CD : COMPILER DESIGN

Lexical Analysis

Marwadi University
Marwadi Chandarana Group
NAAC A+

Department of CE

Unit no : 2
Lexical Analysis
(01CE0714)

Prof. Shilpa Singhal

Outline :

- Role of Lexical Analyser
- Tokens, Lexemes, and Patterns
- Input Buffering
- Specification and Recognition of Tokens
- Regular expression and Regular Definition
- Transition Diagram
- Finite Automata
- Regular expression to NFA using Thompson's rule
- NFA to DFA conversion using subset construction method
- DFA Optimization / Minimization
- Regular expression to Direct DFA conversion

Marwadi University
Department of CE

Unit no : 2
Lexical Analysis
(01CE0714)

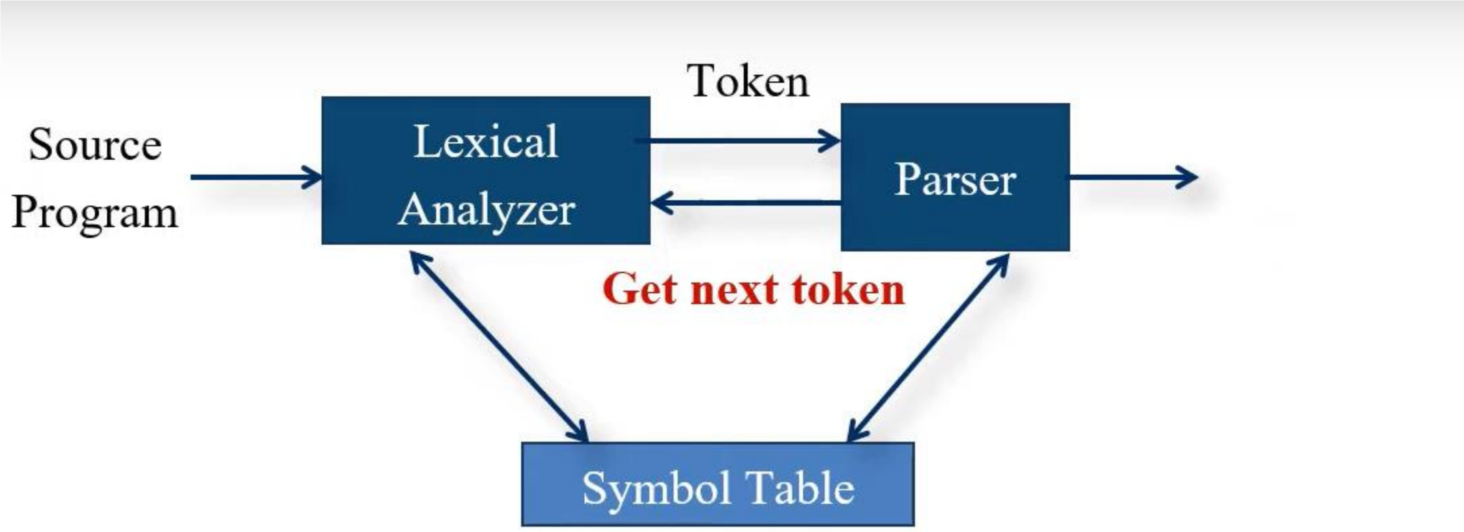
Prof. Shilpa Singhal

Role of Lexical Analyser

- Remove comments and white spaces in the form of blanks, tabs, and newline characters (aka scanning)
- Read input characters from the source program
- Group them into lexemes
- Produce as output a sequence of tokens
- Interact with the symbol table
- Correlate error messages generated by the compiler with the source program
- Lexical analysers are divided into two parts

1) Scanning
2) Lexical analysis

Role of Lexical Analyser



Communication between Scanner and Parser

- After receiving a **"Get next token"** command from parser, the lexical analyzer reads the input character until it can identify the next token.

Tokens, Lexemes and Patterns

- **Token:**

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

  1) Identifiers 2) keywords 3) operators 4) Delimiters 5)constants 6) Literals

- **Pattern:**

  A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.

- **Lexeme:**

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

Example

Token	lexeme	pattern
else	else	characters e, l, s, e
if	if	characters i, f
comparision	<=, < >, >=, >	< or <= or = or < > or >=
id	pi, ce, ec1	letter followed by letters & digit
num	3.14, 0, 3.09e17	any numeric constant
literal	“6th MU”	any character b/w “and ”

Example

Total = Ans + 30

- Tokens:
  - Total : Identifier 1
  - = : Operator 1
  - Ans : Identifier 2
  - + : Operator 2
  - 30 : Constant 1
- Lexems:
  - Lexems of identifiers : Total, Ans
  - Lexems of operators : =, +
  - Lexems of constant : 30

Attributes for Tokens

The tokens and associated attribute-values for the statement (given below) can be written as a sequence of pairs :

E = M * C ** 2

<**id** , pointer to symbol table entry for E>

<**assign_op** ,>

<**id** , pointer to symbol table entry for M>

<**multi_op** ,>

<**id** , pointer to symbol table entry for C>

<**exp_op** ,>

<**num** , integer value of 2>

Dealing with errors

Lexical Error

- **Lexical error** occurs when a sequence of characters does not match the pattern of any token. E.g A lexical analyser can not tell whether fi is a misspelling of the keyword *if* or an undeclared function identifier.Some examples are:

1. Exceeding length of identifier or numeric constants.

int a=2147483647 +1;

2. Spelling Error

int 3num= 1234;

3. Replacing a character with an incorrect character.

int x = 12$34;

4.Transposition of two characters.

int mian()

Dealing with errors

How lexical analyzer deals with errors?

1. **Panic mode recovery**: delete successive characters from remaining input until token or delimiter is found.

E.g **int a, 5abcd, sum, $2;**

int a, ~~5abcd~~ , sum, ~~$2~~ ; parser discards input symbol one at a time.

2. Transpose two adjacent characters

E.g unoin{ } → union{ }

3. Insert missing character

E.g it 5; → int 5;

4. Delete a character

E.g intt 5; → int 5;

5.Replace character by another

E.g itt 5; → int 5;

Input Buffering

Input Buffering

- Lexical analyzer reads the source program character by character from the secondary storage but it is costly. Therefore, a block of data is first read into a buffer and then scanned by lexical analyzer.
- It uses two pointer **begin(ptr)** and **forward(ptr)** to keep track of the pointer of input scanned.
- There are two types of buffer input scheme that is useful when look ahead is necessary.
    - Buffer Pairs
    - Sentinels

Input Buffering

- Lexical Analyser scans the input string from left to right, and character by character.
- Initially both the pointers point to the first character of input string:

Fig: initial configuration of buffer

1.Buffer Pairs

- Buffer is divided into **two N-characters halves**.
- N is number of characters on one disk block. E.g. 1024 or 4096 bytes.
- N characters are read from the input file to the buffer using one system read command.
- **eof** is inserted at the end if the number of characters is less than N.

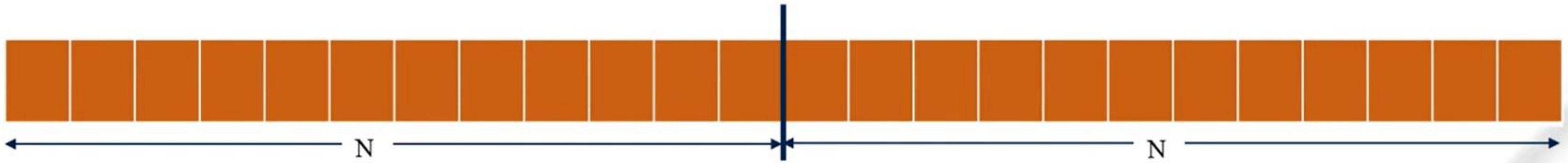


Fig: input buffer

Buffer Pairs

- Once a lexeme is found, lexemebegin is set to the character immediately after the lexeme which is just found and forward is set to the character at its right end.
- Current lexeme is the set of characters between two pointers

Fig: input buffer with buffer pair

Buffer Pairs

Code to advance forward pointer

if *forward at end of first half* ***then begin***

reload second half:

forward := forward + 1

end

else if *forward at end of second half* ***then begin***

reload first half;

move forward to beginning of first half

end

else *forward := forward + 1;*

2.Sentinels

- In buffer pairs, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
  - ***Test 1:*** For end of buffer.
  - ***Test 2:*** To determine what character is read.

Sentinels

- The **sentinel is a special character that cannot be part of the source program**. (**eof** character is used as sentinel).
- So each buffer half will extended to hold a sentinel eof.
- This technique optimize the code by reducing the two tests to one by extending each buffer half to hold a sentinel character at the end.

Fig: input buffer with sentinel

Sentinels

Code to advance forward pointer

```
forward := forward + 1;
if forward = eof then beign
        if forward at end of first half then begin
                reload second half:
                forward := forward + 1;
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else terminate lexical analysis;
end
```

Specification of tokens

Specification of tokens

- There are 3 specifications of tokens:
  - Strings
  - Language
  - Regular expression
- Strings and Languages:
  - An **alphabet** or character class is a finite set of symbols.
  - A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - A **language** is any countable set of strings over some fixed alphabet.

Operations of Strings

- A **prefix of string** s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.
- A **suffix of string** s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.
- A **substring of s** is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.
- The **proper prefixes, suffixes, and substrings of a string** s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
- A **subsequence of s** is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.

Exercise

- Write prefix, suffix, substring, proper prefix, proper suffix and subsequence for following strings:
  - Revolution

Prefix : Revo
Suffix : on
Substring : vol
proper prefix and suffix : as above
subsequence : eoton

Operations of Languages

The following are the operations that can be applied to languages:

(Applying these operations on L and S)

- **Union** (L U S) : $\{t \mid t\ is\ in\ L\ or\ t\ is\ in\ S\ \}$
- **Concatenation** (LS) : $\{tz \mid t\ is\ in\ L\ and\ z\ is\ in\ S\ \}$
- **Kleene Closure** (L^*) :
  L^* *denotes "zero or more concatenation of" L.*
- **Positive Closure** (L^+) :
  L^+ *denotes "one or more concatenation of" L.*

Example

Let $L = \{0, 1\}$ and $S = \{a, b, c\}$

- Union : $L \cup S = \{0, 1, a, b, c\}$
- Concatenation : $L . S = \{0a, 1a, 0b, 1b, 0c, 1c\}$
- Kleene Closure : $L^* = \{\varepsilon, 0, 1, 00, \ldots\ldots\}$
- Positive Closure : $L^+ = \{0, 1, 00, \ldots\ldots\}$

Regular Expression
&
Regular Definition

Regular Expression

- Regular Expression is a sequence of characters that define a pattern.
- **Notations:**
  - One or more instances : +
  - Zero or more instances: *
  - Zero or one instance: ?
  - Alphabets : Σ
  - Regular expression r and regular language for it is L(r)

Regular Expression

Rules to define Regular Expression:

- ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
- If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}.
- Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,
  - (r)|(s) is a regular expression denoting the language L(r) U L(s).
  - (r)(s) is a regular expression denoting the language L(r)L(s).
  - (r)* is a regular expression denoting (L(r))*.
  - (r) is a regular expression denoting L(r).

Regular Expression

- L = Zero or more occurences of a = a^*
  - $a^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\ldots\}$ (Infinite elements)

- L = One or more occurences of a = a^+
  - $a^+ = \{a, aa, aaa, aaaa, \ldots\ldots\}$ (Infinite elements)

Precedence and Associativity

- The unary operator * has highest precedence and is left associative.
- Concatenation has second highest precedence and is left associative.
- | has lowest precedence and is left associative.

Example :

(a) | ((b)*(c)) is equivalent to

a | b*c

Examples (Regular Expression)

Language	String	Regular Expression
0 or 1	0, 1	0 | 1
1 or 10 or 111	1, 10, 111	1 | 10 | 111
Strings having one or more 0	0, 00, 000, 0000,……	0^+
All possible binary strings over Σ = {0, 1}	ε, 0, 1, 00, 01, 10, 11, 000,……	$(0 \mid 1)^*$
All possible strings of length 3 over Σ = {a, b, c}	aaa, aba, abc, abb,……	(a|b|c) (a|b|c) (a|b|c)

Language	String	Regular Expression
One or more occurrences of 0 or 1 or both	0, 1, 00, 01, 10, 11, 111, 101,……	$(0 \mid 1)^+$
Binary string ending with 0	0, 10, 100, 110, 00, 010,……	$(0 \mid 1)^* \, 0$
Binary string starting with 1	1, 10, 100, 110, 101, 1101,……	$1 \, (0 \mid 1)^*$
Binary string starting with 1 and ending with 0	10, 110, 100, 110, 1100, 1110, 1000,……	$1 \, (1 \mid 0)^* \, 0$
String starting and ending with same character for Σ = {0, 1}	00, 11, 010, 000, 101, 1101, 0110, 1011,……	$1 \, (1 \mid 0)^* \, 1 \; + \; 0 \, (1 \mid 0)^* \, 0 \; + \; 0 \; + \; 1 \; + \; \varepsilon$
String ending with 01 for Σ = {0, 1}	01, 101, 001, 1001, 1101,……	$(0 \mid 1)^* 01$

Language	String	Regular Expression
Language consisting of exactly two 0's for $\Sigma = \{0, 1\}$	00, 010, 000, 001, 0101,……	$1^*01^*01^*$
All binary strings with length at least 3 for $\Sigma = \{0, 1\}$	000, 010, 110, 1111, 1011,……	$(0\|1)\ (0\|1)\ (0\|1)\ (0\|1)^*$
All binary strings where 2nd symbol from starting is 0 for $\Sigma = \{0, 1\}$	00, 10, 101, 100,……	$(0\|1)0(0\|1)^*$
Any number of a's followed by any number of b's followed by any number of c's	ε, abc, aaabc, abbbbbc, abccc, ab, accc,……	$a^*b^*c^*$

Exercise

Write regular expression for language specified

Over Σ = {a,b}

- String Length exactly 2
- String length atleast 2
- String length atmost 2
- Strings having Even length
- Strings having Odd Length
- String containing exactly two a's
- String starting with 0 and having odd length
- String starting or ending with 01 or 111

Regular Definition

- For notational convenience, we may give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.
- These names are known as regular definition.
- Regular definition is a sequence of definitions of the form:

  $d_1 \rightarrow r_1$

  $d_2 \rightarrow r_2$

  ……

  $d_n \rightarrow rn$

  Where d_i is a **distinct name** & r_i is a **regular expression**.

Example

- **Regular definition for identifier**
- letter → A|B|C|………..|Z|a|b|………..|z
- digit → 0|1|…….|9|
- id → letter (letter | digit)*

Regular Definition for Even Numbers

- (+|-|ε) (0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)
- Sign → + | -
- OptSign → Sign | ε (Sign ?)
- Digit → [0 – 9] (0 | 1 | ….. | 9)
- EvenDigit → [02468] (0 | 2 | 4 | 6 | 8)
- EvenNumber → OptSign Digit* EvenDigit

Example

Regular Definition for Unsigned Numbers

- digit → 0 | 1 | ..... | 9
- digits → digit digit*
- optionalFraction → .digits | ε
- optionalExponent →(E (+ | - | ε) digits) | ε
- number → digits optionalFraction optionalExponent

This can be simplified as:

- digit → [0-9]
- digits → digit+
- number → digits (. digits)? (E [+-]? digits)?

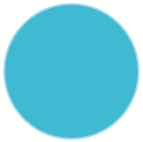
Transition Diagram

Transition Diagram

- **Transition diagram** is a special kind of flowchart for language analysis.
- In **transition diagram** the boxes of flowchart are drawn as circle and called as states.
- States are connected by arrows called as edges.
- The label or weight on edge indicates the input character that can appear after that state

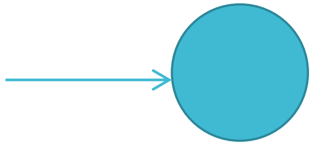
Transition Diagram

- Symbolized representation of transition diagram uses:

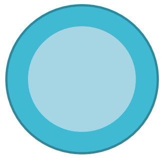


is a state

is a transition



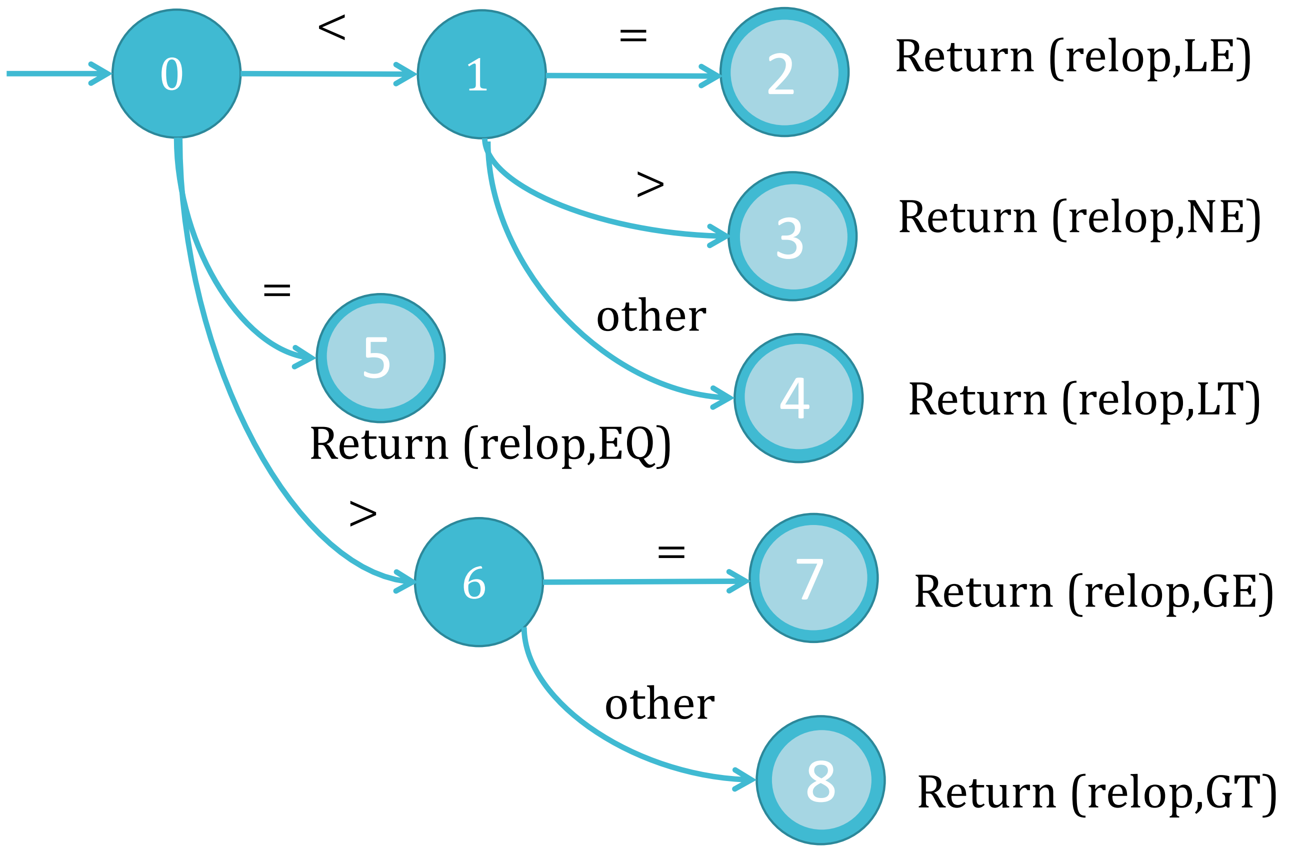
is a start state



is a final state

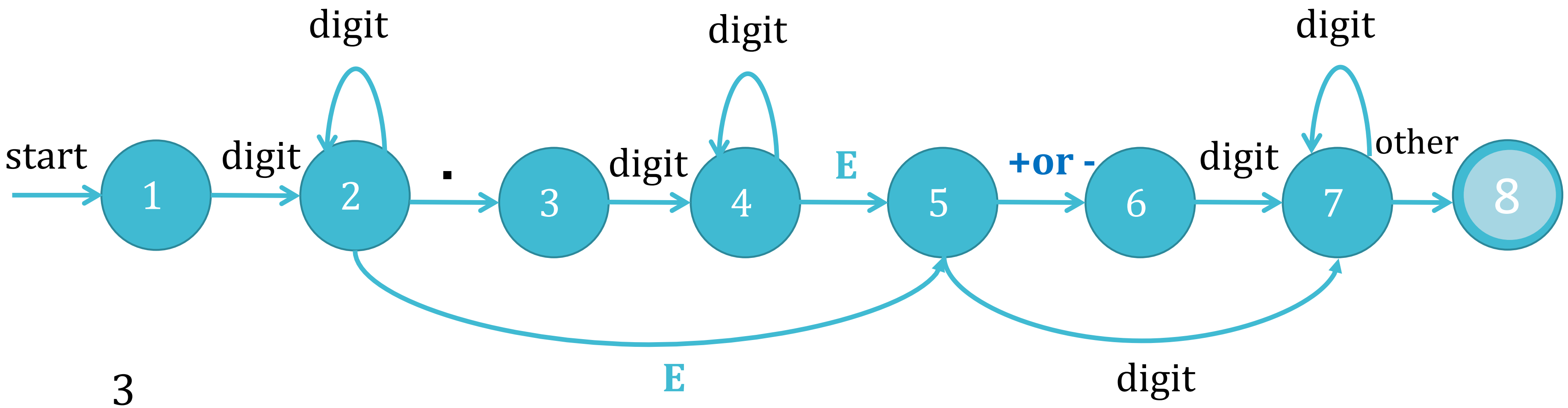
Transition Diagram : Example

Transition Diagram for Relational Operators



Transition Diagram : Example

Transition Diagram for Unsigned Numbers



3
5280
39.37
1.894 E - 4
2.56 E + 7
45 E + 6
96 E 2

Finite Automata

Finite Automata

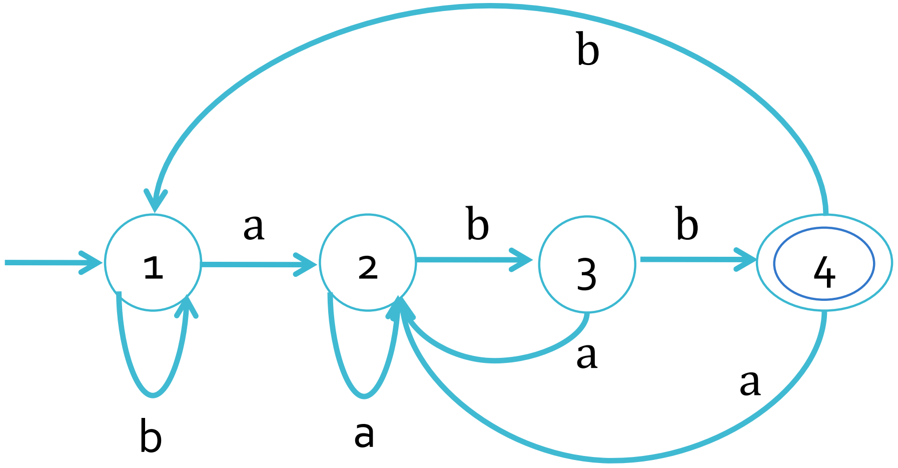
- A recognizer for a language is a program that takes input string as ‘x’ and answer “yes” if x is sentence of the language and “no” otherwise.
- FA results in “yes” or “no” based on each input string.
- FA M = (Q, $\boldsymbol{\Sigma}$, qo, A , δ)
  - Q : Set of finite states
  - $\boldsymbol{\Sigma}$: Set of input symbol
  - qo : Initial state qo ∈ Q
  - A : Set of Accepting States A ⊂ Q
  - δ : Transition function

$$Q \times \boldsymbol{\Sigma} \rightarrow Q$$

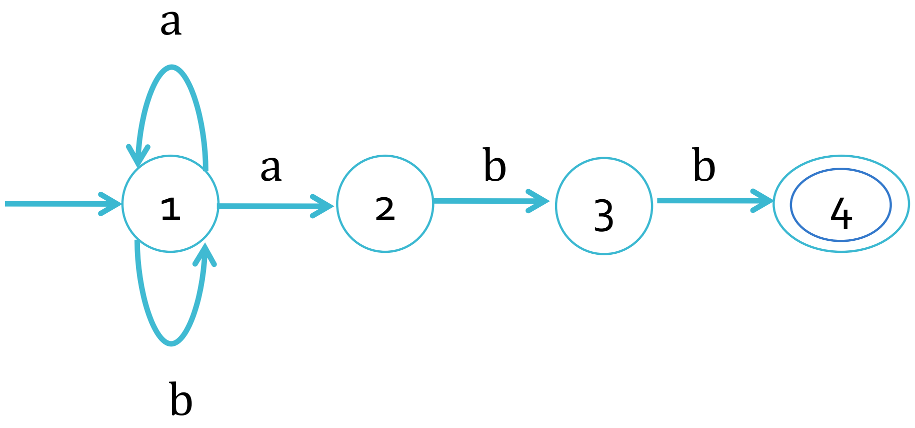
Finite Automata

- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Deterministic – faster recognizer, but it may take more space
- Non-deterministic – slower, but it may take less space
- Deterministic automatons are widely used lexical analysers.

