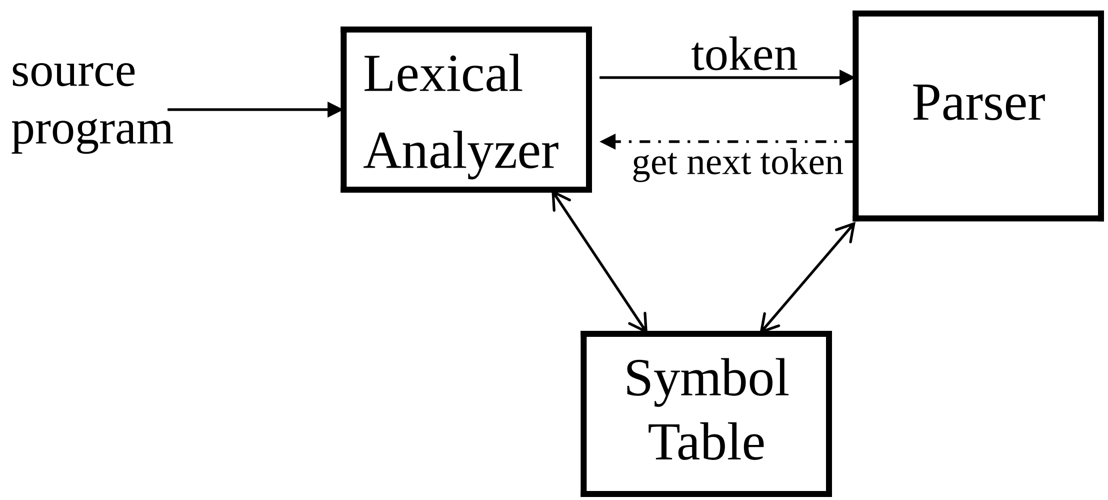
Lexical Analysis
1st Phase of Compiler Construction

Section 1.1
Lexical Analysis- Introduction

1.1 Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character to produce **tokens**.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Roles of the Lexical analyzer

Lexical analyzer performs following tasks:

- Helps to identify token in the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

Tokens, Lexemes and Patterns

- **Token:** Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are:

  Identifiers 2) keywords 3) operators 4) special symbols 5)constants

- **Lexeme:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

- **Pattern:** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Tokens, Lexemes and Patterns

Token	Lexeme (element of a kind)	Pattern
ID	x y n_0	letter followed by letters and digits
NUM	-123 1.456e-^{5}	any numeric constant
IF	if	if
LPAREN	(	(
LITERAL	``Hello"	any string of characters (except ``) between `` and ``

- ***Regular expressions*** are widely used to specify patterns.

Example

```
#include <stdio.h>
  int maximum(int x, int y){
        // This will compare 2 numbers
```

Tokens Generated

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator

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

Non-Tokens

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. s ε = s / ε s = s
  - $s^n = s\ s\ s \ldots s$ (n times) $s^0 = \varepsilon$

Operations on Languages

- **Concatenation:**
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- **Union**
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- **Exponentiation:**
  - $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$
- **Kleene Closure**
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- **Positive Closure**
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d)
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)(r_2)$	$L(r_1) L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$
- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon, 0,00,000,0000,....\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter $\rightarrow$ A | B | ... | Z | a | b | ... | z

  digit $\rightarrow$ 0 | 1 | ... | 9

  id $\rightarrow$ letter (letter | digit)*

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9))*

- Ex: Unsigned numbers in Pascal

  digit $\rightarrow$ 0 | 1 | ... | 9

  digits $\rightarrow$ digit$^+$

  opt-fraction $\rightarrow$ (. digits) ?

  opt-exponent $\rightarrow$ (E (+|-)? digits) ?

  unsigned-num $\rightarrow$ digits opt-fraction opt-exponent

Notational Shorthand

- The following shorthand are often used:

$$r^{+} = rr^{*}$$
$$r? = r \mid \varepsilon$$
$$[a\text{-}z] = a \mid b \mid c \mid \ldots \mid z$$

- Examples:

digit → [0-9]

digits → **digit**$^{+}$

optional_fraction → (. **digits**)?

optional_exponent → (E (+ | -)? **digit**$^{+}$)?

num → **digits optional_fraction optional_exponent**

Recognition of Tokens

- e.g.

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ε

expr → *term* **relop** *term*

| *term*

term → **id**

| **num**

Regular Definitions

if → if

then → then

else → else

relop → < | <= | = | <> | > | >=

id → **letter (letter | digit)***

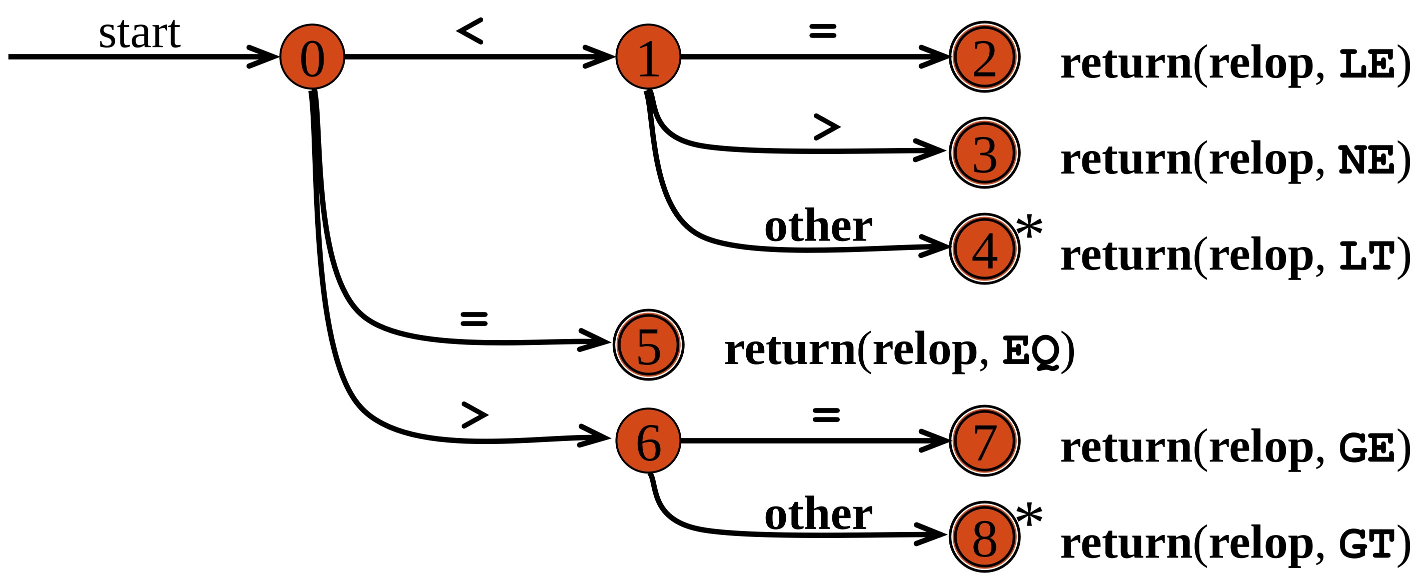
num →**digits optional_fraction optional_exponent**

Assumptions

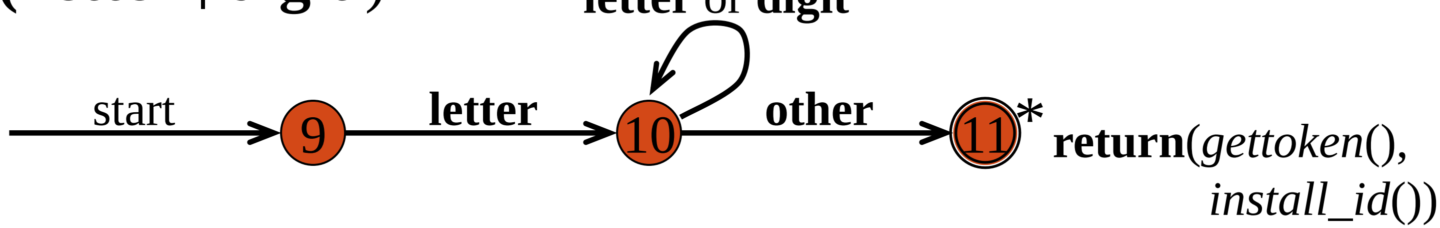
delim → **blank | tab | newline**

Transition Diagrams

relop → < | <= | <> | > | >= | =



id → letter (letter | digit)*



Transition Diagrams: Code

- ```
  token nexttoken()
  { while (1) {
      switch (state) {
      case 0: c = nextchar();
         if (c==blank || c==tab || c==newline) {
           state = 0;
           lexeme_beginning++;
         }
         else if (c=='<') state = 1;
  ```
- ```
         else if (c=='=') state = 5;
         else if (c=='>') state = 6;
         else state = fail();
         break;
       case 1:
         …
       case 9: c = nextchar();
         if (isletter(c)) state = 10;
         else state = fail();
         break;
       case 10: c = nextchar();
         if (isletter(c)) state = 10;
         else if (isdigit(c)) state = 10;
         else state = 11;
         break;
       …
  ```

