Chapter 3: Scanning

陳奇業 成功大學資訊工程系

Scanner (or Lexical Analyzer)

the interface between source & compiler

could be a separate pass and places its output on an intermediate file.

more commonly, it is a routine called by parser.

scans character stream from where it left off and returns next token to parser. Actually the token's **lexical category** is returned in the form of a simple index number and a value for the token is left in the global variables. For some tokens only token type is returned.

Tokens

- Tokens are logical entities that are usually defined as an enumberated type. For example, tokens might be defined in C as

```
typedef enum
   {IF, THEN, ELSE, PLUS, MINUS, NUM, ID, ...}
   TokenType;
```

Tokens

- Although the task of the scanner is to convert the entire source program into a sequence of tokens, the scanner will rarely do this all at once.
- Instead, the scanner will operate under the control of the parser, returning the single next token from the input on demand via a function that will have a declaration similar to the C declaration

```
TokenType getToken(void);
```

Input Buffering

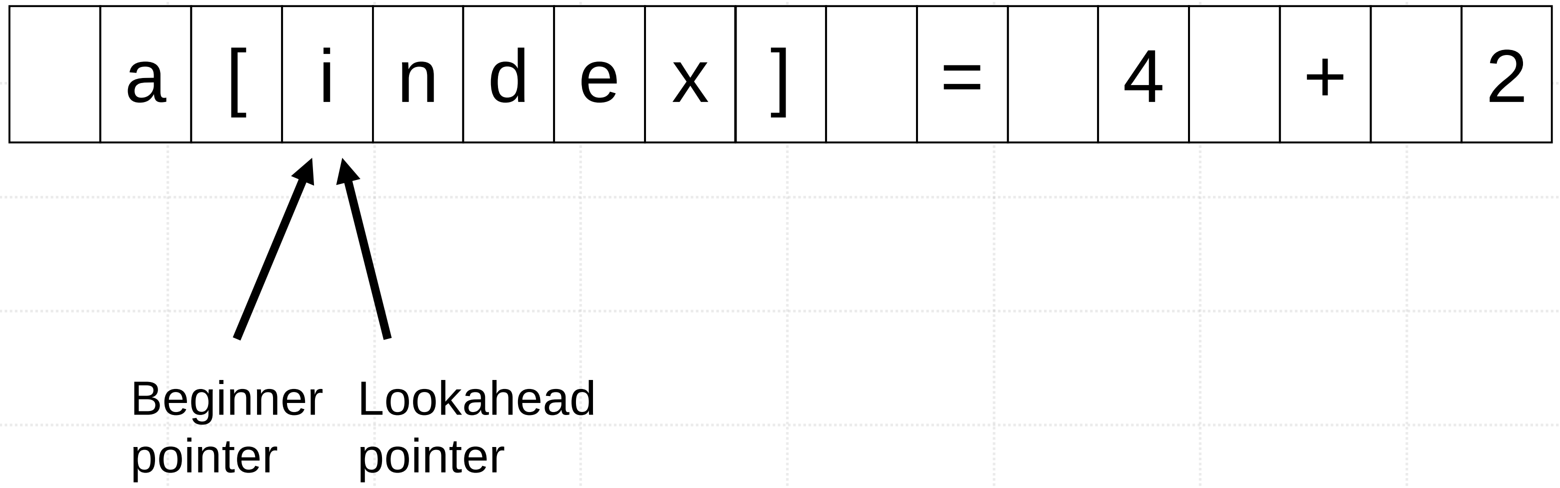
Why input buffer is needed?

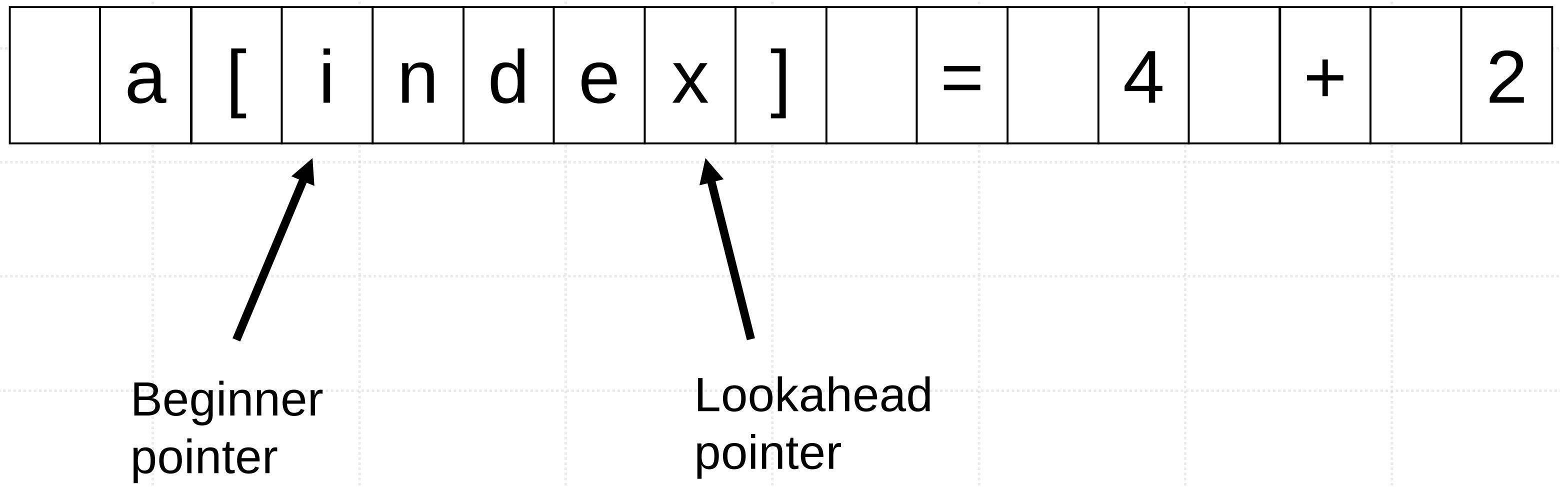
we can identify some token only when many characters beyond the token have been examined.

Two pointers (one marks the beginning of the token & the other one is a lookahead pointer) delimit the context of the token string.

However, the lookahead is limited.

a [index] = 4 + 2


a
[
i
n
d
e
x
]
=
4
+
2
Beginner pointer
Lookahead pointer


a
[
i
n
d
e
x
]
=
4
+
2
Beginner pointer
Lookahead pointer

Token, Pattern & Lexeme

A token is a sequence of characters that represents a unit of information in the source program.

In general, there is 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. The pattern is said to match each specific string in the set.

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

Pattern vs. Regular Expression

Regular Expression: A notation suitable for describing tokens (patterns).

Each pattern is a regular language, i.e., it can be described by a regular expression

Two kinds of token

specific string (e.g., "if" ","
"=="), that is, a single string.

class of string (e.g., identifier,
number, label), that is, a
multiple strings.

Examples of tokens

Token (詞彙)	Informal Description	Pattern (模式)	Sample Lexemes (詞素)
if	characters i, f	if →**if**	if
else	characters e, 1, s, e	$else$ →**else**	else
comparison	< or > or <= or >= or <>	$relop$ →**< | > | <= | >= | = | <>**	<, <=
id	letter followed by letters and digits	$letter$ →[**A-Za-z**] $digit$ →[**0-9**] $id \rightarrow letter(letter\|digit)^*$	pi, score, D2
number	any numeric constant	$digits \rightarrow digit^+$ $number \rightarrow digits(.\ digits)?$	3.14159, 0, 6.02e23
literal	anything but “, surrounded by “'s	\“[^”]*\“	“Hello world"

Common Lexical Categories

identifiers

literals

keywords (not necessarily to be a reserved word)
- What is the difference between keyword and reserved word?

operators

numbers

punctuation symbols: e.g. '(‘ , ';‘ , ','

- The lexical categories are regular languages, and they can be described by giving regular expressions.

An instance: Scanning E = M * C ** 2

// return token type (an index) and value

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

< assign_op >

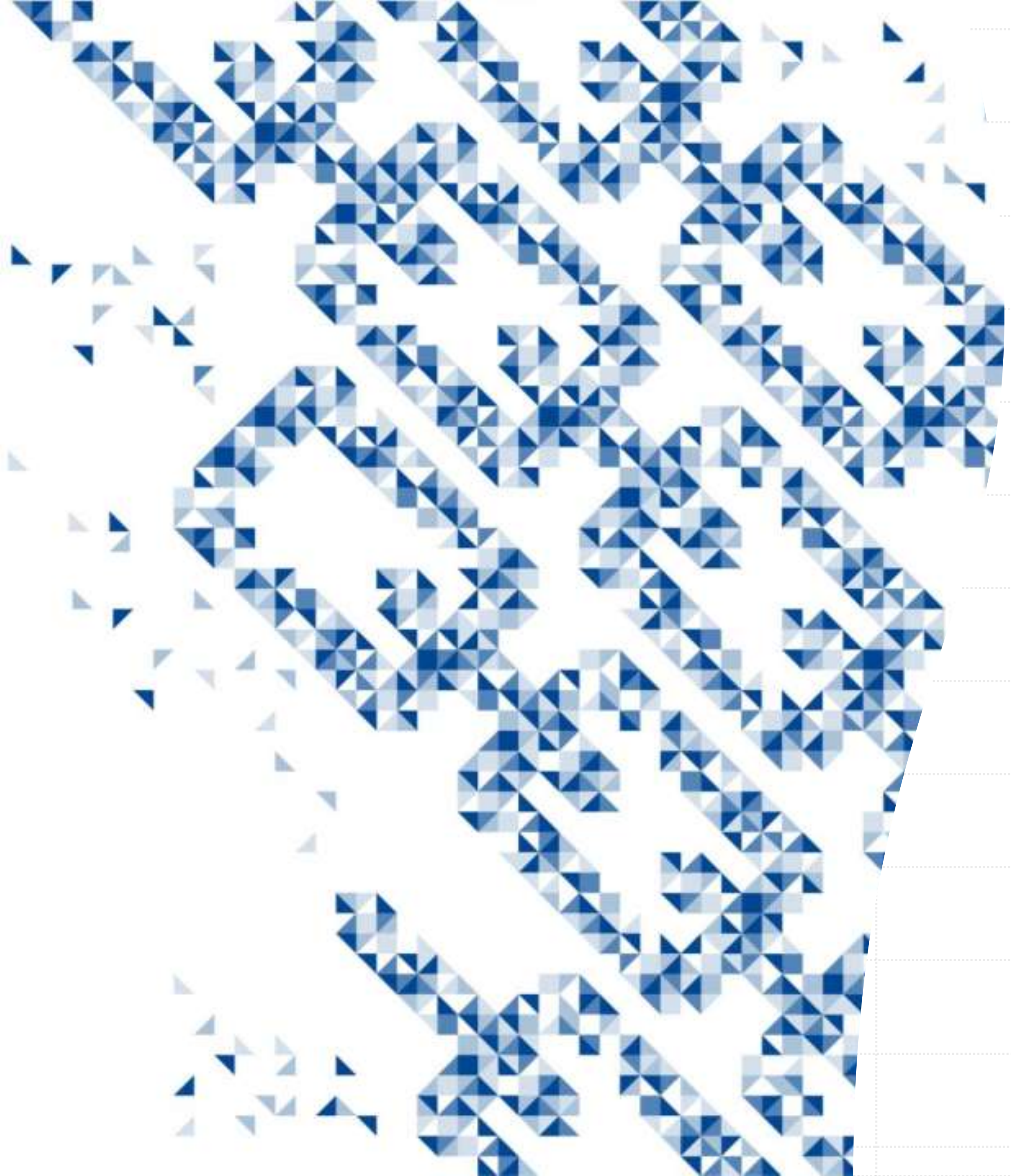
< id, pointer to symbol table entry for M >

< mult_op >

< id, pointer to symbol table entry for C >

< expo_op >

< num, integer value 2 >



- The compiler may store the character string that forms a number in a symbol table and let the attribute of token be a pointer to the table entry.

Regular Expression vs. Finite Automata

Regular Expression: A notation suitable for describing tokens (patterns).

