ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 3b

Lexical Analysis

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Recognition of Tokens

`if` expressions and relational operators

```
   if → if
 then → then
 else → else
relop → < | <= | = | <> | > | >=
   id → letter(letter|digit)*
  num → digit+(.digit+)?(E(+|-)?digit+)?
```

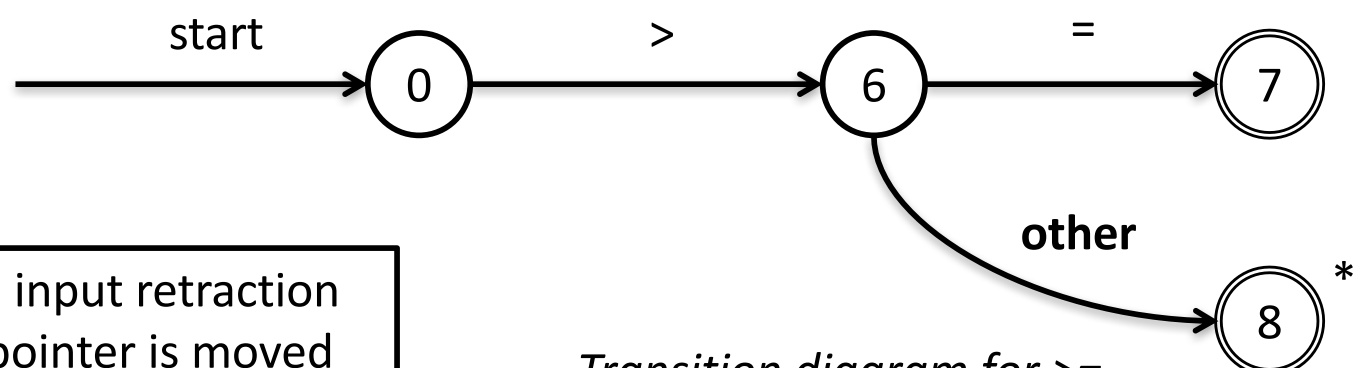
Trim whitespace

```
delim → blank | tab | newline
   ws → delim+
```

Transition Diagram

Διάγραμμα Μετάβασης

- Intermediate visual representation
- The graph depicts how the pointer moves from character to character
- Circles are called *states*
  - They represent the pointer's positions
- *Edges* leaving state *s* have labels indicating the characters required for moving to the next state
  - **Other** is special (refers to any character that is not indicated by any of the other edges leaving *s*)

