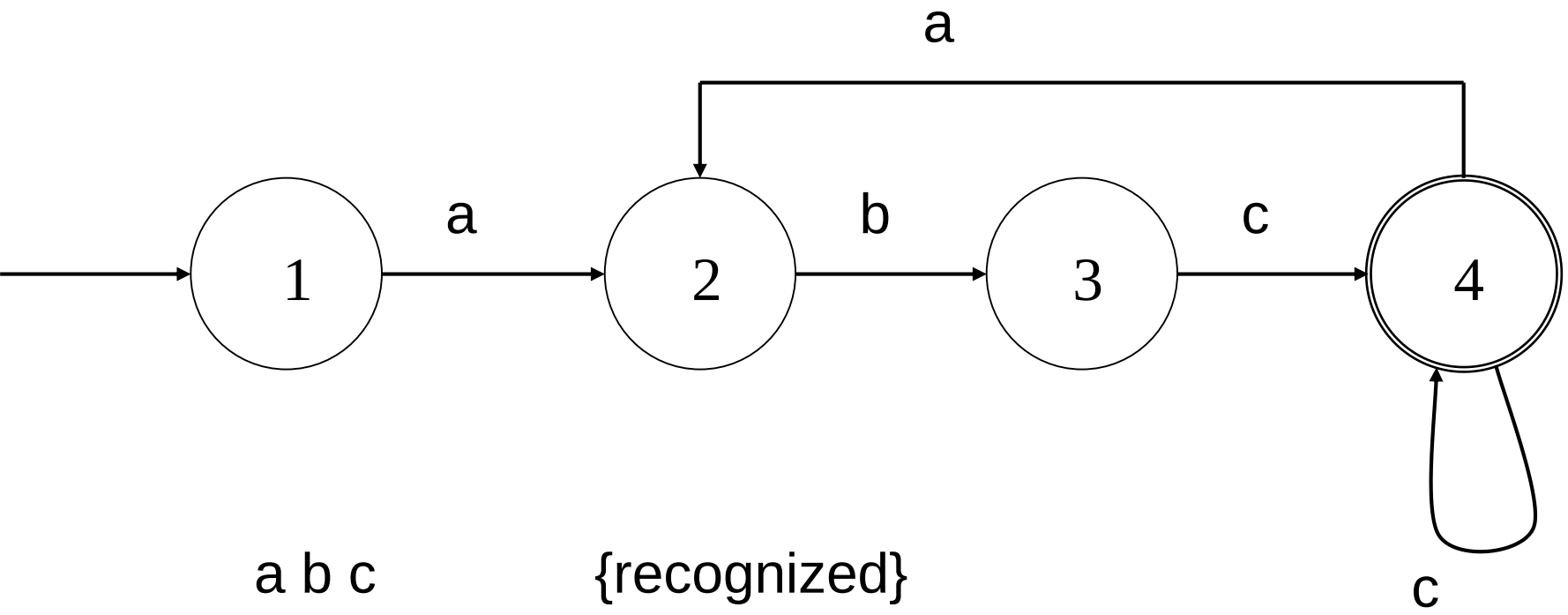
Scanner

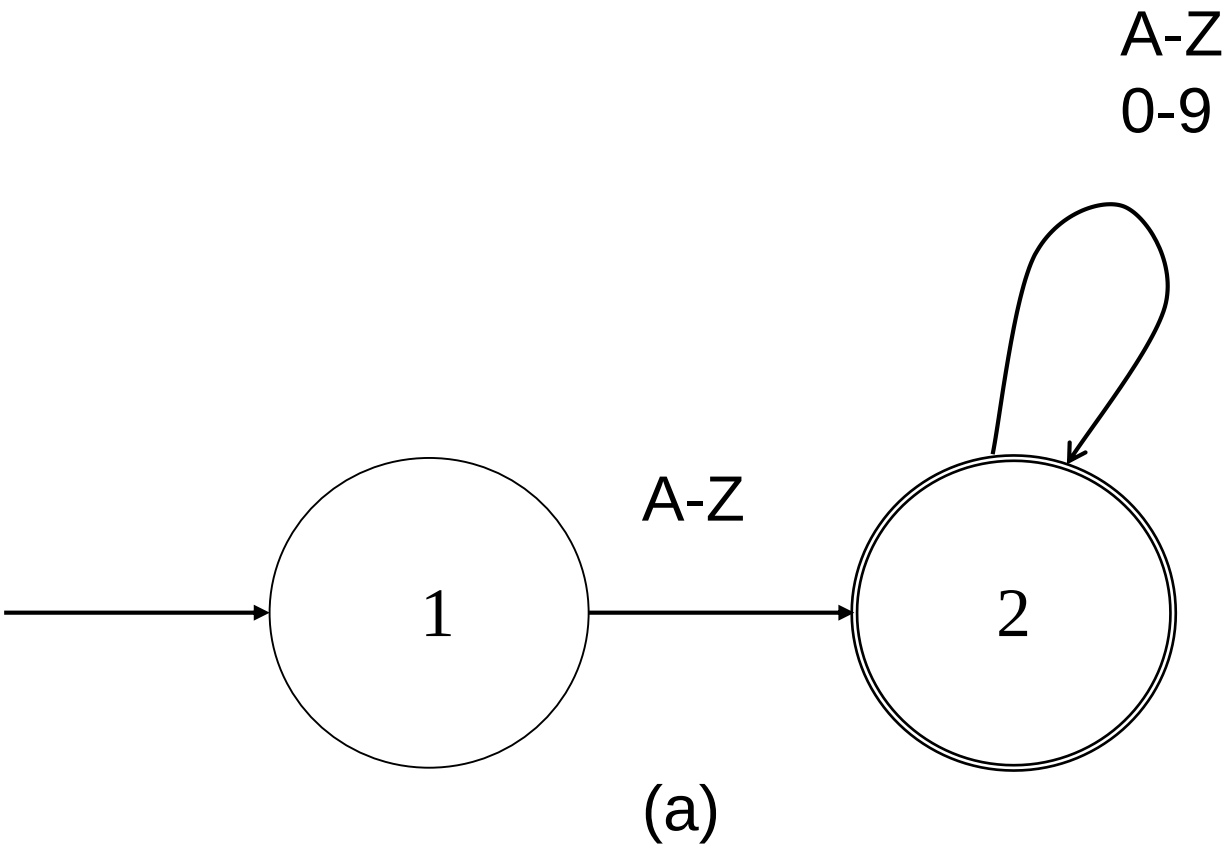
<<Scanner1.ppt>>

Modeling Scanners as Finite Automata

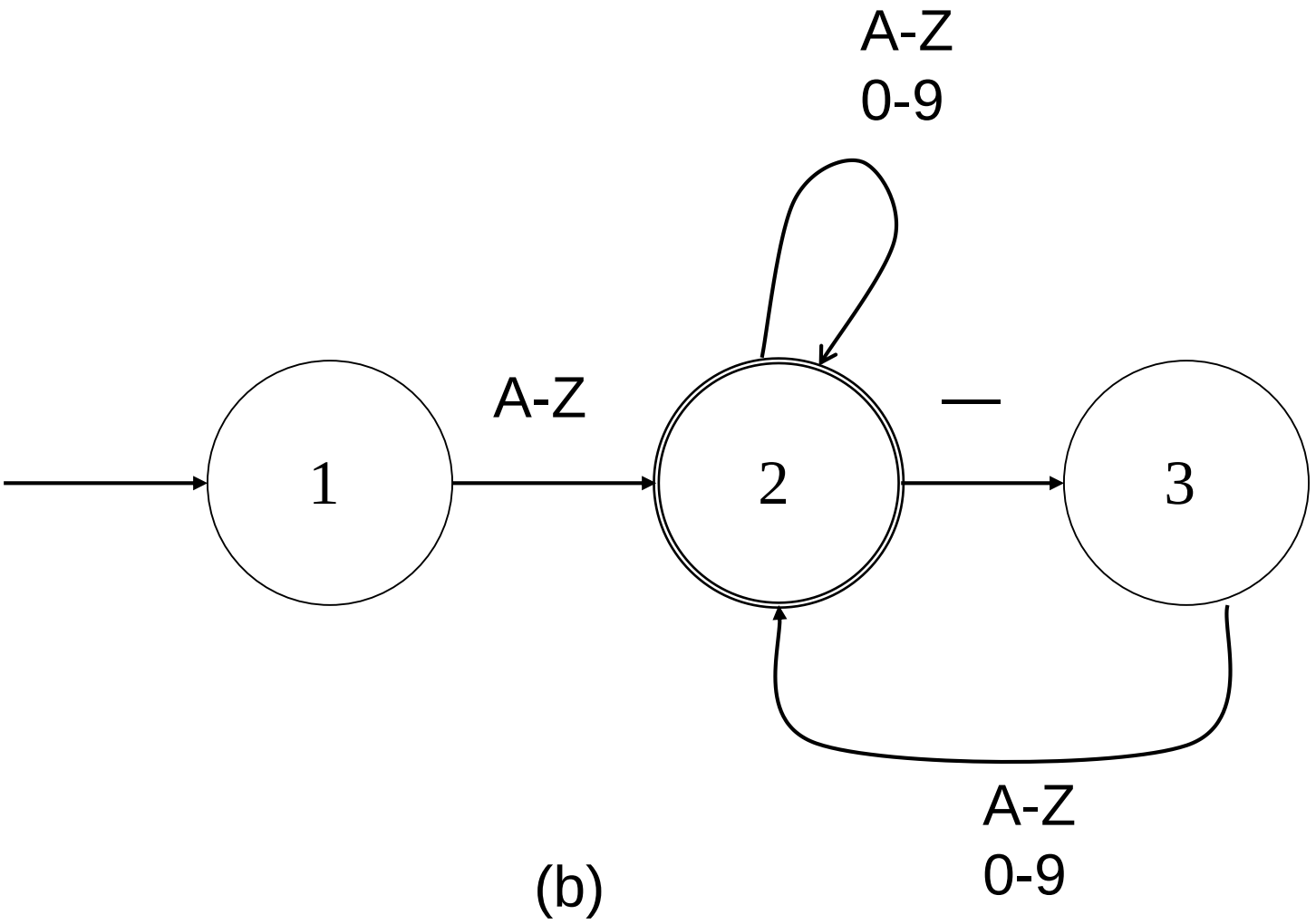


a b c {recognized}
a b c c a b c {recognized}
a c { not recognized}

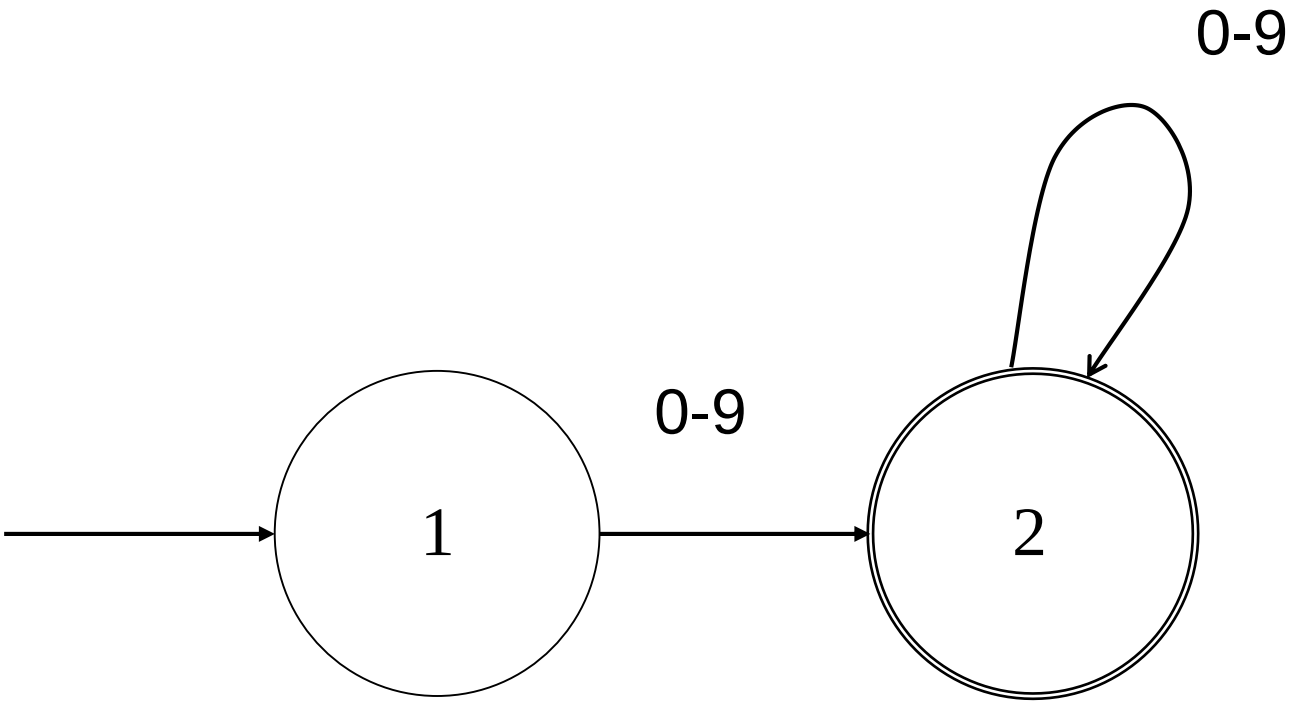
Graphical representation of a finite automaton