Regular Expression can be converted into Finite Automata

Finite Automata: Formal specifications of transition diagrams

Definitions of String

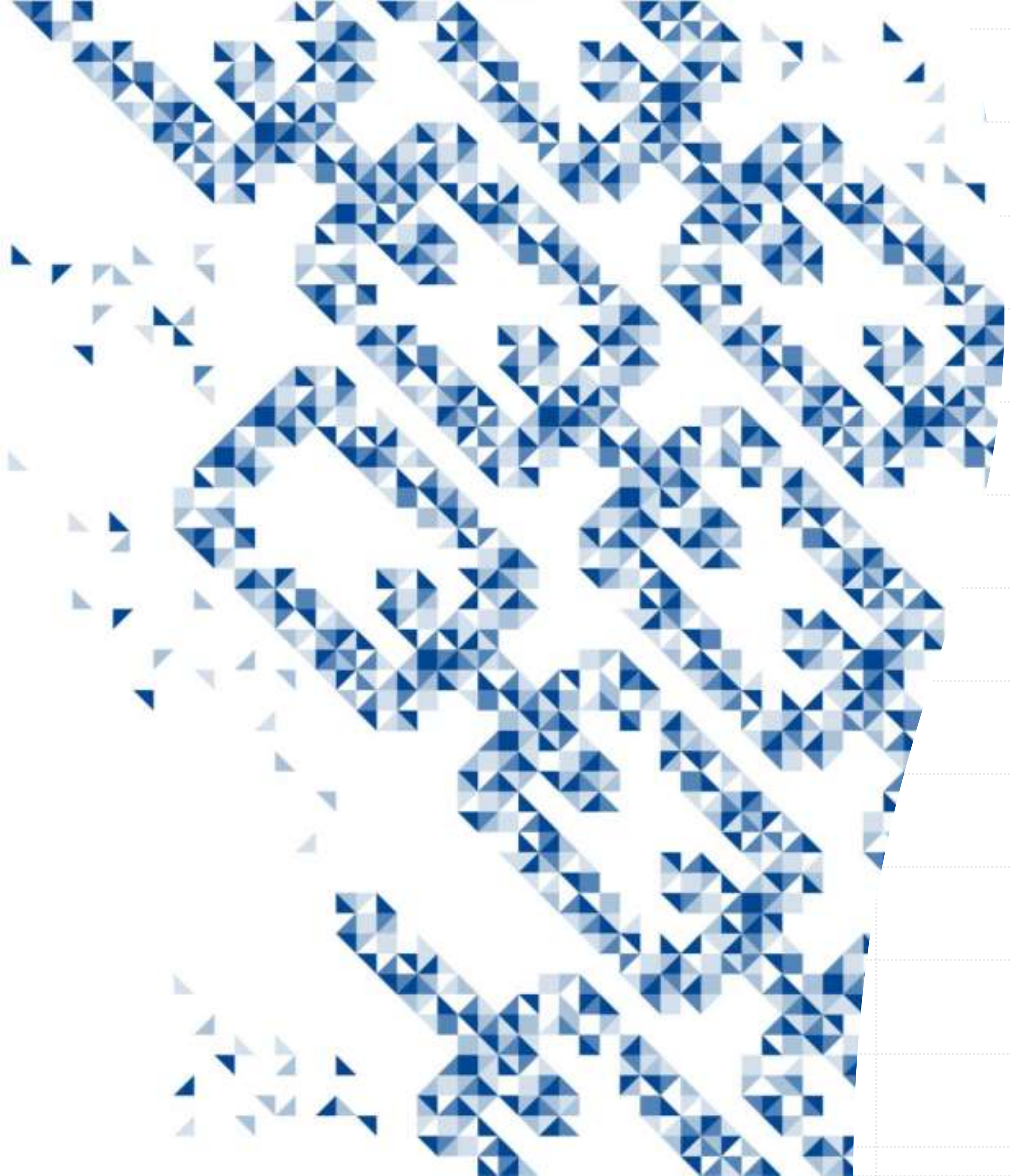
- symbol: undefined entity.
  e.g. digits, letters
- alphabet (character class): any finite set of symbols. (Usually it is denoted as the symbol Σ)
  e.g. the set {0,1} is an alphabet
- string (sentence, word): a finite sequence of symbols.
  e.g. 001, 0, 111

Operations of string (I)

- Length
  e.g., |0| = 1, |x| = the total number of symbols in string x, empty string denoted ε, $|\varepsilon| = 0$
  (note: $\{\varepsilon\} \neq \emptyset$)
- Concatenation
  e.g., $x.y = xy$, $\varepsilon x = x$, $x\varepsilon = x$, $x^1 = x$, $x^0 = \varepsilon$, $x^2 = xx$
- Prefix - any # of leading symbols at the string
- proper prefix (shown below)

Operations of string (II)

- Suffix - any # of trailing symbols at the string
- proper suffix (explained below)
- Substring - any string obtained by deleting a prefix and a suffix.
- proper substring (explained below)



- A nonempty string y is a proper prefix, suffix, sub-string of x if it is a prefix, suffix, sub-string of x and $x \neq y$.

Definition of Language

- language: any set of strings formed from a specific alphabet.
  e.g. $\emptyset$, $\{\varepsilon\}$, $\{abc, a\}$, the set of all Fortran programs, the set of all English sentences.
  (Note: this definition does not assign any meaning to strings in the language.)

Three major operations for languages

let L and M be languages

1. concatenation ==> $LM = \{xy \mid x \text{ is in } L \text{ and } y \text{ is in } M\}$
e.g., $L\{\varepsilon\} = L$,$\{\varepsilon\}L = L$, $L^0 = \{\varepsilon\}$, $L^i = LLL \cdots L$ (i times), $L\emptyset = \emptyset L = \emptyset$

2. union ==> $L \cup M = \{x \mid x \text{ is in } L \text{ or } x \text{ is in } M\}$
e.g., $L \cup \emptyset = L$,

3. kleen closure (means 'any number of instances') ==> $L^* = L^0 \cup L^1 \cup L^2 \cdots$

e.g., if D is a language and $D = \{0, 1, \dots, 9\}$ then D^* is all strings of digits.

* 4. positive closure (means 'one or more instances of') ==> $L^+ = L^1 \cup L^2 \cup L^3 \cdots$

Regular Expression vs. Regular Language

- Regular Expression is the notation we use to describe regular sets (regular languages).
  e.g. $identifier = letter(letter|digit)^*$ where | means 'or', i.e. union
  (Note: Regular expressions over some alphabet Σ)

- Each regular expression denotes a (regular) language.
  Let r be a regular expression, then $L(r)$ be the (regular) language generated by r.

$$letter = a|b|..|z|A|B|..|Z$$
$$digit = 0|1|2|..|9$$
$$identifier = letter(letter|digit)^*$$

Rules for constructing regular expressions

- 1. $\emptyset$ is a regular expression denoting $\{\,\}$. i.e., $L(\emptyset) = \{\,\}$

- 2. $\boldsymbol{\varepsilon}$ is a regular expression denoting $\{\varepsilon\}$, the language containing only empty string ε. $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$

- 3. For each a in Σ, $\boldsymbol{a}$ is a regular expression denoting $\{a\}$, a single string a. i.e., $L(\boldsymbol{a}) = \{a\}$

- 4. If R and S are regular expressions then $(R)|(S)$, $(R)(S)$, $(R)^*$ are regular expressions denoting union, concatenation, kleen closure of R and S.

Rules of Regular Expressions

- $R|S = S|R$
- $R|(S|T) = (R|S)|T$
- $R(S|T) = RS|RT, (S|T)R = SR|TR$
- $R(ST) = (RS)T$
- $\varepsilon R = R\varepsilon = R$

* Precedence: ( ), $*$, $\cdot$, |

What do the following regular expressions mean?

Let alphabet= $\{a, b\}$ // Thus the regular expression a denotes $\{a\}$ which is different from just the string $\boldsymbol{a}$.

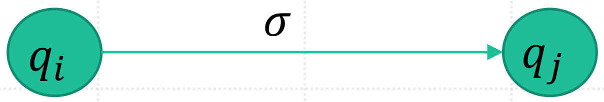
- $\boldsymbol{a}^*$?
- $(\boldsymbol{a}|\boldsymbol{b})^*$? // all strings of a's and b's, including the empty string.
- $\boldsymbol{a}|\boldsymbol{b}\boldsymbol{a}^*$?
- $(\boldsymbol{aa}|\boldsymbol{ab}|\boldsymbol{ba}|\boldsymbol{bb})^*$? // all strings of even length
- $\boldsymbol{\varepsilon}|\boldsymbol{a}|\boldsymbol{b}$? // all strings of length 0 or 1
- $(\boldsymbol{a}|\boldsymbol{b})(\boldsymbol{a}|\boldsymbol{b})(\boldsymbol{a}|\boldsymbol{b})(\boldsymbol{a}|\boldsymbol{b})^*$ and $\boldsymbol{\varepsilon}|\boldsymbol{a}|\boldsymbol{b}|(\boldsymbol{a}|\boldsymbol{b})(\boldsymbol{a}|\boldsymbol{b})(\boldsymbol{a}|\boldsymbol{b})^*$?

- Question: Show how to denote all strings of length not 2.

- Question: Show how to denote the comments of C language.

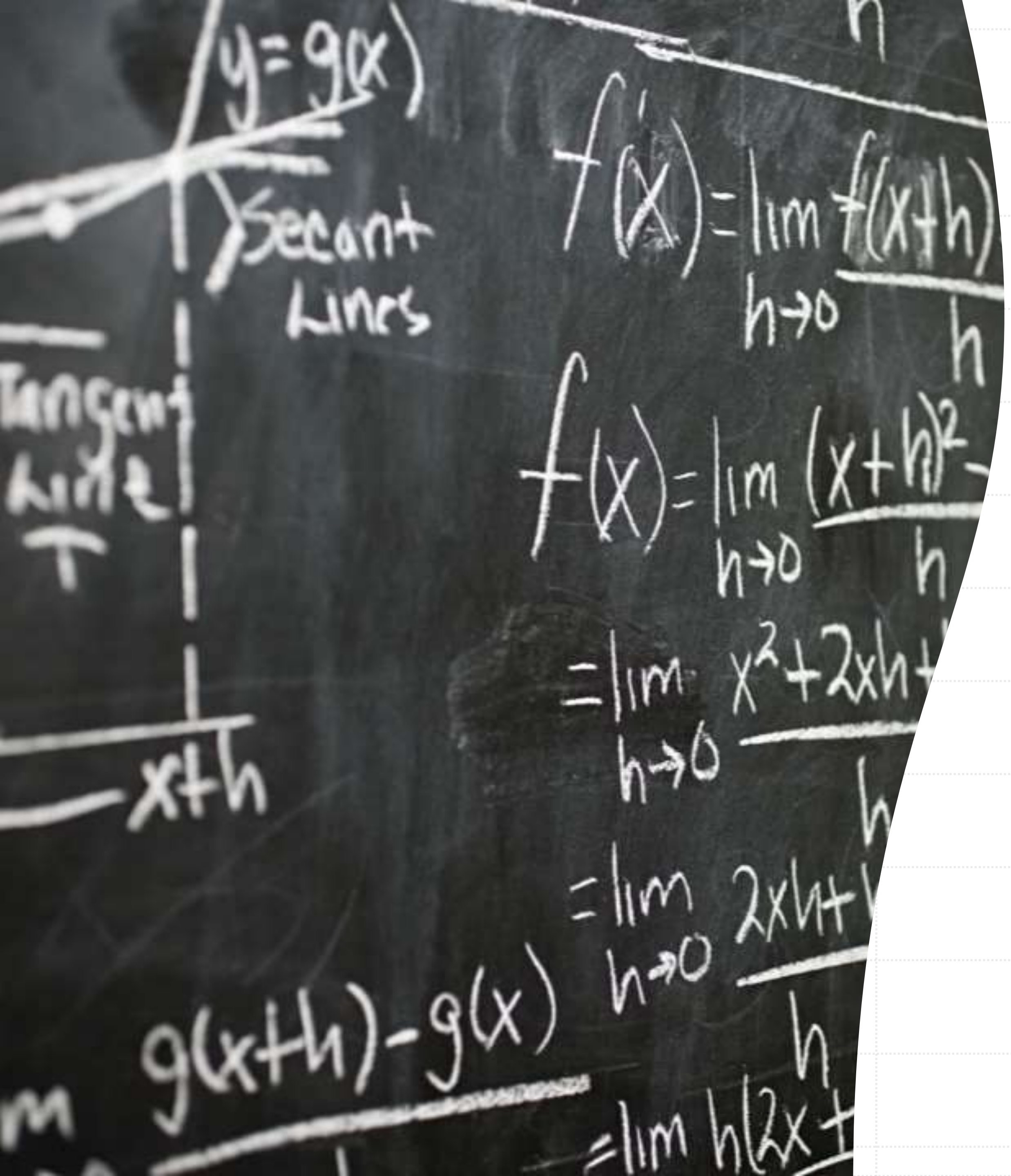
Finite Automata (Finite state machine)

- Depict FA as a directed graph
- each vertex corresponds to a state
- there is an edge from the vertex corresponding to state q_i to vertex corresponding to state q_j iff the FA has a transition from state q_i to q_j on input σ.
- i.e.,



$$\begin{tikzcd} q_i \arrow[r, "\sigma"] & q_j \end{tikzcd}$$

- The language defined by an F. A. (Finite Automata) is the set of input string it accepts.
- regular expression --> non-deterministic F. A. --> deterministic F. A. --> regular expression



Deterministic Finite Automata

- It has no transition on input ε
- For each state s and input symbol a, there is at most one edge labeled a leaving $\boldsymbol{s}$.
- $M = (Q, \Sigma, \delta, q_0, F)$, where
  Q = a finite set of states
  Σ = a set of finite characters
  δ= transition function $\delta: Q \times \Sigma \rightarrow Q$
  q_0 = starting state
  F = a set of accepting states (a subset of Q i.e. $F \subseteq Q$)