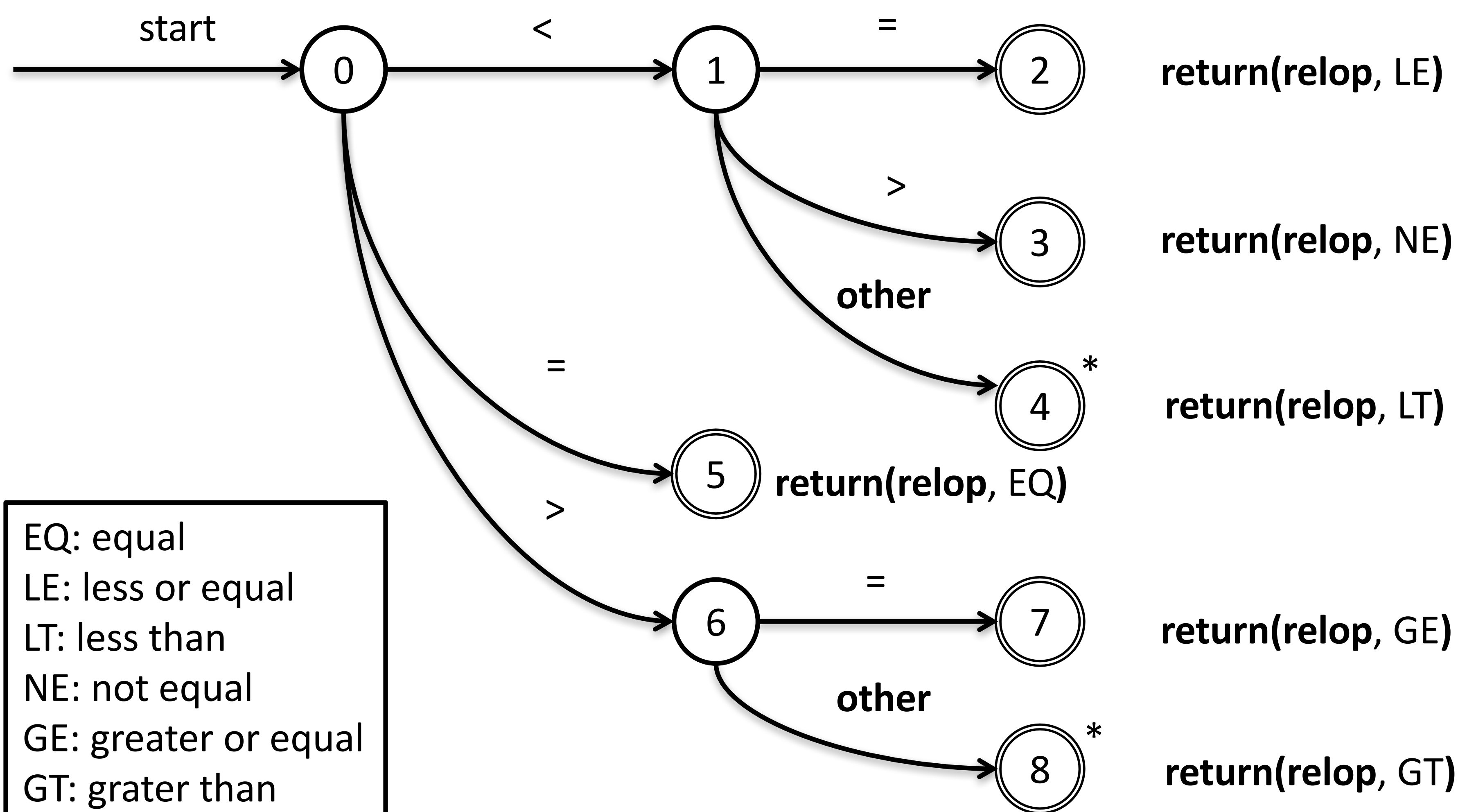


Transition diagram for >=

* denotes states on which input retraction must take place (i.e., the pointer is moved to another transition diagram).

Transition Diagram

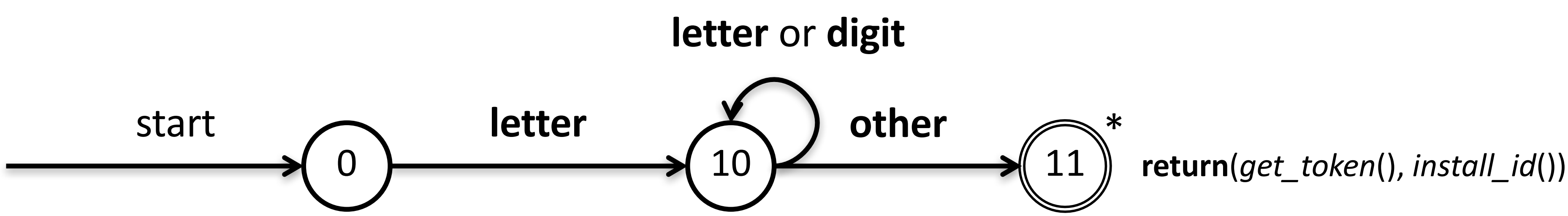
relation operators



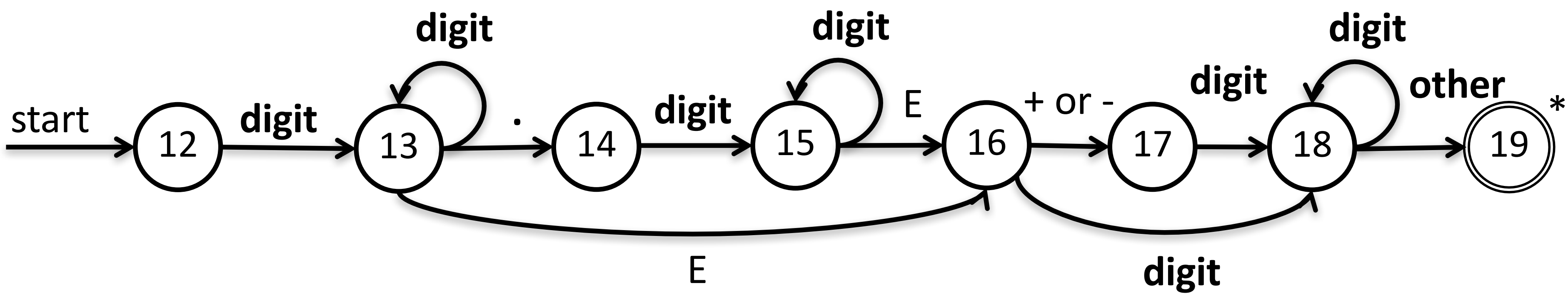
EQ: equal
LE: less or equal
LT: less than
NE: not equal
GE: greater or equal
GT: grater than

Keywords and Identifiers

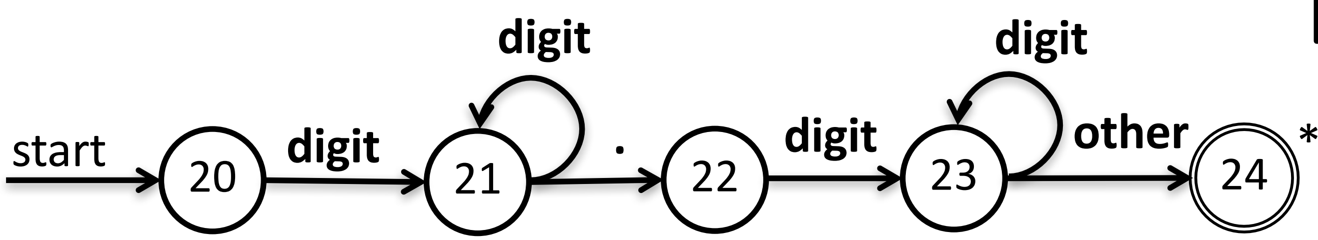
- Keywords is a special case of identifiers
- Once an identifier is recognized we can check if it is a keyword



