



# Lexical Analysis

*Compilers course*

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year



**João M. P. Cardoso**



Universidade do Porto  
**FEUP** Faculdade de Engenharia

Dep. de Engenharia Informática  
Faculdade de Engenharia (FEUP), Universidade do Porto,  
Porto, Portugal  
Email: [jmpc@acm.org](mailto:jmpc@acm.org)

# Formal Languages

## ➤ Natural Languages

- Ambiguous
  - Problem in the language processing
  - Context dependence allows shorter messages

## ➤ Formal (artificial) Languages

- Obey to rules specified rigorously using appropriate formalisms
- Rules guarantee that the languages is not ambiguous

# Definition of Formal Languages

- Necessity to define precisely a language
- Definition of the languages structured in layers
  - Start by the set of the symbols of the language (the alphabet,  $\Sigma$ )
  - **Lexical structure** – identifies “words” of the language (each word is a sequence of symbols)
  - **Syntactic structure** – identifies “sentences” in the language (each sentence is a sequence of words)
  - **Semantic** – meaning of the program (specifies the results that should be output for the inputs)

# Formal Specification of Languages

- Regular expressions (generative method)
  - There exist cases not possible to describe using regular expressions
- Finite Automata (method by recognition)
  - Non-Deterministic (NFAs)
  - Deterministic (DFAs)
  - Implement any regular expression

# Specification of Lexemas Using Regular Expressions (REs)

- Given an alphabet  $\Sigma$  = set of symbols
- Regular Expressions are built with:
  - $\varepsilon$  - empty string
  - Any symbol from alphabet  $\Sigma$
  - $r_1r_2$  – RE  $r_1$  followed by RE  $r_2$ : concatenation (sometimes we use '.')
  - $r_1 \mid r_2$  – RE  $r_1$  or RE  $r_2$  (OR)
  - $r^*$  - Kleene star:  $\varepsilon \mid r \mid rr \mid \dots$
  - Parenthesis to indicate precedences
    - Priority:  $*$ ,  $.$ ,  $\mid$

# Regular Expressions (Res)

- Generation of the strings of the language represented by an RE:
- Rewrite the RE until we have a sequence of alphabet symbols (string)
  - Different application of the rules can conduct to different results

## General Rules

- 1)  $r_1 | r_2 \rightarrow r_1$
- 2)  $r_1 | r_2 \rightarrow r_2$
- 3)  $r^* \rightarrow rr^*$
- 4)  $r^* \rightarrow \varepsilon$

## Example 1

$(0 | 1)^* \cdot "(0 | 1)^*$   
 $(0 | 1)(0 | 1)^* \cdot "(0 | 1)^*$   
 $1(0 | 1)^* \cdot "(0 | 1)^*$   
 $1" \cdot "(0 | 1)^*$   
 $1" \cdot "(0 | 1)(0 | 1)^*$   
 $1" \cdot "(0 | 1)$   
 $1" \cdot "0$

## Example 2

$(0 | 1)^* \cdot "(0 | 1)^*$   
 $(0 | 1)(0 | 1)^* \cdot "(0 | 1)^*$   
 $0(0 | 1)^* \cdot "(0 | 1)^*$   
 $0" \cdot "(0 | 1)^*$   
 $0" \cdot "(0 | 1)(0 | 1)^*$   
 $0" \cdot "(0 | 1)$   
 $0" \cdot "1$

# Language Generated by a Regular Expression

- Set of all the strings generated by the regular expression is a language of regular expressions
- In general, a language can be infinite
- A String of the language is known as token

# Regular Languages

- $\Sigma = \{0, 1, "."\}$ 
  - $(0 \mid 1)^* "." (0 \mid 1)^*$  - binary numbers with integer and fractional part (representing real numbers)
- $\Sigma = \{0\}$ 
  - $(00)^*$  - sequences of 0's with odd length
- $\Sigma = \{0, 1\}$ 
  - $(1^*01^*01^*)^*$  - strings in the alphabet  $\{0,1\}$  with an odd number of 0's
- $\Sigma = \{a, b, c, 0, 1, 2\}$ 
  - $(a \mid b \mid c)(a \mid b \mid c \mid 0 \mid 1 \mid 2)^*$  - alphanumeric identifiers
- $\Sigma = \{0, 1, 2\}$ 
  - $(0 \mid 1 \mid 2)^*$  - ternary numbers



# Regular Expressions

## ➤ Other constructs:

- $r^+$  - one or more occurrences of  $r$ :  $r \mid rr \mid rrr \dots$ 
  - Equivalent to:  $r.r^*$
- $r^?$  - zero or one occurrence of  $r$ :  $(r \mid \epsilon)$
- $[ ]$  - symbol classes:
  - $[ac]$  is the same as:  $(a \mid c)$
  - $[a-c]$  is the same as:  $(a \mid b \mid c)$

# Regular Expressions

## ➤ Exercises

- Specify the language of the integers
- Specify the language of the identifiers (a letter followed by zero or more letters and numbers)
- Enumerate algebraic properties of regular expressions
- Give examples of languages that cannot be specified by regular expressions

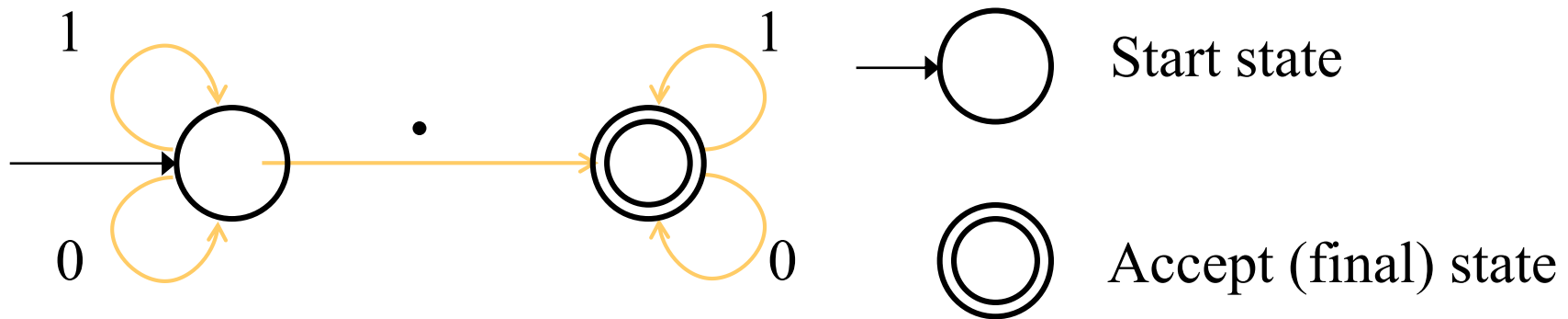
# Finite Automata (FAs)

- Set of states
  - 1 start state
  - 1 or more final states (or accepting states)
- Alphabet of symbols:  $\Sigma$  (it can include the empty string symbol:  $\epsilon$ )
- Transitions between states is triggered by the occurrence of a symbol of the alphabet
- Transitions are labeled with symbols

# Finite Automata (FAs)

➤ Example:

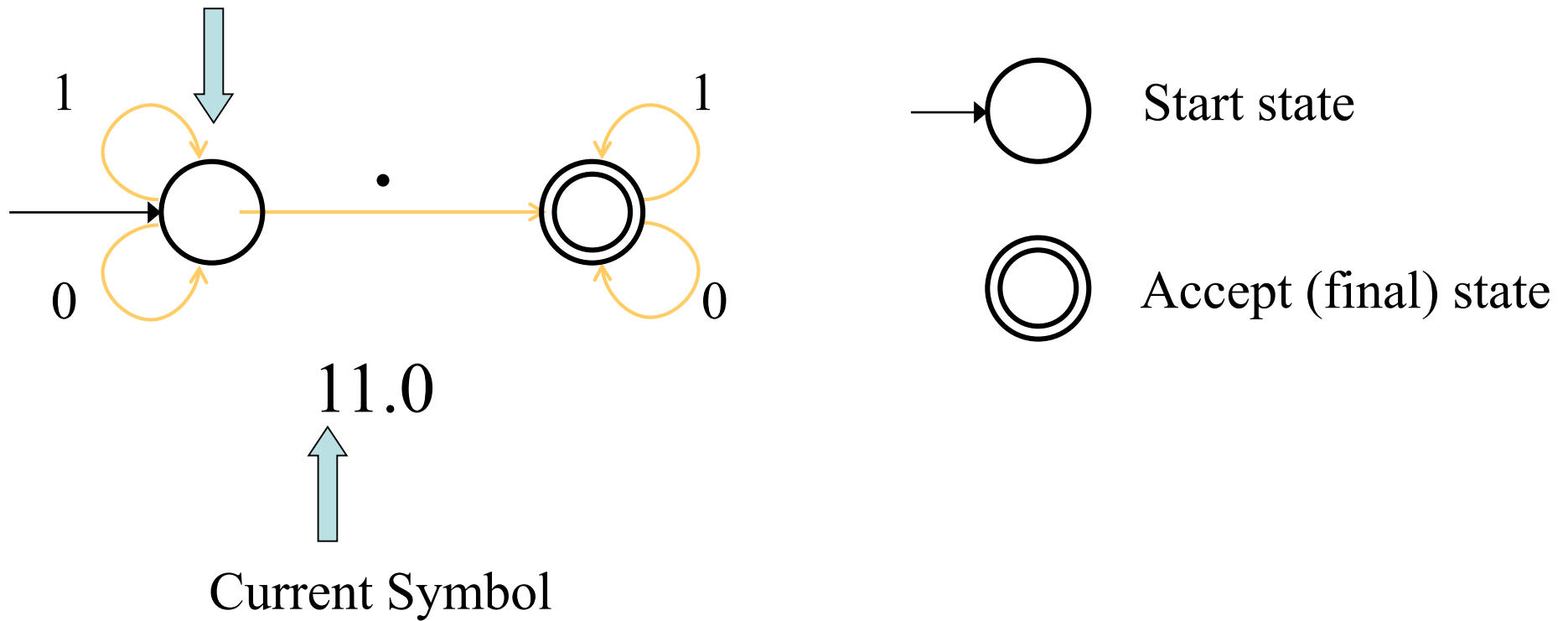
$(0 \mid 1)^* \cdot (0 \mid 1)^*$



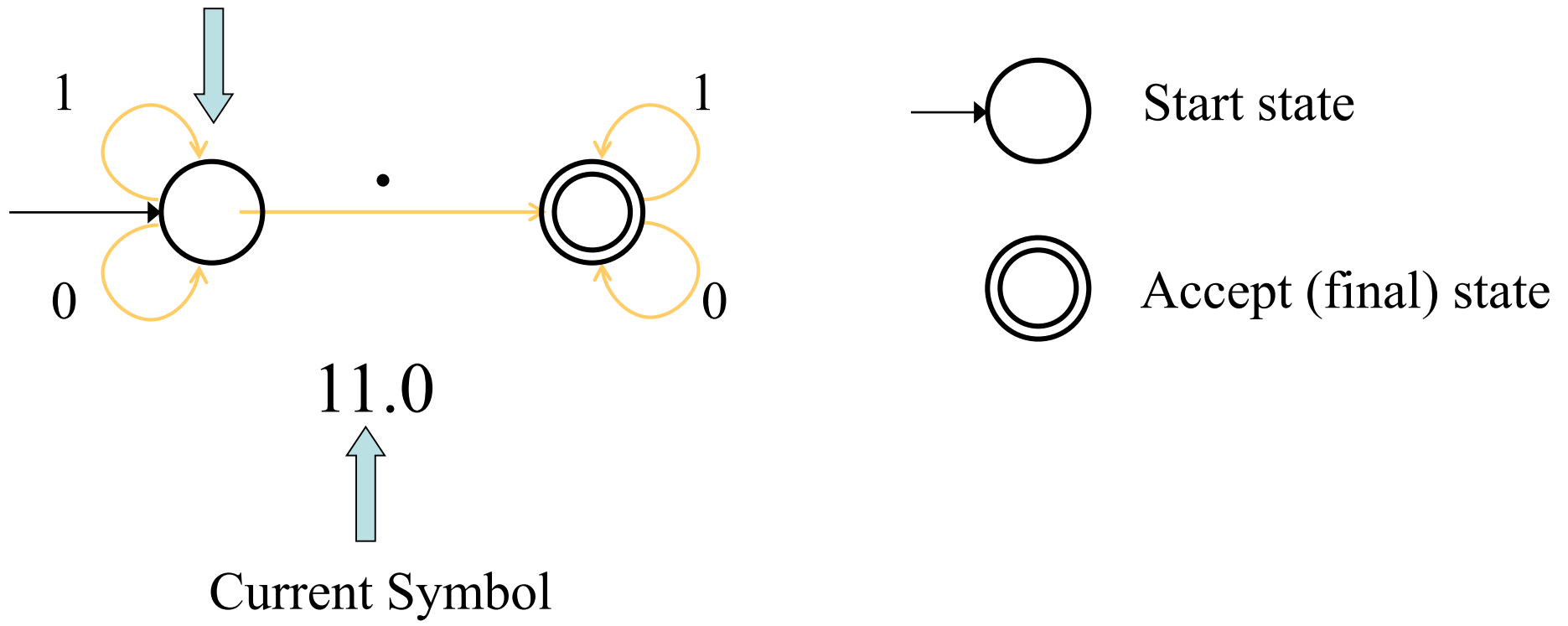
# Accepting the String

- Recognition through the execution of the automaton
  - Start with the start state and with the first symbol of the string
  - Store current state and the current symbol of the string
  - In each step, match the current symbol with the transition labeled with that symbol
  - Continue until the end of the string or until the match fails
  - If the state after processing the last symbol of the string is a final state, then the string is accepted
  - The language of the automaton consists of the string it accepts

