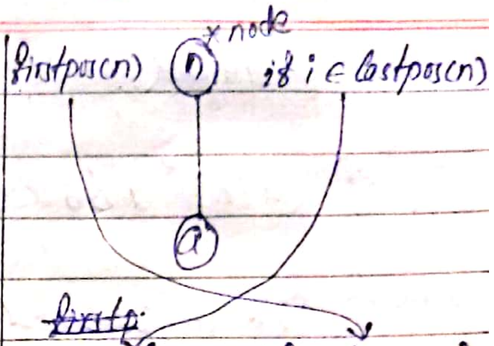
| | n | nullable(n) | firstpos(n) | lastpos(n) |
|---|---|---|---|---|
| 3. |  | nullable(C₁)<br>OR<br>nullable(C₂) | firstpos(C₁) U firstpos(C₂) | lastpos(C₁) U<br>lastpos(C₂) |
| 4. |  | nullable(C₁)<br>AND<br>nullable(C₂) | if (nullable(C₁)=TRUE)<br>firstpos(C₁) U firstpos(C₂)<br>else<br>firstpos(C₁) | if (nullable(C₂)<br>= TRUE)<br>lastpos(C₁) U<br>laspos(C₂)<br>else<br>lastpos(C₂) |
| 5. |  | true | firstpos(C₁) | lastpos(C₁) |

Computation of followpos (Algorithm)

```
for each node n in the tree do
    if n is a cat-node with left child C1 and right child C2 then
        for each i in lastpos(C1) do
            followpos(i) = followpos(i) U firstpos(C2)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) = followpos(i) U firstpos(n)
        end do
    end if
end do
```



if $i \in lastpos(c_1)$ $\longrightarrow$ $followpos(i) = \{ firstpos(c_2) \}$



$followpos(i) = \{ firstpos(n) \}$

## Algorithm to create DFA from RE

```
1. Create a syntax tree of (r)#
2. Calculate the functions: nullable, firstpos, lastpos & followpos
3. Put firstpos(root) into the state of DFA as an unmarked state.
4. While (there is unmarked state s in the states of DFA) do
     - Mark s
     - For each input symbol a ∈ Σ do
         • let $s_1, \ldots, s_n$ are positions in s and symbols in those
           position is a.
         • $s' \leftarrow followpos(s_1) \cup \ldots \cup followpos(s_n)$
         • $move(s, a) \leftarrow s'$
         • if (s' is not empty and not in the states of DFA)
             put s' into the states of DFA as an unmarked state.
```

→ The start state of DFA is firstpos(root)

→ The accepting states of DFA are all states containing the position of #.

