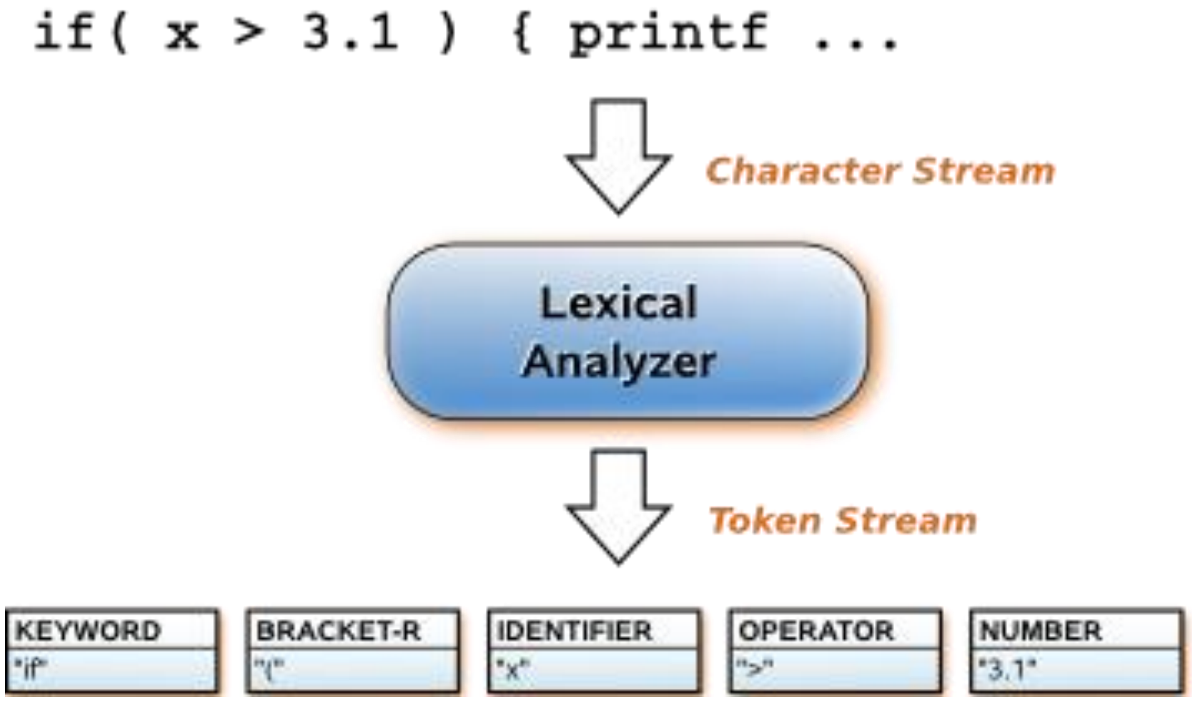
Lecture 2 :

Lexical analysers



Outline

- Lexical analysis
- Regular expression (RE)
- Implementation of Regular Expression

  RE → NFA → DFA → Tables
- Non-deterministic Finite Automata (NFA)

  Converting a RE to NFA
- Deterministic Finite Automata (DFA)
  - Converting NFA to DFA
  - Converting RE to DFA directly

Compiler phases

```
#include<iostream>
using namespace std;

int main() {
cout<<"Enter The Size Of Array:   ";
int size;
cin>>size;


int array[size], key,i;

// Taking Input In Array
    for(int j=0;j<size;j++){
    cout<<"Enter "<<j<<" Element: ";
    cin>>array[j];
    }

//Your Entered Array Is
    for(int a=0;a<size;a++){
       cout<<"array[ "<<a<<" ]  =  ";
       cout<<array[a]<<endl;
    }
```

Source code

1. **Lexical analysis**
2. Parsing
3. Semantic analysis
4. Optimization
5. Code Generation

Target code

```
FE30-  20 B4 FC 90 F7 60 B1 3C
*FDEDL

FDED-   6C 36 00    JMP   ($0036)
FDF0-   C9 A0       CMP   #$A0
FDF2-   90 02       BCC   $FDF6
FDF4-   25 32       AND   $32
FDF6-   84 35       STY   $35
FDF8-   48          PHA
FDF9-   20 78 FB    JSR   $FB78
FDFC-   68          PLA
FDFD-   A4 35       LDY   $35
FDFF-   60          RTS
FE00-   C6 34       DEC   $34
FE02-   F0 9F       BEQ   $FDA3
FE04-   CA          DEX
FE05-   D0 16       BNE   $FE1D
FE07-   C9 BA       CMP   #$BA
FE09-   D0 BB       BNE   $FDC6
FE0B-   85 31       STA   $31
FE0D-   A5 3E       LDA   $3E
FE0F-   91 40       STA   ($40),Y
FE11-   E6 40       INC   $40
*
```

Lexical analysis

- Lexical analysis: reads the input characters of the source program as taken from preprocessors, and group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- Roles of lexical analyzer
  - Breaks source program into small lexical units, and produces tokens
  - Remove white space and comments
  - If there is any invalid token, it generates an error

Dividing source code

Human format

```
if (i==3)
    X=0;
else
    X=1;
```

Lexical analyzer format

```
\tif (i==3)\n\t\tX=0;\n\telse\n\t\tX=1;
```

- Divide the program into lexical units

```
\t|if| |(|i|==|3|)|\n\t\t|X|=|0|;|\n\t|else|\n\t\t|X|=|1|;|
```

Grouping (classifying) lexemes

- In English
  - Verb , Noun, Adj, Adv.
- In Programming language
  - Keywords, Identifier, operators, assignment, semicolon
- Token = <token name , attribute value>
  - Example of creating class token

```
int a = 3;
```

