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: ε
- Set of all strings: Σ^*

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. aa|bc vs. ((aa)|b)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

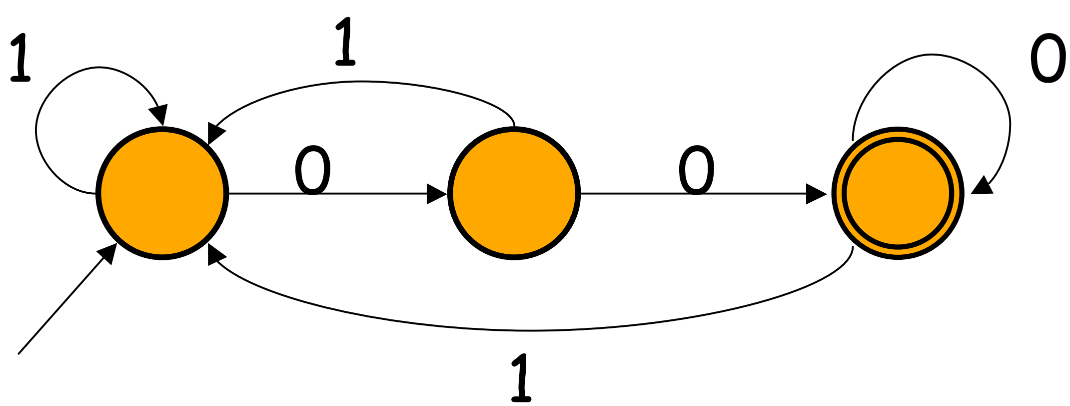
- To describe all lexemes that form a token as a *pattern*
  - (0|1|2|3|4|5|6|7|8|9)+
- Need decision procedure: to which token does a given sequence of characters belong (if any)?
  - Finite State Automata

Finite Automata: Recap

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

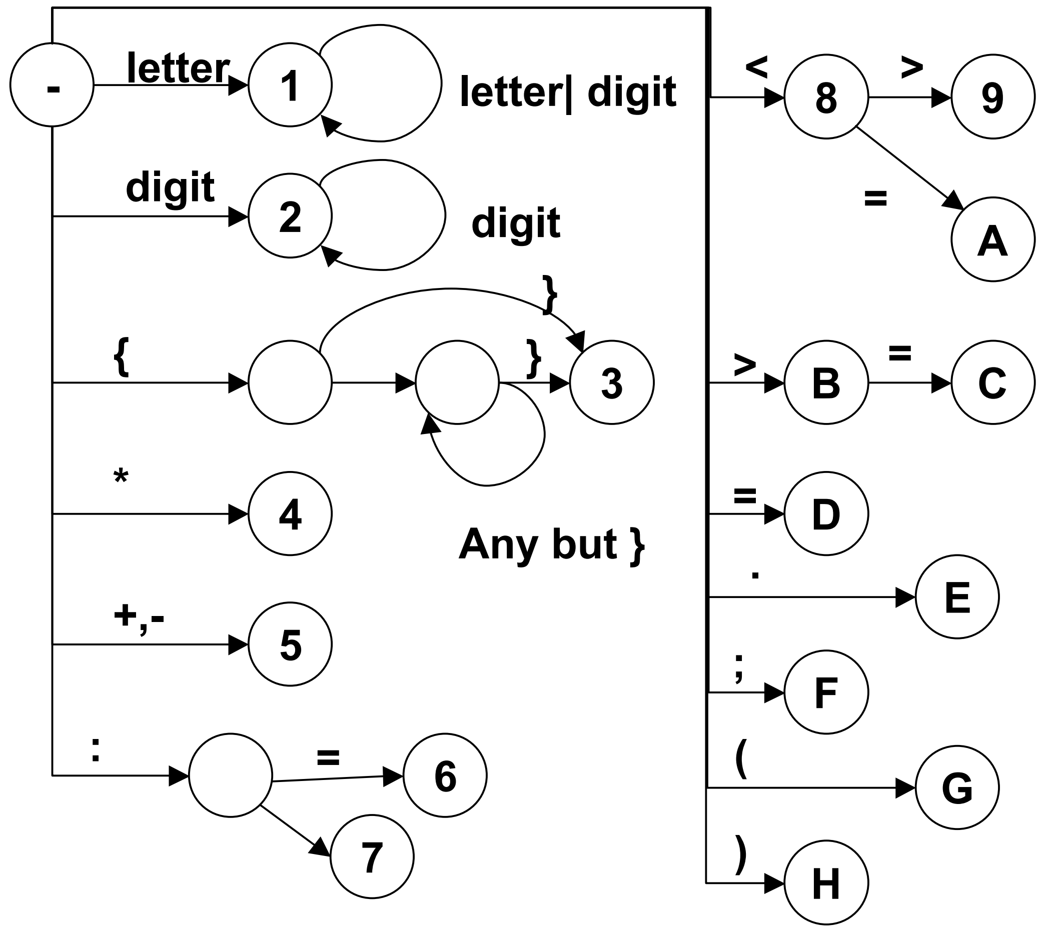
Finite Automata: Example

- 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 ε, meaning states can be reached without reading any input, that is,

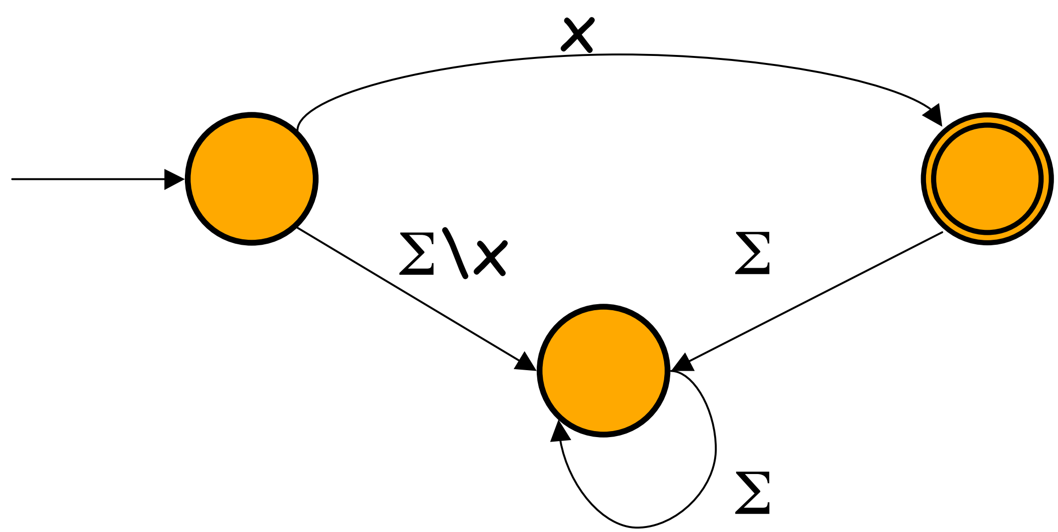
$$\delta: S \times \Sigma \cup \{ \varepsilon \} \Rightarrow 2^S$$

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*)