- **Deterministic finite automata (DFA):** From each state **exactly one edge** leaving out (for each symbol).
- **Nondeterministic finite automata (NFA):** There are **no restrictions on the edges** leaving a state. There can be several with the same symbol as label and some edges can be labeled with ϵ.



DFA



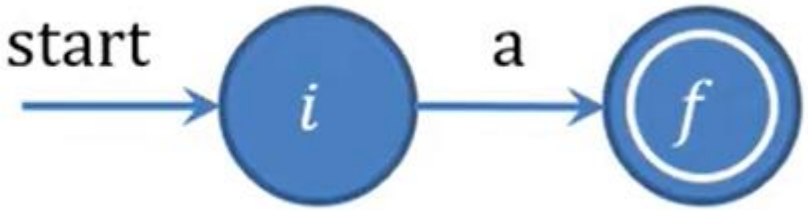
NFA

Regular Expression to NFA (Thompson’s Rule)

Base Cases (For NFA)

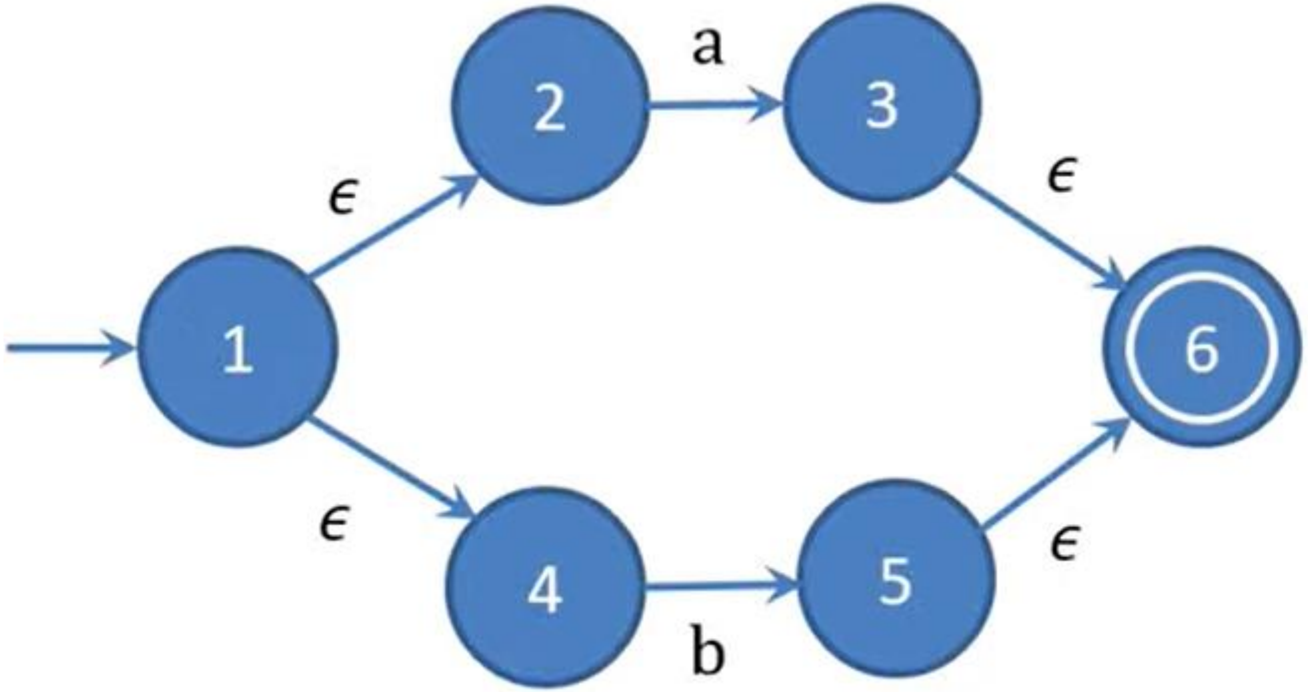
1. For ϵ , construct the NFA

2.For a in $\sum$, construct the NFA



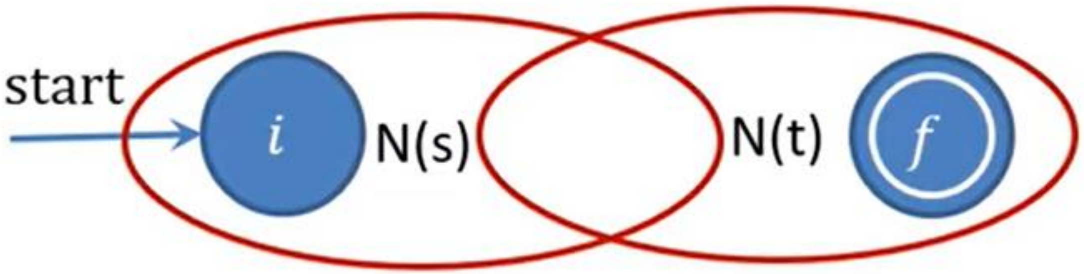
Base Cases (For NFA)

3.For a|b or (a+b)



Base Cases (For NFA)

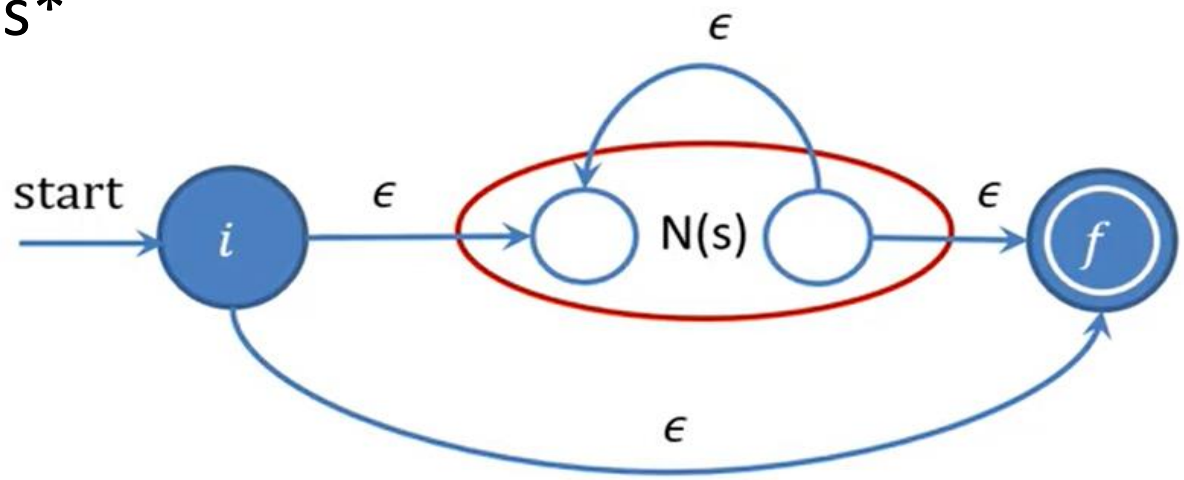
4. For st



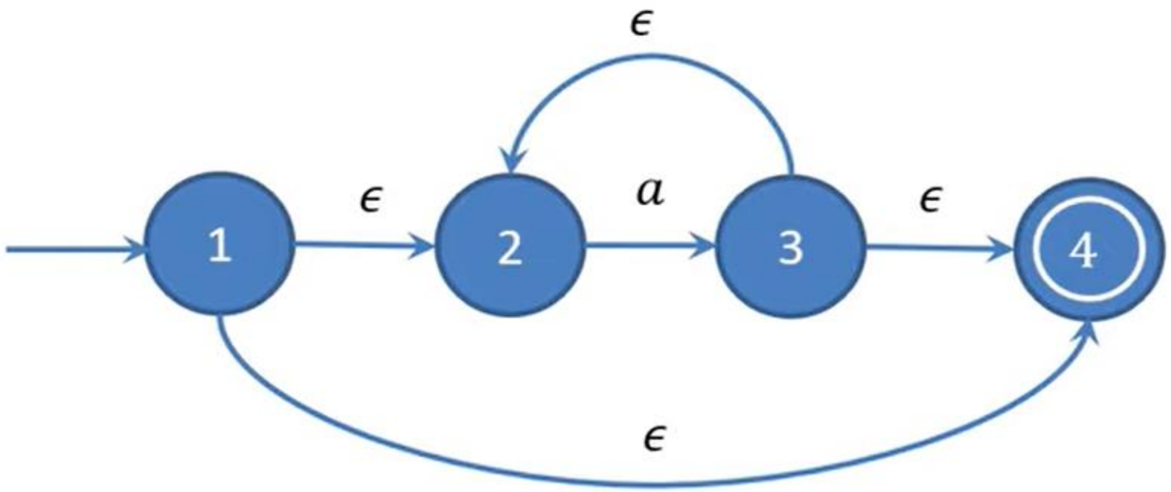
E.g ab

Construction for s^*

5. For s*

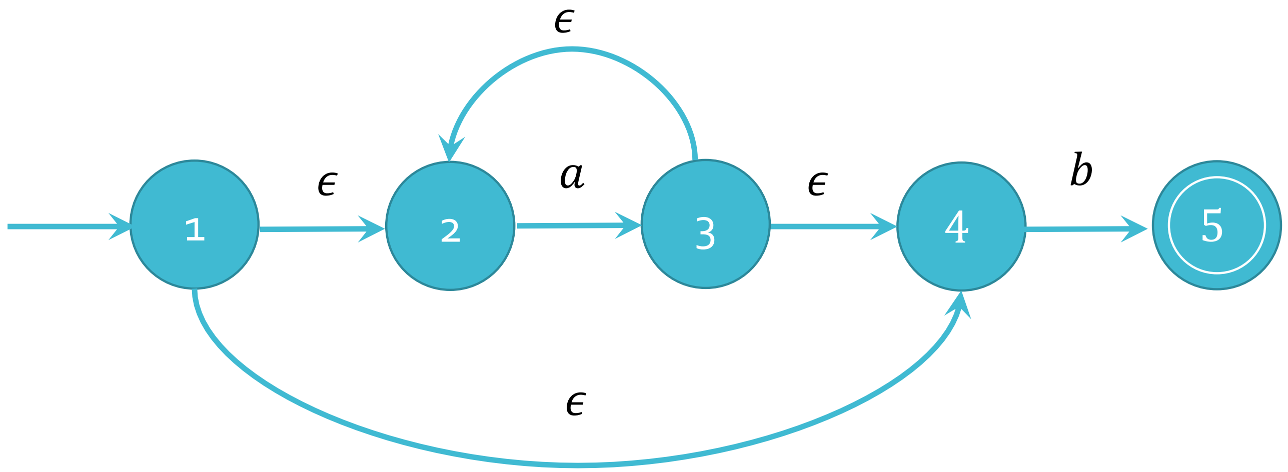


E.g a*

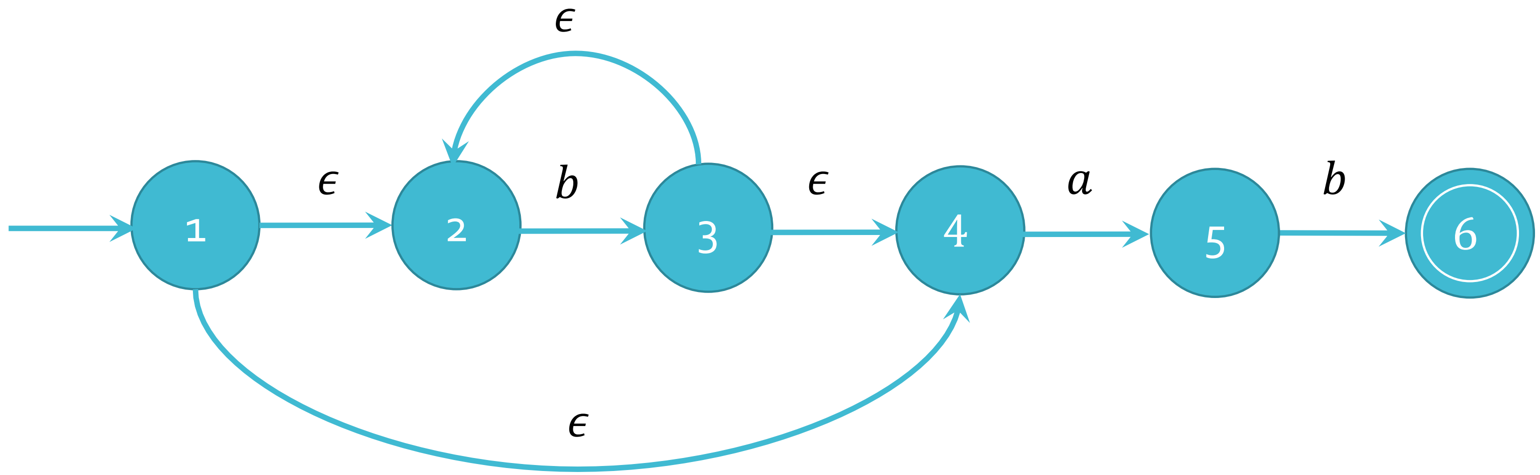


Example (RE to NFA)

a^*b

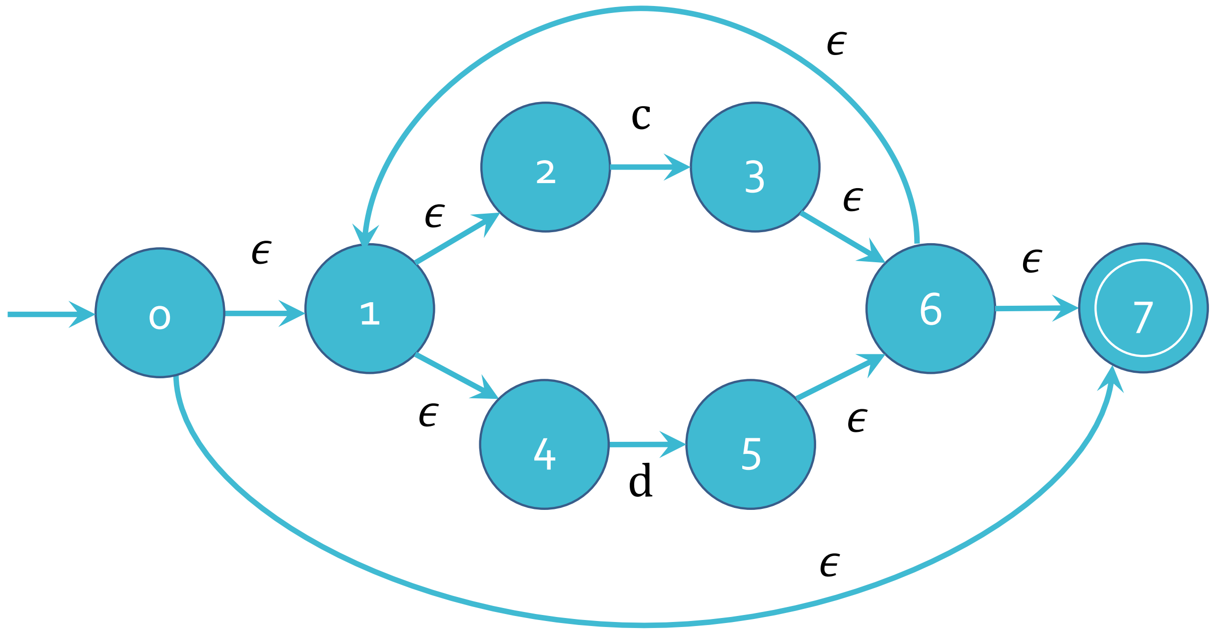


b^*ab



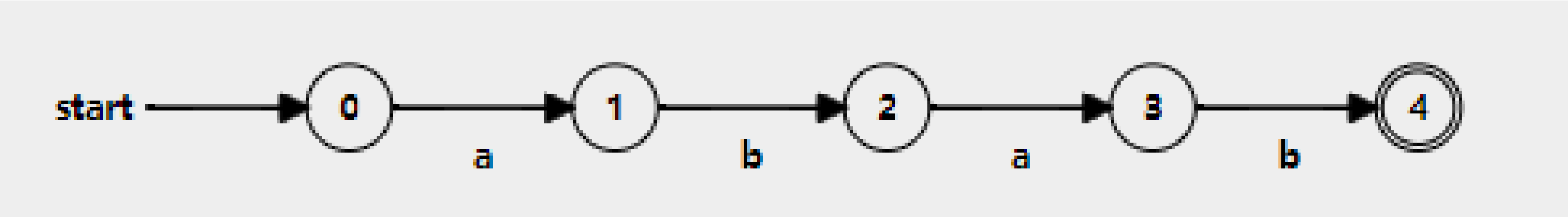
Example (RE to NFA)