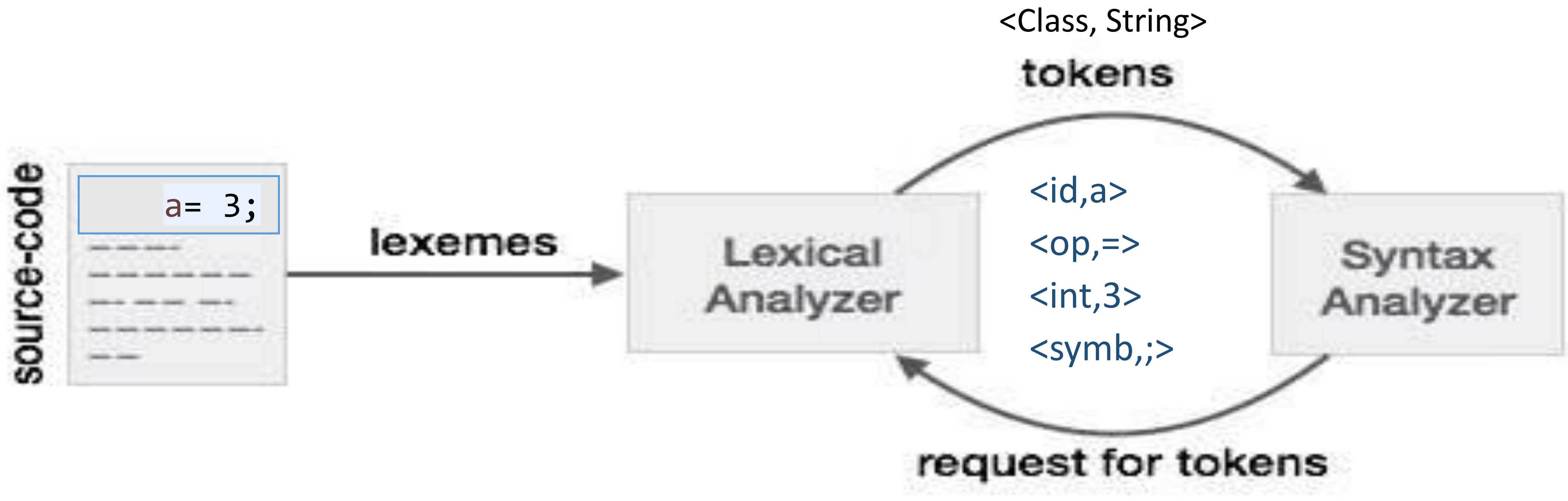
```
<keyword, int>  <identifier, a> <assignment,=>
<constant, 3> <symbol,;>
```

Token class

Token classes

- Token classes correspond to set of strings, such as followings
- *Identifiers :* String of letters or digits start with letters
  - Identifier = (letter)(letter | digit)*
- *Integers* : non-empty digit of strings.
  - integers= $(sign)^{?}(digit)^{+}$
- *Keywords* : fixed set of reserved words
  - Else , if , for , while , do.
- *Whitespace* : blanks, newlines, tabs

Lexical analyzer




```
\tif (i==3)\n\t\tX=0;\n\telse\n\t\tX=1;
```

Regular expressions

- Regular expression (RE) is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.
- one of the RE’s applications is to describe programming language token classes

Regular expression

- letter = [a – z] or [A – Z]
- digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
- sign = [+ | -]
- Decimal = $(\text{sign})^{?}(\text{digit})^{+}$
- Identifier = (letter)(letter | digit)*
- Float = $(\text{sign})^{?}\,(\text{digit})^{+}\,(.\text{digit})^{*}$

Strings And Languages

- Strings:
  - A string is a data type used in programming, such as an integer and floating-point unit, but is used to represent text rather than numbers. It is comprised of a set of characters that can also contain spaces and numbers. For example, the word “hamburger”. Even "12345" could be considered a string, if specified correctly.
  - Two important examples of programming language alphabets are ASCII character sets.

Strings (contd…):

- A string is a finite sequence of symbols such as 001. The length of a string x, usually denoted $|x|$, is the total number of symbols in x.
- For eg.: 110001 is a string of length 6. A special string is a empty string which is denoted by ε. This string is of length 0(zero).

Epsilon

- If x and y are strings, then the concatenation of x and y, written as $x.y$ or just xy, is the string formed by following the symbols of x by the symbols of y.
- For eg.:*abd.ce = abdce* i.e. if $x= abd$ & $y=ce$, then

xy = abdce.

Strings (contd…)

it talks about +
A+ regex

- The concatenation of the empty string with any string is that string i.e. $\varepsilon x = x\varepsilon = x$.
- Concatenation is not any sort of product, thus it is an iterated product in form of exponential.
- E.g.: $x^1 = x$, $x^2 = xx$, $x^3 = xxx$
  - In general x^i is the string *x* repeated *i* times. We take x^0 to be ε for any string x. Thus, ε plays the role of 1, the multiplicative identity.

- (x^i) means you repeat the string (x) exactly (i) times. For instance, if x is " abc" and i is 3, then (x^3) is " abcabcabc."

- When (i) is 0, we represent it as (x^0). This means you don't repeat the string at all, and it's like an empty or null string, represented as "."

- So, "" (epsilon) is like the number 1 in multiplication. Just as 1 is the identity for multiplication (anything times 1 is itself), is the identity for string repetition. It doesn't change the string when you repeat it.

Strings (contd...)