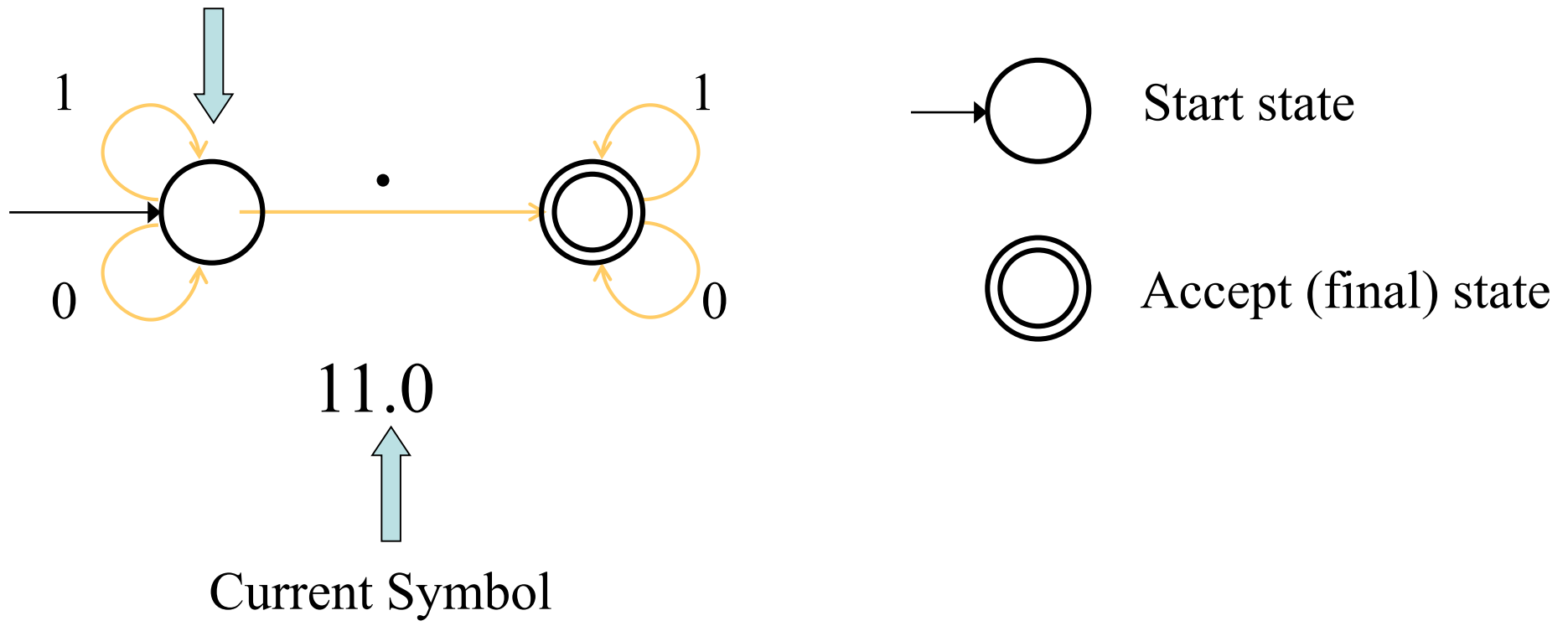
# Example



# Example

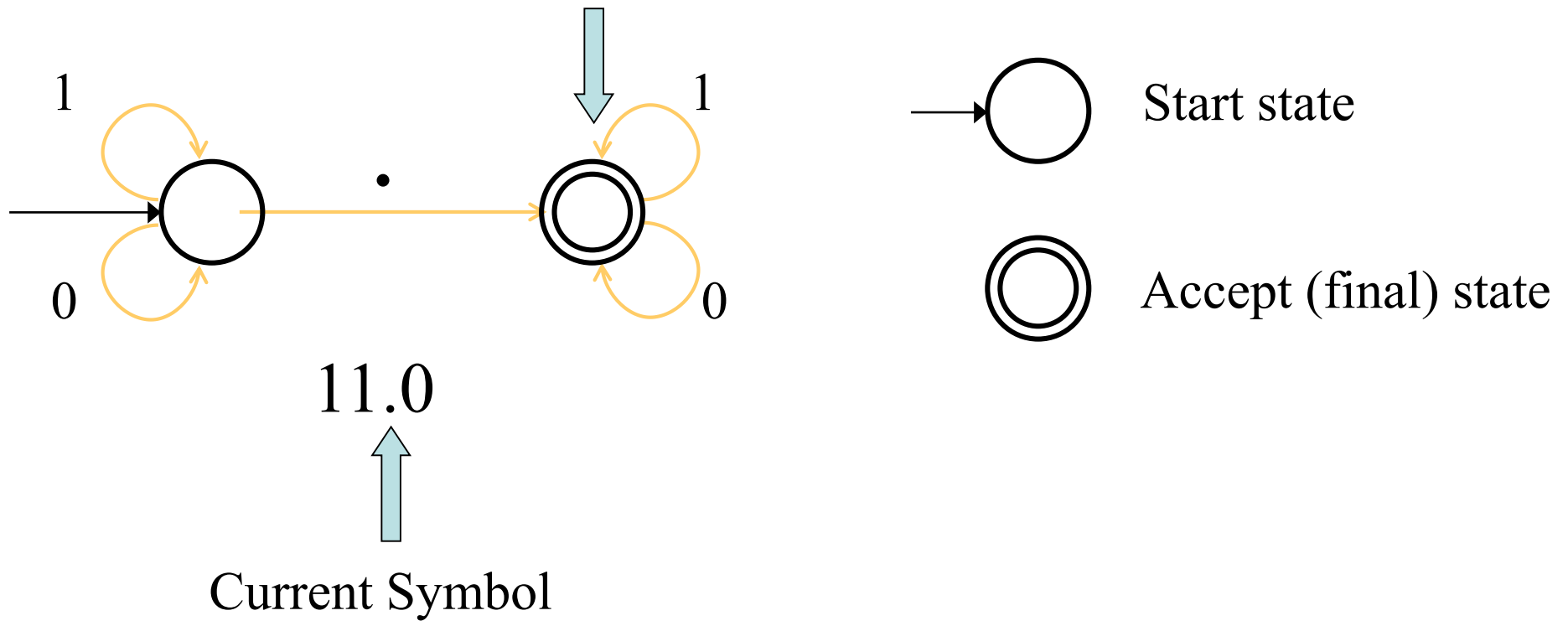


# Example

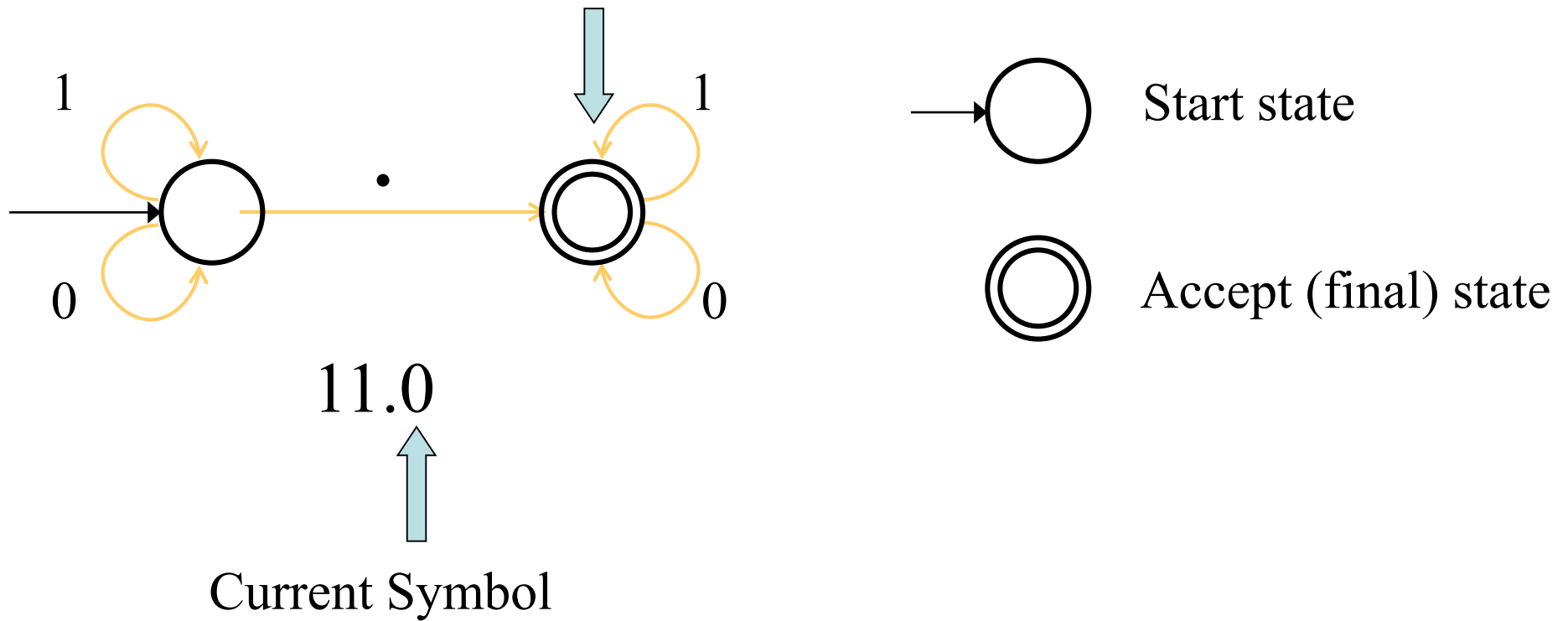




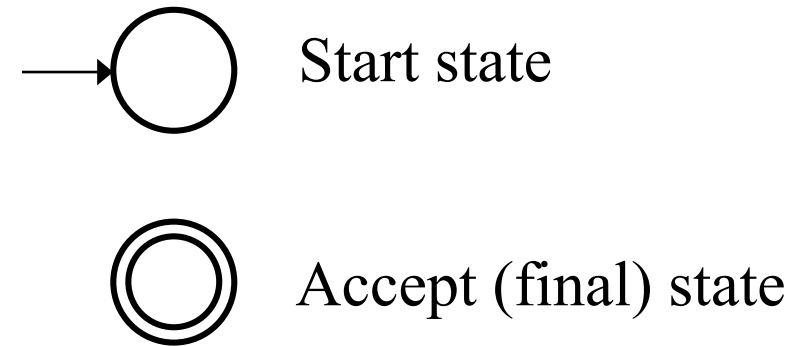
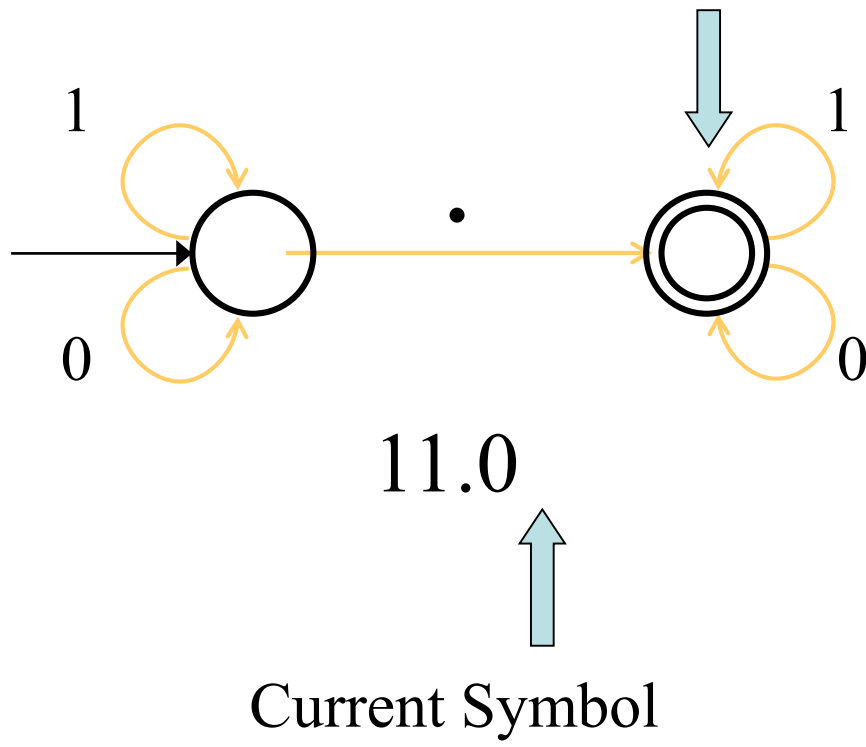
# Example



# Example



# Example



String accepted!

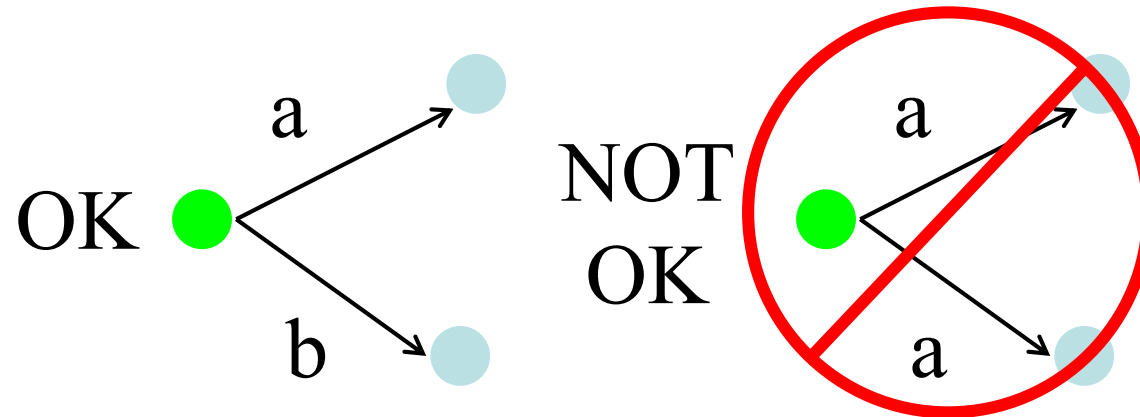
# Finite Automata (FAs)

- NFA: Non Deterministic Finite Automata
  - A state may have more than one output transition labeled with the same symbol (the same occurrence can lead to different states)
- DFA: Deterministic Finite Automata
  - The occurrence of a symbol cannot lead to different states
- NFAs may have  $\varepsilon$  transitions (sometimes these FAs are called  $\varepsilon$ -NFAs)
- In DFAs the input string is always fully processed: the execution of the DFA only finishes after matching the last input symbol

# NFA vs DFA

## ➤ DFA

- Without  $\epsilon$  transitions
- A maximum of one transition from each state for each symbol



## ➤ NFA – none of these restrictions

# Finite Automata (FA)

- Deterministic Finite Automata (DFAs)
  - Faster execution than NFAs, but
  - More complexity of the automaton (usually!)

# Generative vs Recognize

- Regular expressions are a mechanism to generate the strings of a language
- Finite automata (FAs) are a mechanism to recognize if a string belongs to the language
- Standard approach
  - Use regular expressions when defining the language (regular languages), usually the lexemes of a programming language
  - Translation of the regular expressions to FAs to implement the lexical analysis

# From the Regular Expression to the FA

- Translating the regular expression to an FA
  - Construction using induction on the structure
  - Given an arbitrary regular expression **r**,
  - Assume we can convert it to an automaton with
    - one start state
    - one accept state
  - Using the method of Thompson-McNaughton-Yamada (aka Thompson construction)
- Implementation of the FA