(c|d)*

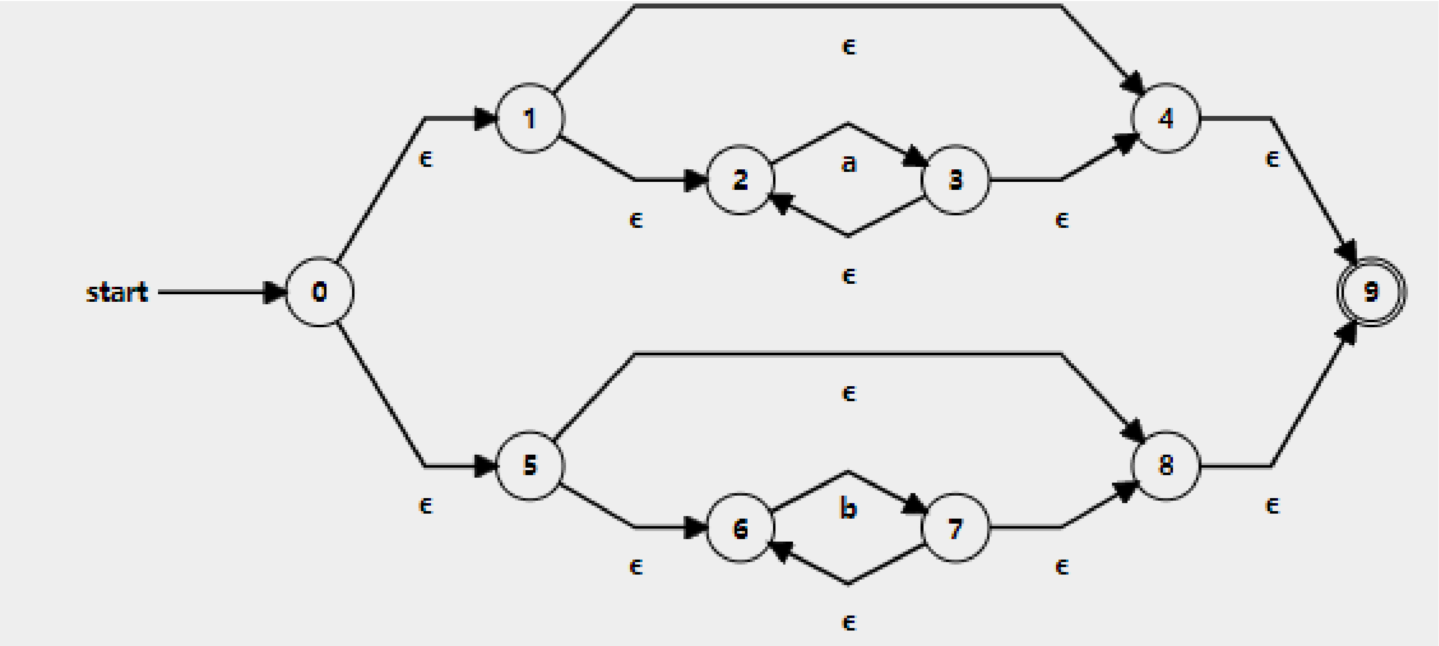


Exercise

- abab

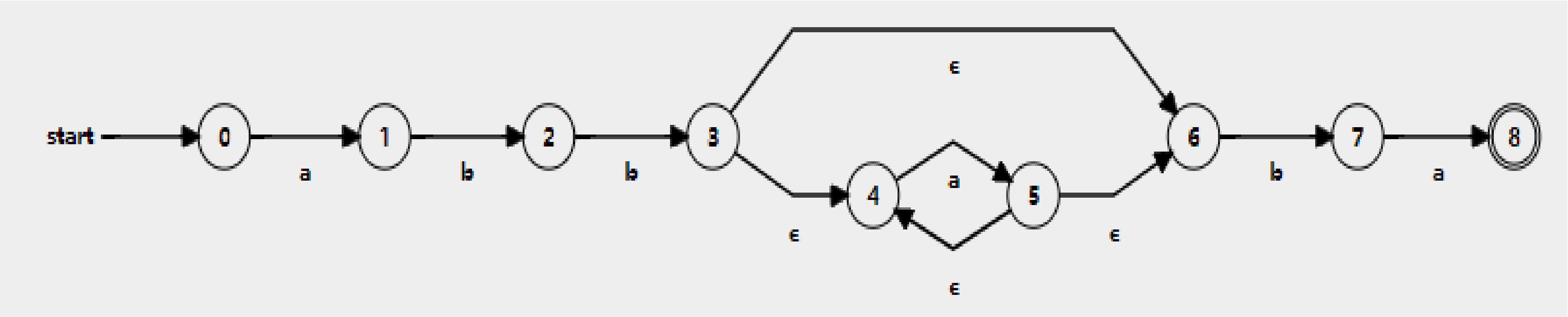


- a*|b*

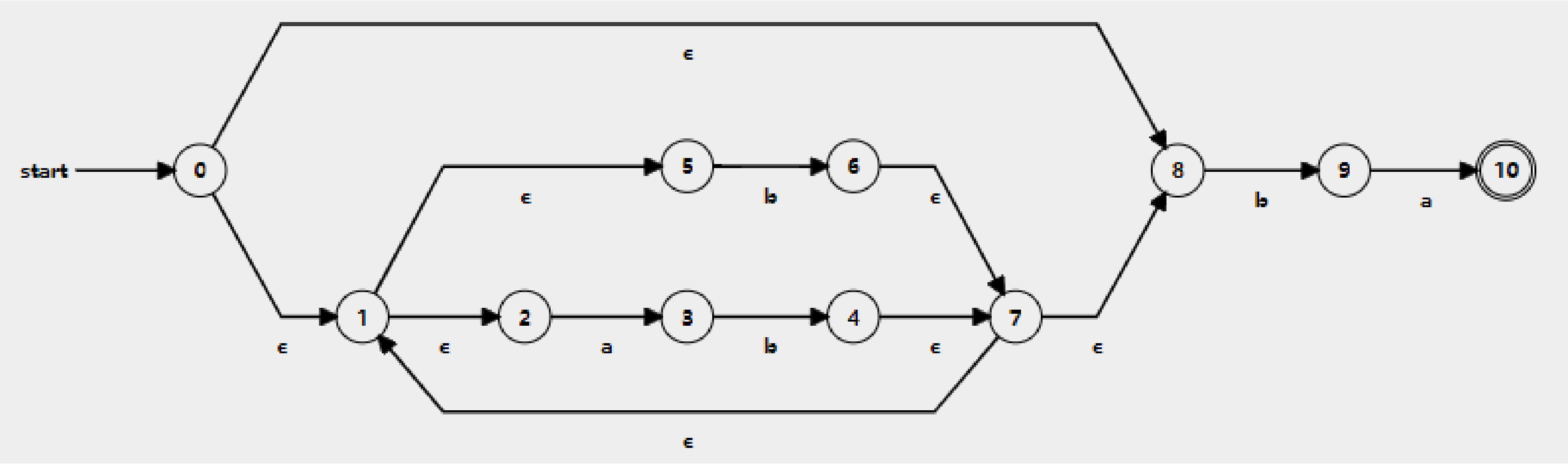


Exercise

- abb(a)*ba

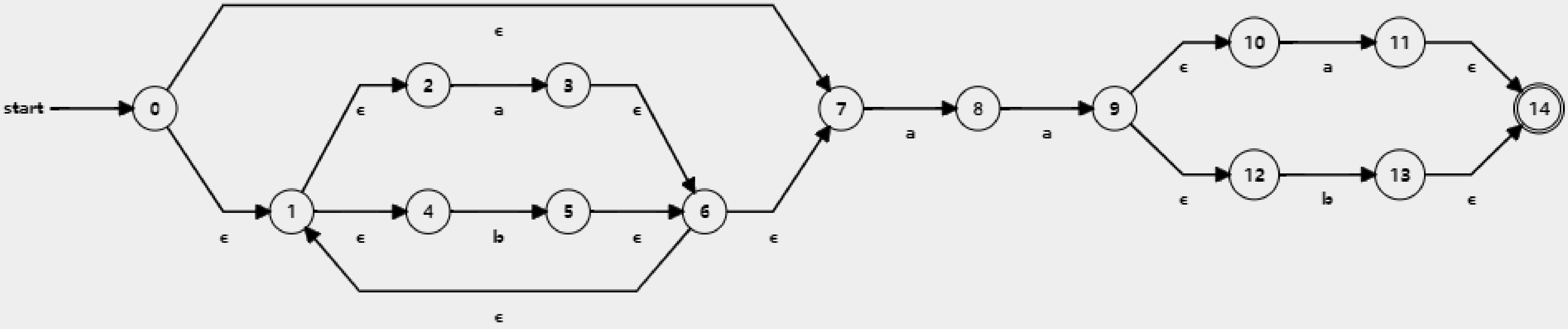


- (ab + b)*ba



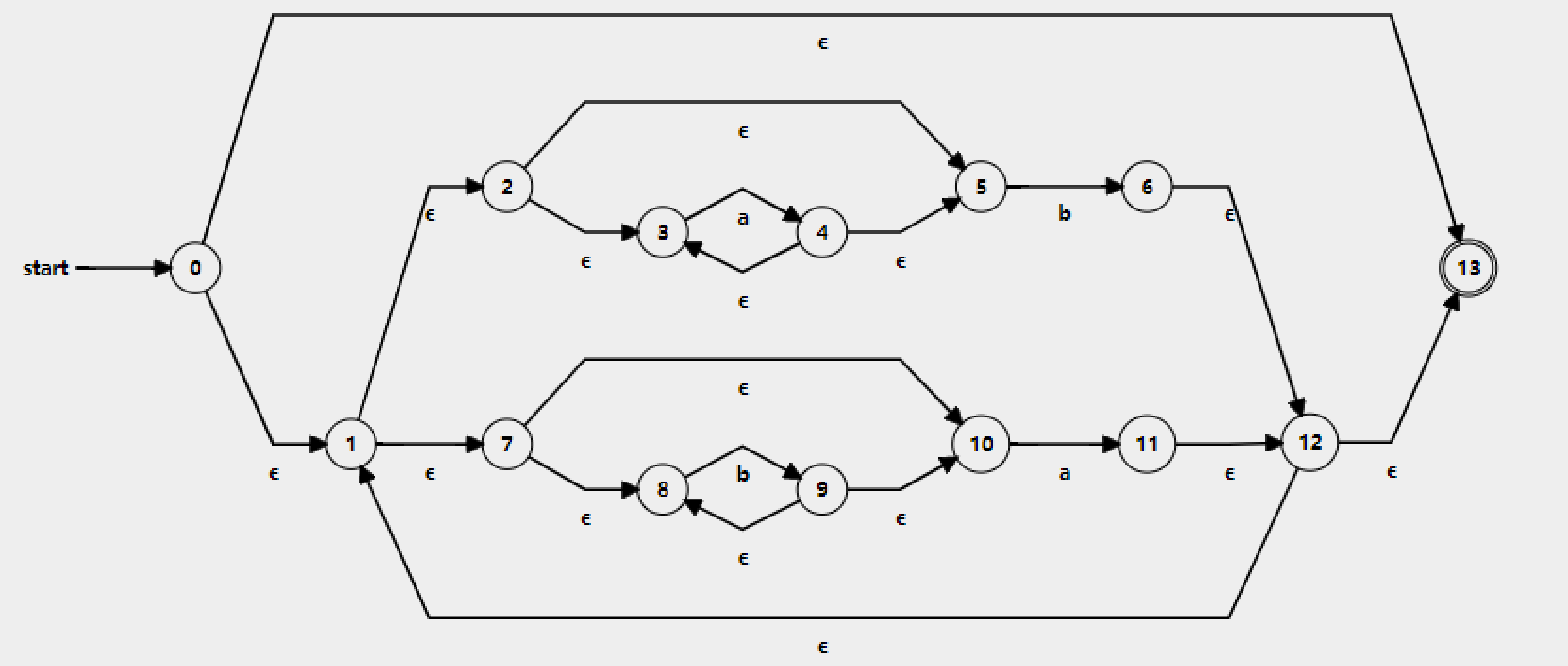
Exercise

- (a + b)*aa(a+b)



Exercise

- (a*b|b*a)*



Exercise

Convert following regular expression to NFA:

1. abba
2. bb(a)*
3. (a|b)*
4. a* | b*
5. a(a)*ab
6. aa*+ bb*
7. (a+b)*abb
8. 10(0+1)*1
9. (a+b)*a(a+b)
10. (0+1)*010(0+1)*
11. (010+00)*(10)*
12. 100(1)*00(0+1)*

NFA to DFA conversion
(Using subset construction method)

Subset Construction Algorithm

- Input : NFA (N)
- Output : DFA (D) (Accepting the same language)
- Method : Apply Algorithm, Make Tansition Table, Dtran for DFA .

Operations to perform:

Operation	Description
Є – closure(s)	Set of NFA States reachable from NFA State s on Є – transition alone.
Є – closure(T)	Set of NFA States reachable from some NFA State s in T on Є –transition alone.
Move (T,a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Subset Construction Algorithm

Initially Є –closure (s0) be the only state in Dstates and it is unmarked;

While there is unmarked states in T in Dstates ***do begin***

Mark T;

for each input symbol a ***do begin***

U = Є –closure (move (T,a));

If *U is not in Dstates* ***then***

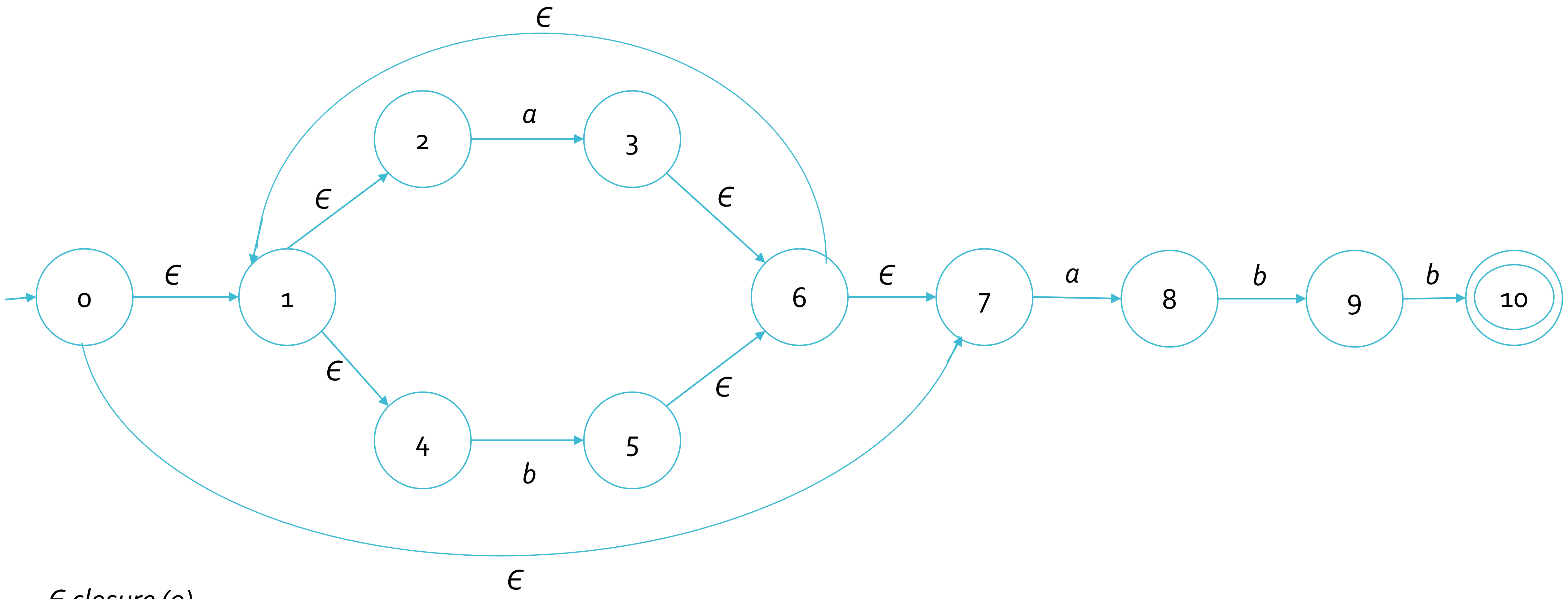
add U as unmarked state to Dstates;

Dtran [T, a] = U

end

end

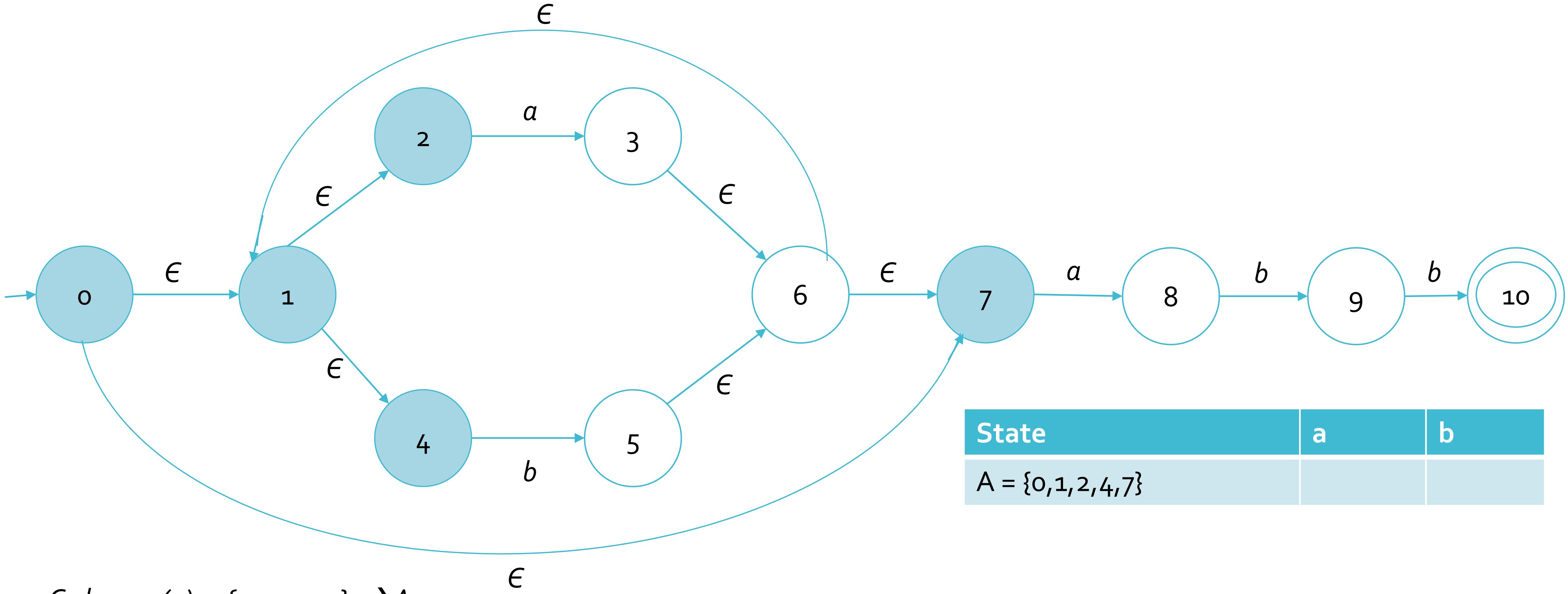
Example : (a|b)*abb



ϵ closure (0)

{0,1,2,4,7} → A

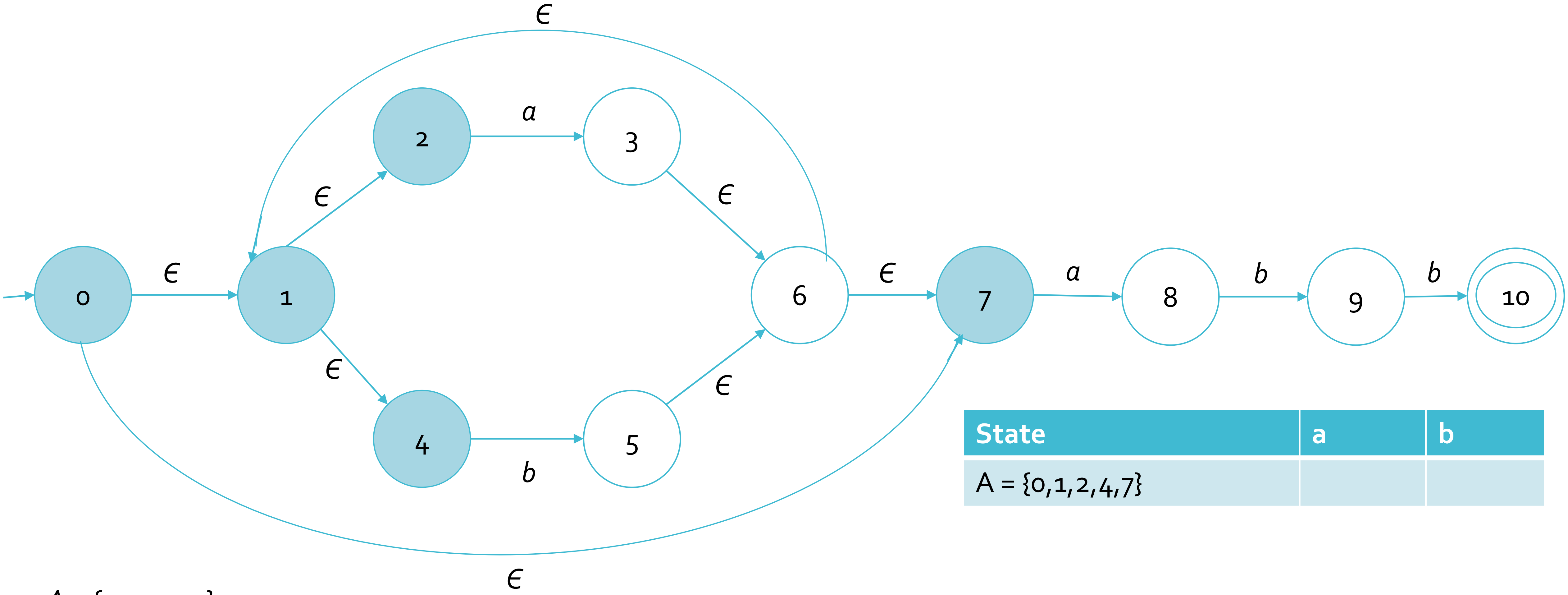
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}		

ϵ closure (0) = {0,1,2,4,7} → *A*

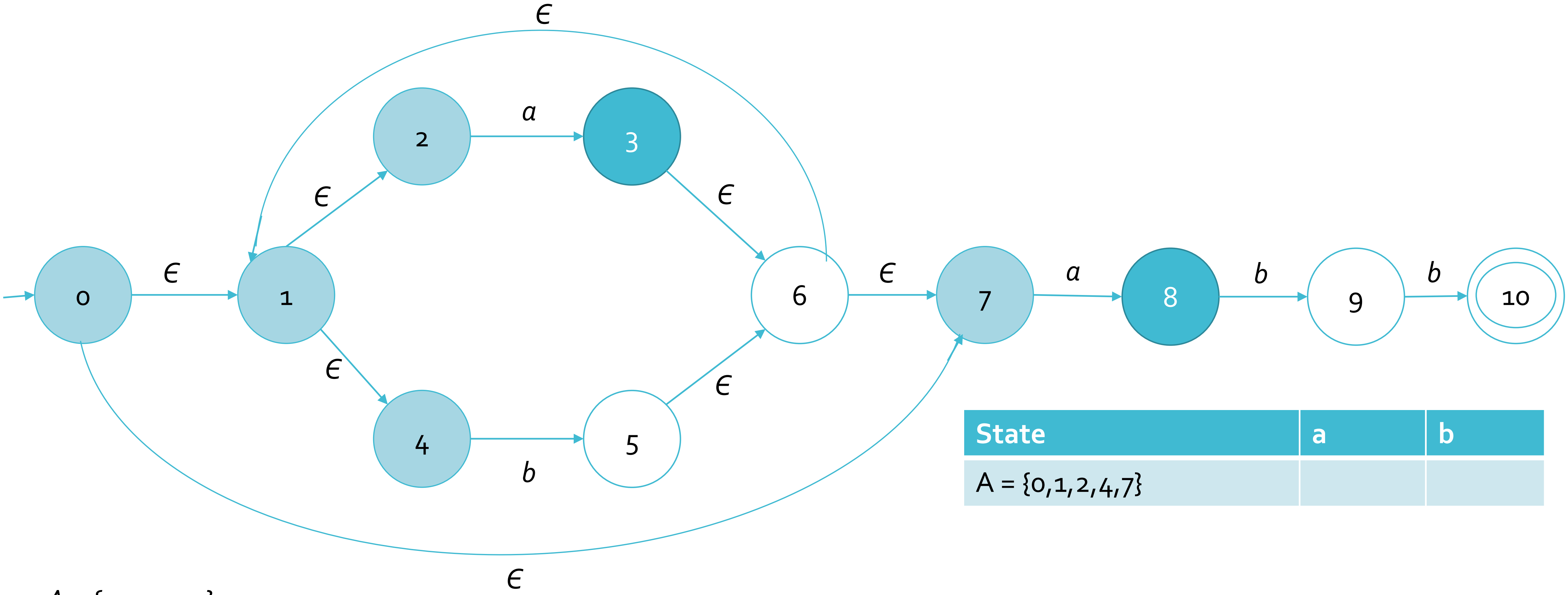
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}		

A = {0,1,2,4,7}
Move (A,a) = {3,8}

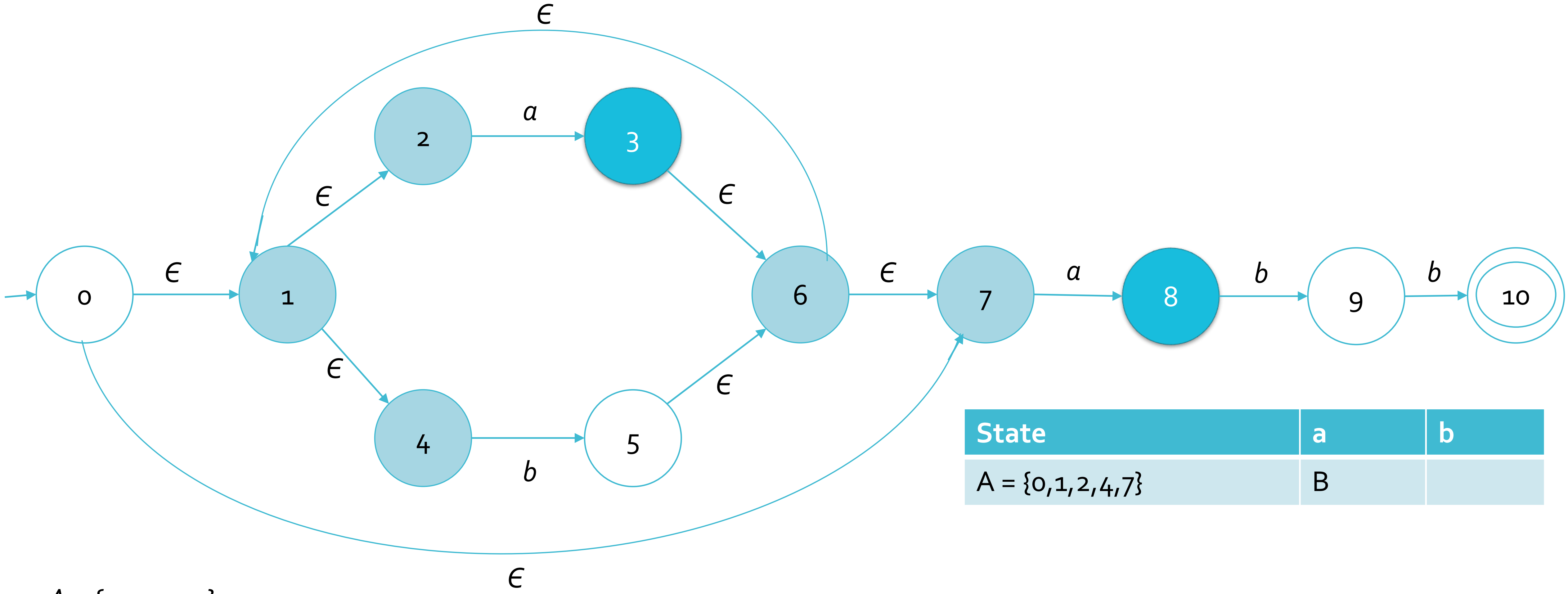
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}		

A = {0,1,2,4,7}
Move (A,a) = {3,8}

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	

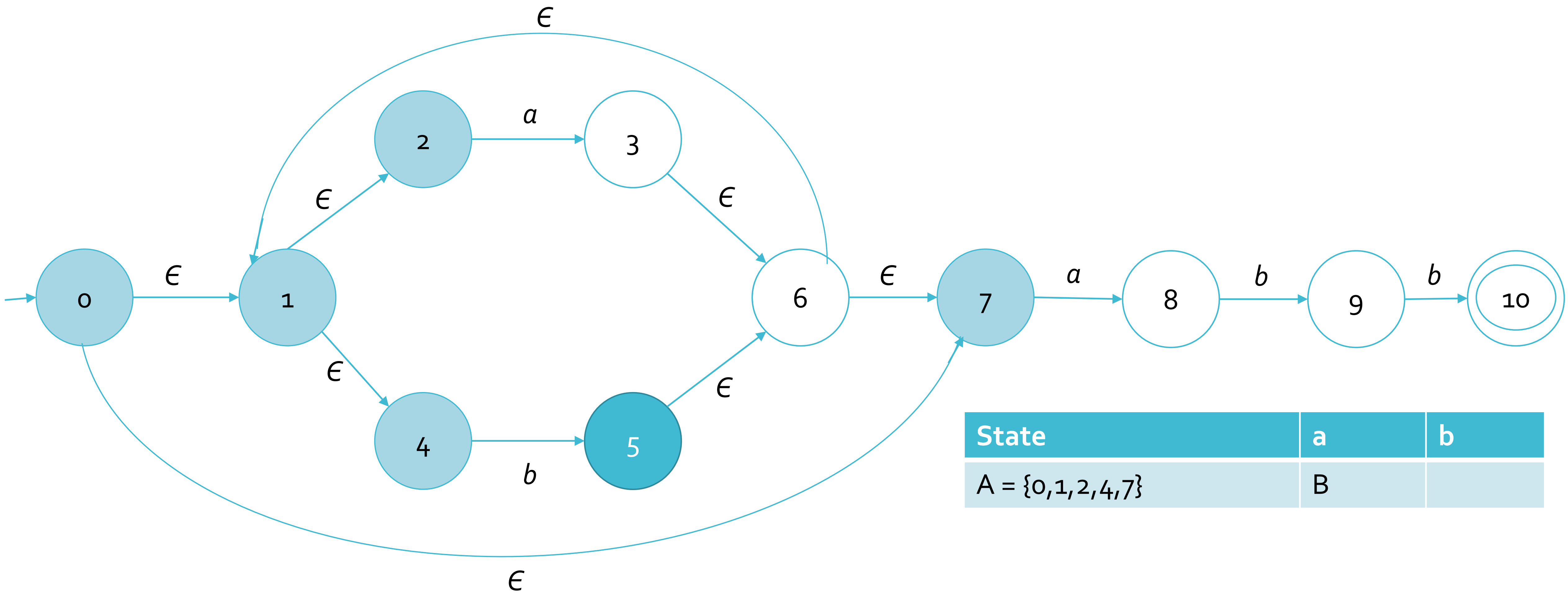
A = {0,1,2,4,7}

Move (A,a) = {3,8}

ϵ closure (Move(A,a)) = {3,6,7,1,2,4,8}

= {1,2,3,4,6,7,8} → B

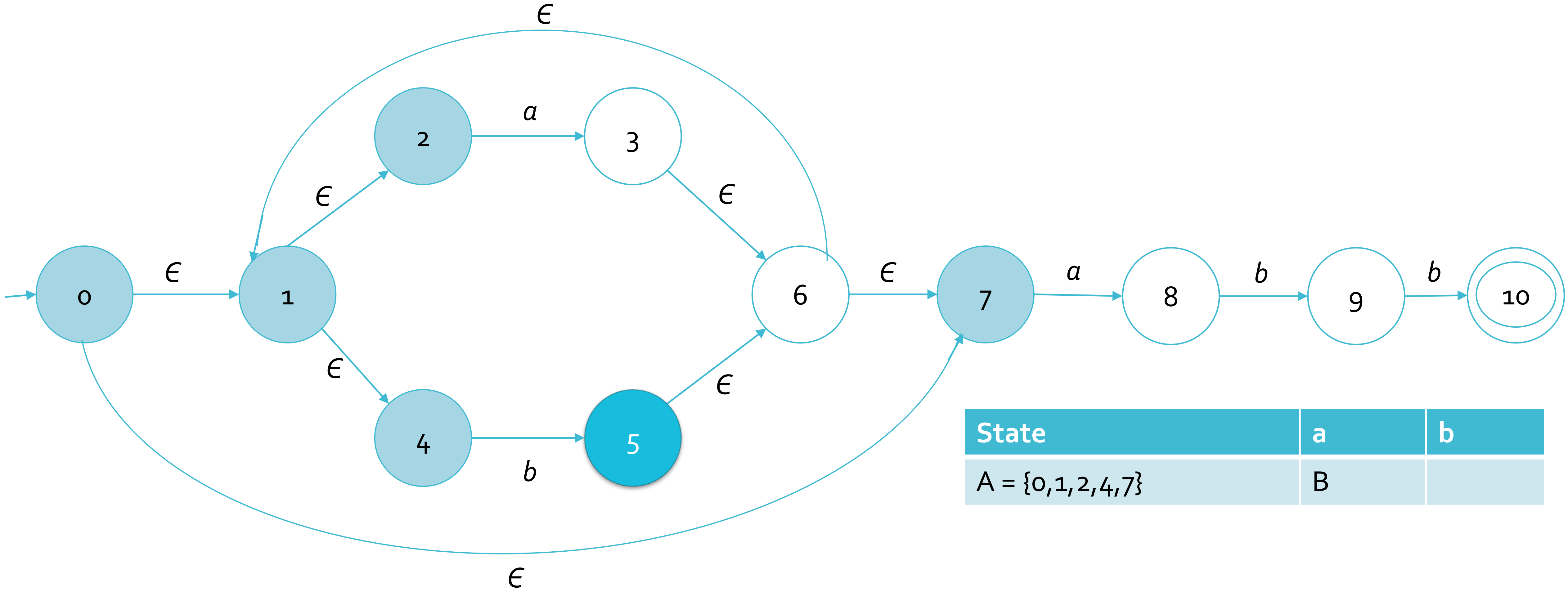
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	

