

# CMPT 379

## Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

# Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
- Example:

```
double f = sqrt(-1);
```

T_DOUBLE	("double")
T_IDENT	("f")
T_OP	("=")
T_IDENT	("sqrt")
T_LPAREN	(" (")
T_OP	(" -")
T_INTCONSTANT	("1")
T_RPAREN	(" )")
T_SEP	(" ;")

# Token Attributes

- Some tokens have attributes
  - T\_IDENT “sqrt”
  - T\_INTCONSTANT 1
- Other tokens do not
  - T\_WHILE
- *Token*=T\_IDENT, *Lexeme*="sqrt", *Pattern*
- Source code location for error reports

# Lexical errors

- What if user omits the space in “doublef”?
  - No lexical error, single token  
T\_IDENT(“doublef”) is produced instead of  
sequence T\_DOUBLE, T\_IDENT(“f”)!
- Typically few lexical error types
  - E.g., illegal chars, opened string constants or  
comments that are not closed

# Implementing Lexers: Loop and switch scanners

- Ad hoc scanners
- Big nested switch/case statements
- Lots of `getc()/ungetc()` calls
  - Buffering
- Can be error-prone, use only if
  - Your language's lexical structure is simple
  - Tools don't do what you want
- Changing or adding a keyword is problematic
- Key idea: separate the defn from the implementation

# Formal Languages: Recap

- Symbols:  $a, b, c$
- Alphabet  $:=$  finite set of symbols  $\Sigma = \{a, b\}$
- String: sequence of symbols  $bab$
- Empty string:  $\epsilon$
- Set of all strings:  $\Sigma^*$

# Regular Expressions: Definition

- Every symbol of  $\Sigma \cup \{ \varepsilon \}$  is a regular expression
- If  $r_1$  and  $r_2$  are regular expressions, so are
  - Concatenation:  $r_1 r_2$
  - Alternation:  $r_1 | r_2$
  - Repetition:  $r_1^*$
- Nothing else is.
  - Grouping re's: e.g.  $aalbc$  vs.  $((aa)lb)c$

# Regular Expressions: Examples

- Alphabet  $\{ 0, 1 \}$
- All strings that represent binary numbers divisible by 4 (but accept 0)  $((0|1)^*00)|0$
- All strings that do not contain “01” as a substring  $1^*0^*$



# Regular Expressions

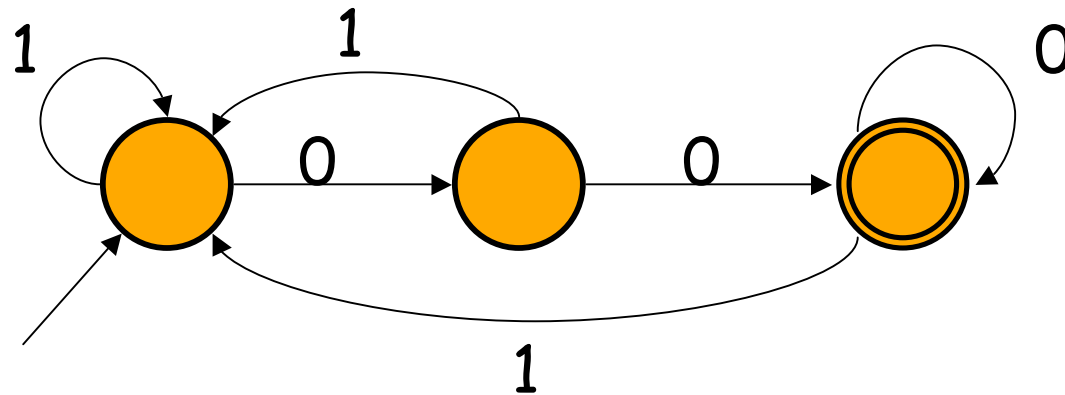
- To describe all lexemes that form a token as a *pattern*
  - $(0|1|2|3|4|5|6|7|8|9)^+$
- Need decision procedure: to which token does a given sequence of characters belong (if any)?
  - Finite State Automata

# Finite Automata: Recap

- A set of states  $S$ 
  - One start state  $q_0$ , zero or more final states  $F$
- An alphabet  $\Sigma$  of input symbols
- A transition function:
  - $\delta: S \times \Sigma \Rightarrow S$
- Example:  $\delta(1, a) = 2$

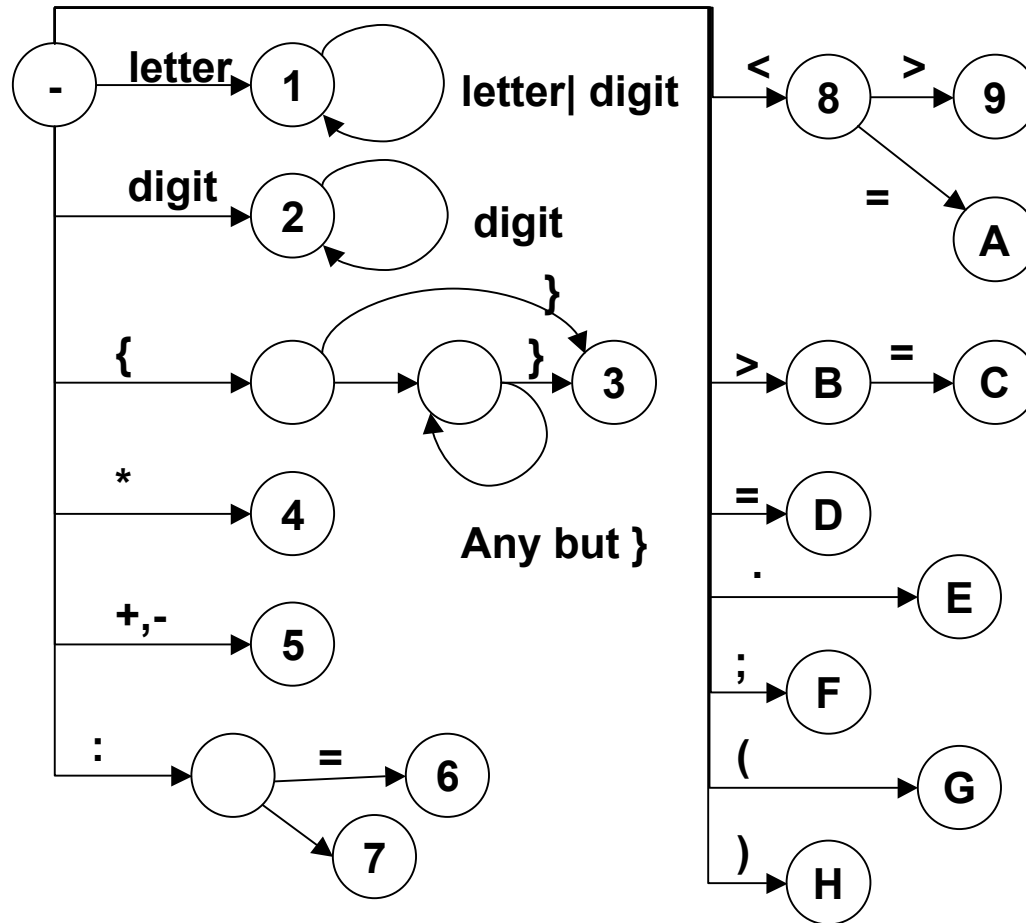
# Finite Automata: Example

- What regular expression does this automaton accept?



Answer:  $(0|1)^*00$

# FA: Pascal Example



# Building a Lexical Analyzer

- Token  $\Rightarrow$  Pattern
- Pattern  $\Rightarrow$  Regular Expression
- Regular Expression  $\Rightarrow$  NFA
- NFA  $\Rightarrow$  DFA
- DFA  $\Rightarrow$  Lexical Analyzer

# NFAs

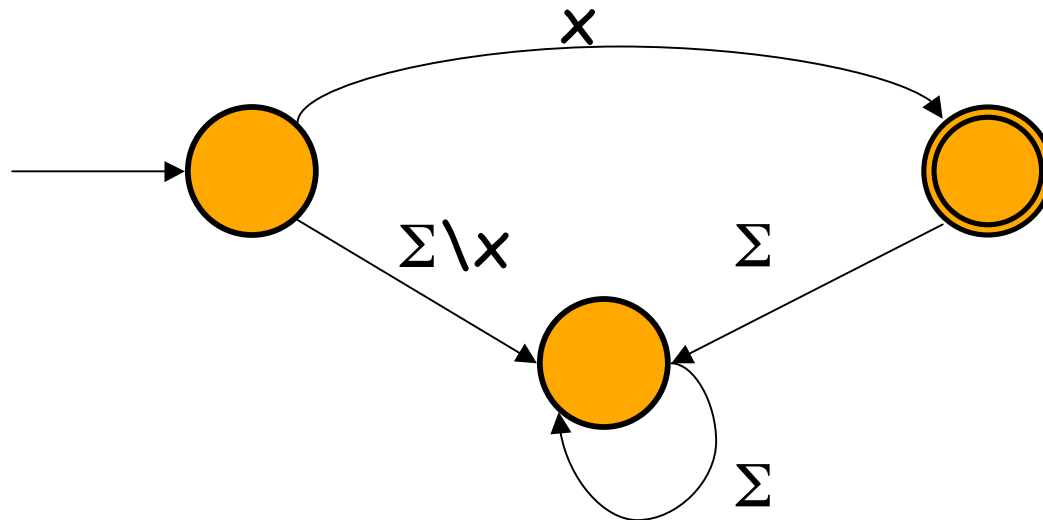
- NFA: like a DFA, except
  - A transition can lead to more than one state, that is,  $\delta: S \times \Sigma \Rightarrow 2^S$
  - One state is chosen non-deterministically
  - Transitions can be labeled with  $\epsilon$ , meaning states can be reached without reading any input, that is,  
$$\delta: S \times \Sigma \cup \{ \epsilon \} \Rightarrow 2^S$$