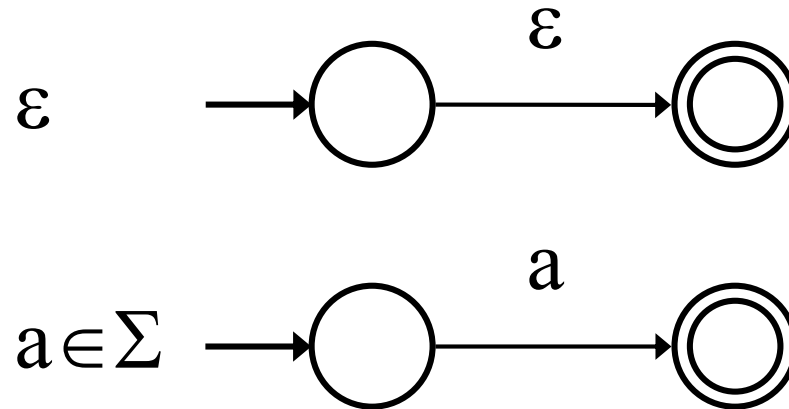


Thompson-McNaughton-Yamada Method

# **FROM REGULAR EXPRESSIONS TO FAS**

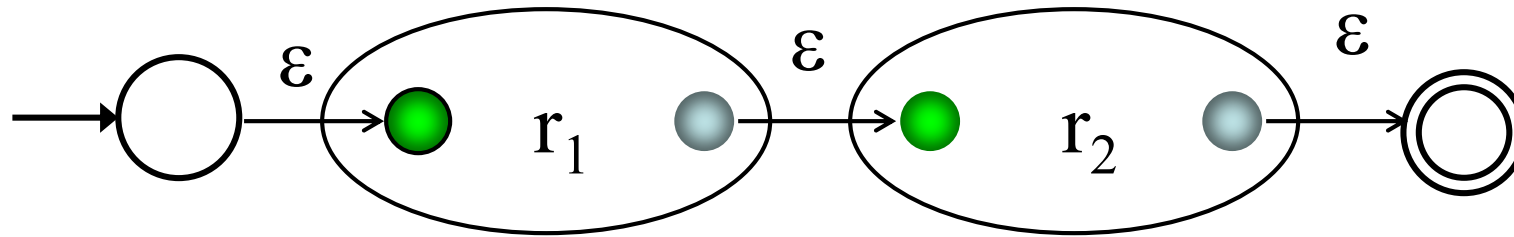
# Basic Constructs

- Empty expression and a symbol



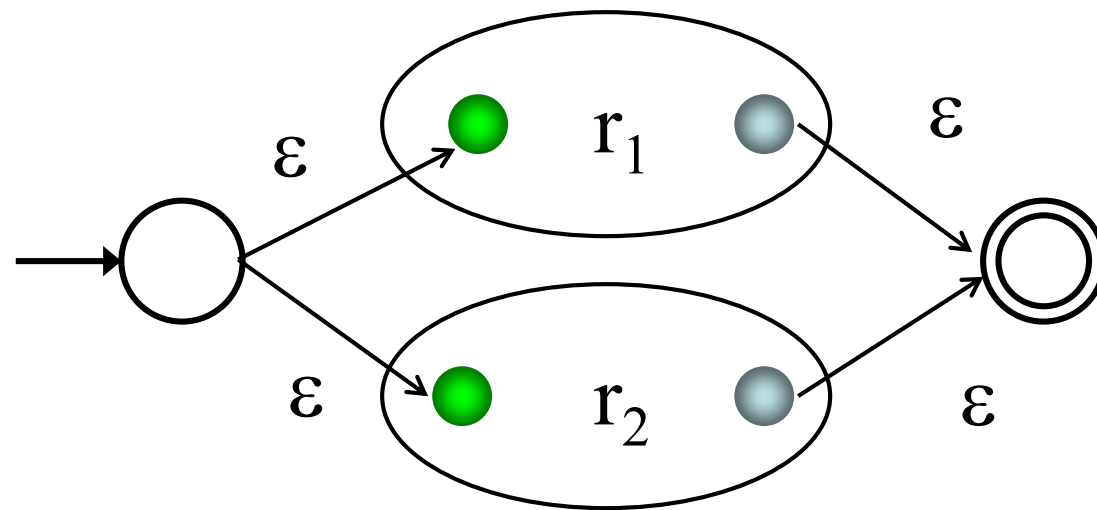
# Concatenation

➤  $r_1.r_2$



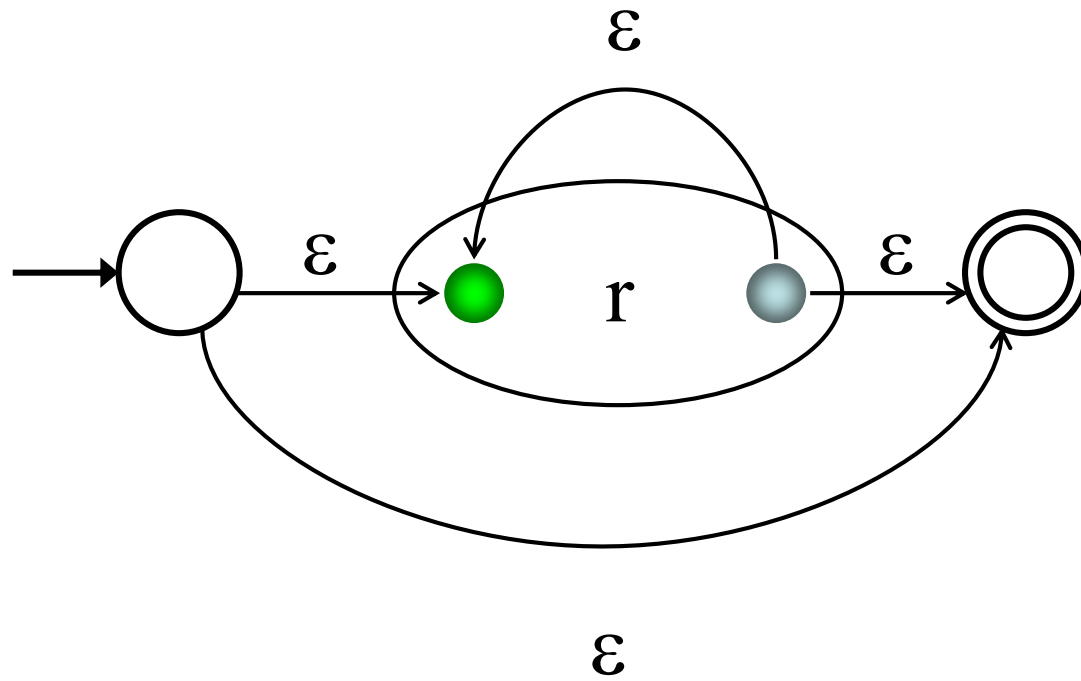
# Union

➤  $r_1 \mid r_2$

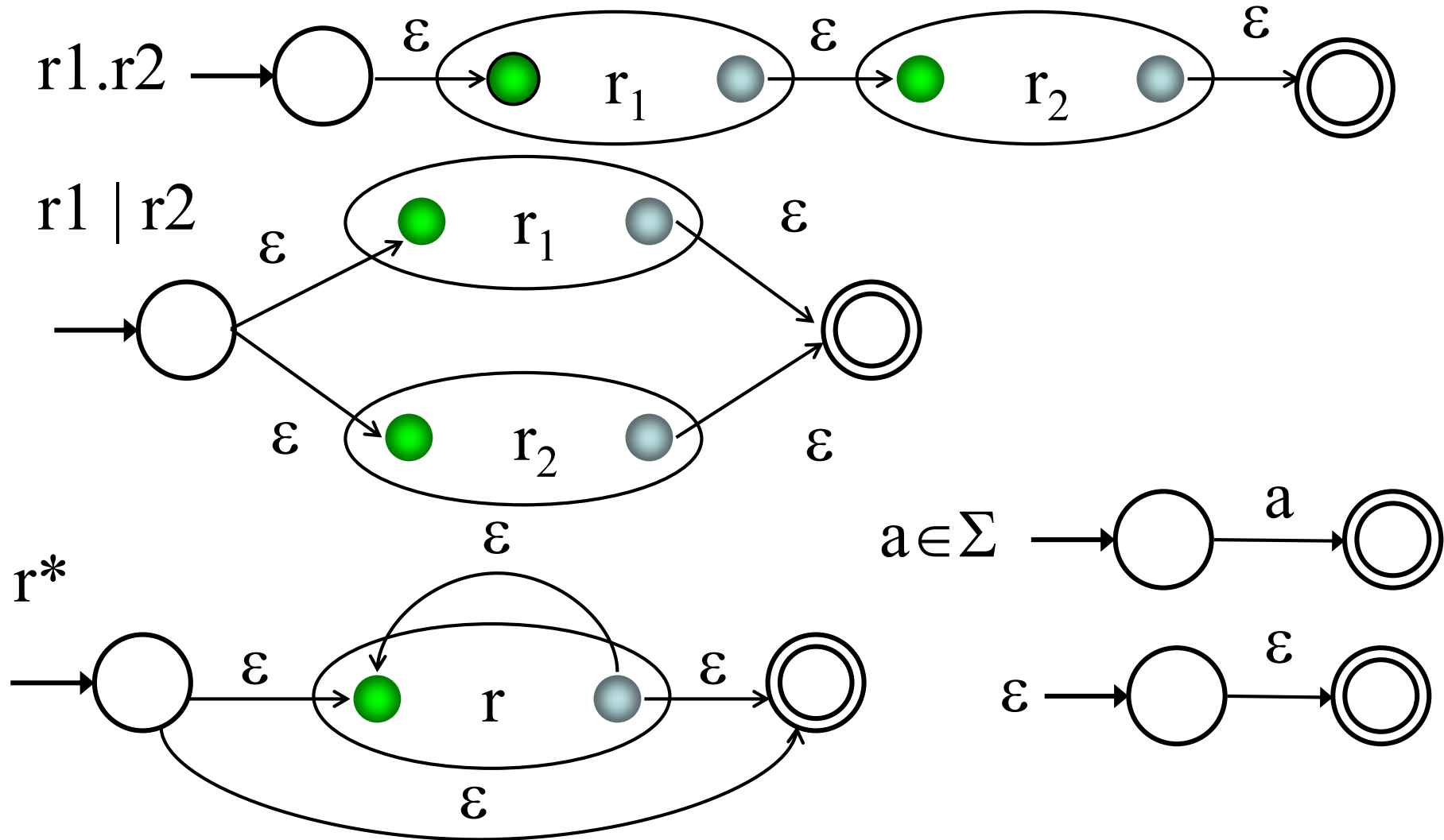


# Kleene Star

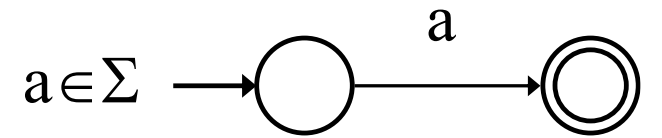
➤  $r^*$