A = {0,1,2,4,7}
Move (A,b) = {5}

Example : (a|b)*abb

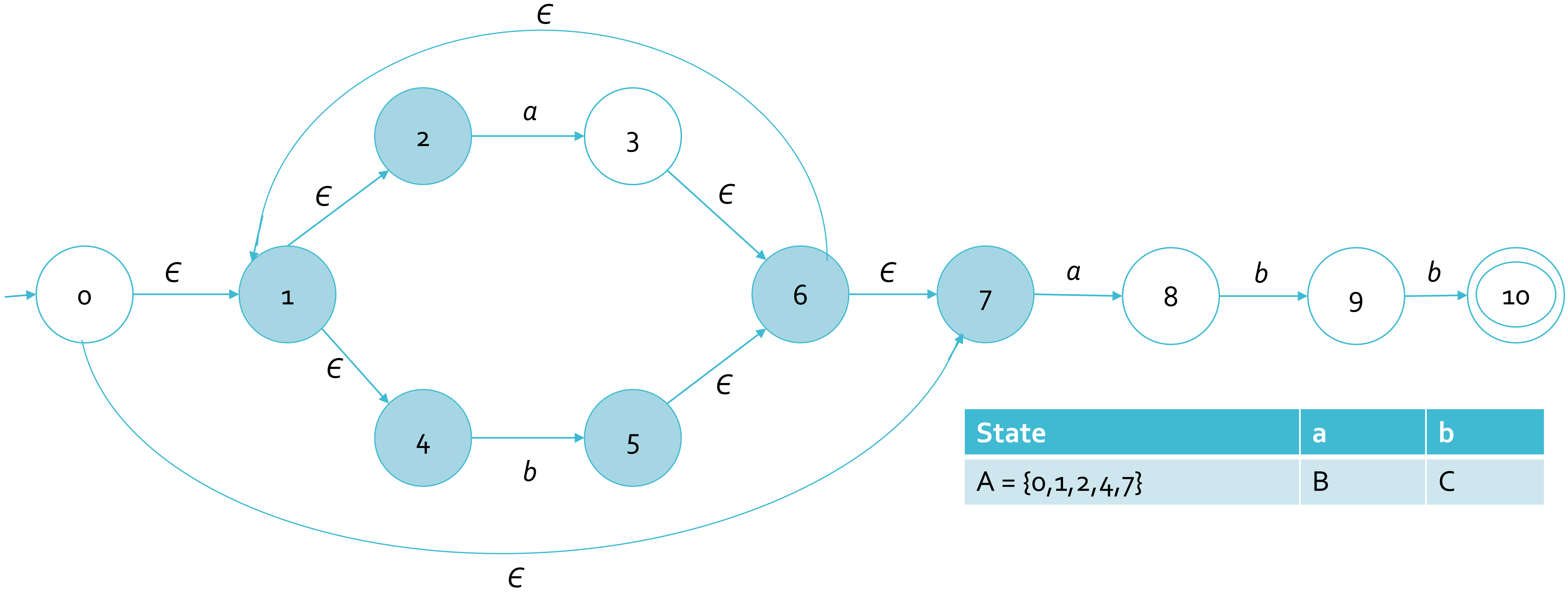


State	a	b
A = {0,1,2,4,7}	B	

A = {0,1,2,4,7}
Move (A,b) = {5}

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

Example : (a|b)*abb

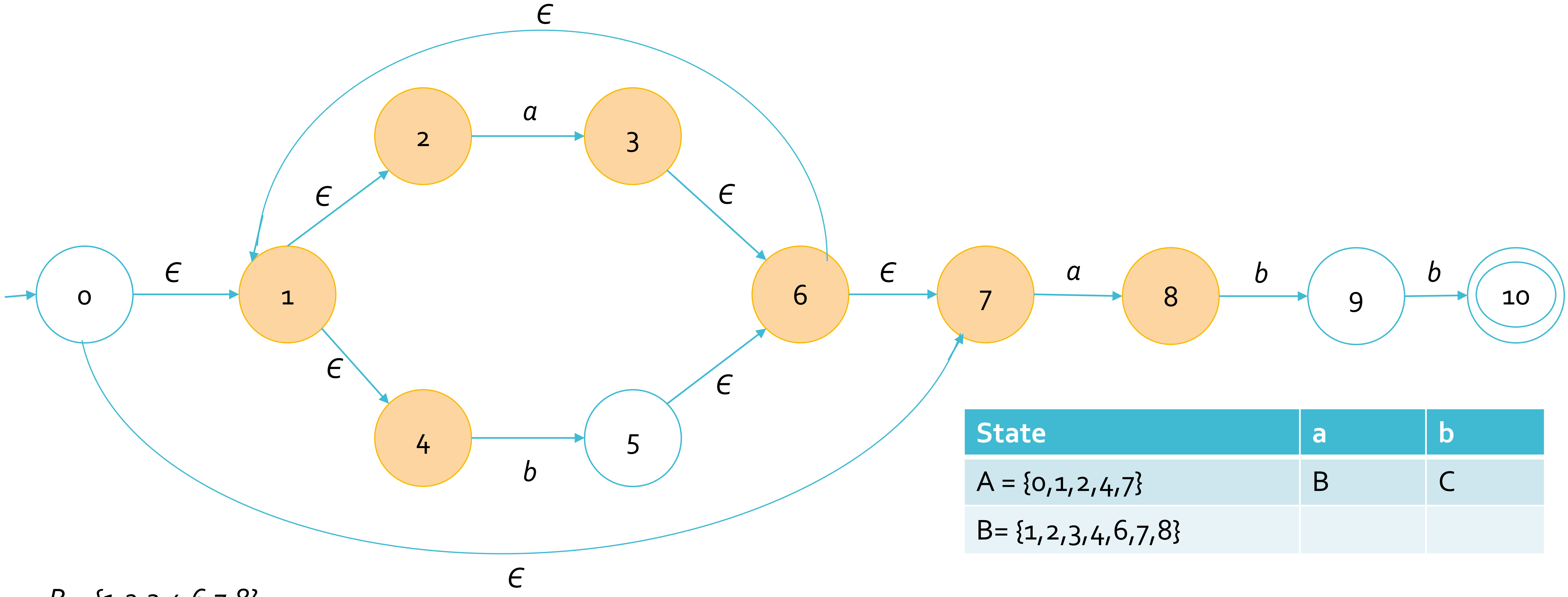


State	a	b
A = {0,1,2,4,7}	B	C

A = {0,1,2,4,7}
Move (A,b) = {5}

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

Example : (a|b)*abb

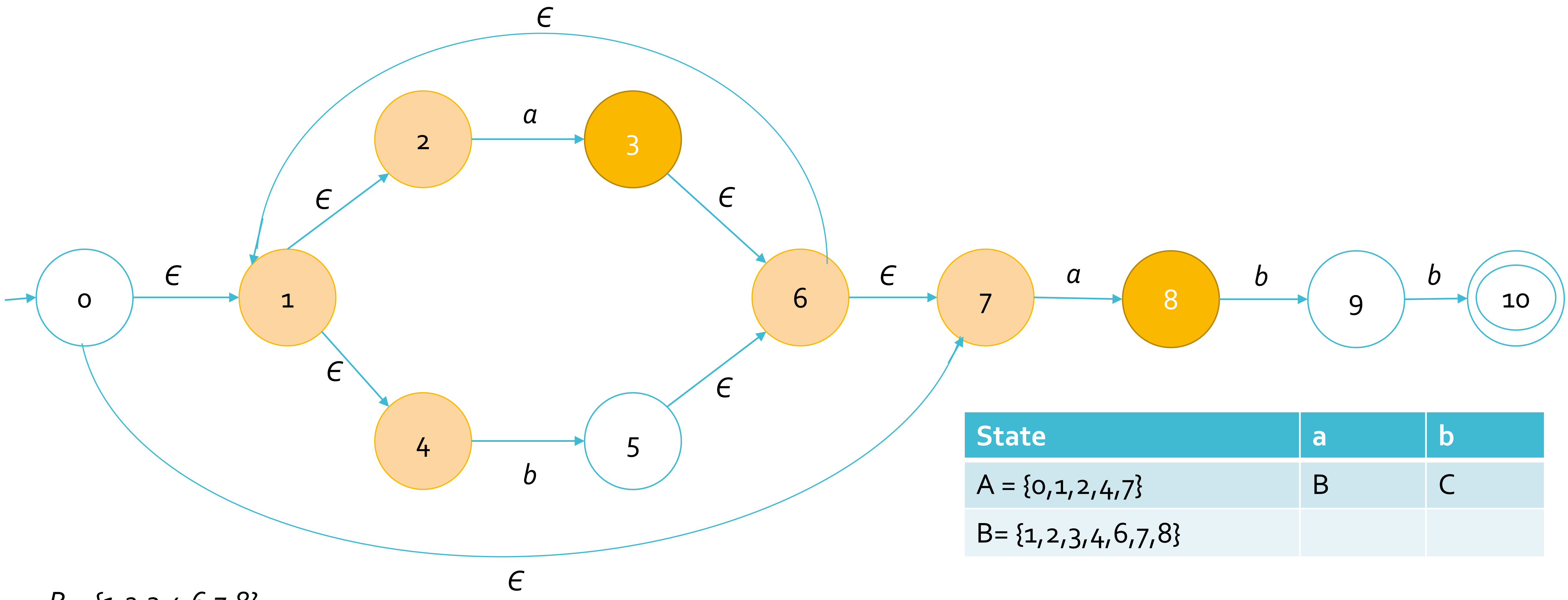


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}		

B = {1,2,3,4,6,7,8}

Move (B,a) = {3,8}

Example : (a|b)*abb

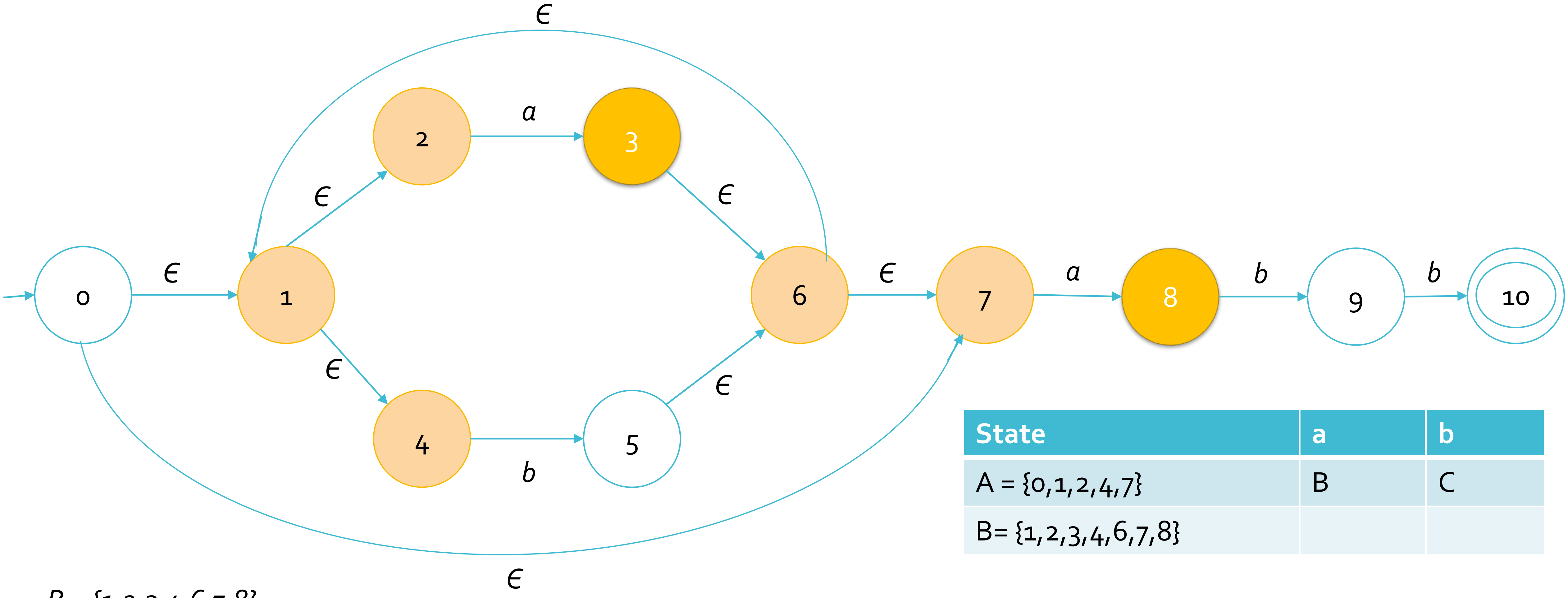


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}		

B = {1,2,3,4,6,7,8}

Move (B,a) = {3,8}

Example : (a|b)*abb

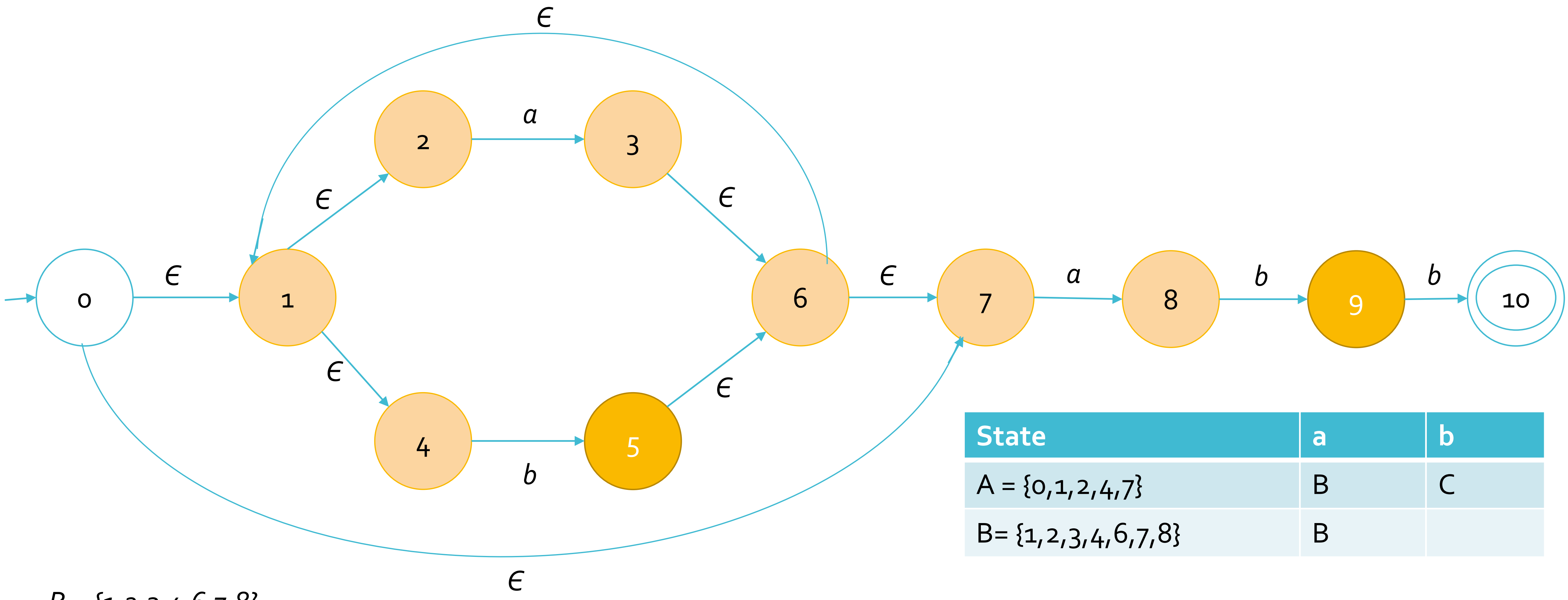


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}		

B = {1,2,3,4,6,7,8}
Move (B,a) = {3,8}

ϵ Closure (Move (B,a)) = { 3,6,7,1,2,4,8}
= {1,2,3,4,6,7,8} → B

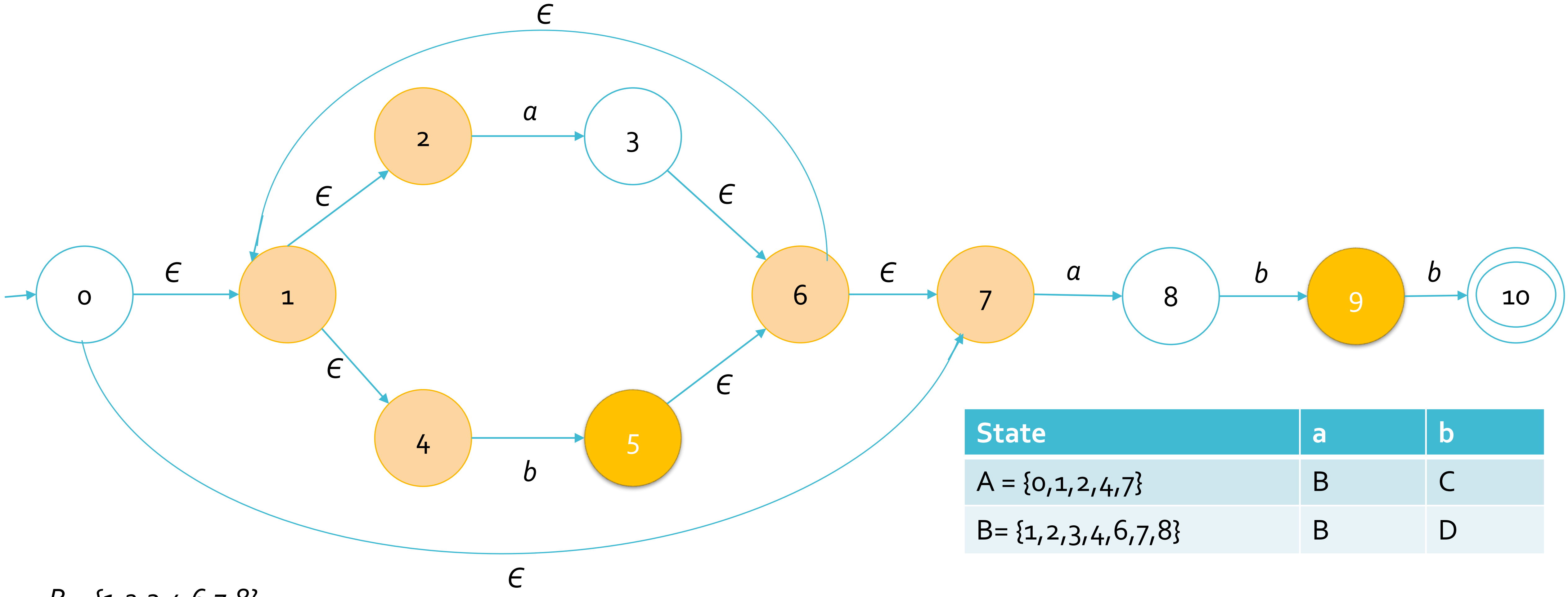
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	