# Thompson's construction

- Converts regexps to NFA
- Five simple rules
  - Symbols
  - Empty String
  - Alternation ( $r_1$  or  $r_2$ )
  - Concatenation ( $r_1$  followed by  $r_2$ )
  - Repetition ( $r_1^*$ )

# Thompson Rule 1

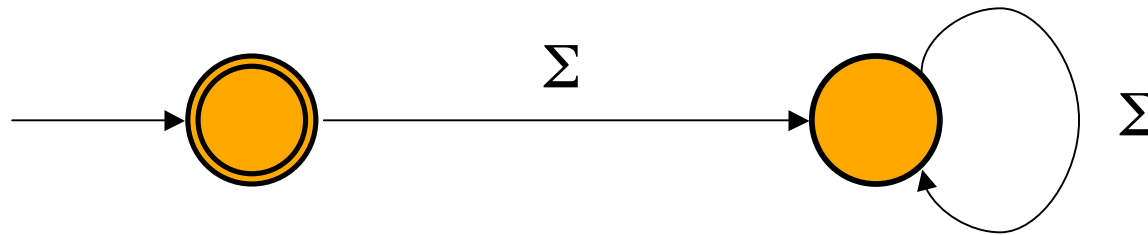
- For each symbol  $x$  of the alphabet, there is a NFA that accepts it (include a *sinkhole* state)





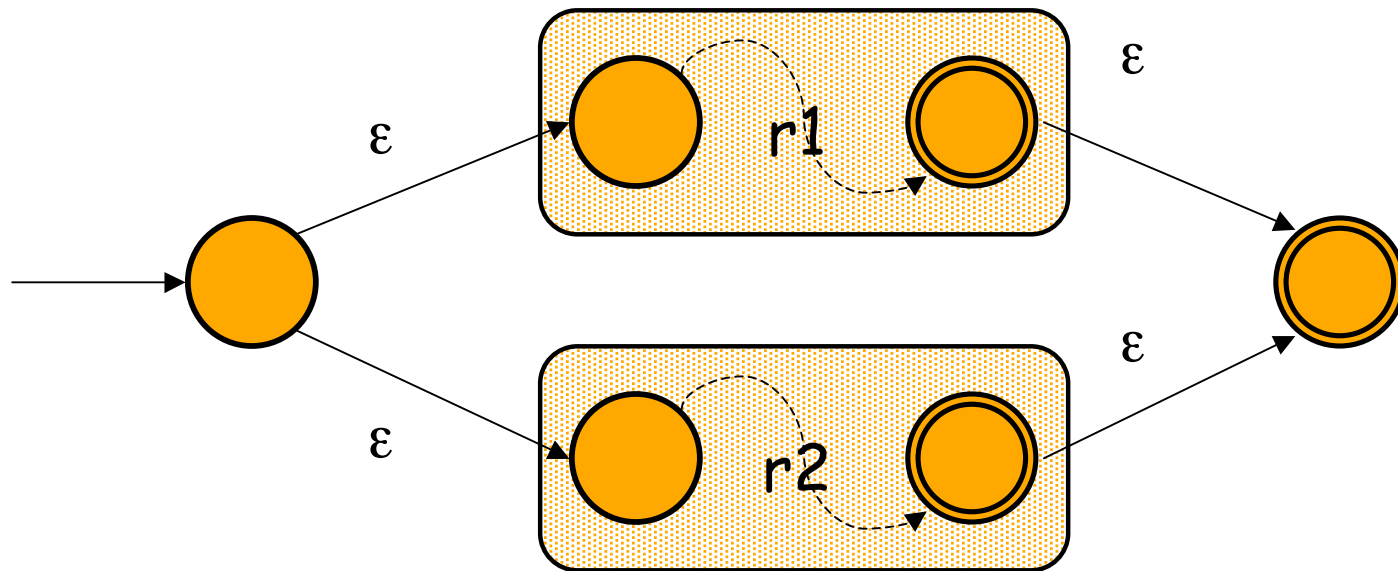
# Thompson Rule 2

- There is an NFA that accepts only  $\varepsilon$



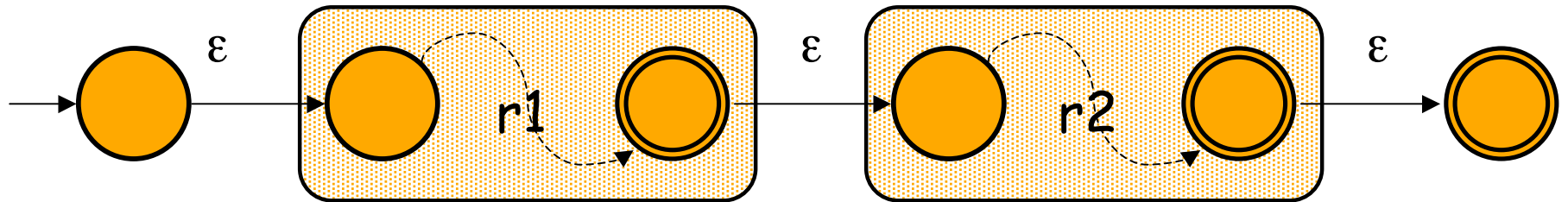
# Thompson Rule 3

- Given two NFAs for  $r_1$ ,  $r_2$ , there is a NFA that accepts  $r_1|r_2$



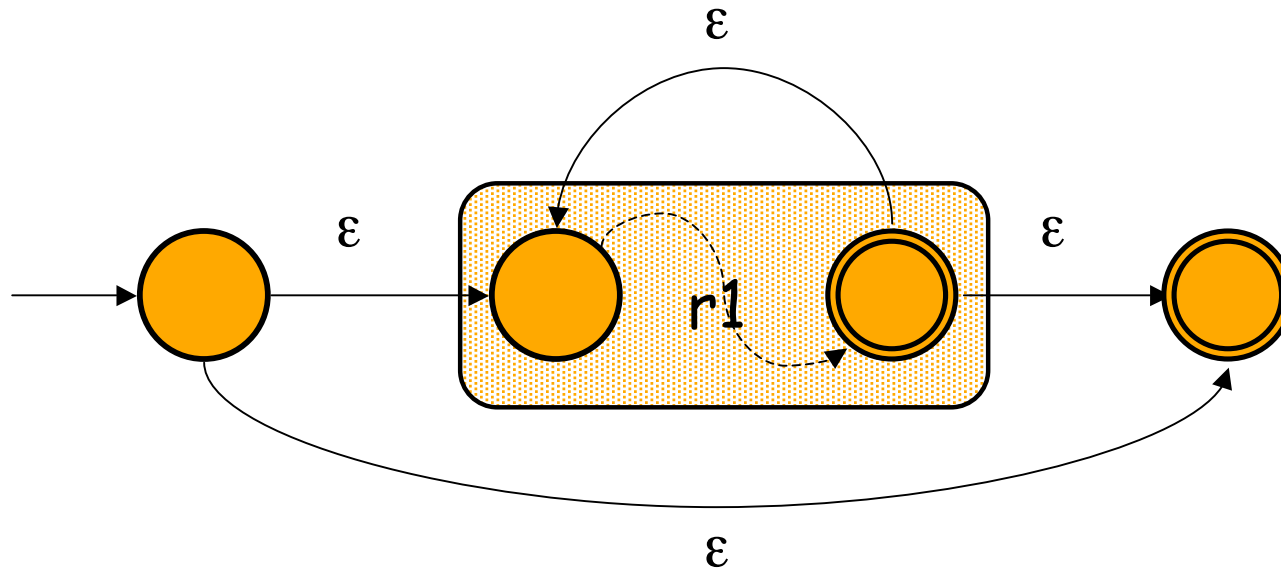
# Thompson Rule 4

- Given two NFAs for  $r_1$ ,  $r_2$ , there is a NFA that accepts  $r_1 r_2$



# Thompson Rule 5

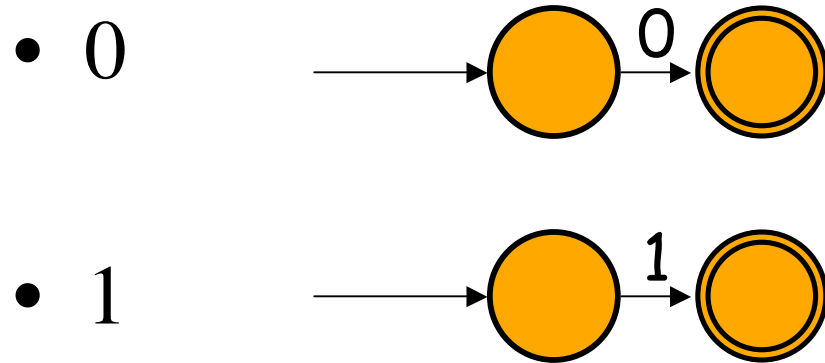
- Given a NFA for  $r_1$ , there is an NFA that accepts  $r_1^*$