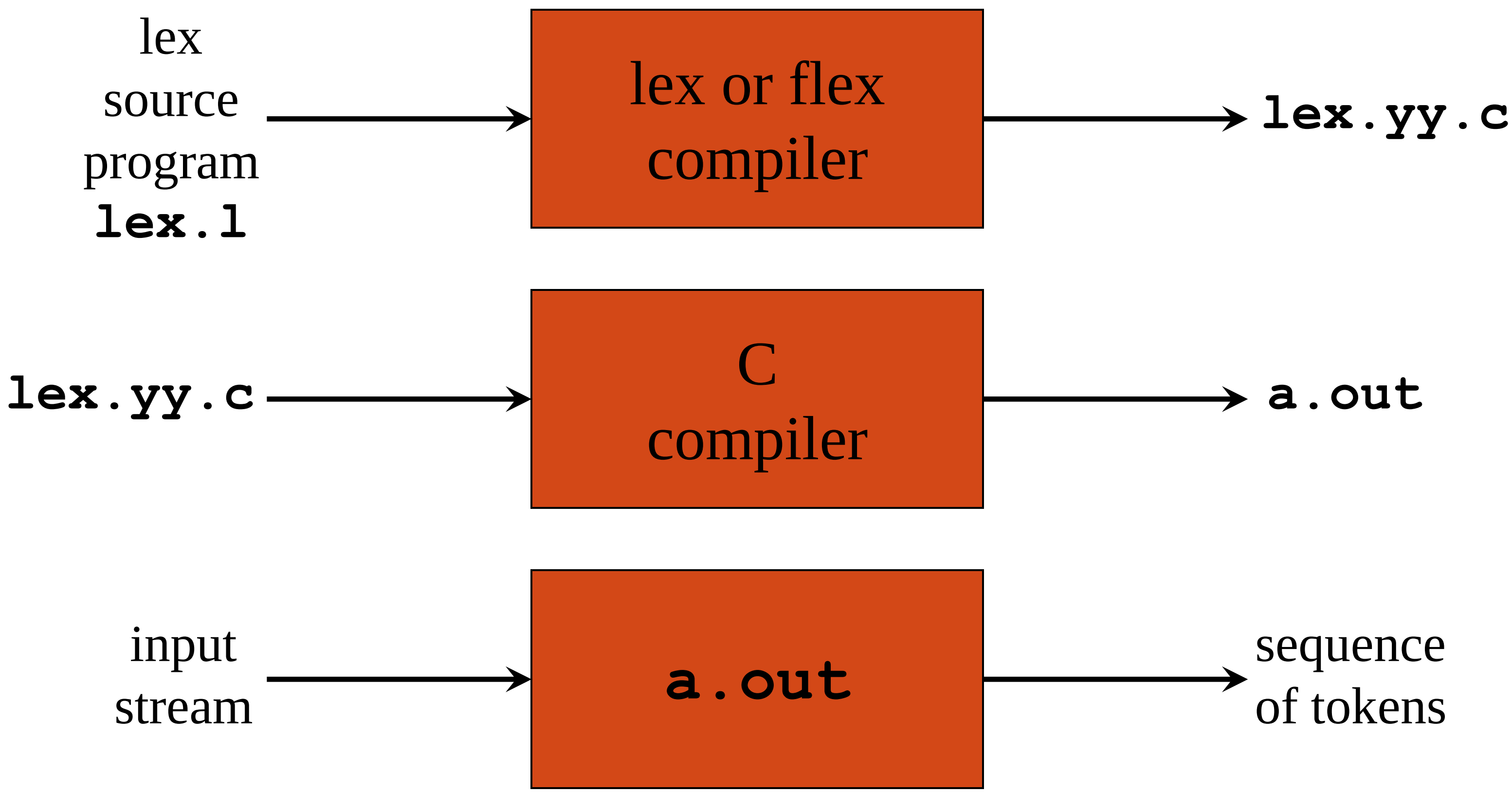
Decides the next start state to check

```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex and Flex



Lex Specification

- A *lex specification* consists of three parts:

  regular definitions, C declarations in **%{ %}**

  %%

  translation rules

  %%

  user-defined auxiliary procedures

- The *translation rules* are of the form:

  p_1 { $action_1$ }

  p_2 { $action_2$ }

  …

  p_n { $action_n$ }

Regular Expressions in Lex

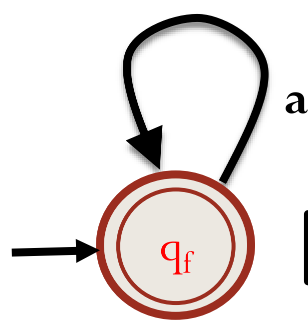
`x`	match the character `x`
`\.`	match the character `.`
"*string*"	match contents of string of characters
`.`	match any character except newline
`^`	match beginning of a line
`$`	match the end of a line
`[xyz]`	match one character `x`, `y`, or `z` (use `\` to escape `-`)
`[^xyz]`	match any character except `x`, `y`, and `z`
`[a-z]`	match one of `a` to `z`
r`*`	closure (match zero or more occurrences)
r`+`	positive closure (match one or more occurrences)
r`?`	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
r_1`|`r_2	match r_1 or r_2 (union)
`(` r `)`	grouping
r_1`\`r_2	match r_1 when followed by r_2
`{`d`}`	match the regular expression defined by d

Star operation (Kleene Closure)

$$a^* = \{a^0, a^1, a^2, a^3, a^4, \ldots a^\infty\} = \{\varepsilon, a, aa, aaa, aaaa, \ldots\ldots a^\infty\}$$

Important Characteristics

- **Value of * ranges from 0 to ∞ i.e. the elements of set a* will include $\{a^0, a^1, a^2, a^3, a^4, a^5 \ldots a^\infty\}$**
- **a^0 means zero number of a's and this is represented by ε.**
- *** is represented in finite automata by a loop on that particular state; if value of a is 3 i.e. a^3 loop iterates for 3 times.**
- **If value of a is 0 i.e. a^0 loop will not iterate at all.**



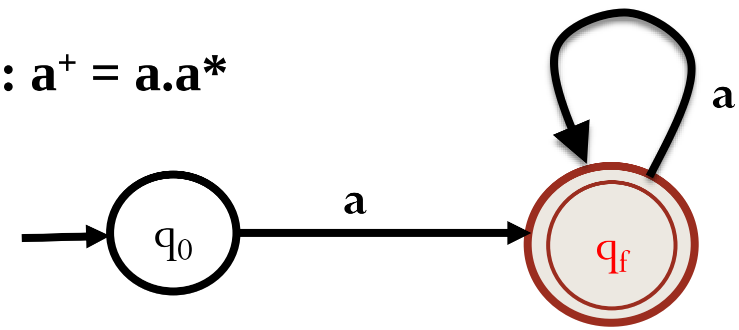
m/c for a*

Positive Closure

$$a^+ = \{a^1, a^2, a^3, a^4, \ldots, a^\infty\} = \{ a, aa, aaa, aaaa, \ldots\ a^\infty\}$$

Important Characteristics

- **value of + ranges from 1 to ∞ *i.e.* the elements of set a^+ will include $\{a^1, a^2, a^3, a^4, a^5 \ldots a^\infty\}$**
- **There is no a^0 move i.e. ε is not part of this set.**
- **Value of a will start from 1 *i.e.* at least one will come which can be followed by 0 or more 1's.**
- **Please remember: $a^+ = a.a^*$**



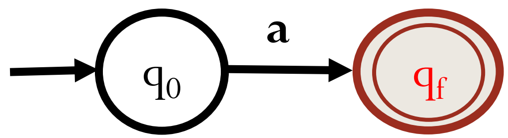
m/c for a^+

Concatenation Operation

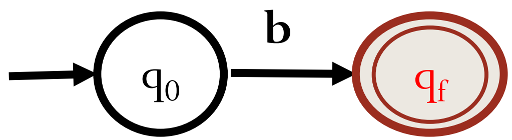
Concatenation means joining (a.b)

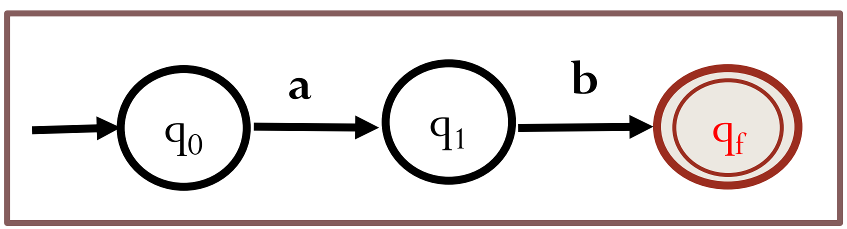
Important Note: a.b ≠ b.a *i.e.* order of join will change the design of automata

m/c for a

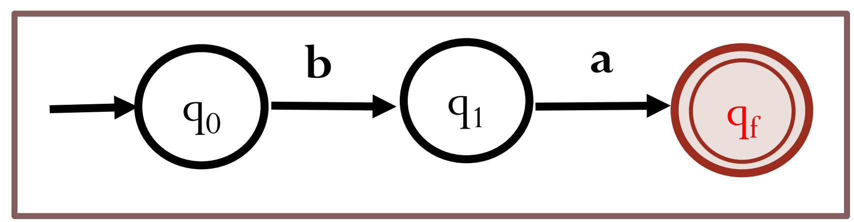


m/c for b





≠

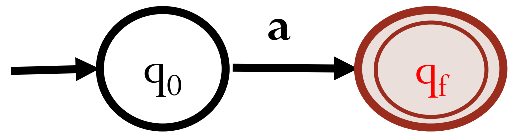


m/c for a.b

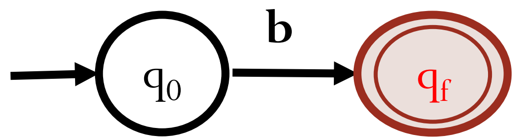
m/c for b.a

OR Operation

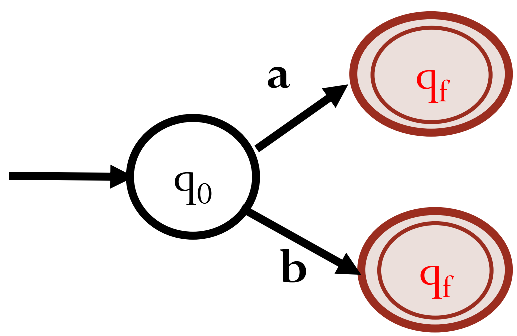
m/c for a



m/c for b



NFA for a+b (a/b)



m/c for a/b

Section 1.2
Introduction to Finite Automata

FINITE AUTOMATA

Automata means machine

Finite Automata consist of 5 tuples:

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

Q	A finite set of states
Σ	A finite set of input alphabet
δ	A transition function