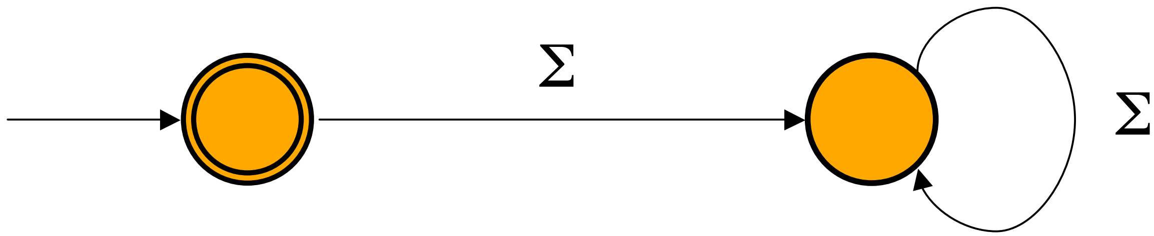
Thompson Rule 1

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



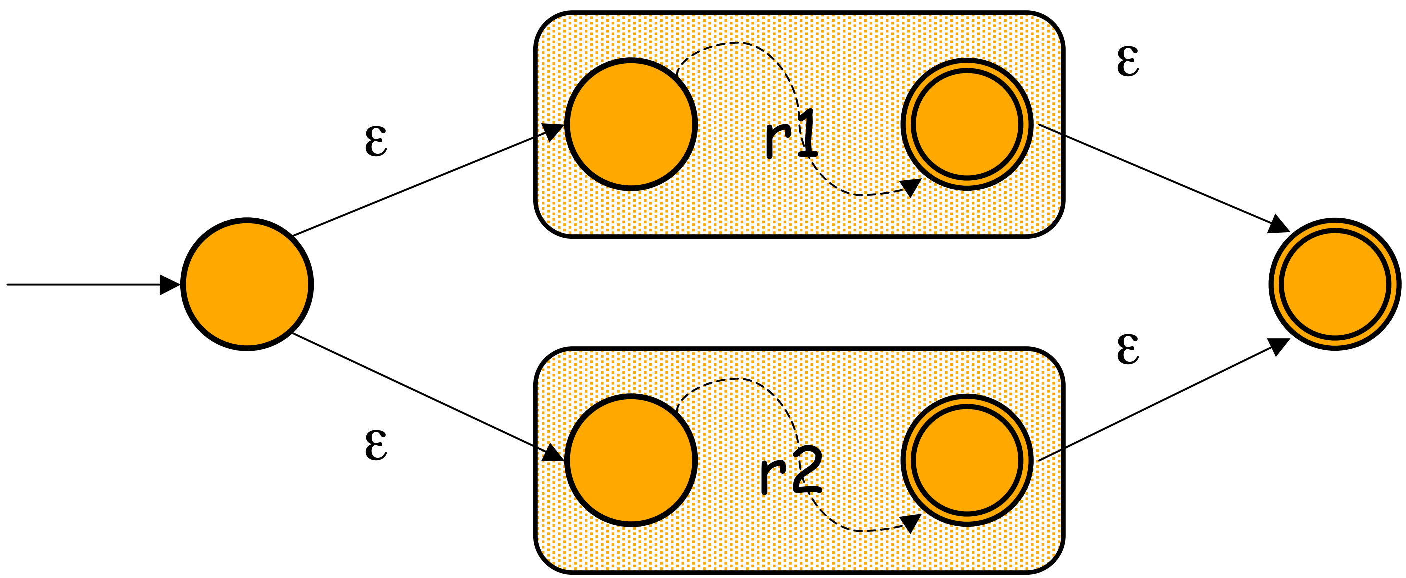
Thompson Rule 2

- There is an NFA that accepts only ε



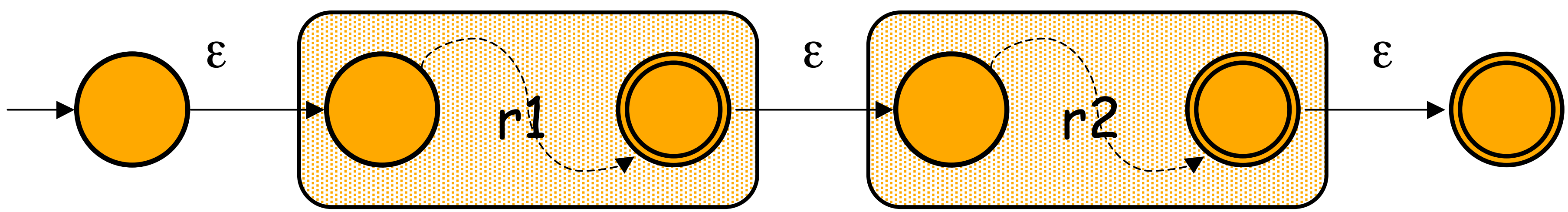
Thompson Rule 3

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



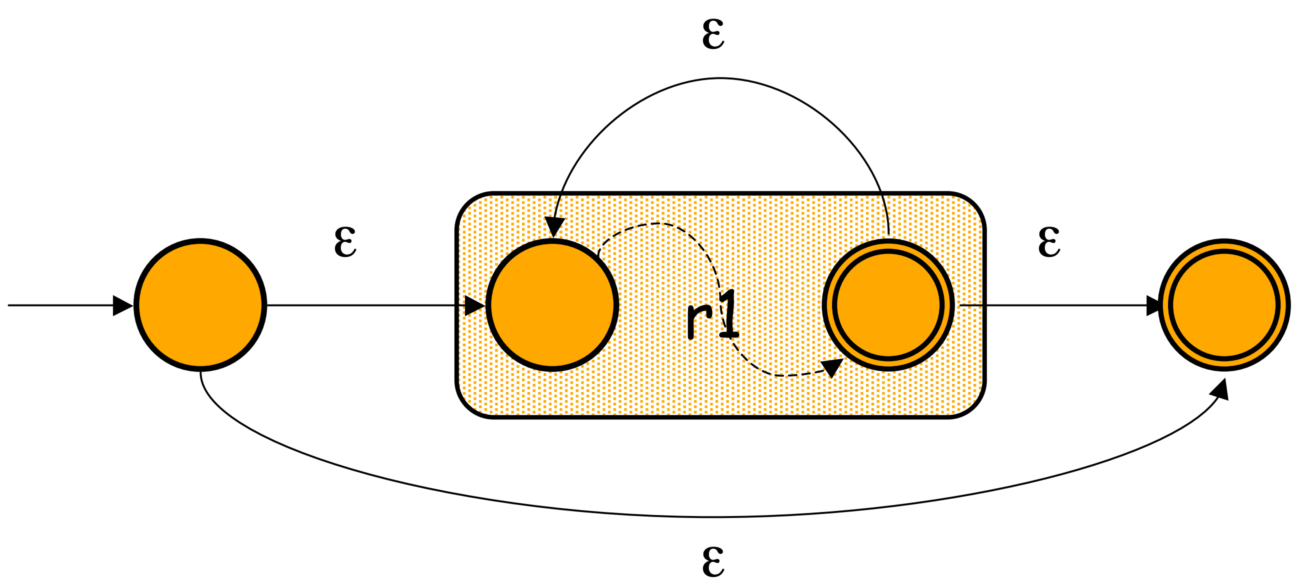
Thompson Rule 4

- 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*

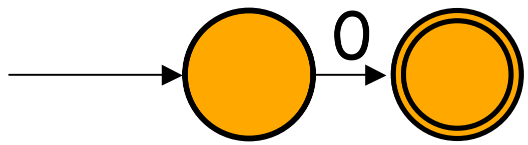


Example

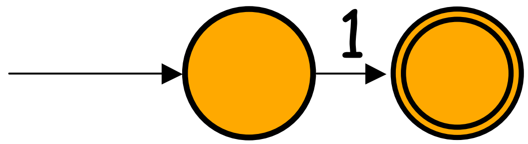
- Set of all binary strings that are divisible by four (include 0 in this set)
- Defined by the regexp: ((0|1)*00) | 0
- Apply Thompson's Rules to create an NFA