- If *x* is some string, then any string formed by discarding zero or more trailing symbols of *x* is called a *prefix* of x.
- For e.g.: abc is a *prefix* of abcde.
- A *suffix* of *x* is a string formed by deleting zero or more of the leading symbols of *x*. cde is a *suffix* of abcde.
- A substring of x is any string obtained by deleting a *prefix* and a *suffix* from *x*.
- For any string *x*, both *x* and ε are *prefixes*, *suffixes* and *substrings* of *x.*
- Any *prefix* or *suffix* of *x* is a substring of *x*, but a substring need not be a prefix or suffix.
- For e.g..: cd is a substring of abcde but not a prefix or suffix .

Languages

- The term language means any set of strings formed from some specific alphabets.
- Simple set such as Φ, the empty set {ε} having no members or the set containing only the empty string, are languages.
- The notation of concatenation can also be applied to languages.
- For e.g.: If *L* and *M* are languages, then *L*.*M*, or just *LM*
- *LM* is language consisting of all strings *xy* which can be formed by selecting a string *x* from *L*, a string *y* from *M*, and concatenating them in that order.

LM= {*xy*| *x* is in *L* and *y* is in *M*}

means where

Languages (contd…)

The right one

- E.g.: If L={0, 01, 110} and M= {10, 110}. Then LM={010, 0110, 01110, 11010, 110110} {010, 0110, 0110, 01110, 11010, 110110}
- 11010 can be written as the concatenation of 110 from L and 10 from M.
- 0110 can be written as either 0.110 or 01.10 i.e. it is a string from L followed by one from M.
- In analogy with strings, we use L^i to stand for $LL....L$ *(i times)*. It is logical to define L^0 to be {ε}, since {ε} is the identity under concatenation of languages. i.e. $\{\varepsilon\}L = L\{\varepsilon\} = L$
- The union of languages L & M is given by

  $L \cup M = \{x \mid x \text{ is in } L \text{ or } x \text{ is in } M\}$

Languages (contd...)

- If concatenation is analogous to multiplication. Then ø, the empty set is the identity under union (analogous to zero)

$$\emptyset \cup L = L \cup \emptyset = L$$

$$\&$$

$$\emptyset L = L \emptyset = \emptyset$$

- Any string in the concatenation of ø with L must be formed from x in ø and y in L.

- There is another operation in specifying tokens, that is *closure* or "any number of" operator. We use L^* to denote the concatenation of language L with itself any number of times.

- $$L^* = \bigcup_{i=0}^{\infty} L^i$$

- Consider D be the language consisting of the strings 0,1,……9 i.e. each string is a single decimal digit. Then D^* is all strings of digits including empty string.

- If $L = \{aa\}$, the L^* is all strings of an even number of *a's.*

- $L^O = \{\varepsilon\}$, $L^i = \{aa\}$, the L^* is all strings of an even number of a's $L^O = \{\varepsilon\}$, $L^i = \{aa\}$, $L^2 = \{aaaa\}$ & so on.
- If ε is excluded $L.(L^*)$ is denoted by,
- The Unary postfix operator + is called *positive closure* and denotes "one or more instances of".