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of F

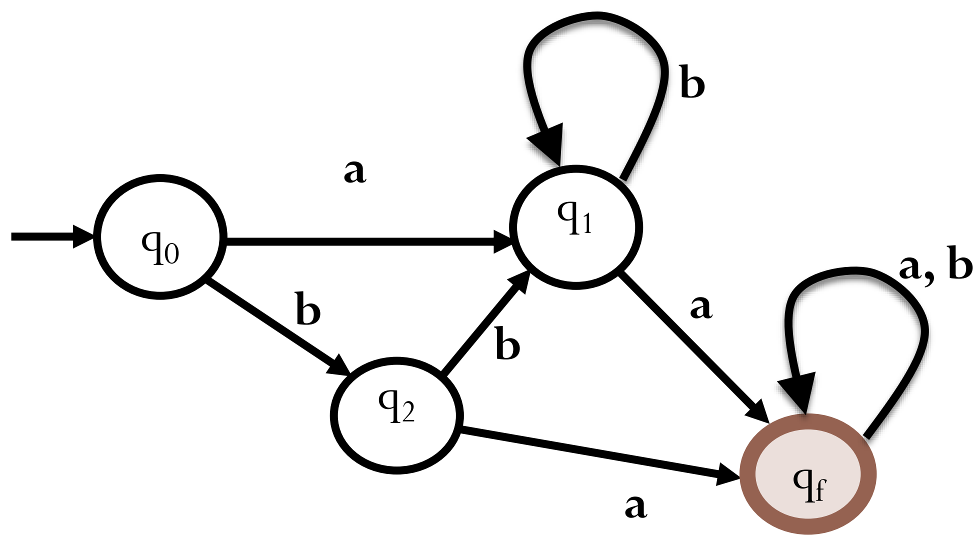
Types of Automata

There are two types of finite Automata:

- Deterministic Finite Automata (DFA)
- Non-deterministic finite Automata (NFA)

Deterministic Finite Automata

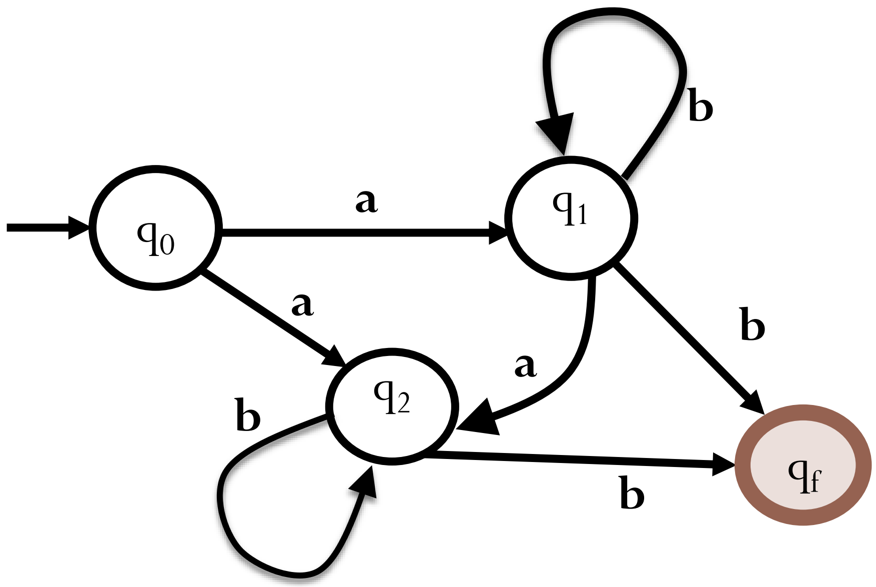
Deterministic Finite Automata is a Machine where corresponding to a every input of **Σ**, there can be only one output from every state.



Here Σ = { a, b} and at every state there is one O/P from 'a' and one O/P from 'b'. None of the states have more then one output corresponding to a or b.

Non-Deterministic Finite Automata

Non-Deterministic Finite Automata is a machine where corresponding to a single input of Σ (a,b), there can be more than one output from a particular state.



Here state q_0 has two moves from a, one to q_1 and other to q_2, like wise state q_2 has two moves on 'b' one self loop to q_1 and another to q_f

Types of NFA

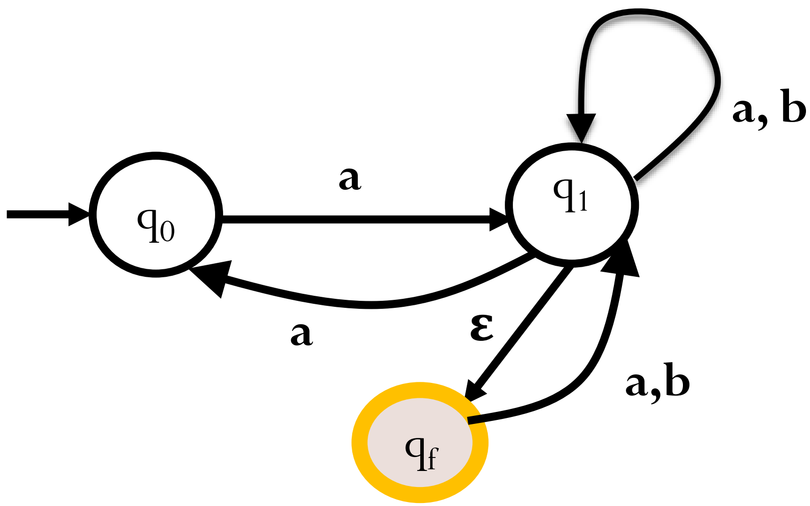
There are two type of NFA

i. NFA without **ε** -move

ii. NFA with **ε** -move

NFA with ε-move

Consider the following NFA, here corresponding q_1 there is an ε-move.



Difference between DFA and NFA

Deterministic Finite Automata	Non-Deterministic Finite Automata
• Deterministic Finite Automata is a Machine where corresponding to a every input of **Σ**, there can be only one output from every state. • DFA will not have ε-move	• Non-Deterministic Finite Automata is a machine where corresponding to a single input of Σ (a,b), there can be more than one output from a particular state. • NFA can have ε-move

Section 1.3
Thomson's Construction

Thompson's Construction

We have three operations on Regular Expressions:

i) **Star operation**

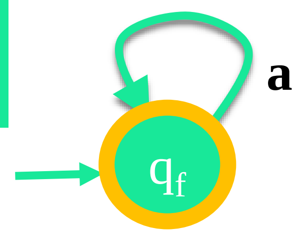
ii) **Concatenation**

iii) **OR operation**

For each operation we have defined rules to build a NFA with ε-move

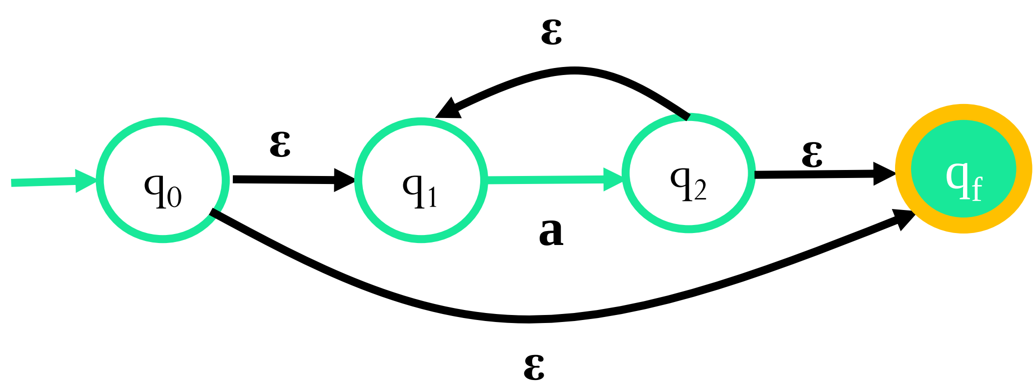
Thompson’s Construction for Star Operation

a* = {ε, a, aa, aaa, aaaa,…..}



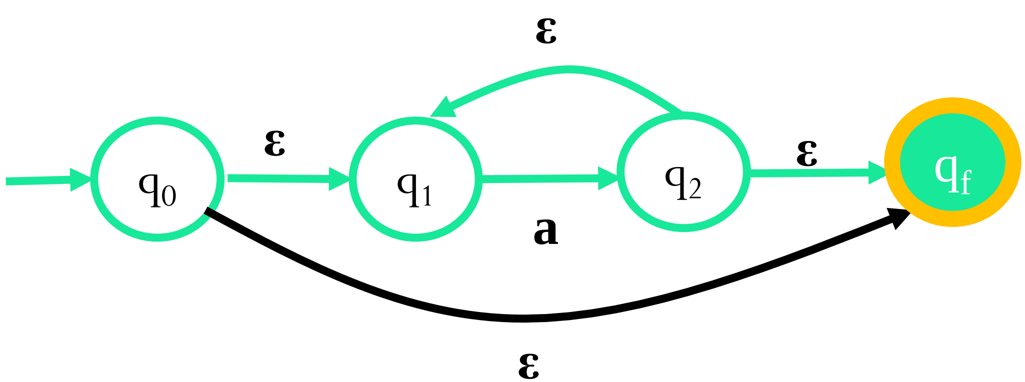
NFA for a*

NFA for a* using Thomson’s Construction:

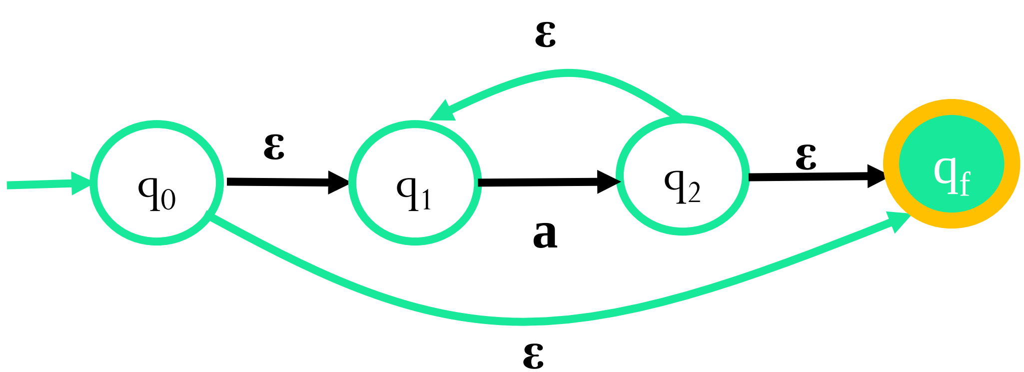


Thompson's Construction for Star Operation

NFA for a* using Thomson's Construction:



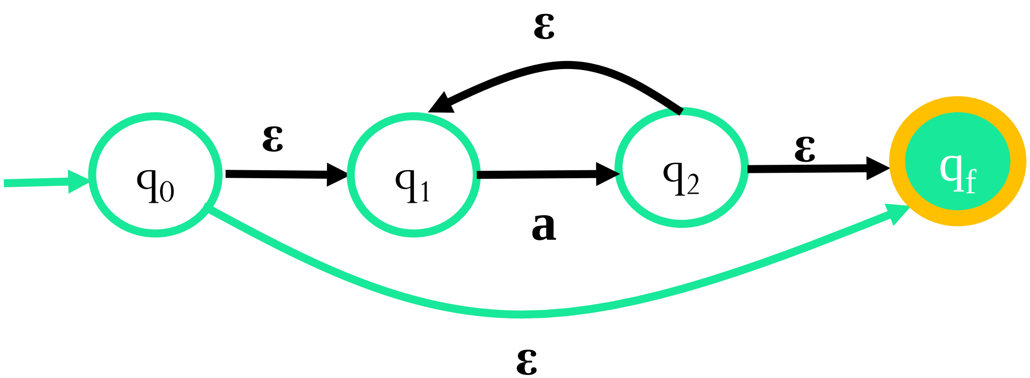
Only ε



Single a

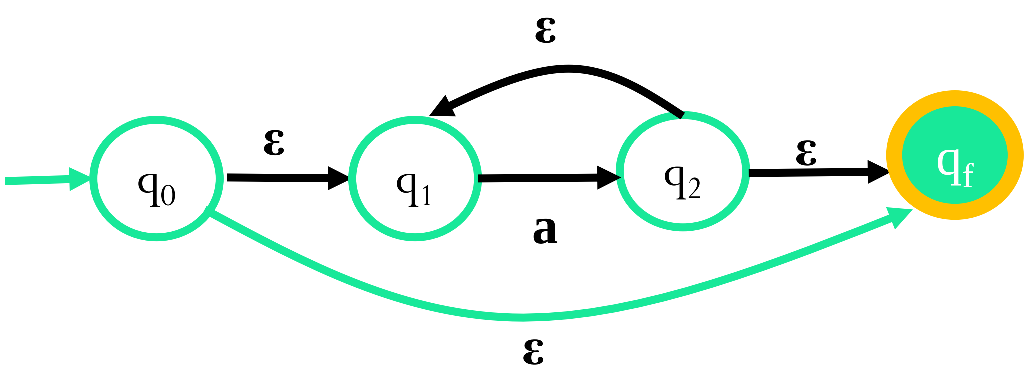
Thompson’s Construction for Star Operation

NFA for a* using Thomson’s Construction:



Two a’s

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_1 \rightarrow q_2 \rightarrow q_f$



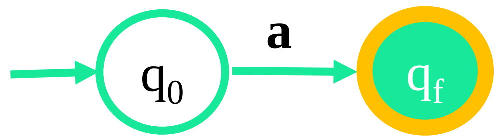
N number of a’s

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_1 \rightarrow q_f$

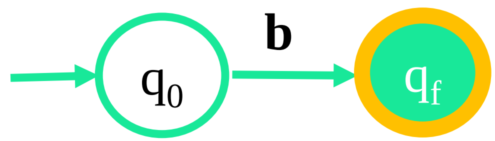
$q_1 \rightarrow q_2 \rightarrow q_1$ loops for N times where N varies from 2 to∞

Thompson's Construction for Concatenation Operation

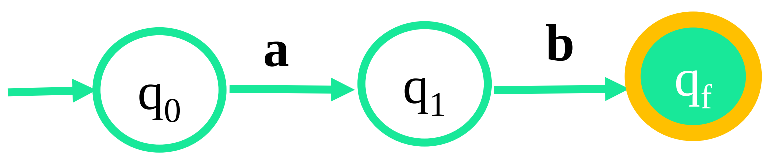
NFA for a



NFA for b

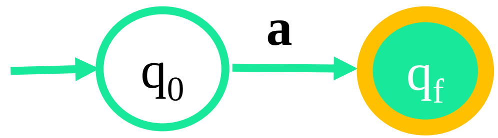


NFA for ab using Thomson's Construction

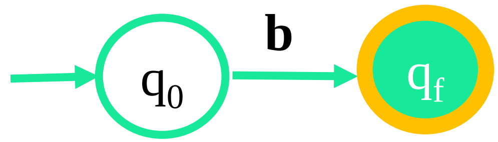


Thompson's Construction for OR Operation

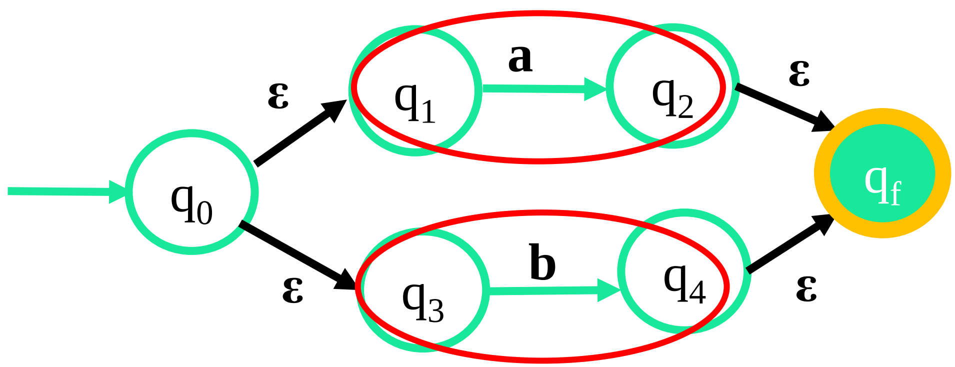