Basic Blocks 0 and 1

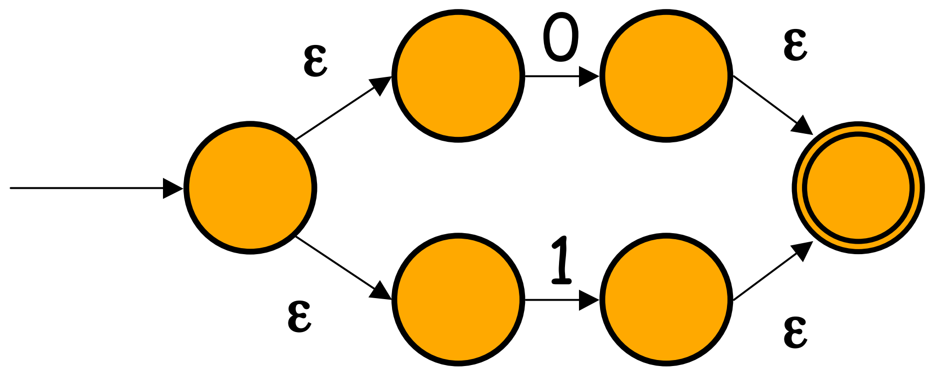
- 0



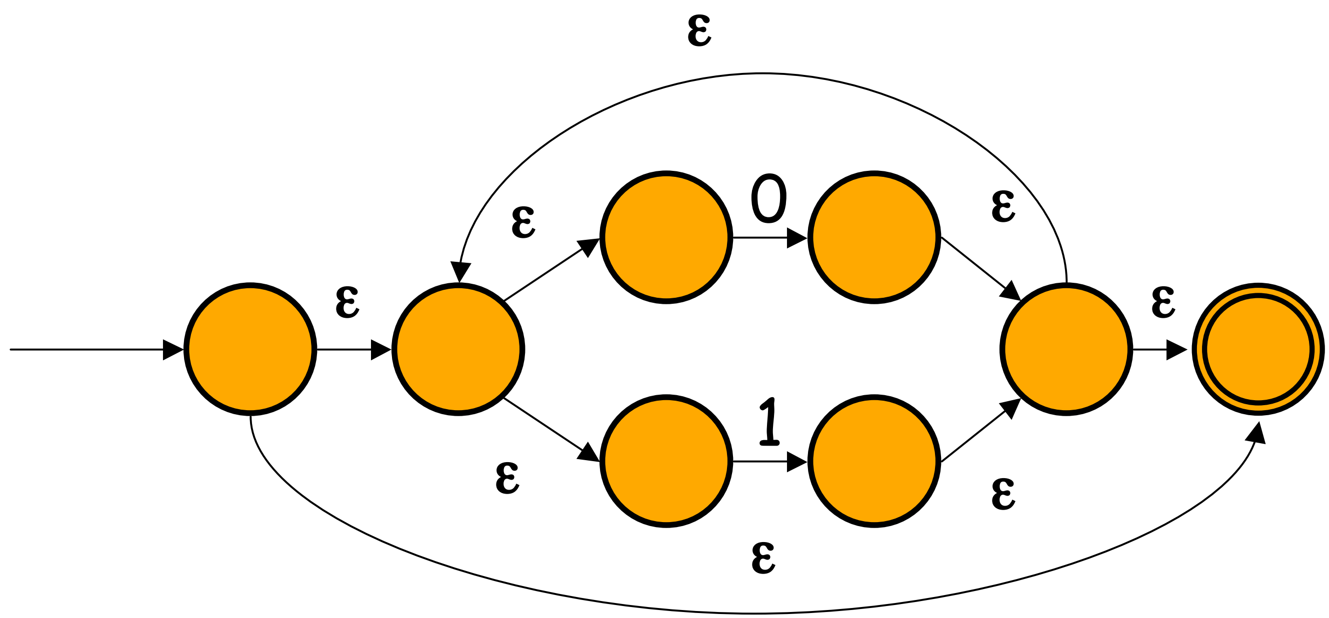
- 1



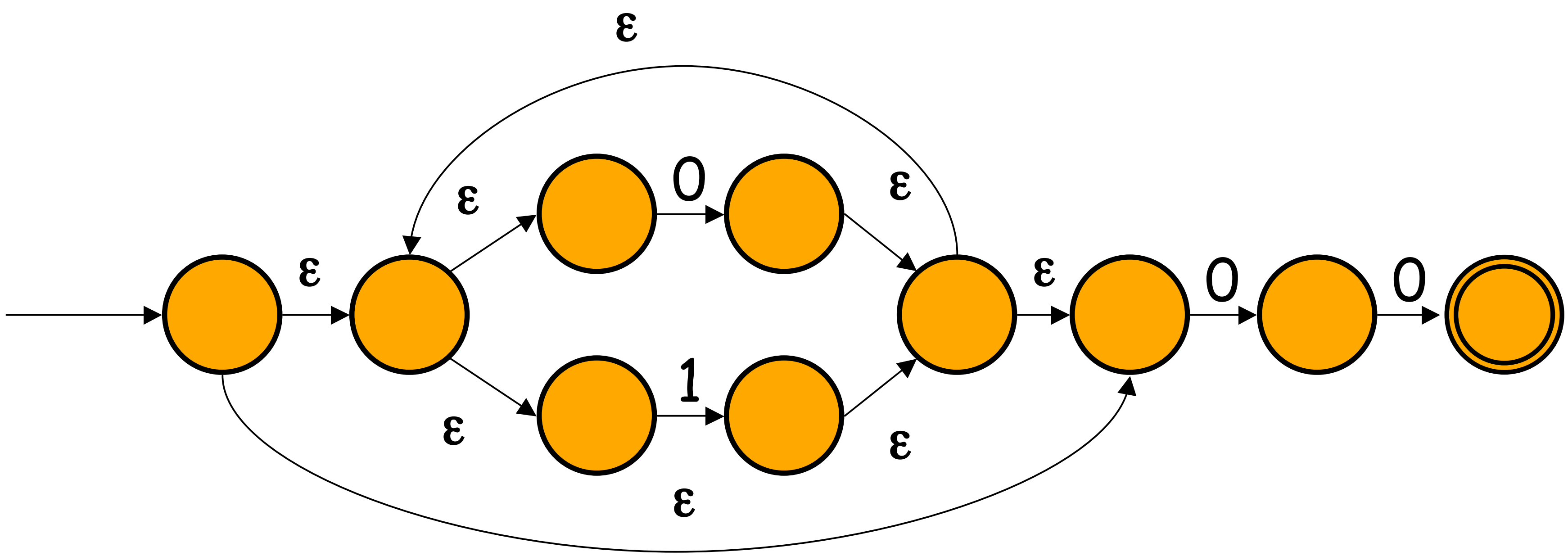
(this version does not report errors: no *sinkholes*)