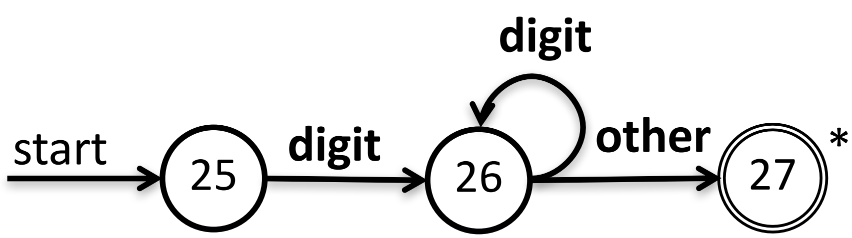
Unsigned numbers



Recognizes 12.3E4
(**digits fraction**? **exponent**?)



Recognizes 12.3
(**digits fraction**)



Recognizes 12
(**digits**)

Finite Automata

Πεπερασμένα Αυτόματα

- Recognizer for a language
  - A program that takes as input a string *x* and answers "yes" if *x* is a sentence of the language and "no" otherwise.
- Compile regular expressions to recognizers
  - Construct a **generalized transition diagram** called a *finite automaton*
- Two classes of finite automata
  - Deterministic, DFA (*ντετερμινιστικό*)
  - Non-deterministic, NFA (*μη-ντετερμινιστικό*)

DFAs and NFAs

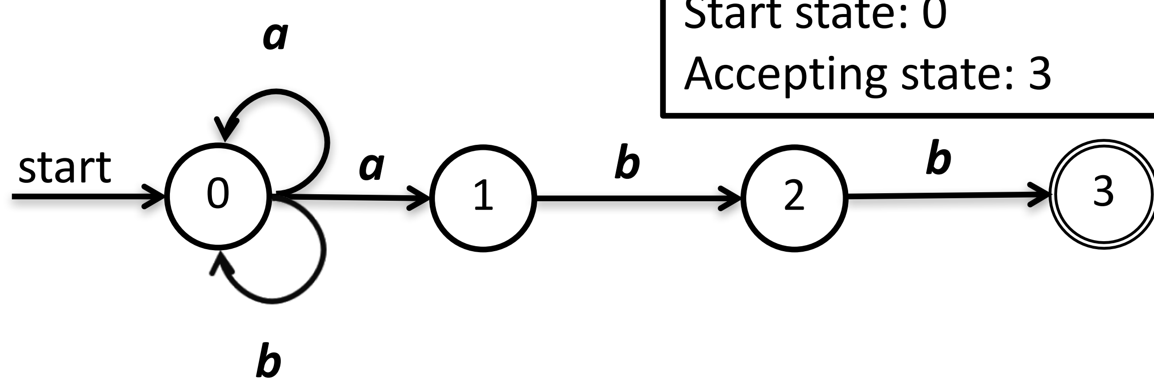
- Both a DFA and an NFA are capable of recognizing precisely the regular sets
- Time-space trade-off
  - DFAs implement faster recognizers
  - DFAs are bigger (more states, more memory)
- Regular expressions can be compiled in both a DFA and an NFA

NFA

- Mathematical model that consists of
  1. a set of states ***S***
  2. a set of input symbols ***Σ*** (the *input symbol alphabet)*
  3. a transition functions ***move*** that maps state-symbol pairs to sets of states
  4. a state $\boldsymbol{s_0}$ that is distinguished as the *start* (or initial) *state*
  5. a set of states ***F*** distinguished as *accepting* (or final) *states*

NFA for *(a|b)*abb*

States: {0, 1, 2, 3}
Symbol alphabet: {*a*, *b*}
Start state: 0
Accepting state: 3



An NFA looks like a **transition diagram**, but the **same character** can label **two or more transitions** out of **one state**:

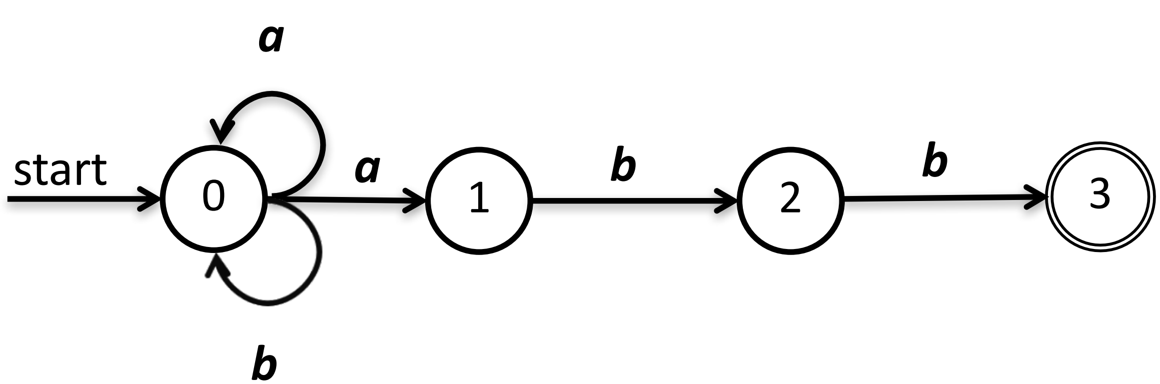
Example: *a* can transit control:
from **State 0** to **State 0**
from **State 0** to **State 1**