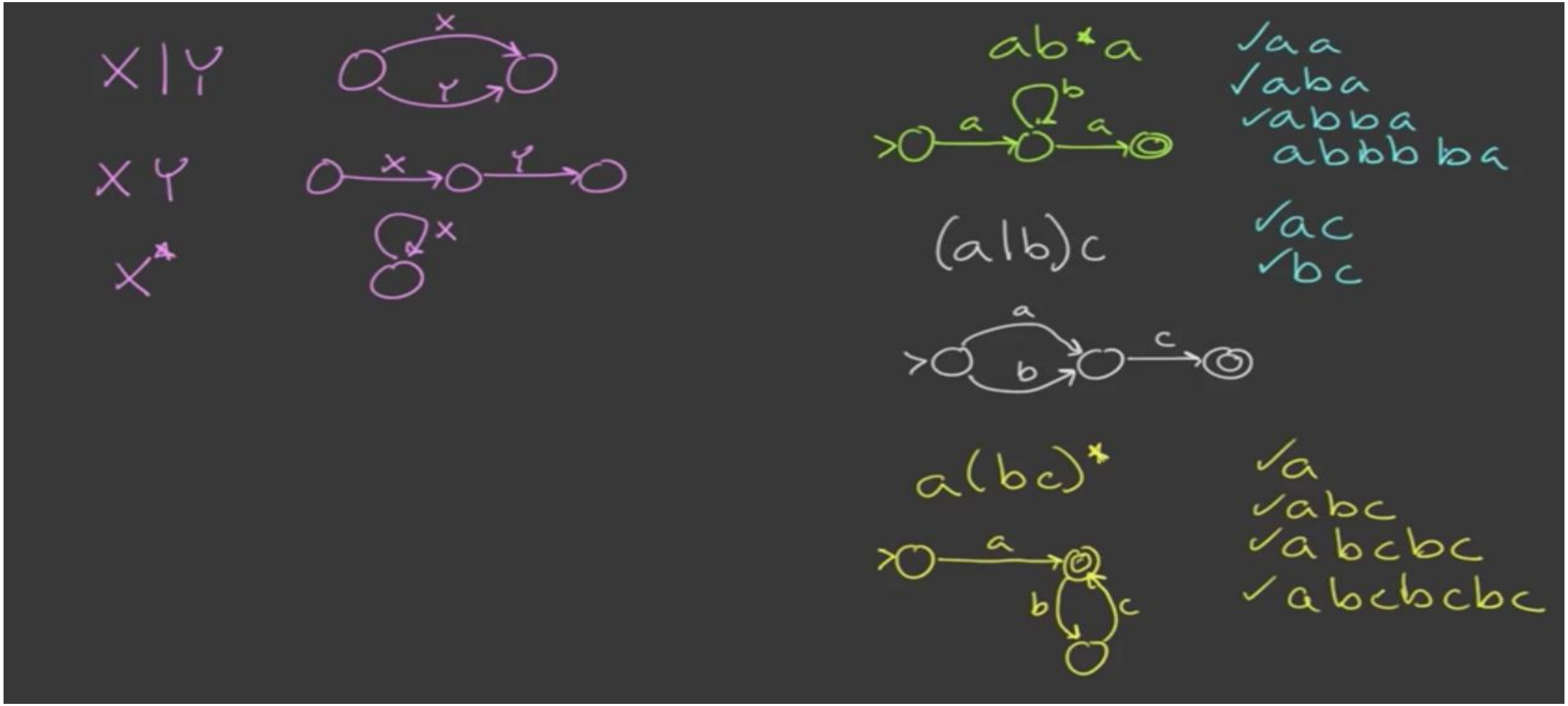
Finite Automata

- A finite automaton (FA) is a simple idealized machine used to recognize patterns within input taken from some character set (or alphabet) C. The job of an FA is to *accept* or *reject* an input depending on whether the pattern defined by the FA occurs in the input.
- A finite automaton consists of:
- a finite set S of N states
- a special start state
- a set of final (or accepting) states
- a set of transitions T from one state to another, labeled with chars in C

Finite Automata Cont’d

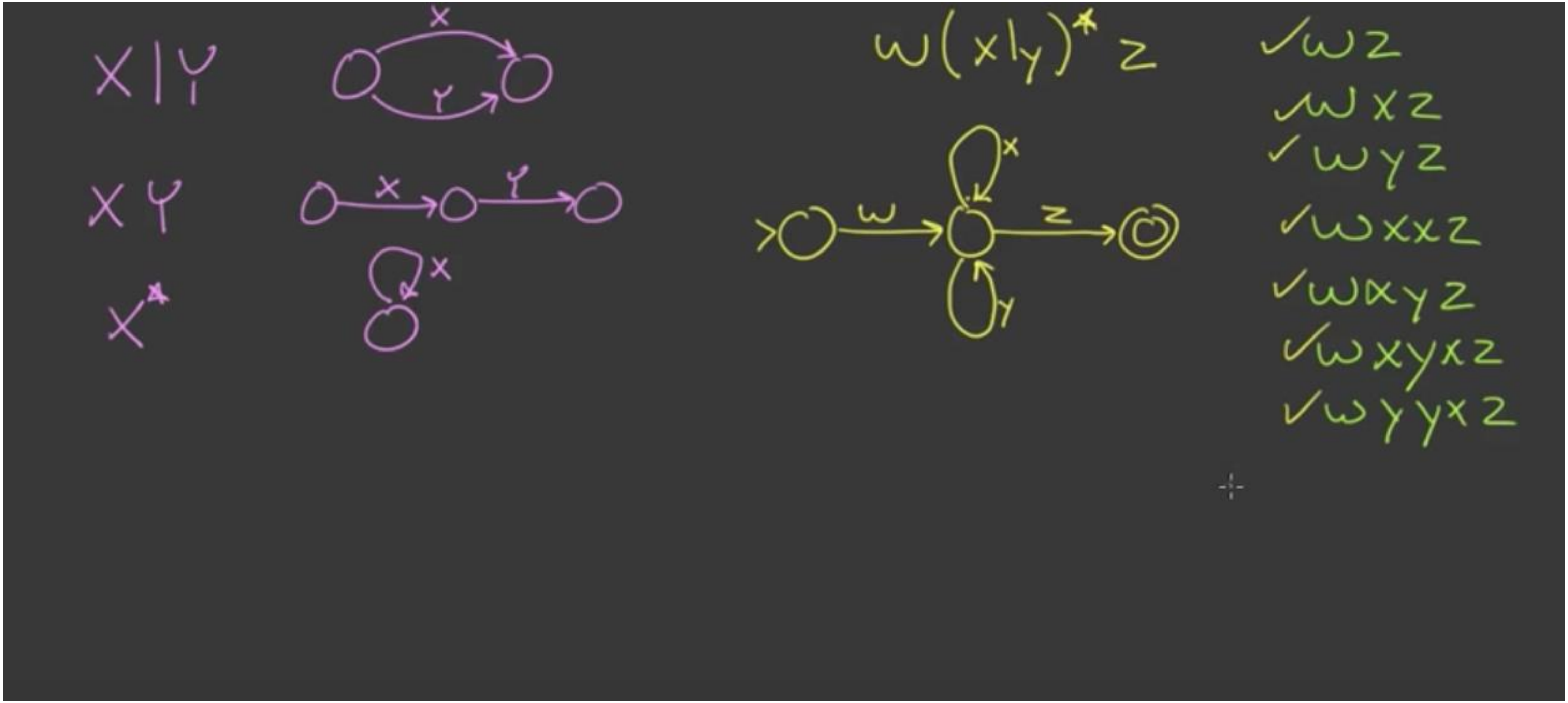
- we can represent a FA graphically, with nodes for states, and arcs for transitions.
- We execute our FA on an input sequence as follows:
- Begin in the start state
- If the next input char matches the label on a transition from the current state to a new state, go to that new state
- Continue making transitions on each input char
- If no move is possible, then stop
- If in accepting state, then accept