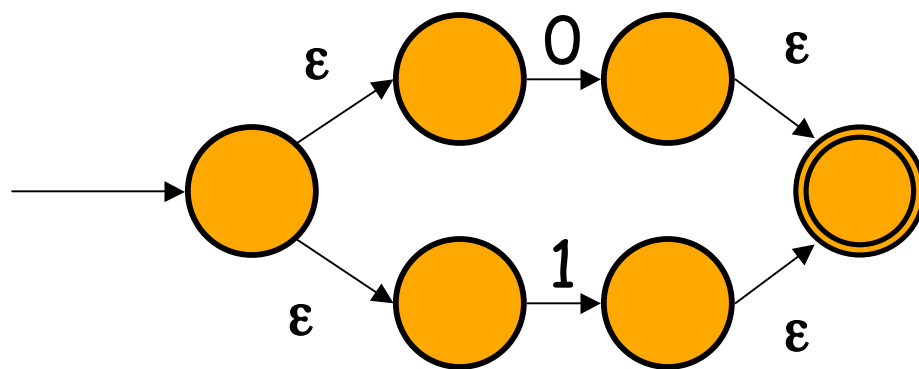
# Example

- Set of all binary strings that are divisible by four (include 0 in this set)
- Defined by the regexp:  $((0|1)^*00) | 0$
- Apply Thompson's Rules to create an NFA

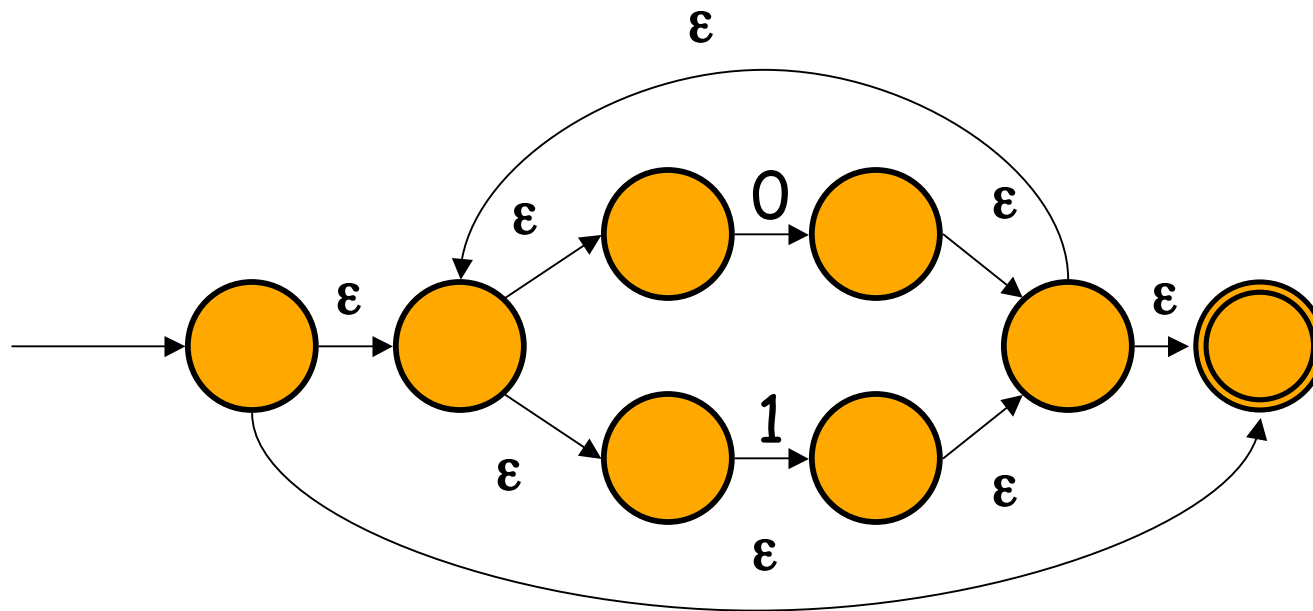
# Basic Blocks 0 and 1



(this version does not report errors: no *sinkholes*)