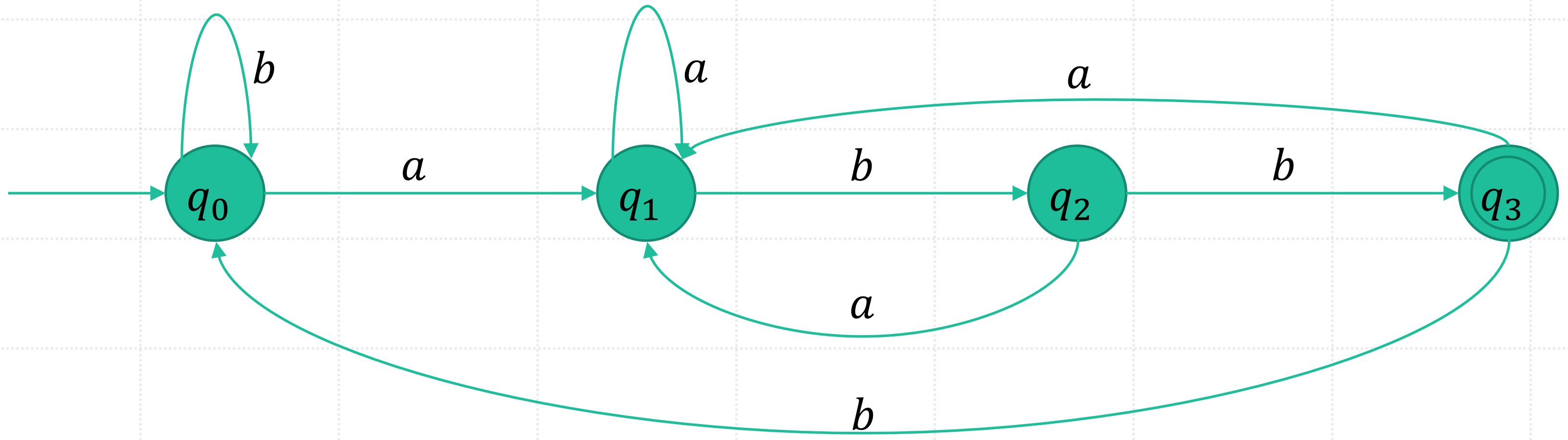
e.g. DFA accepting (a|b)*abb

$Q = \{q_0, q_1, q_2, q_3\}$

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

q_0 =starting state

$F = \{q_3\}$


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "b"] \arrow[r, "a"] & q_1 \arrow[loop above, "a"] \arrow[r, "b"] & q_2 \arrow[r, "b"] \arrow[l, bend left, "a"] & q_3 \arrow[ll, bend right, "a"'] \arrow[lll, bend left, "b"]
\end{tikzcd}$$

$\delta(q_0, a) = q_1$ $\delta(q_0, b) = q_0$ $\delta(q_1, a) = q_1$

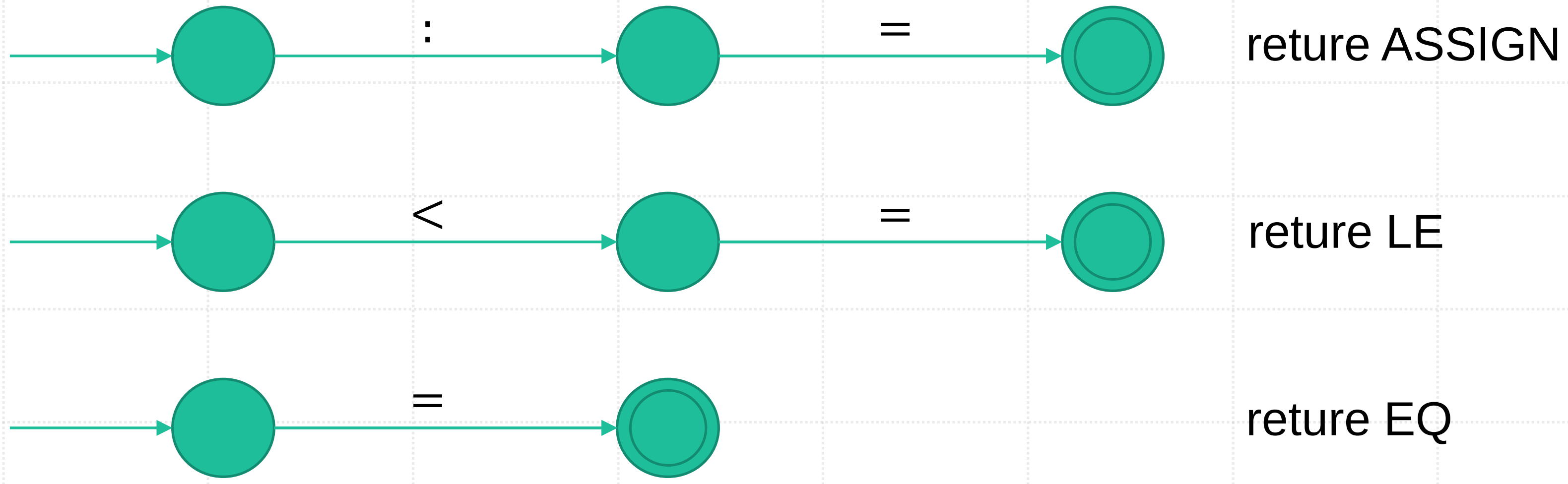
$\delta(q_1, b) = q_2$ $\delta(q_2, a) = q_1$ $\delta(q_2, b) = q_3$

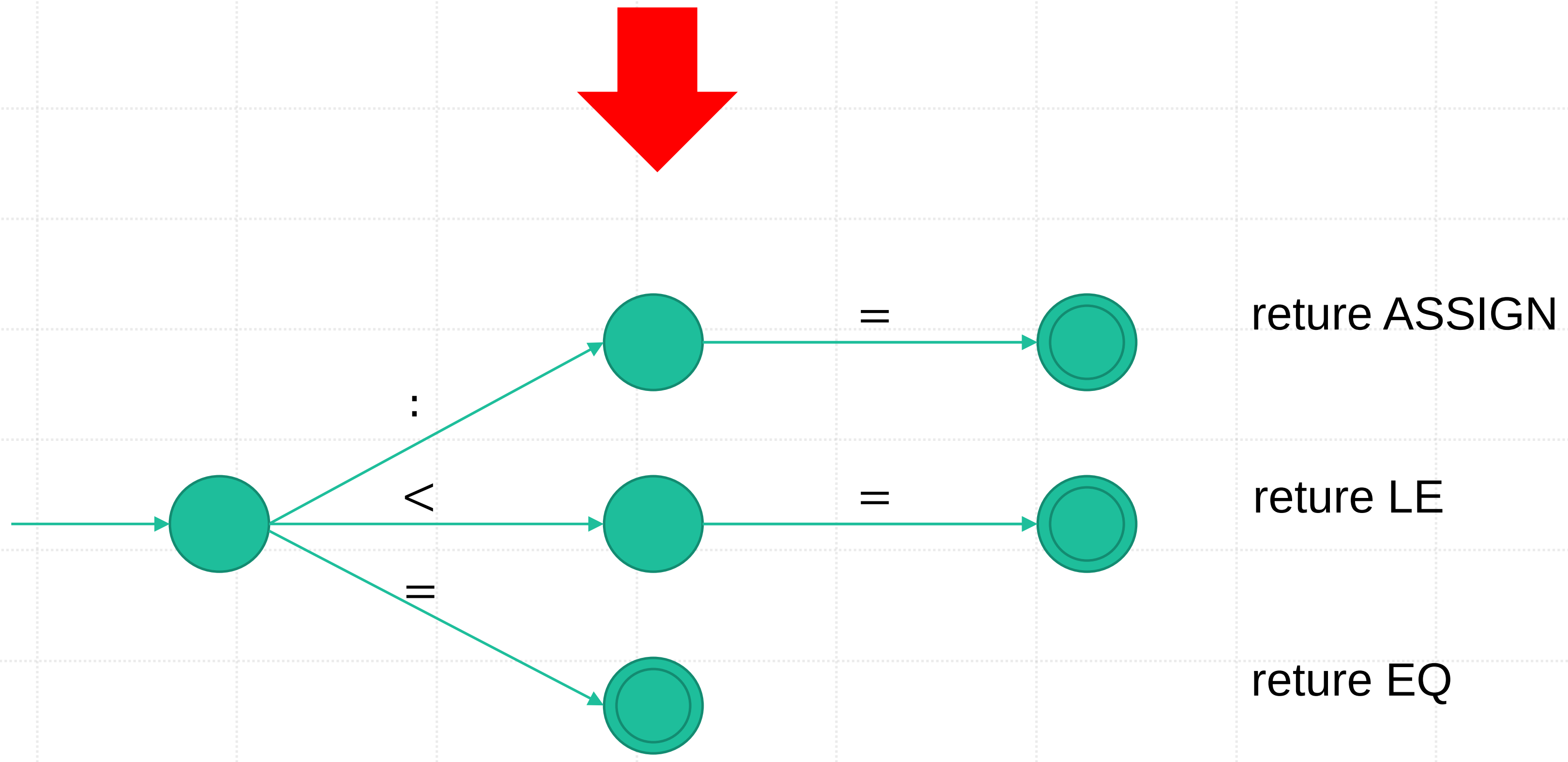
$\delta(q_3, a) = q_1$ $\delta(q_3, b) = q_0$

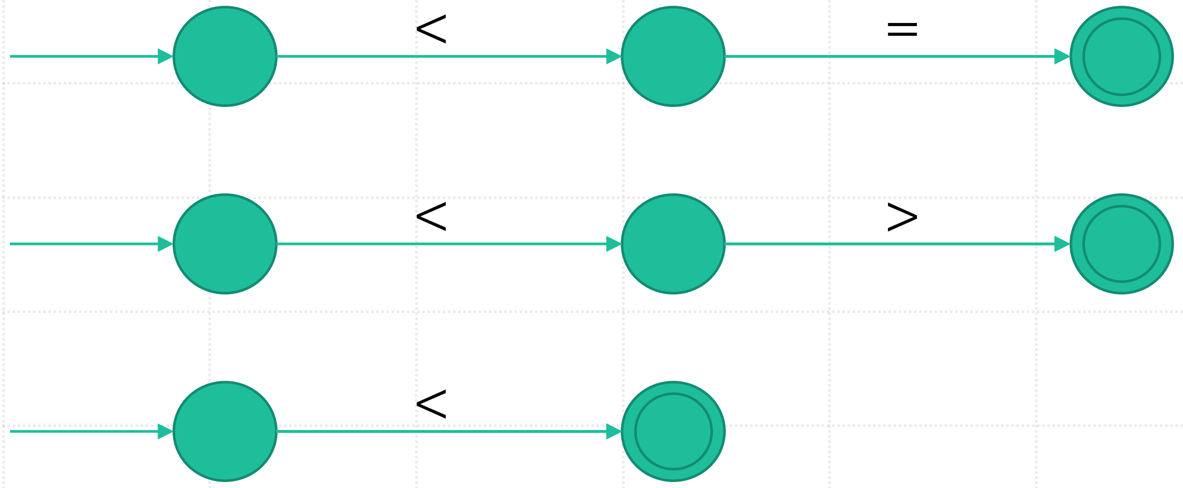
- Given a string x, let $\delta(q_0, x)$ be the final state in the path where the symbols along the path spell out x.

- A string x is accepted by a F.A. if $\delta(q_0, x) \in F$

- So, the language accepted by M is $L(M) = \{x | \delta(q_0, x) \in F\}$


:
=
reture ASSIGN
<
=
reture LE
=
reture EQ

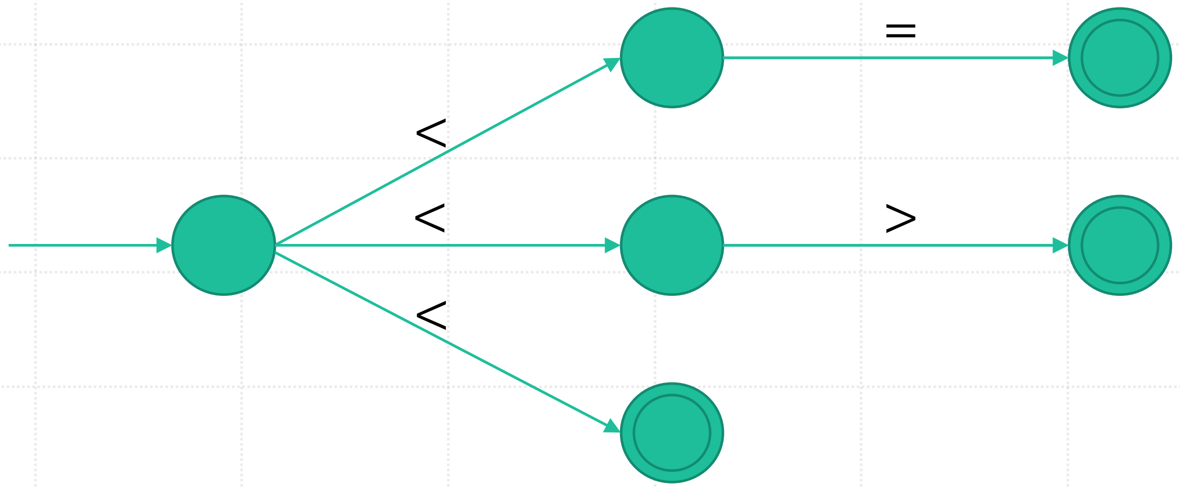

=
reture ASSIGN
:
<
=
reture LE
=
reture EQ



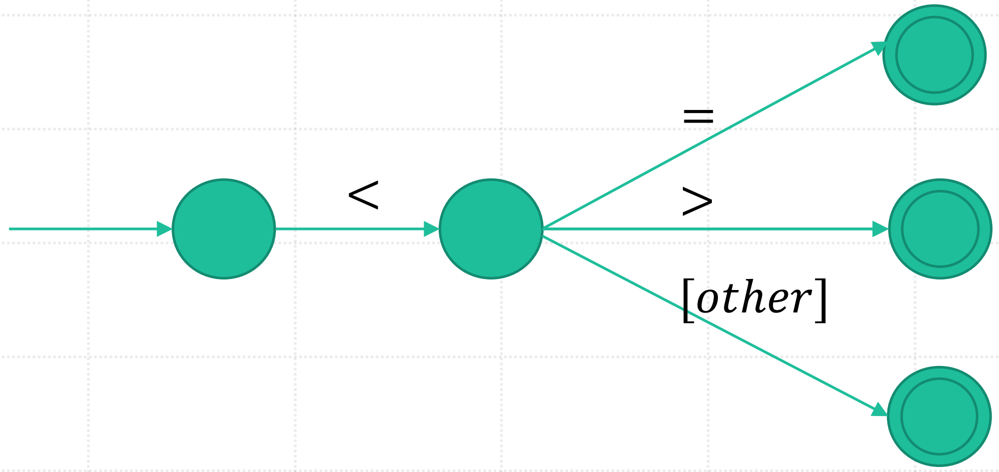
reture LE

reture NE

reture LT



It is not a DFA!!