NFA for a



NFA for b



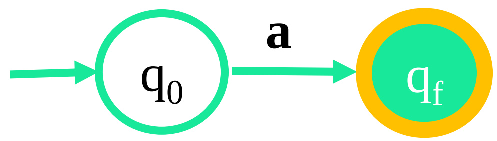
NFA for a+b (a/b) using Thomson's Construction



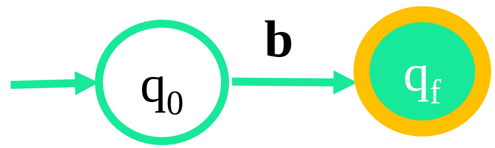
Thompson's construction for aa*b

Question 1

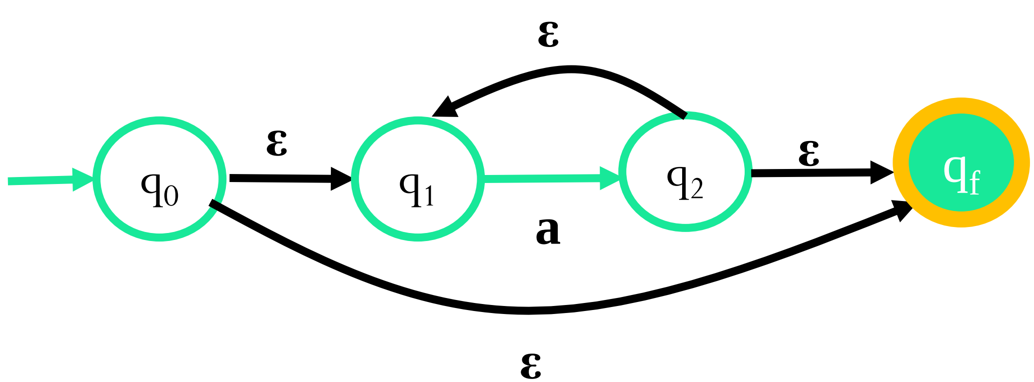
Thompson's for a:



Thompson's for b:



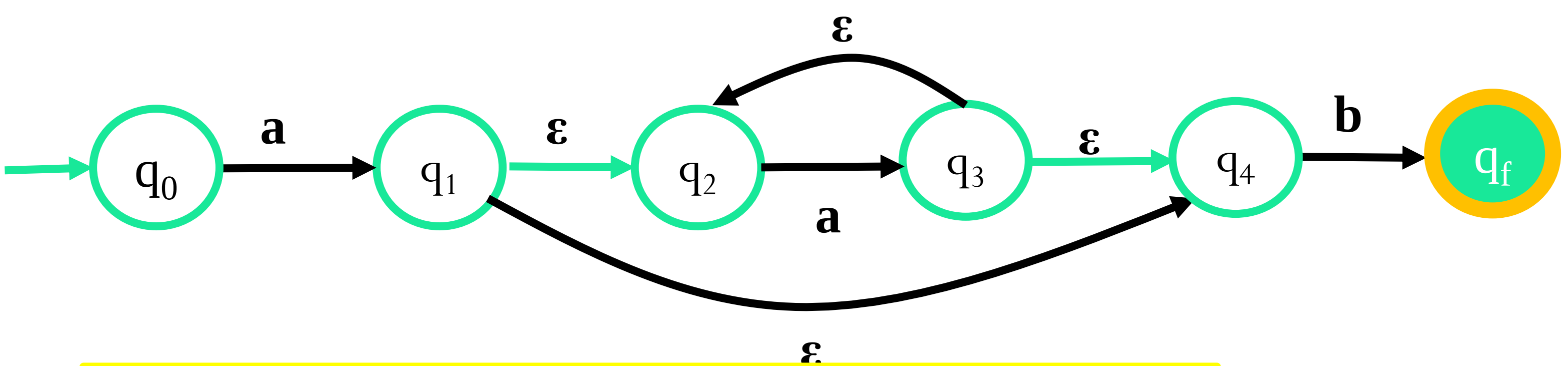
Thompson's for a*:



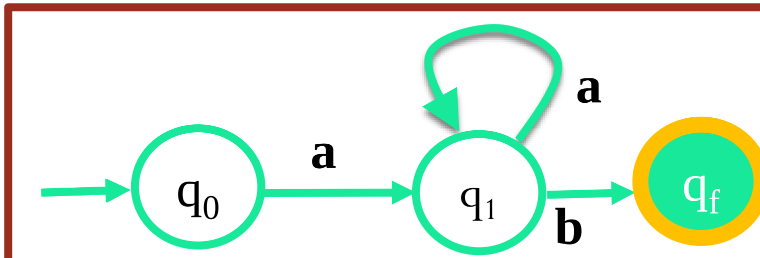
Thompson's construction for aa*b

Question 1

Thompson's Construction for aa*b:



NFA using Thompson's Construction

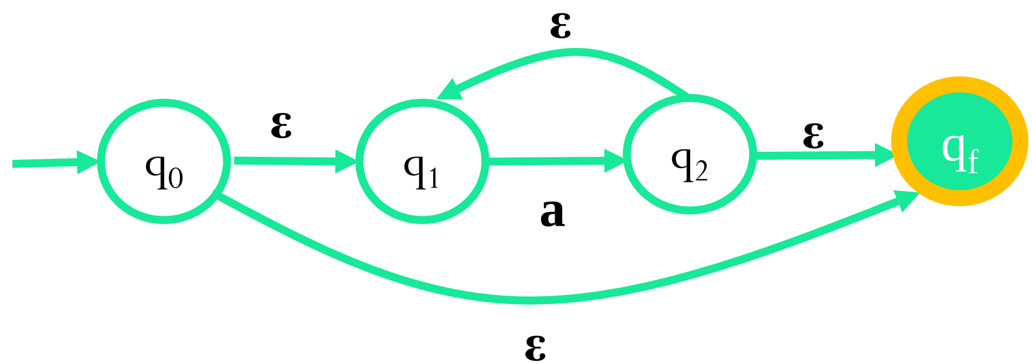


NFA without Thompson's

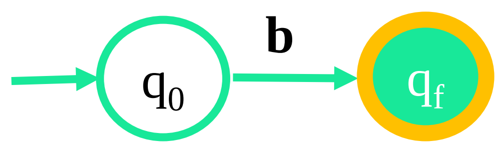
Question 2

Thompson’s construction for a*b(a/b)

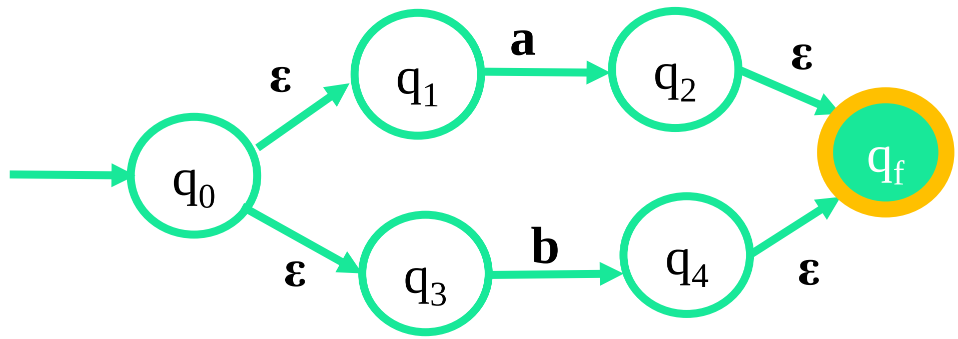
Thompson’s for a*:



Thompson’s for b:



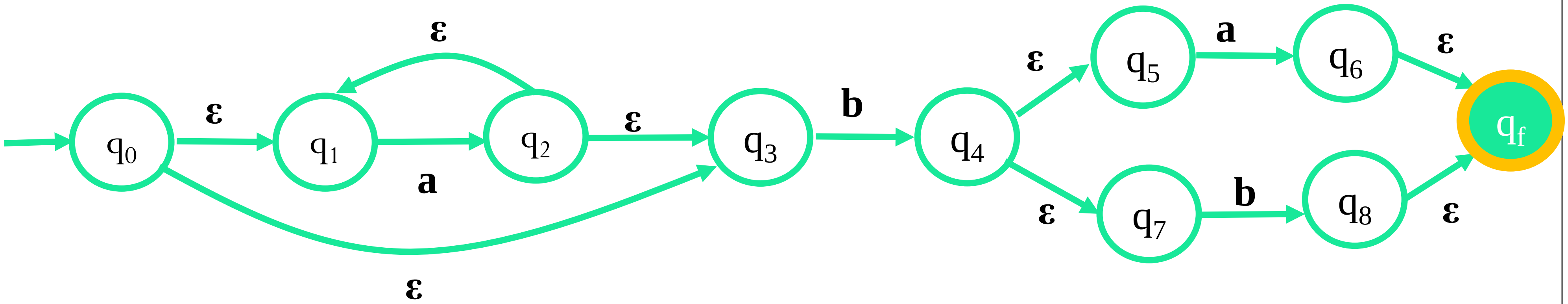
Thompson’s for a/b:

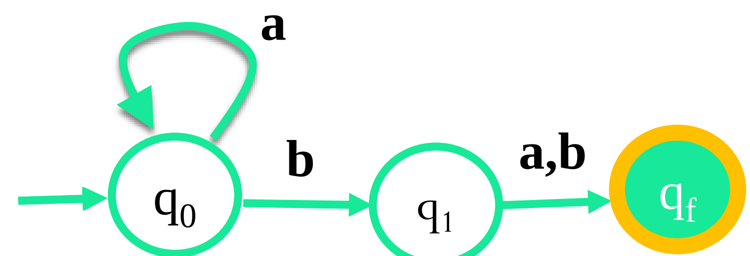


Thompson's construction for a*b(a/b)

Question 2

NFA using Thompson's Construction

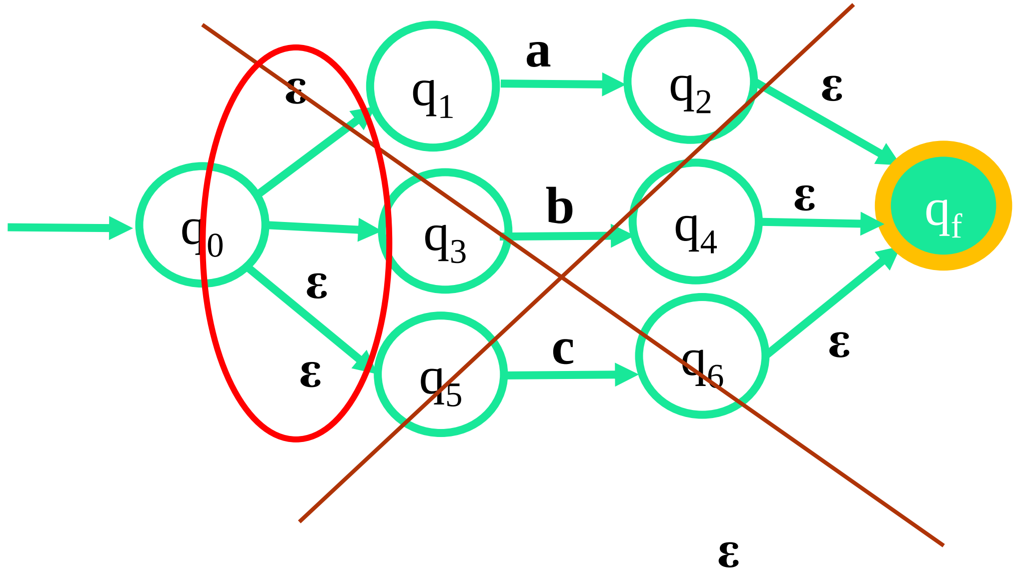




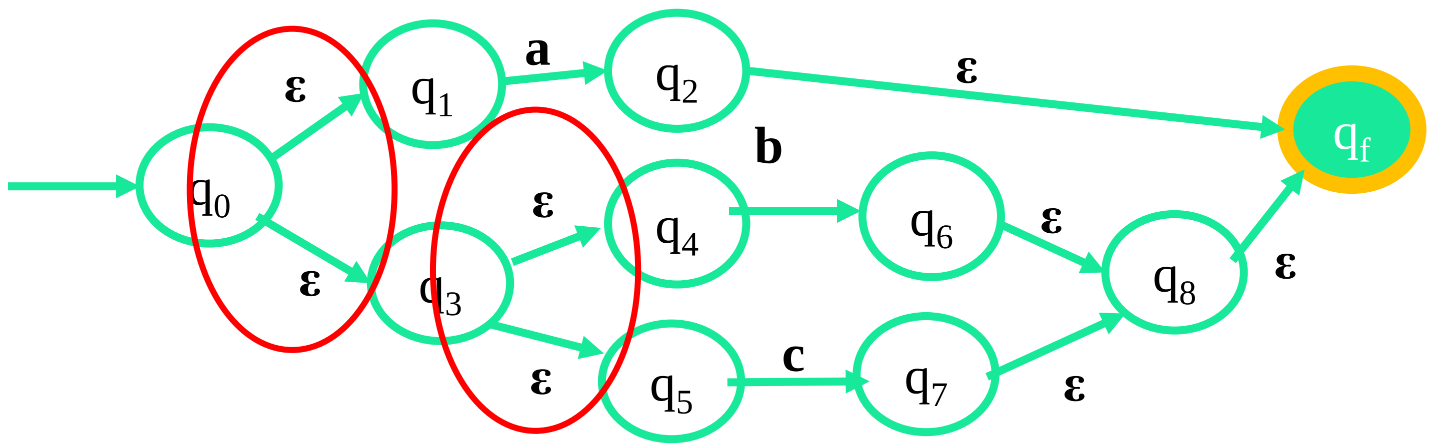
NFA without Thompson's

Thompson’s construction for (a/b/c)

Question 3



Three **ε** out moves moves from a state are not allowed

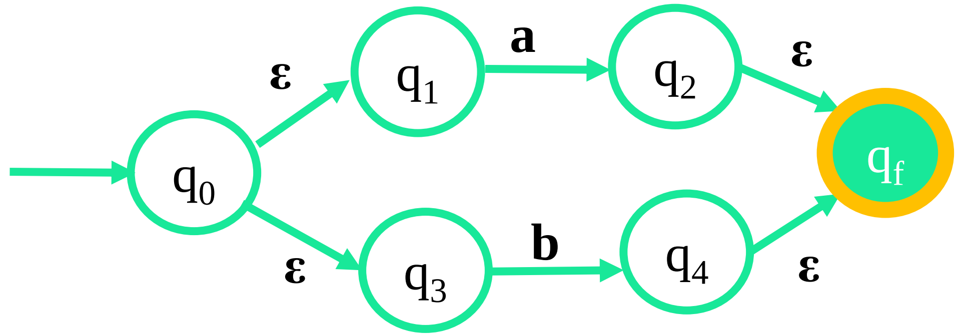


Final Output

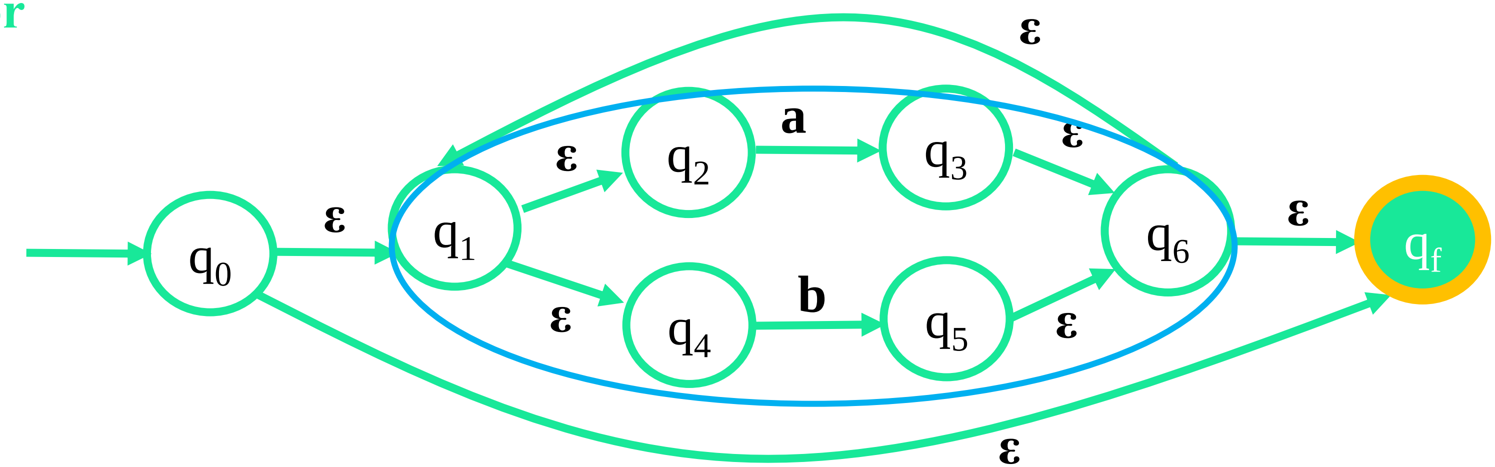
Thompson’s construction for ab(a/b)*

Question 4

Thompson’s for a/b:



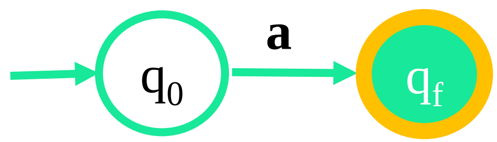
Thompson’s for (a/b)*:



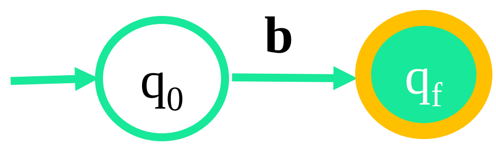
Thompson's construction for ab(a/b)*

Question 4

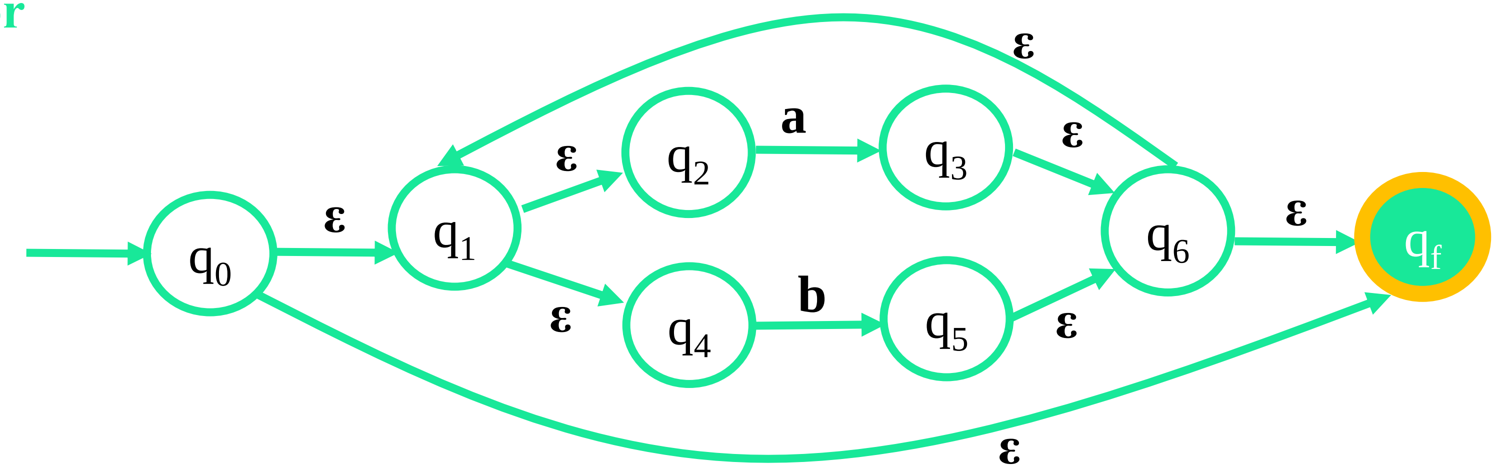
Thompson's for a:



Thompson's for b:

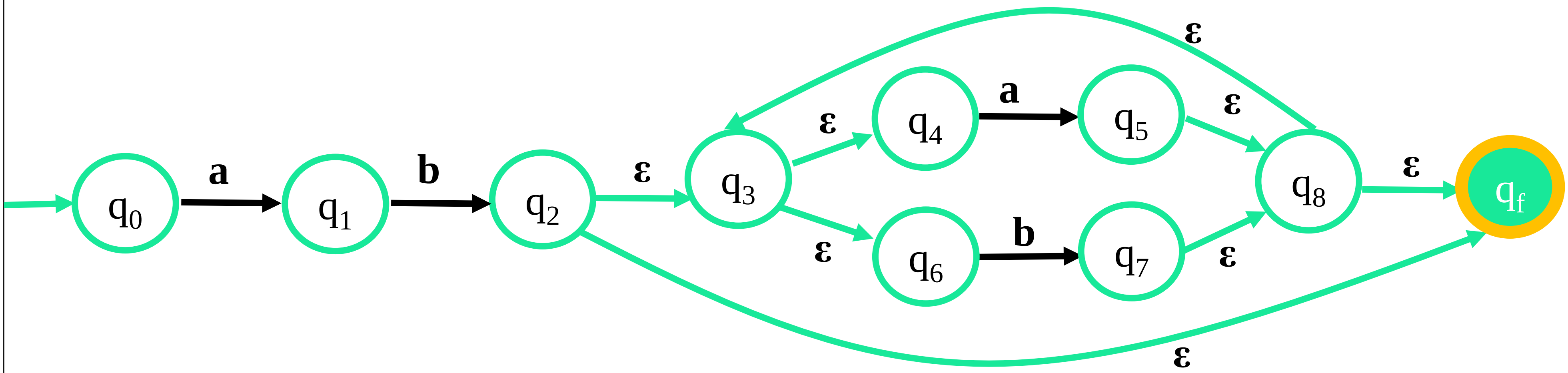


Thompson's for (a/b)*:

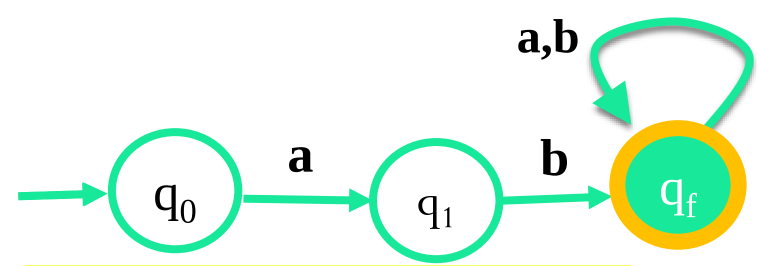


Give the Thompson’s construction for ab(a/b)*

Question 4



NFA using Thompson’s Construction



NFA without Thompson’s

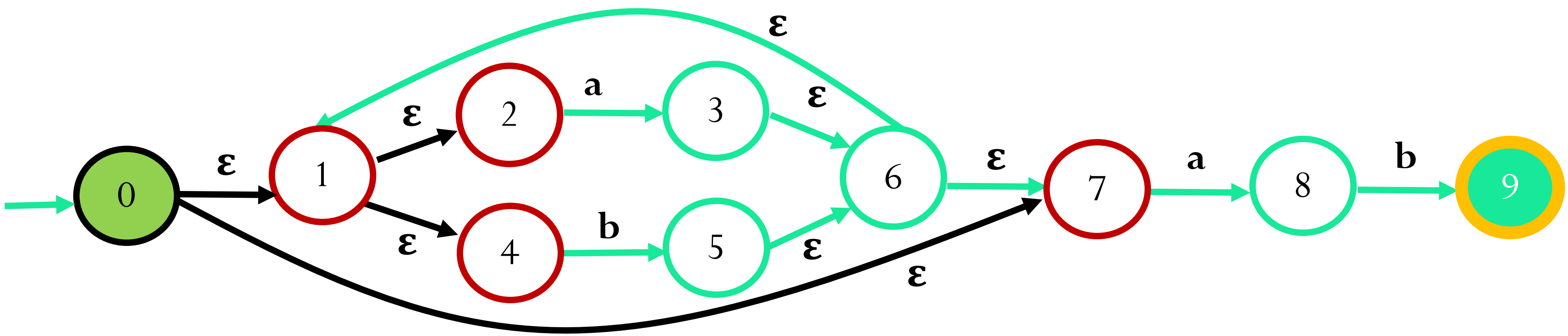
Section 1.4
Subset Construction

How to work with ε-Closure Function

Steps for ε-Closure function:

- First step is to take ε-Closure of the start state , for *e.g. i*f the start state is 0 so take ε-Closure(0).
- ε-Closure(n) will include set of all the states which can be traversed from state n without consuming any input *i.e.* through ε move only.
- Most Imp.- “ε-Closure of a state will include that state itself in the set”, *i.e.* ε-Closure(n) will include n in its set of states.

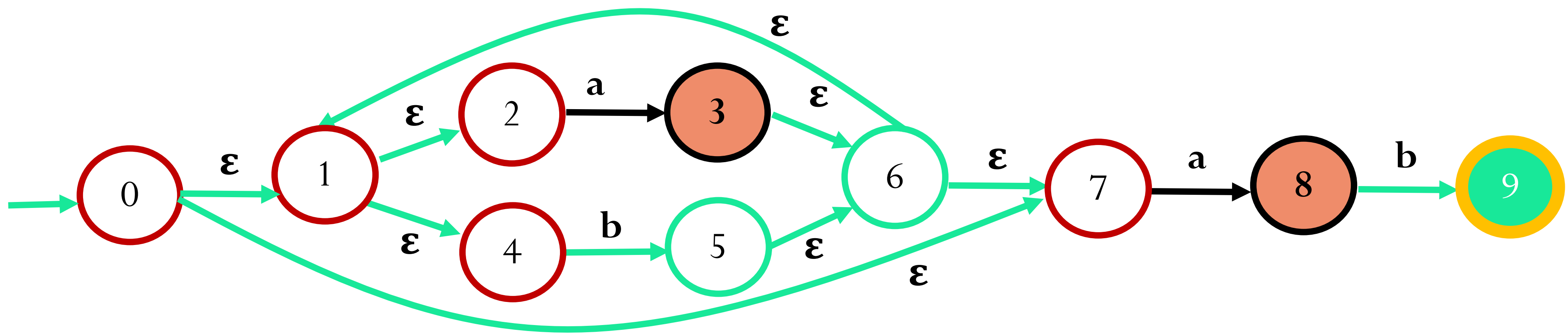
Subset Construction for (a/b)*ab



Start with the start state: state 0
ε-closure(0):{0,1,2,4,7} = A

State	a	b
A (0,1,2,4,7)		

Subset Construction for (a/b)*ab

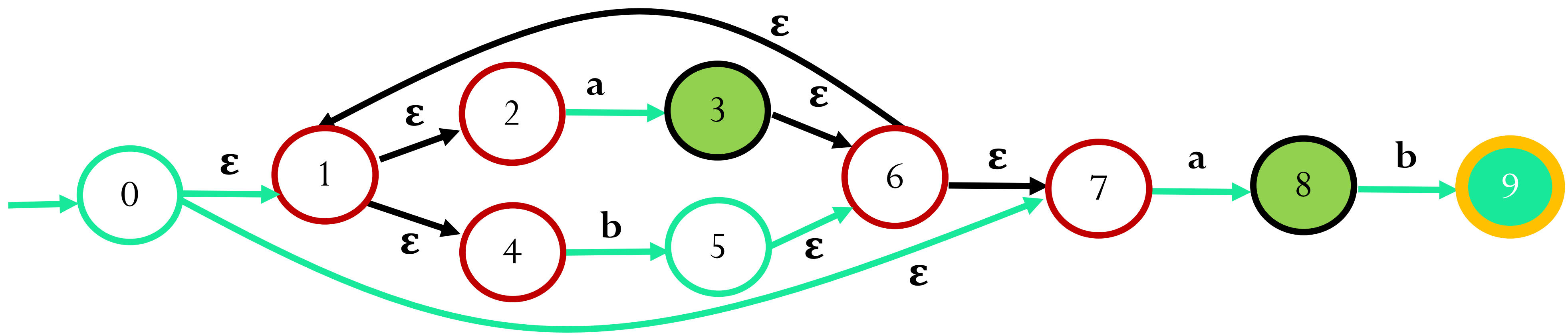


Start with the start state:
ε-closure(0):{0,1,2,4,7} = A
(A, a)= ({0,1,2,4,7}, a) = {0,a} ∪{1,a} ∪{2,a} ∪{4,a} ∪{7,a}
= Φ ∪ Φ ∪{3} ∪ Φ ∪ {8}

= ε -closure (3) ∪ ε -closure (8)

State	a	b
A (0,1,2,4,7)		

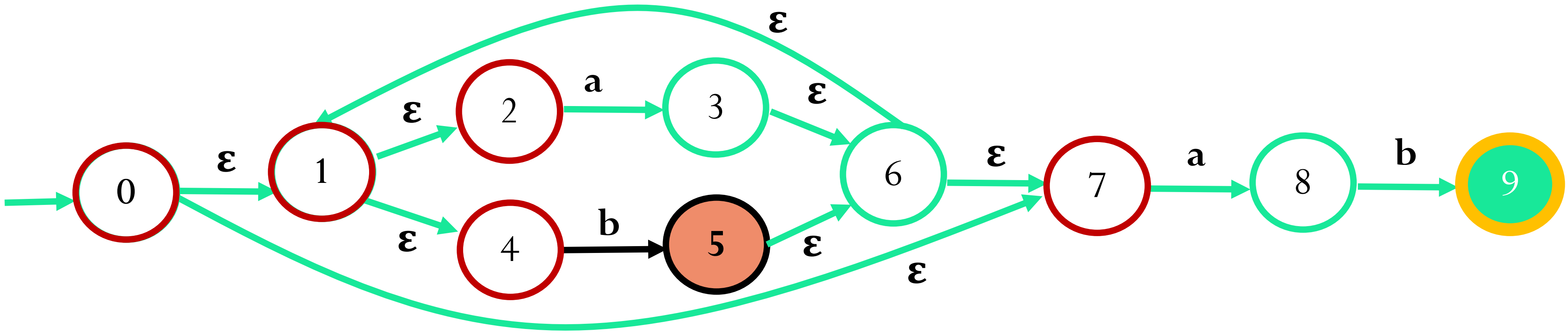
Subset Construction for (a/b)*ab



State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	

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

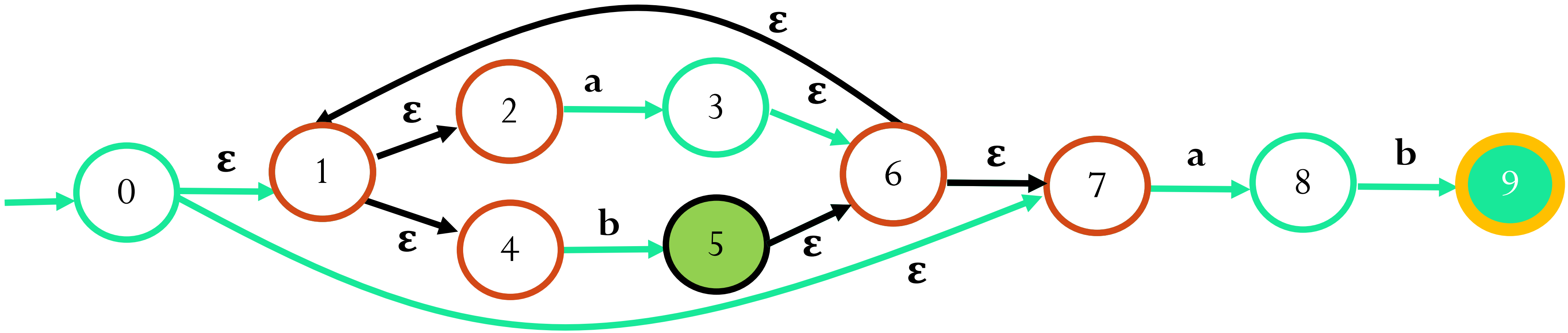
Subset Construction for (a/b)*ab



(A, b)= ({0,1,2,4,7}, b)
=\{0,b\} U\{1,b\} U\{2,b\}U\{4,b\} U\{7,b\}