Date ______
Page ______

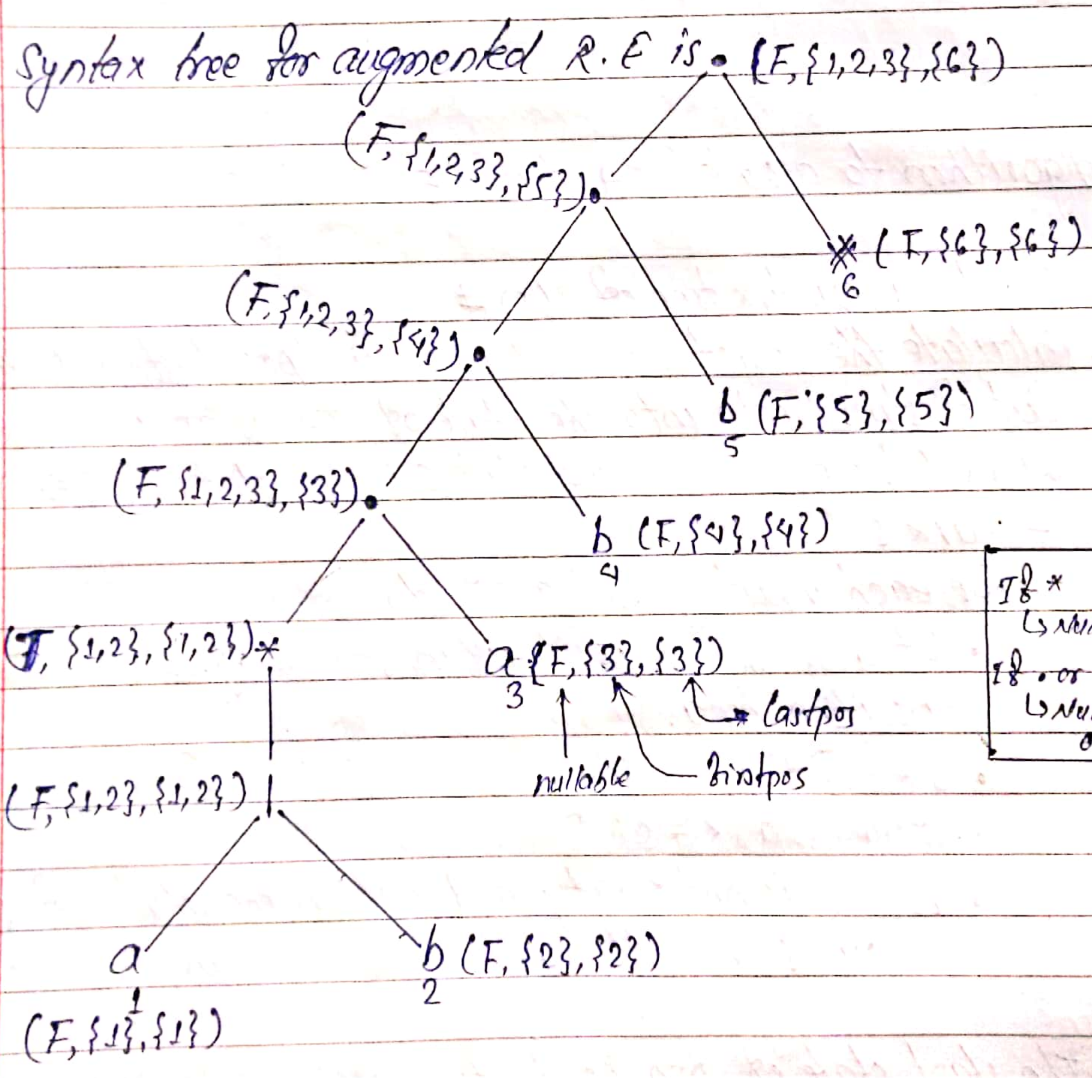
## Examples:

① $(a|b)^*abb$

The augmented regular expression of given regular expression is

$$\underset{1}{(a}|\underset{2}{b)}^*\underset{3}{a}\underset{4}{b}\underset{5}{b}\underset{6}{\#}$$

Syntax tree for augmented R.E is



If *
↳ Nullable is true
If • or |
↳ Nullable depends on its child

## Calculating followpos:

**For (*) Node:**

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

**For (.) Node:**

followpos (1) = {3}
followpos (2) = {3}
followpos (3} = {4}
followpos {4} = {5}
followpos {5} = {6}

Finally,

followpos(1) = {1, 2, 3}
followpos(2) = {1, 2, 3}
followpos (3) = {4}
followpos (4) = {5}
followpos (5} = {6}.
followpos(6) = { }

Now,

1) start state of DFA = firstpos (root) = {1, 2, 3} = $S_0$

Use followpos of symbol representing position in R.E to obtain the next states of DFA.