complexity ↑

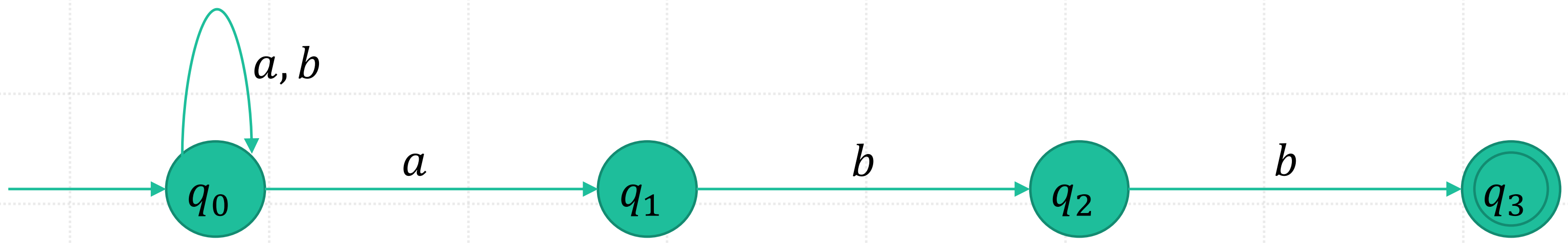
if we want to combine multiple DFA

Non-deterministic Finite Automata

- $M' = (Q', \Sigma, \delta', q_0, F)$, where
  Q' = a finite set of states
  Σ = a set of finite characters
  δ'= transition function $\delta': Q' \times (\Sigma \cup \varepsilon) \rightarrow 2^{|Q'|}$ $\text{where} 2^{|Q'|} \text{means it can map to multiple states}$
  q_0 = starting state
  F = a set of accepting states (a subset of Q' i.e. $F \subseteq Q'$)

* Given a string x belonging to Σ^* if there exists a path in a NFA M' s.t. M' is in a final state after reading x, then x is accepted by M'. i.e., if $\delta'(q_0, x) = \alpha \subseteq Q'$ and $\alpha \cap F \neq \emptyset$ then x belongs to $L(M')$.

e.g. NFA accepting (a|b)*abb


$$\begin{tikzcd} \arrow[r] & q_0 \arrow[loop above, "{a,b}"] \arrow[r, "a"] & q_1 \arrow[r, "b"] & q_2 \arrow[r, "b"] & q_3 \end{tikzcd}$$

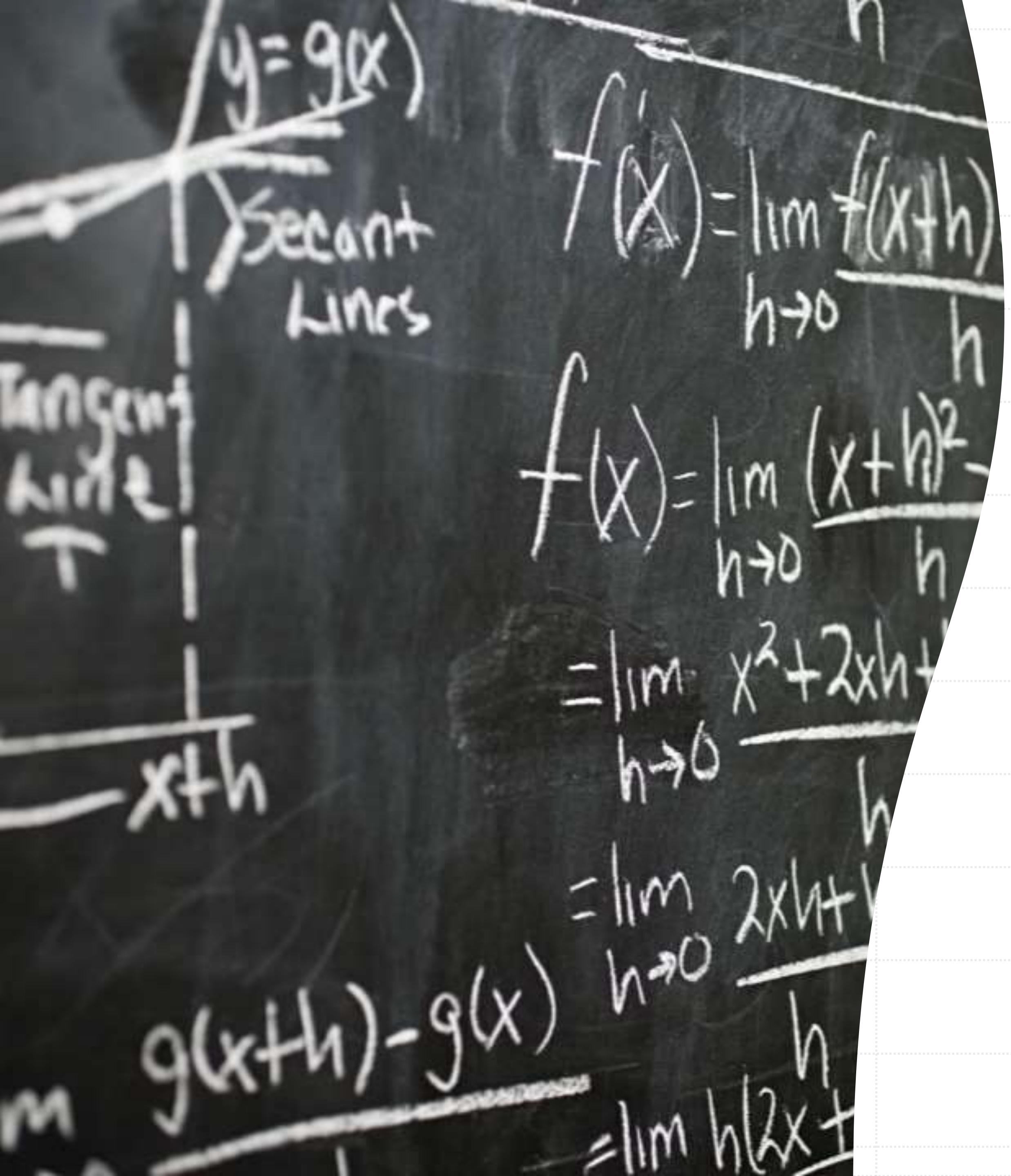
- $Q' = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, q_0 = starting state
- $F = \{q_3\}$
- $\delta(q_0, a) = \{q_0, q_1\}, \delta(q_0, b) = \{q_0\}, \delta(q_1, b) = \{q_2\}$
- $\delta(q_2, b) = \{q_3\}$

Regular Definition

- Give names to regular expressions and then define regular expressions using these names as if they were symbols. A sequence of definitions of the form

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

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



An Example

$$letter = a|b|..|z|A|B|..|Z$$
$$digit = 0|1|2|..|9$$
$$identifier = letter(letter|digit)^*$$

- Lexical Analyzer: May return token's lexical category (an index) and value (in global variable) or token's lexical category only

- e.g.

```
letter(letter|digit)*  {yylval = install(); return ID;}
digit+                   {yylval = install(); return NUM;}
"<"                       {yylval = LT; return RELOP;}
```

Trick for differentiating keyword and identifier

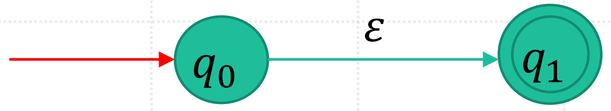
- install all keywords in symbol table first

Theorem

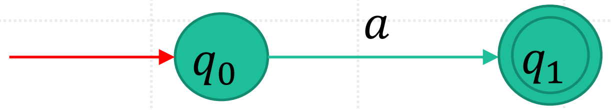
- The followings are equivalent:
  - regular expression
  - NFA
  - DFA
  - regular language
  - regular grammar

Algorithm: Constructing an NFA from a regular expression

- 1. For ε we construct the NFA ==>



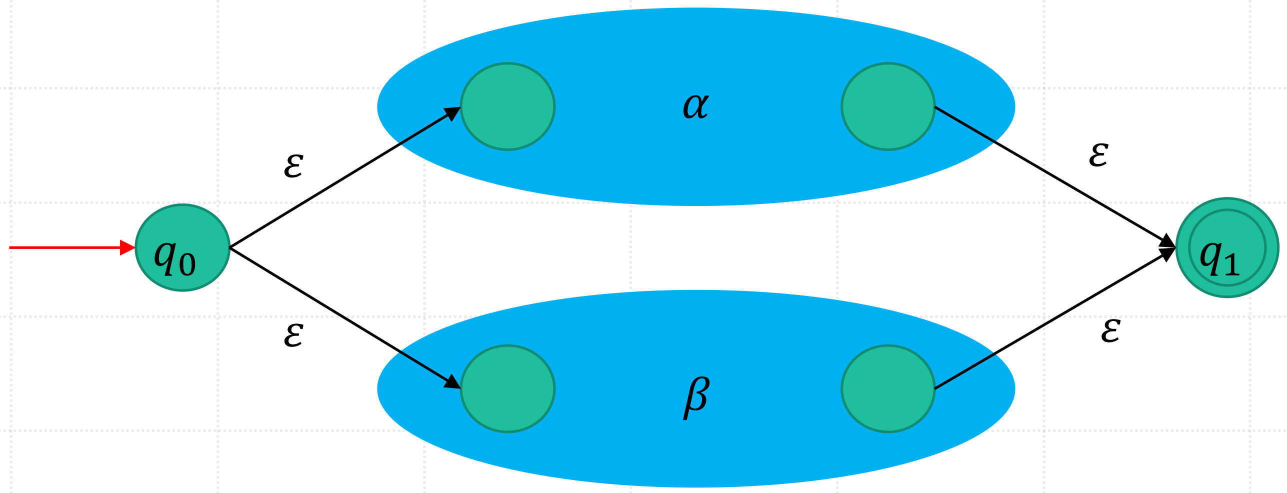
- 2. For a in Σ we construct the NFA ==>



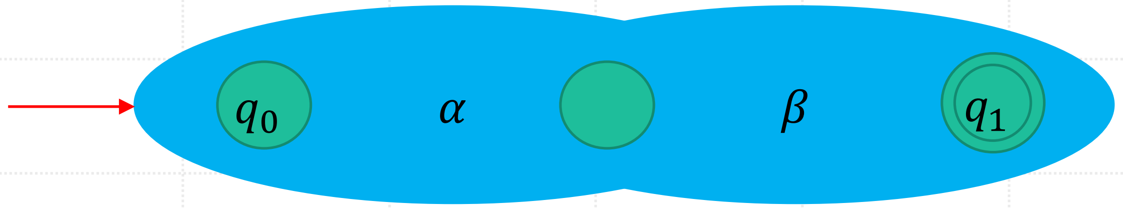
- 3. Proceed to combine them in ways that correspond to the way compound regular expressions are formed, i.e., transition diagram for α, β.

$$\alpha|\beta$$
$$\alpha\beta$$
$$\alpha^*$$
$$\alpha^+$$

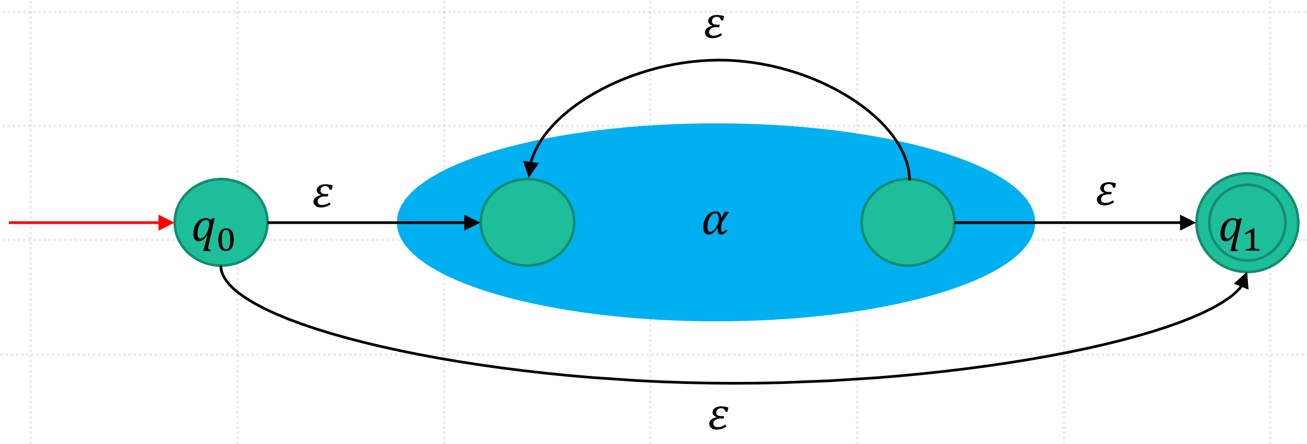
$\alpha|\beta$



$\alpha\beta$



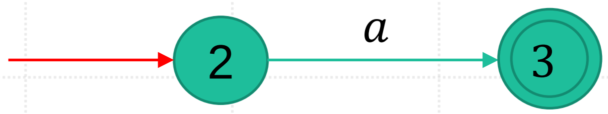
α^*



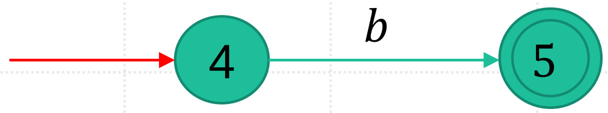
Example:

- Construct the r.e. $(a|b)^*abb$ based on the above algorithm.