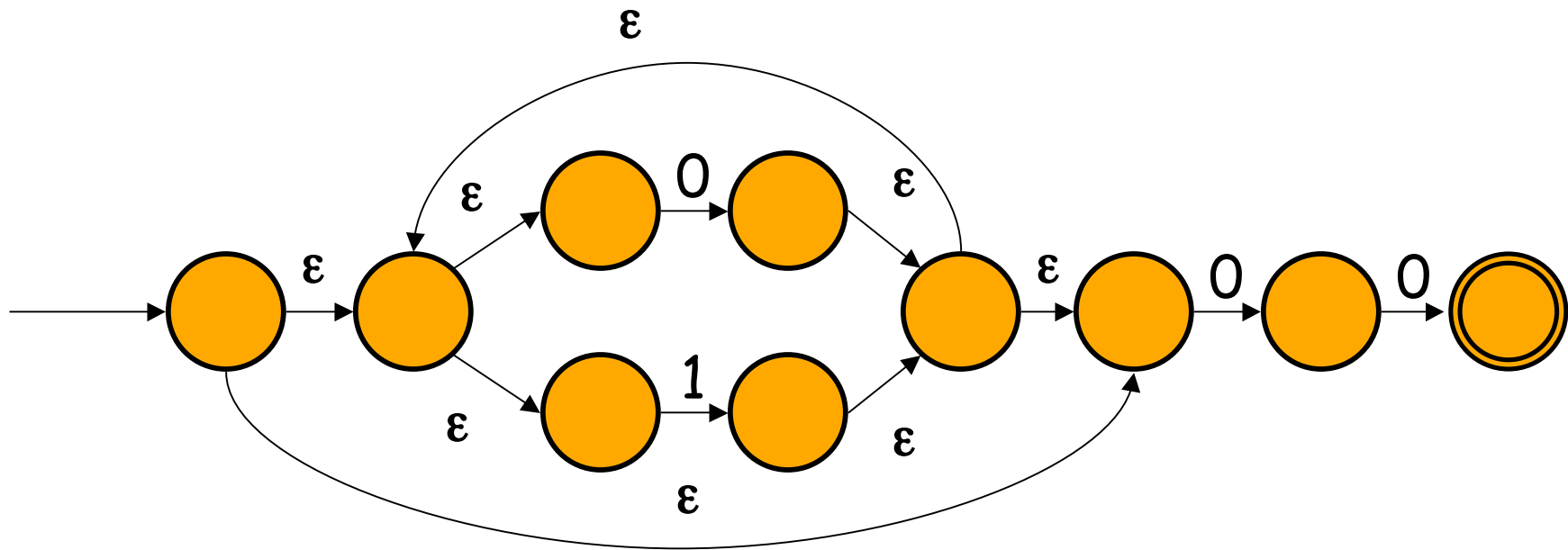


0|1

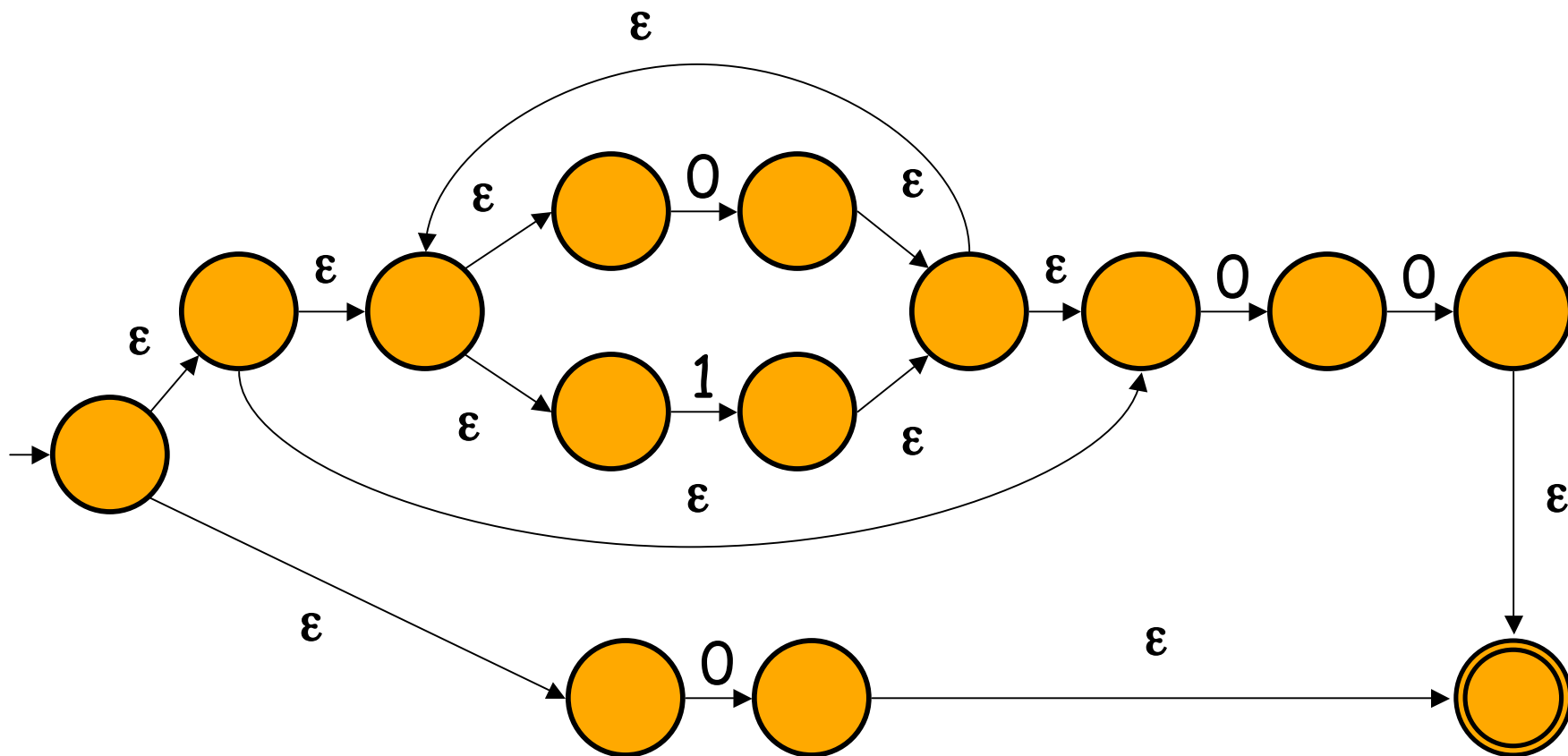


$(0|1)^*$





$(0|1)^*00$



$((0|1)^*00)|0$

# Simulating NFAs

- Similar to DFA simulation
- But have to deal with  $\epsilon$  transitions and multiple transitions on the same input
- Instead of one state, we have to consider *sets* of states
- Simulating NFAs is a problem that is closely linked to converting a given NFA to a DFA

# NFA to DFA Conversion

- Subset construction
- Idea: subsets of set of all NFA states are *equivalent* and become one DFA state
- Algorithm simulates movement through NFA
- Key problem: how to treat  $\epsilon$ -transitions?

# $\epsilon$ -Closure

- Start state:  $q_0$
- $\epsilon$ -closure( $S$ ):  $S$  is a set of states

**initialize:**  $S \leftarrow \{q_0\}$

$T \leftarrow S$

**repeat**  $T' \leftarrow T$

$T \leftarrow T' \cup [\cup_{s \in T'} \text{move}(s, \epsilon)]$

**until**  $T = T'$

## $\epsilon$ -Closure (T: set of states)

```
push all states in T onto stack
initialize  $\epsilon$ -closure(T) to T
while stack is not empty do begin
    pop t off stack
    for each state u with  $u \in \text{move}(t, \epsilon)$  do
        if  $u \notin \epsilon\text{-closure}(T)$  do begin
            add u to  $\epsilon\text{-closure}(T)$ 
            push u onto stack
        end
    end
end
```

# NFA Simulation

- After computing the  $\epsilon$ -closure move, we get a set of states
- On some input extend all these states to get a new set of states

$$\mathbf{DFAedge}(T, c) = \epsilon\text{-closure}(\cup_{q \in T} \mathbf{move}(q, c))$$

# NFA Simulation

- Start state:  $q_0$
- Input:  $c_1, \dots, c_k$

$T \leftarrow \epsilon\text{-closure}(\{q_0\})$

**for**  $i \leftarrow 1$  **to**  $k$

$T \leftarrow \text{DFAedge}(T, c_i)$



# Conversion from NFA to DFA

- Conversion method closely follows the NFA simulation algorithm
- Instead of simulating, we can collect those NFA states that behave identically on the same input
- Group this set of states to form one state in the DFA

# Subset Construction

```
add  $\epsilon$ -closure( $q_0$ ) to  $Dstates$  unmarked
while  $\exists$  unmarked  $T \in Dstates$  do begin
    mark  $T$ ;
    for each symbol  $c$  do begin
         $U := \epsilon$ -closure(move( $T, c$ ));
        if  $U \notin Dstates$  then
            add  $U$  to  $Dstates$  unmarked
         $Dtrans[d, c] := U$ ;
    end
end
```