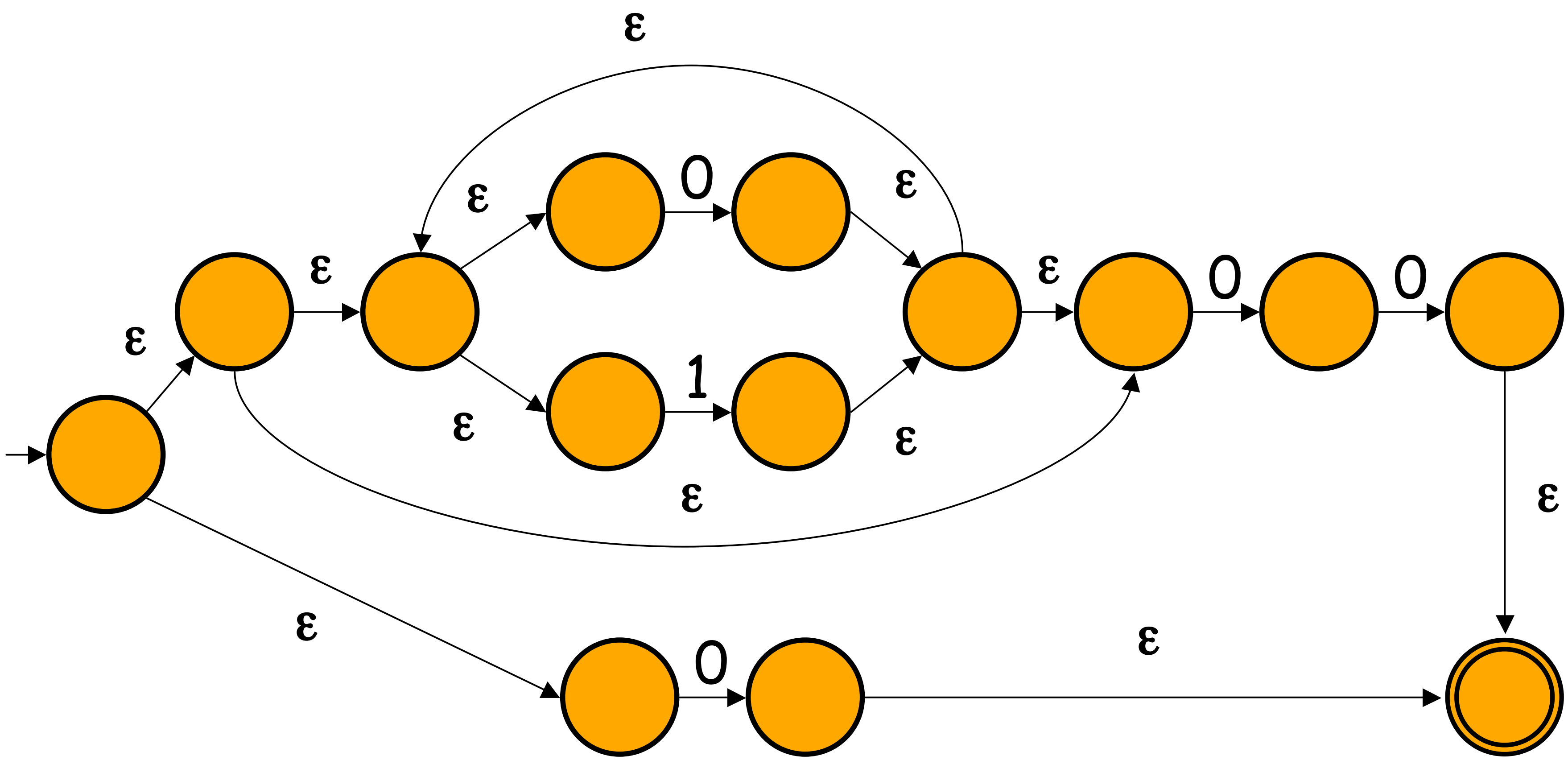
0|1



(0|1)*



(0|1)*00



((0|1)*00)|0

Simulating NFAs

- Similar to DFA simulation
- But have to deal with ε 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 ε-transitions?

ε-Closure

- Start state: q_0
- ε-closure(S): S is a set of states

$$
\begin{aligned}
&\textbf{initialize: } S \leftarrow \{q_0\} \\
&T \leftarrow S \\
&\textbf{repeat } T' \leftarrow T \\
&\quad T \leftarrow T' \cup [\cup_{s \in T'} \textbf{move}(s, \epsilon)] \\
&\textbf{until } T = T'
\end{aligned}
$$

ε-Closure (T: set of states)

```
push all states in T onto stack
initialize ε-closure(T) to T
while stack is not empty do begin
      pop t off stack
      for each state u with u ∈ move(t, ε) do
            if u ∉ ε-closure(T) do begin
                  add u to ε-closure(T)
                  push u onto stack
            end
end
```

NFA Simulation

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

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

NFA Simulation

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

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

$$\textbf{for } i \leftarrow 1 \textbf{ to } k$$

$$\quad T \leftarrow \textbf{DFAedge}(T, c_i)$$

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

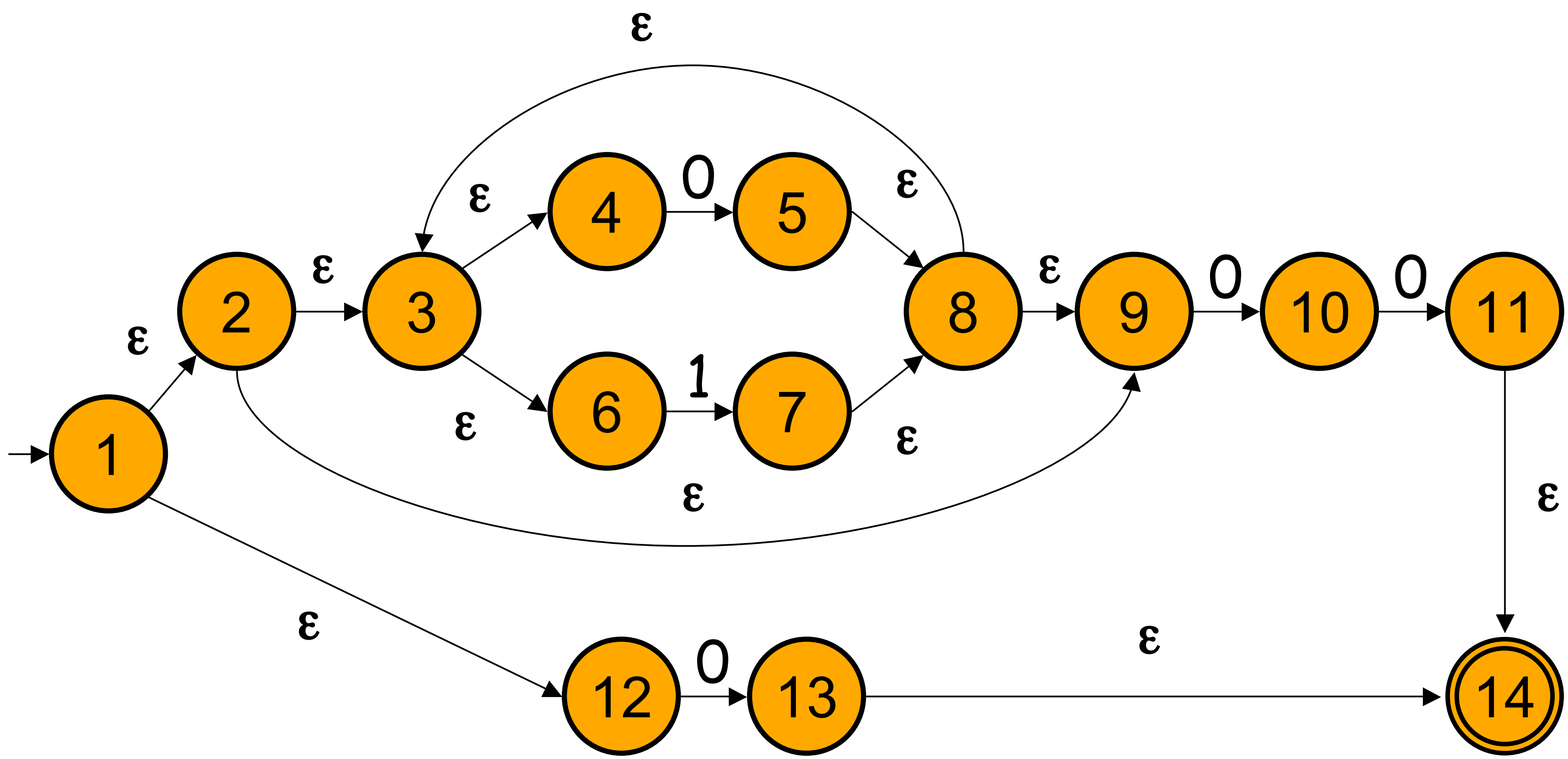
Subset Construction

```
add ε-closure(q0) to Dstates unmarked
while ∃ unmarked T ∈ Dstates do begin
      mark T;
      for each symbol c do begin
            U := ε-closure(move(T, c));
            if U ∉ Dstates then
                  add U to Dstates unmarked
            Dtrans[T, c] := U;
      end
end
```