B = {1,2,3,4,6,7,8}
Move (B,b) = {5,9}

Example : (a|b)*abb

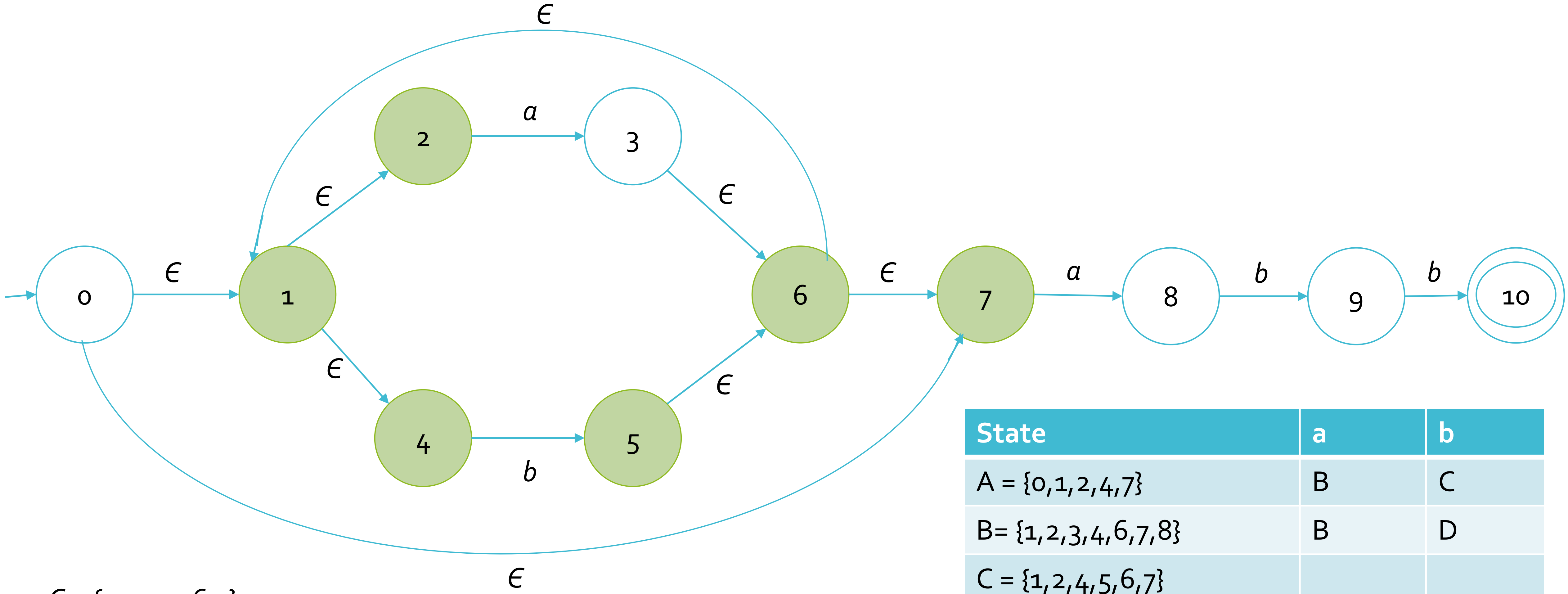


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D

B = {1,2,3,4,6,7,8}
Move (B,b) = {5,9}

ϵ Closure (Move (B,b)) = {5,6,7,1,2,4,9}
= {1,2,4,5,6,7,9} → D

Example : (a|b)*abb

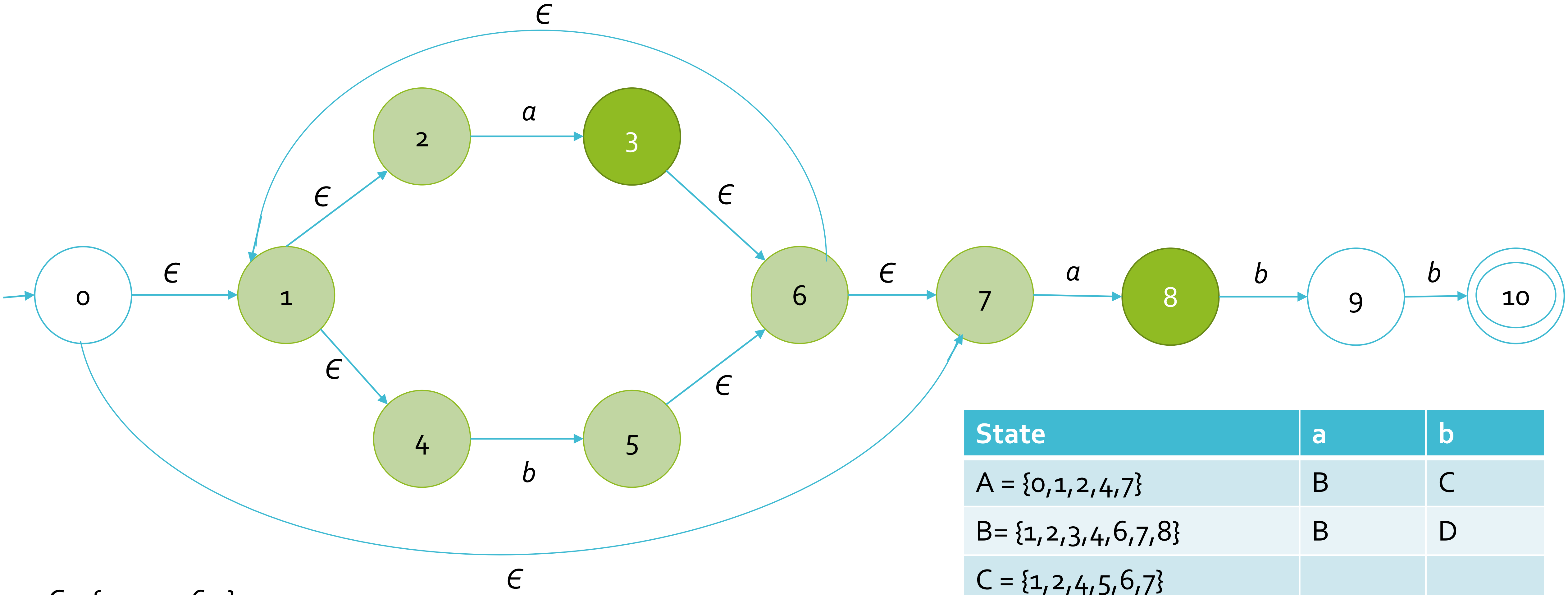


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}		

C = {1,2,4,5,6,7}

Move (C,a) = {3,8}

Example : (a|b)*abb

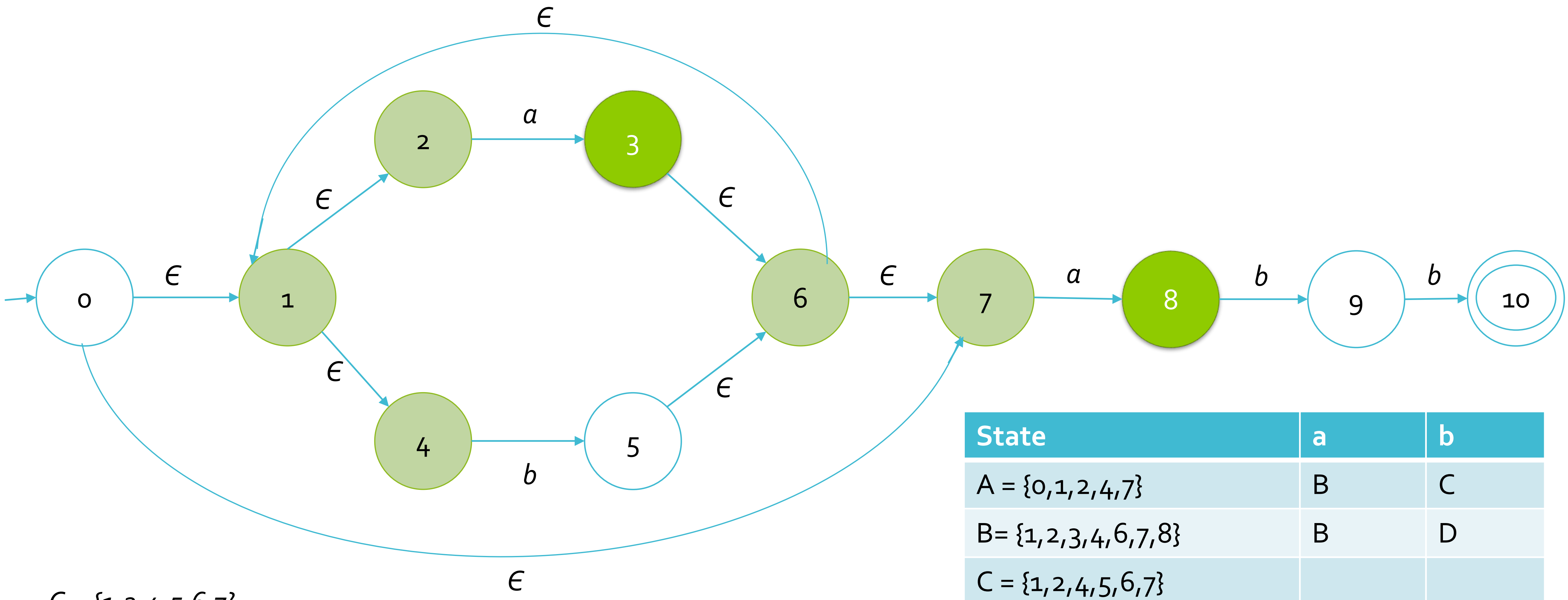


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}		

C = {1,2,4,5,6,7}

Move (C,a) = {3,8}

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}		

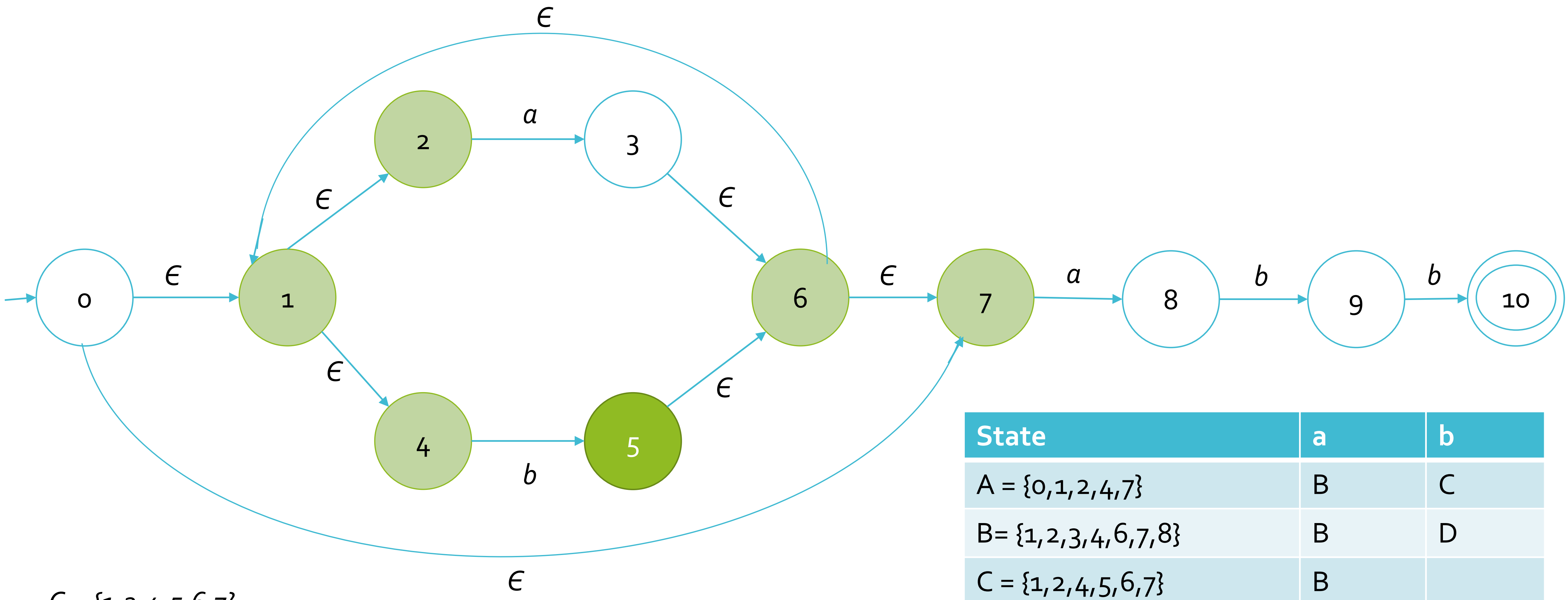
C = {1,2,4,5,6,7}

Move (C,a) = {3,8}

ϵ Closure (Move (c,a)) = {3,6,7,1,2,4,8}

= {1,2,3,4,6,7,8} → B

Example : (a|b)*abb

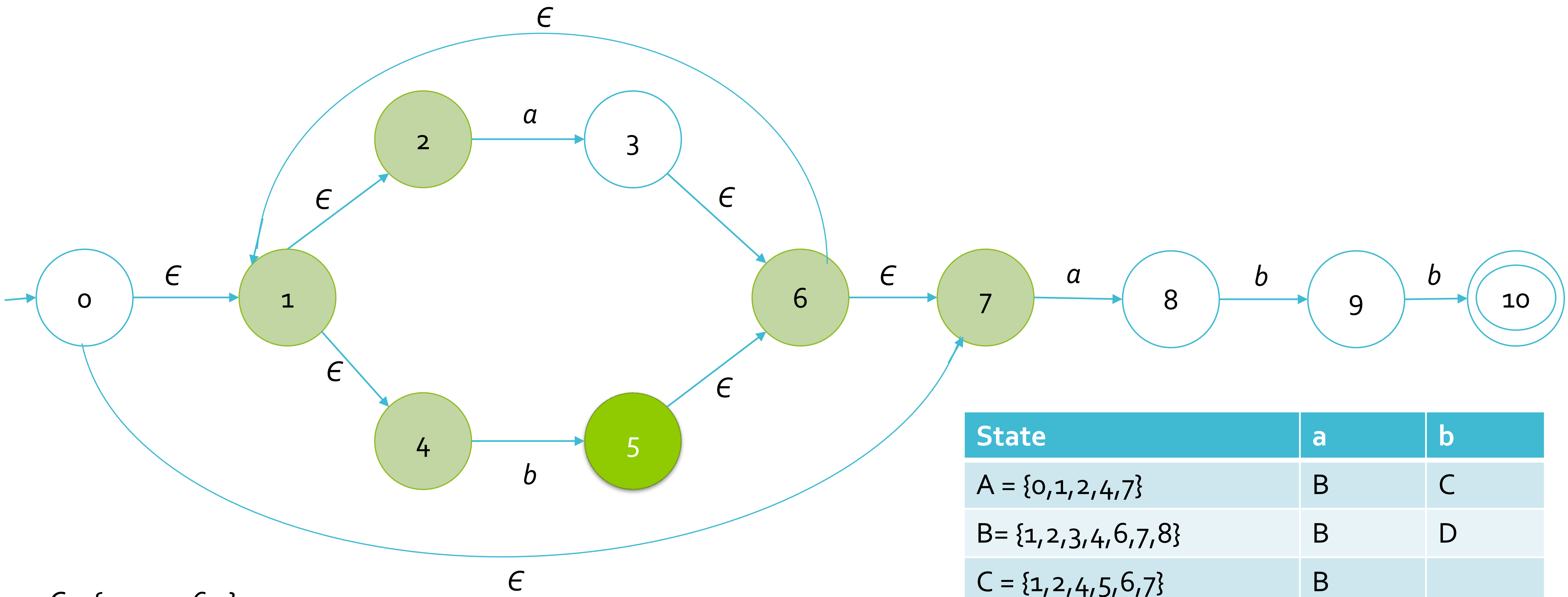


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	

C = {1,2,4,5,6,7}

Move (C,b) = {5}

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	

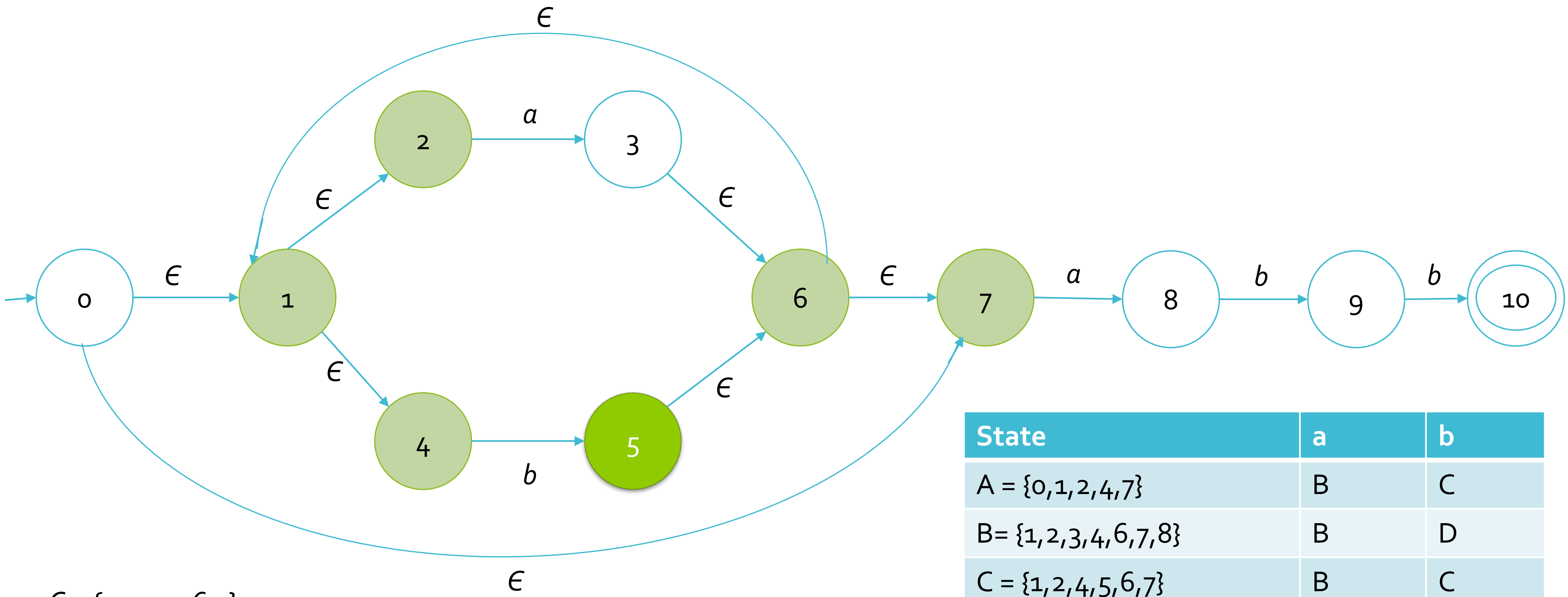
C = {1,2,4,5,6,7}

Move (C,b) = {5}

ϵ Closure (Move (C,b)) = {5,6,7,1,2,4}

= {1,2,4,5,6,7} → C

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C

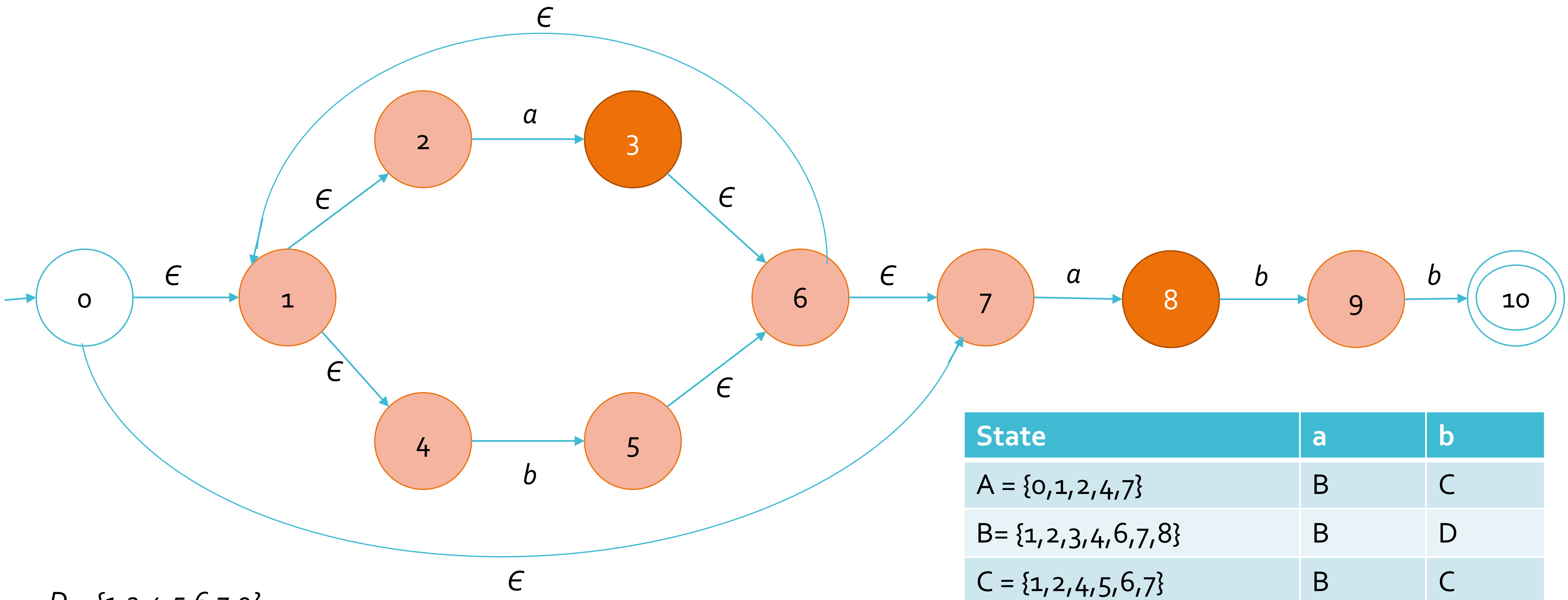
C = {1,2,4,5,6,7}

Move (C,b) = {5}

ϵ Closure (Move (C,b)) = {5,6,7,1,2,4}

= {1,2,4,5,6,7} → C

Example : (a|b)*abb

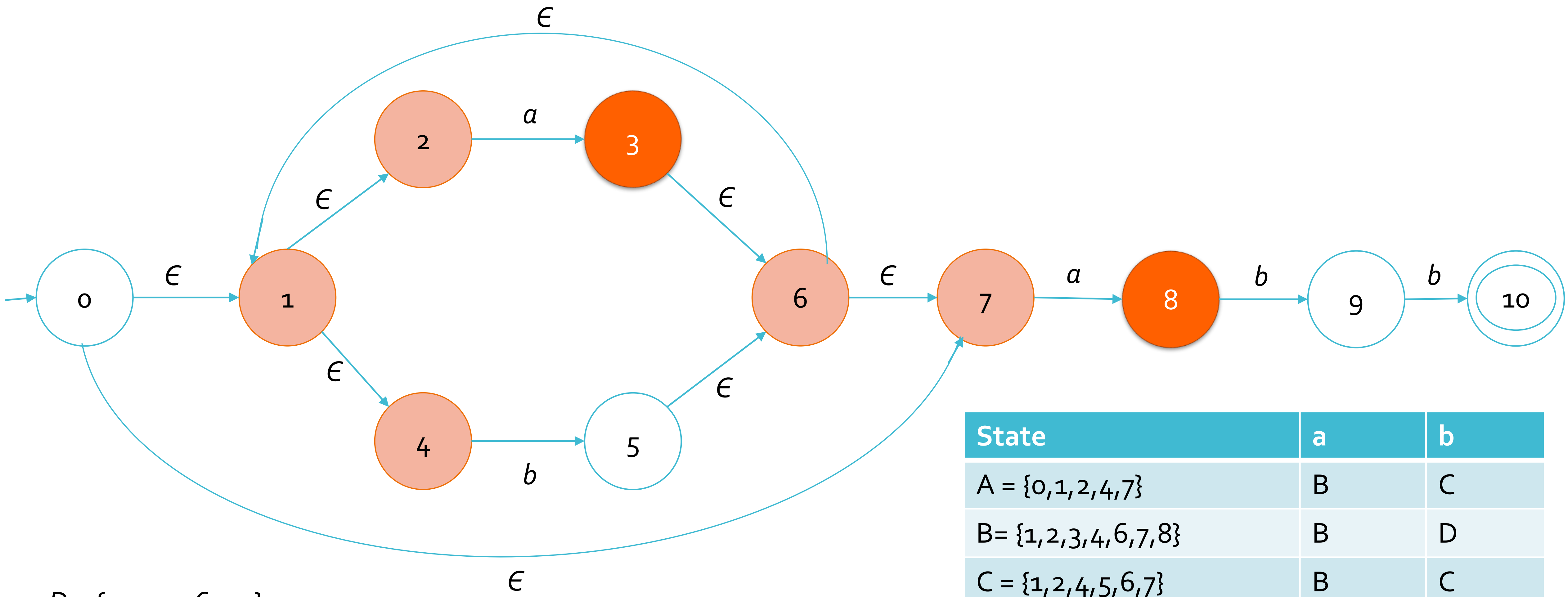


State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}		

D = {1,2,4,5,6,7,9}

Move (D,a) = {3,8}

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}		

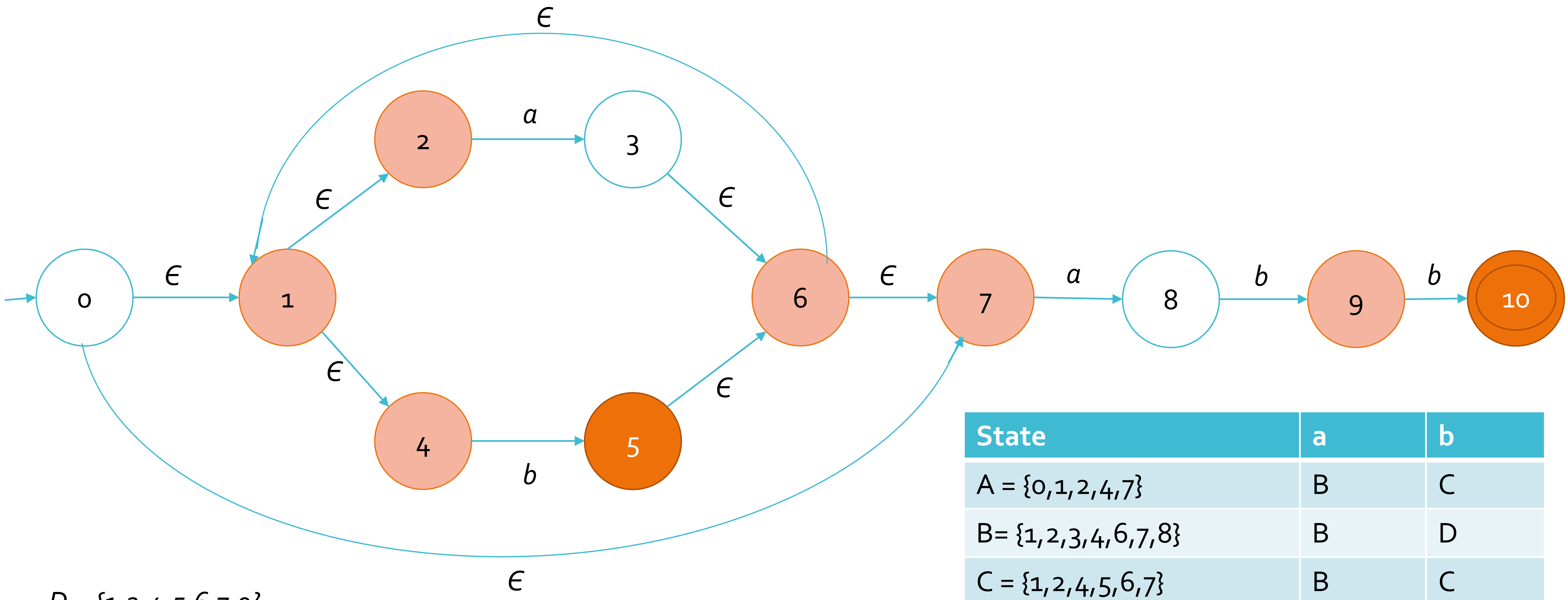
D = {1,2,4,5,6,7,9}

Move (D,a) = {3,8}

ϵ Closure (Move (D,a)) = {3,6,7,1,2,4,8}

= {1,2,3,4,6,7,8} → B

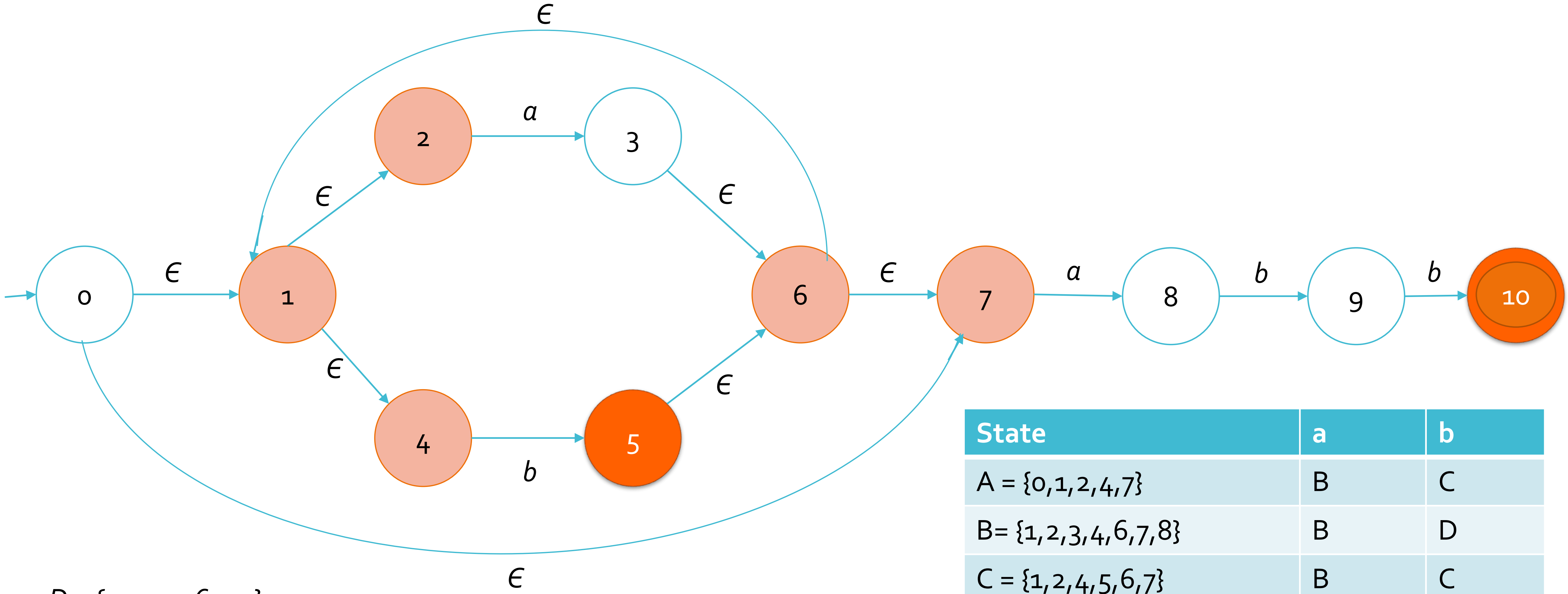
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	