RE to FA

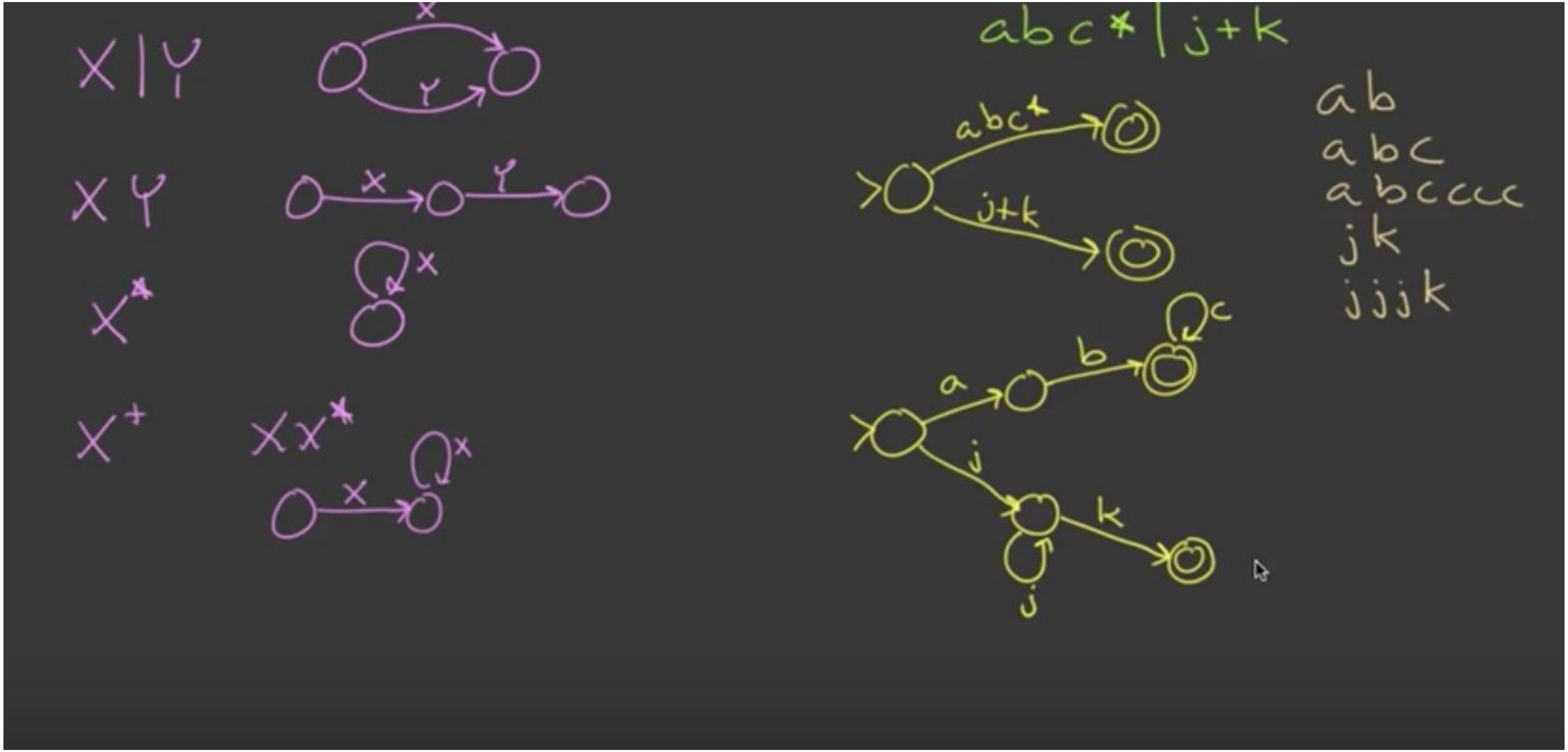


X|Y
X
Y
XY
X
Y
X*
X
ab*a
b
a
a
✓aa
✓aba
✓abba
abbbba
(a|b)c
✓ac
✓bc
a
b
c
a(bc)*
a
b
c
✓a
✓abc
✓abcbc
✓abcbcbc