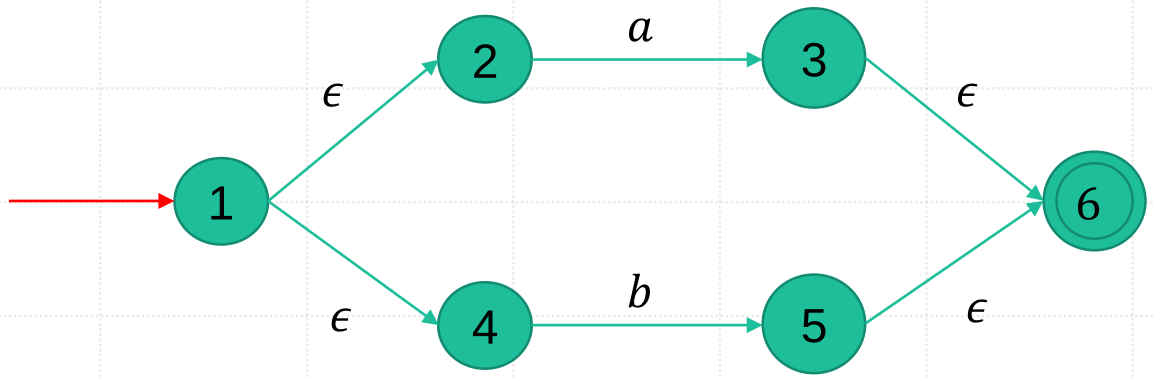
a



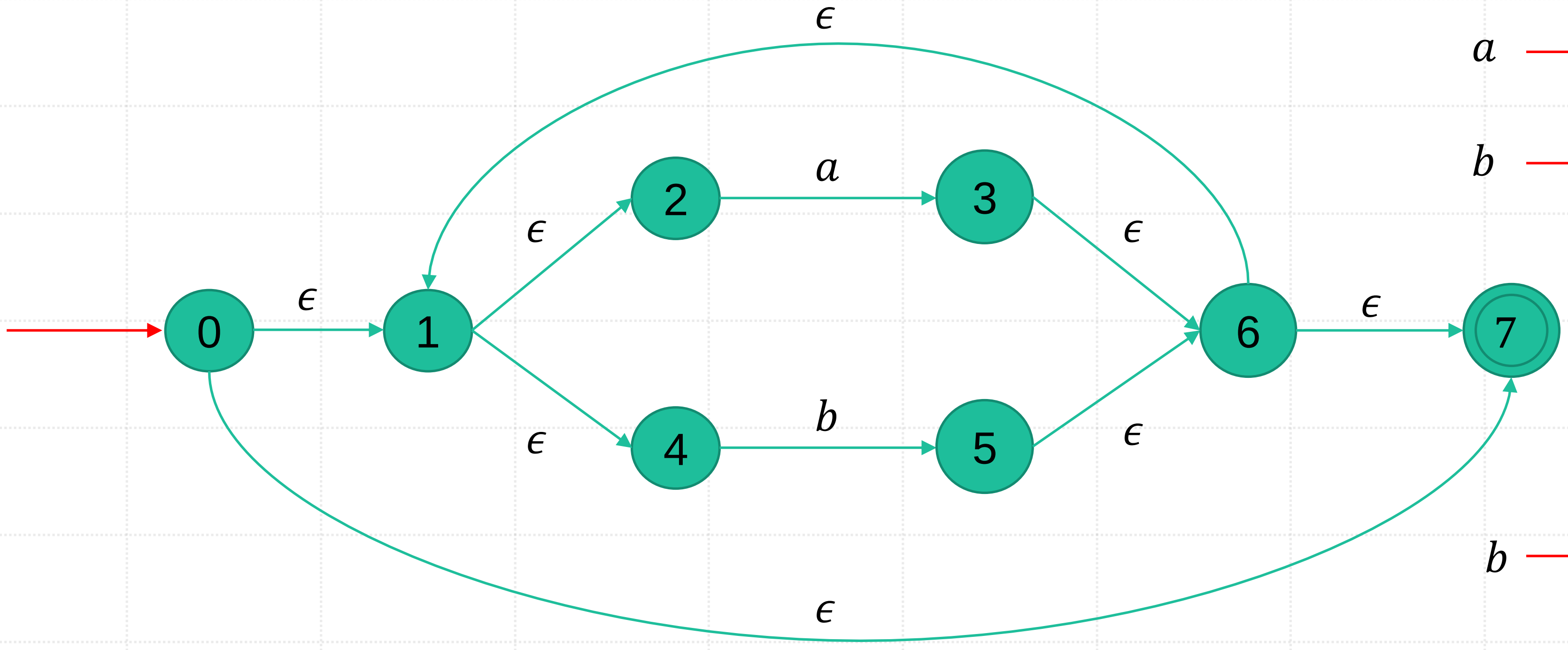
b

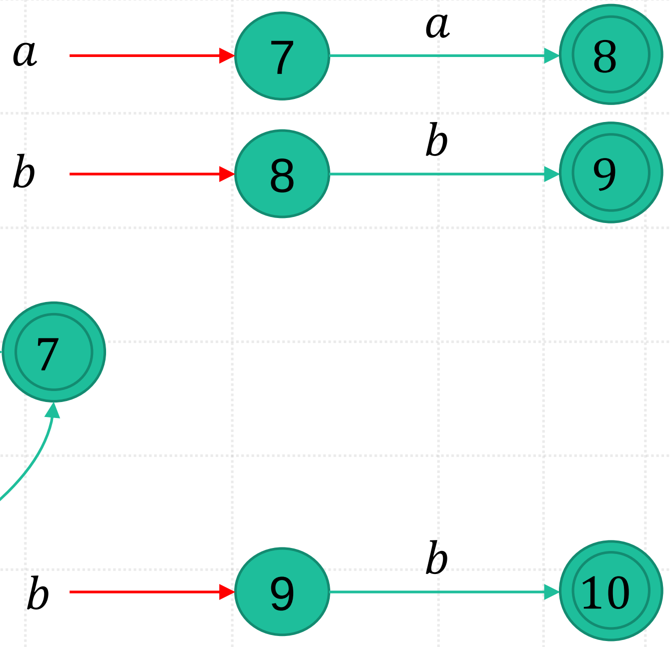


$a|b$

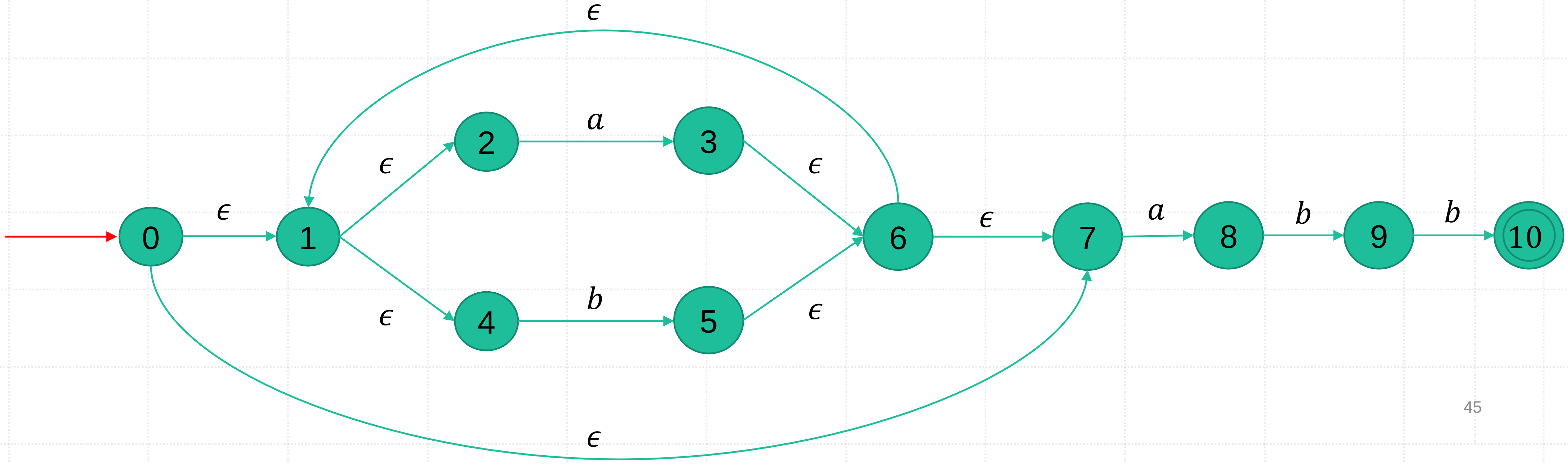


$(a|b)^*$


0
1
2
3
4
5
6
7
ϵ
a
b


a
7
a
8
b
8
b
9
b
9
b
10

$(a|b)^*abb$


0
1
2
3
4
5
6
7
8
9
10
ϵ
a
b
a
b
b

How to construct a DFA based on an equivalent NFA?

- Each state of the DFA is a set of states which the NFA could be in after reading some sequence of input symbols.

- The initial state of the DFA is the set consisting of 0, the initial state of NFA, together with all states of NFA that can be reached from 0 by means of ε-transition.

ε-closure(s)

- Def. The set of states that can be reached from s on ε-transition alone.

- Method:
  1. s is added to ε-closure(s).
  2. if t is in ε-closure(s), and there is an edge labeled ε from t to u then u is added to ε-closure(s) if u is not already there.

- ε-closure(T) : The union over all states s in T of ε-closure(s). (Note: T is a set of states)

- move(T, a): a set of NFA states to which there is a transition on input symbol a from some state s in T.