= Φ U Φ U Φ U\{5\} U Φ
= ε -closure (5)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	

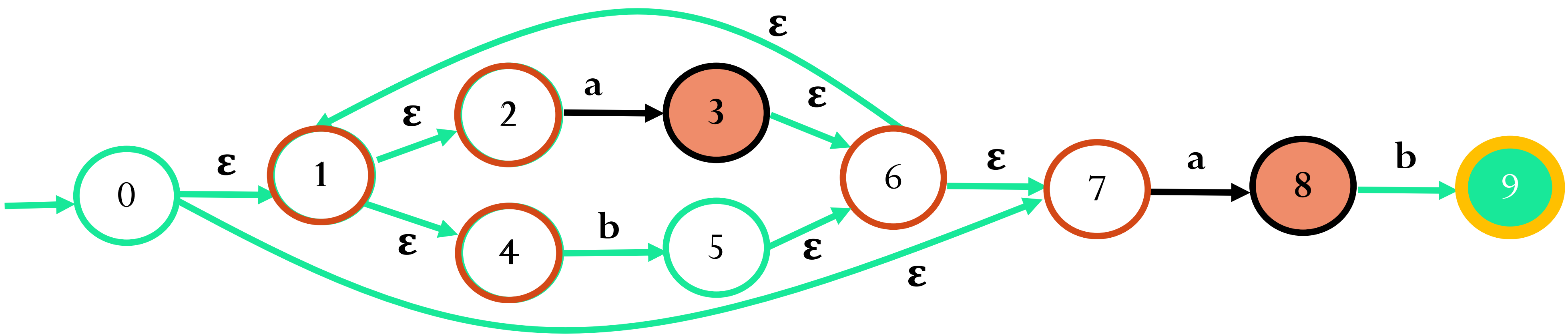
Subset Construction for (a/b)*ab



(A, b)= ε -closure (5)
= {1,2,4,5,6,7}=C

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)

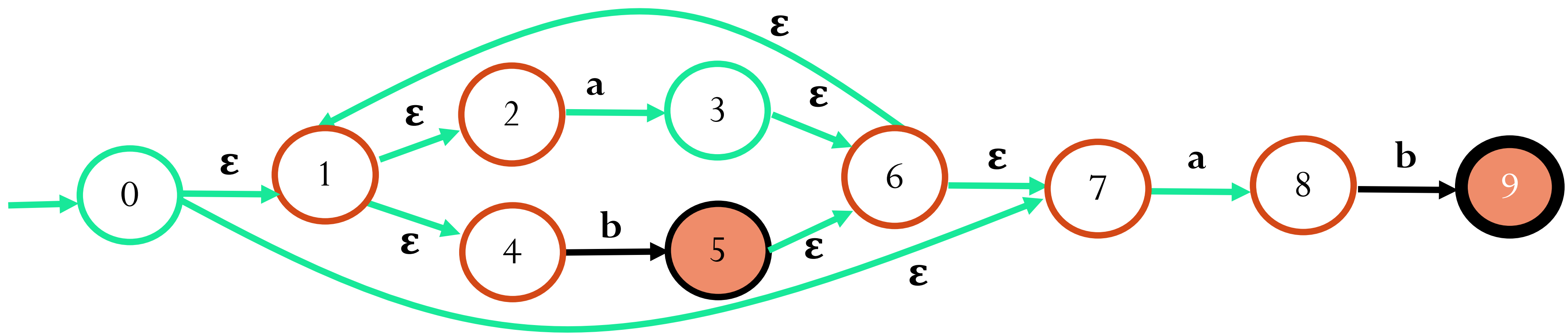
Subset Construction for (a/b)*ab



(B, a)= ({1,2,3,4,6,7,8}, a)
= {1,a}U{2,a} U{a,a} U{4,a}U{6,a}U{7,a} U{8,a}
= Φ U{3} U Φ U Φ U Φ U {8} U Φ
= ε -closure (3) U ε -closure (8)
= {1,2,3,4,6,7,8}=B (Slide No. 55)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	

Subset Construction for (a/b)*ab



(B, b)= ({1,2,4,5,6,7,8}, b)

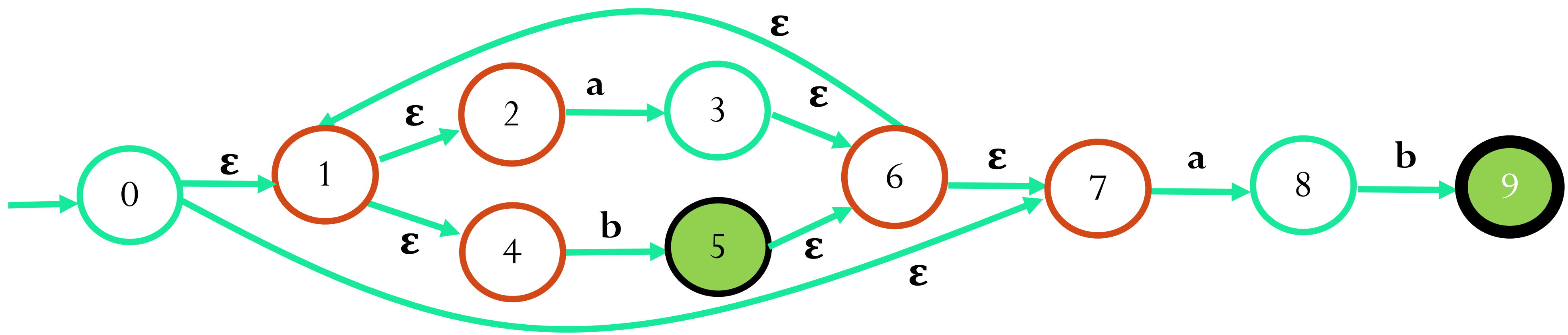
={1,b} U{2,b} U{4,b} U{5,b} U{6,b} U{7,b} U{8,b

= Φ U Φ U{5} U Φ U Φ U Φ{9}

= ε -closure (5) U ε -closure (9)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	

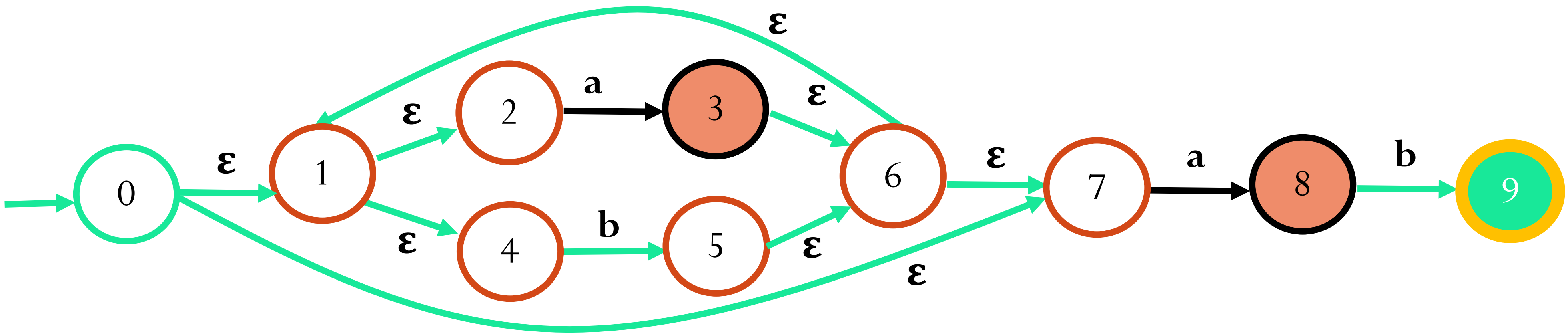
Subset Construction for (a/b)*ab



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

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)

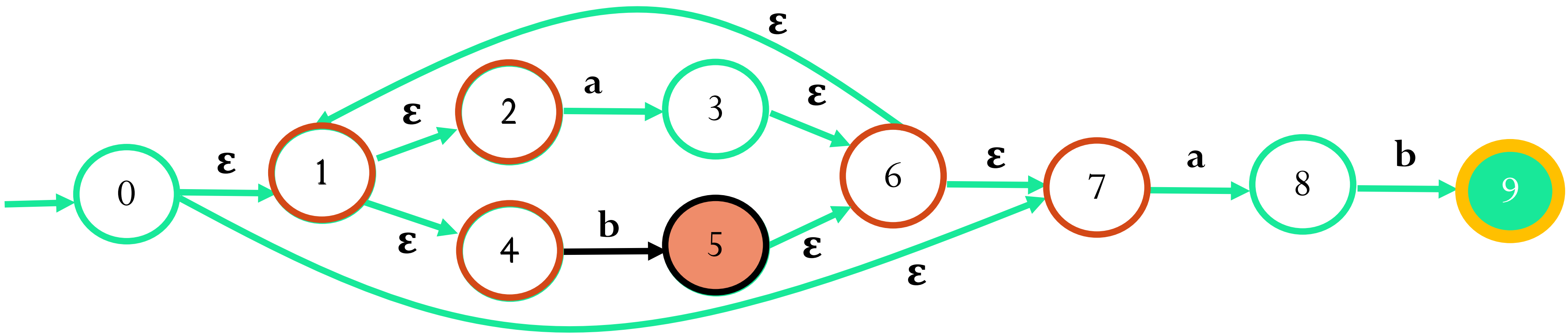
Subset Construction for (a/b)*ab



(C, a)= ({1,2,4,5,6,7}, a)
= {1,a}U{2,a}U{4,a}U{5,a}U{6,a} U{7,a}
= Φ U{3} U Φ U Φ U Φ U{8
= ε -closure (3) U ε -closure (8)
= {1,2,3,4,6,7,8}=B (Slide no. 55)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)
C	B	

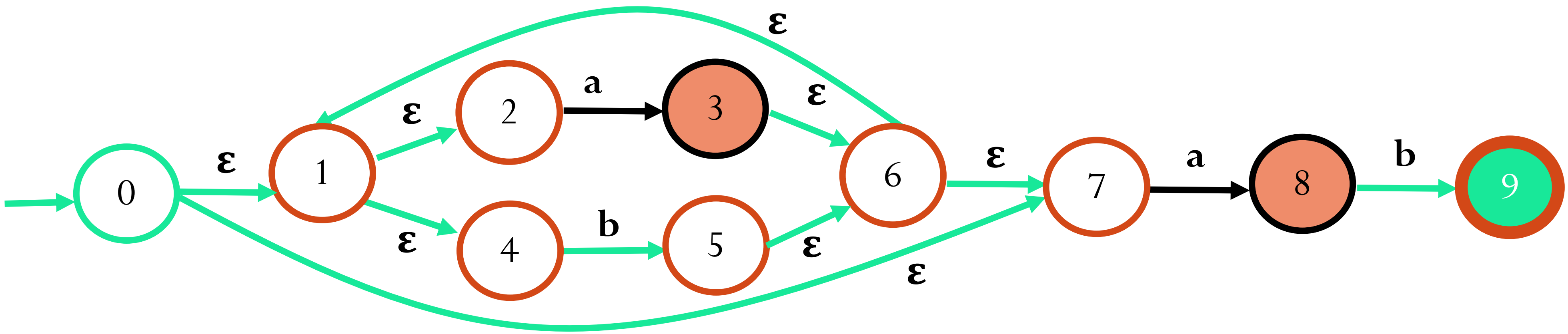
Subset Construction for (a/b)*ab



(C, b)= ({1,2,4,5,6,7}, b)
= {1,b} U{2,b} U{4,b}U{5,b} U{6,b} U{7,b}
= Φ U Φ U{5} U Φ U Φ U Φ
= ε -closure (5)= {1,2,4,5,6,7}=C (Slide no. 57)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)
C	B	C

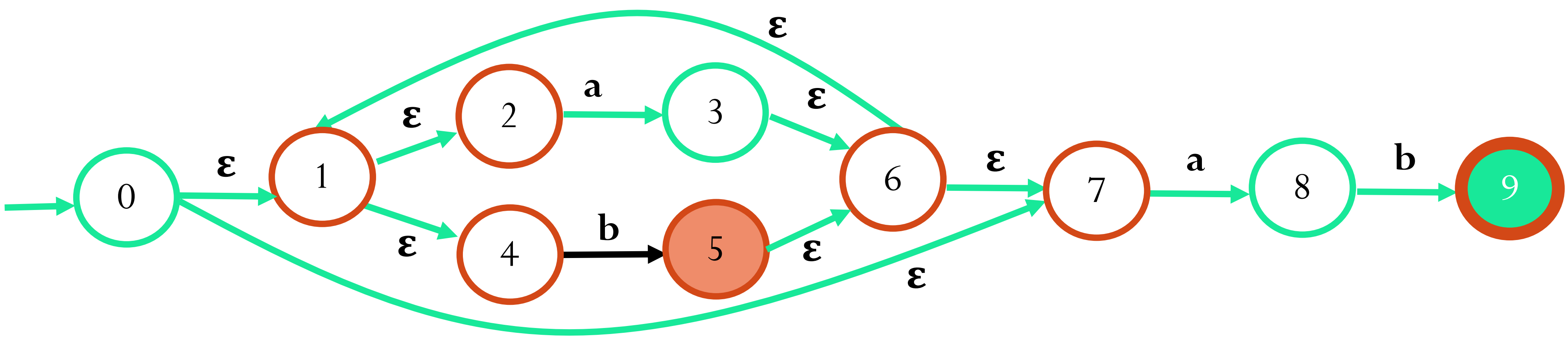
Subset Construction for (a/b)*ab



(D, a)= ({1,2,4,5,6,7,9}, a)
= {1,a}U{2,a}U{4,a}U{5,a}U{6,a} U{7,a} U{9,a}
= Φ U{3} U Φ U Φ U Φ U{8} U Φ
= ε -closure (3) U ε -closure (8)
= {1,2,3,4,6,7,8}=B (Slide no. 55)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)
C	B	C
D	B	

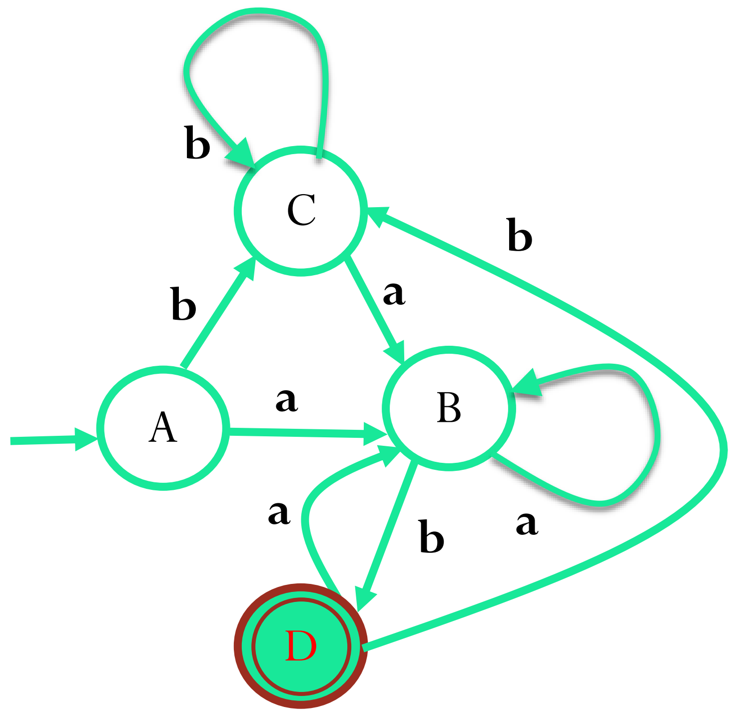
Subset Construction for (a/b)*ab



(D, b)= ({1,2,4,5,6,7,9}, b)
= {1,b}U{2,b}U{4,b}U{5,b}U{6,b} U{7,b} U{9,b}
= Φ U Φ U{5} U Φ U Φ U Φ U Φ
= ε -closure (5)= {1,2,4,5,6,7}=C (Slide no. 57)

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)
C	B	C
D	B	C

Subset Construction for (a/b)*ab



Final Output

State	a	b
A (0,1,2,4,7)	B (1,2,3,4,6,7,8)	C (1,2,4,5,6,7)
B	B	D (1,2,4,5,6,7,9)
C	B	C
D	B	C

- **Here state A is start state since set ‘A’ has state ‘0’ in its subset which is start state in the NFA with Thompson’s construction.**
- **D is final state since the set D has state ‘9’ which is final state in the NFA with Thompson’s Construction**

ϵ-closure(T)

push all states of *T* onto *stack*
initialize ϵ-*closure(T)* to *T*
while (*stack* is not empty) do
    begin
    pop *t*, the top element, off *stack*;
    for (each state u with an edge from *t* to *u* labelled ϵ do
        begin
        if (*u* is not in ϵ-*closure(T)*) do
            begin
            add *u* to ϵ-*closure(T)*
            push *u* onto *stack*
            end
        end
    end

Converting a NFA into a DFA (subset construction)