RE to FA



RE to FA



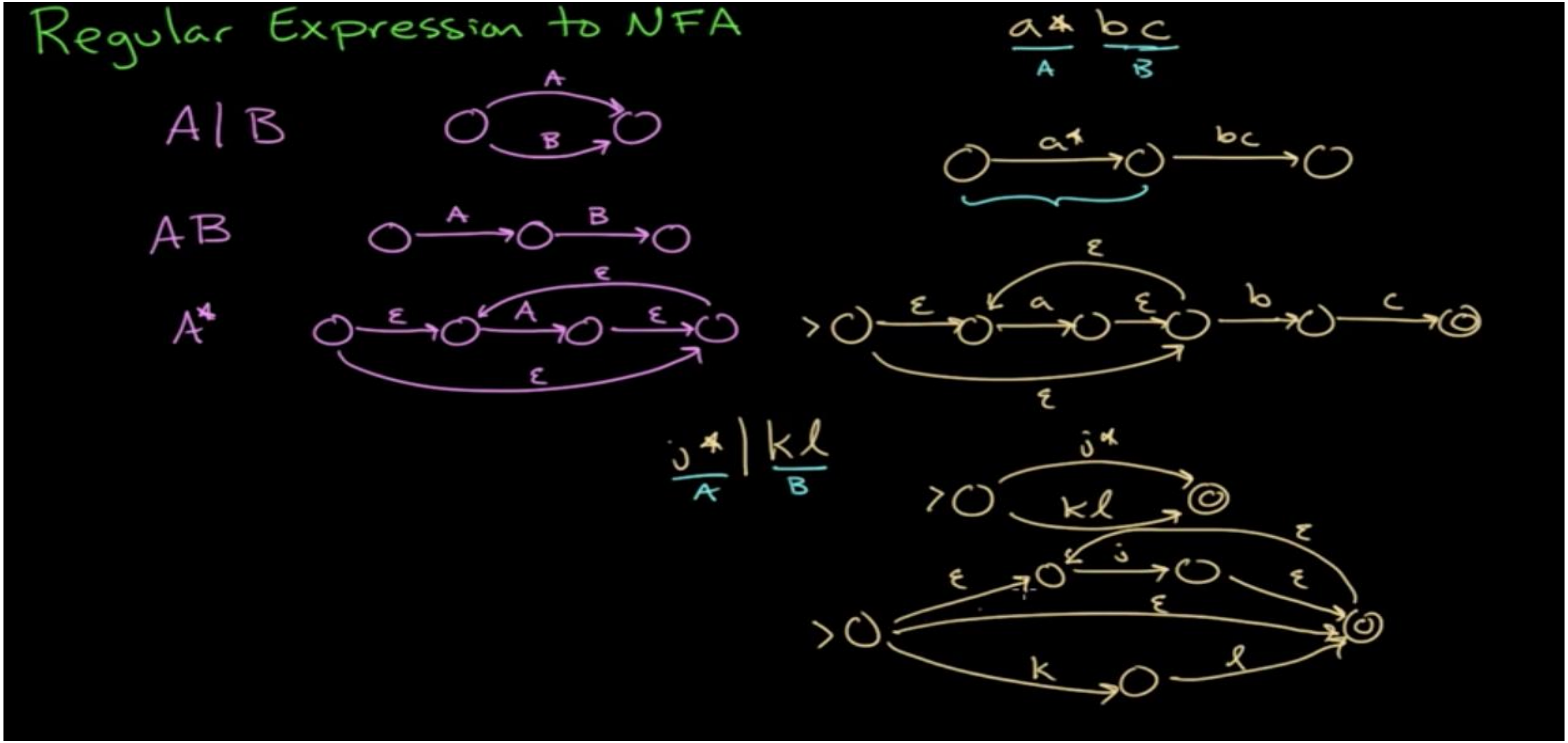
NFA and DFA

In automata theory, a finite state machine is called a deterministic finite automaton (DFA), if

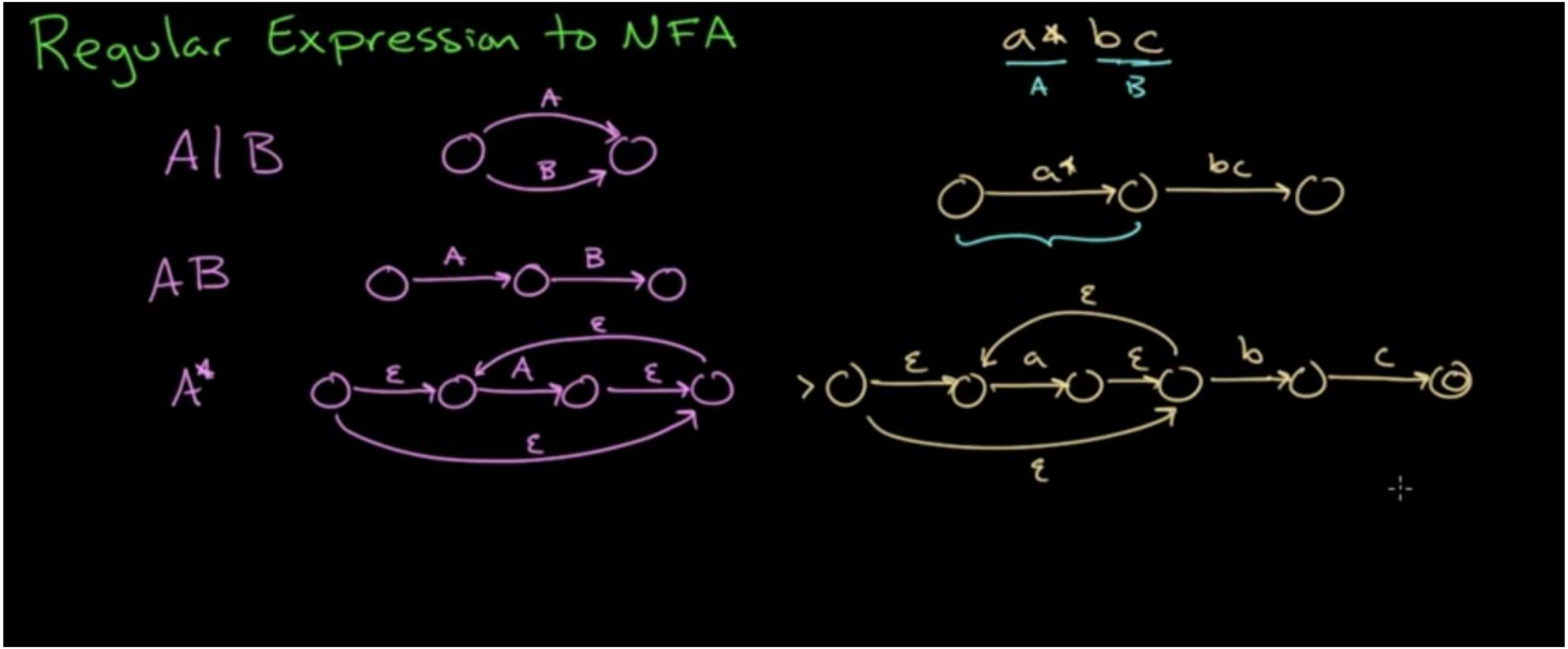
- each of its transitions is uniquely determined by its source state and input symbol, and
- reading an input symbol is required for each state transition.

A **nondeterministic finite automaton** (**NFA**), or nondeterministic finite state machine, does not need to obey these restrictions. In particular, every DFA is also an NFA.