Also: edges can be label by the special symbol **ε**

Implementation using a Transition Table

STATE	INPUT SYMBOL	
	a	*b*
0	{0, 1}	{0}
1	-	{2}
2	-	{3}

If I am in state 0 and the input character is *a*, then I can move to states 0 or 1
If I am in state 0 and the input character is *b,* then I can move to state 0
If I am in state 1 and the input character is *a*, then there is no state to move
If I am in state 1 and the input character is b, then I can move to state 2



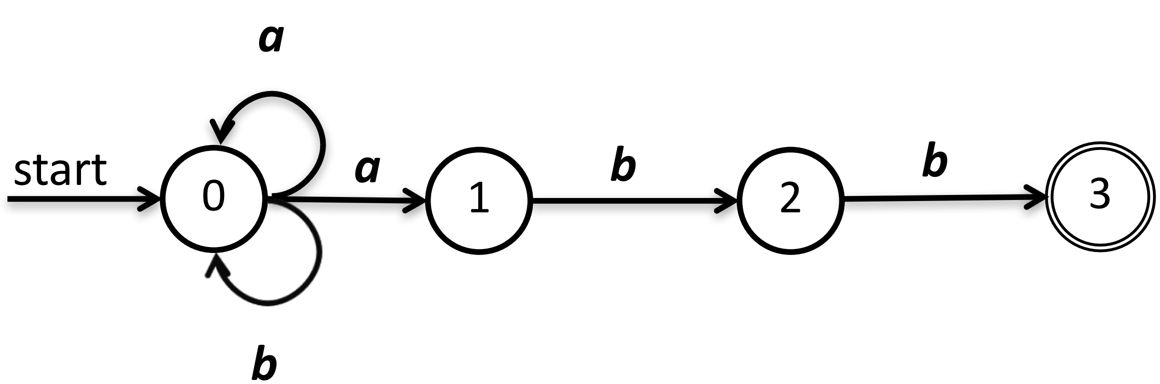
Accepted input strings
*(a|b)*abb*

Accepted input strings: *abb, aabb, babb, aaabb, …*

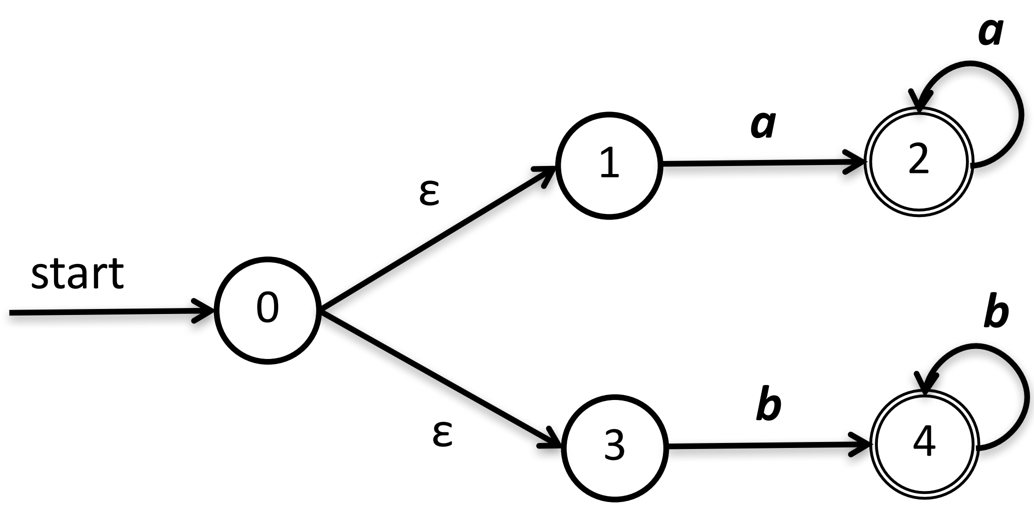
$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Several other sequences of moves may be made on the input string *aabb,* but none of the others happened to end in an accepting state:

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

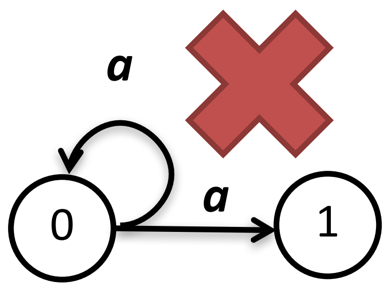


NFA for *aa**/*bb**



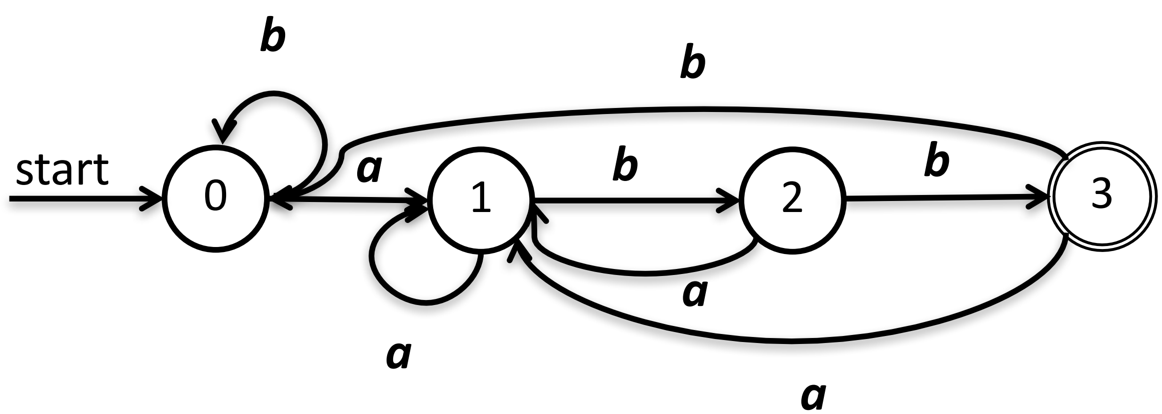
DFA

1. no state has an ε-transition, i.e., a transition on input ε,
2. For each state *s* and input symbol *a*, there is **at most** one edge labeled *a* leaving *s*

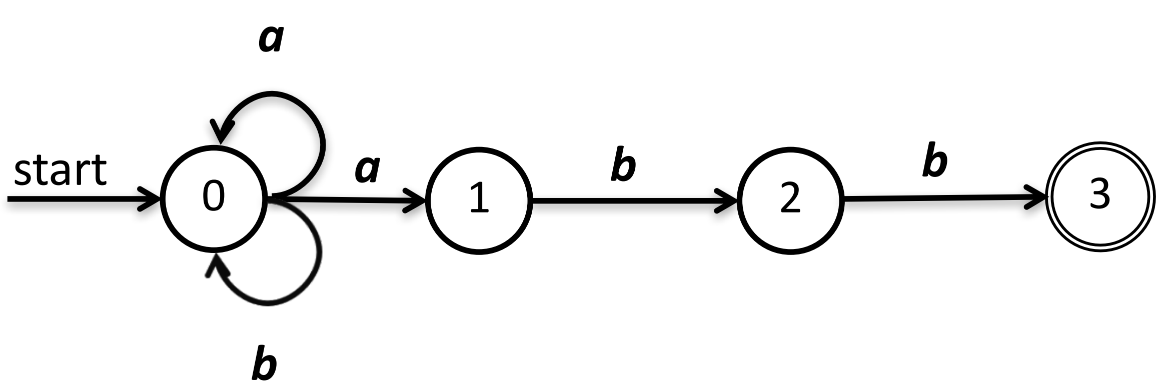


You can't have *a* leaving state 0 and being able to reach two states, i.e., state 0 and state 1

DFA for *(a|b)*abb*



Recall the NFA version:



DFA is easy to code

```
s := s0
c := nextchar
while c != eof do
   s := move(s, c)
   c := nextchar
end
if s in F then
   return “yes”
else
   return “no”
```

What do we do?

- NFAs are easy to conceive and draw
  - Multiple edges on the same characters leaving one state can cause **ambiguity** (**αμφισημιά**)
  - Many paths that spell out the same input string
  - Hard to code
- DFAs are easy to implement in a computer program

Subset Construction

CONVERSION OF AN NFA INTO A DFA

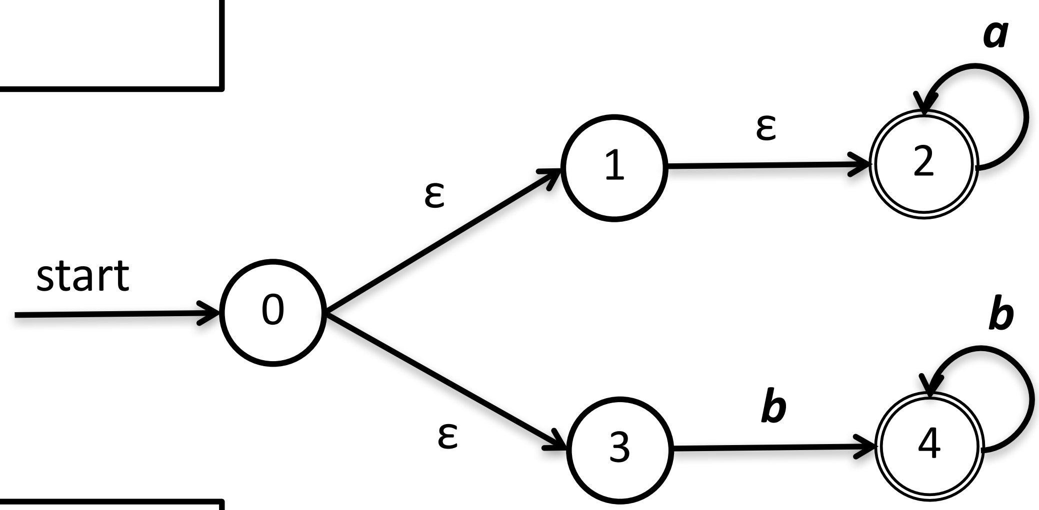
Operations

OPERATION	DESCRIPTION
ε-closure(s)	Set of NFA states reachable from NFA state *s* on ε-transitions alone.
ε-closure(T)	Set of NFA states reachable from some NFA state *s* in *T* on ε-transitions 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.*