```
put ε-closure({s0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
    begin
        mark S1
        for each input symbol a do
            begin
                S2 ← ε-closure(move(S1,a))
                if (S2 is not in DS) then
                        add S2 into DS as an unmarked state
                transfunc[S1,a] ← S2
            end
        end
```

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state s in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

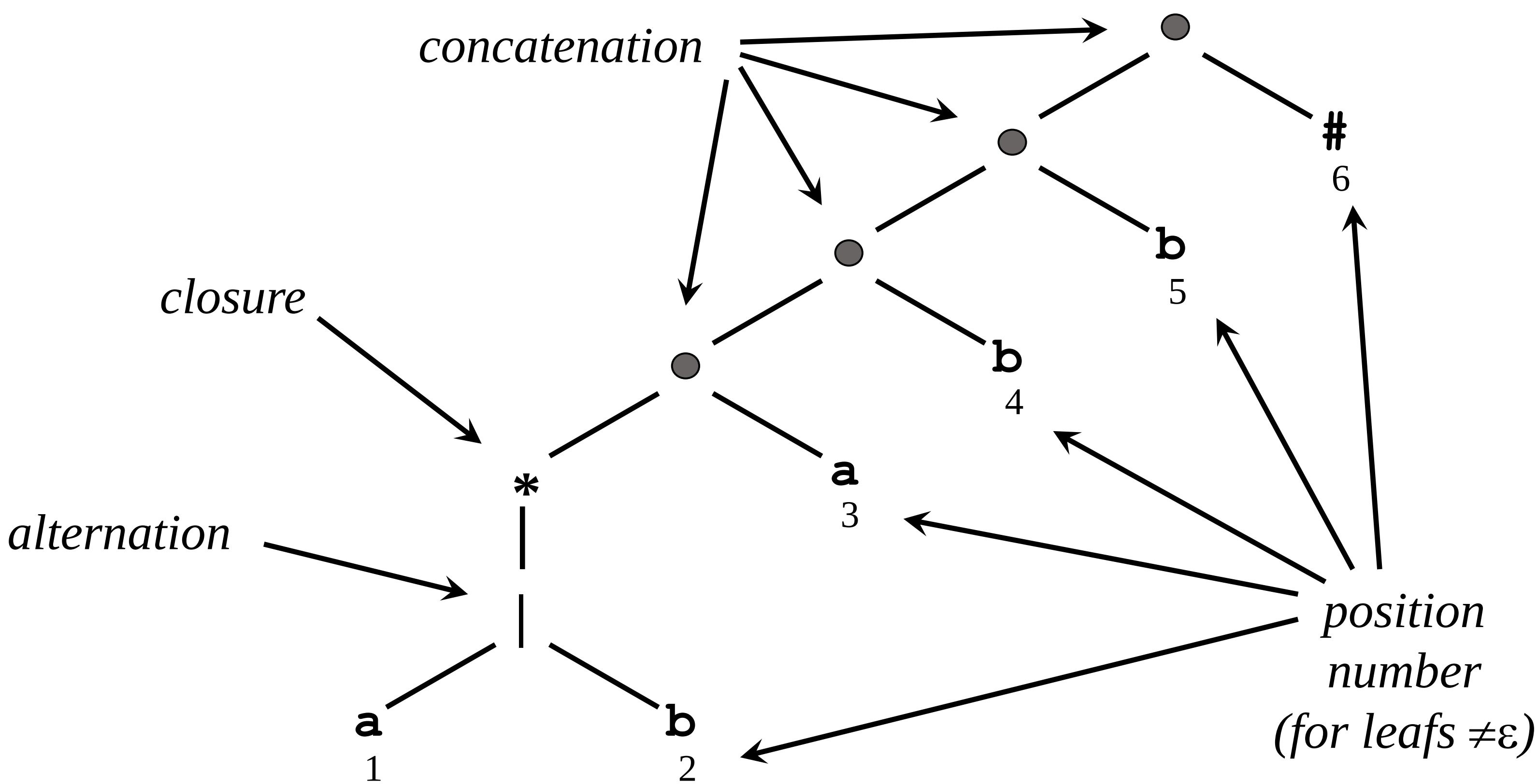
Section 1.5
RE to DFA through Syntax Tree Method or Direct Method

Converting Regular Expressions Directly to DFAs

- Important state
- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

  r ➔ (r)# augmented regular expression
- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #) will be numbered (position numbers).

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



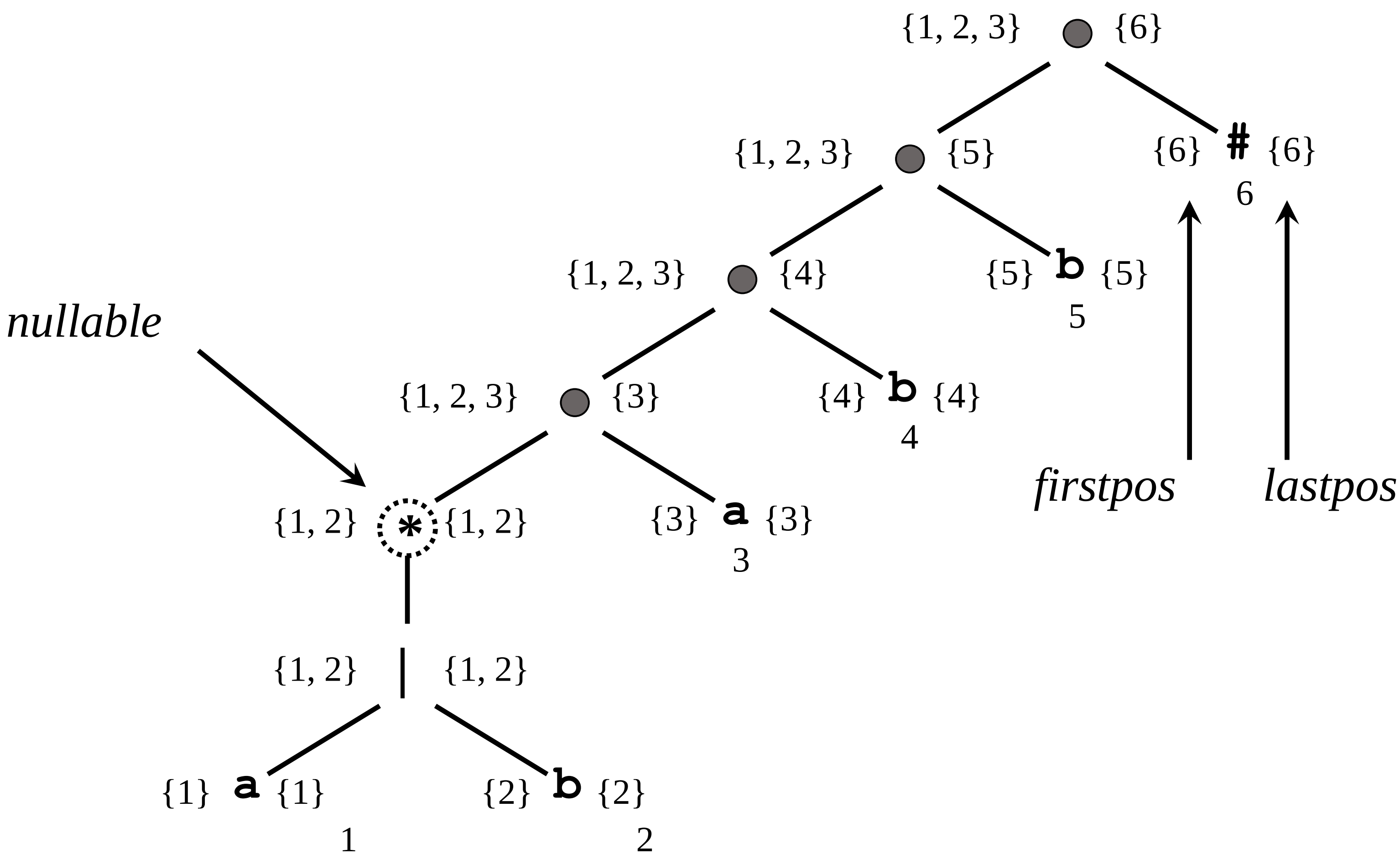
From Regular Expression to DFA Directly: Annotating the Tree

- *nullable*(*n*): the subtree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subtree at node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated by the subtree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree

From Regular Expression to DFA Directly: Annotating the Tree

Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
| / \\ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
• / \\ c_1 c_2	$nullable(c_1)$ and $nullable(c_2)$	**if** $nullable(c_1)$ **then** $firstpos(c_1) \cup$ $firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \cup$ $lastpos(c_2)$ **else** $lastpos(c_2)$
* | c_1	true	$firstpos(c_1)$	$lastpos(c_1)$

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



From Regular Expression to DFA Directly: Example

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-

(a/b)*a b b #

1 2 3 4 5 6

From RE to DFA Directly

(a/b)*a b b #

1 2 3 4 5 6

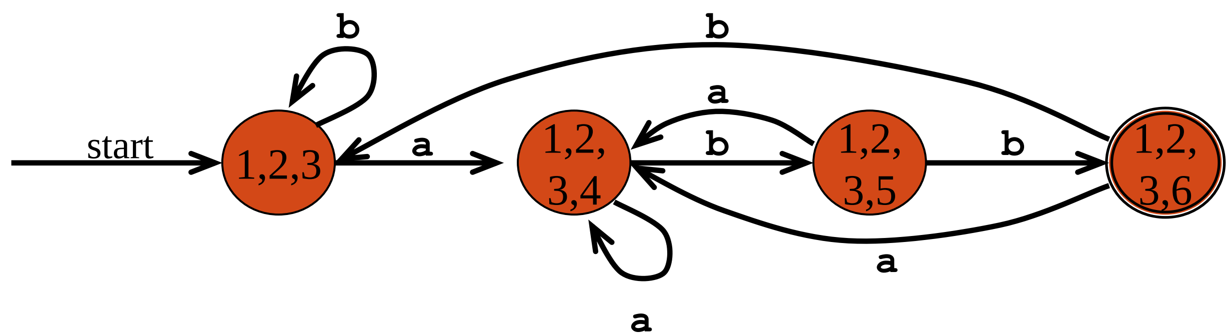
Let {1,2,3}=A				
A,a	({1,2,3},a)	followpos (1) U followpos(3)	{1,2,3,4}	B
A,b	({1,2,3},b)	followpos (2)	{1,2,3}	A
B,a	({1,2,3,4},a)	followpos (1) U followpos(3)	{1,2,3,4}	B
B,b	({1,2,3,4},b)	followpos (2) U followpos(4)	{1,2,3,5}	C
C,a	({1,2,3,5},a)	followpos (1) U followpos(3)	{1,2,3,4}	B
C,b	({1,2,3,5},b)	followpos (2) U followpos(5)	{1,2,3,6}	D
D,a	({1,2,3,6},a)	followpos (1) U followpos(3)	{1,2,3,4}	B
D,b	({1,2,3,6},b)	followpos (2)	{1,2,3}	A

Node Name	Symbol	*followpos*
1	a	{1, 2, 3}
2	b	{1, 2, 3}
3	a	{4}
4	b	{5}
5	b	{6}
6	#	-

State	a	b
A	B	A
B	B	C
C	B	D
D	B	A

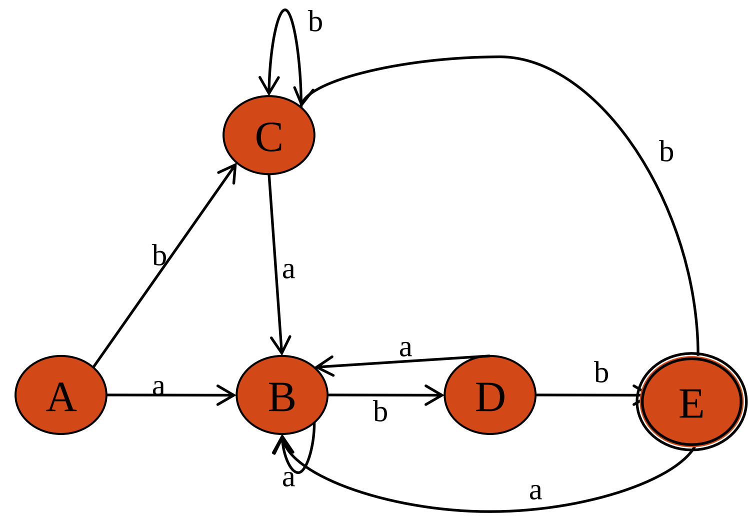
From Regular Expression to DFA Directly: Example

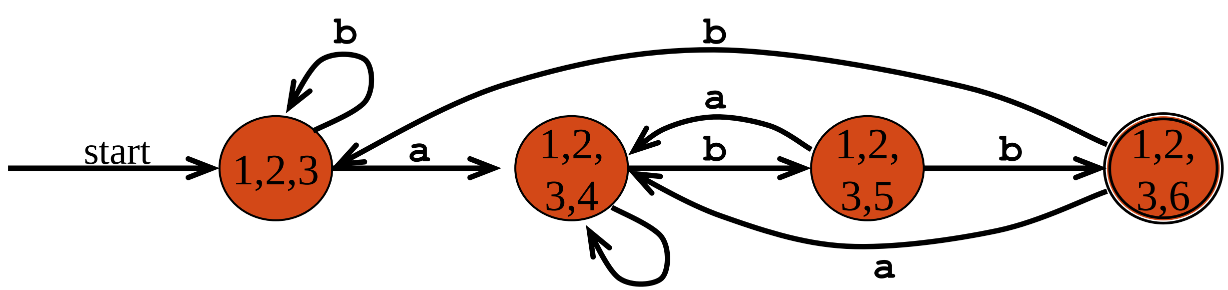
Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-



Different DFA's for (a|b)*abb

State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C





State	a	b
A	B	A
B	B	C
C	A	D
D	B	A

From Regular Expression to DFA Directly: *followpos*