Finite automata for typical programming language tokens

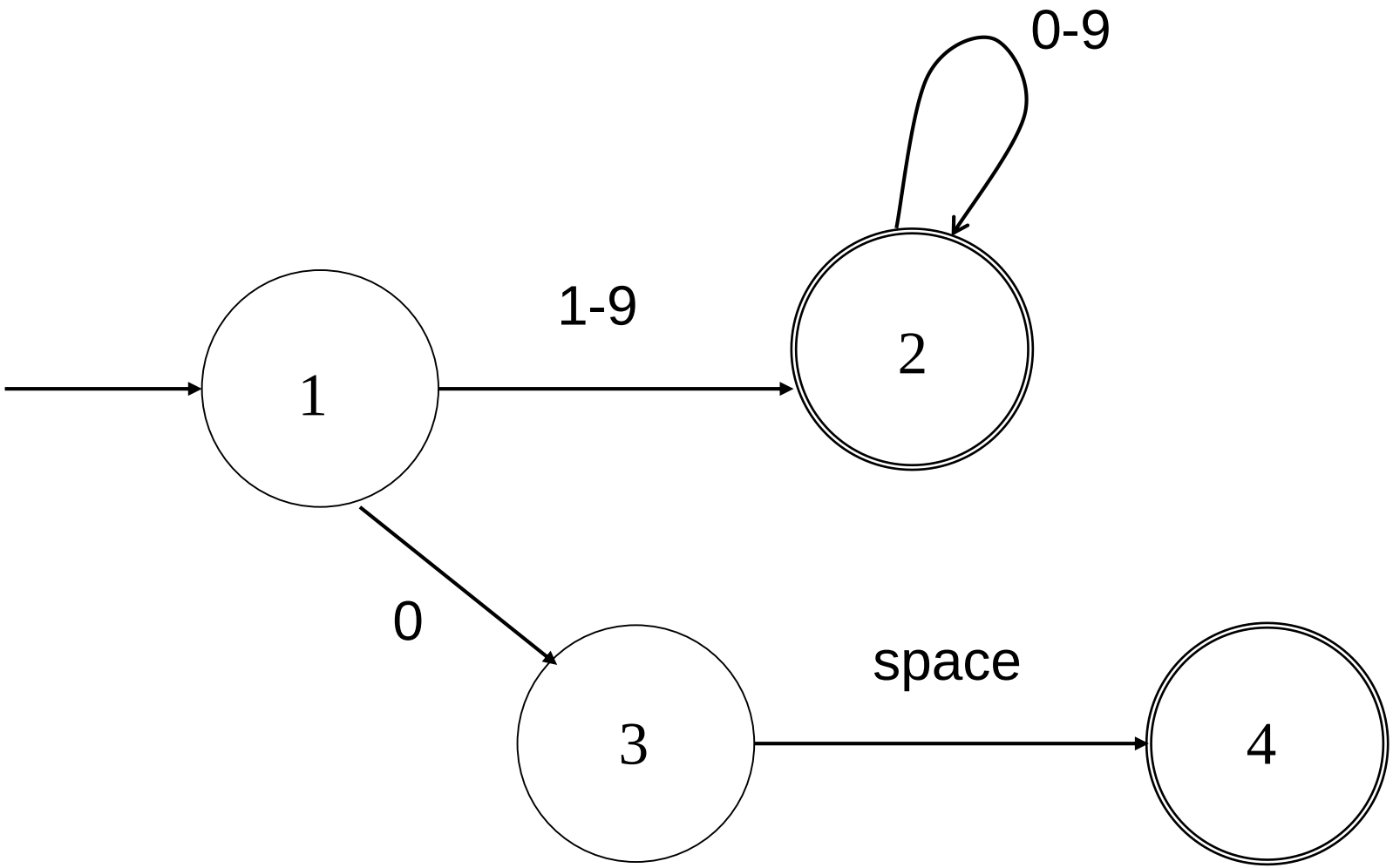


Finite automata for typical programming language tokens



(c)

Finite automata for typical programming language tokens



(d)

Finite automata for typical programming language tokens

Token recognition : algorithmic code

```
get first Input_Character
if Input_Character in[‘A’..’Z’] then
  begin
    while Input_Character in[‘A’..’Z’,‘0’..’9’] do
       begin
          get next Input_Character
          if Input_Character = ‘_’ then
               begin
                 get next Input_Character
                 Last_Char_Is_Underscore := true;
               end  {if‘_’}
          else
              Last_Char_Is_Underscore := false;
       end  {While}    { next page }
```

Token recognition : algorithmic code

```
    if Last_Char_Is_Underscore then
       return ( Token_Error )
    else
       return ( Valid_Token )
  end  { if first in['A'..'Z'] }
else
   return  (Token_Error)
```

- Token recognition : tabular representation of finite automaton.

State	A-Z	0-9	—	
1	2			{starting state}
2	2	2	3	{final state}
3	2	2		

- Notation as follows:

  Nat = [0-9]+

  signedNat = (+|-)? Nat

  number =signedNat (“.” Nat)? (E signedNat)?

- We would like to write down DFAs for the strings matched by these definitions, but it is helpful to first rewrite them as follows:

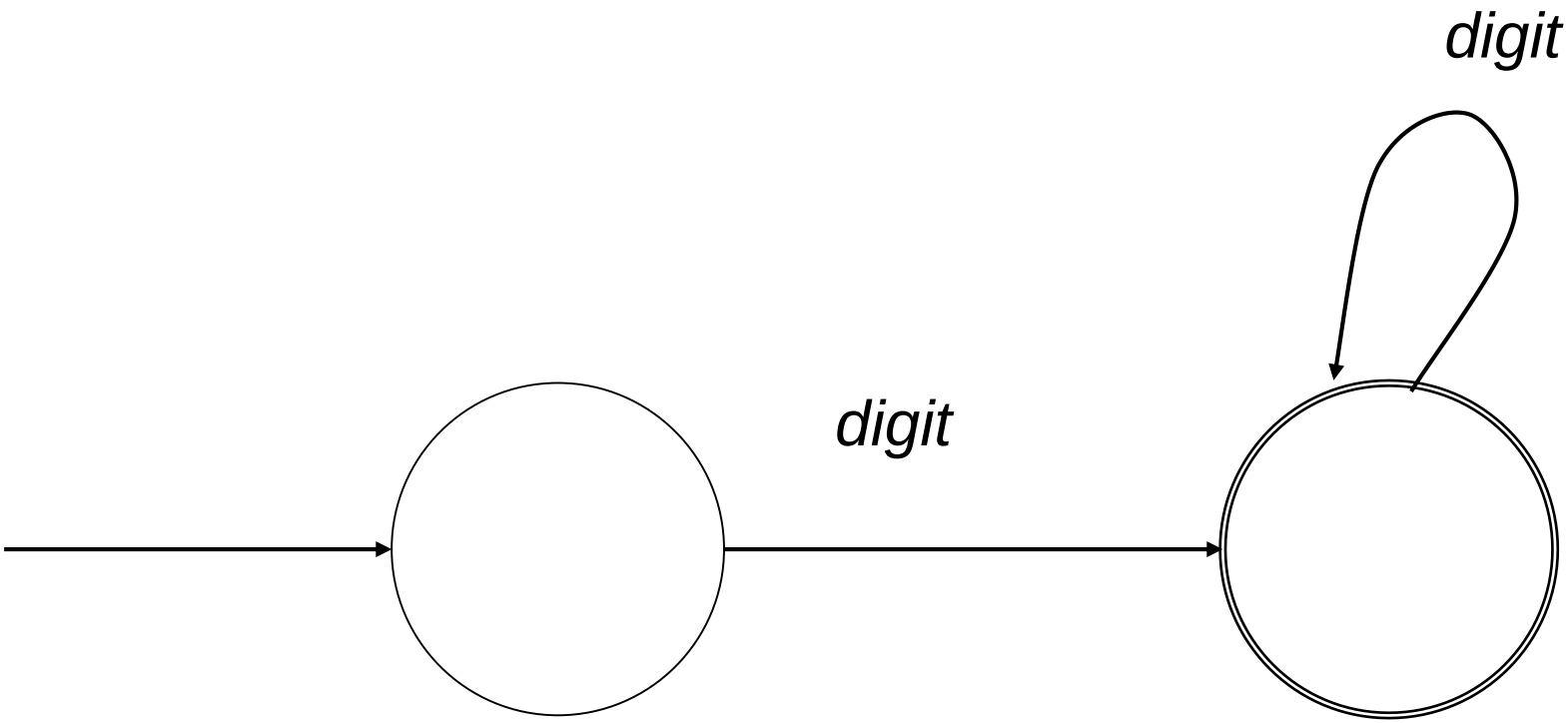
  digit = [0-9]

  Nat = digit+

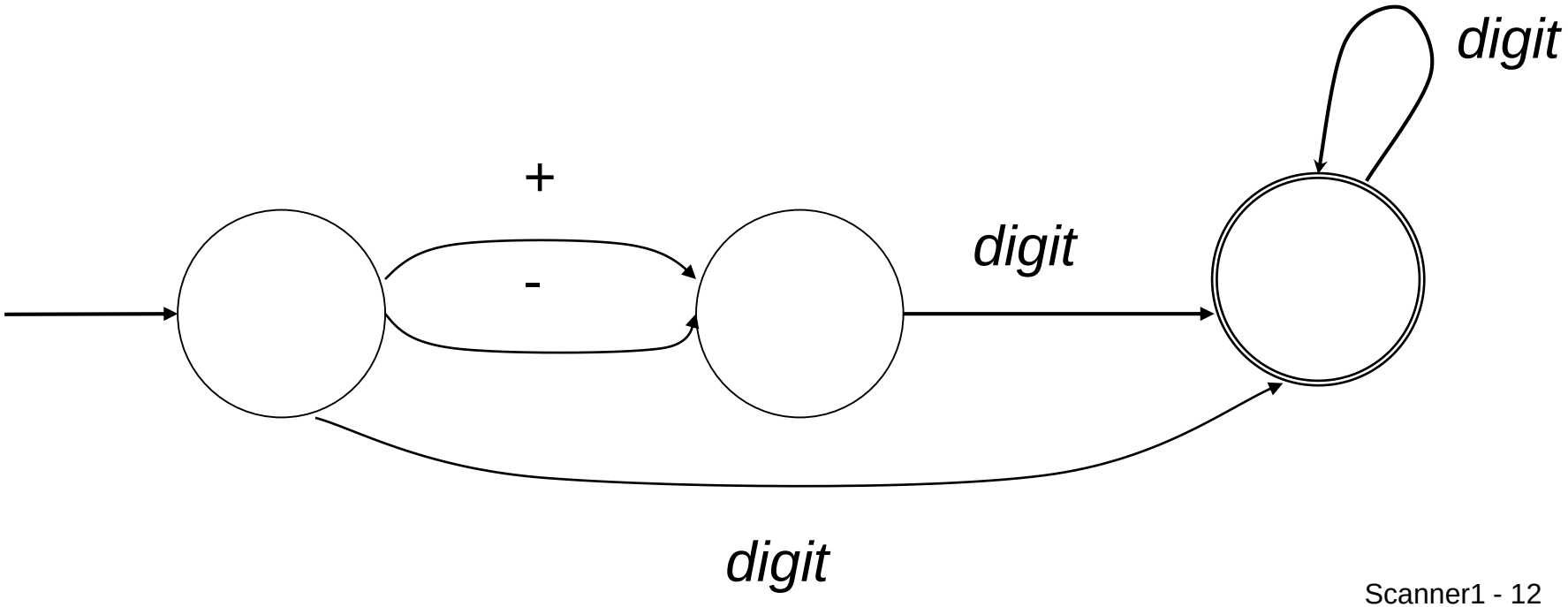
  signedNat = (+|-)? Nat

  number = signedNat(“.” Nat)? (E signedNat)?

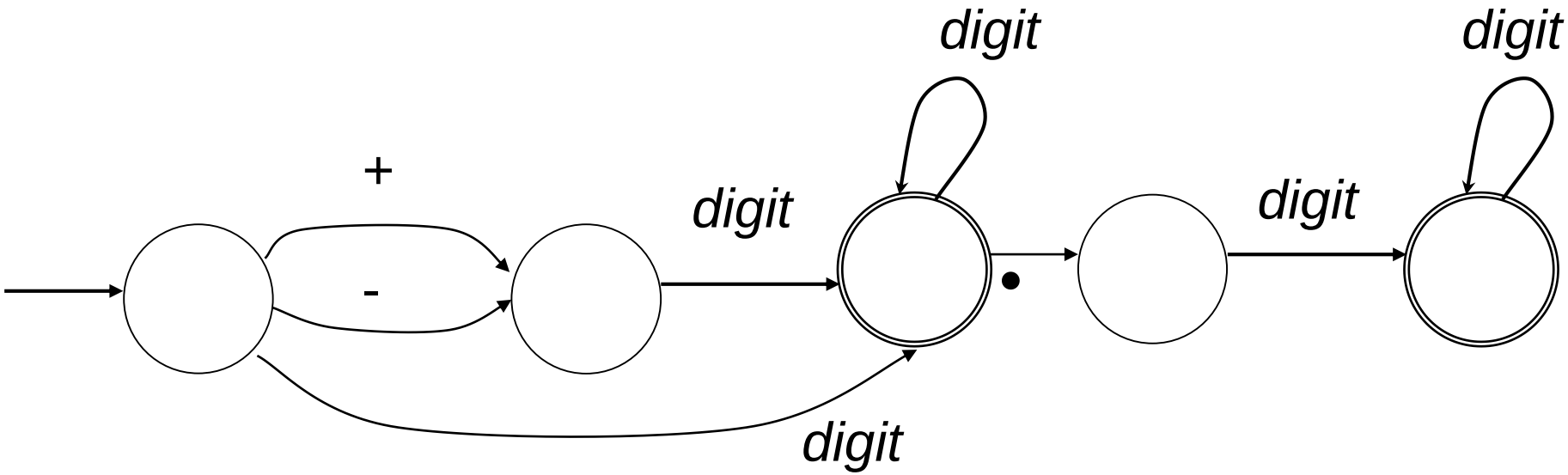
- It is easy to write down a DFA for **Nat** as follows (recall that **a+ = aa*** for any **a**):



- A signedNat is a little more difficult because of the optional sign. However, we may note that a signedNat begins either with a digit or a sign and a digit and then write the following DFA:



- It is also easy to add the optional fractional part, as follows:



- Note that we have kept both accepting states, reflecting the fact that the fractional part is optional.

- Finally, we need to add the optional exponential part. To do this, we note that the exponential part must begin with the letter *E* and can occur only after we have reached either of the previous accepting states. The final diagram is given in Figure 2.3.