2) Here 1 and 3 represent 'a'
2 represent 'b'
followpos ({1,3}) = {1,2,3,4} = $S_1$
$\delta(S_0, a) = S_1$
followpos (2) = {1,2,3} = $S_0$
$\delta(S_0, b) = S_0$

3) From $S_1$ = {1, 2, 3, 4}
1, 3 → 'a'
2, 4 → 'b'
followpos (1,3) = {1,2,3,4} = $S_1$
$\delta(S_1, a) = S_1$
followpos (2,4) = {1,2,3,5} = $S_2$
$\delta(S_1, b) = S_2$

4) From $S_2$ = {1, 2, 3, 5}
1, 3 → 'a'
2, 5 → 'b'

followpos (1,3) = {1, 2, 3, 4} = $S_1$

$\delta(S_2, a) = S_1$

followpos (2,5) = {1, 2, 3, 6} = $S_3$

$\delta(S_2, b) = S_3$

5) From $S_3$ = {1, 2, 3, 6}

1, 3 → 'a'

2 → 'b'

6 → '#'

followpos (1,3) = {1, 2, 3, 4} = $S_1$

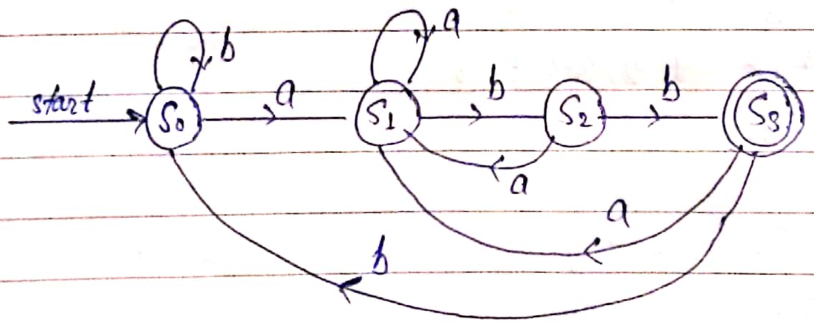
$\delta(S_3, a) = S_1$

followpos(2) = {1, 2, 3} = $S_0$

$\delta(S_3, b) = S_0$

6) Final state = {$S_3$}

~~Draw~~ the DFA is:



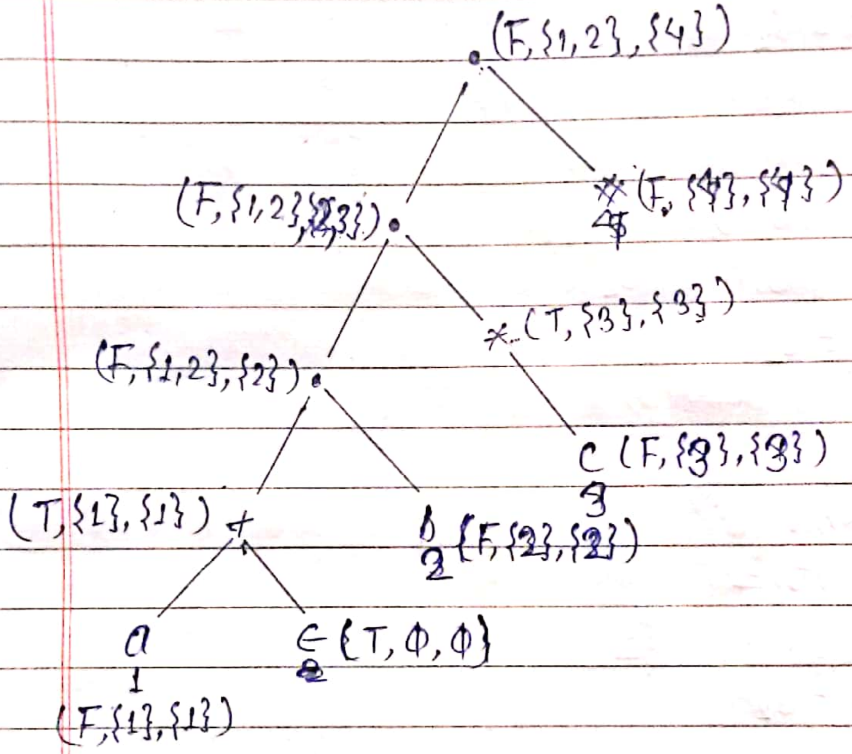
---

(2) $(a+e)bc^*$

The augmented R.E is

$\underset{1}{(a+e)}\,\underset{2}{b}\,\underset{3}{c^*}\,\underset{4}{\#}$

Syntax tree for augmented R.E is



Calculating followpos:

followpos (1) = { 2 }
followpos (2) = { 3,4 }
followpos (3) = { 3, 4 }
followpos (4) = { }

Now,
start state of DFA = firstpos (root) = {1, 2} = $S_0$

Use followpos of symbol representing position in R.E to obtain the next state of DFA.