Notation: *s* an NFA state, *T* a set of NFA states

Examples

$move(\{1, 2\}, a) = 2$



$\varepsilon\text{-}closure(0) = \{0, 1, 2, 3\}$

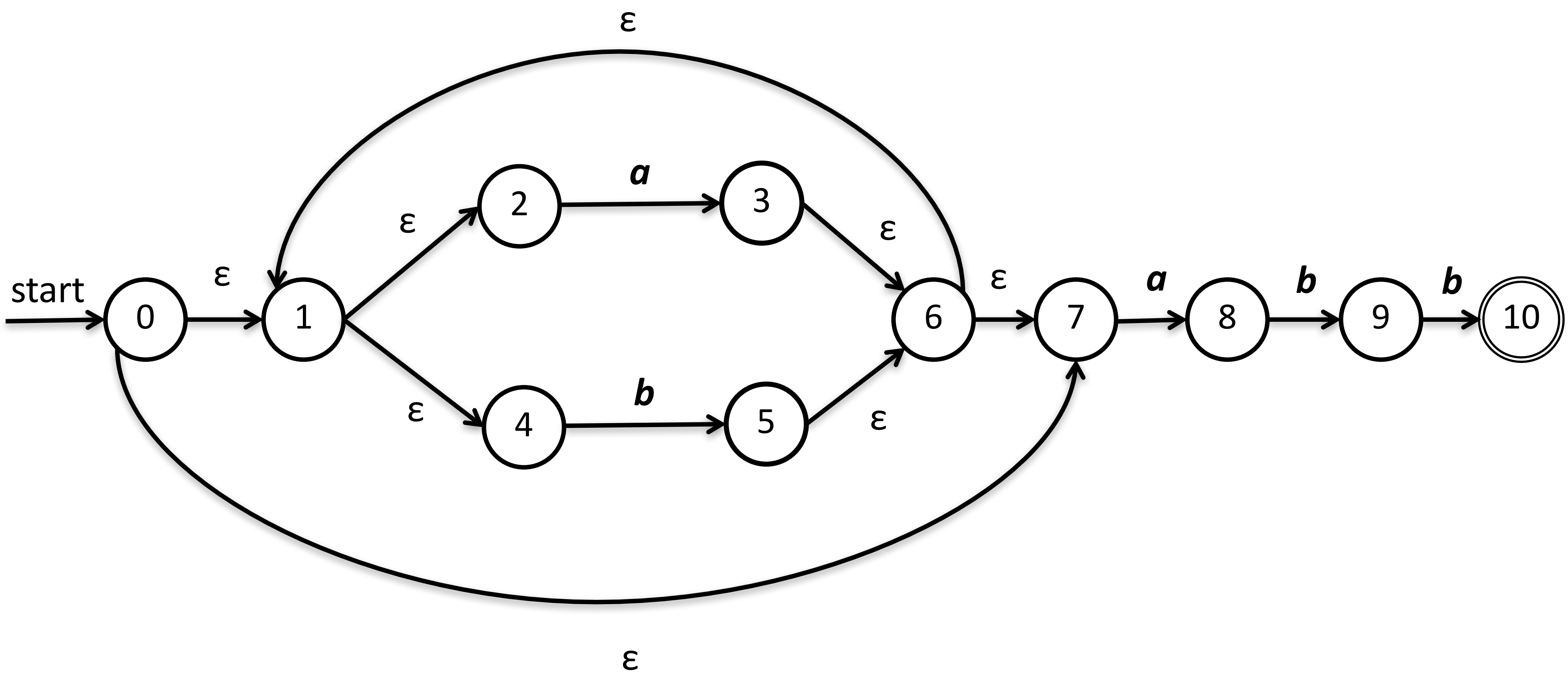
$\varepsilon\text{-}closure(1) = \{1, 2\}$