# Conversion Rules

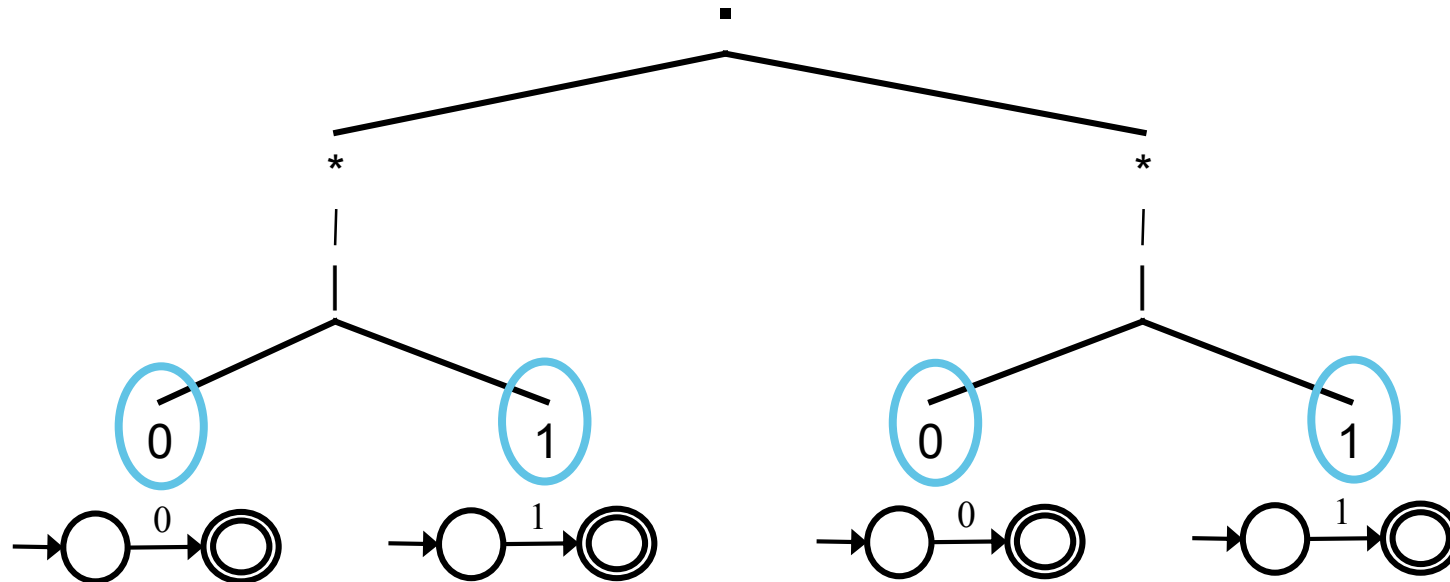


# RE to DFA



➤ Example:  $(0 \mid 1)^*.(0 \mid 1)^*$

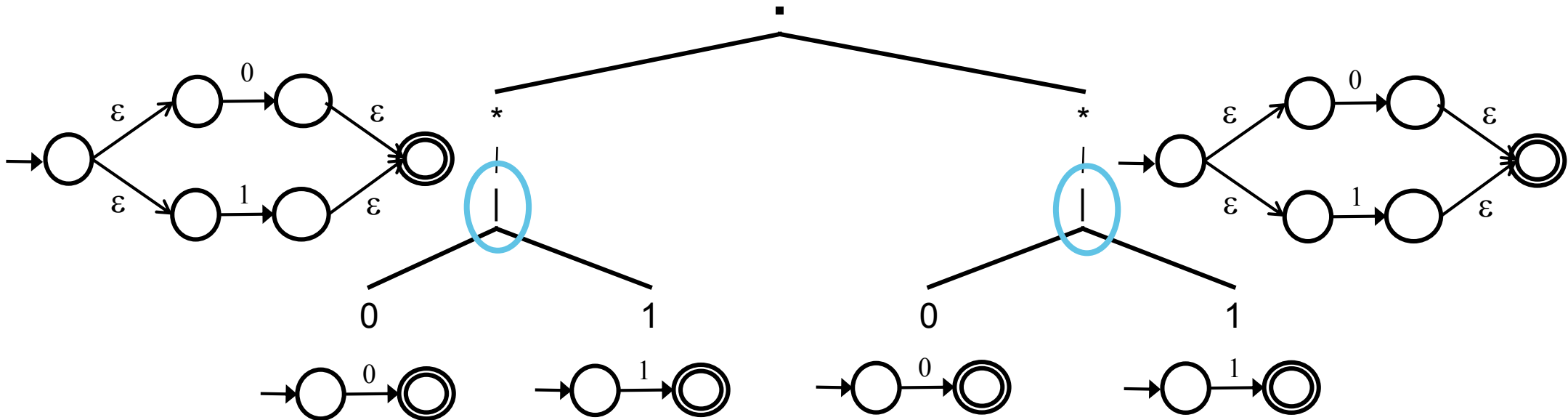
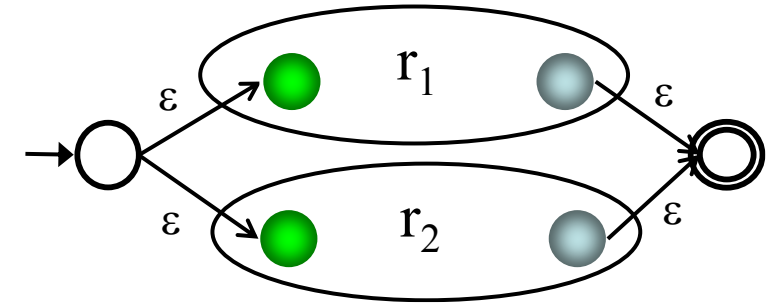
Tree representing the  
input RE



A bottom up approach.

# RE to DFA

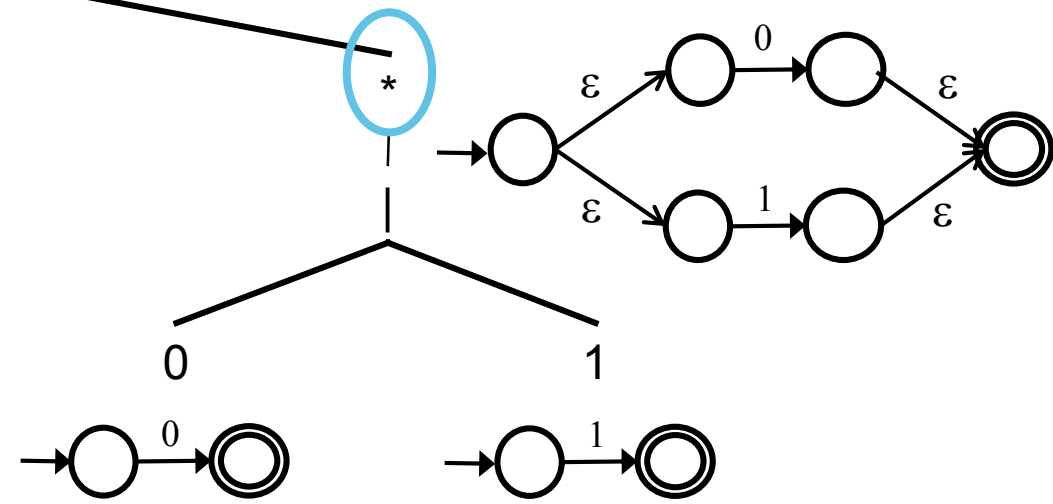
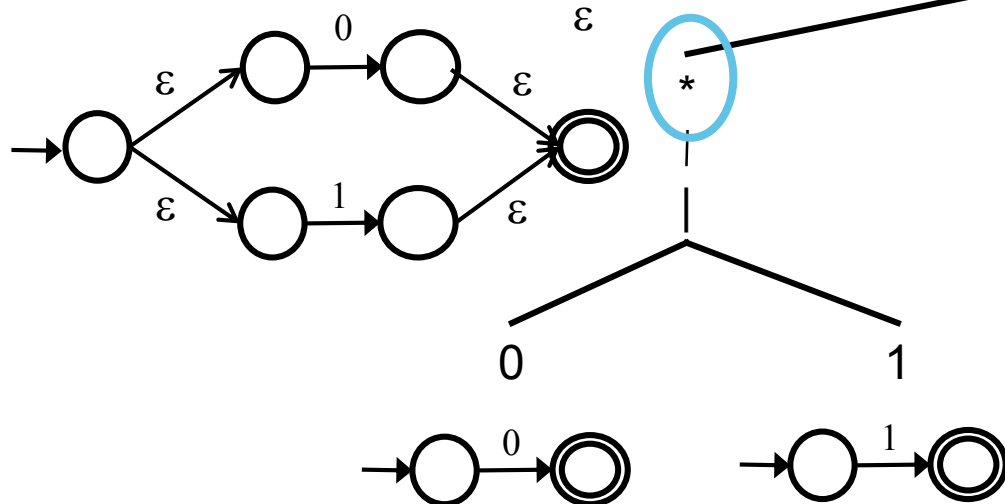
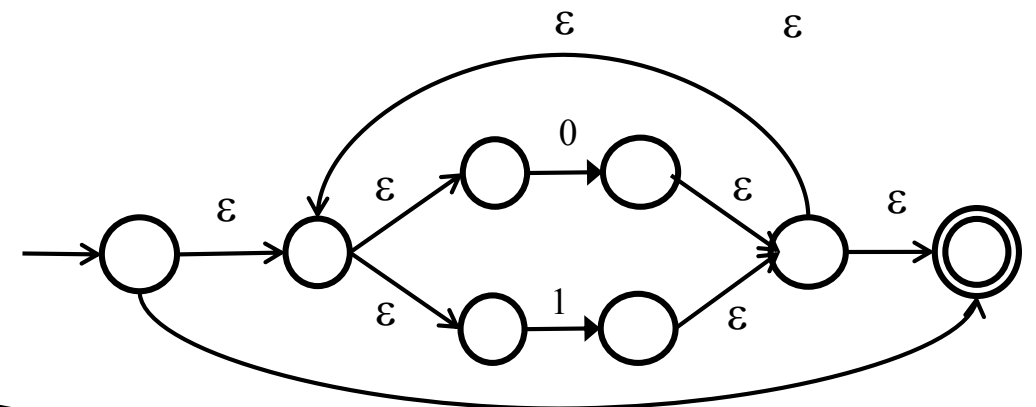
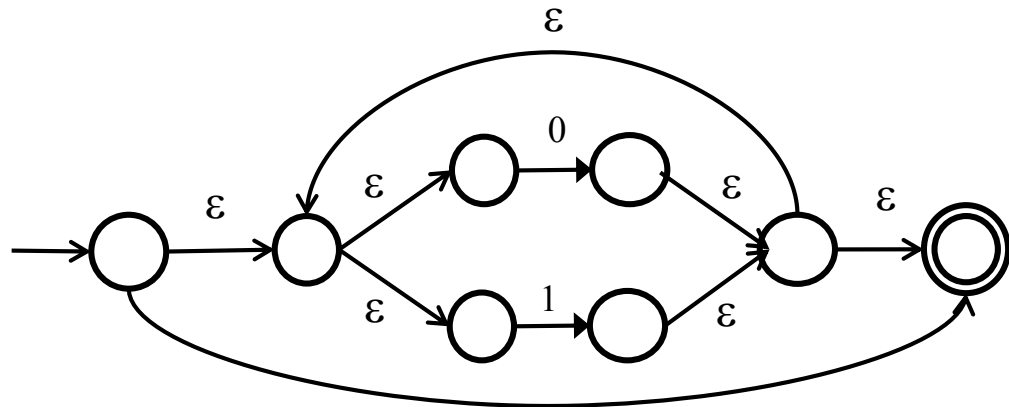
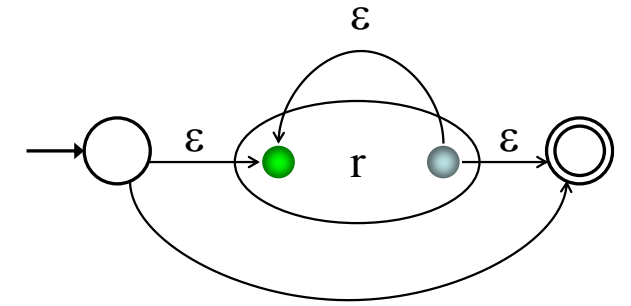
➤ Example:  $(0 \mid 1)^*.(0 \mid 1)^*$