D = {1,2,4,5,6,7,9}
Move (D,b) = {5,10}

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	

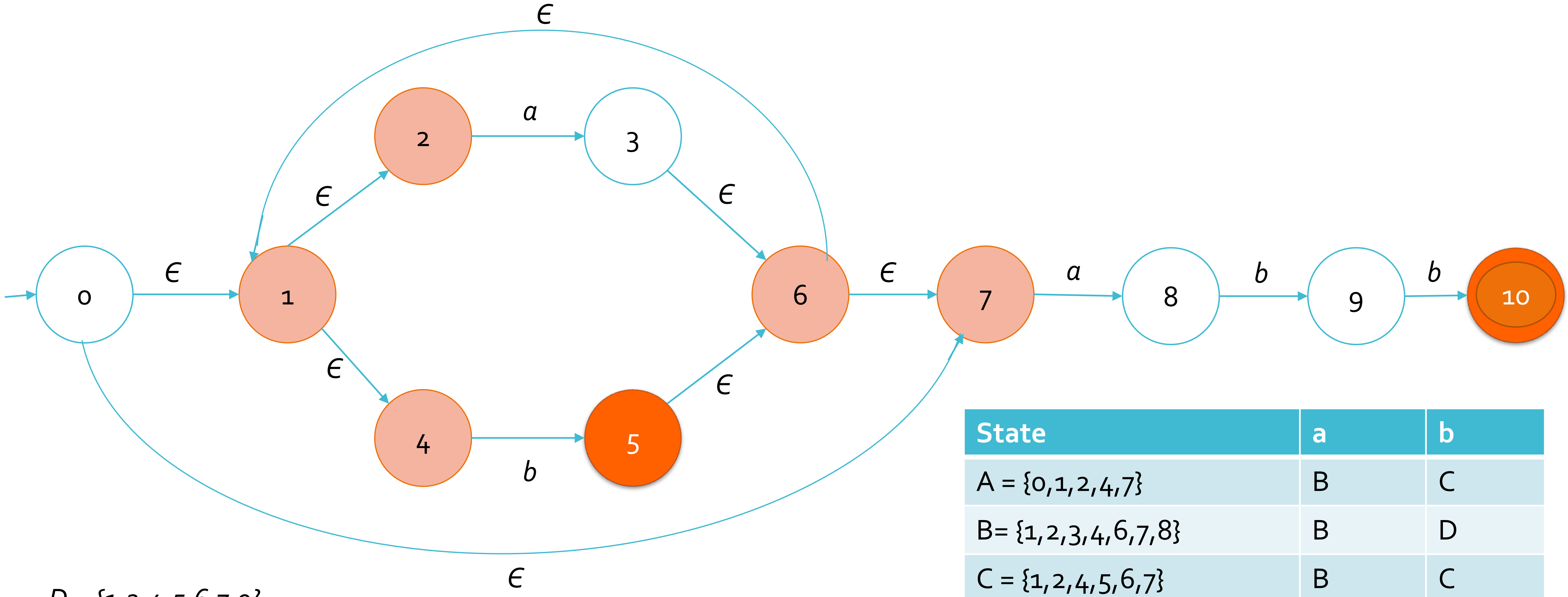
D = {1,2,4,5,6,7,9}

Move (D,b) = {5,10}

ϵ Closure (Move (D,b)) = {5,6,7,1,2,4,10}

= {1,2,4,5,6,7,10} → E

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E

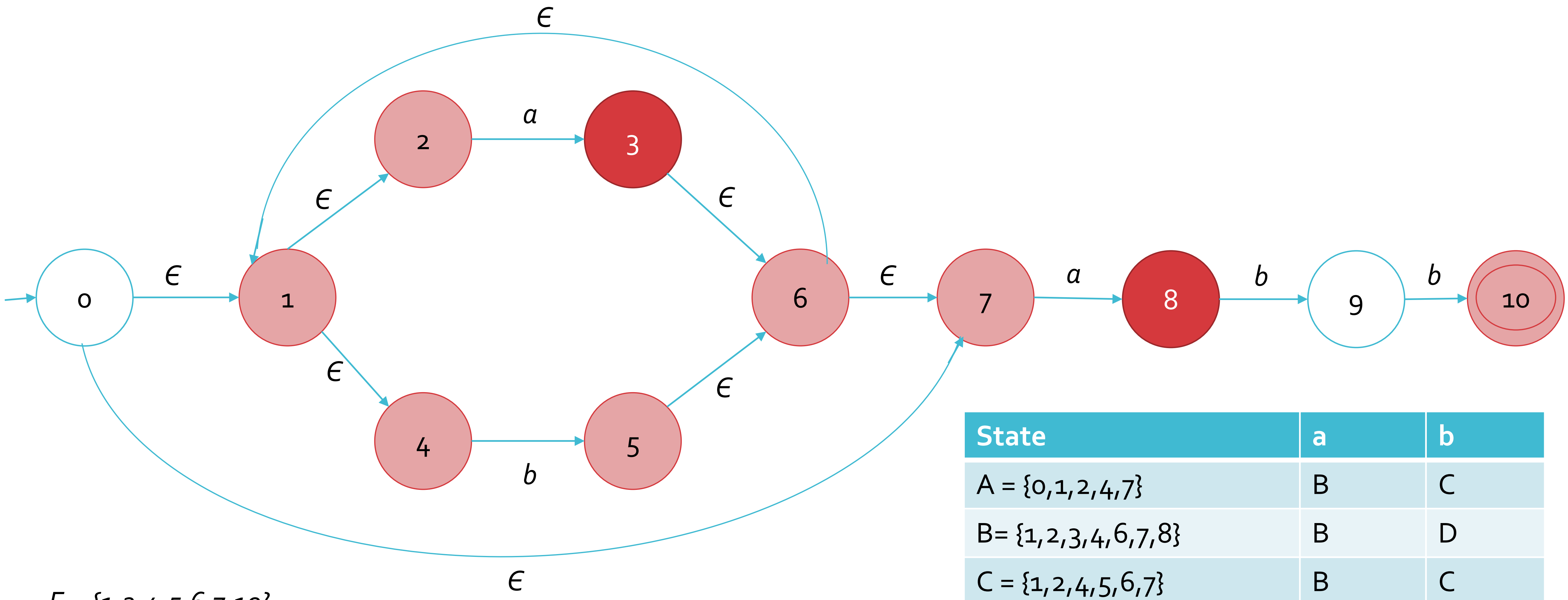
D = {1,2,4,5,6,7,9}

Move (D,b) = {5,10}

ϵ Closure (Move (D,b)) = {5,6,7,1,2,4,10}

= {1,2,4,5,6,7,10} → E

Example : (a|b)*abb

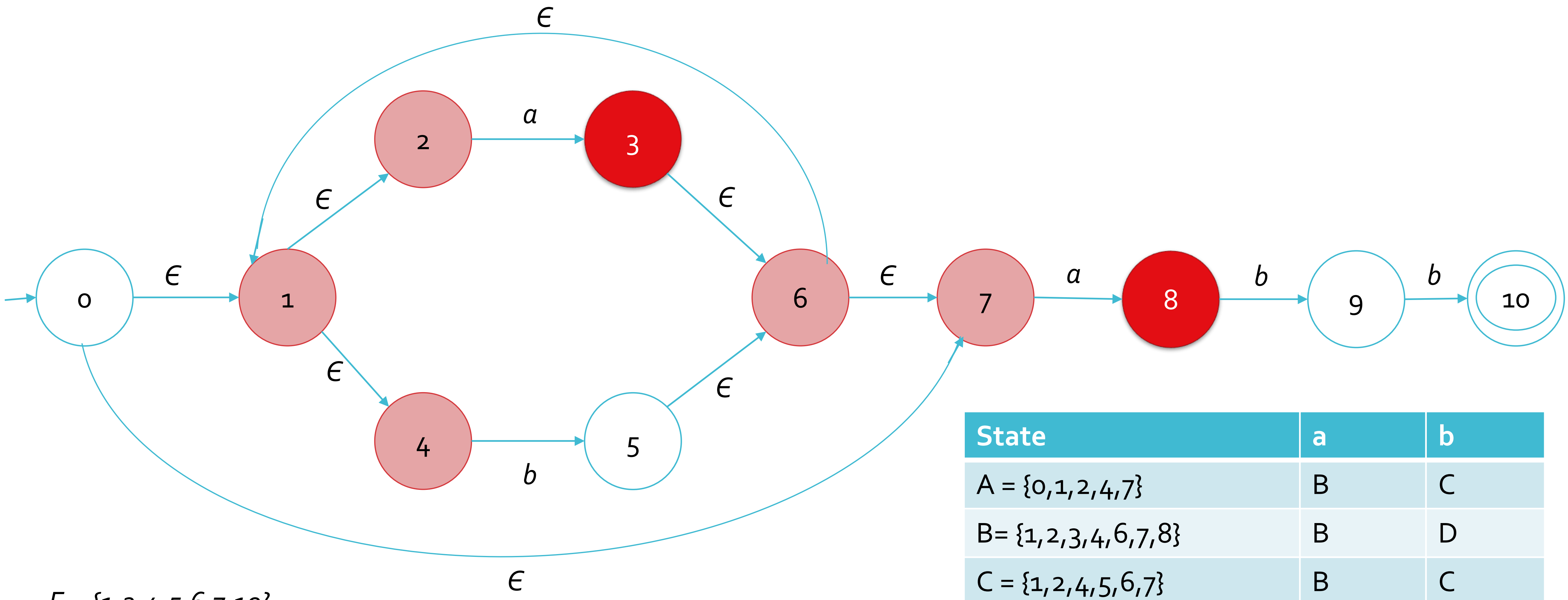


E = {1,2,4,5,6,7,10}

Move (E,a) = {3,8}

State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}		

Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}		

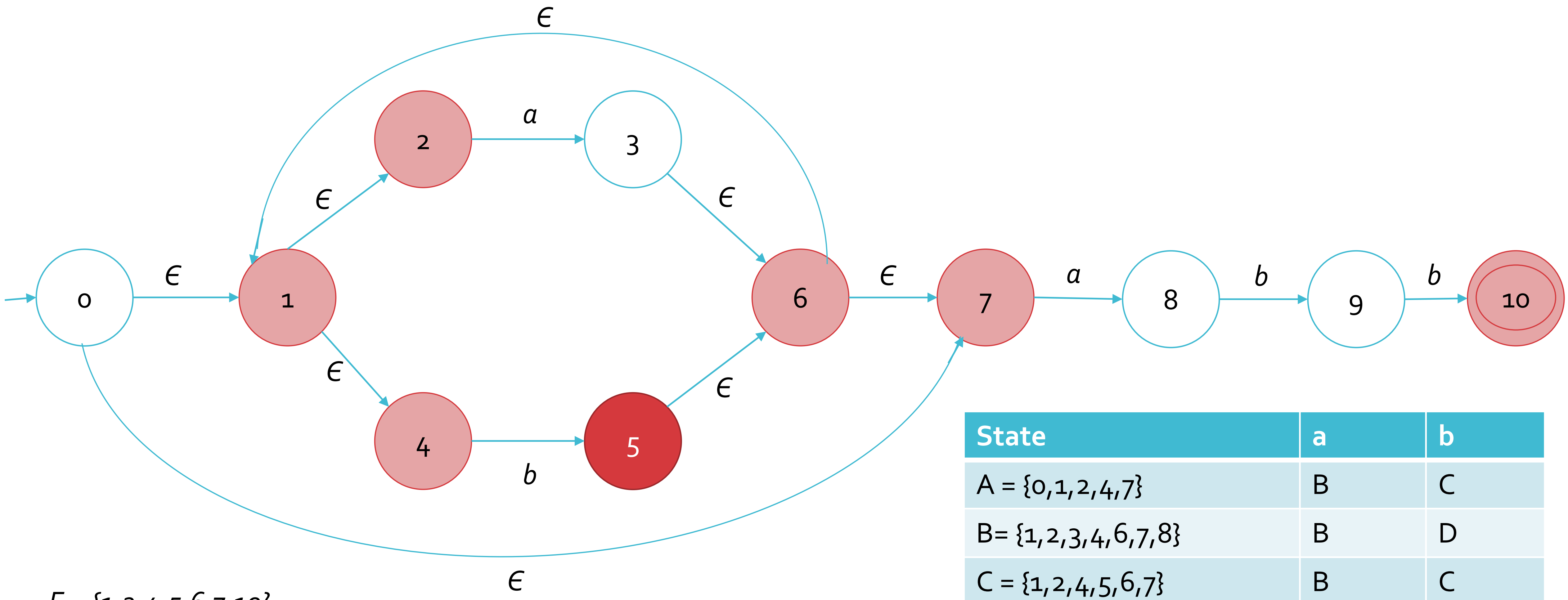
E = {1,2,4,5,6,7,10}

Move (E,a) = {3,8}

ϵ Closure (Move (E,a)) = {3,6,7,1,2,4,8}

= {1,2,3,4,6,7,8} → B

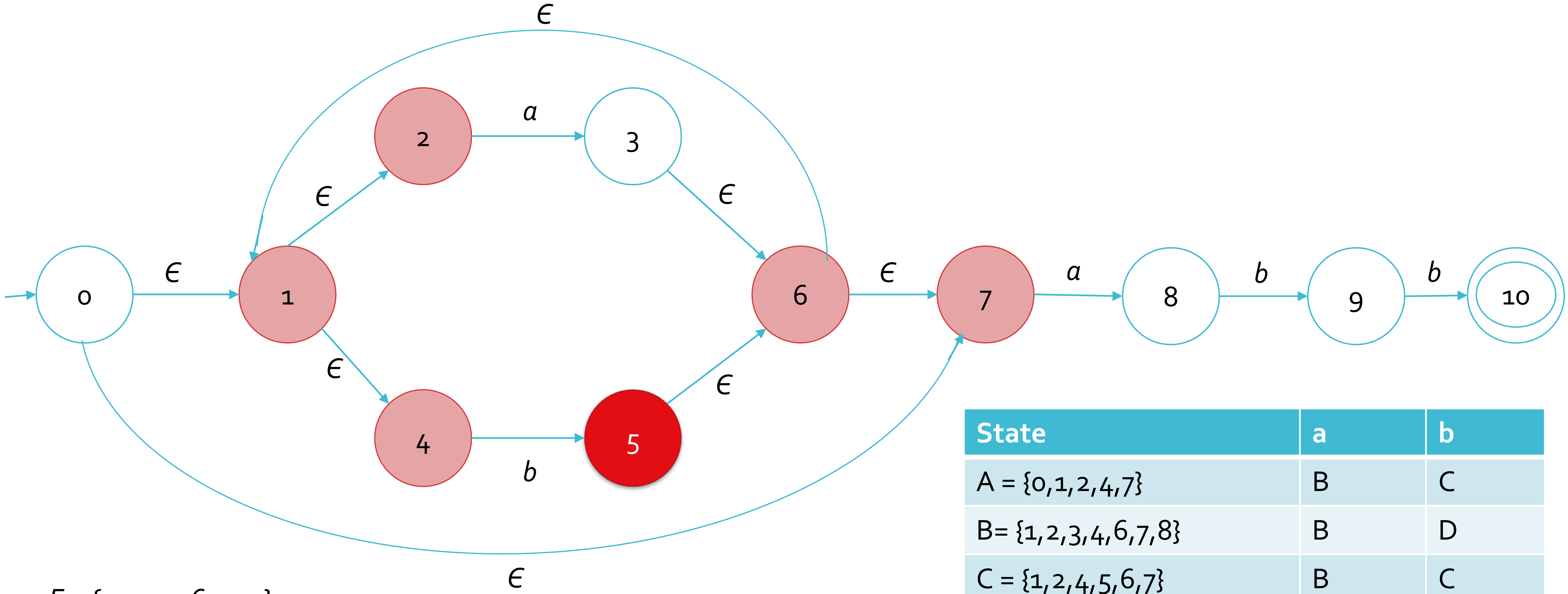
Example : (a|b)*abb



State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	

E = {1,2,4,5,6,7,10}
Move (E,b) = {5}

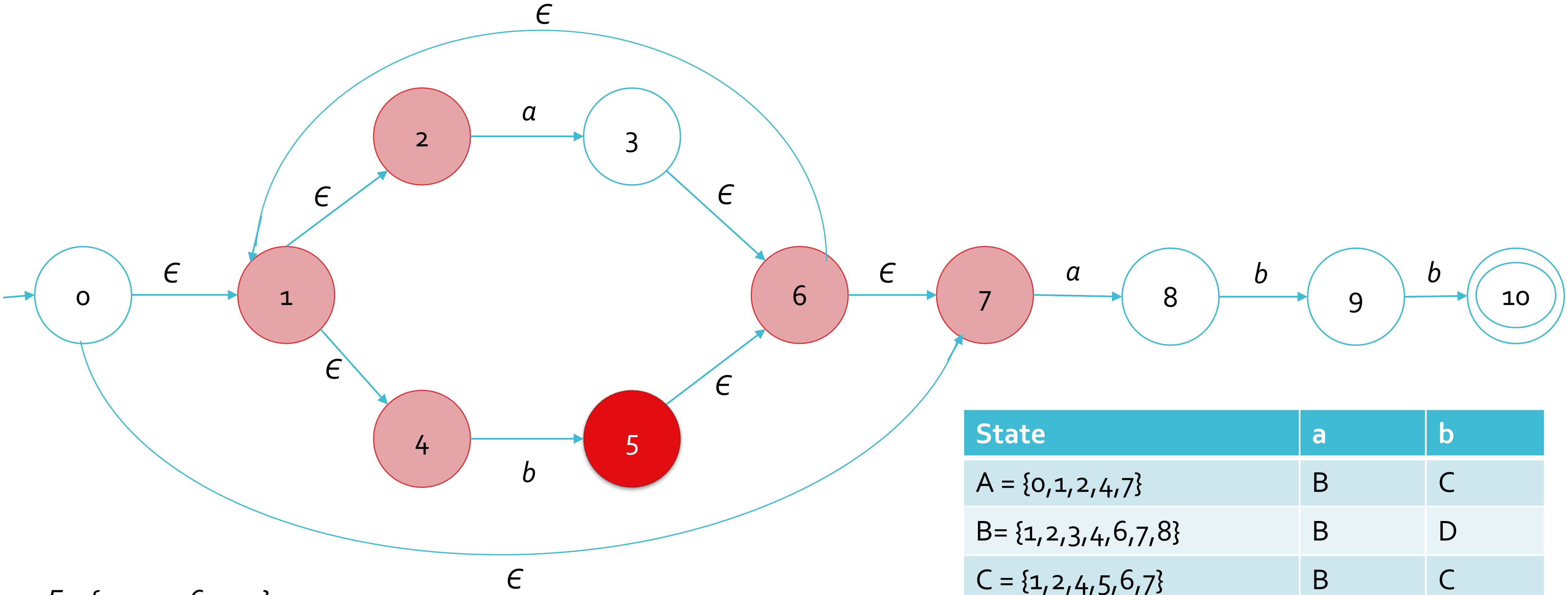
Example : (a|b)*abb



E = {1,2,4,5,6,7,10}
Move (E,b) = {5}

State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	

Example : (a|b)*abb



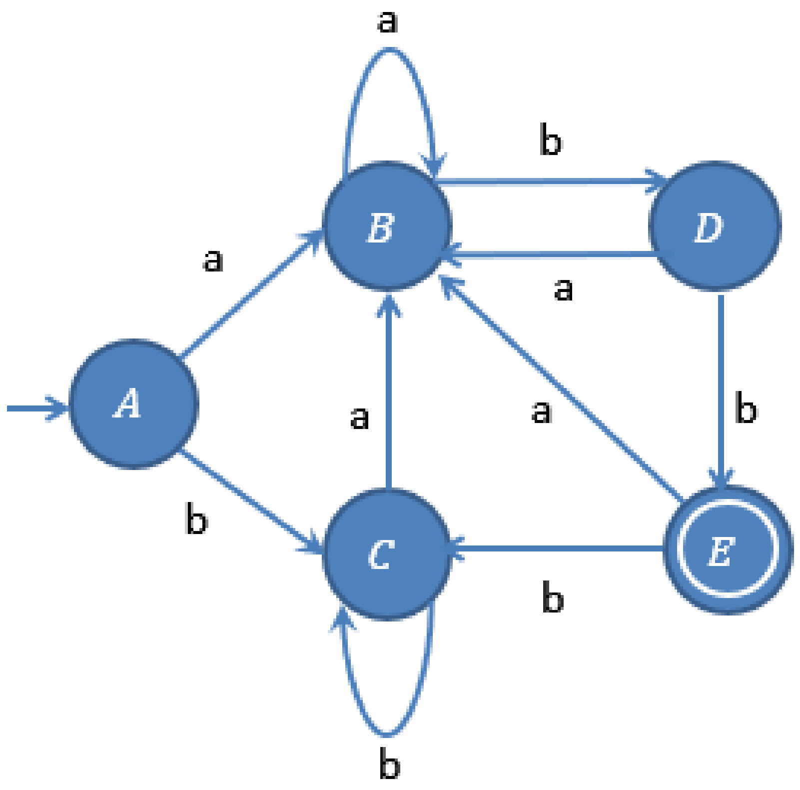
State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	C

$E = \{1,2,4,5,6,7,10\}$

$Move(E,b) = \{5\}$

Example : (a|b)*abb

State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	C



DFA

Convert NFA to DFA : (a|b)*abb

ϵ –closure (0) = {0,1,2,4,7} → **Let A**

Move (A,a) = {3,8}

ϵ –closure ((A,a)) = {1,2,3,4,6,7,8} → **Let B**

Move (A,b) = {5}

ϵ –closure ((A,b)) = {1,2,4,5,6,7} → **Let C**

Convert NFA to DFA : (a|b)*abb (Cont…)

Move (B,a) = {3,8}

∈ –closure ((B,a)) = {1,2,3,4,6,7,8} → ***Let B***

Move (B,b) = {5,9}

∈ –closure ((B,b)) = {1,2,4,5,6,7,9} → ***Let D***

Move (C,a) = {3,8}

∈ –closure ((C,a)) = {1,2,3,4,6,7,8} → ***Let B***

Move (C,b) = {5}

∈ –closure ((C,b)) = {1,2,4,5,6,7} → ***Let C***

Convert NFA to DFA : (a|b)*abb (Cont…)