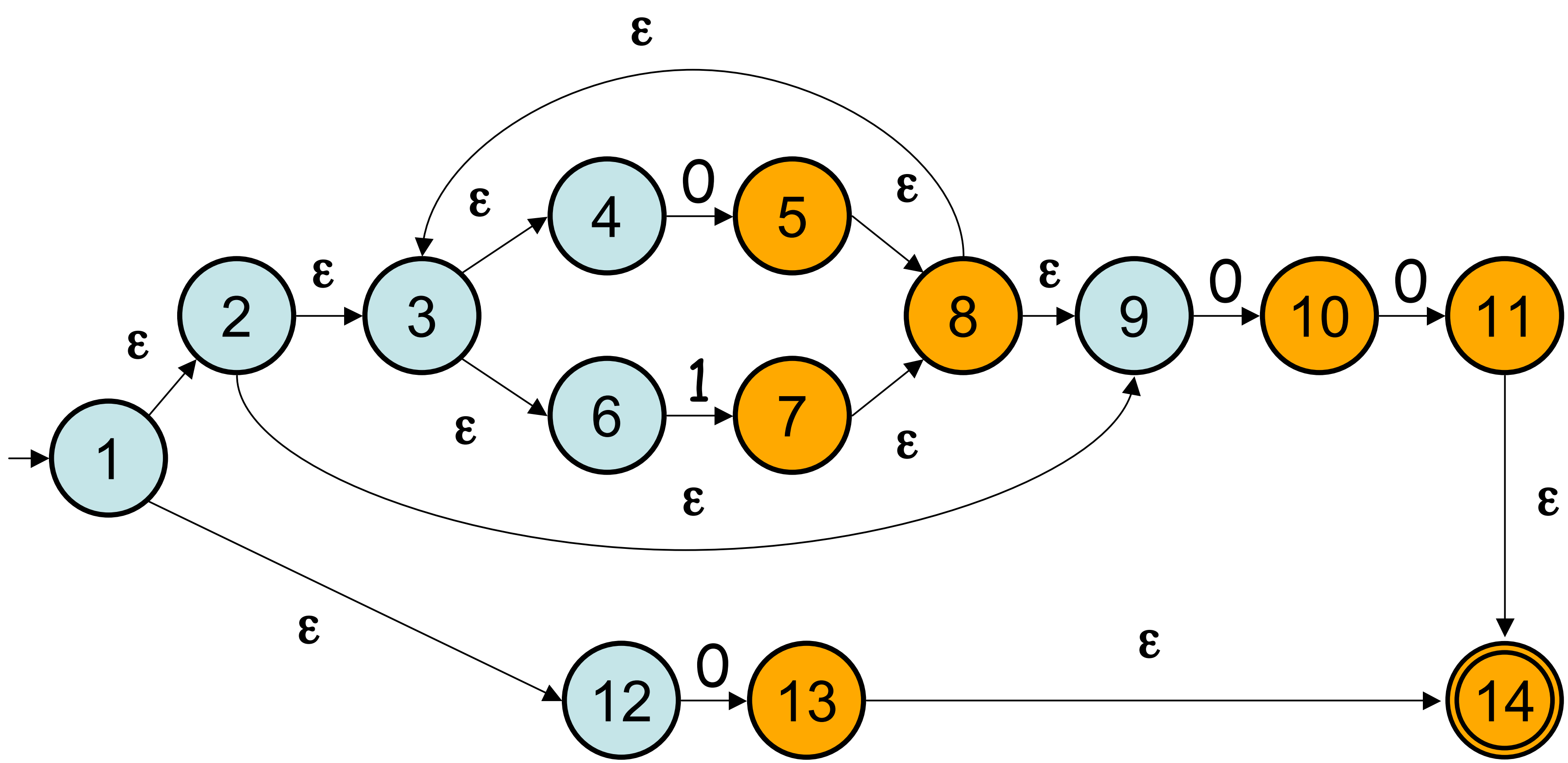
Subset Construction

```
states[0] = ε-closure({q0})
p = j = 0
while j ≤ p do begin
        for each symbol c do begin
                e = DFAedge(states[j], c)
                if e = states[i] for some i ≤ p
                then    Dtrans[j, c] = i
                else    p = p+1
                        states[p] = e
                        Dtrans[j, c] = p
        j = j + 1
        end
end
```