```
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) ∪ firstpos(c2)
        end do
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) := followpos(i) ∪ firstpos(n)
        end do
    end if
end do
```

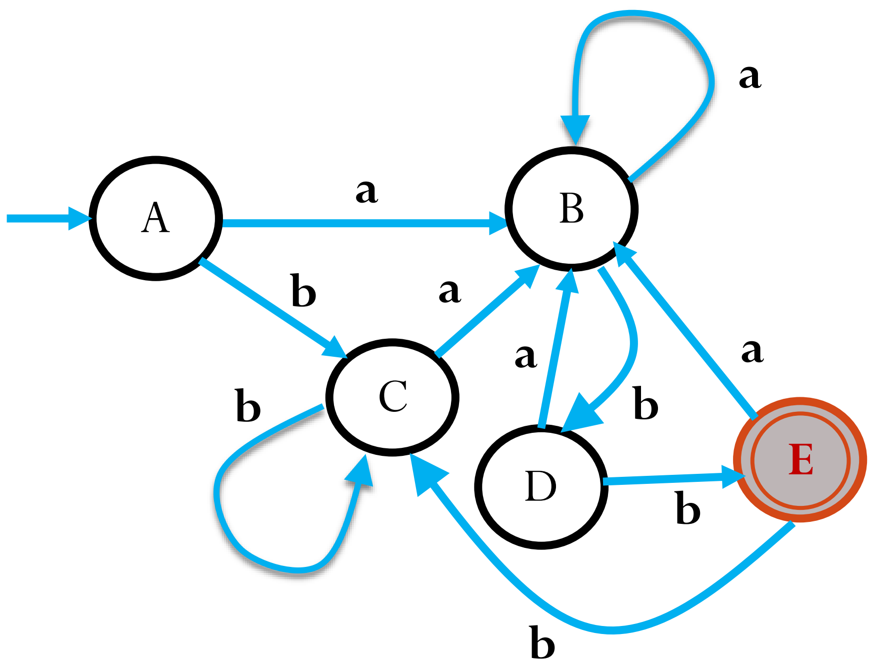
From Regular Expression to DFA Directly: Algorithm

s_0 := *firstpos*(*root*) where *root* is the root of the syntax tree
Dstates := $\{s_0\}$ and is unmarked
while there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    for each input symbol $a \in \Sigma$ **do**
        let *U* be the set of positions that are in *followpos*(*p*)
            for some position *p* in *T*,
            such that the symbol at position *p* is *a*
        if *U* is not empty and not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        end if
        Dtran[*T*,*a*] := *U*
    end do
end do

Section 1.6
Minimization of DFA

Minimization the following DFA, if possible

Question 1



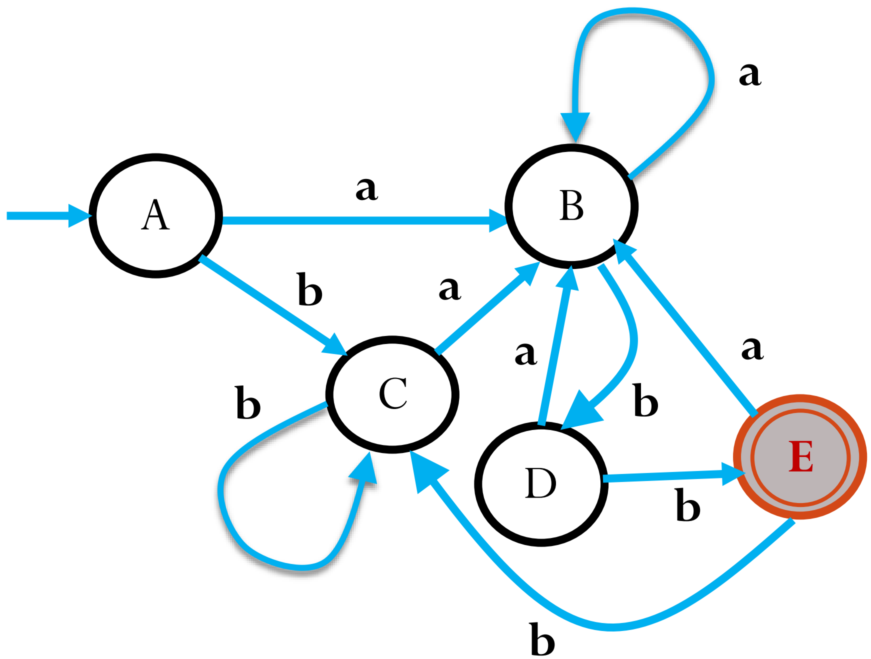
Using final and non final state

- Divide the entire set of states into two subsets: Set of final States and set of non final states.

- Consider each sub-set as a separate entity and identify if they need to be split further or can they be combined together

DFA Minimization using Partitioning Method

Question 1



	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

Draw the transition table corresponding to the given DFA

DFA Minimization using Partitioning Method

Question 1

Divide the states into two subsets- final and non-final

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

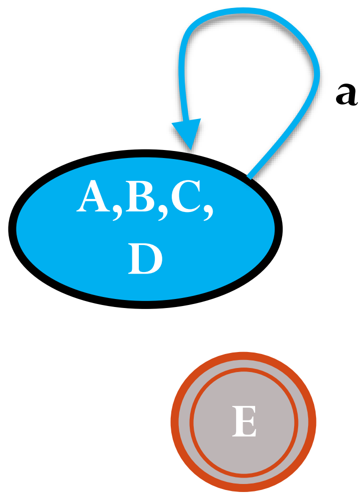
Set of non Final States (NF): {A,B,C, D}
Set of Final States (F): {E}

DFA Minimization using Partitioning Method

Question 1

Check O/P of all clubbed states (A,B,C,D) with Σ=a

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C



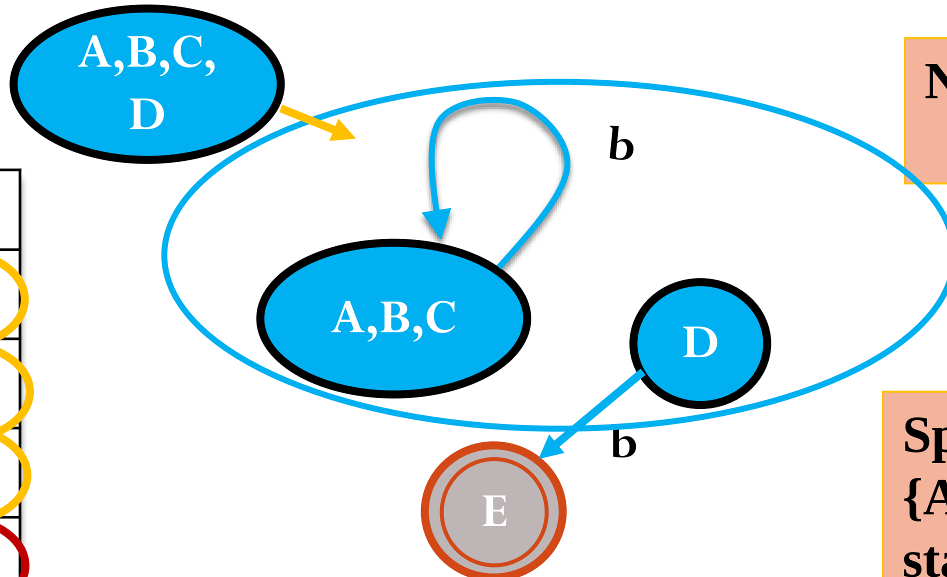
NF= {A,B,C,D}
F= {E}

DFA Minimization using Partitioning Method

Question 1

Check O/P of all clubbed states (A,B,C,D) with Σ=b

State	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
* E	B	C



NF= ({A,B,C} {D})
F= {E}

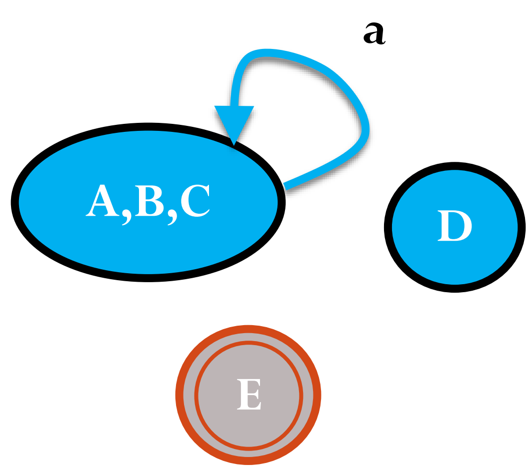
Split into two since {A,B,C} goes on states within {A,B,C) while state D goes to State {E}

DFA Minimization using Partitioning Method

Question 1

Check O/P of all clubbed states (A,B,C) with Σ=a

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C



NF= ({A,B,C}, {D})

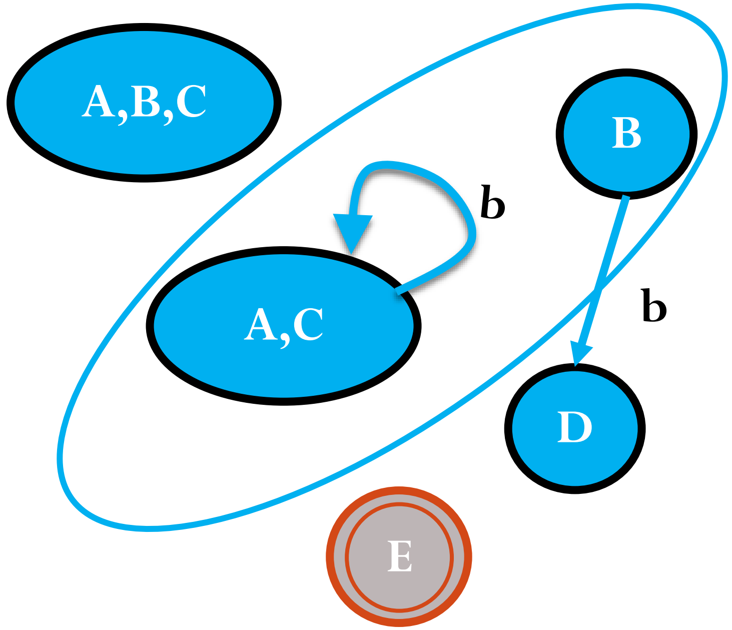
NO SPLIT

DFA Minimization using Partitioning Method

Question 1

Check O/P of all clubbed states (A,B,C) with Σ=b

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C



NF= ({A,C}, {B} {D})

Split into two since {A,C} goes to state {C} while {B} goes to State {D} which is already separated.

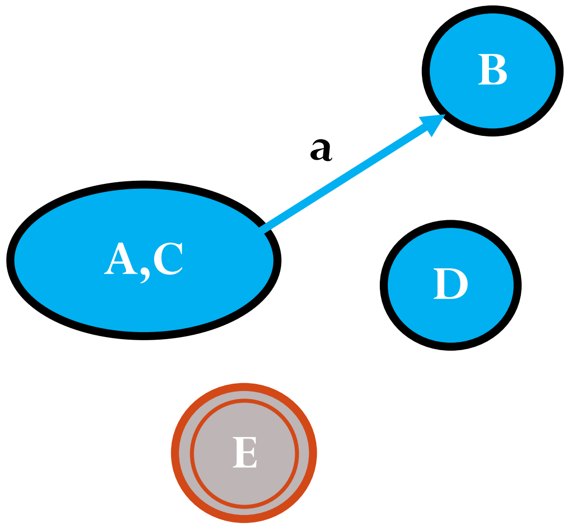
DFA Minimization using Partitioning Method

Question 1

Check O/P of all clubbed states (A,C) with Σ=a

NO SPLIT

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C



NF= ({A,C}, {B} {D})

Both A and C go to state B which is already separated

DFA Minimization using Partitioning Method

Question 1

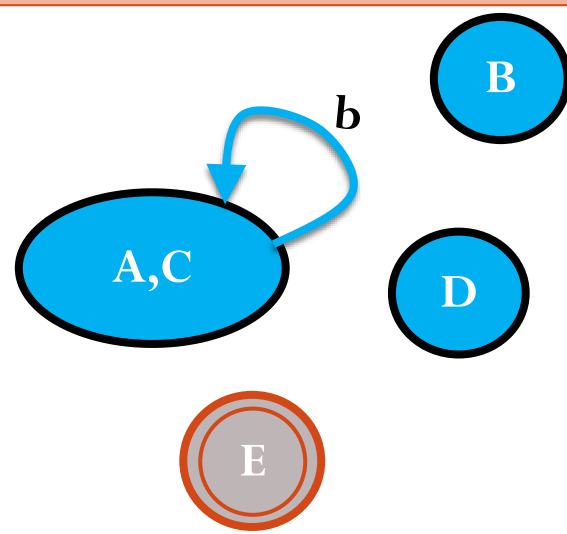
NO SPLIT

Check O/P of all clubbed states (A,C) with Σ=b

NF= ({A,C}, {B} {D})

Both A and C state go to same group {A,C} on Σ=b

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

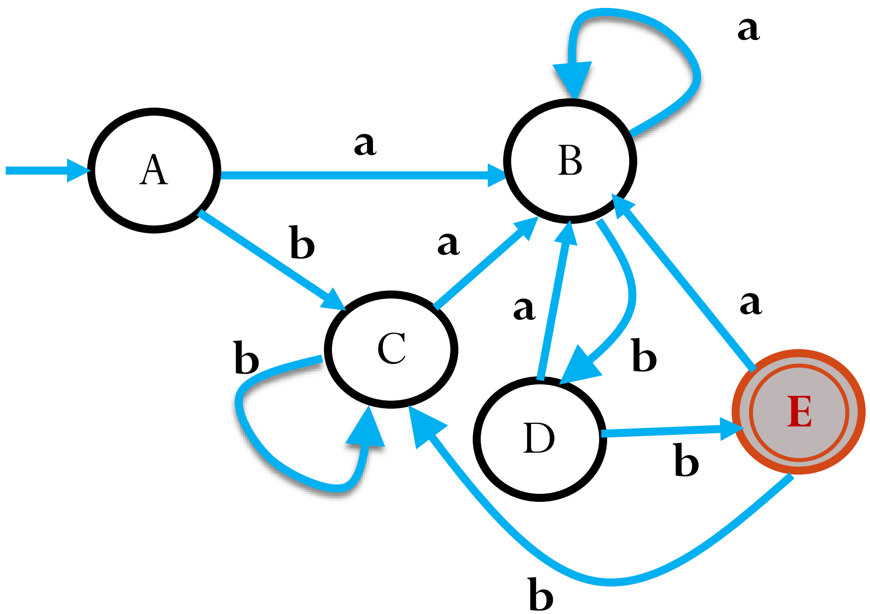