$\varepsilon\text{-}closure(2) = \{2\}$

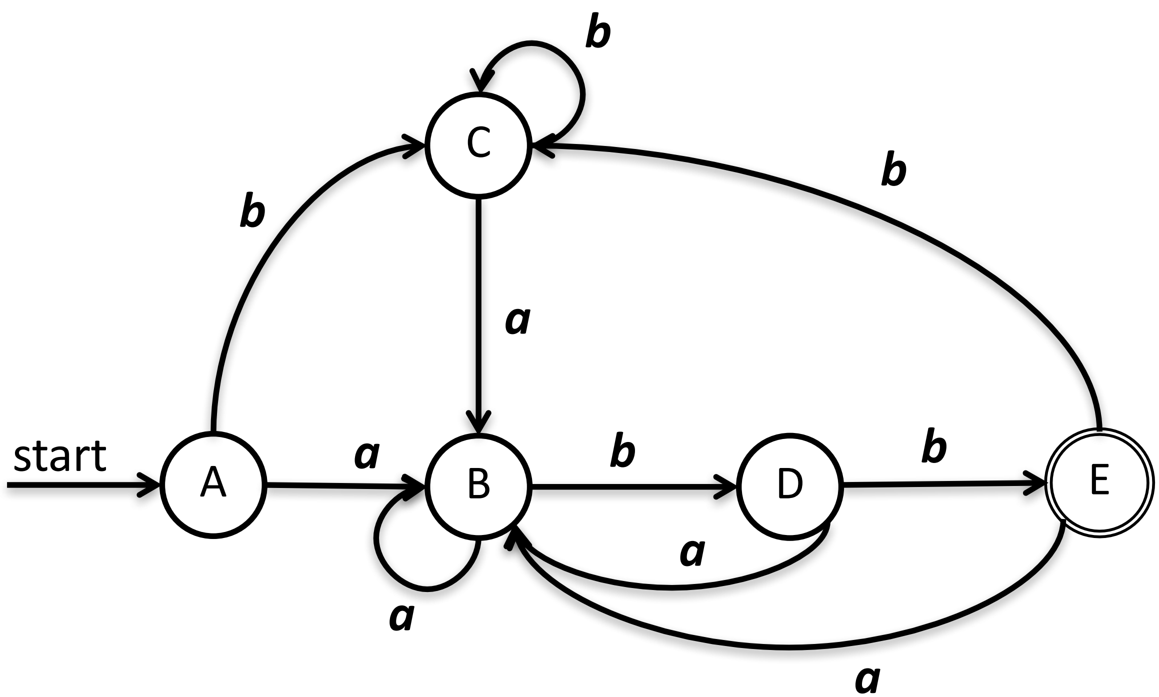
$\varepsilon\text{-}closure(3) = \{3\}$

$\varepsilon\text{-}closure(4) = \{4\}$

Example
Initial NFA, for *(a|b)*abb*



Equivalent DFA



No ε transitions

No two edges with the same symbol leaving one state

Easy to transform to a computer program

Step 1

- The *start state* of the equivalent DFA is ε-*closure*(0)
  - A = {0, 1, 2, 4, 7}, these are exactly the states reachable from state 0 via a path in which every edge is labeled ε

Step 2

- The input symbol is {*a, b*}, we mark A, and compute ε-*closure*(*move*(A, *a*))
  - *move*(A, *a*) is the set of states of the NFA having transitions on *a* from members of A, that is states 2 and 7 (moving to 3 and 8)
  - ε-*closure*(*move*({0, 1, 2, 4, 7}, *a*)) = ε-*closure*({3, 8}) = {1, 2, 3, 4, 6, 7, 8}
  - This is B = {1, 2, 3, 4, 6, 7, 8}

Step 3

- Among the states in A, only 4 has a transition on *b* to 5
  - the DFA has a transition from A to C, and C = ε-*closure*({5}) = {1, 2, 4, 5, 6, 7}