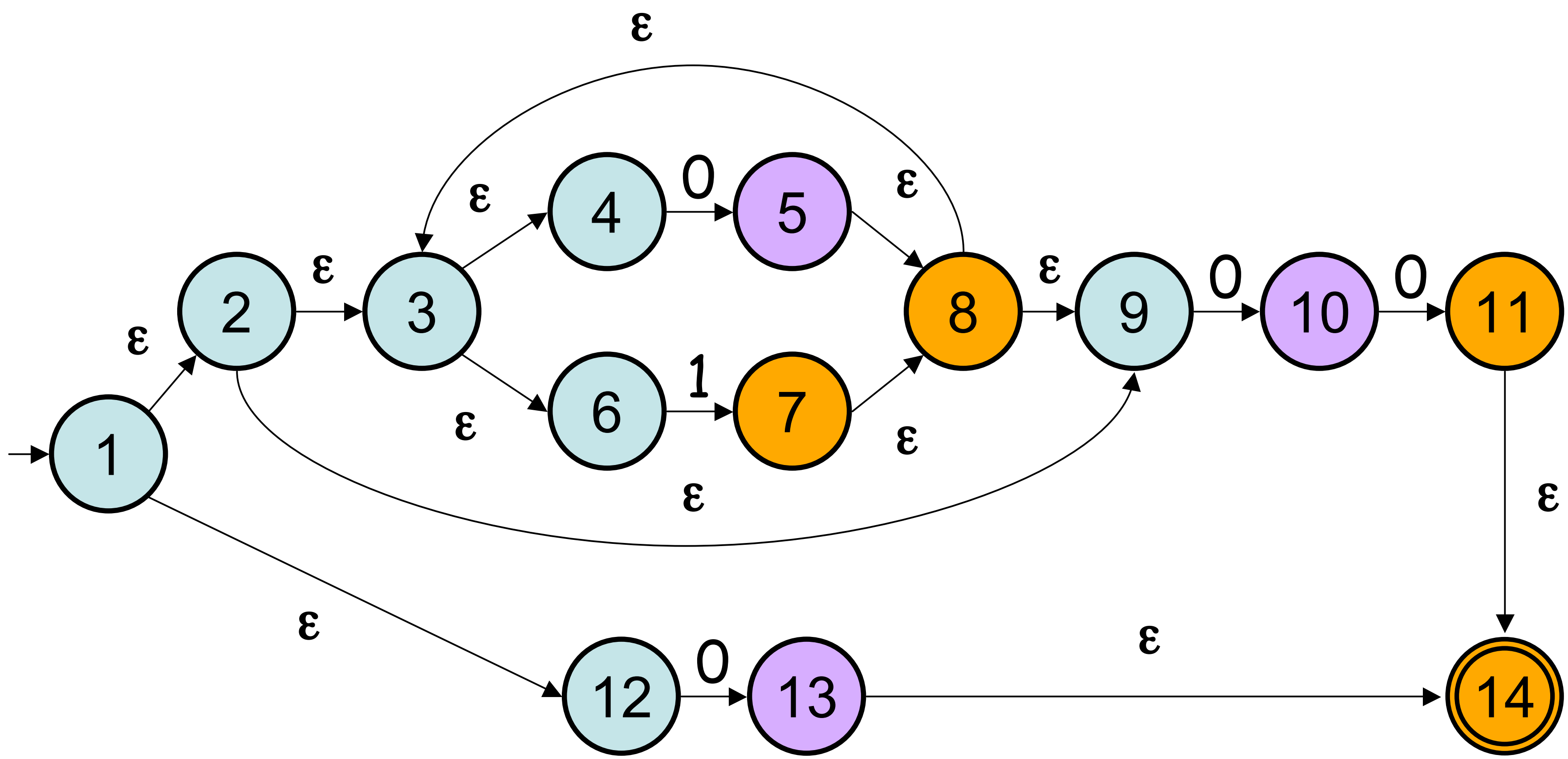
Example: subset construction



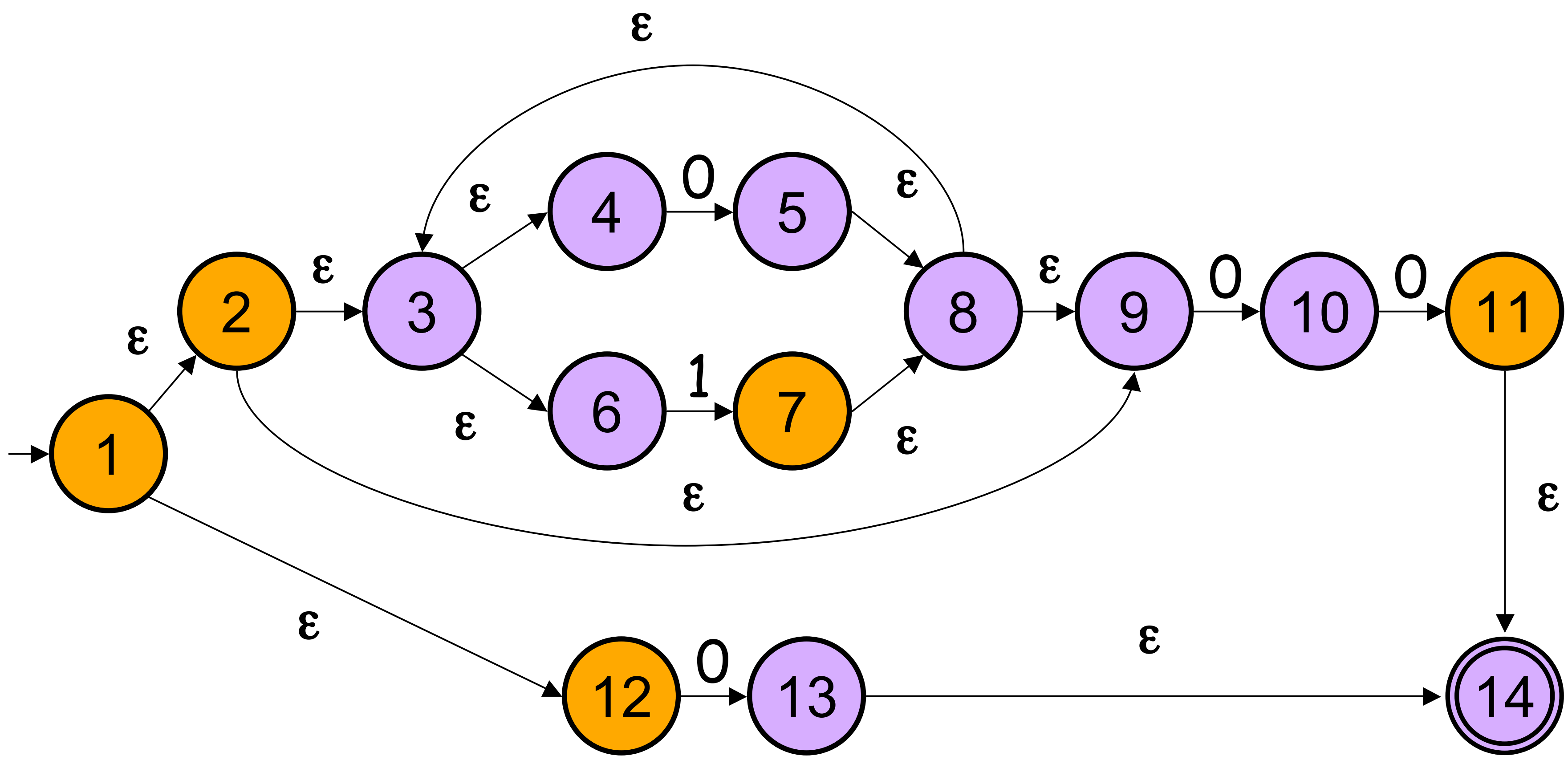
ε-*closure*(q_0)