Here 1 represents 'a'
2 represents 'b'

followpos(1) = {2} = $S_1$

$\delta(S_0, a) = S_1$

followpos(2) = {3,4} = $S_2$

$\delta(S_0, b) = S_2$

From $s_1$ = {2}

Here 2 represents 'b'

followpos(2) = {3,4} = $S_2$

$\delta(S_1, b) = S_2$

From $s_2$ = {3,4}

Here 3 represents 'c'

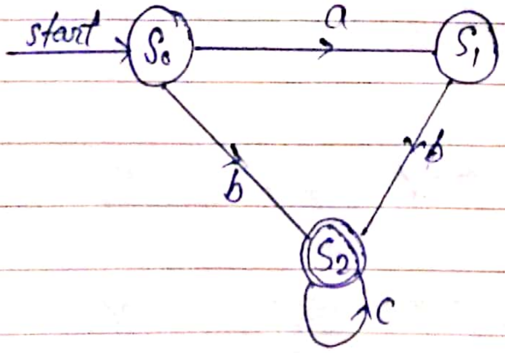
4 represents '#'

followpos(3) = {3,4} = $S_2$

$\delta(S_2, c) = S_2$

Accepting state = {$S_2$}

Now the DFA is



H.W.

$(a+\epsilon)^* b^* a$

$(a+\epsilon)^* ba$

$(a+b)ab+a^*$

## State Minimization in DFA

→ DFA minimization refers to the task of transforming a given DFA into an equivalent DFA which has minimum number of states.

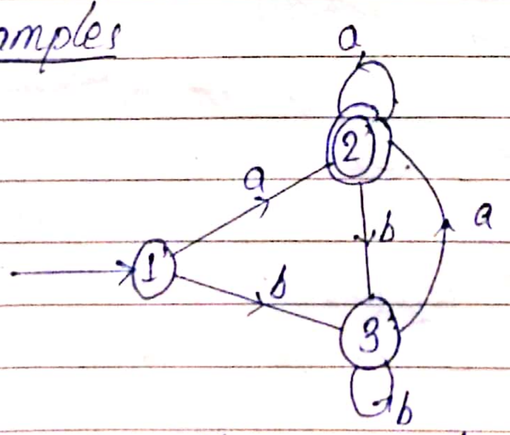
Procedure:

1. Partition the set of states into two groups: Set of accepting states and set of non-accepting states.
2. For each new group G
   - Partition G into subgroups such that states $s_1$ and $s_2$ are in the same group iff for all input symbol a, states $s_1$ and $s_2$ have transition to states in the same group.
3. Process until all the partition contains equivalent states only or have single state.

- start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Examples

①



Partition the set of states as

$G_1 = \{2\}$

$G_2 = \{1, 3\}$

For $G_2$:

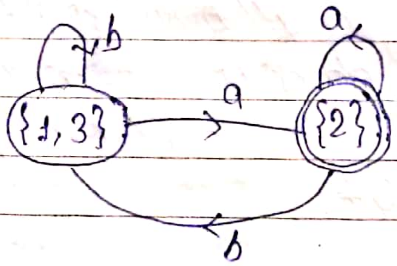
| | 1 | 3 |
|---|---|---|
| a | $G_1$ | $G_1$ |
| b | $G_2$ | $G_2$ |

$G_2$ Cannot divided further.