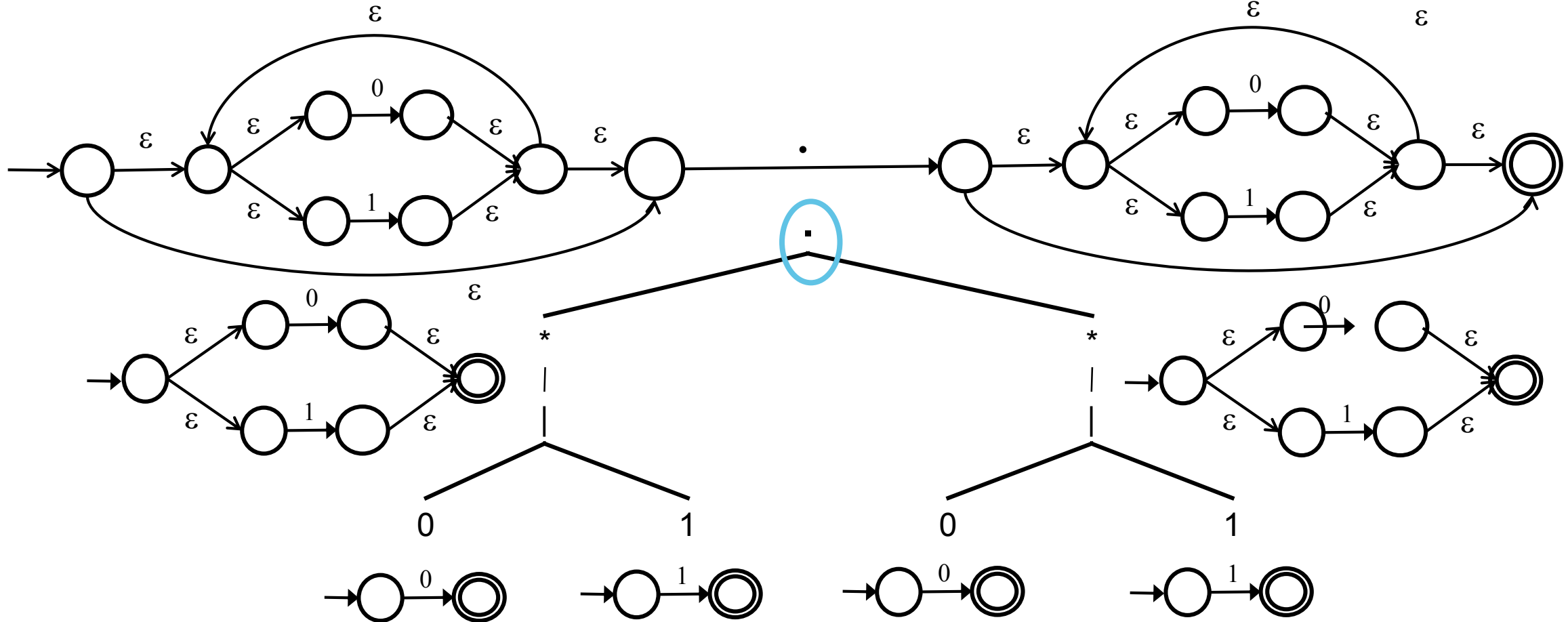
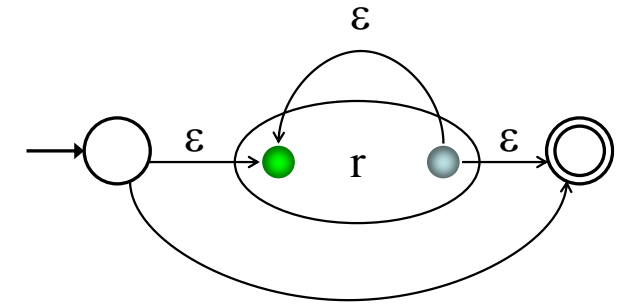
# RE to DFA

➤ Example:  $(0 \mid 1)^*.(0 \mid 1)^*$



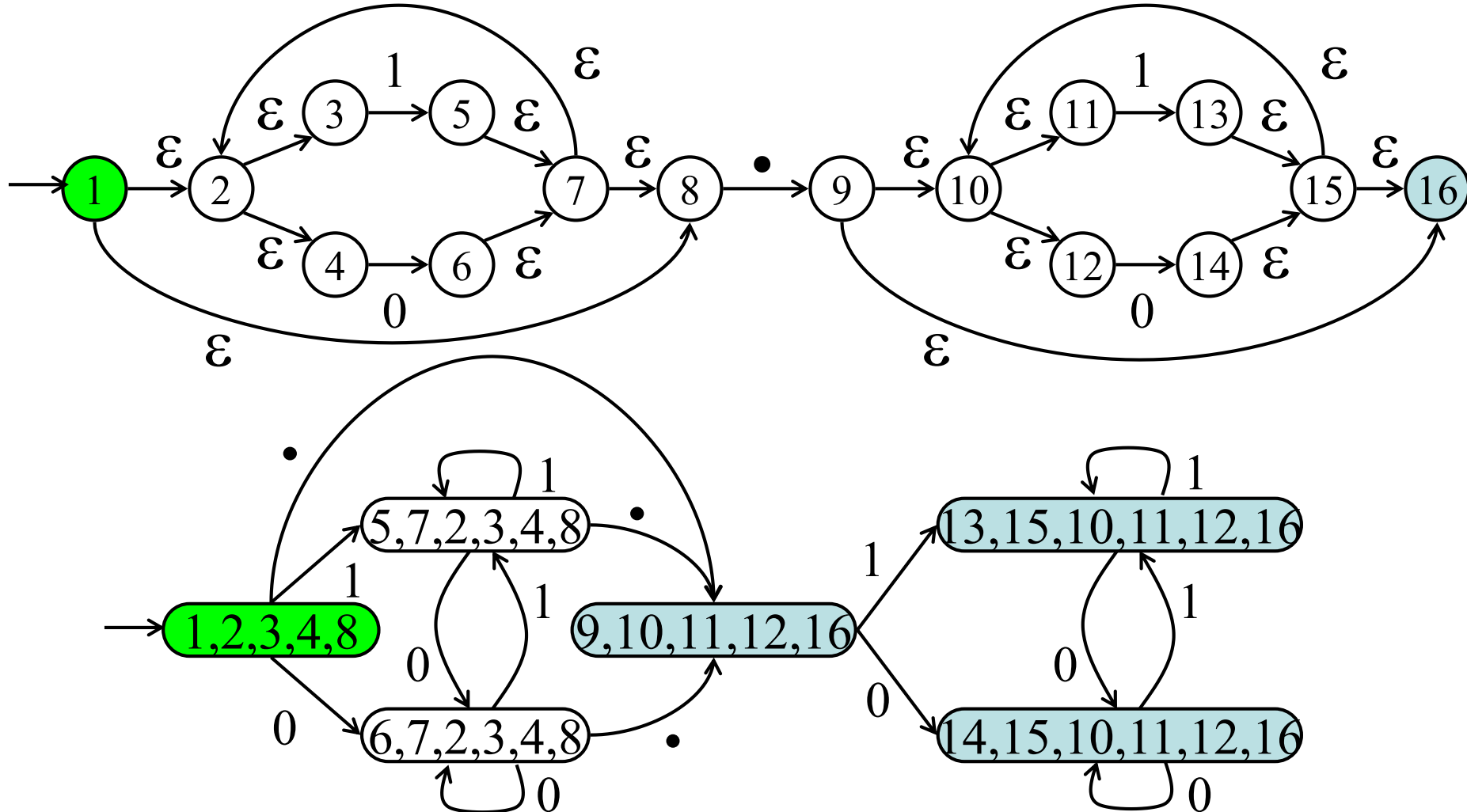
# RE to DFA

➤ Example:  $(0 \mid 1)^* \cdot (0 \mid 1)^*$



# NFA to DFA

➤ Example:  $(0 \mid 1)^*.(0 \mid 1)^*$



# Conversion

- The conversion  $RE \rightarrow FA$  with the previous rules produces an NFA
- The resultant NFA can be automatically transformed into a DFA
  - The DFA can be exponentially larger than the NFA
    - An NFA with  $N$  states can result in a DFA with  $2^N - 1$  states ( $2^N$  states if we count the dead state)
  - Simplification of the DFA involves its minimization