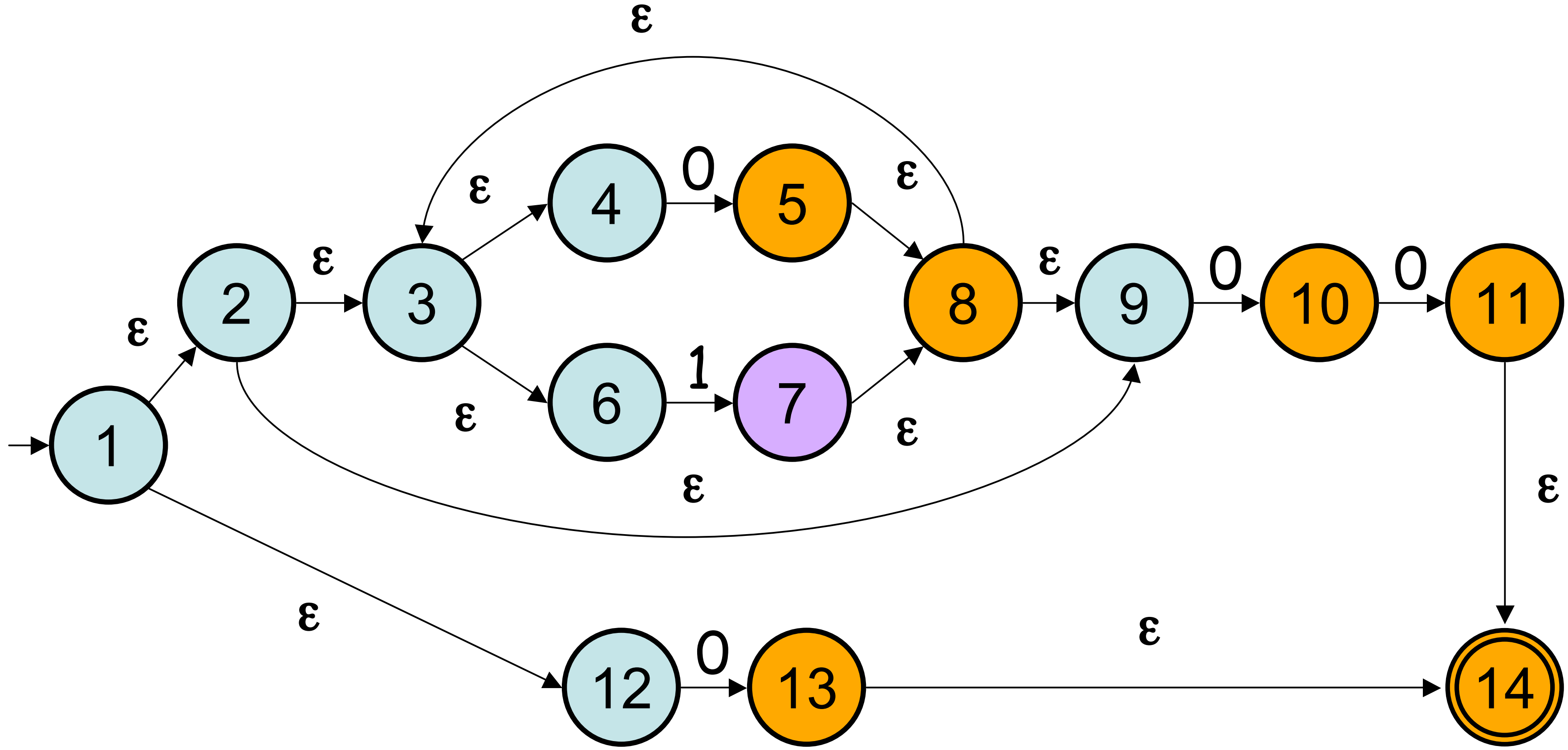
move(ε-*closure*(q_0), 0)



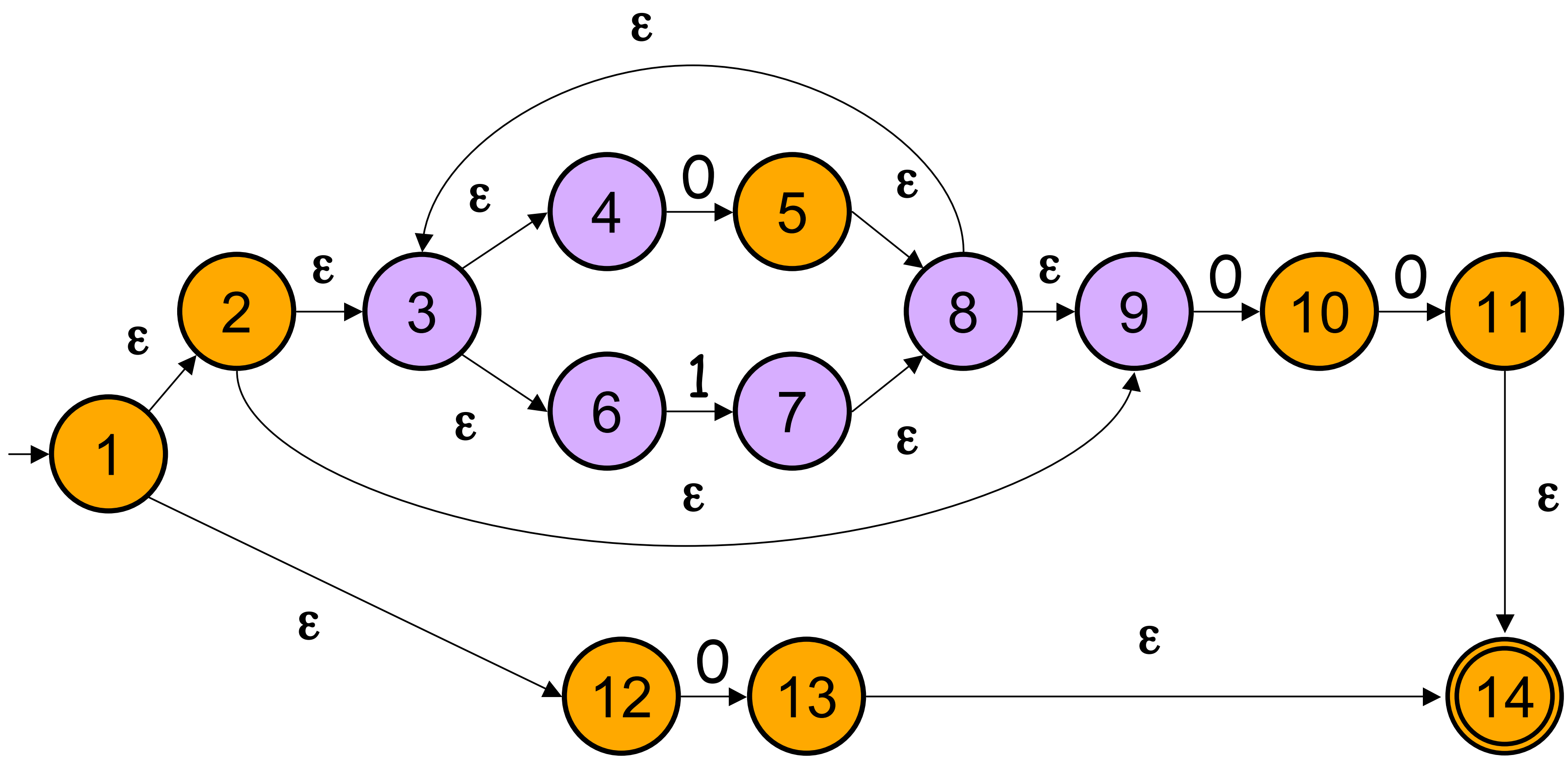
ε-*closure*(move(ε-*closure*(q_0), 0))