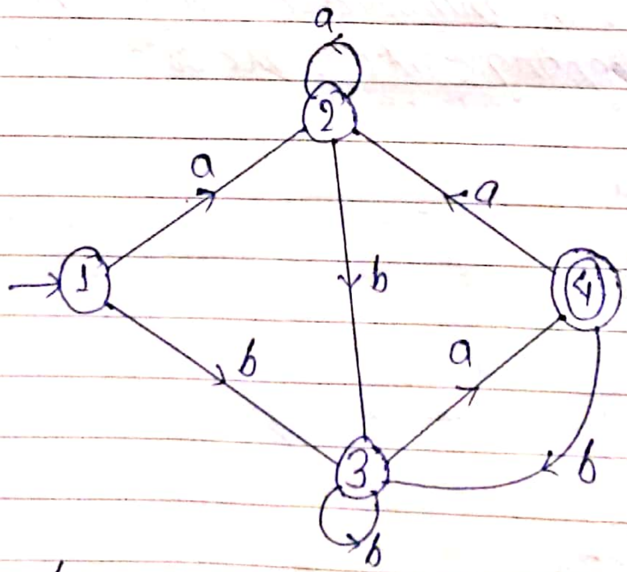
Here,

Equivalent states are 1 & 3.

so, the minimized ~~stak~~ DFA is



②



Accepting & non-accepting states are grouped as

$G_1 = \{4\}$

$G_2 = \{1,2,3\}$

For $G_2$ :

|  | 1 | 2 | 3 |
|---|---|---|---|
| a | $G_2$ | $G_2$ | $G_1$ |
| b | $G_2$ | $G_2$ | $G_2$ |

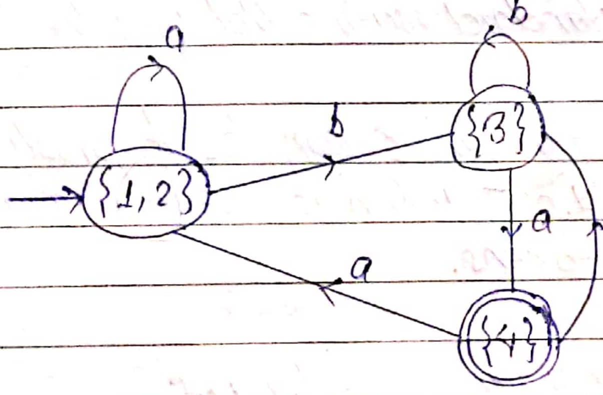
$G_2$ is further divided

$G_2 = \{1, 2\}$

$G_3 = \{3\}$

~~Also~~. Here, no more partitioning for $G_2$.