# Subset Construction

$\text{states}[0] = \varepsilon\text{-closure}(\{q_0\})$

$p = j = 0$

**while**  $j \leq p$  **do begin**

**for** each symbol  $c$  **do begin**

$e = \mathbf{DFAedge}(\text{states}[j], c)$

**if**  $e = \text{states}[i]$  for some  $i \leq p$

**then**    $\text{Dtrans}[j, c] = i$

**else**    $p = p+1$

$\text{states}[p] = e$

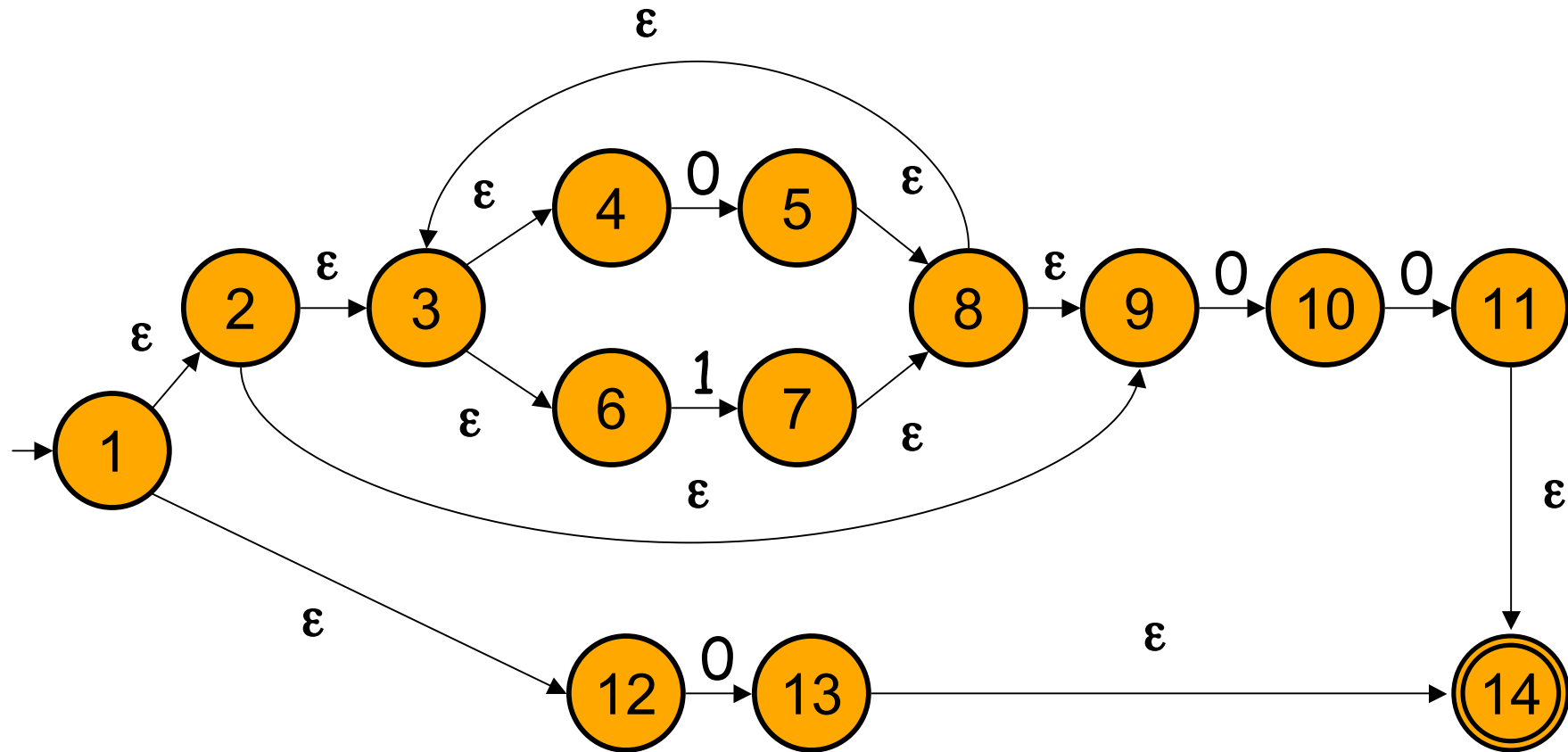
$\text{Dtrans}[j, c] = p$

$j = j + 1$

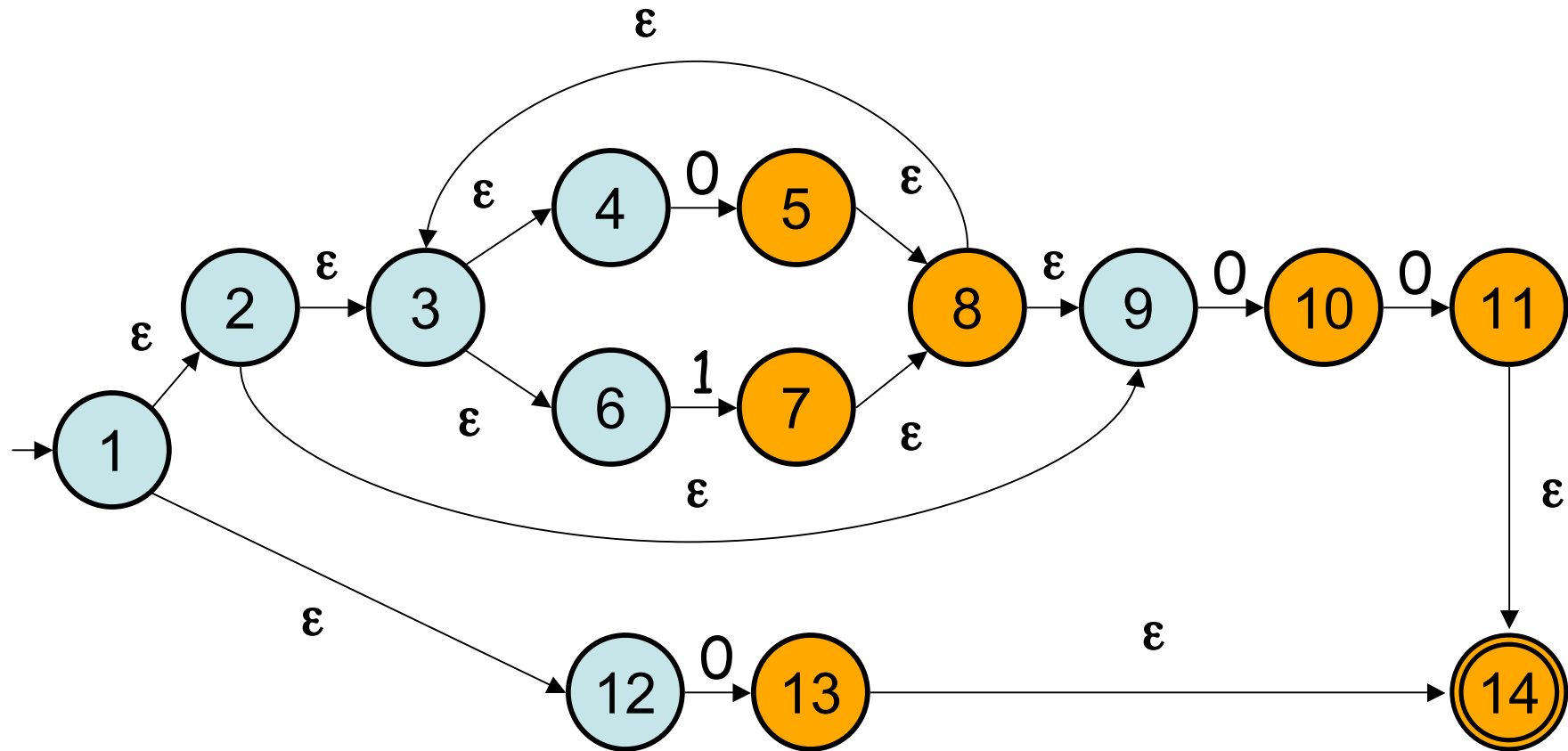
**end**

**end**

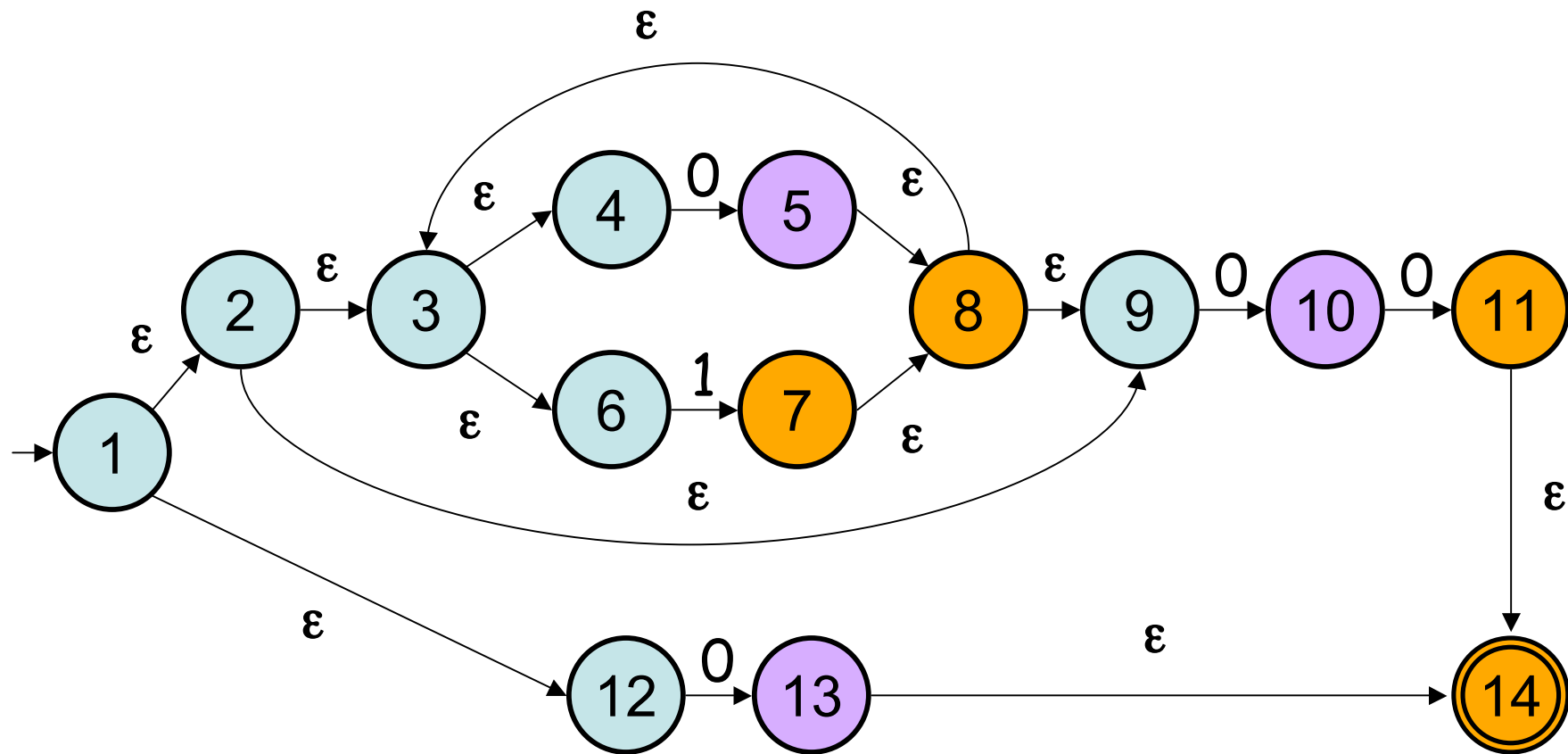
# Example: subset construction



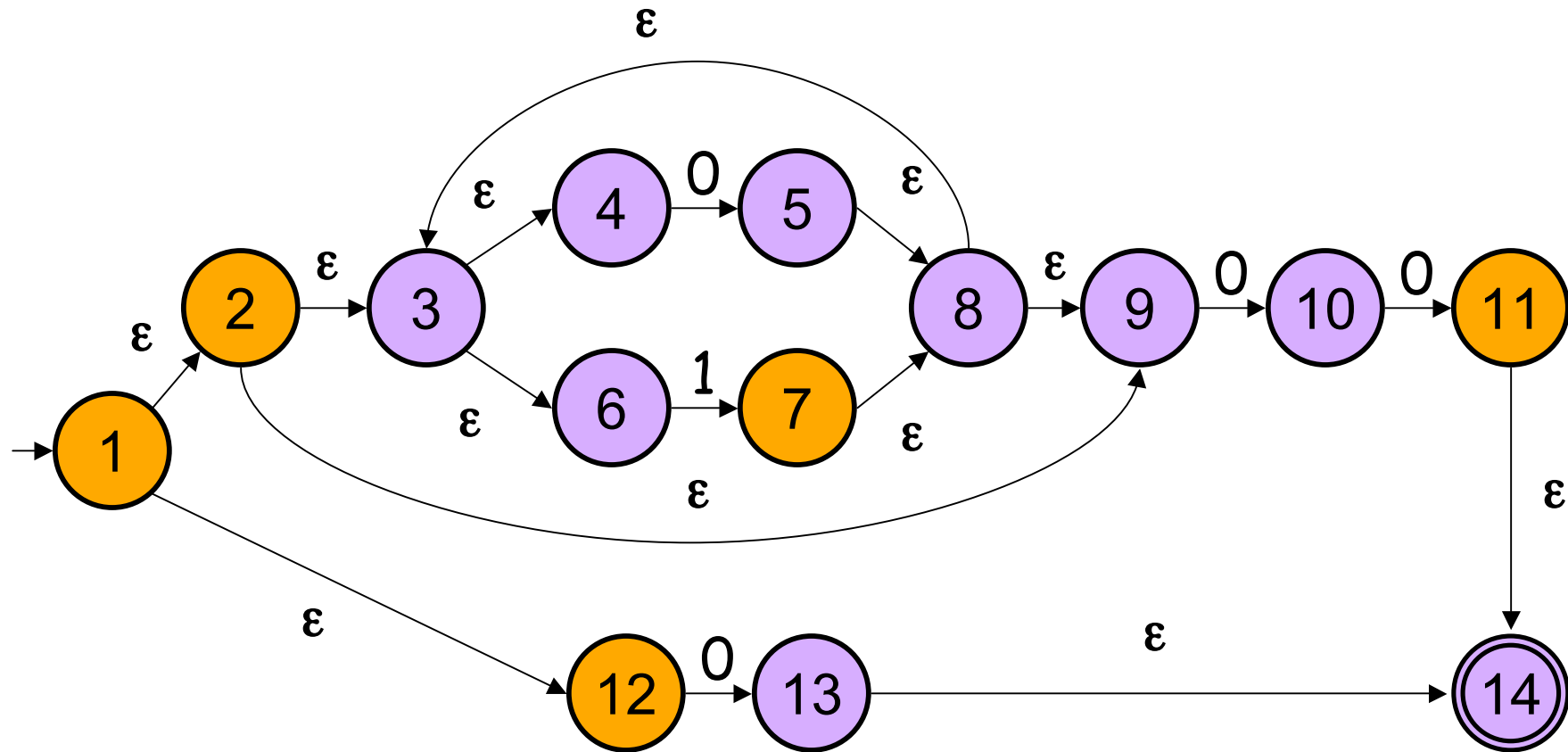
$\varepsilon$ -closure( $q_0$ )



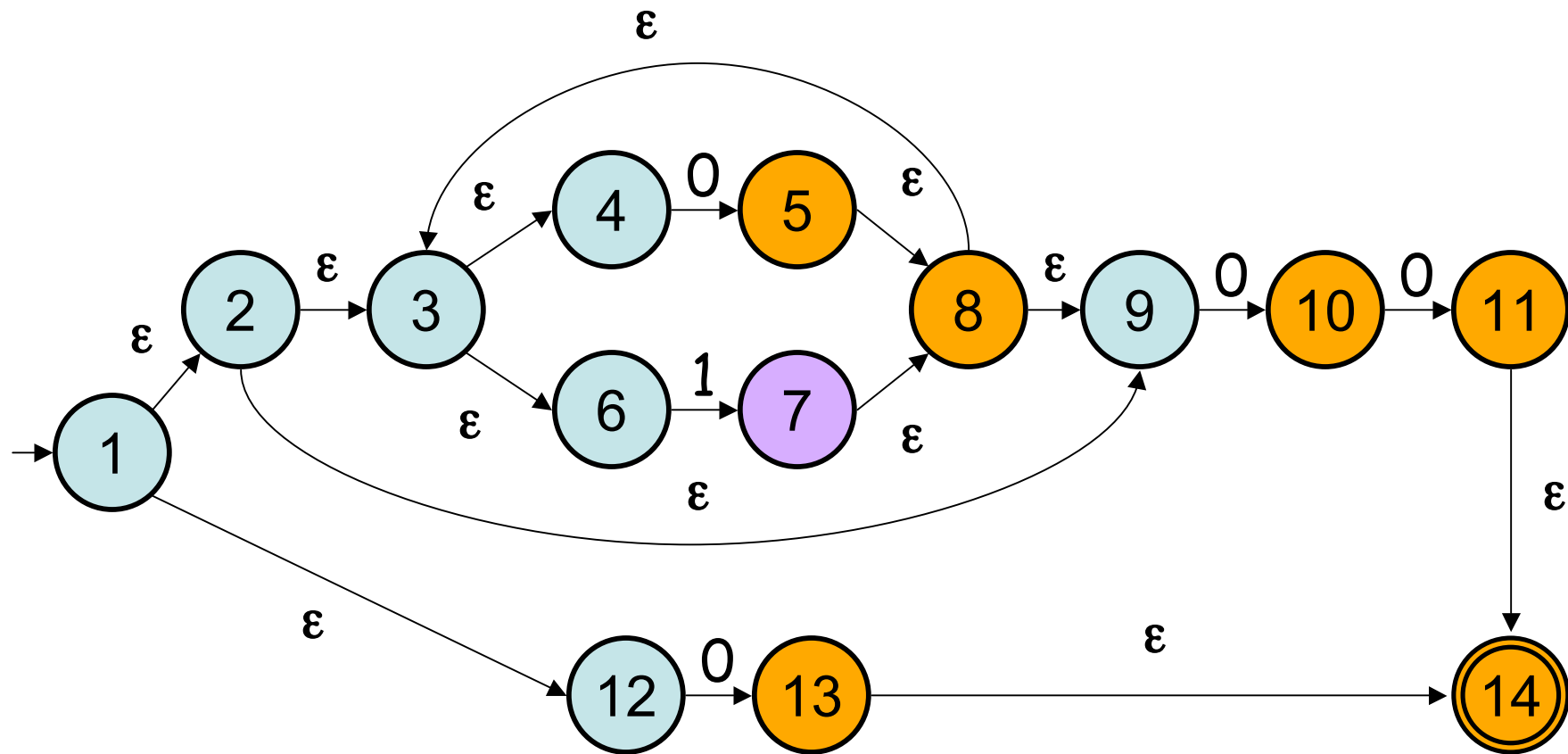
$\text{move}(\epsilon\text{-closure}(q_0), 0)$