RE to NFA

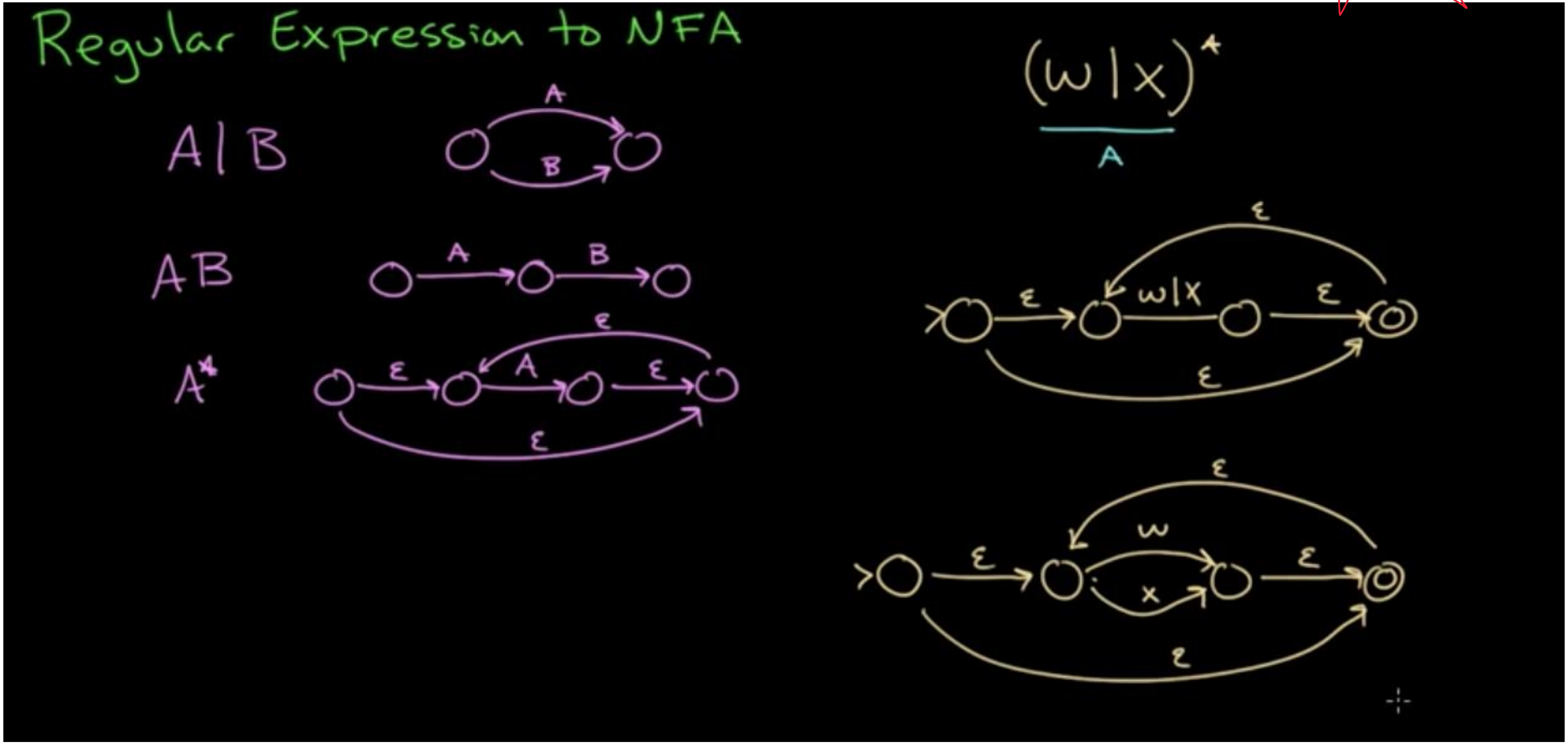

Regular Expression to NFA
A|B
A
B
AB
A
B
A*
ε
ε
A
ε
ε
a* bc
A B
a*
bc
ε
ε
a
ε
b
c
ε
j* | kl
A B
j*
kl
ε
j
ε
ε
ε
k
l