Step 4

- We mark the new sets B and C, and we repeat Step 1-3

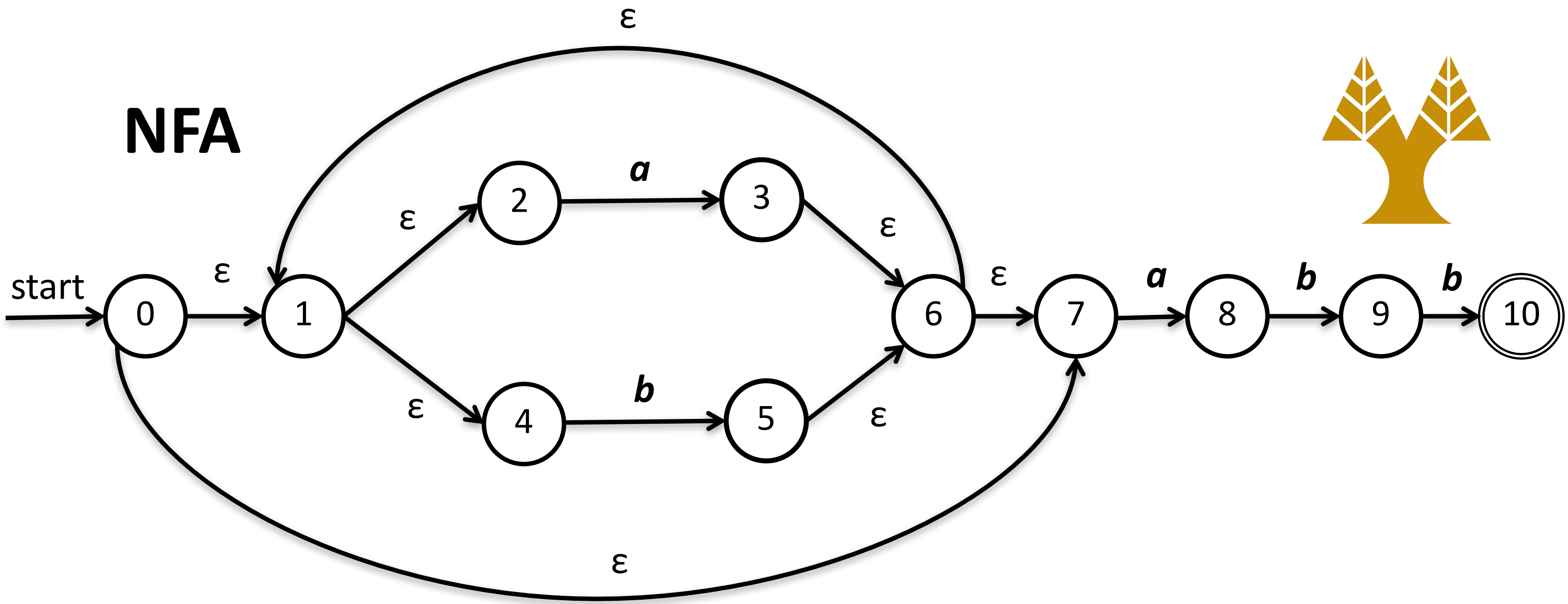
Repeat steps

- Until all sets of the DFA are marked
- Final sets
  - A = {0, 1, 2, 4, 7}
  - B = {1, 2, 3, 4, 6, 7, 8}
  - C = {1, 2, 4, 5, 6, 7}
  - D = {1, 2, 4, 5, 6, 7, 9}
  - E = {1, 2, 3, 5, 6, 7, 10}

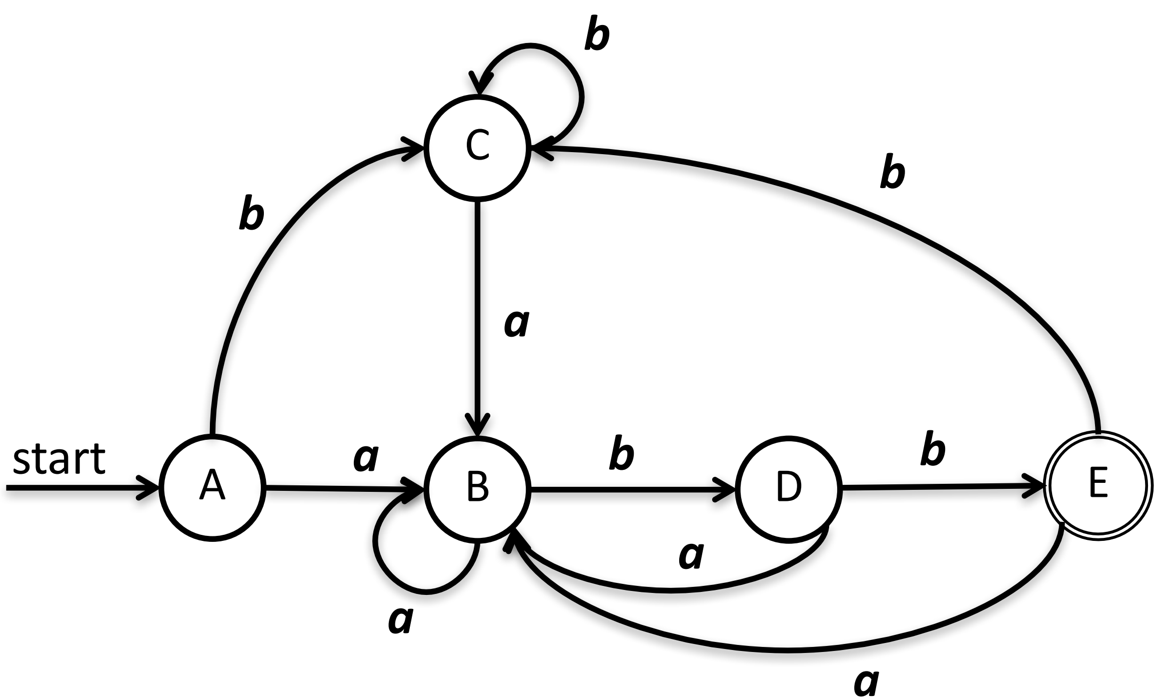
Transition Table for DFA

STATE	INPUT SYMBOL	
	a	*b*
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

NFA



DFA



The subset construction

```
initially, ε-closure(s0) is the only
state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
   mark T
   for each input symbol a do begin
    U = ε-closure(move(T,a))
    if U is not in Dstates then
        add U as an unmarked state to Dstates;
    Dtran(T,a) := U
   end for
end while
```

ε-closure(T)

```
push all states in T onto stack
initialize  ε-closure(T) to T;
while stack is not empty do begin
   pop t
   for each state u with an edge from t to u labeled ε do
    if u not in ε-closure(T)
        add u to ε-closure(T)
        push u
    end if
   end for
end while
```