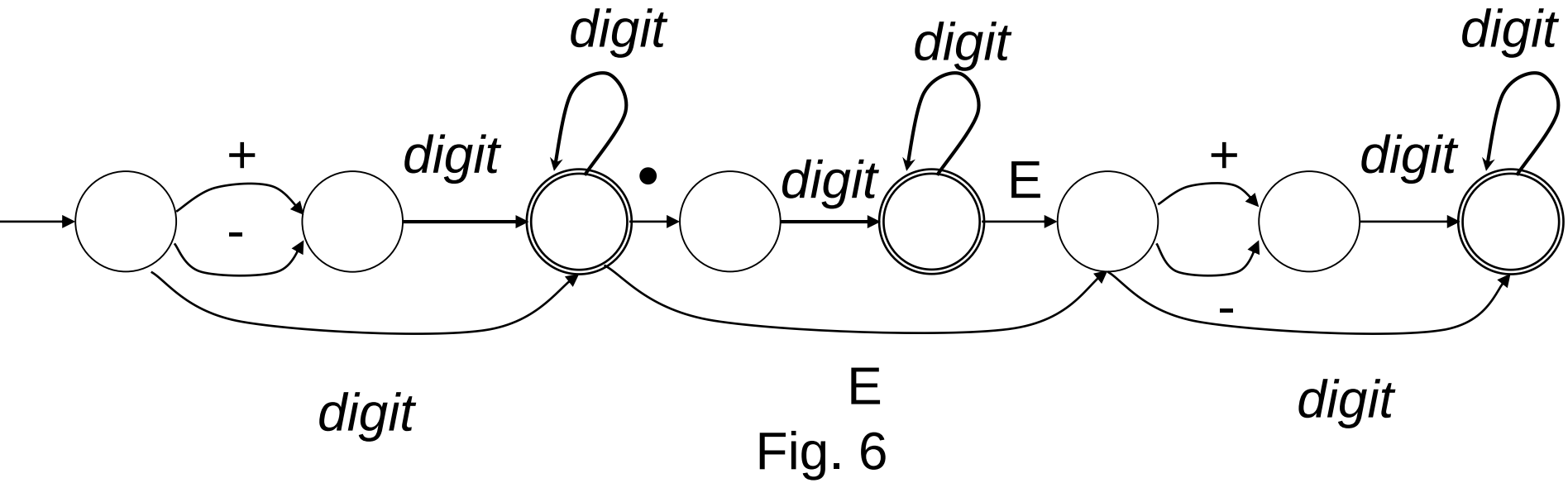
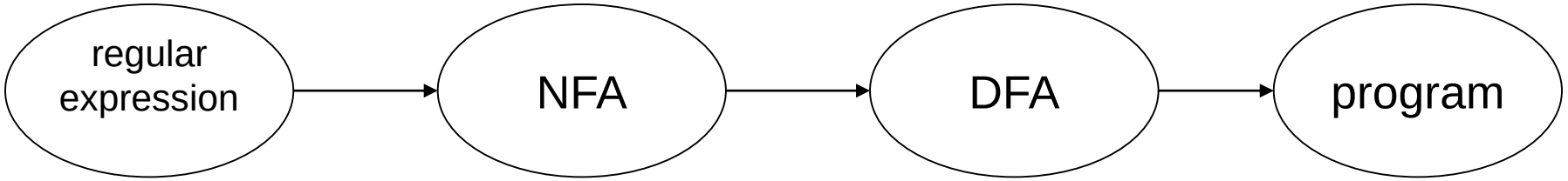


Fig. 6

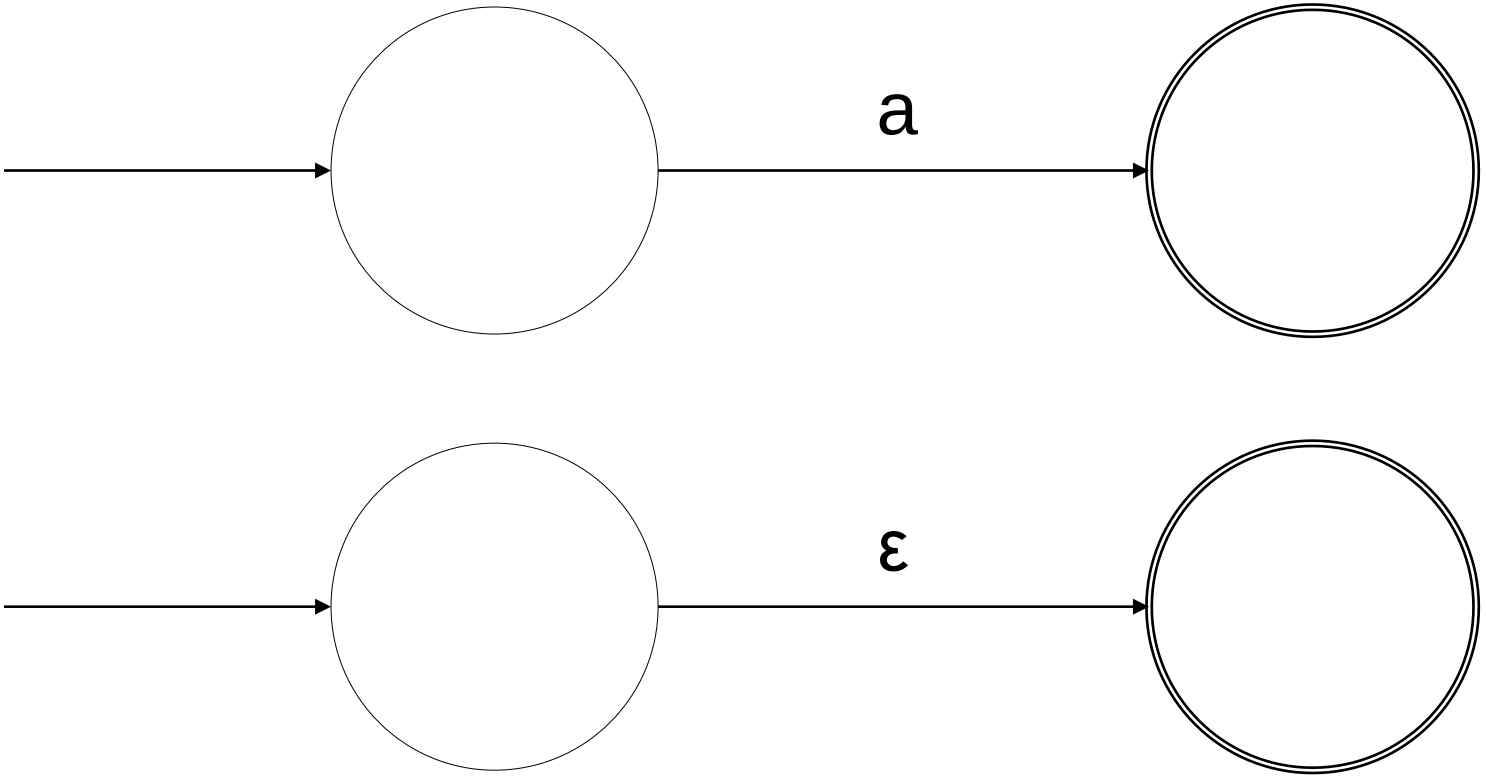
Figure 2.3 A finite automaton for floating-point numbers