# Conversion NFA to DFA

- The DFA has a state for each subset of states of the NFA
  - The start state of the DFA corresponds to the states reached following the  $\epsilon$  transitions from the NFA start state
  - A state  $q_i$  of the DFA is an accept state if an accept state of the NFA is included in the group of states associated to  $q_i$
- To determine the transition with symbol “a” of a state D of the DFA
  - Consider S an empty state
  - Find a set N of states D in the NFA
    - For all the states of the NFA in N
      - Determine the set of states N' in which the NFA can be after matching “a”
      - Update S with the union of S with N'
  - If S is not empty
    - there is a transition “a” from D to the DFA state which has the set of states S of the NFA
  - Else
    - there is none transition “a” from D

# **IMPLEMENTING THE LEXICAL ANALYZER**

# Lexical Structure of the Programming Languages

- Each language has various categories of classes.
- In a programming language:
  - Keywords (if, while)
  - Arithmetic operations (+, -, \*, /)ord
  - Integer numbers (1, 2, 45, 67)
  - Floating point numbers (1.0, .2, 3.337)
  - Identifiers (abc, i, j, ab345)
- Typically we have a lexical category for each keyword
- Each lexical category is defined by regular expressions

# Examples of Lexical Categories

- If-keyword = if
- While-keyword = while
- Operator = + | - | \* | /
- Integer = [0-9][0-9]\*
- Float = [0-9]\*. [0-9]\*
- Identifier = [a-z]([a-z] | [0-9])\*
- In the syntactic analysis we will use these categories



# From the Regular Expressions to the Lexical Analyzer

- Translation of the regular expression to an NFA
- Translation of the NFA to a DFA
- State minimization of the DFA
- Implementing in software the DFA

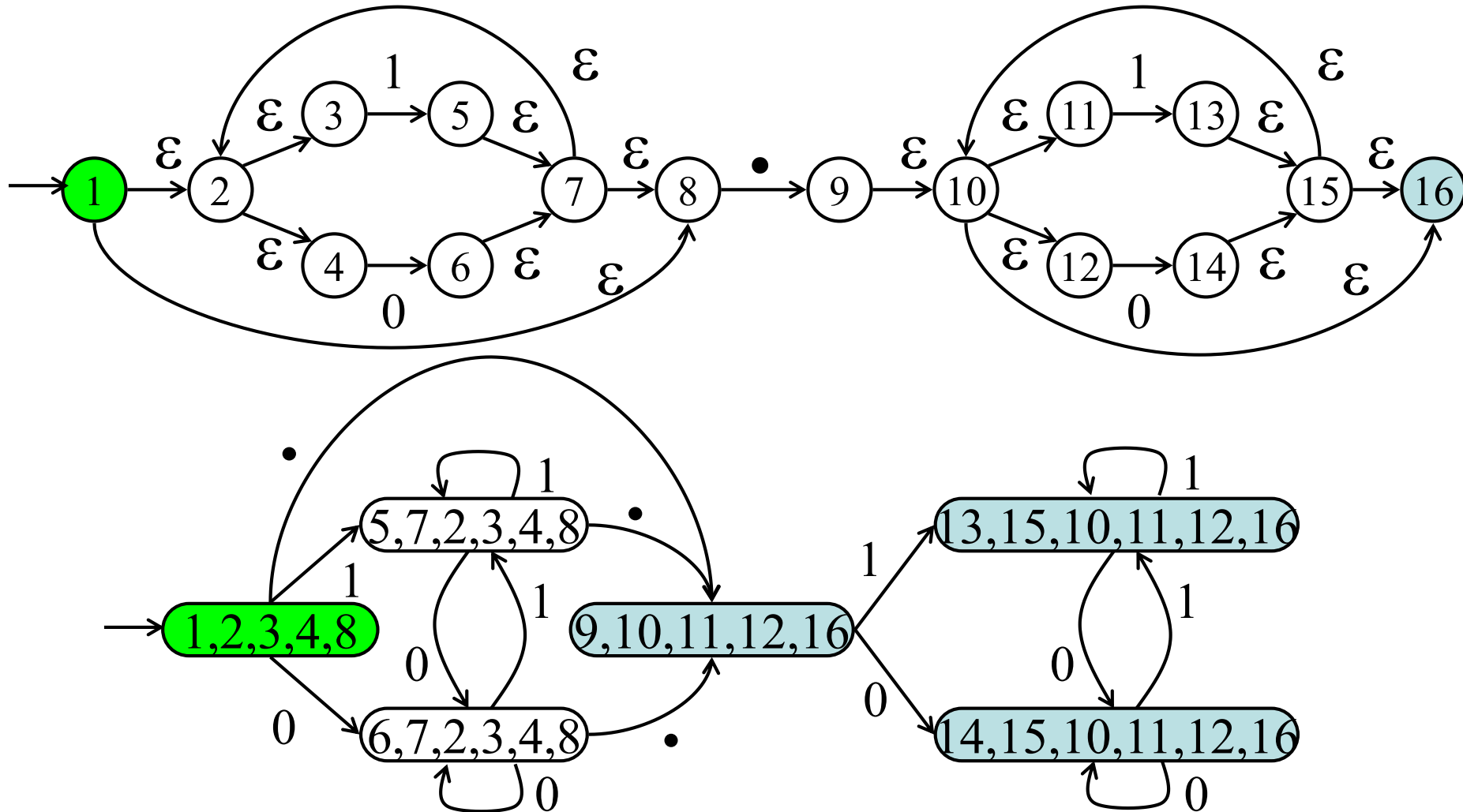
# A DFA Interpreter

## ➤ Pseudo-code of the interpreter

```
State = DFA initial state;  
inputChar = getchar();  
While(inputChar) {  
    State = trans(State, inputChar);  
    inputChar = getchar();  
}  
If(State is an accept state)  
    do action related to accept state (recognize String)  
Else  
    do other action
```