Algorithms

- 1. Computation of ε-closure
- 2. The subset construction
  Given a NFA $M = (Q, \Sigma, \delta, q_0, F)$, construct a corresponding DFA $M' = (Q', \Sigma, \delta', q_0, F')$
  (1) Compute the ε-closure(s), s is the start state of M. Let it be S. S is the start state of M' and $Q' = \{S\}$
  (2) ….. (see next page)
  (3) $F' = \{q' \in Q' | q' \cap F \neq \emptyset\}$

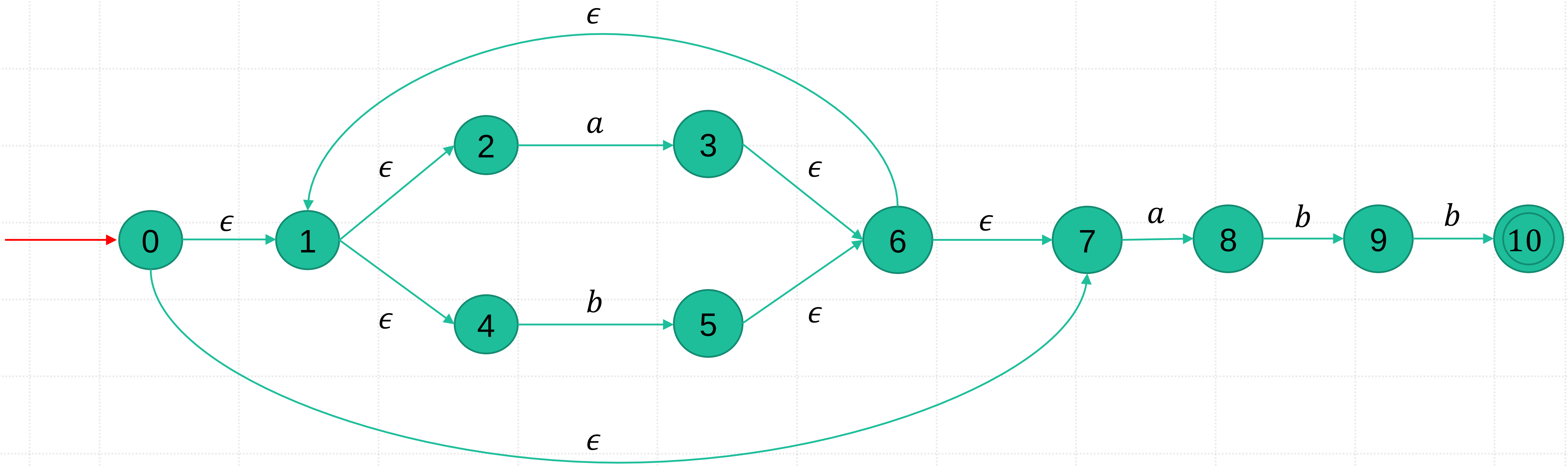
```
initially, $\varepsilon$-closure($s$) is the only state in $Q'$, and it is unmarked;
while ( there is an unmarked state $T$ in $Q'$) {
    mark $T$;
    for ( each input symbol $a \in \Sigma$) {
        $U = \varepsilon$-closure(move($T, a$));
        if ($U$ is not in $Q'$)
            add $U$ as an unmarked state to $Q'$;
        $\delta'(T, a) = U$;
    }
}
$F' = \{q' \in Q' | q' \cap F \neq \emptyset\}$;
```

Computing ε-closure(T)

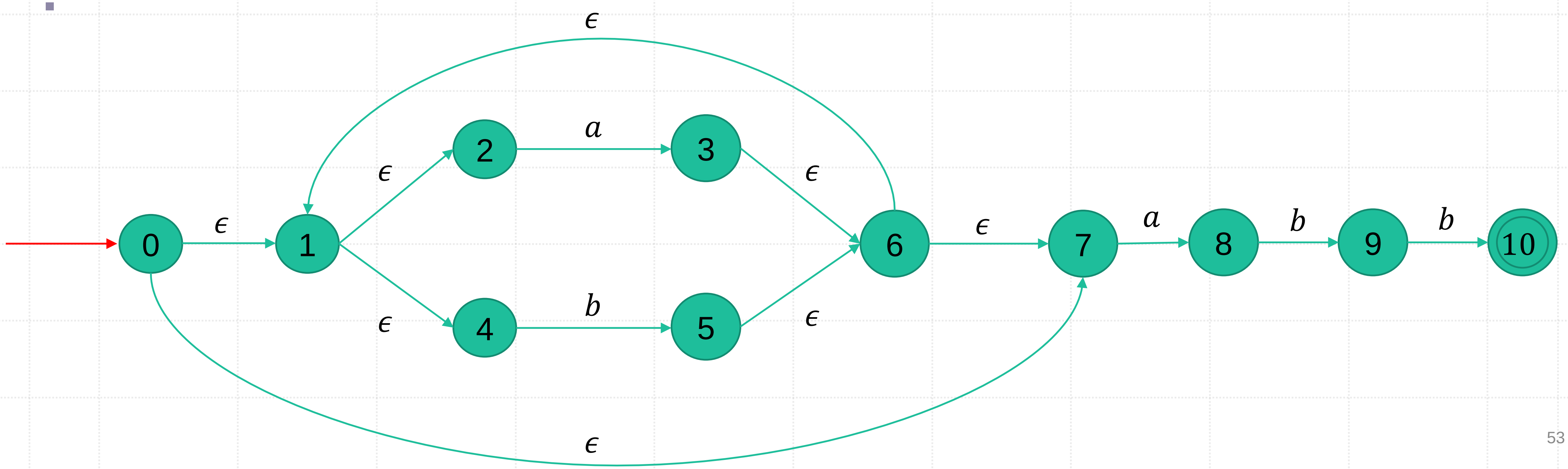
```
push all states of $T$ onto stack;
initialize $\varepsilon$-closure($T$) to $T$;    %% a path can have zero edges
while ( stack is not empty ) {
        pop $t$, the top element, off stack;
        for ( each state $u$ with an edge from $t$ to $u$ labeled $\varepsilon$ )
                if ( $u$ is not in $\varepsilon$-closure($T$)) {
                        add $u$ to $\varepsilon$-closure($T$) ;
                        push $u$ onto stack ;    %% search a path with $\varepsilon$ edge
                }
}
```

Example: $(a|b)^{*}abb$

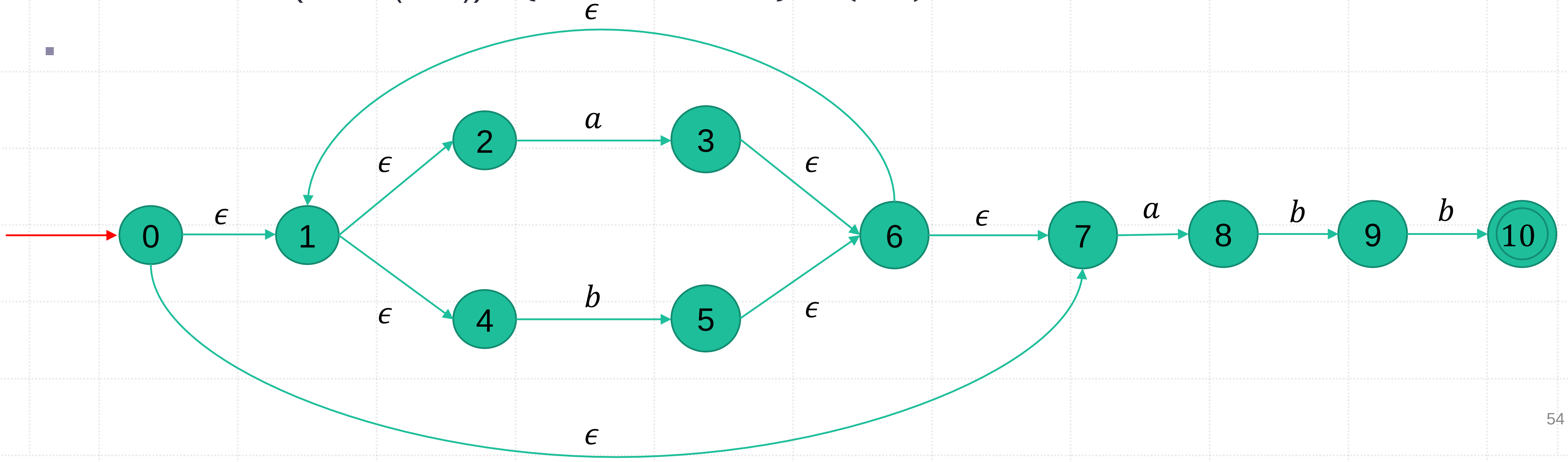
- The start state A of the equivalent DFA is ε-closure(0), or $A = \{0, 1, 2, 4, 7\}$



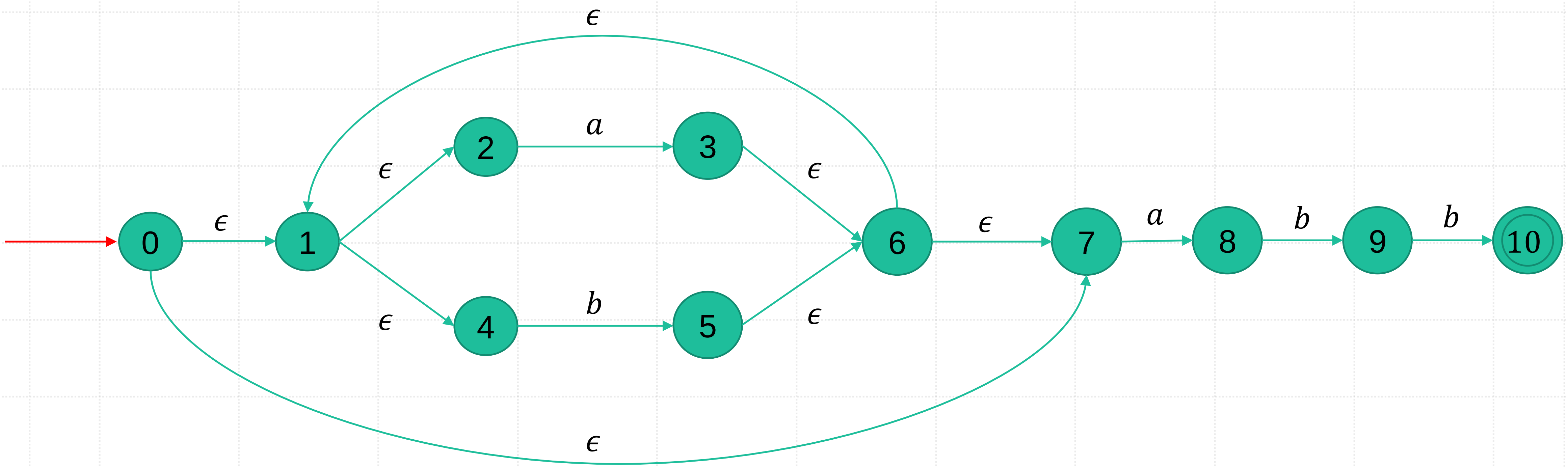
- $A = \{0, 1, 2, 4, 7\}$
- move(A, a)= $\{3, 8\}$, move(A, b)= $\{5\}$
- $B = \varepsilon$-closure(move(A, a))= $\{1, 2, 3, 4, 6, 7, 8\}$, $\delta'(A, a) = B$
  $C = \varepsilon$-closure(move(A, b))= $\{1, 2, 4, 5, 6, 7\}$, $\delta'(A, b) = C$
- move(B, a)= $\{3, 8\}$, move(B, b)= $\{5, 9\}$
- $\delta'(B, a) = B$
  $D = \varepsilon$-closure(move(B, b))= $\{1, 2, 4, 5, 6, 7, 9\}$, $\delta'(B, b) = D$
-



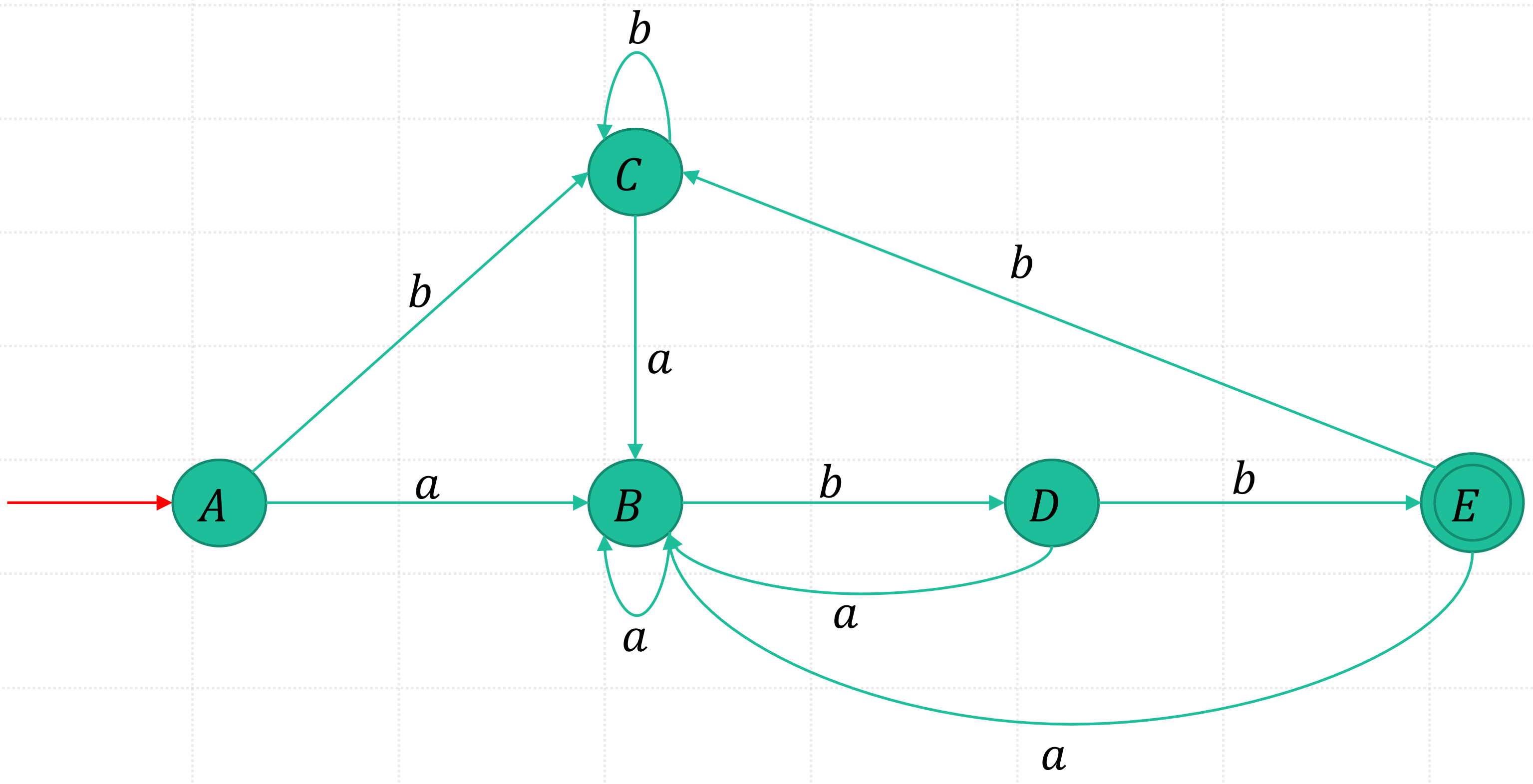
- $C = \{1, 2, 4, 5, 6, 7\}$
- move(C, a)= $\{3, 8\}$, move(C, b)= $\{5\}$
- ε-closure(move(C, a))= B, $\delta'(C, a) = B$
  ε-closure(move(C, b))= C, $\delta'(C, b) = C$
- $D = \{1, 2, 4, 5, 6, 7, 9\}$
- move(D, a)= $\{3, 8\}$, move(D, b)= $\{5, 10\}$
- ε-closure(move(D, a))= B, $\delta'(D, a) = B$
  $E = \varepsilon$-closure(move(D, b))= $\{1, 2, 4, 5, 6, 7, 10\}$, $\delta'(C, b) = E$
-



- $E = \{1, 2, 4, 5, 6, 7, 10\}$
- move(E, a)= $\{3, 8\}$, move(E, b)= $\{5\}$
- ε-closure(move(E, a))= B, $\delta'(E, a) = B$
  ε-closure(move(E, b))= C, $\delta'(E, b) = C$
-



NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C



Minimizing the number of states of DFA

Principle: only the states in a subset of NFA that have a non-ε-transition determine the DFA state to which it goes on input symbol.