From REGULAR EXPRESSIONS To DFAs

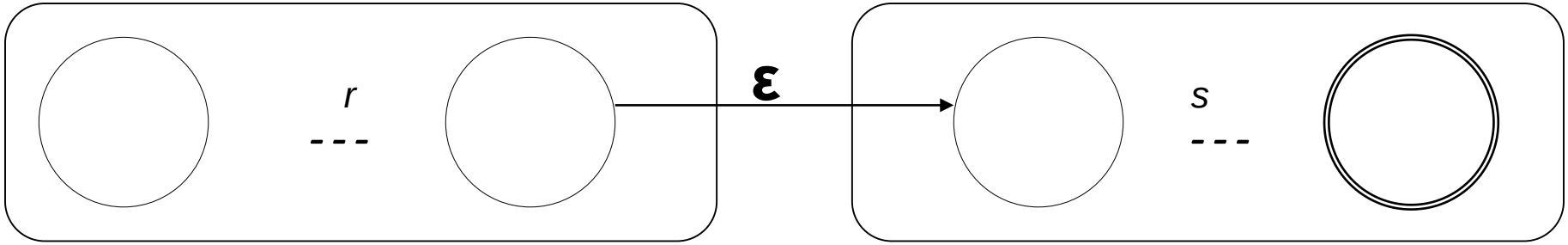
- From a Regular Expression to an NFA



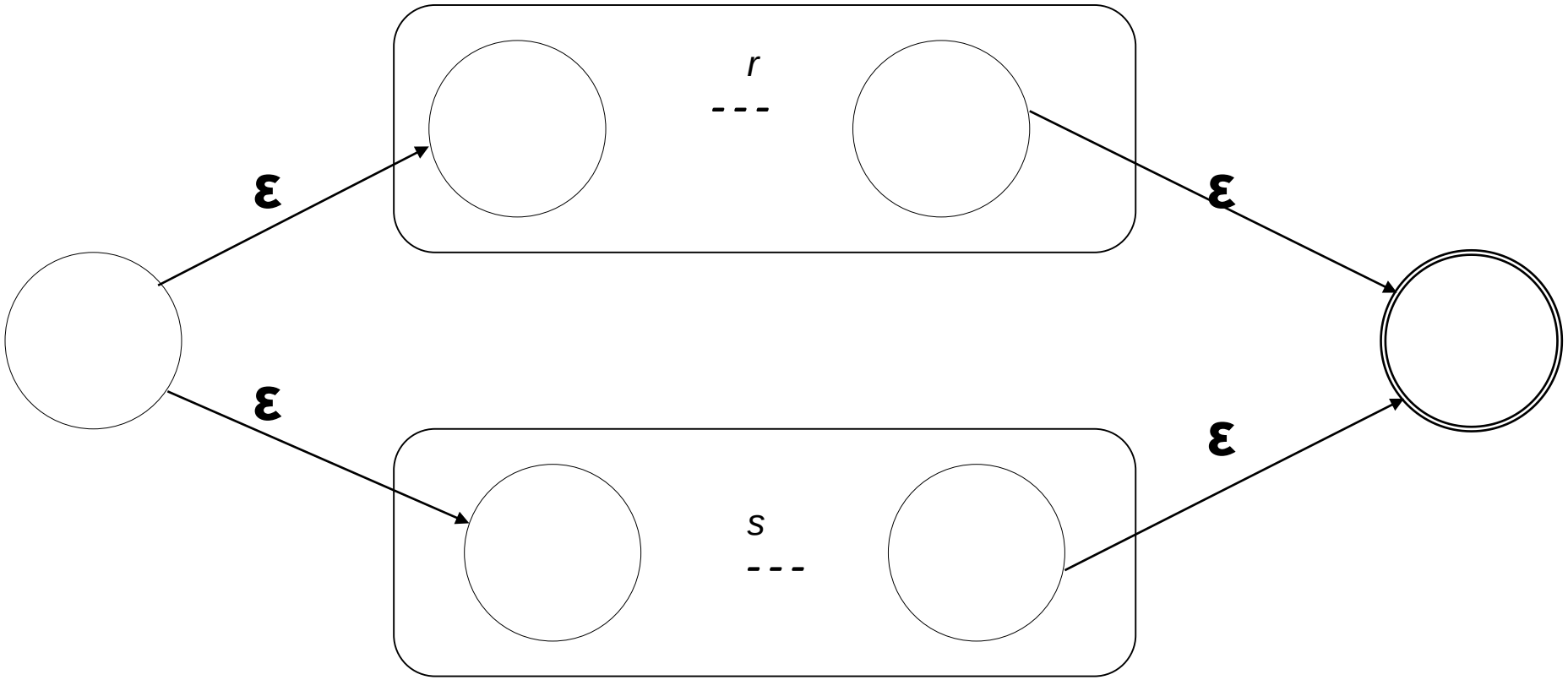
Basic Regular Expressions



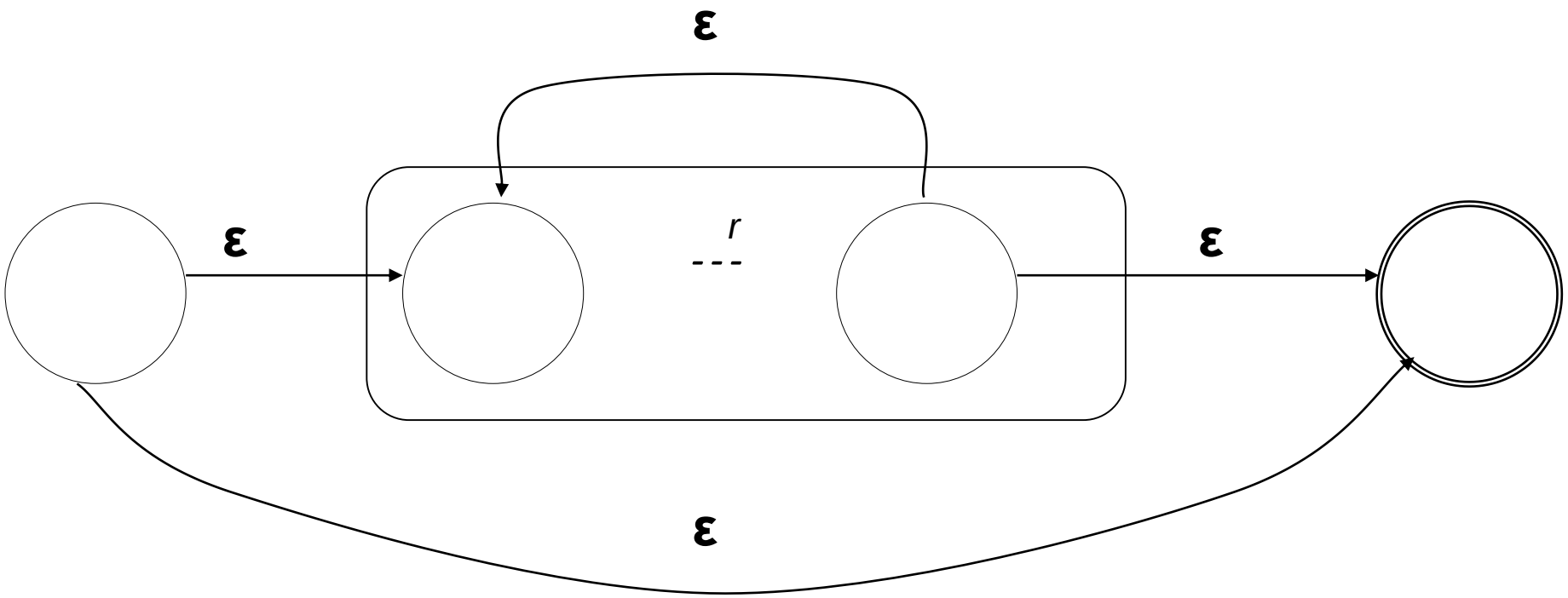
Concatenation



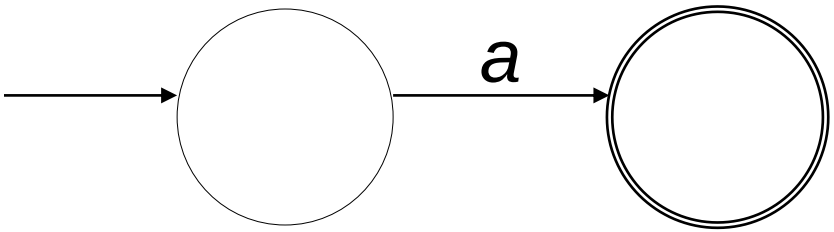
Choice Among Alternatives

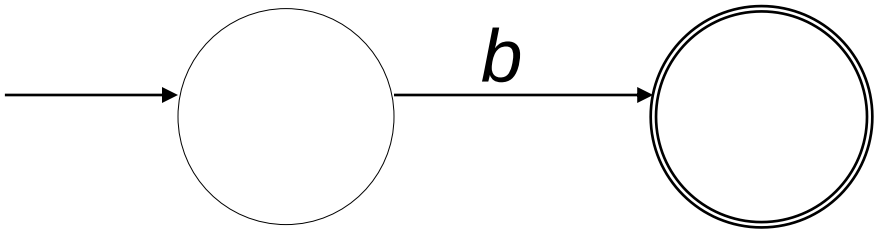


Repetition

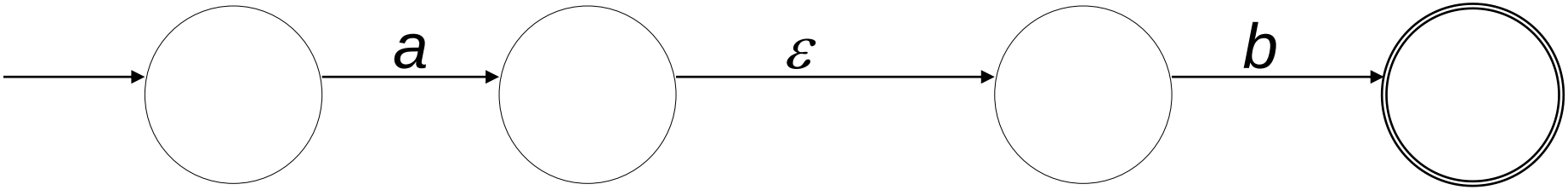


We translate the regular expression on **ab|a** into an NFA according to Thompson's construction. We first form the machines for the basic regular expressions **a** and **b**:

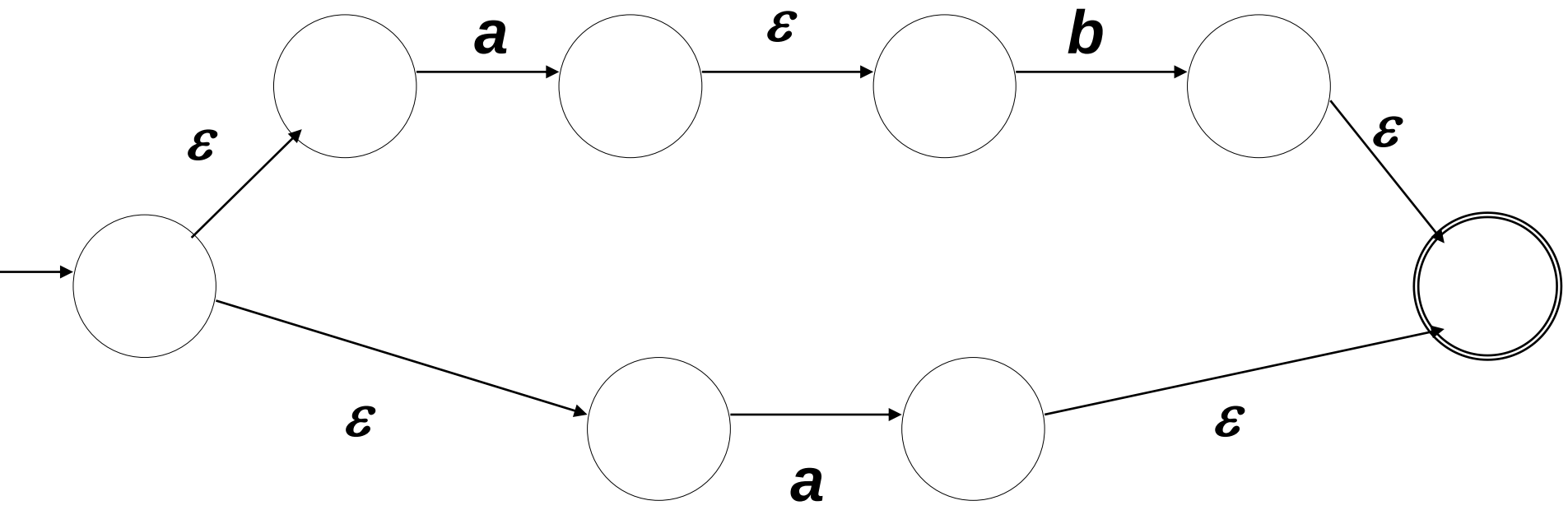




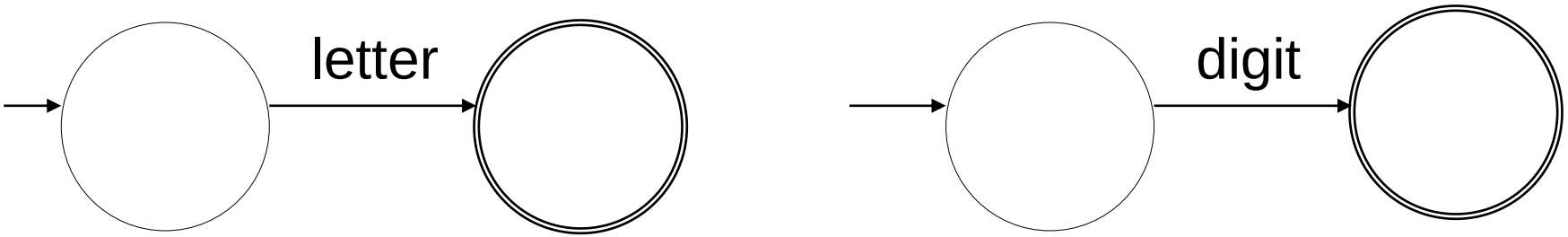
We then form the machine for the concatenation **ab**:



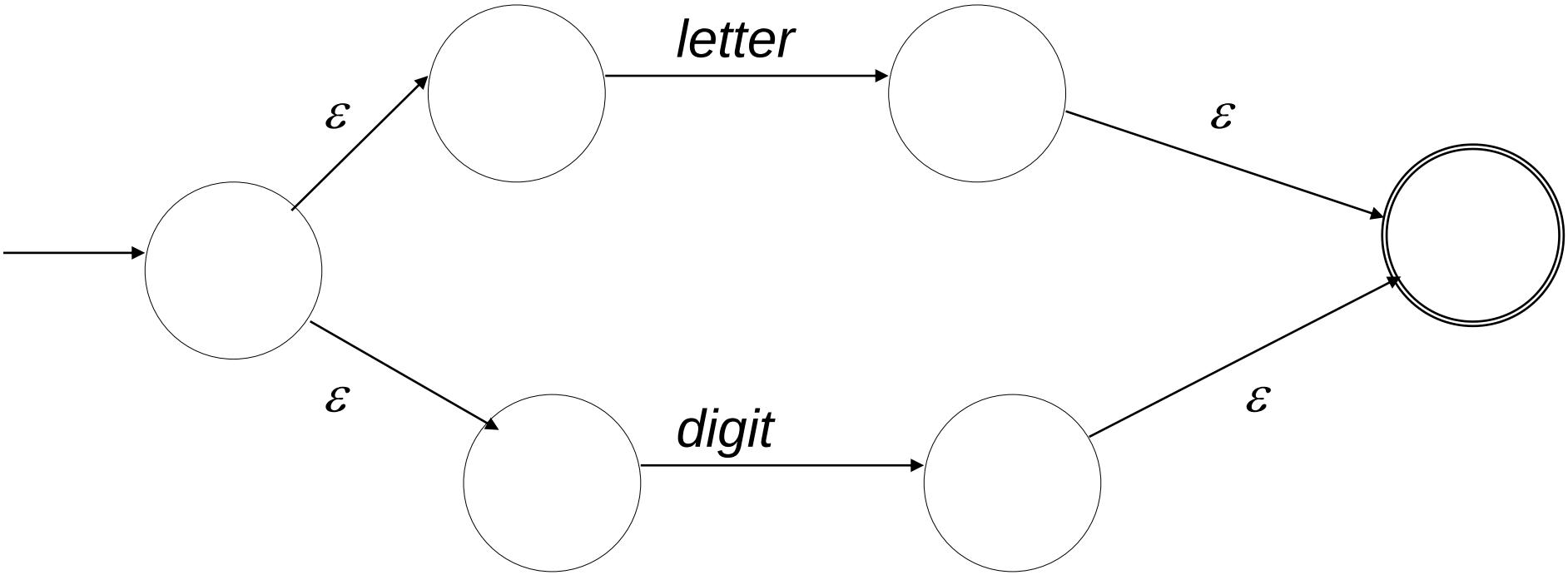
- Now we form another copy of the machine for **a** and use the construction for choice to get the complete NFA for **ab|a** :



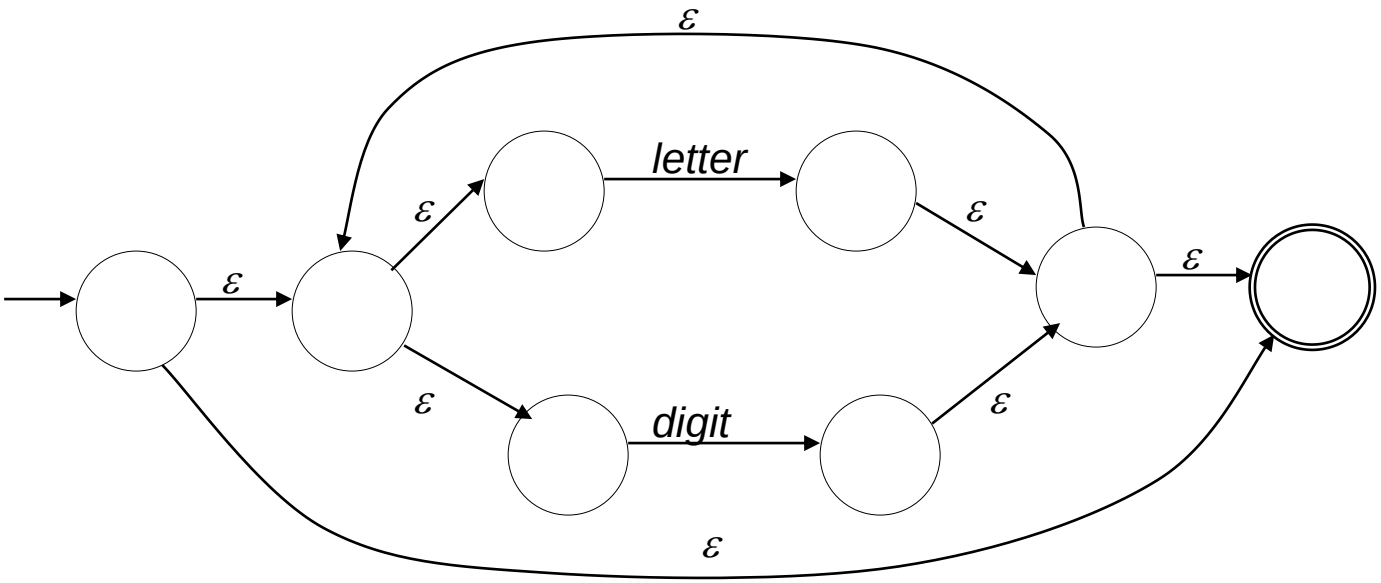
- We form the NFA of Thompson's construction for the regular expression ***letter (letter|digit)****. As in the previous example, we form the machines for the regular expressions ***letter*** and ***digit***:



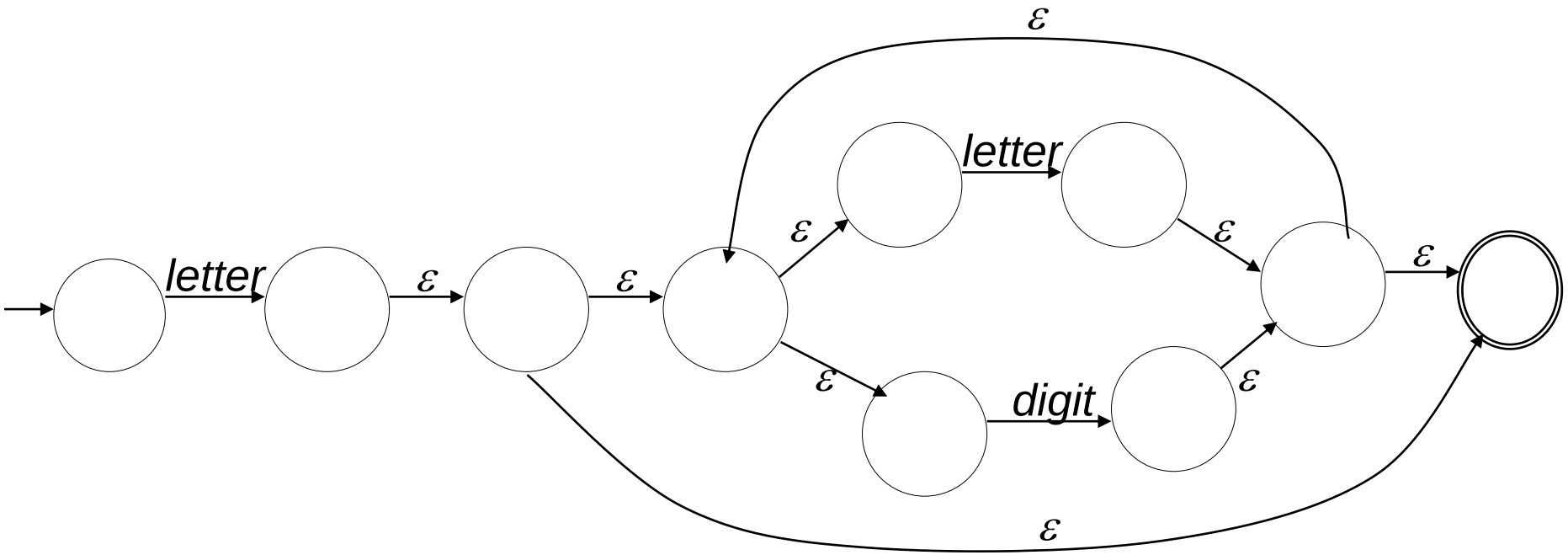
- We then form the machine for the choice ***letter*|*digit***:



Now we form the NFA for the repetition ***(letter|digit)**** as follows:

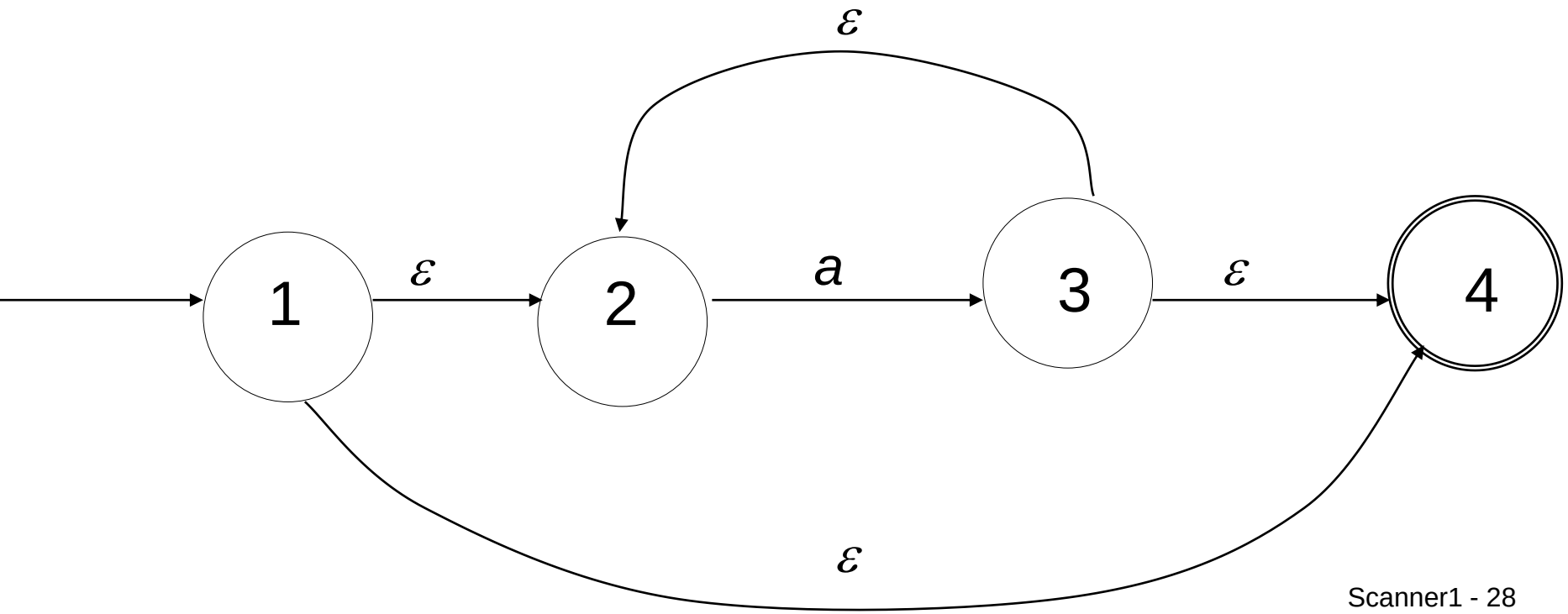


- Finally, we construct the machine for the concatenation of ***letter*** with ***(letter|digit)**** to get the complete NFA

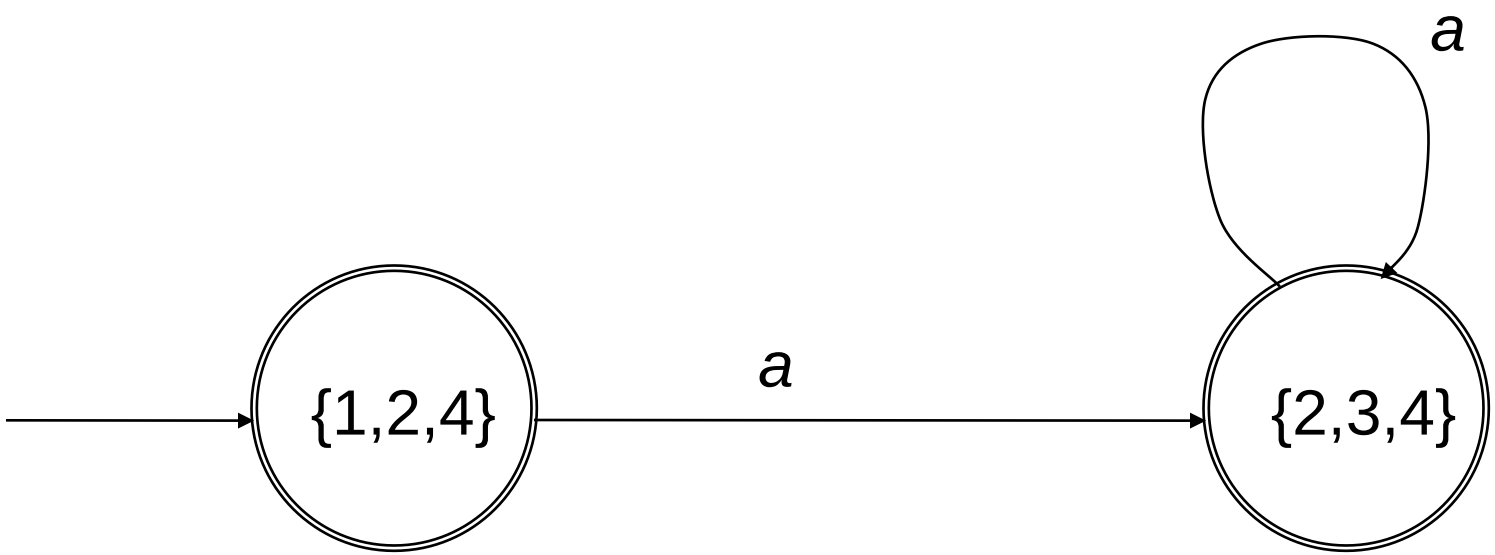


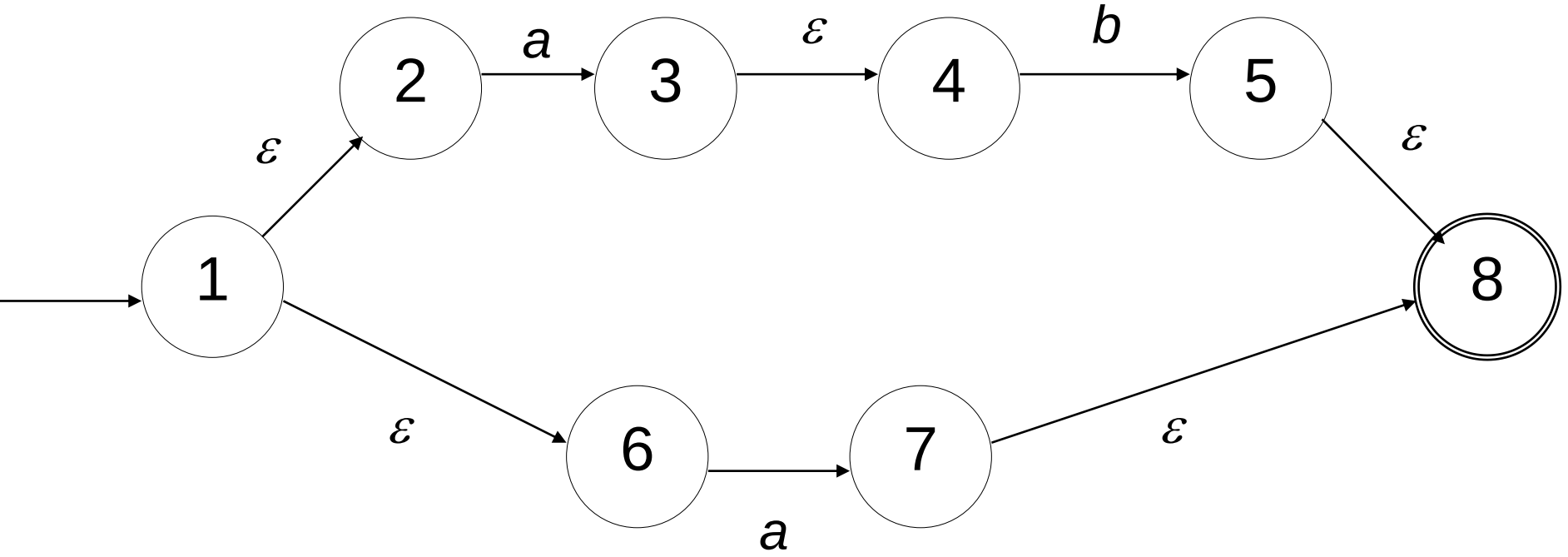
From an NFA to a DFA

- Consider the following NFA corresponding to the regular expression ***a**** under Thompson's construction:



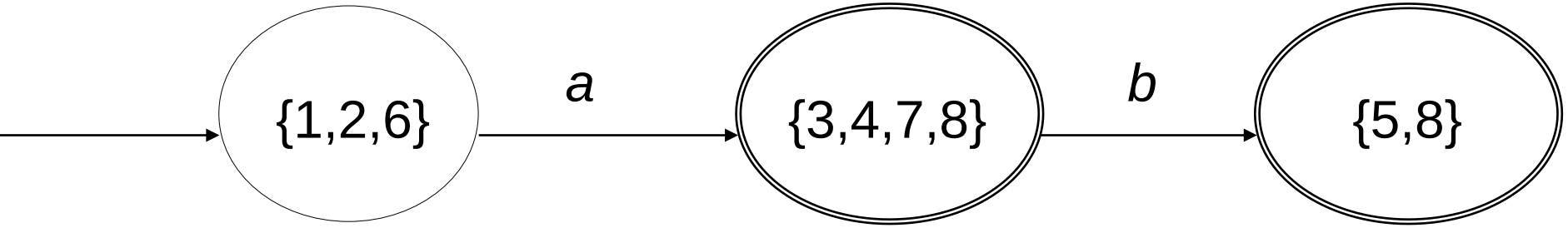
- In this NFA, we have $\overline{\{1\}}$={1,2,4},$\overline{\{2\}}$={2}, $\overline{\{3\}}$ = {2,3,4}, and $\overline{\{4\}}$ ={4}.



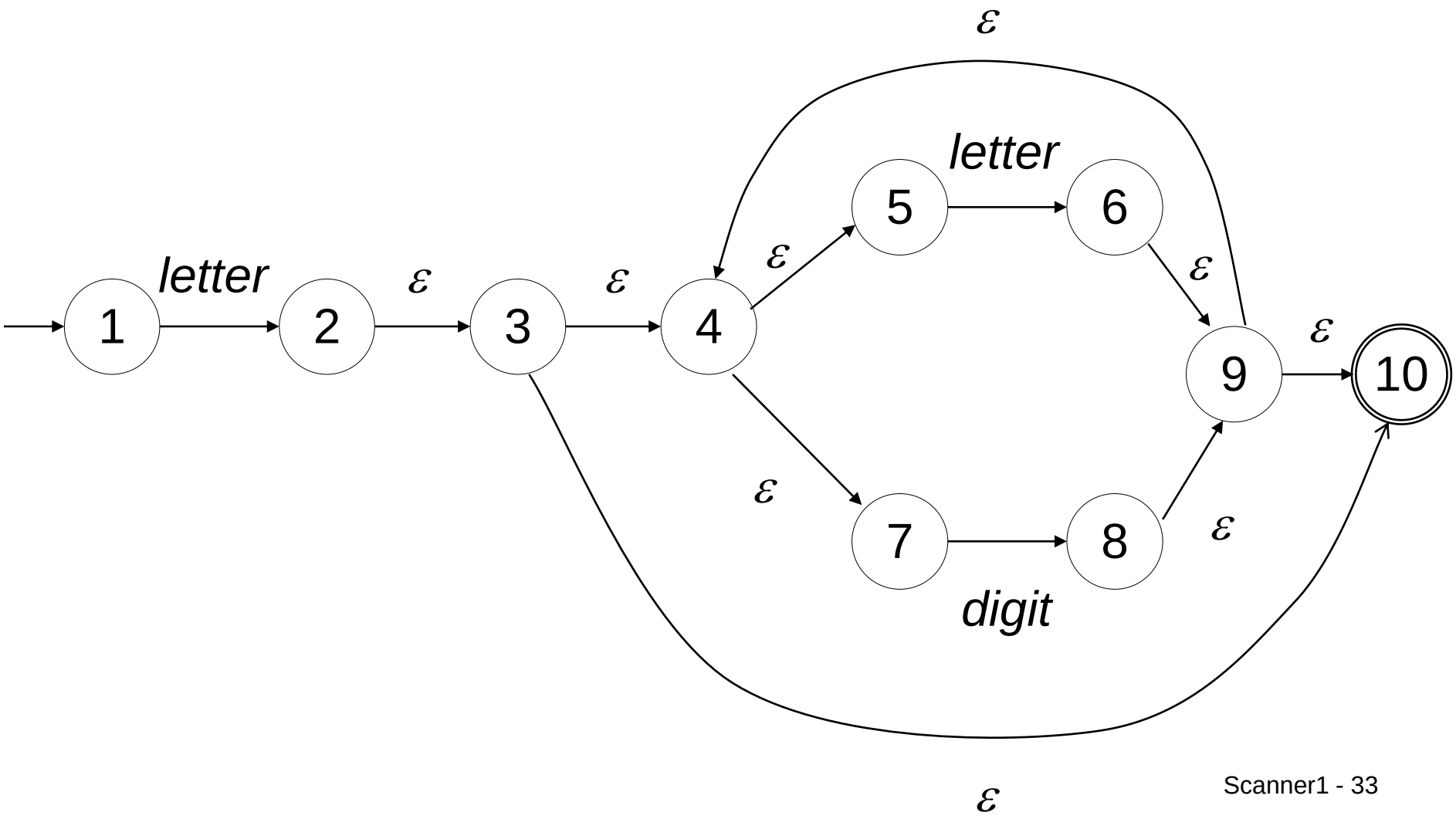

1
2
3
4
5
6
7
8
ε
a
ε
b
ε
ε
ε
a
ε

- The DFA subset construction has as its start state $\{ \overline{1} \} = \{ 1 , 2, 6 \}$. There is a transition on *a* from state 2 to state 3, and also from state 6 to state 7. Thus, $\{ 1, 2, 6 \}_a = \{ \overline{3, 7} \} = \{3, 4, 7, 8\}$, and we have $\{ 1, 2, 6 \} \rightarrow^a \{ 3, 4, 7, 8 \}$. Since there are no other character transitions from 1, 2, or 6, we go on to { 3 ,4 ,7, 8 }. There is a transition on b from 4 to 5 and $\{ 3, 4, 7, 8 \}_b = \{ \overline{5} \} = \{ 5, 8 \}$, and we have the transition $\{ 3, 4, 7, 8\} \rightarrow^b \{ 5, 8 \}$. There are no other transitions.