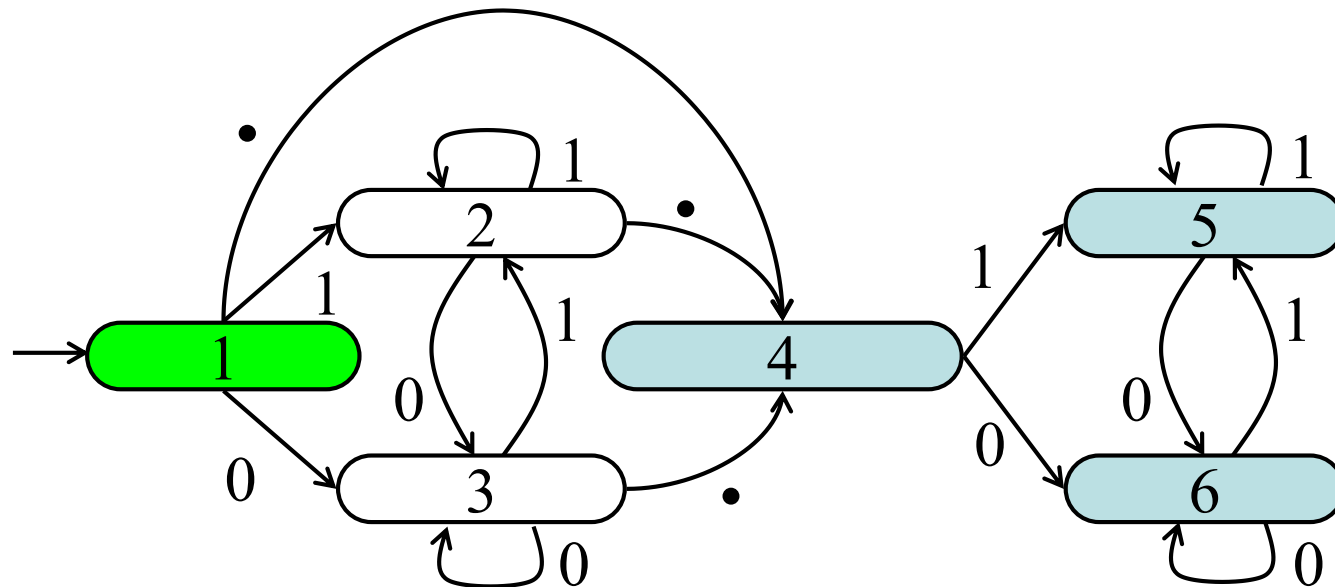
# NFA to DFA

➤ Example:  $(0 \mid 1)^*.(0 \mid 1)^*$



# DFA

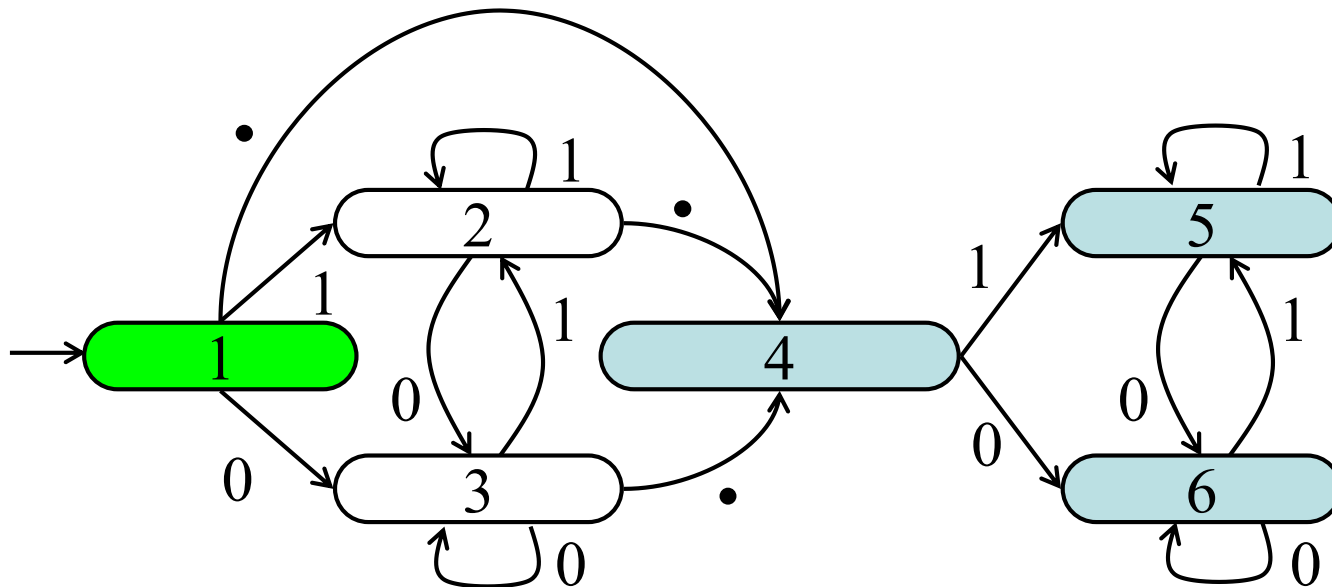
- State transition table (added state 0 – dead state for the transitions not present)



Current state	Next state		
	"0"	"1"	"."
→ 0	0	0	0
1	3	2	4
2	3	2	4
3	3	2	4
*4	6	5	0
*5	6	5	0
*6	6	5	0

# Implementing the DFA

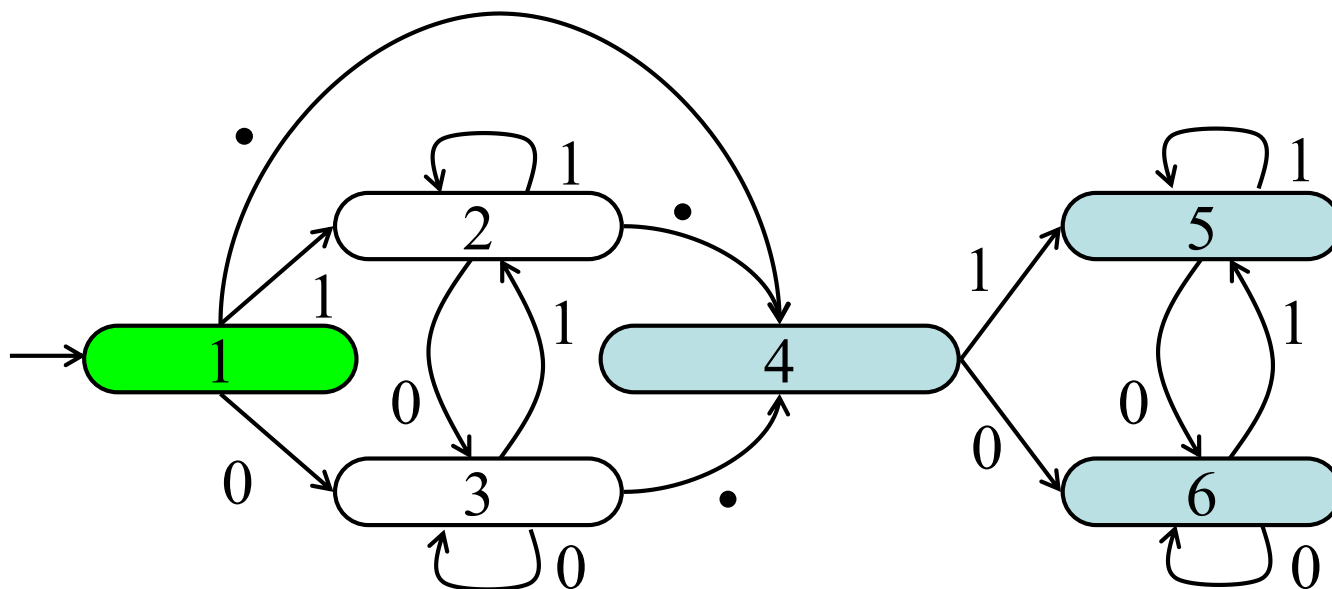
- Implementing the state transition table with a 2D array



Current state	Next state		
	"0"	"1"	"."
→ 0	0	0	0
1	3	2	4
2	3	2	4
3	3	2	4
*4	6	5	0
*5	6	5	0
*6	6	5	0

# Implementing the DFA

- The array will output an index based on the state number and symbol
- Supposing the possibility of 256 symbols:
  - `int Edge[NumStates][256]` (NumStates = 7)



Current state	Next state		
	"0"	"1"	"."
→ 0	0	0	0
1	3	2	4
2	3	2	4
3	3	2	4
*4	6	5	0
*5	6	5	0
*6	6	5	0

# Implementing the DFA

```
int Edge[NumStates][256] =
{
    /*... 0 1 2 ... 9 ... "." ... */
    /* estado 0 */ {..., 0, 0, 0, ..., 0, ..., 0, ...},
    /* estado 1 */ {..., 3, 2, 0, ..., 0, ..., 4, ...},
    /* estado 2 */ {..., 3, 2, 0, ..., 0, ..., 4, ...},
    /* estado 3 */ {..., 3, 2, 0, ..., 0, ..., 4, ...},
    /* estado 4 */ {..., 6, 5, 0, ..., 0, ..., 0, ...},
    /* estado 5 */ {..., 6, 5, 0, ..., 0, ..., 0, ...},
    /* estado 6 */ {..., 6, 5, 0, ..., 0, ..., 0, ...}
}
```

Example:

Edge[3][(int) '.']  $\leftarrow$  4

Current state	Next state		
	"0"	"1"	"."
$\rightarrow 0$	0	0	0
1	3	2	4
2	3	2	4
3	3	2	4
*4	6	5	0
*5	6	5	0
*6	6	5	0

# Implementing the DFA

- A 1D array translates the state number into the action number to do:

```
// state          0 1 2 3 4 5 6
int final[NumStates] = {0, 0, 0, 0, 1, 1, 1}
```

Example:

$\text{final}[4] \leftarrow 1$  (thus, action 1 must be done)

Current state	Next state		
	"0"	"1"	"."
→ 0	0	0	0
1	3	2	4
2	3	2	4
3	3	2	4
*4	6	5	0
*5	6	5	0
*6	6	5	0



# Implementing the DFA

## ➤ Pseudo-code of the interpreter

```
State = 1; // DFA initial state
inputChar = input.read();
While(inputChar) {
    State = edge[State][inputChar];
    inputChar = input.read();
}
If(final[State] == 1) // 1 identifies na action
    action (recognize String)
Else
    other action
```

# Optimizations of the DFA Implementation

- Table of transitions implemented by a 2D array can be further optimized
- Indirect mapping:

```
int Edge[NumStates][4] = {  
    /* estado 0 */   {0, 0, 0, 0},  
    /* estado 1 */   {3, 2, 4, 0},  
    /* estado 2 */   {3, 2, 4, 0},  
    /* estado 3 */   {3, 2, 4, 0},  
    /* estado 4 */   {6, 5, 0, 0},  
    /* estado 5 */   {6, 5, 0, 0},  
    /* estado 6 */   {6, 5, 0, 0}  
}
```

```
/*... "0" "1" "2" "3"... 9 ... "." ... */
```

```
Int map[256] = { ..., 0, 1, 3, 3, ... 3, ... 2, ... }
```

Exemplo:

Edge[3][(int) '.'] is translated to:

Edge[3][map[(int) '.']]

Another possibility is to use if or switch constructs for the map function:

```
Switch(ch1) {  
    case '0': return 0; break;  
    case '1': return 1; break;  
    case '.': return 2; break;  
    otherwise: return 3;  
}
```

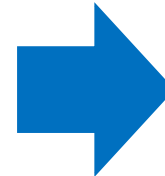
# Optimizations of the DFA Implementation

- For the presented example and using tables:
  - Optimization allows to go from an array with  $256 \times 7$  (1,792) elements to an array with 256 and other with  $7 \times 4$  (28) elements
  - From 1,792 to 284 elements
- There are other optimization techniques...

# With JavaCC

- The Lexical Analyzer is not implemented using tables
- Instead, code using switch and if-else constructs is generated
- Use of tables might be better when developing a lexical analyzer manually (easy to understand, change, and maintain)

```
ex1.jj
...
TOKEN: {
    < REALBIN: ("0"|"1")*"."("0"|"1")*>
}
...
```



```
ex1TokenManager.java:
...
switch(jjstateSet[--i])
{
    case 3:
        if ((0x300000000000000L & l) != 0L)
            { jjCheckNAddTwoStates(0, 1); }
        else if (curChar == 46)
        {
            if (kind > 1)
                kind = 1;
            { jjCheckNAdd(2); }
        }
        break;
    case 0:
        if ((0x300000000000000L & l) != 0L)
            { jjCheckNAddTwoStates(0, 1); }
        break;
    case 1:
        if (curChar != 46)
            break;
        kind = 1;
        { jjCheckNAdd(2); }
        break;
    case 2:
        if ((0x300000000000000L & l) == 0L)
            break;
        if (kind > 1)
            kind = 1;
        { jjCheckNAdd(2); }
        break;
    default : break;
}
...
```

# Summary

- Lexemes/strings of the language are specified using regular expressions (REs)
  - They are grouped in categories
  - Note that each token must be identified and its value may need to be stored
- Lexical analysis can be efficiently implemented using Finite Automata

# Further Reading

- About regular expressions:
  - Jeffrey E.F. Friedl, [Mastering Regular Expressions, Third Edition](#), O'Reilly Media, Inc., Aug. 2006.
  - PERL Compatible Regular Expression (PCRE): <http://www.pcre.org/>
- Programming languages may have built-in support to regular expressions or may provide libraries/APIs:
  - Regular Expressions in Java: `java.util.regex`