RE to NFA

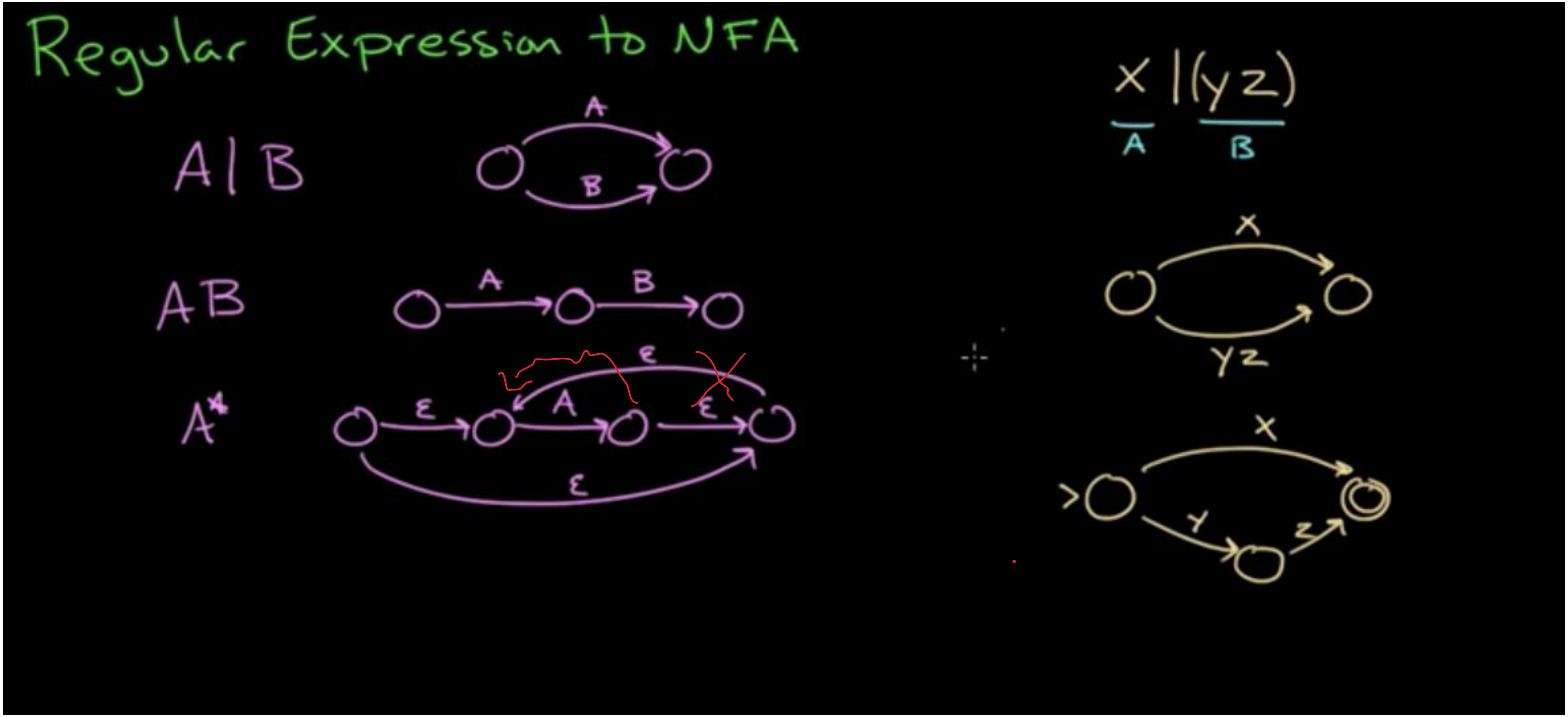


Regular Expression to NFA
A|B
A
B
AB
A
B
A*
ε
A
ε
ε
ε
a* bc
A
B
a*
bc
ε
ε
a
ε
b
c
ε

RE to NFA



RE to NFA

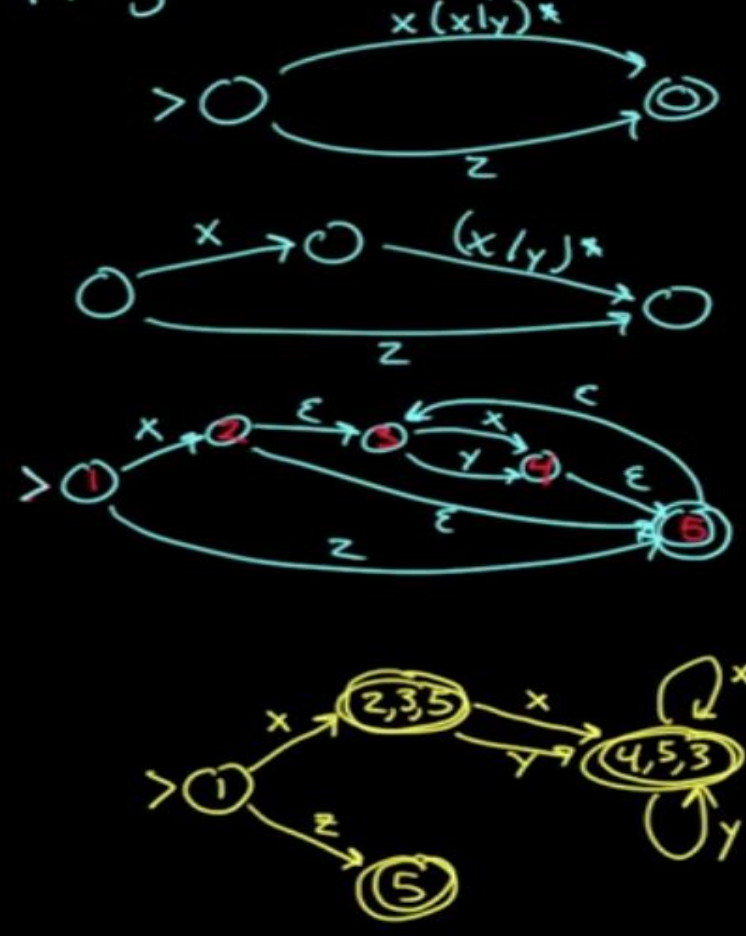


RE to DFA

Regular Expression to DFA

$x(x|y)^*|z$

- ✓ z
- ✓ x
- ✓ x x
- ✓ x y x
- ✓ x y y

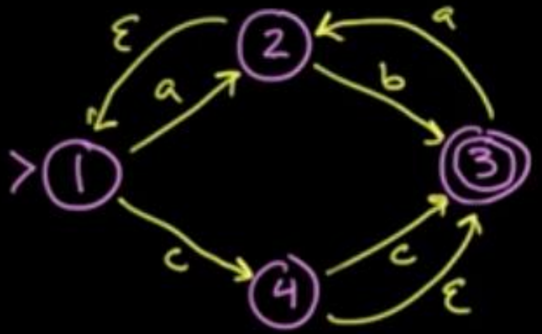


	x	y	z	ε*
>1	2	—	5	1
2	—	—	—	2,3,5
3	4	4	—	3
4	—	—	—	4,5,3
5	—	—	—	5

	xε*	yε*	zε*
> 1	2,3,5	—	5
2,3,5	4,5,3	4,5,3	—
5	—	—	—
4,5,3	4,5,3	4,5,3	—

NFA to DFA

NFA to DFA



	a	b	c	ε*
>1	2	—	4	1
2	—	3	—	2,1
3	2	—	—	3
4	—	—	3	4,3

yes	no
ab	b
abab	a
cc	cb
c	caa
ccab	
ccacc	
ccac	
abacab	

	aε*	bε*	cε*
> 1	2,1	—	4,3
2,1	2,1	3	4,3
4,3	2,1	—	3
3	2,1	—	—