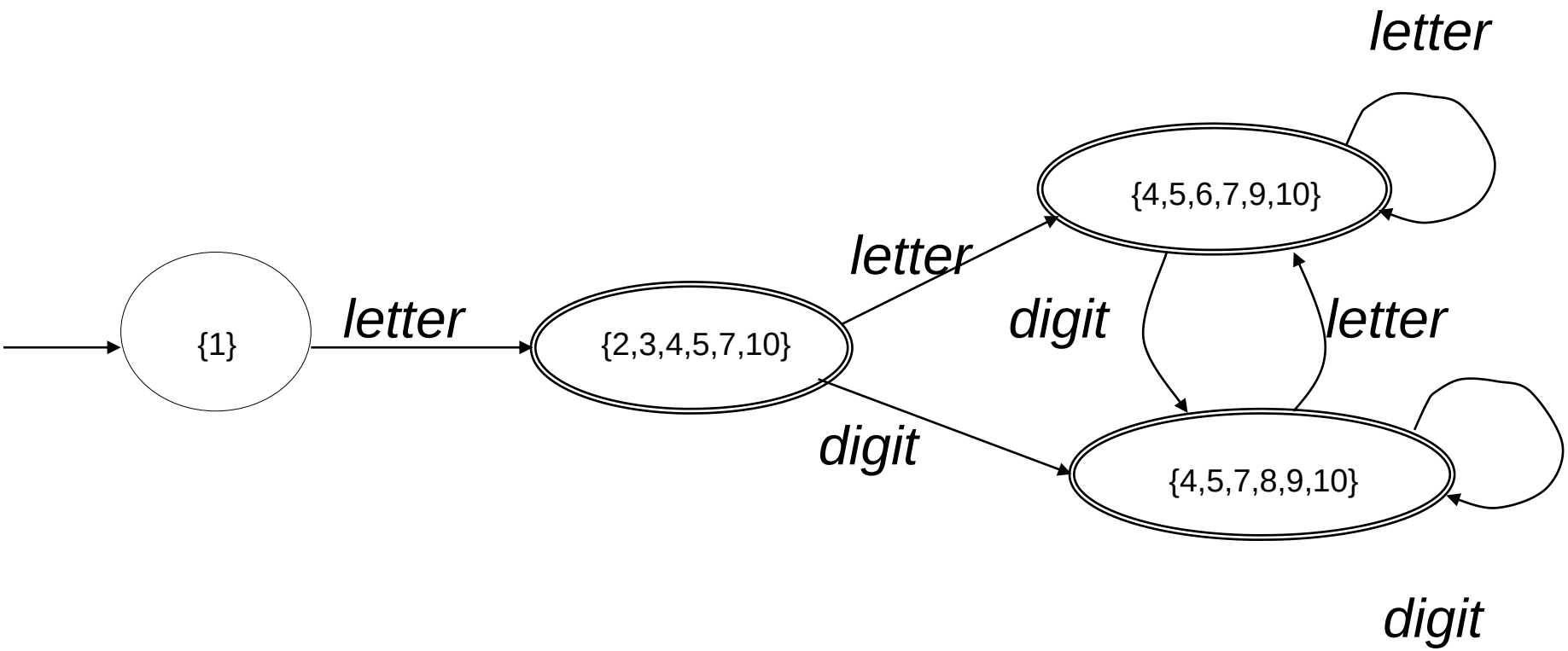
- Thus, the subset construction yields the following DFA equivalent to the previous NFA:



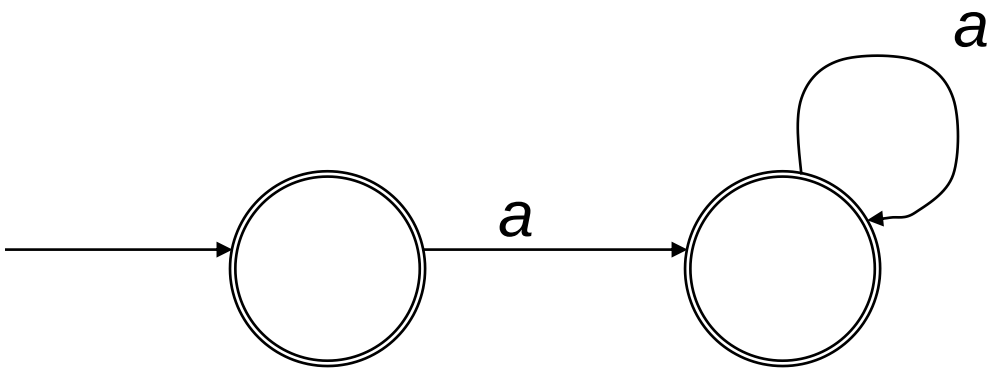
- letter (letter | digit)*:



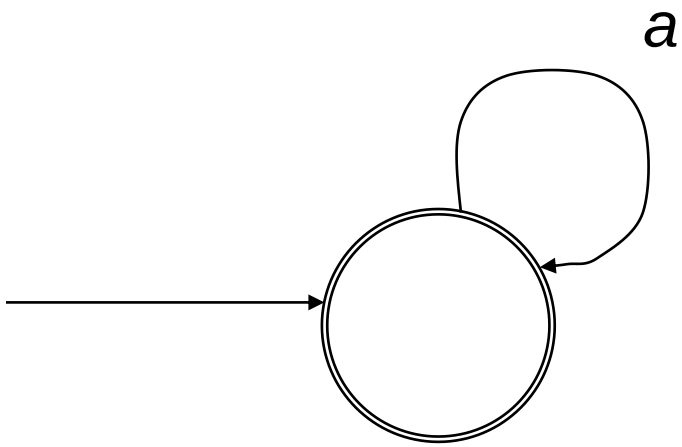
- The subset construction proceeds as follows. The start state is { $\overline{1}$ } = { 1 }. There is a transition on ***letter*** to { 2 } = { 2, 3, 4, 5, 7, 10 }. From this state there is a transition on ***letter*** to { $\overline{6}$ } = { 4, 5, 6, 7, 9. 10 } and a transition on ***digit*** to { $\overline{8}$ } = { 4, 5, 7, 8, 9, 10 }. Finally, each of these states also has transitions on ***letter*** and ***digit***, either to itself or to the other. The complete DFA is given in the following picture:


letter
{4,5,6,7,9,10}
letter
digit
letter
{1}
letter
{2,3,4,5,7,10}
digit
{4,5,7,8,9,10}
digit

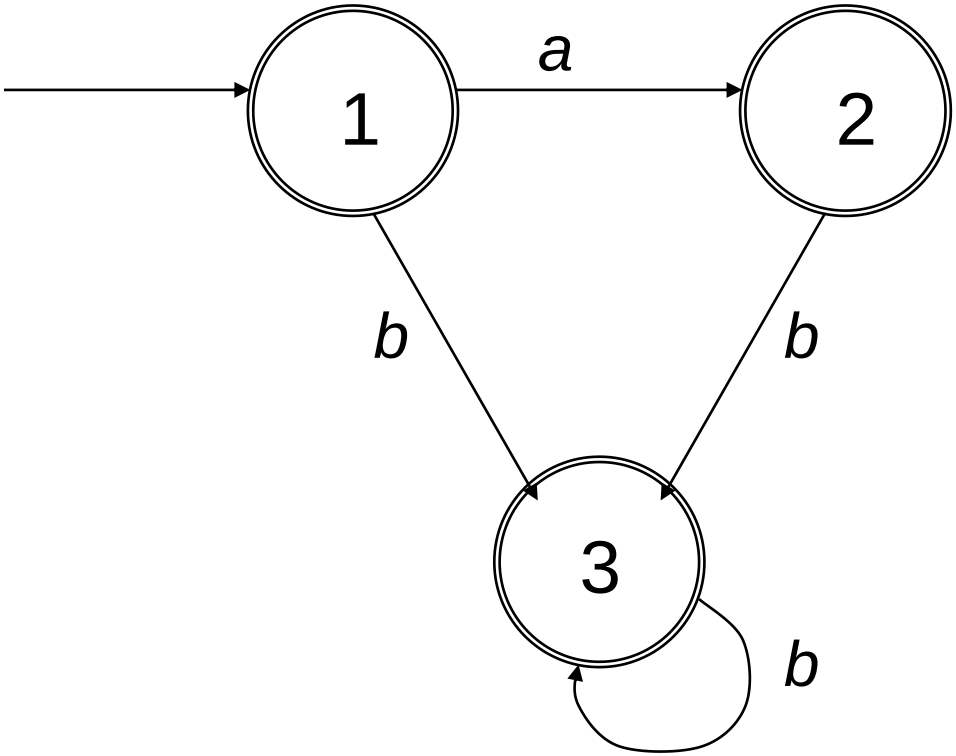
- Minimizing the Number of States in a DFA

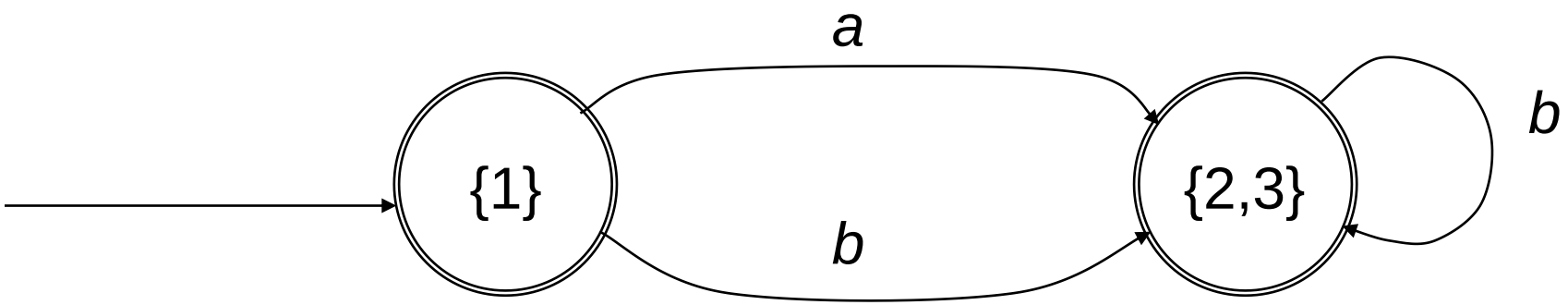


- for the regular expression **a*** . whereas the DFA

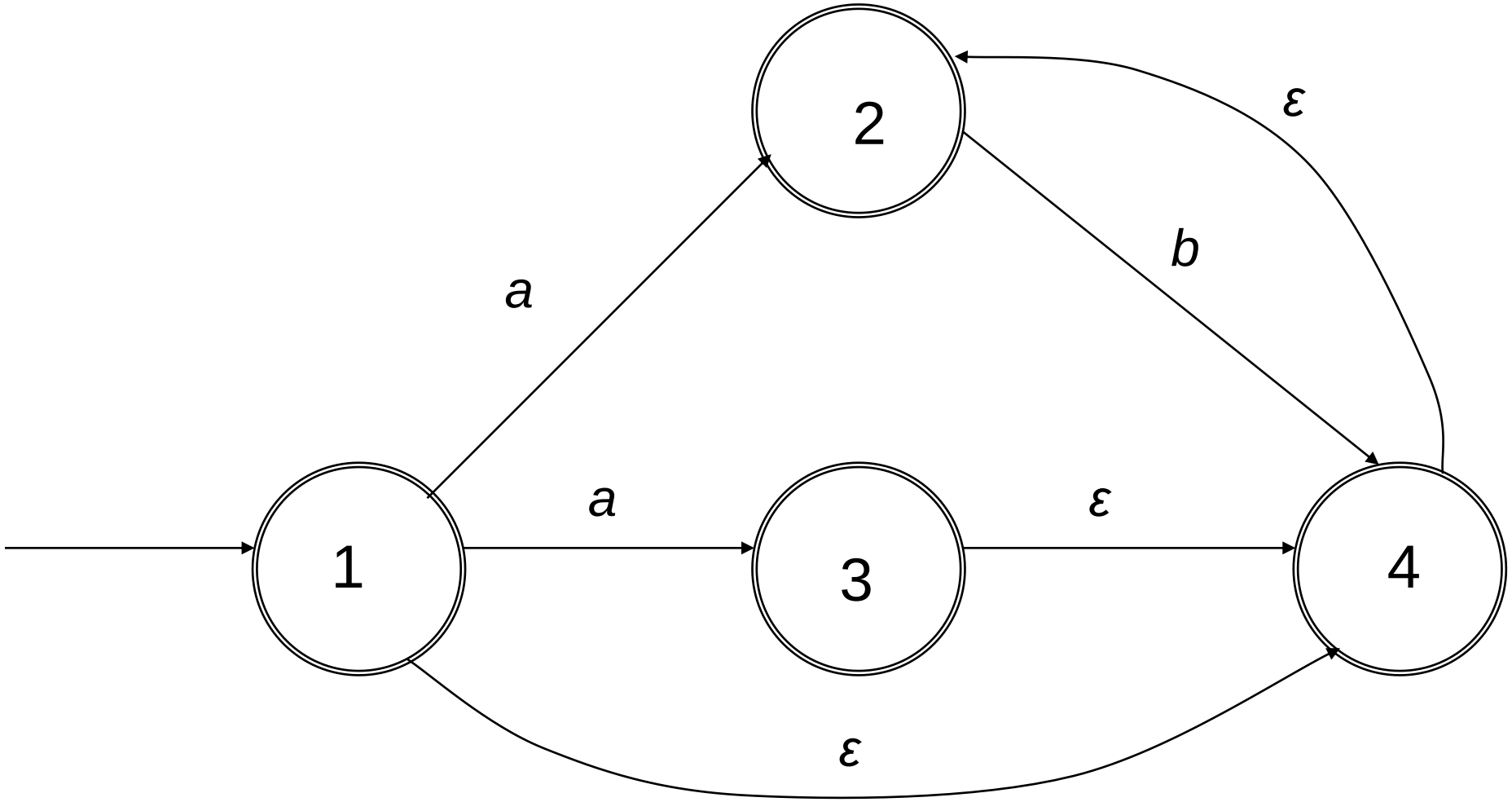


- (a | ε) b* :