$\epsilon\text{-closure}(\text{move}(\epsilon\text{-closure}(q_0), 0))$

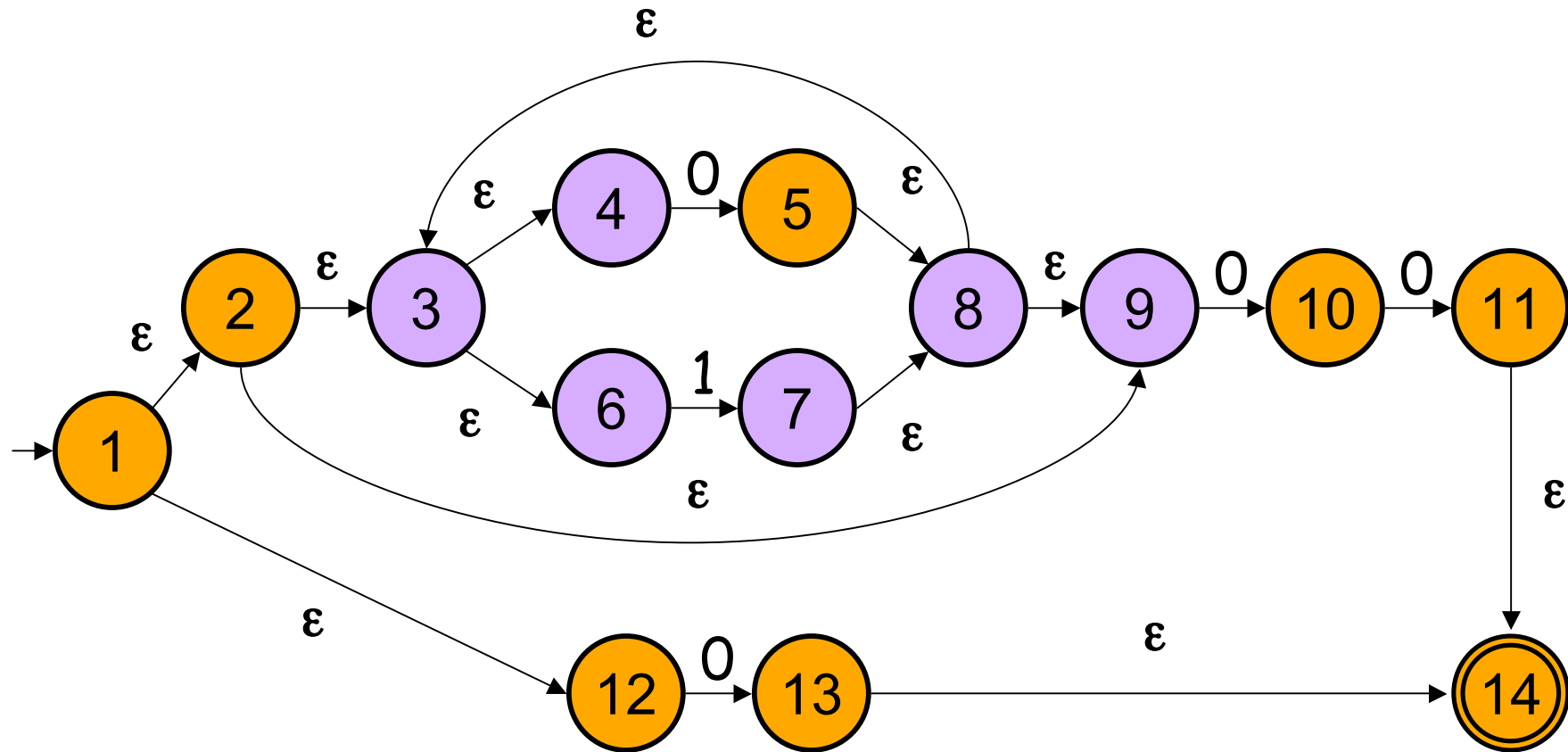


$\text{move}(\epsilon\text{-closure}(q_0), 1)$

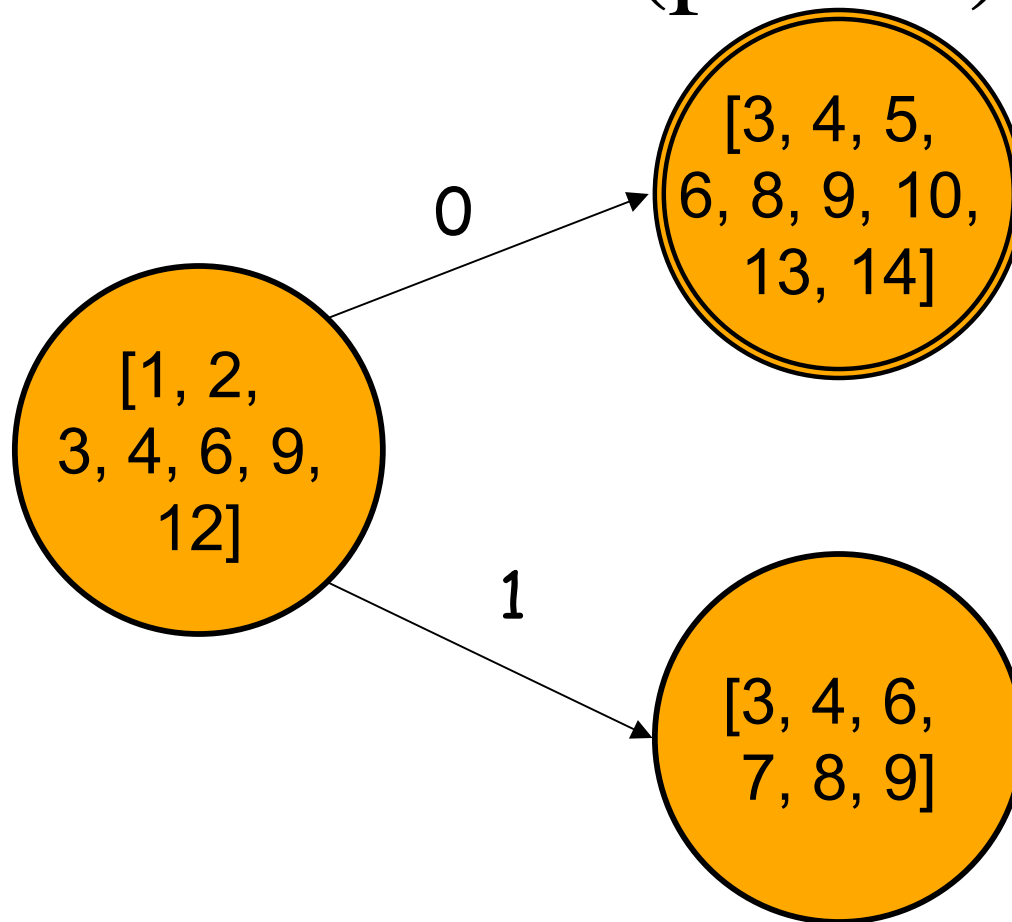




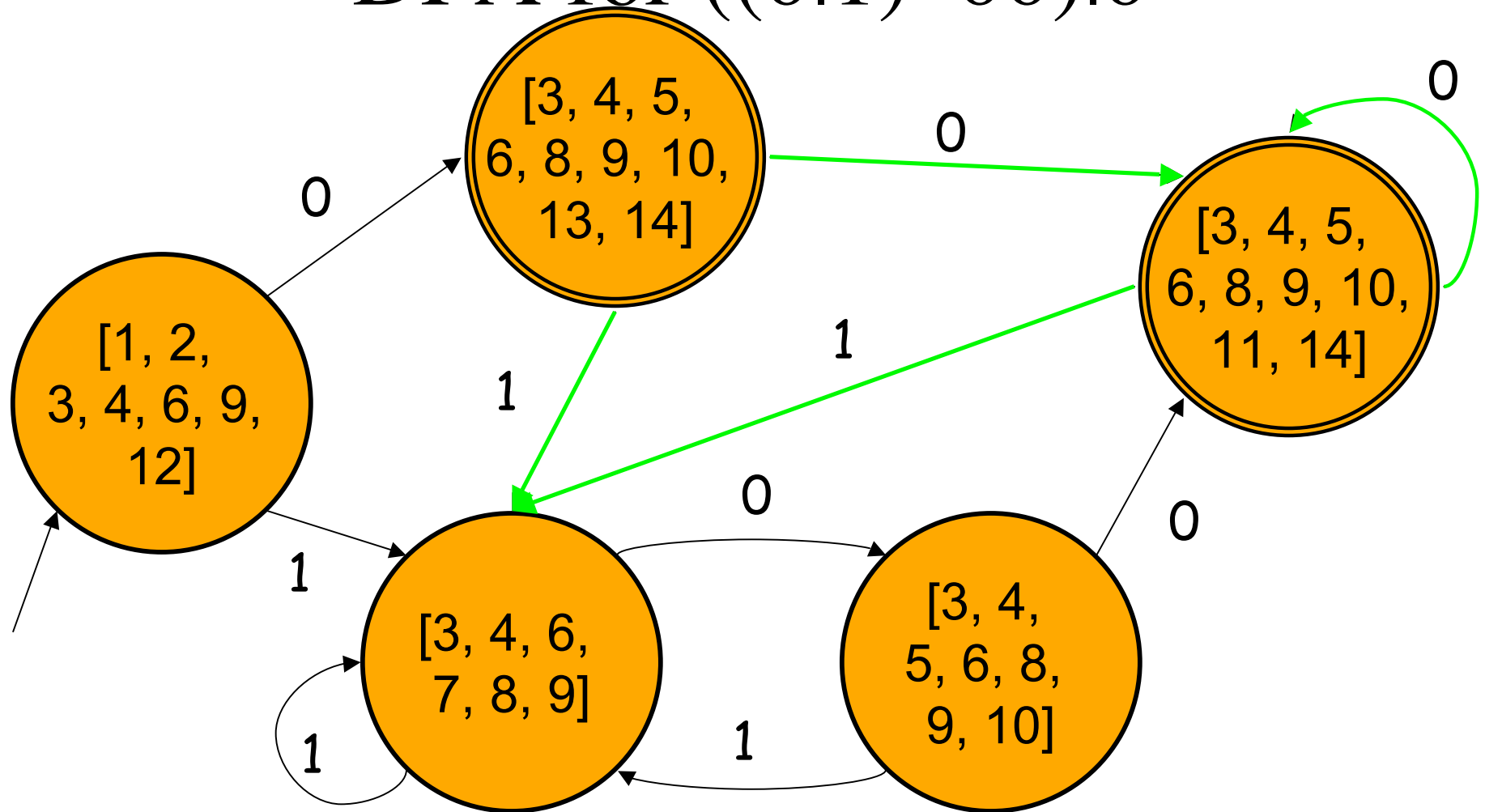
$\epsilon\text{-closure}(\text{move}(\epsilon\text{-closure}(q_0), 1))$