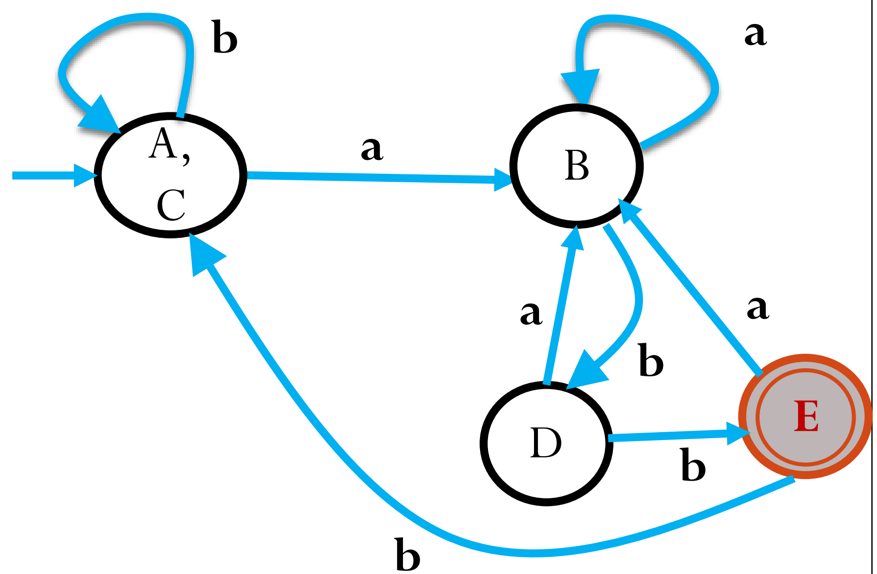
Since subset {A,C} remain as single combined state till end, both states will be joined together as a single state

DFA Minimization using Partitioning Method

	State	a	b
→	A	B	C
	B	B	D
	C	B	C
	D	B	E
*	E	B	C

	State	a	b
→	A,C	B	A,C
	B	B	D
*	D	B	E
	E	B	A,C

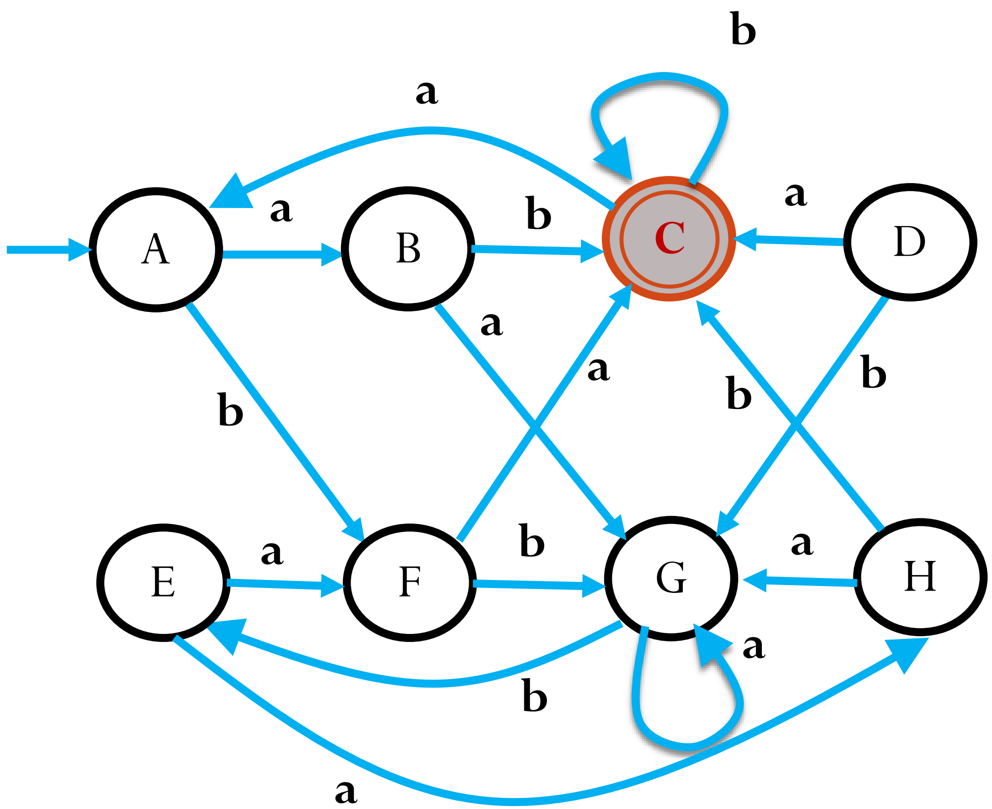




Final Output

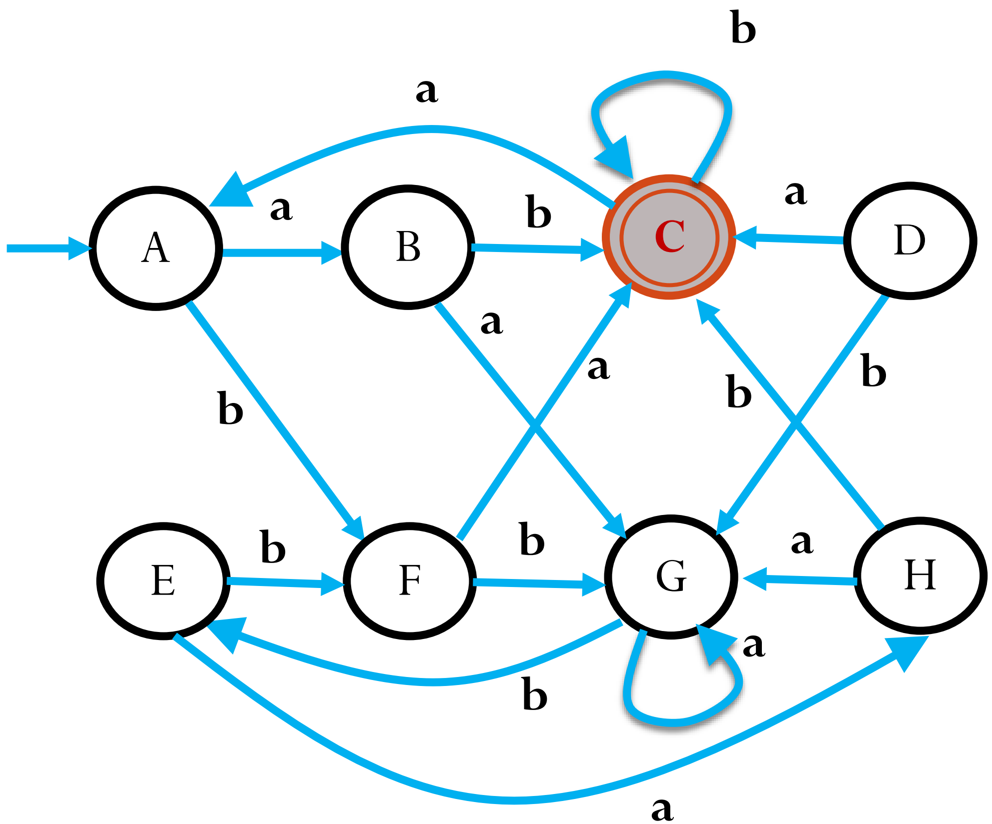
Minimization the following DFA, if possible

Question 2



DFA Minimization using Partitioning Method

Question 2



State	a	b
→ A	B	F
B	G	C
* C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

Draw the transition table corresponding to the given DFA

DFA Minimization using Partitioning Method

Question 2

Divide the states into two subsets- final and non-final

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

Set of Non Final States (NF): {A,B,D,E,F,G,H}

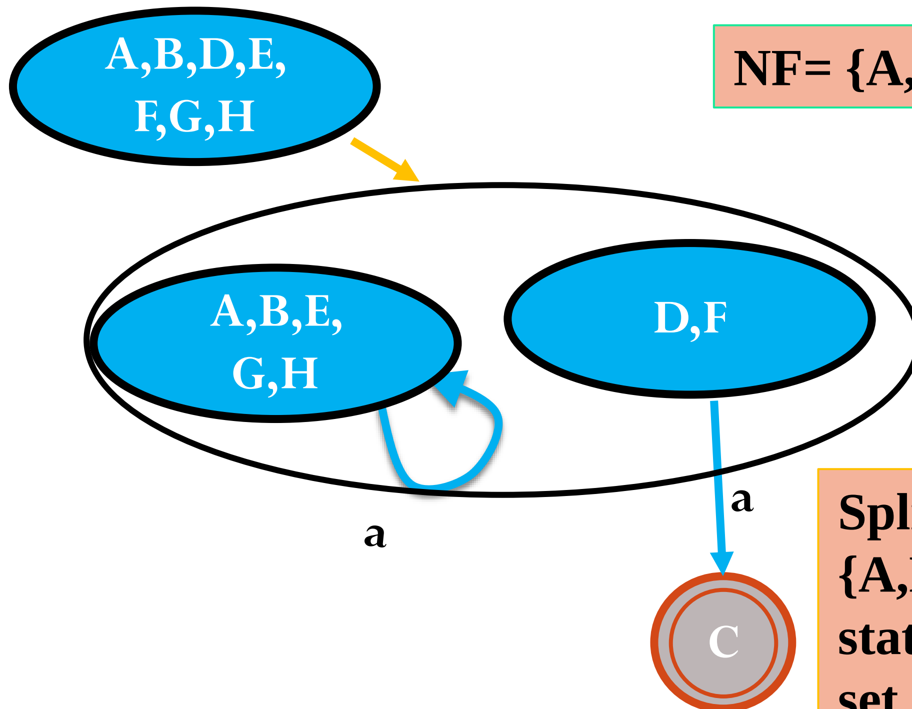
Set of Final States (F): {C}

DFA Minimization using Partitioning Method

Question 2

Check O/P of all clubbed states (A,B,D,E,F,G,H) with Σ=a

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C



NF= {A,B,E,G,H}, {D,F}

Split into two since {A,B,E,G,H} go to state states within its set while {D,F} goes to State {C}

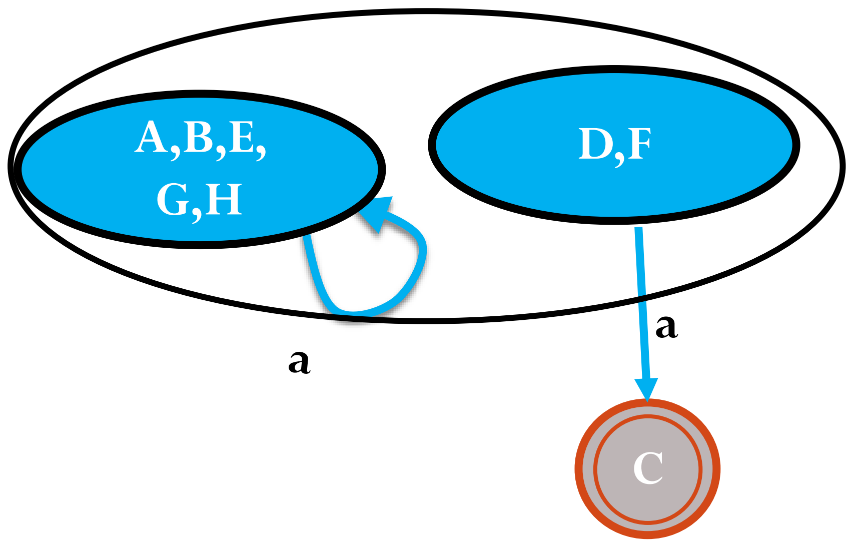
DFA Minimization using Partitioning Method

Question 2

Check O/P of all clubbed states (A,B,E,G,H) with Σ=a

NO SPLIT

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

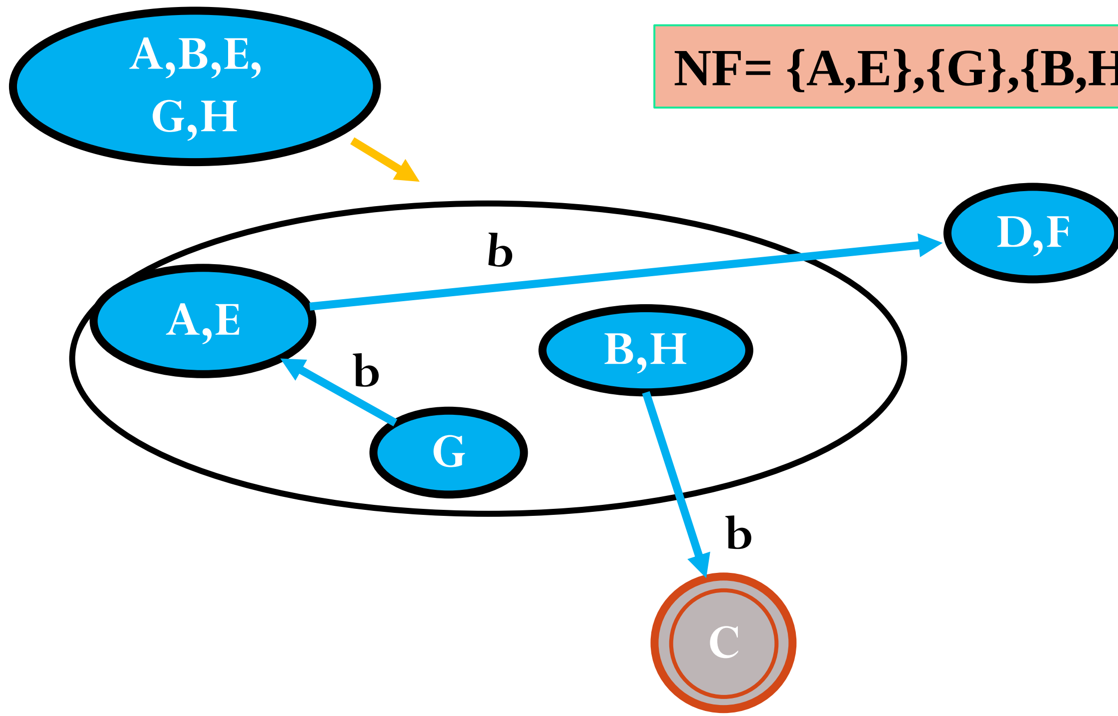


DFA Minimization using Partitioning Method

Question 2

Check O/P of all clubbed states (A,B,E,G,H) with Σ=b

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C



NF= {A,E},{G},{B,H},{D,F}

DFA Minimization using Partitioning Method

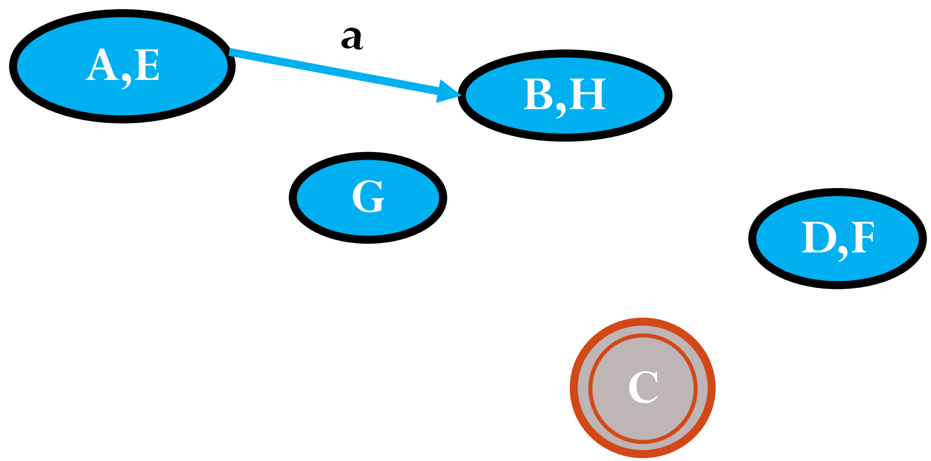
Question 2

Check O/P of all clubbed states (A,E) with Σ=a

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

NO SPLIT

NF= {A,E},{G},{B,H},{D,F}



DFA Minimization using Partitioning Method

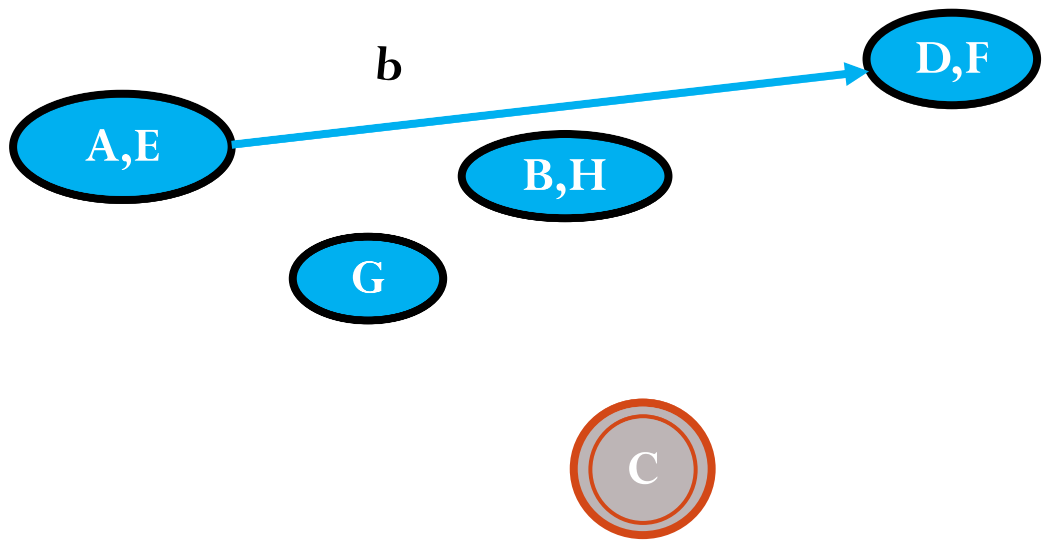
Question 2

Check O/P of all clubbed states (A,E) with Σ=b

NO SPLIT

NF= {A,E},{G},{B,H},{D,F}

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C



DFA Minimization using Partitioning Method

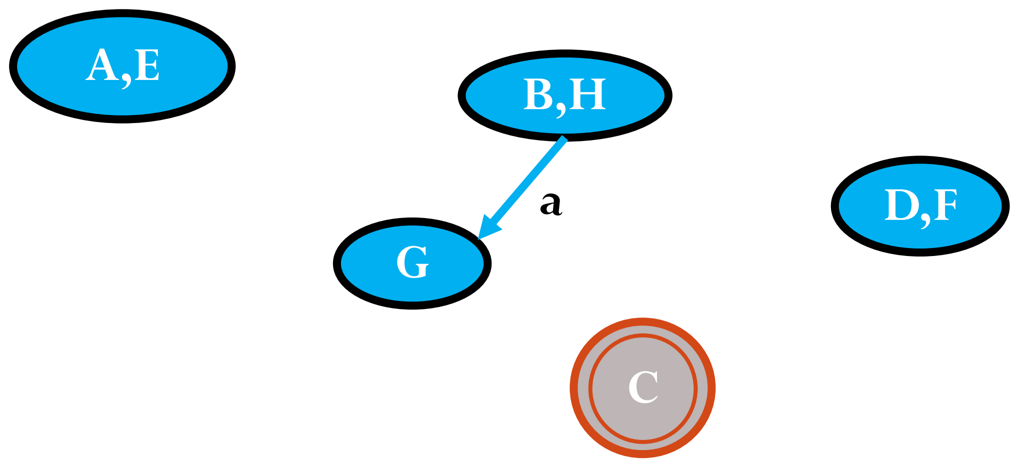
Question 2

Check O/P of all clubbed states (B,H) with Σ=a

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

NO SPLIT

NF= {A,E},{G},{B,H},{D,F}



DFA Minimization using Partitioning Method

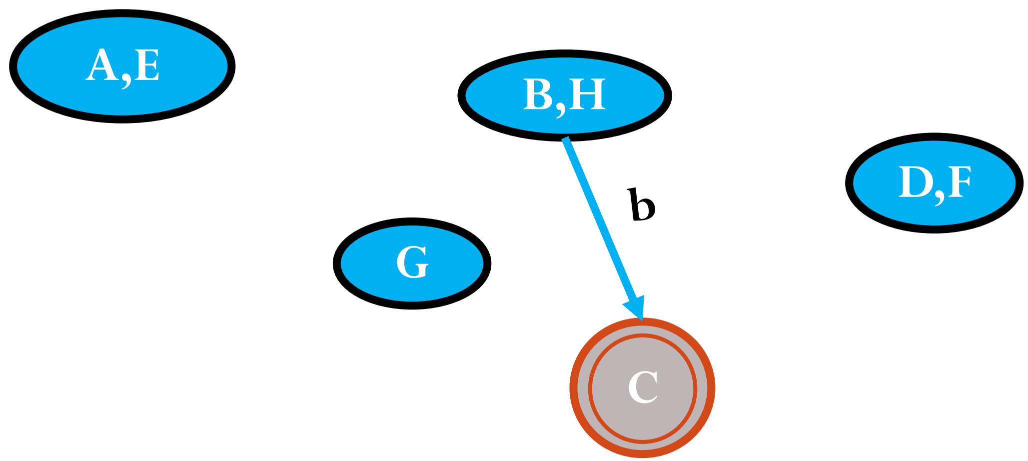
Question 2

Check O/P of all clubbed states (B,H) with Σ=b

NO SPLIT

NF= {A,E},{G},{B,H},{D,F}

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C



DFA Minimization using Partitioning Method

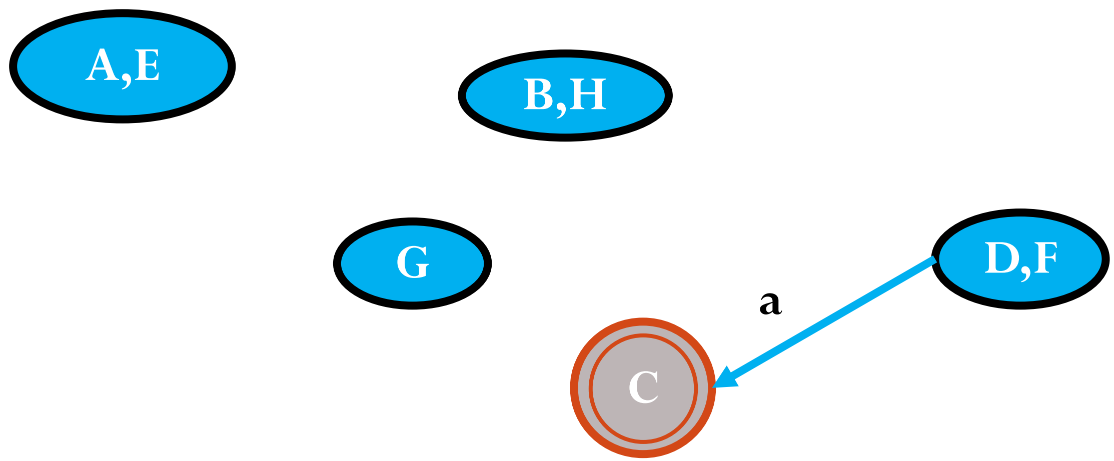
Question 2

Check O/P of all clubbed states (D,F) with Σ=a

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

NO SPLIT

NF= {A,E},{G},{B,H},{D,F}



DFA Minimization using Partitioning Method

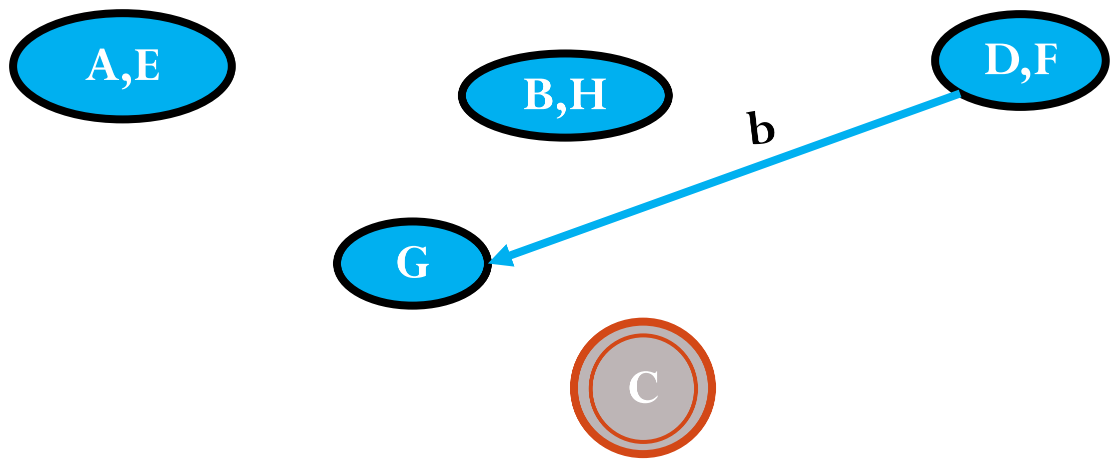
Question 2

Check O/P of all clubbed states (D,F) with Σ=b

	State	a	b
→	A	B	F
	B	G	C
*	C	A	C
	D	C	G
	E	H	F
	F	C	G
	G	G	E
	H	G	C

NO SPLIT

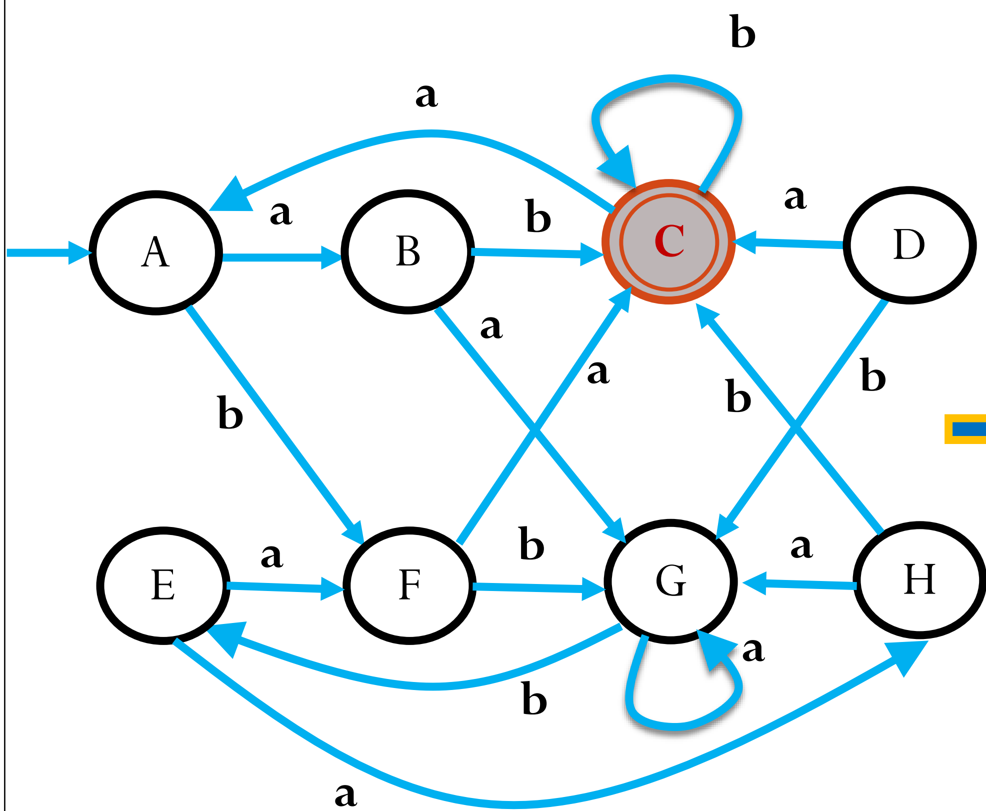
NF= {A,E},{G},{B,H},{D,F}

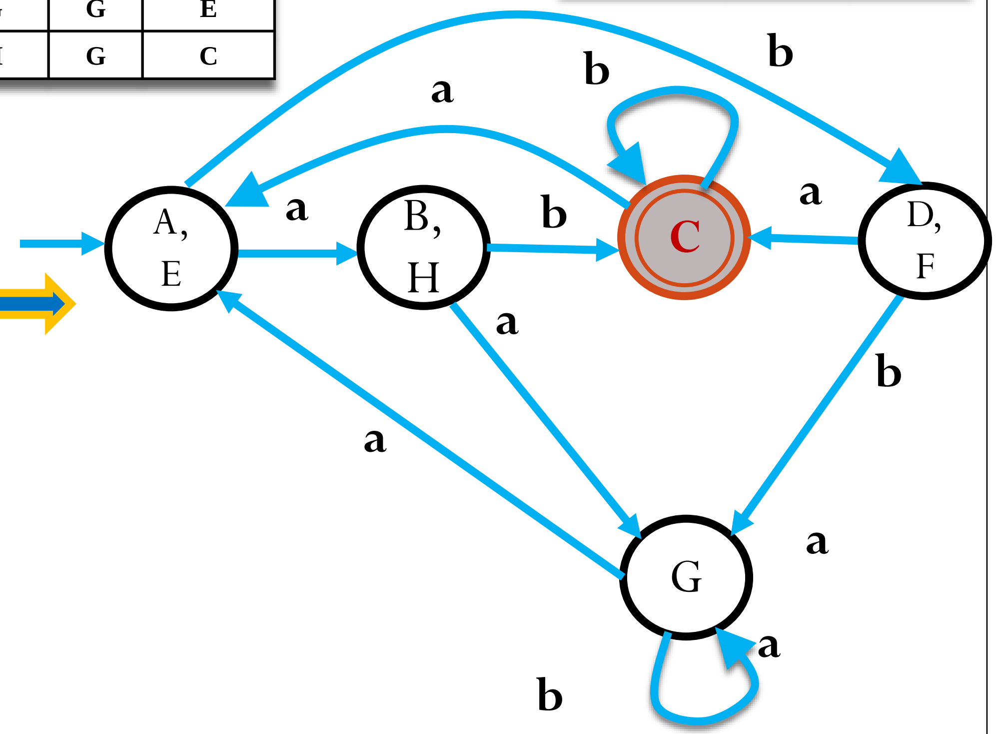


DFA Minimization using Partitioning Method

State	a	b
→ A	B	F
B	G	C
* C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

State	a	b
→ A,E	B,H	D,F
B, H	G	C
C	A,E	C
* D,F	C	G
G	G	A,E





Final Output

Thanks