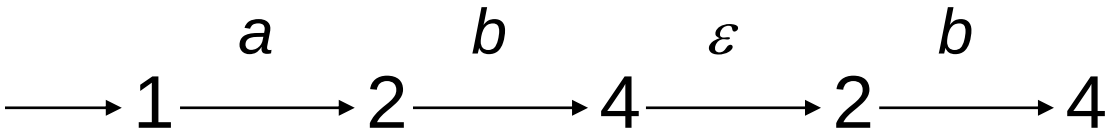


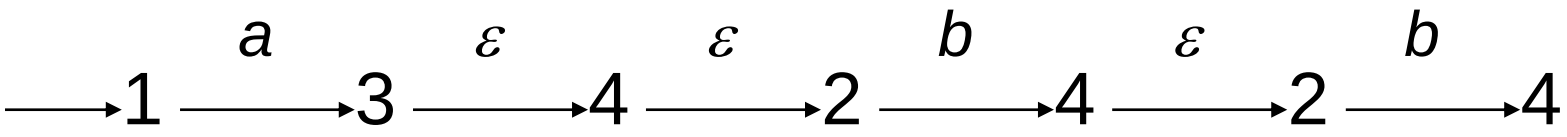


- Consider the following diagram of an NFA.



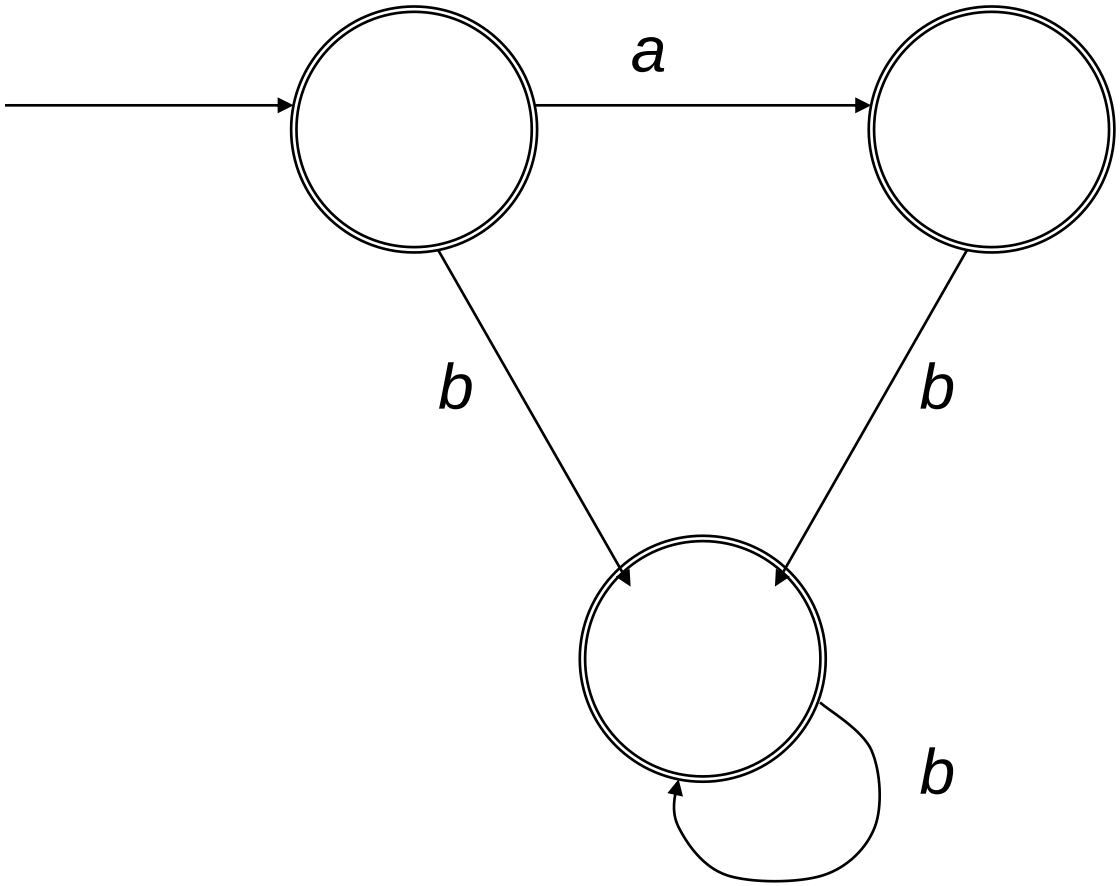
- The string **abb** can be accepted by either of the following sequences of transitions:

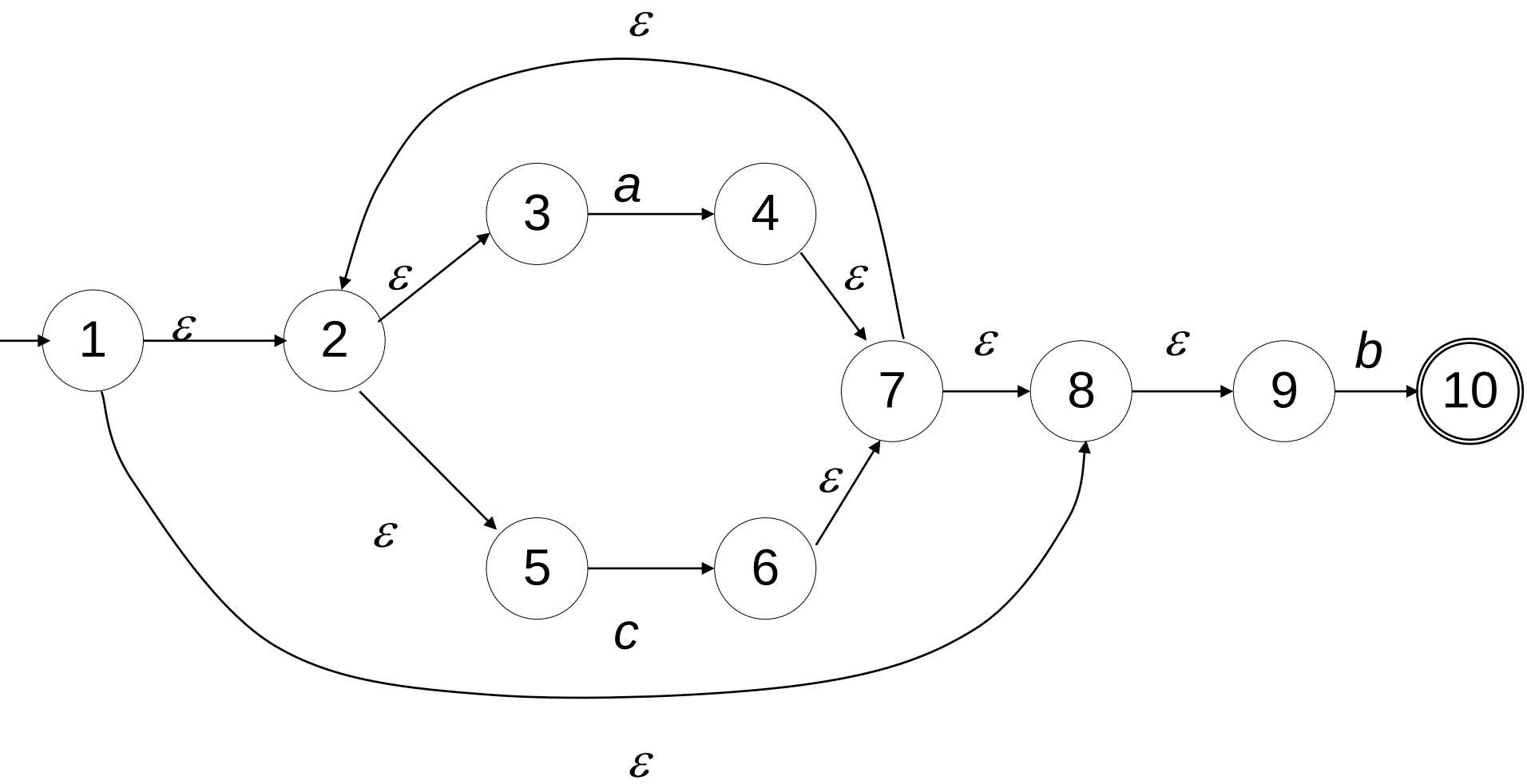




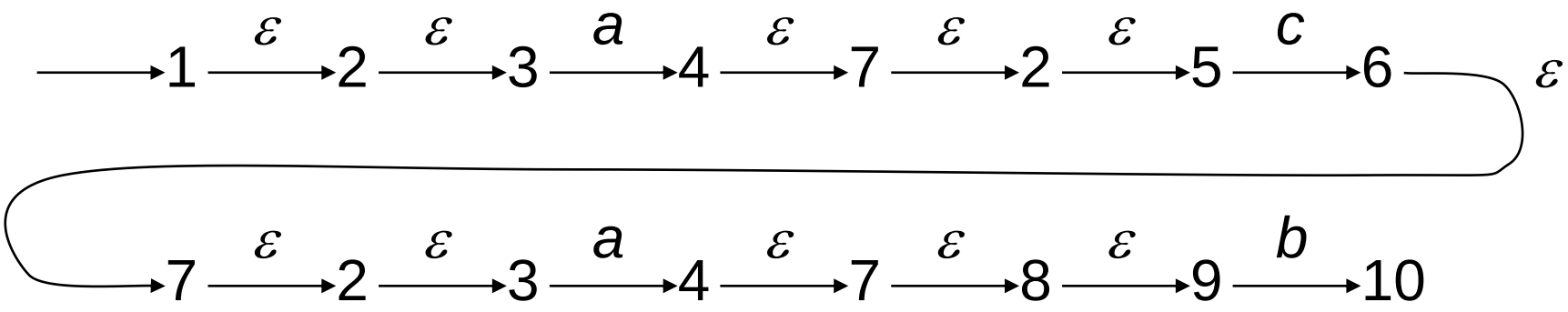
- Indeed the transitions from state 1 to state 2 on a, and from state 2 to state 4 on b, allow the machine to accept the string ab, and then, using the ε -transition from state 4 to state 2, all strings matching the regular expression **ab+**. Similarly, the transitions from state 1 to state 3 on a, and from state 3 to state 4 on ε , enable the acceptance of all strings matching **ab***. Finally, following the ε -transition from state 1 to state 4 enables the acceptance of all strings matching b*. Thus, this NFA accepts the same language as the regular **ab+|ab*|b***. A simpler regular expression that generates the same language is **(a|** ε **)b***.