so the minimized DFA

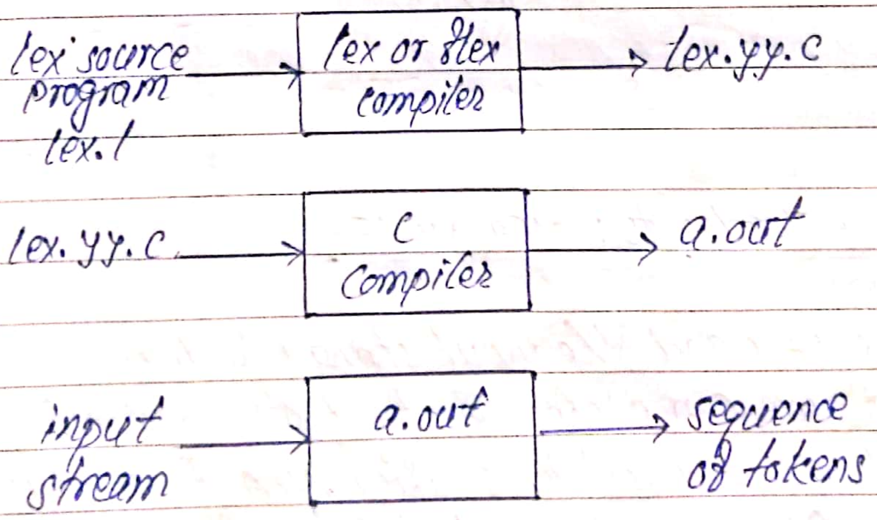


## Space Time Tradeoffs : NFA vs DFA

- Given the RE r and the input string s to determine whether s is is in L(r) we can either Construct NFA and test or we can Construct DFA and test for s after NFA is constructed from r.
- ϵ-NFA (for NFA only constant time differs)
  - space complexity : $O(|r|)$
  - Time complexity : $O(|r| * |s|)$
- DFA
  - Space complexity : $O(2^{|r|})$ [ϵ-NFA Construction & then subset Construction]
  - Time complexity : $O(|s|)$
- If we can create DFA from RE by avoiding transition table, then we can improve the performance.

## Creating a lexical Analyzer Generator (lex) / Flex

- Systematically translate regular definitions into C source code for efficient scanning.
- Flex is a latter version of lex.
- Generated code from lex is easy to integrate in C applications by integrating.
- Firstly, specification of a lexical analyzer is prepared by creating a program lex.l in lex/Flex.
- lex.l is run into lex compiler to produce a C program lex.yy.c.
- lex.yy.c consists of the tabular representation of lex.l, together with a standard routine that uses the table to recognize lexeme.
- lex.yy.c is run through C compiler to produce the object program a.out which is a lexical analyzer that input into tokens.



→ A lex/flex specification consists of three parts:

1. regular definitions, C declarations in %{ %}

%%

a? → a|ε (zero or one occurrences)

2. translation rules

%%

3. User-defined auxiliary procedures

The translation rules are of the form:

Pattern → $P_1$ {$action_1$}

$P_2$ {$action_2$}

: :

$P_n$ {$action_n$}

→ lex/Flex regular definitions are of the form:

name definition

e.g.

| | |
|---|---|
| Digit | [0-9] |
| letter | [A-Za-z] |
| ID | ({letter}}({letter}|{digit})* or [a-z][a-z0-9]* |

→ Action in lex are of the form

Pattern action

where pattern must be unintended and action must be on the same line.

| Pattern | Action |
|---|---|
| if/then/else/for/while/do | {printf("A keyword:%s\n",yytext);} |

Global function & variables

- yylex() : is the scanner function that can be invoked by the Parser
- yytext : extern char*yytext ; is a global char pointer holding the currently matched lexeme.
- yyleng : extern int yyleng; is a global int that contains the length of the currently matched lexeme.

## lex Examples

① 
```
%{
 #include<stdio.h>
 %}
 %%
[0-9]+   {printf {"%s\n", yytext);}
 .|\n    {   }
 %%
 main()
 {  yylex();
 }
```

Translation rules → [0-9]+

Contains the matching lexeme → yytext

Invokes the lexical analyzer → yylex();

② 
```
%{
 #include <stdio.h>
 #include <math.h>
%}
DIGIT   [0-9]
 ID     [a-zA-Z_][a-zA-Z0-9_]*
 OP     "+"|"-"|"*"/"/"
%%
{DIGIT}+   {Printf("An integer%s(%d)",yytext, atoi(yytext));}
{DIGIT}+"."{DIGIT}+ }printf("A float no:%s(%f)", yytext,atof(yytext}
 {ID}       {Printf ("An identifier:%s", yytext);}
[\t\n]+     {   }
  .         { printf ("An unrecognized char:%s", yytext);}
%%
int main (int argc, char ** argv)
{
  ++argv, --argc;
```

ascii to integer → atoi

In lex
a+b is denoted as [ab]
a+b+c [abc]
(a+b+c)* [abc]*

Date ______
Page ______

```
if (argc>0)
    yyin = fopen (argv[1], "r");
else
    yyin = stdin;
yylex();
}
```

/* lex program that recognizes the identifier of c languages */

```
%{
 #include<stdio.h>
%}
%%
^[a-z A-Z _][a-z A-Z 0-9]*   printf("Valid Identifier");
^[^a-z A-Z _]  printf("Invalid Identifier");
 .;
%%
main()
{
    yylex();
}
```