- Two subsets can be identified as one state of the DFA, provided:
  1. they have the same non-ε-transition-only states
  2. they either both include or both exclude accepting states of the NFA

Algorithm:

- If there are two accepting states s and t that have transitions on a that land in different sets, then no a-transition can be defined for this grouping of the states. We say that a distinguishes the states s and t.
- Two states are distinguished if there exists a string which leads to final states and non-final states when it is fed to the two states.
- Find all groups of states that can be distinguished by some input string.

1. Construct a partition and repeat the algorithm of getting a new partition.
2. Pick a representative for each group.
3. Delete a dead state and states unreachable from the initial state. Make the transition to the dead states undefined.

e.g. $(a|b)^*abb$

1. $(A\ B\ C\ D)(E)$ $\leftarrow$ accepting state

2. $(A\ B\ C)(D)(E)$

3. $(A\ C)(B)(D)(E)$

DFA STATE	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

$\Downarrow$

DFA STATE	a	b
A	B	C
B	B	D
D	B	E
E	B	C

Steps for developing a lexical analyzer:

1. For each of these regular expression we construct an NFA with a single start and single accepting state.
2. Piece these together into a single NFA
   a. add a start state and epsilon-transitions from it to the start state of the individual NFA
   b. index the accepting state with the lexical category that they accept.

3. Convert the NFA to produce a corresponding DFA
   a. A DFA state accepts for a given lexical category iff it contains a NFA accepting state for that category.
   b. If more than one such category we allow only the one of highest precedence.
   c. either the state or the previous states containing one or more final states of the NFA.
4. Minimize the number of states of the DFA

An Example:

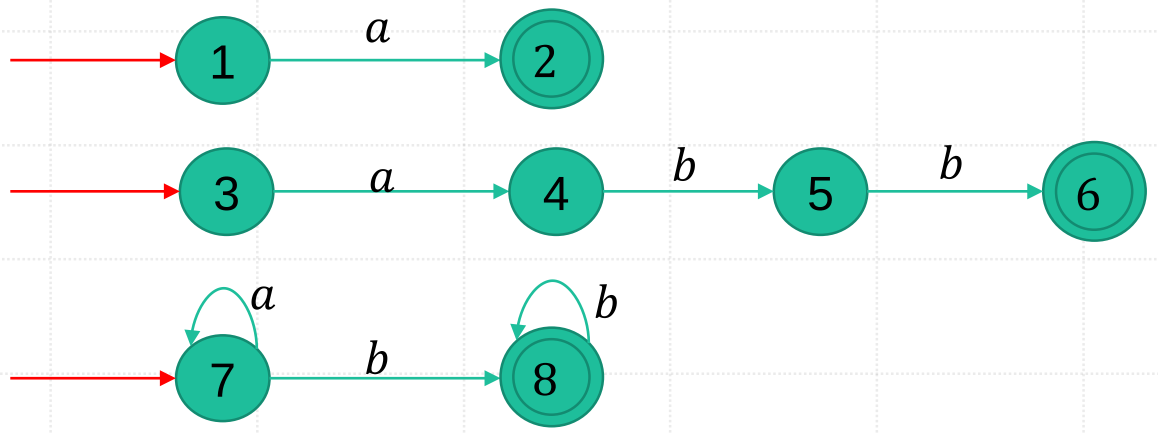
- Build a lexical analyzer for 3 tokens denoted as:

  a // regular expression

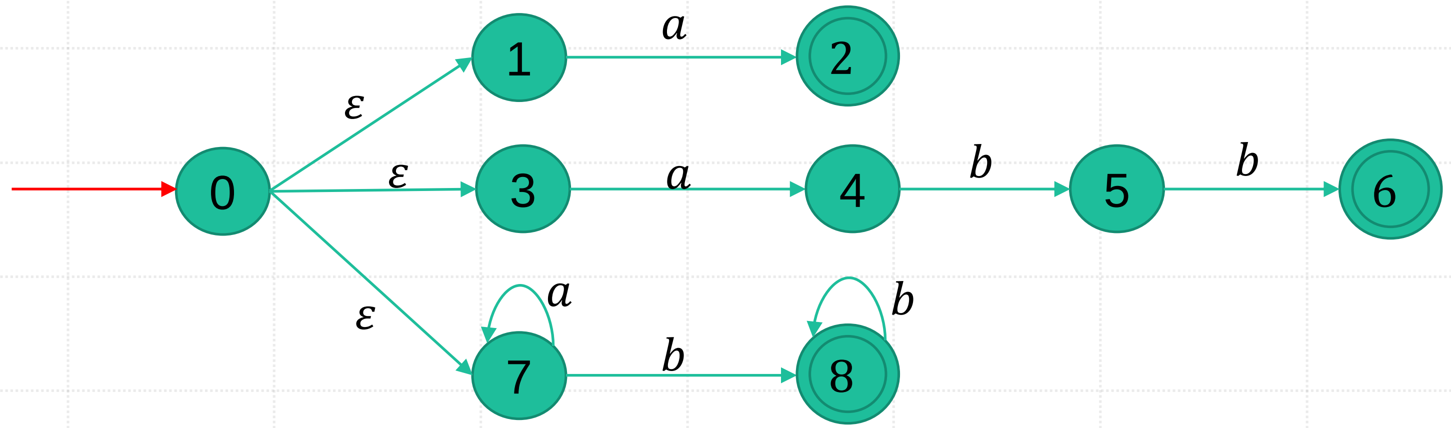
  abb // regular expression

  a^*b^+ // regular expression

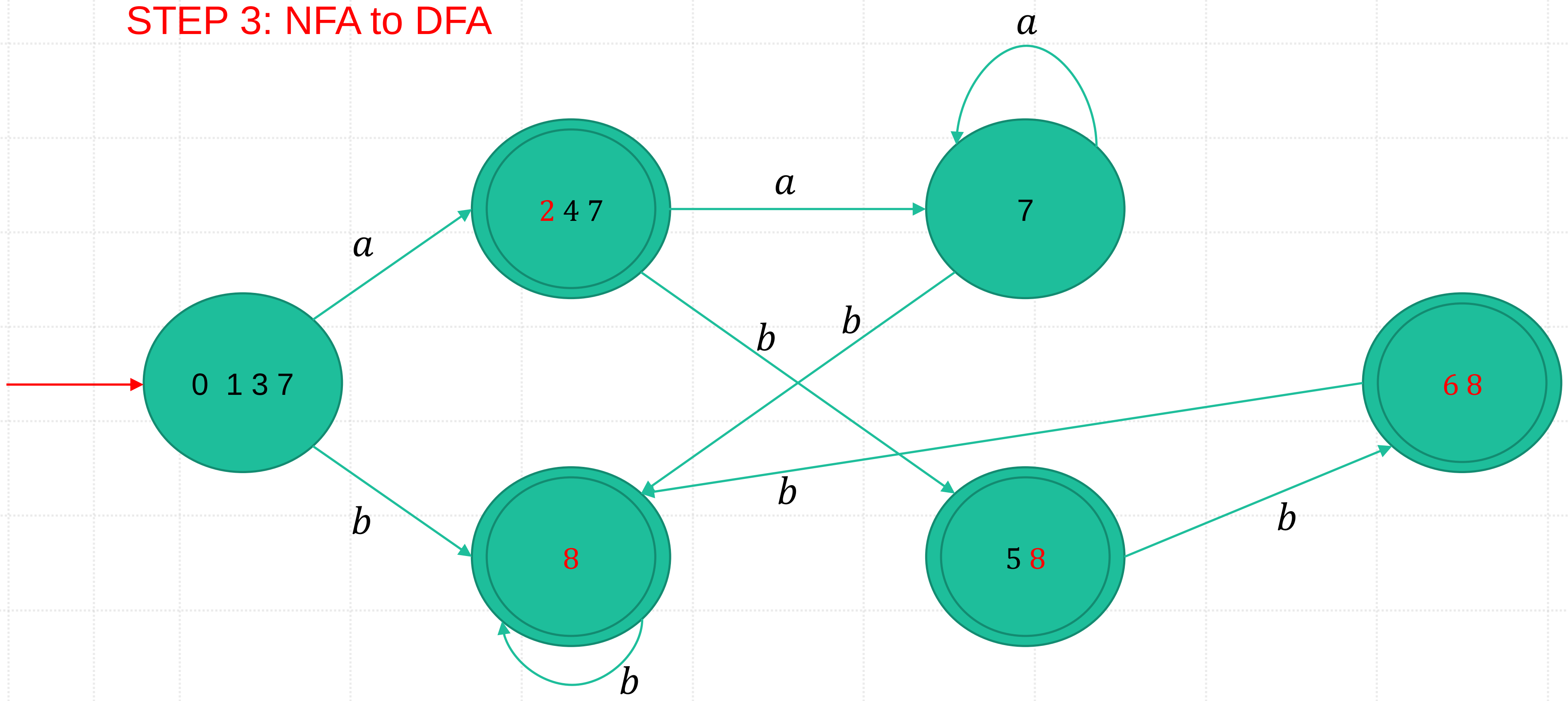
STEP 1: Construct NFA for each regular expression


$$\begin{tikzcd} \arrow[r] & 1 \arrow[r, "a"] & \boxed{2} & & \\ \arrow[r] & 3 \arrow[r, "a"] & 4 \arrow[r, "b"] & 5 \arrow[r, "b"] & \boxed{6} \\ \arrow[r] & 7 \arrow[loop above, "a"] \arrow[r, "b"] & \boxed{8} \arrow[loop above, "b"] & & \end{tikzcd}$$