- The following DFA also accepts this language:



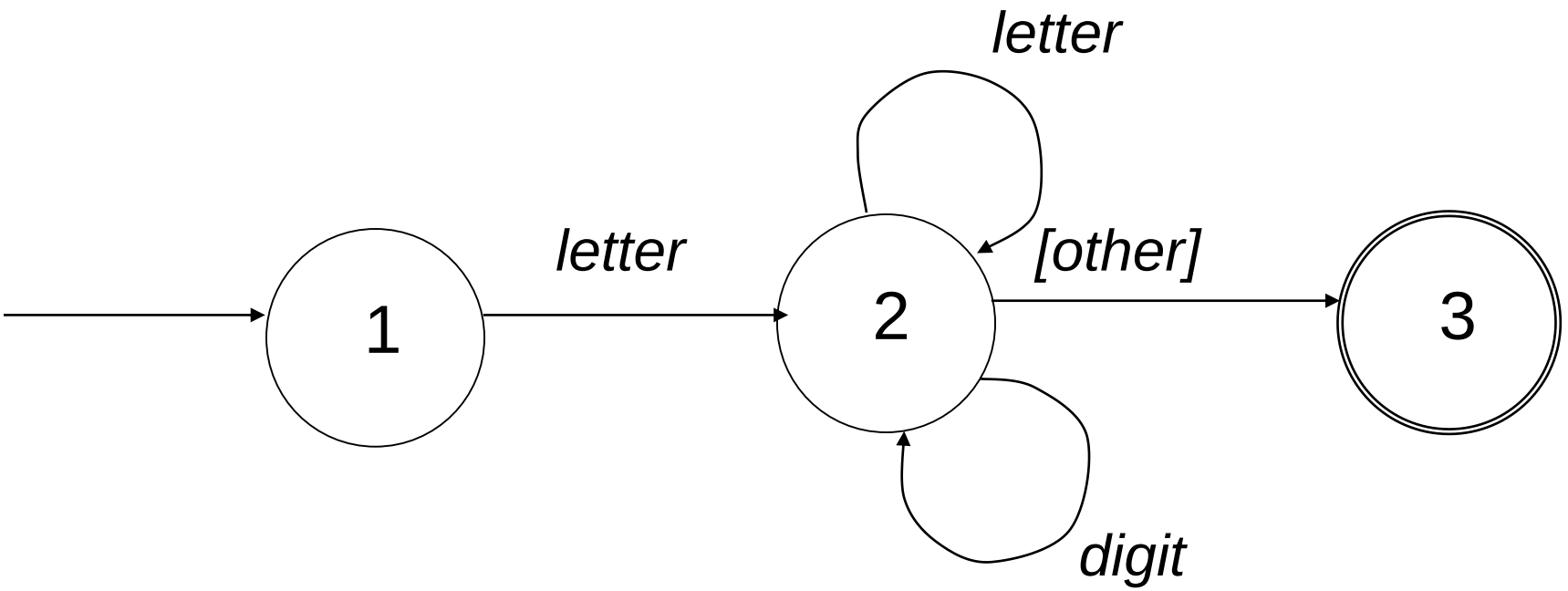

ε
3
a
4
ε
ε
1
ε
2
7
ε
8
ε
9
b
10
ε
ε
5
6
c
ε

- It accepts the string *acab* by making the following transitions:



- In fact, it is not hard to see that this NFA accepts the same language as that generated by the regular expression **(a|c)*b**

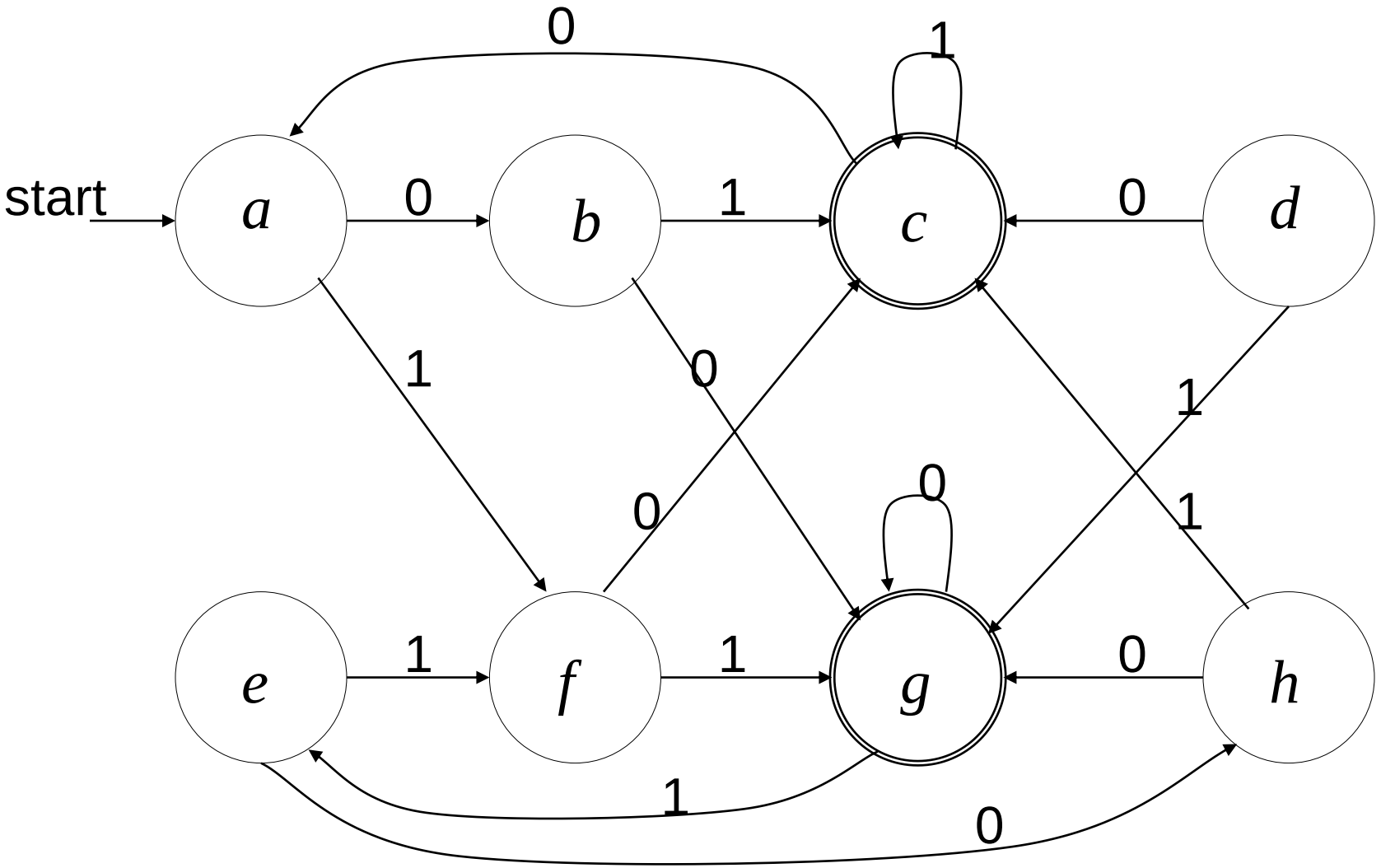
Implementation of Finite Automata in Code



- The first and easiest way to simulate this DFA is to write code in the following form:

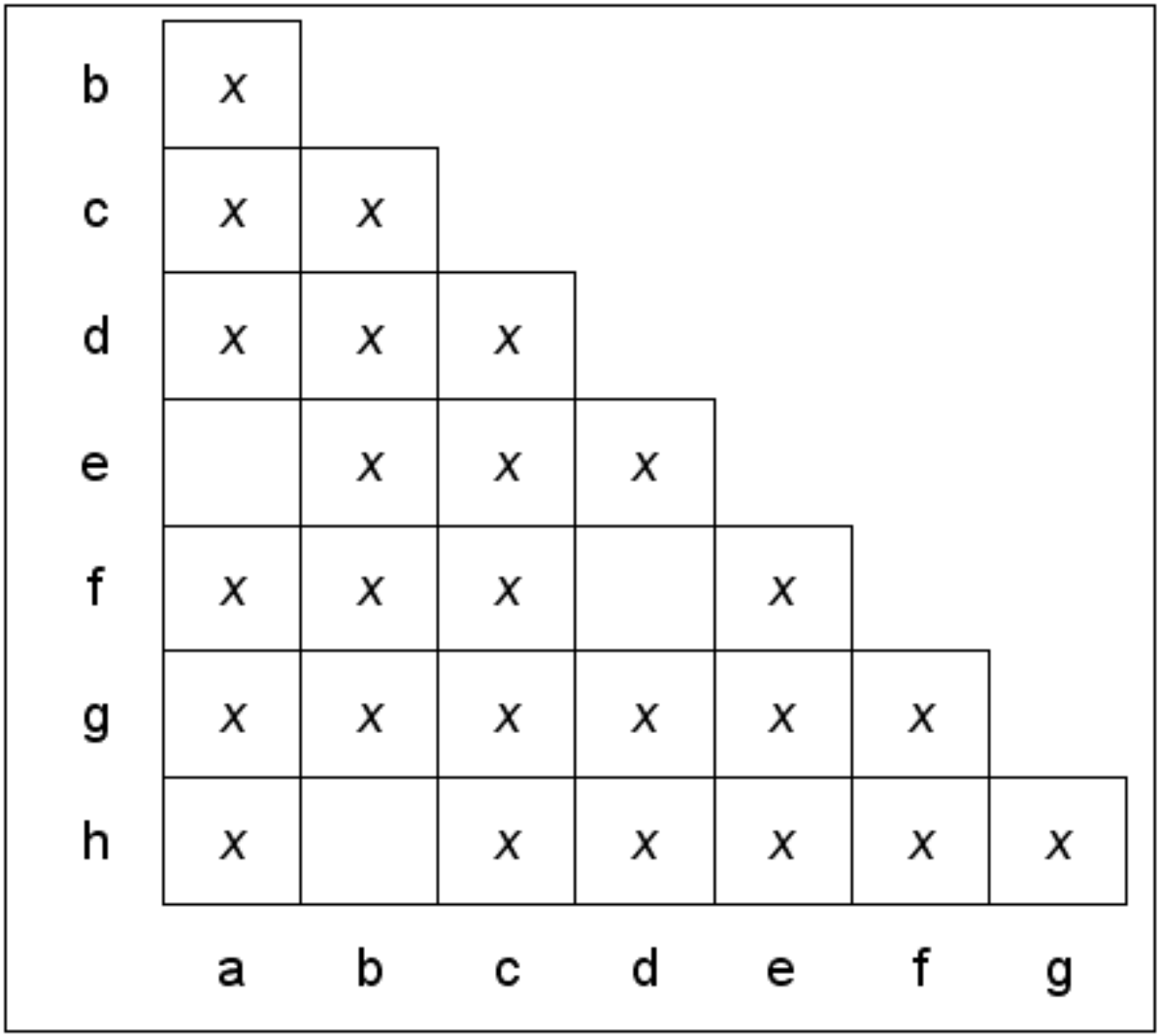
```
{ starting in state 1 }
if the next character is a letter then
  advance the input;
  { now in state 2 }
  while the next character is a letter or a digit do
    advance the input; { say in state 2 }

  end while;
{ go to state 3 without advancing the input }
accept;
else
{ error or other cases }
end if;
```

- Minimized DFA



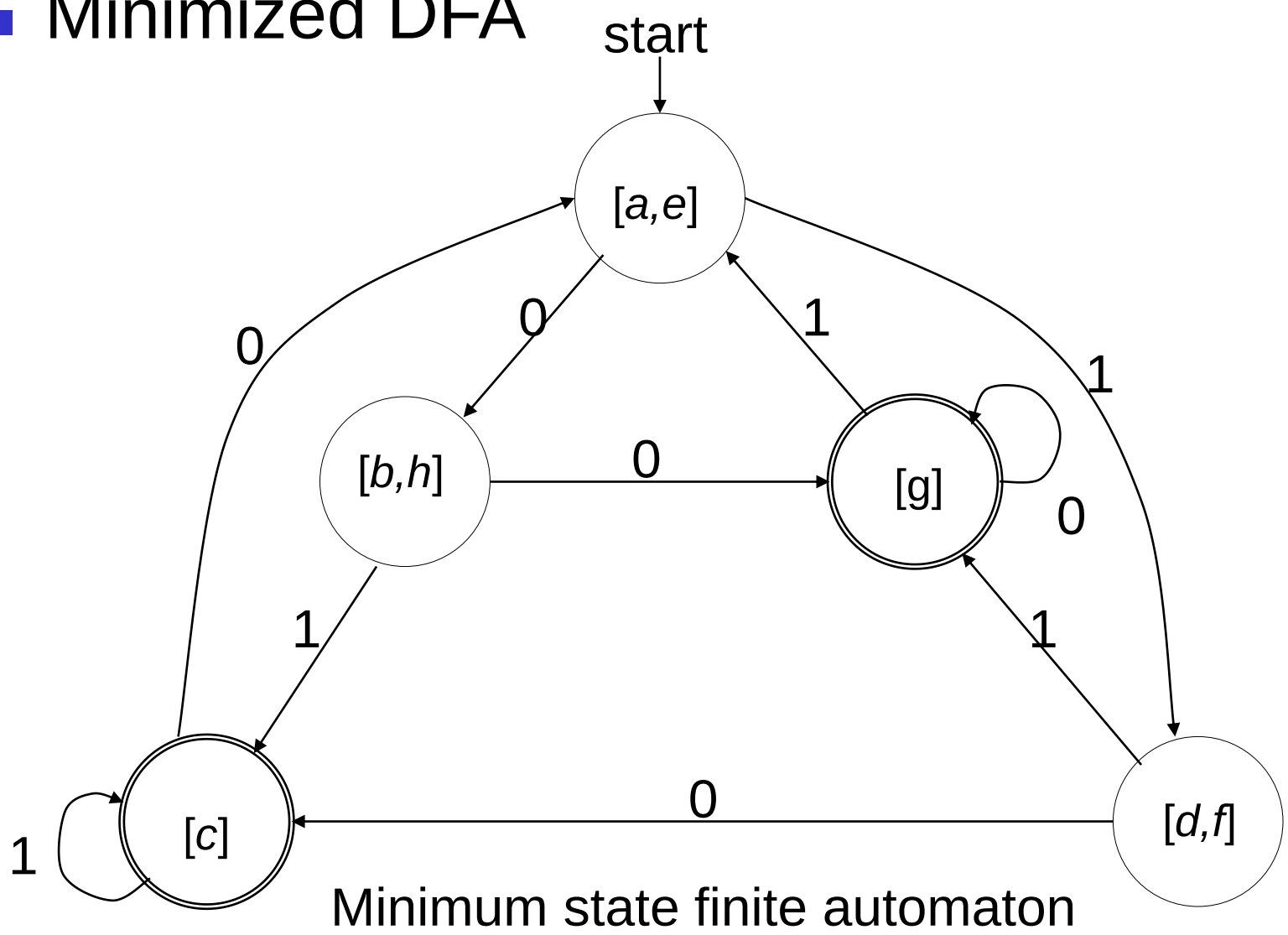
Finite automaton

- Minimized DFA



Calculation of equivalent states

- Minimized DFA



Minimum state finite automaton

Algorithm for marking pairs of inequivalent states.

```
begin
for p in F and q in Q-F do mark (p, q);
for each pair of distinct states (p,q) in F×F or
    (Q-F)×(Q-F) do
  if for some input symbol a,(δ(p,a),δ(q,a)) is marked
  then
    begin
      mark (p,q);
      recursively mark all unmarked pairs on the list
      for (p,q) and on the lists of other pairs that
      are marked at this step.
    end
  else /* no pair (δ(p,a),(δ(q,a)) is marked */
    for all input symbols a do
      put (p,q) on the list for (δ(p,a),δ(q,a)) unless
     δ(p,a)=δ(q,a)
end
```