Move (D,a) = {3,8}

∈ –closure ((D,a)) = {1,2,3,4,6,7,8} → **Let B**

Move (D,b) = {5,10}

∈ –closure ((D,b)) = {1,2,4,5,6,7,10} → **Let E**

Move (E,a) = {3,8}

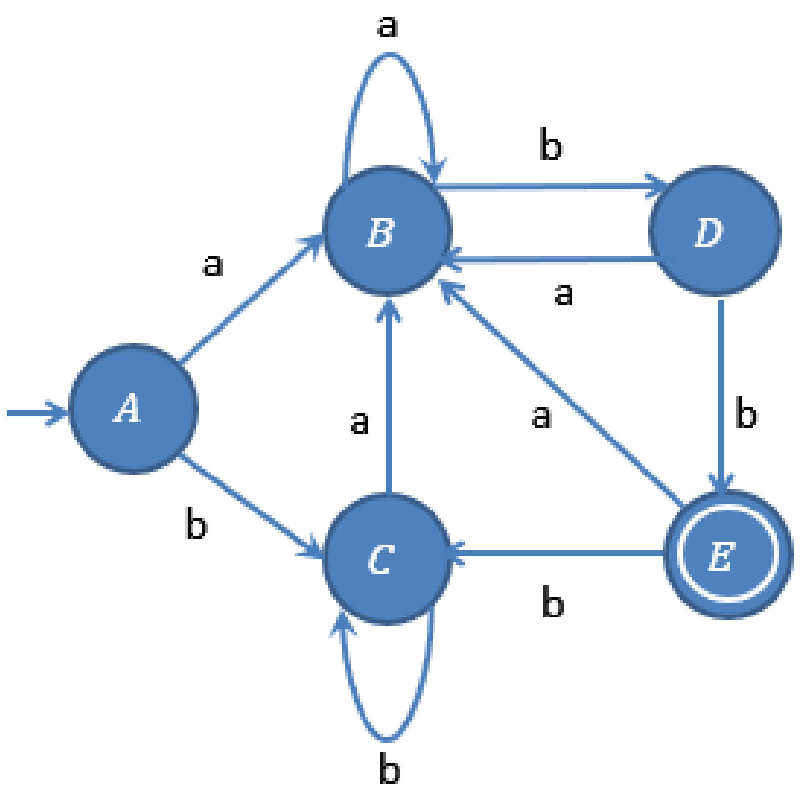
∈ –closure ((E,a)) = {1,2,3,4,6,7,8} → **Let B**

Move (E,b) = {5}

∈ –closure ((E,b)) = {1,2,4,5,6,7} → **Let C**

Convert NFA to DFA : (a|b)*abb (Cont…)

State	a	b
A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	C



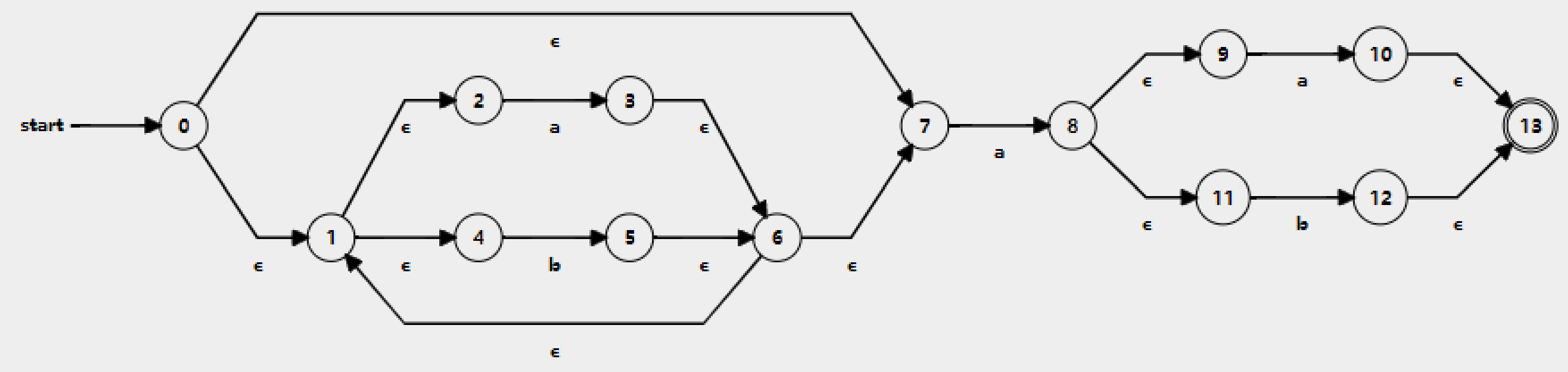
DFA

Example

- Convert following regular expression to DFA using subset construction method:
- (a|b)*a(a|b)
- $(0 + 1)^* 1(0 + 1)$
- $(0 + 1)^* 01^*$

Example

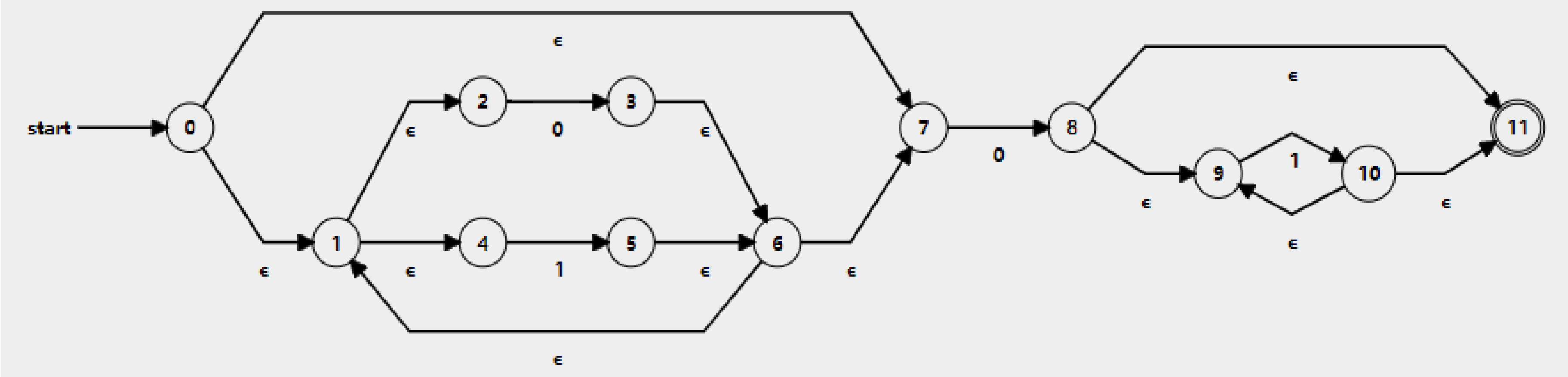
- Convert following regular expression to DFA using subset construction method:
- (a|b)*a(a|b)



State	a	b
→A = {0,1,2,4,7}	B	C
B= {1,2,3,4,6,7,8,9,11}	D	E
C = {1,2,4,5,6,7}	B	C
* D = {1,2,3,4,5,6,7,8,9,10,11,13}	D	E
* E = {1,2,4,5,6,7,12,13}	B	C

Example

- Convert following regular expression to DFA using subset construction method:
- (0|1)*01*



State	a	b
→A = {0,1,2,4,7}	B	C
* B= {1,2,3,4,6,7,8,9,11}	B	D
C = {1,2,4,5,6,7}	B	C
* D = {1,2,3,4,5,6,7,9,10,11}	B	D

DFA State	a	b
→{AC}	B	C
* {BD}	B	D

DFA Optimization

DFA Optimization

- The procedure can also be known as minimization of DFA.
- Minimization/optimization refers to the detection of those states of a DFA, whose presence or absence in a DFA does not affect the language accepted by the automata.
- The states that can be eliminated from automata, without affecting the language accepted by automata, are:
  - Unreachable or inaccessible states
  - Dead states
  - Non-distinguishable or indistinguishable state or equivalent states.
- Partitioning algorithm helps in DFA optimization process.

Partitioning Algorithm

1. Remove all the states that are unreachable from initial state via any set of transition of DFA.
2. Draw the transition table for all pair of states.
3. Now, split the transition table into two tables T_1 and T_2.
    1. T_1 contains all final states
    2. T_2 contains non-final states
4. Find similar rows from T_1 such that;

$$\delta(q, a) = p$$

$$\delta(r, a) = p$$

i.e. find the two states which have same value of a and b and remove one of them

Continued…

5. Repeat step 3 until we find no similar rows available in T_1
6. Repeat step 3 and step 4 for table T_2 also.
7. Now combine the reduced T_1 and T_2 tables.

i.e. the final transition table of minimized DFA.

DFA Optimization

$\{A, B, C, D, E\}$

Nonaccepting States $\{A, B, C, D\}$

Accepting States $\{E\}$

$\{A, B, C\}$

$\{D\}$

$\{A, C\}$

$\{B\}$

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

- Now no more splitting is possible.
- If we chose A as the representative for group (AC), then we obtain reduced transition table

States	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Optimized Transition Table

DFA Optimization

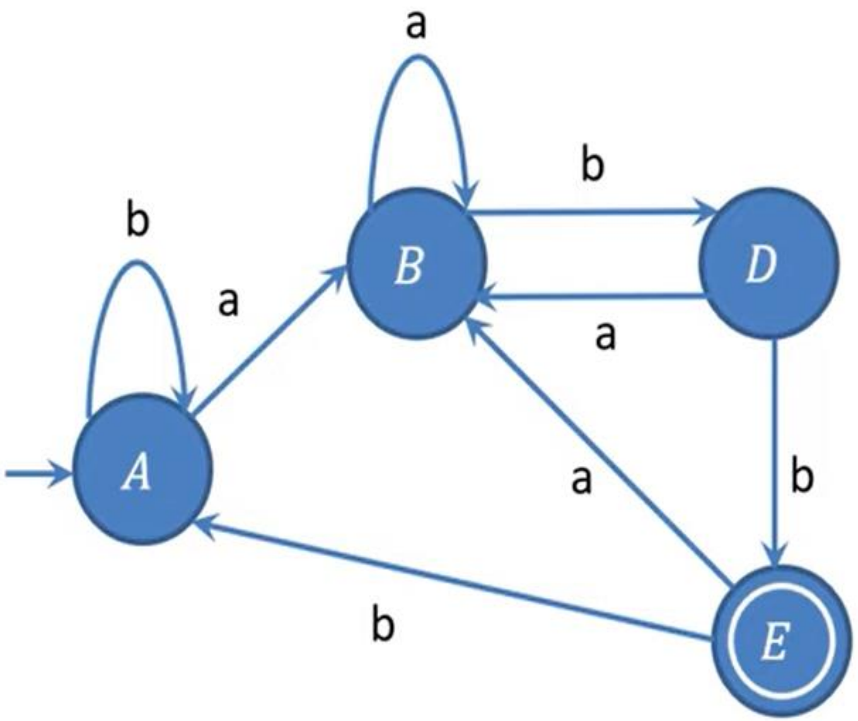


Fig: Minimized DFA for the R.E : (a | b)*abb

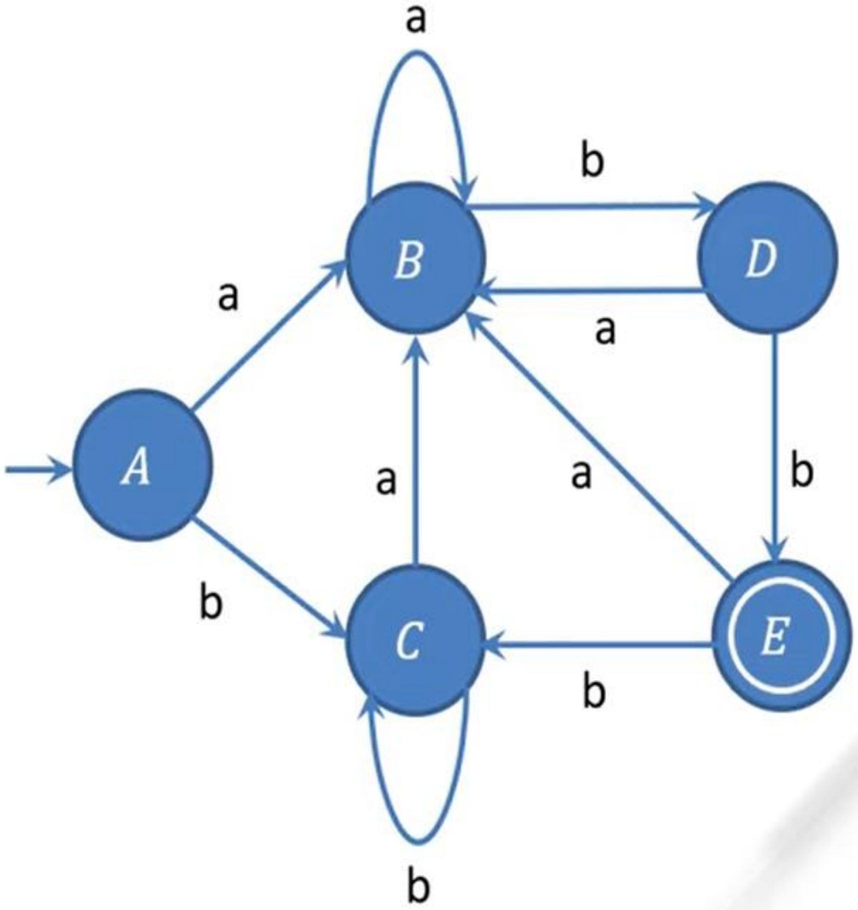


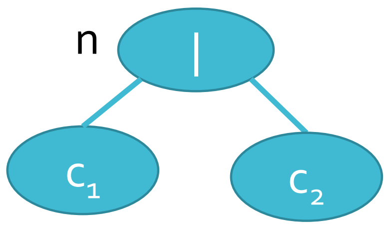
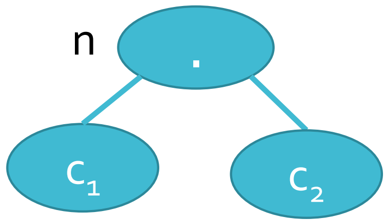
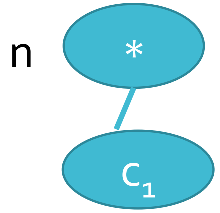
Fig: DFA for the R.E : (a | b)*abb

Conversion from regular expression to DFA

Function computed from syntax tree

- **nullable** *(n):* Is true for * node and node labeled with Ɛ. For other nodes it is false.
- **firstpos** *(n):* Set of positions at node *ti* that corresponds to the first symbol of the sub-expression rooted at *n.*
- **lastpos** *(n):* Set of positions at node *ti* that corresponds to the last symbol of the sub-expression rooted at *n.*
- **followpos** (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

Rules to compute nullable, firstpos, lastpos

Node *n*	***nullable*(*n*)**	***firstpos*(*n*)**	***lastpos*(*n*)**
A leaf labeled by ε	**true**	$\emptyset$	$\emptyset$
A leaf with position ***i***	**false**	$\{i\}$	$\{i\}$
	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
	$nullable(c_1)$ **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** $lastpos(c_2)$
	***true***	$firstpos(c_1)$	$lastpos(c_1)$

Computation of follow pos

The position of regular expression can follow another in the following ways:

- If *n* is a cat node with left child c_1 and right child c_2, then for every position i in *lastpos*(c_1), all positions in *firstpos*(c_2) are in *followpos(i).*
- For cat node, for each position i in *lastpos* of its *left child,* the *firstpos* of its *right child* will be in f*ollowpos(i).*

- If *n* is a star node and i is a position in *lastpos(n),* then all positions in *firstpos(n)* are in *followpos(i).*
- For star node, the *firstpos* of that node is in *f ollowpos* of all positions in *lastpos* of that node.

Conversion from regular expression to DFA

Step 1: Convert the RE into Augmented RE.

Step 2: Construct Syntax Tree

Step 3: Identify Nullable Node

Step 4: Calculate the First Position and Last Position

Step 5: Calculate the Follow Position

Step 6: Draw Transition Table

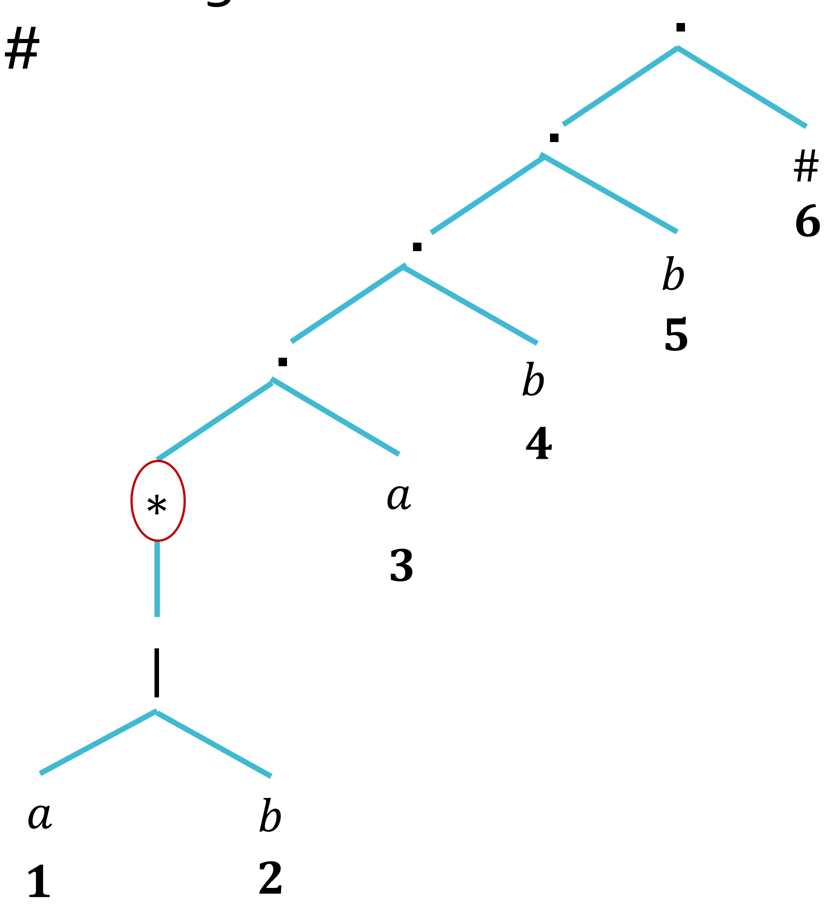
Step 7: Construct DFA

Conversion from regular expression to DFA

(a|b) * abb #

Step 1: Construct Augmented RE

(a|b) *.a.b.b.#



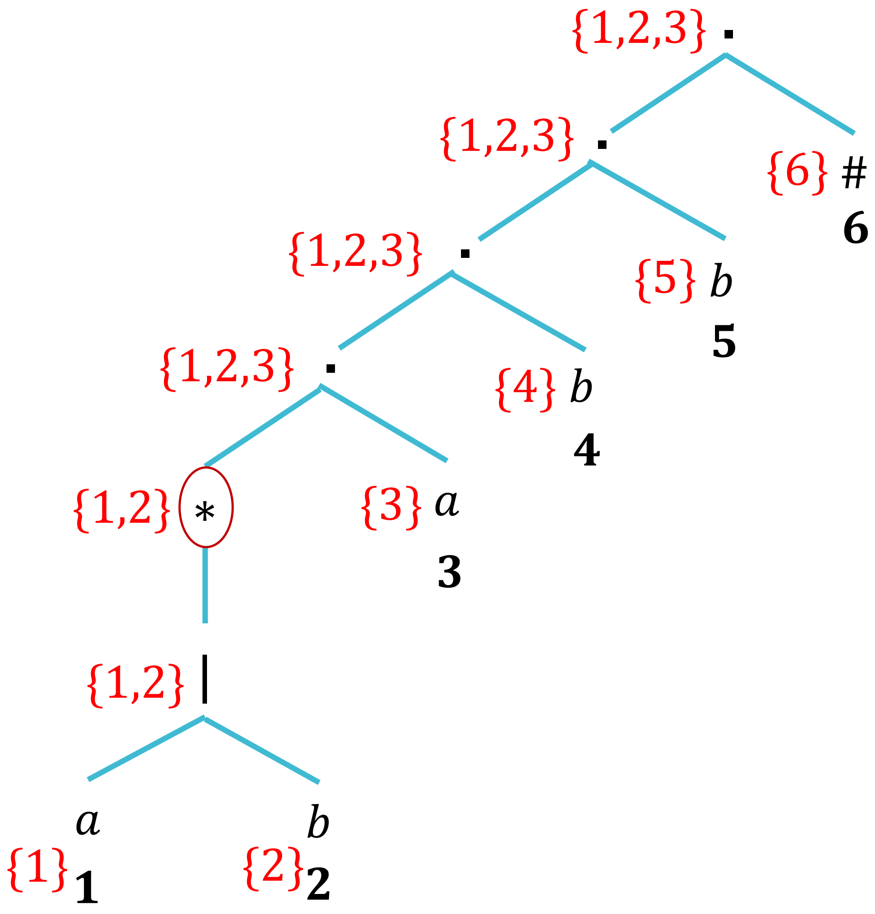
Step 2: Construct Syntax Tree

Step 3: Nullable node

Here, * is only nullable node

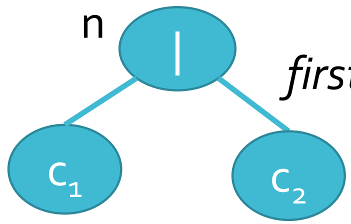
Conversion from regular expression to DFA

Step 4: Calculate firstpos

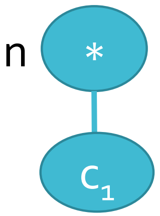


Firstpos —

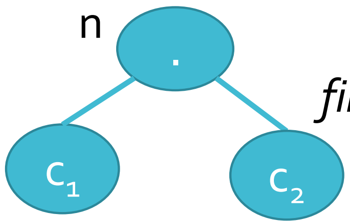
A leaf with position $i = \{i\}$



$firstpos(c_1) \cup firstpos(c_2)$



$firstpos(c_1)$



if $(nullable(c_1))$

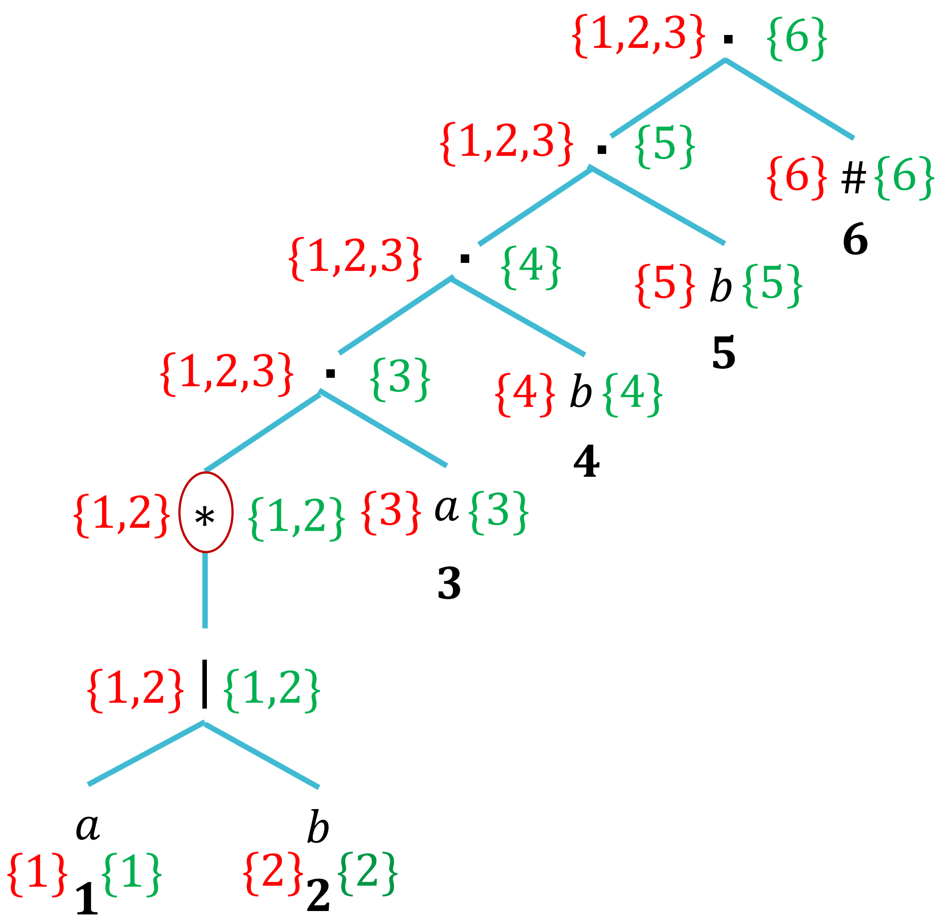
$firstpos(c_1) \cup firstpos(c_2)$

else $firstpos(c_1)$

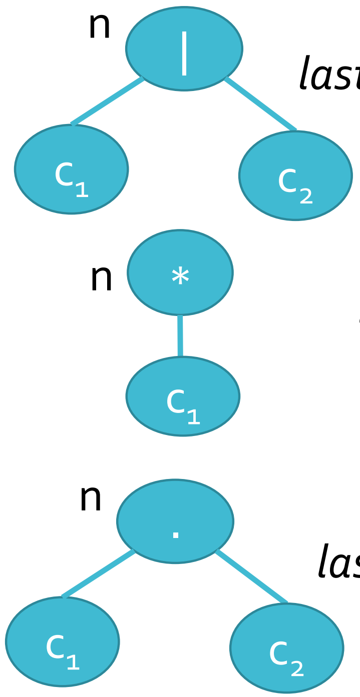
Conversion from regular expression to DFA