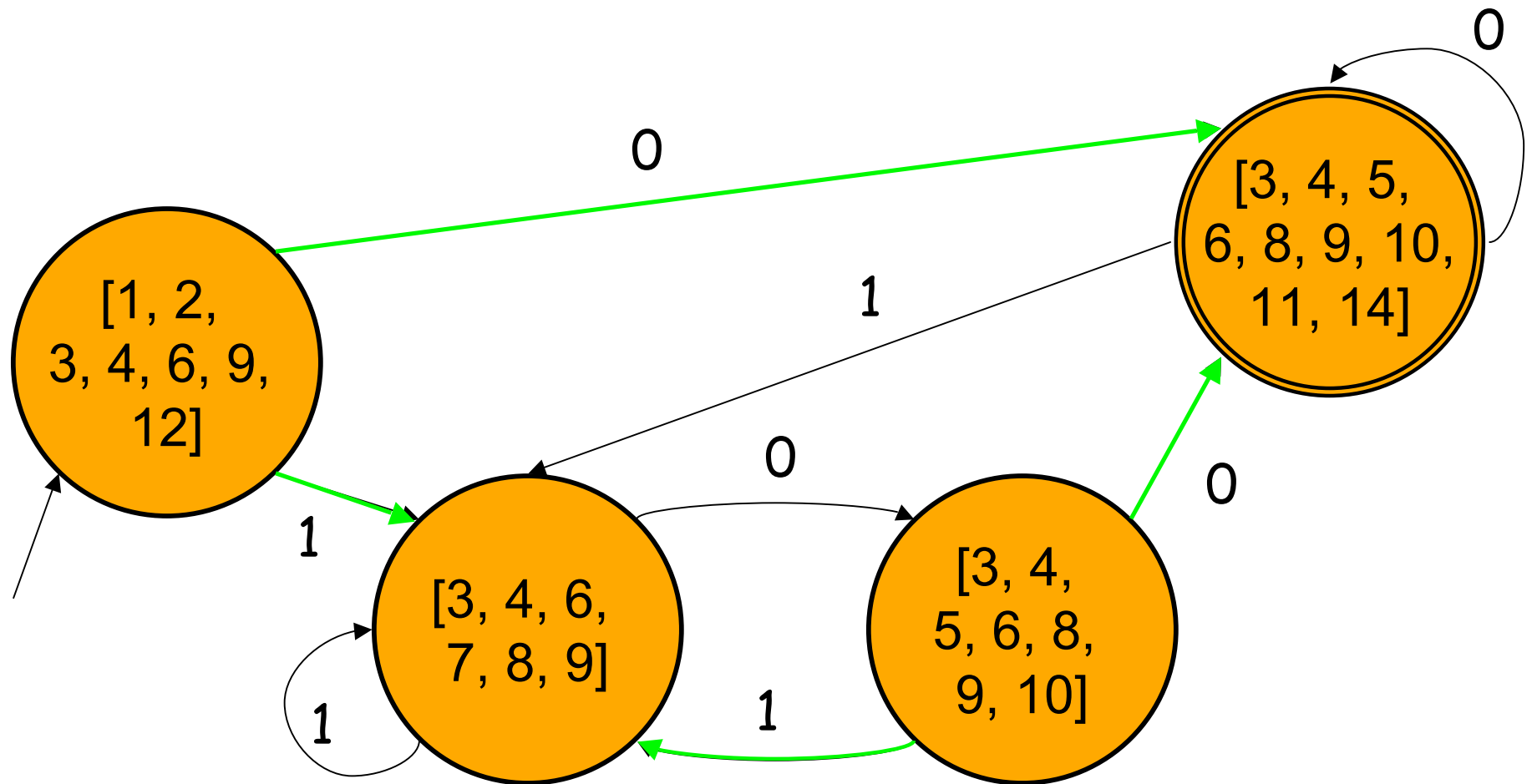
# DFA (partial)



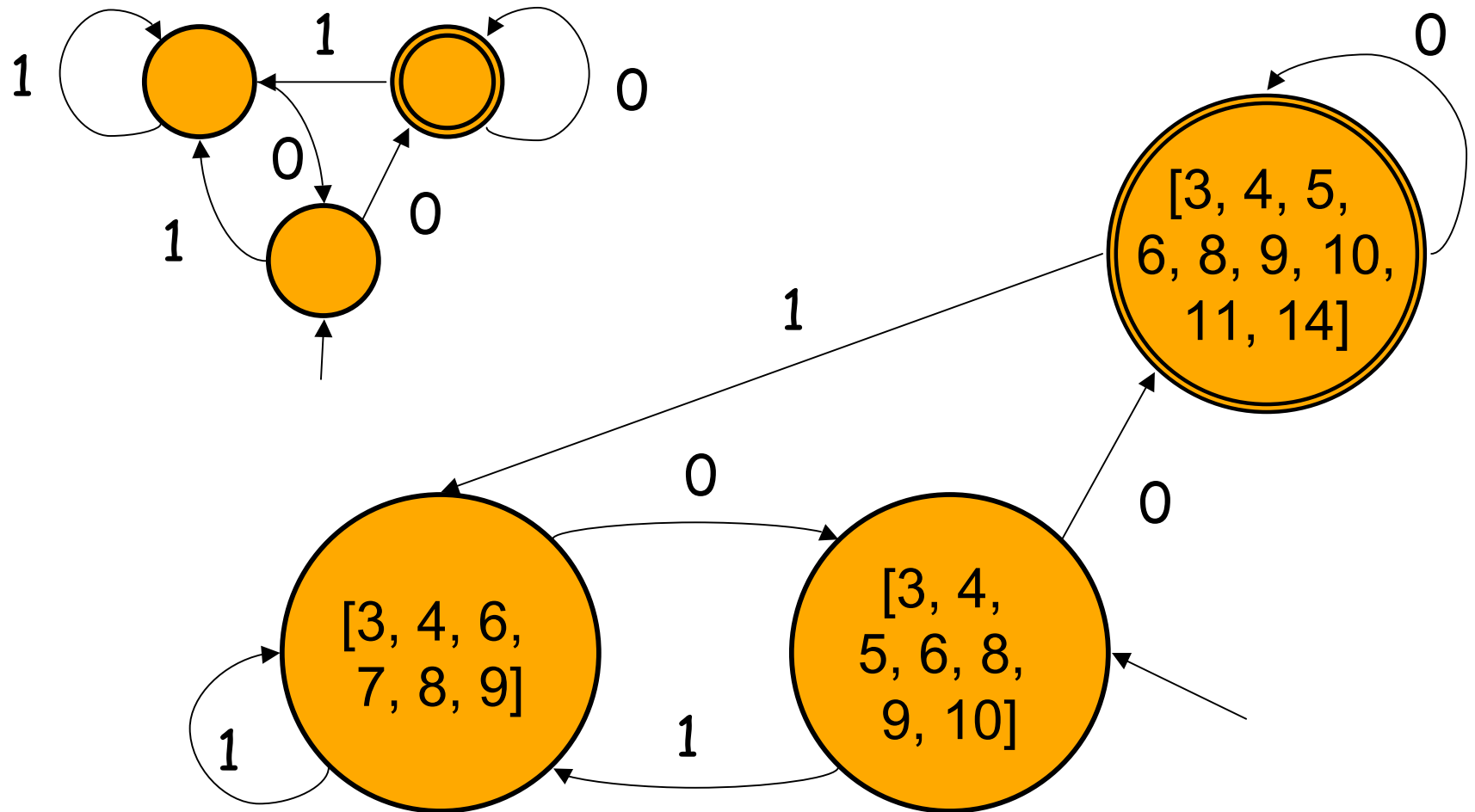
# DFA for $((0|1)^*00)|0$



# Minimization (I)



# Minimization (II)



# NFA vs. DFA in the wild

Engine Type	Programs
DFA	<i>awk</i> (most versions), <i>egrep</i> (most versions), <i>flex</i> , <i>lex</i> , MySQL, Procmail
Traditional NFA	GNU <i>Emacs</i> , Java, <i>grep</i> (most versions), <i>less</i> , <i>more</i> , .NET languages, PCRE library, Perl, PHP (pcre routines), Python, Ruby, <i>sed</i> (most versions), vi
POSIX NFA	<i>mawk</i> , MKS utilities, GNU <i>Emacs</i> (when requested)
Hybrid NFA/DFA	GNU <i>awk</i> , GNU <i>grep/egrep</i> , Tcl

# Lexical Analyzer using DFAs

- Each token is defined using a regexp  $r_i$
- Merge all regexps into one big regexp
  - $R = (r_1 \mid r_2 \mid \dots \mid r_n)$
- Convert  $R$  to an NFA, then DFA, then minimize
  - remember orig NFA final states with each DFA state

# Lexical Analyzer using DFAs

- The DFA recognizer has to find the *longest match* for a token
  - e.g.  $\langle print \rangle$  and not  $\langle pr \rangle$ ,  $\langle int \rangle$
- If two patterns match the same token, pick the one that was listed earlier in R
  - e.g. prefer final state (in the original NFA) of  $r_2$  over  $r_3$



# Implementing DFAs

- 2D array storing the transition table
- Adjacency list, more space efficient but slower
- Merge two ideas: array structures used for sparse tables like DFA transition tables
  - base & next arrays: Tarjan and Yao, 1979
  - Dragon book (default+base & next+check)

# Summary

- Token  $\Rightarrow$  Pattern
  - Pattern  $\Rightarrow$  Regular Expression
  - Regular Expression  $\Rightarrow$   $\epsilon$ -NFA
    - Thompson's Rules
  - $\epsilon$ -NFA  $\Rightarrow$  DFA
    - Subset constructions
  - DFA  $\Rightarrow$  minimal DFA
    - Minimization
- $\Rightarrow$  Lexical Analyzer (multiple patterns)**