STEP 2: NFA recognizing three different tokens

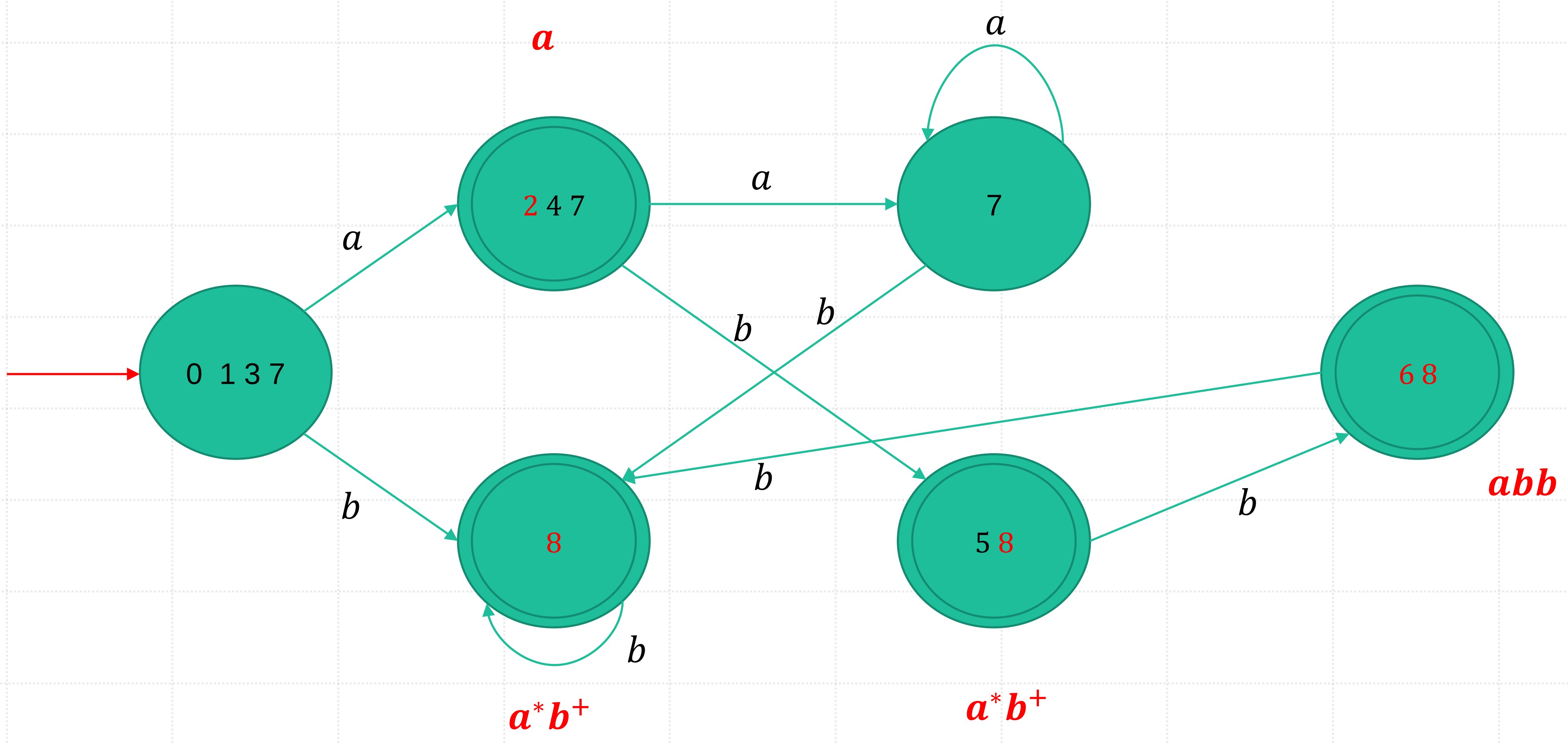


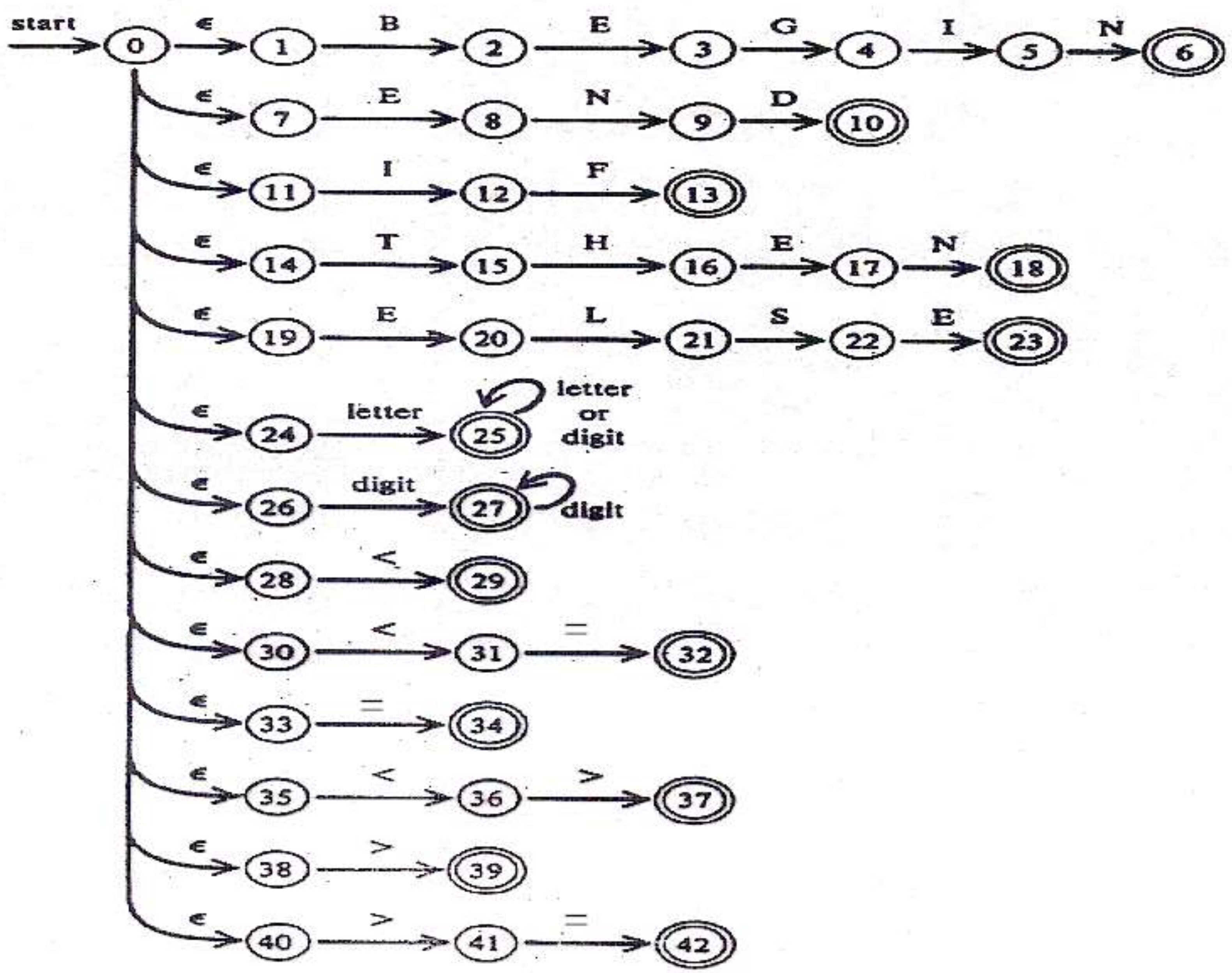
STEP 3: NFA to DFA



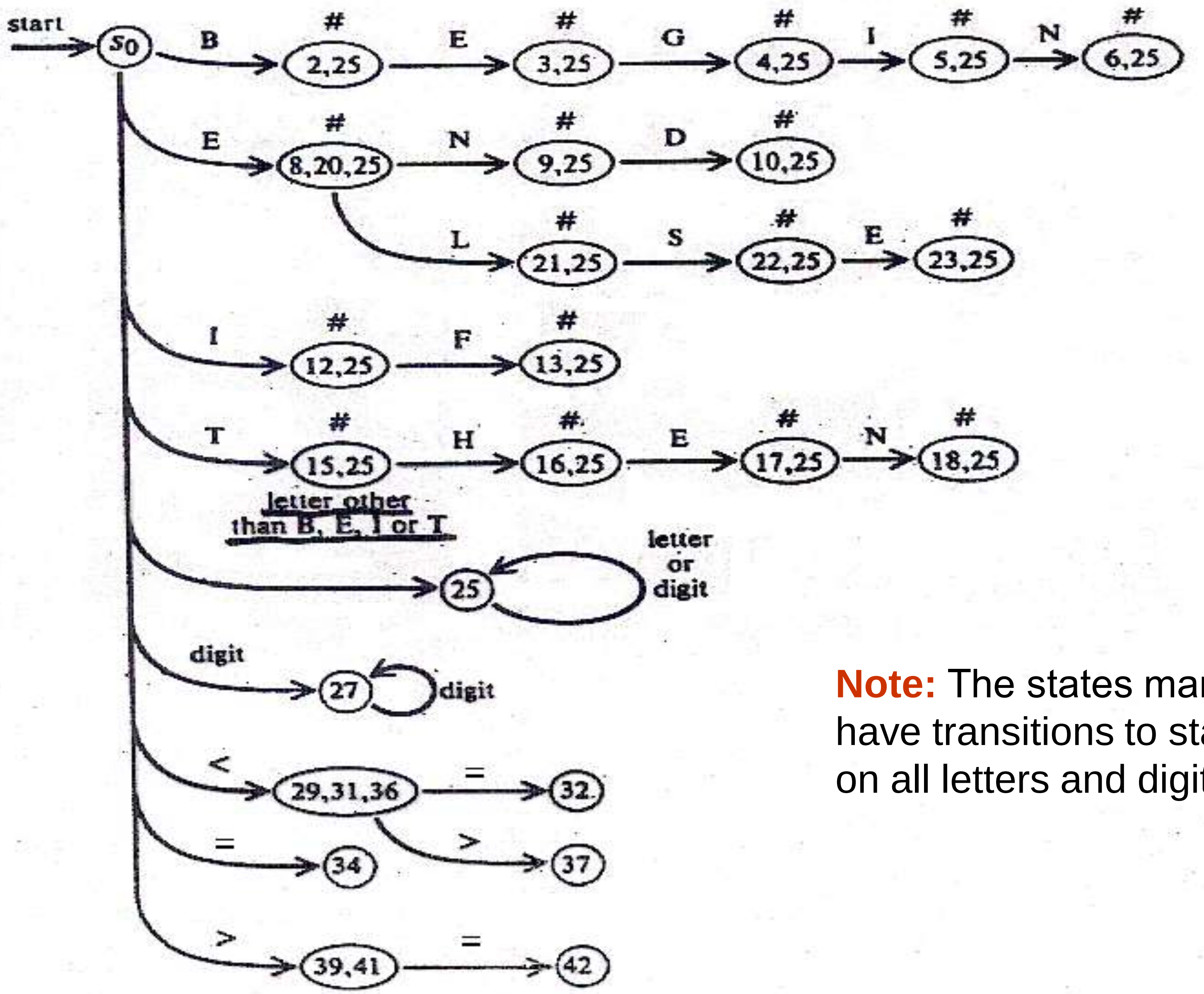
STEP 4: Minimizing the number of states of DFA (No change)

1. Suppose the input character stream is: $aba \rightarrow a^*b^+$ (token recognized ab)
2. Suppose the input character stream is: $abb \rightarrow abb$ (token recognized)





Combined NFA for tokens:



DFA constructed from NFA

Note: The states marked '#' have transitions to state {25} on all letters and digits.

Problems:

1. Y = X + 1

    CFG1: id = function + id

    CFG2: id = id + id

    Ans: Make things as easy as possible for the parser. It should be left to scanner to determine if X is a variable or a function.

2. When to quit? X <> Y

    Ans: Go for longest possible fit

Problems:

3. What if there is a tie for the tokens? e.g., an input which matches patterns for identifier and a keyword
   Ans: Given a precedence to the lexical category. (Usually first mentioned has highest priority)

4. What if you don't know until after the end of the longest pattern?
   Ans: Put a slash at the point where the scanning resumes if it is the desired token, other than the end of the pattern.
   e.g. in Fortran DO 99 I = 1, 100 → DO99I=1,100
   DO 99 I = 1.100 → DO99I=1.100
   So,
   do/({letter}|{digit})* = ({letter}|{digit})*, for pattern "do"

e.g. if/\(.*\){letter} for pattern "if"   (if(i,j) = 3 is legal in Fortran)

That is, we mark the restart position in the regular expression by a slash.

```
for (i=1; i<=100; i++)
   {
       x = …
       y = …
       ……..
       ……..
   }
```

$\Rightarrow$

```
DO 99 I = 1, 100
    X = …
    Y = …
    ……..
    …….
99  Continue
```

When converting the NFA to DFA for such regular expression we create a special state for each “/” transition.

e.g. $$\begin{tikzcd} 1 \arrow[r, "/"] & 2 \end{tikzcd}$$ becomes $$\begin{tikzcd} 1 \arrow[r, "\varepsilon"] & * \arrow[r, "\varepsilon"] & 2 \end{tikzcd}$$ ("*" denotes a state)

$$\begin{tikzcd} 0 \arrow[r, "D"] & 1 \arrow[r, "O"] & 2 \arrow[r, "\varepsilon"] & * \arrow[r, "\varepsilon"] & 3 \arrow[r, "\text{letter}"] & 4 \end{tikzcd}$$

becomes

$$\begin{tikzcd} 0 \arrow[r, "D"] & 1 \arrow[r, "O"] & 2,*,3 \arrow[r, "\text{letter}"] & 4 \end{tikzcd}$$

Note:

We associate the corresponding rule with each of these lookahead state.

A DFA state containing one of these is a lookahead state for the corresponding rule. Note, a DFA state may be a lookahead state for more than one rule and a given rule may have several lookahead states.

Format of a Lex Input File

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

Examples

```
%{
/* a lex program that adds line no. to lines of text, printing the new text to the standard output
*/
#include  <stdio.h>
int lineno = 1;
%}
line   .*\n
%%
{line}     { printf(“%5d %s”, lineno++, yytext); }
%%
main( )
{ yylex ( ); return 0; }
```

```
%{
/* a lex program that changes all numbers from decimal to hexadecimal notation, printing
a summary statistic to  stderr. */
#include  <stdio.h>
#include  <stdlib.h>
int count = 0;
%}
digit    [0-9]
number  {digit}+
%%
{number}     { int n = atoi(yytext); printf(“%x”, n); if (n>9) count++;}
%%
main( )
{ yylex ( ); fprintf (stderr, “number of replacement= %d”, count); return 0; }
```

How to compile Lex program?

% lex file1.l ==> lex.yy.c (% is the Unix prompt)

% cc lex.yy.c ==> a.out   (cc is c compiler)

Note the following Lex names:

- yylex ==> invoke lexical analyzer to read standard input file
- yytext ==> string matched from input
- yyleng ==> length of the string input
- yylineno ==> increment for each new line that is needed

- Question: Show how to denote all strings of length not 2 and its corresponding NFA.