Step 4: Calculate lastpos

Lastpos —



A leaf with position i = $\{i\}$



$lastpos(c_1) \cup lastpos(c_2)$

$lastpos(c_1)$

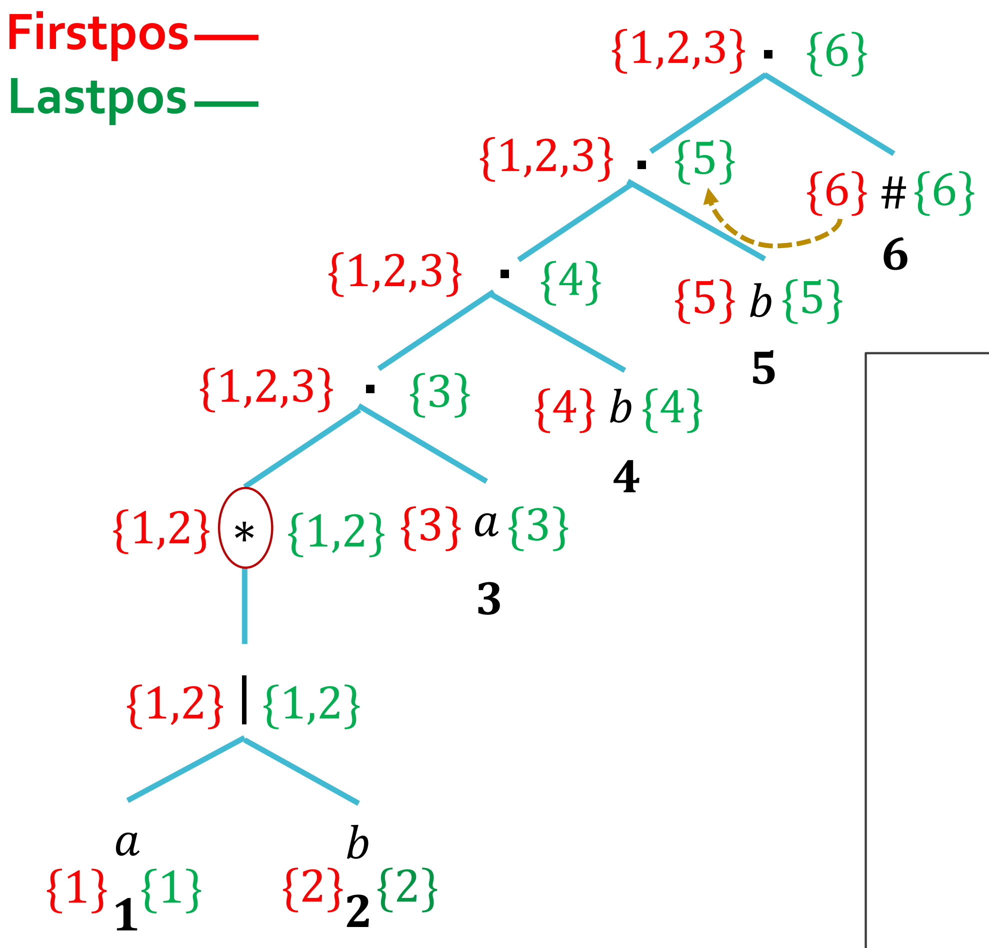
if $(nullable(c_2))$

$lastpos(c_1) \cup lastpos(c_2)$

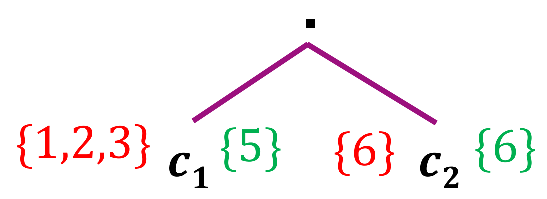
else $lastpos(c_2)$

Conversion from regular expression to DFA

Step 5: Calculate followpos



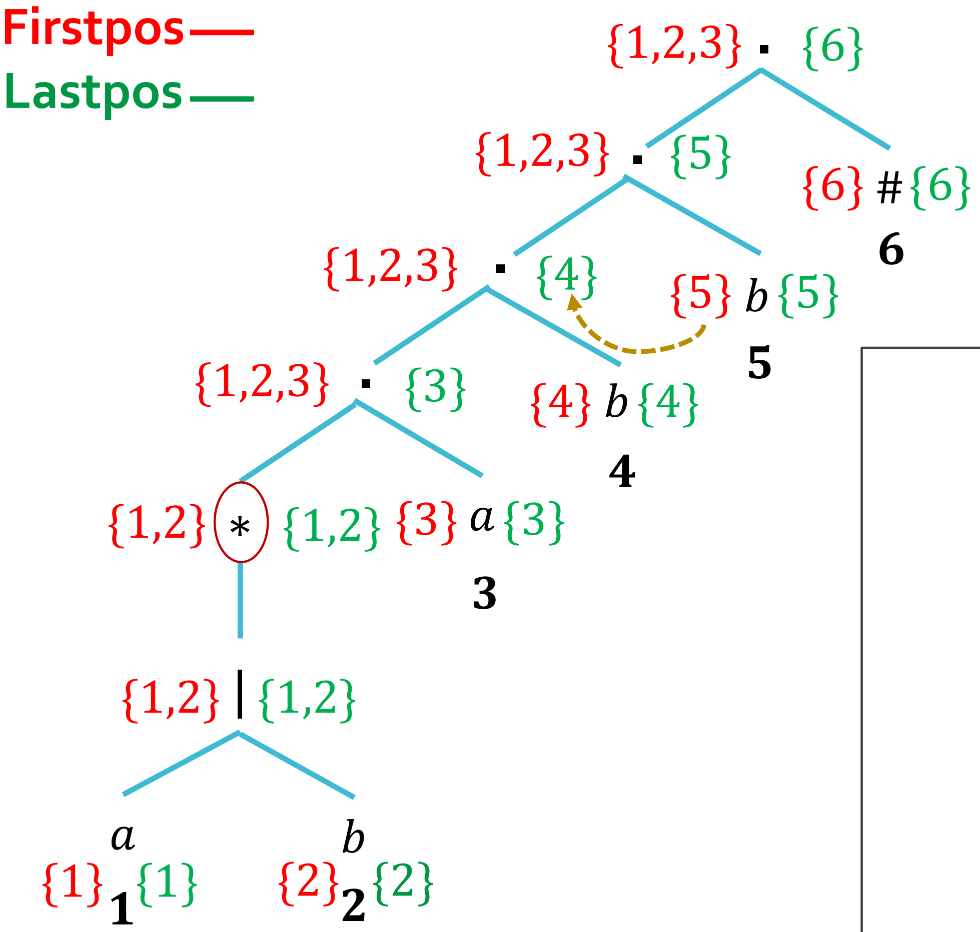
Position	followpos
5	6



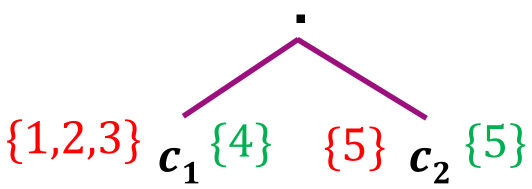
$$i = lastpos(c_1) = \{5\}$$
$$firstpos(c_2) = \{6\}$$
$$followpos(5) = \{6\}$$

Conversion from regular expression to DFA

Step 5: Calculate followpos



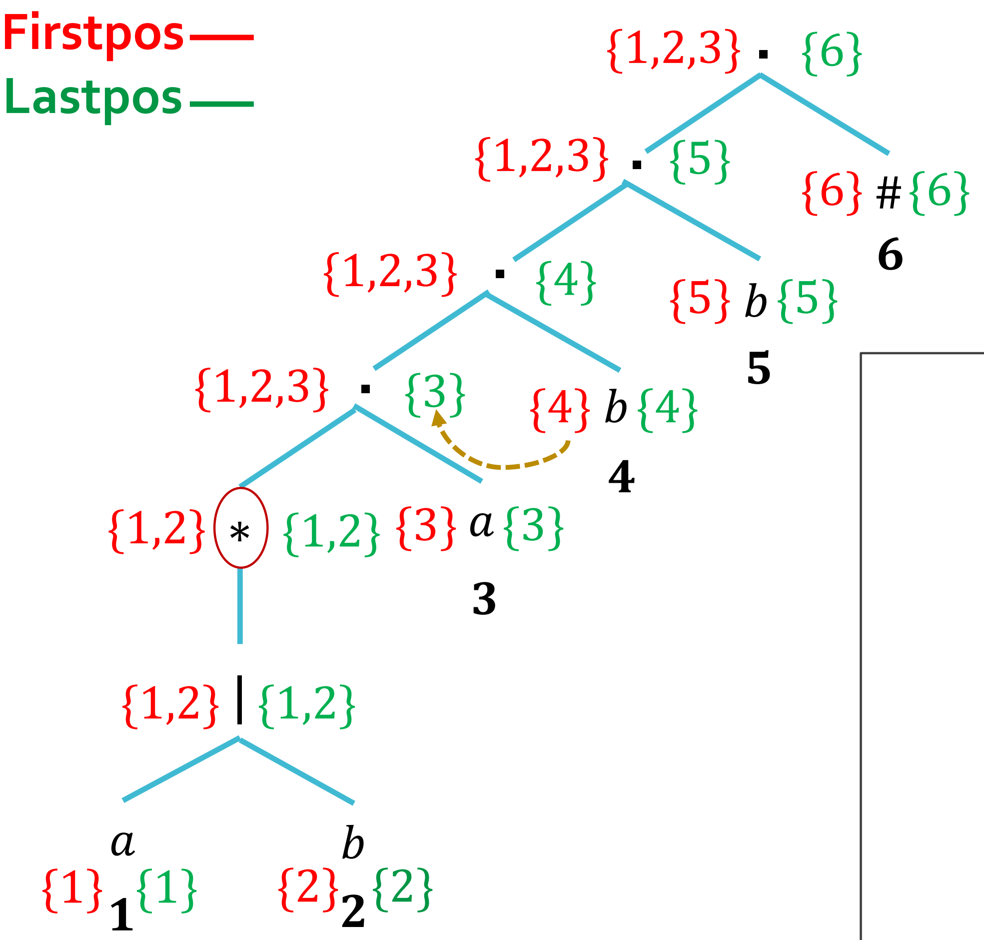
Position	followpos
5	6
4	5



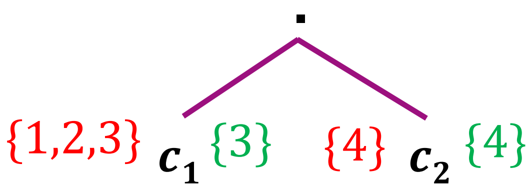
$$i = lastpos(c_1) = \{4\}$$
$$firstpos(c_2) = \{5\}$$
$$followpos(4) = \{5\}$$

Conversion from regular expression to DFA

Step 5: Calculate followpos



Position	followpos
5	6
4	5
3	4

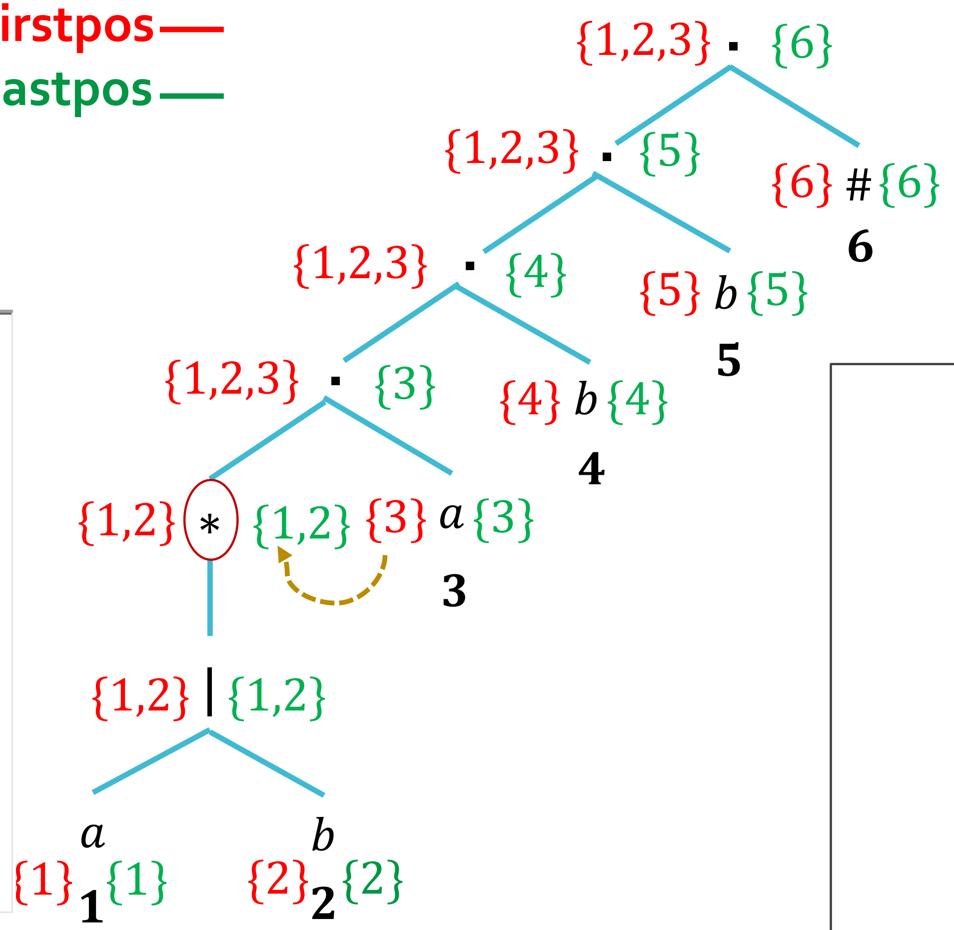


$$i = lastpos(c_1) = \{3\}$$
$$firstpos(c_2) = \{4\}$$
$$followpos(3) = \{4\}$$

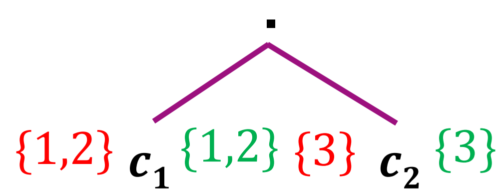
Conversion from regular expression to DFA

Step 5: Calculate followpos

Firstpos
Lastpos



Position	followpos
5	6
4	5
3	4
2	3
1	3

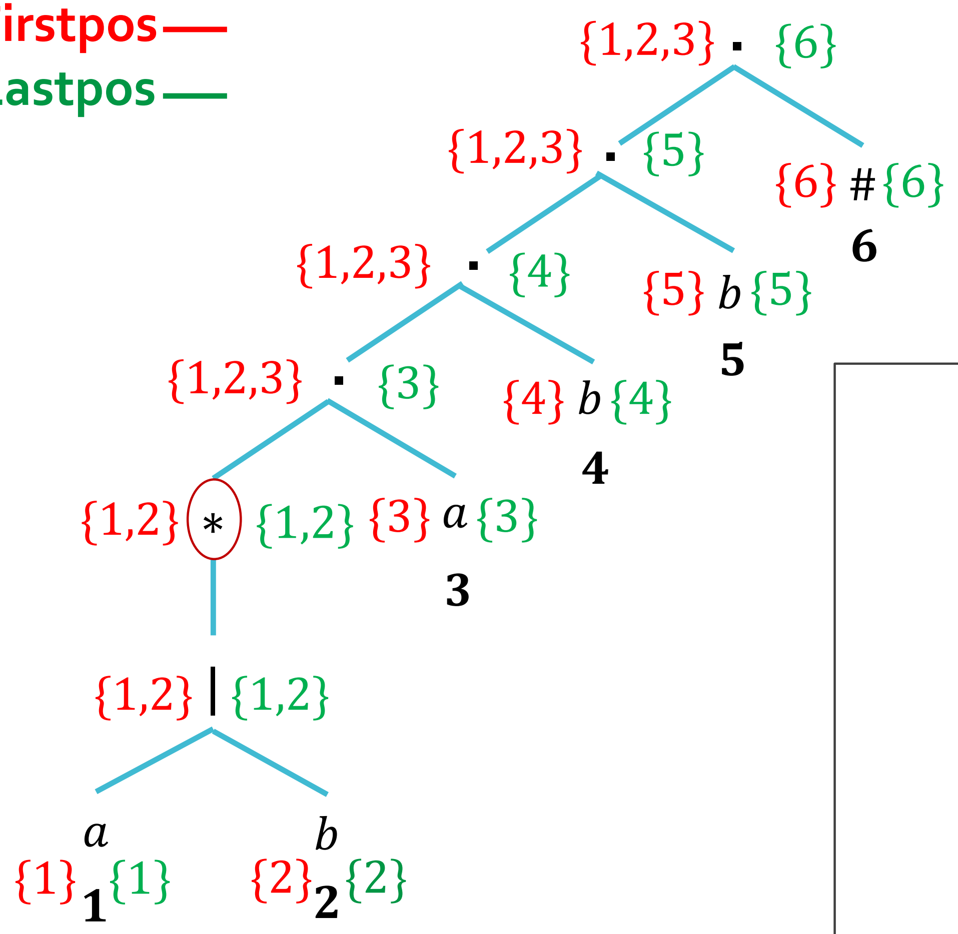


$$i = lastpos(c_1) = \{1,2\}$$
$$firstpos(c_2) = \{3\}$$
$$followpos(1) = \{3\}$$
$$followpos(2) = \{3\}$$

Conversion from regular expression to DFA

Step 5: Calculate followpos

Firstpos
Lastpos



Position	followpos
5	6
4	5
3	4
2	1, 2, 3
1	1, 2, 3

{1,2} * {1,2}

$\boldsymbol{n}$

$$i \; = \; lastpos(n) \; = \; \{1,2\}$$
$$firstpos(n) = \{1,2\}$$
$$followpos(1) = \{1,2\}$$
$$followpos(2) = \{1,2\}$$

Construct DFA

Step 6: Make Transition Table

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

Initial state = $firstpos$ of root = {1,2,3} ----- A

State A

δ(A, a) = followpos(1) U followpos(3)

={1,2,3} U {4} = {1,2,3,4} ----- B

δ(A, b) = followpos(2)

={1,2,3} ----- A

States	a	b
A={1,2,3}	–	
B={1,2,3,4}		

Construct DFA

State B

$\delta(B, a)$ = followpos(1) U followpos(3)

={1,2,3} U {4} = {1,2,3,4} ----- B

$\delta(B, b)$ = followpos(2) U followpos(4)

={1,2,3} U {5} = {1,2,3,5} ----- C

State C

$\delta(C, a)$ = followpos(1) U followpos(3)

={1,2,3} U {4} = {1,2,3,4} ----- B

$\delta(C, b)$ = followpos(2) U followpos(5)

= {1,2,3} U {6} = {1,2,3,6} ----- D

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

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

Construct DFA

State D

$\delta(D , a)$ = followpos(1) U followpos(3)

={1,2,3} U {4} = {1,2,3,4} ----- B

$\delta(D , b)$ = followpos(2)

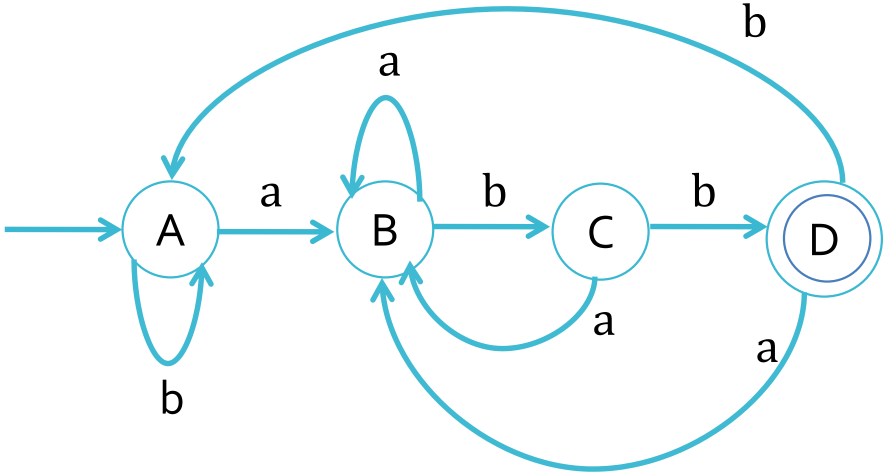
={1,2,3} ----- A

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

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

Construct DFA

Step 7: Construct DFA



DFA

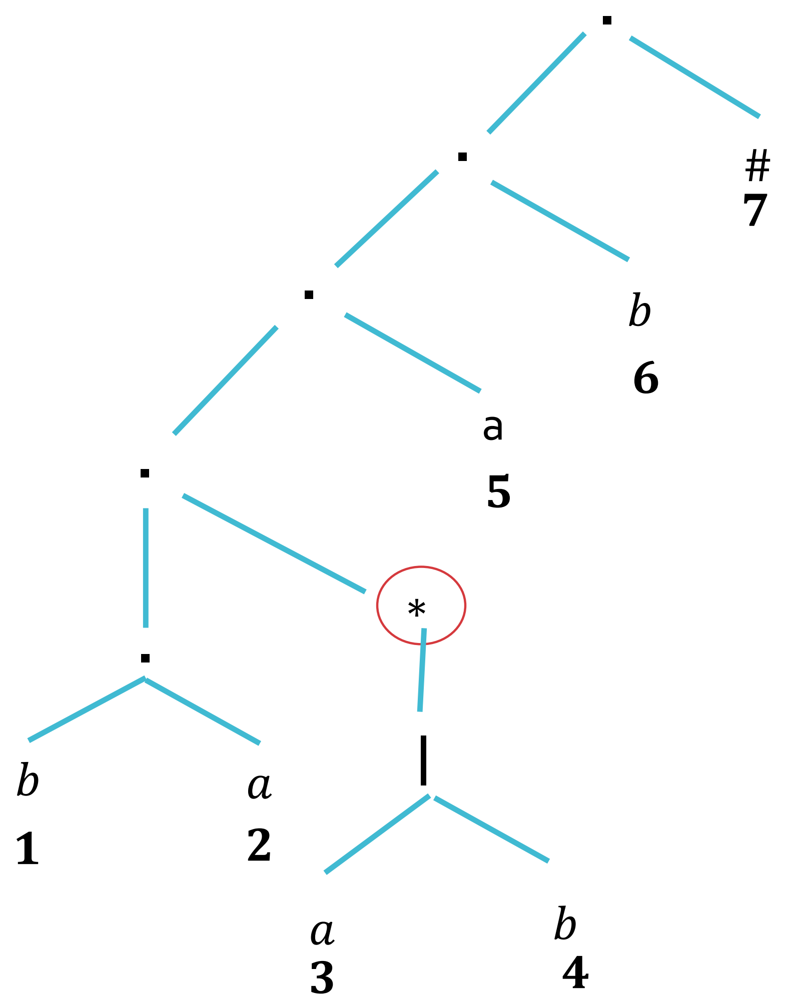
Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

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

Conversion from regular expression to DFA

ba(a|b)*ab

Step 1 Augmented RE = b .a . (a|b)* . a . b .#



Step 2: Construct Syntax Tree

Step 3: Nullable node

Here, * is only nullable node

Construct DFA

Position	followpos
1	2
2	3,4,5
3	3,4,5
4	3,4,5
5	6
6	7
7	-

States	a	b
→A = {1}	-	B
B = {2}	C	-
C = {3,4,5}	D	C
D = {3,4,5,6}	D	E
*E = {3,4,5,7}	D	C

References

1. Aho, Lam, Sethi, and Ullman, Compilers: Principles, Techniques and Tools,SecondEdition, Pearson, 2014
2. D. M. Dhamdhere: Systerm Programming, Mc Graw Hill Publication
3. Dick Grune, Henri E. Bal, Jacob, Langendoen: Modern Compiler Design, Wiley India Publication

MU Questions

1) Explain Pattern, Tokens and Lexeme
2) Draw NFA using Thompson's rule and convert following expression into DFA and also. show the minimization of DFA. (a | b)*a.
3) Draw DFA from given expression using first pos, last pos and follow pos. (a | b)*abb.
4) Draw Transition diagram for relational operator.
5) Draw RE to Epsilon NFA using Thompson's Method. a (a | b)*
6) What is the output of Lexical analyzer?
7) Construct DFA from given RE using Thompson's Subset Construction Method (a* | b)* ab

Thanks

Marwadi University

Unit no : 2
Lexical Analysis
(01CE0714)

Prof. Shilpa Singhal