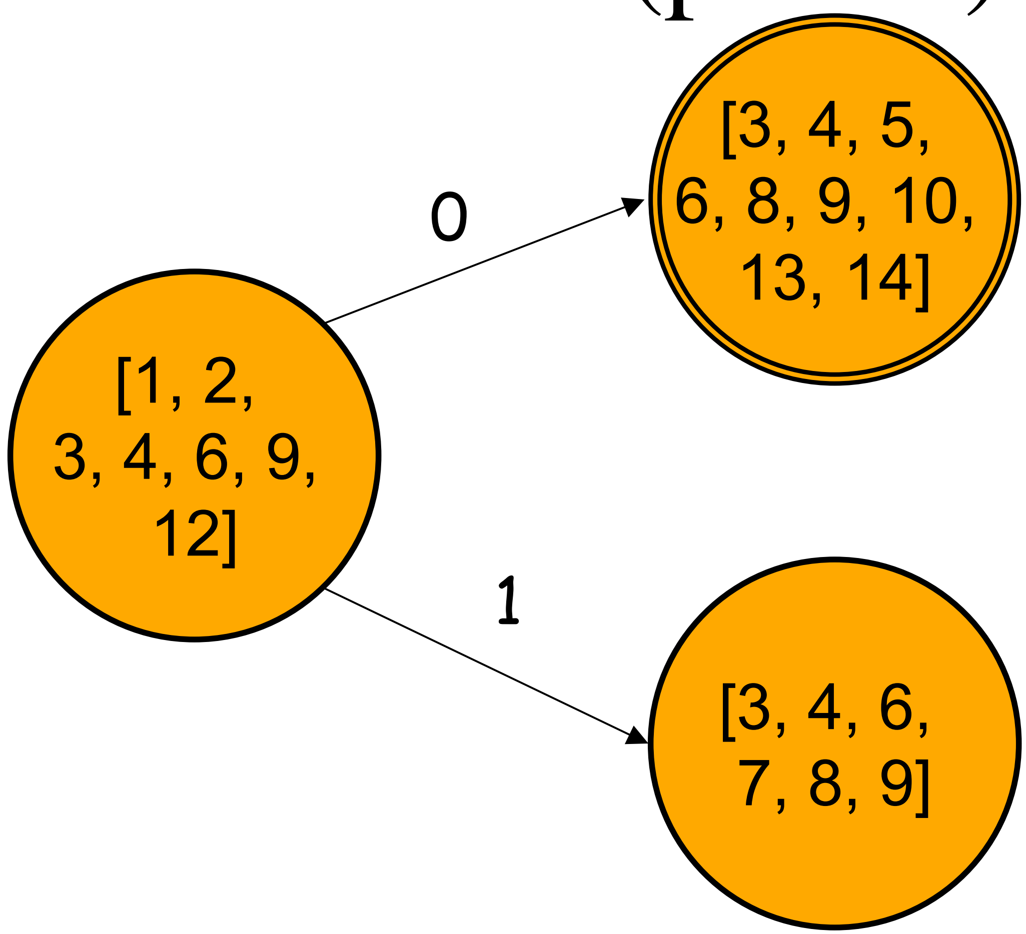
move(ε-*closure*(q_0), 1)



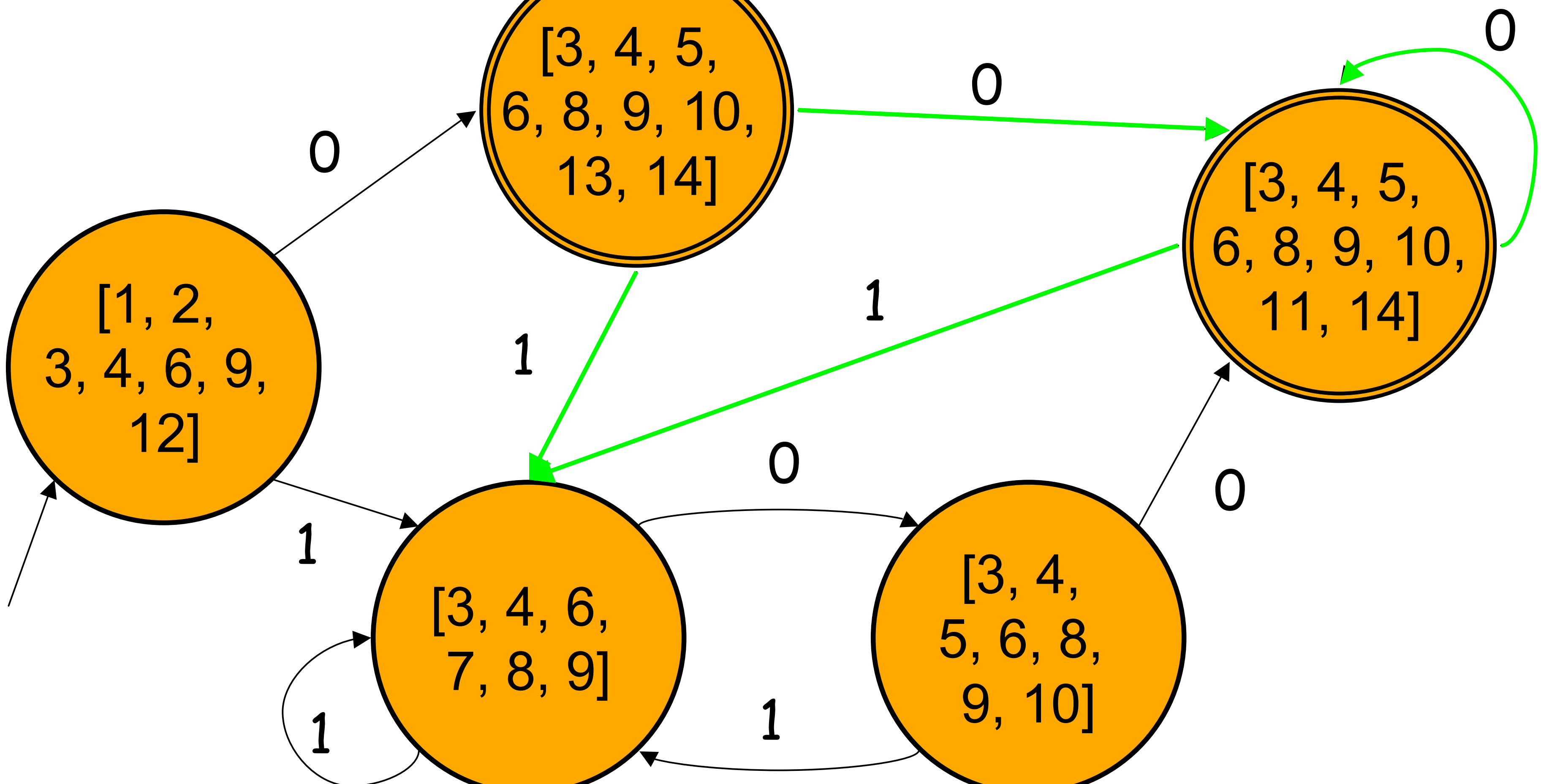
ε-*closure*(move(ε-*closure*(q_0), 1))



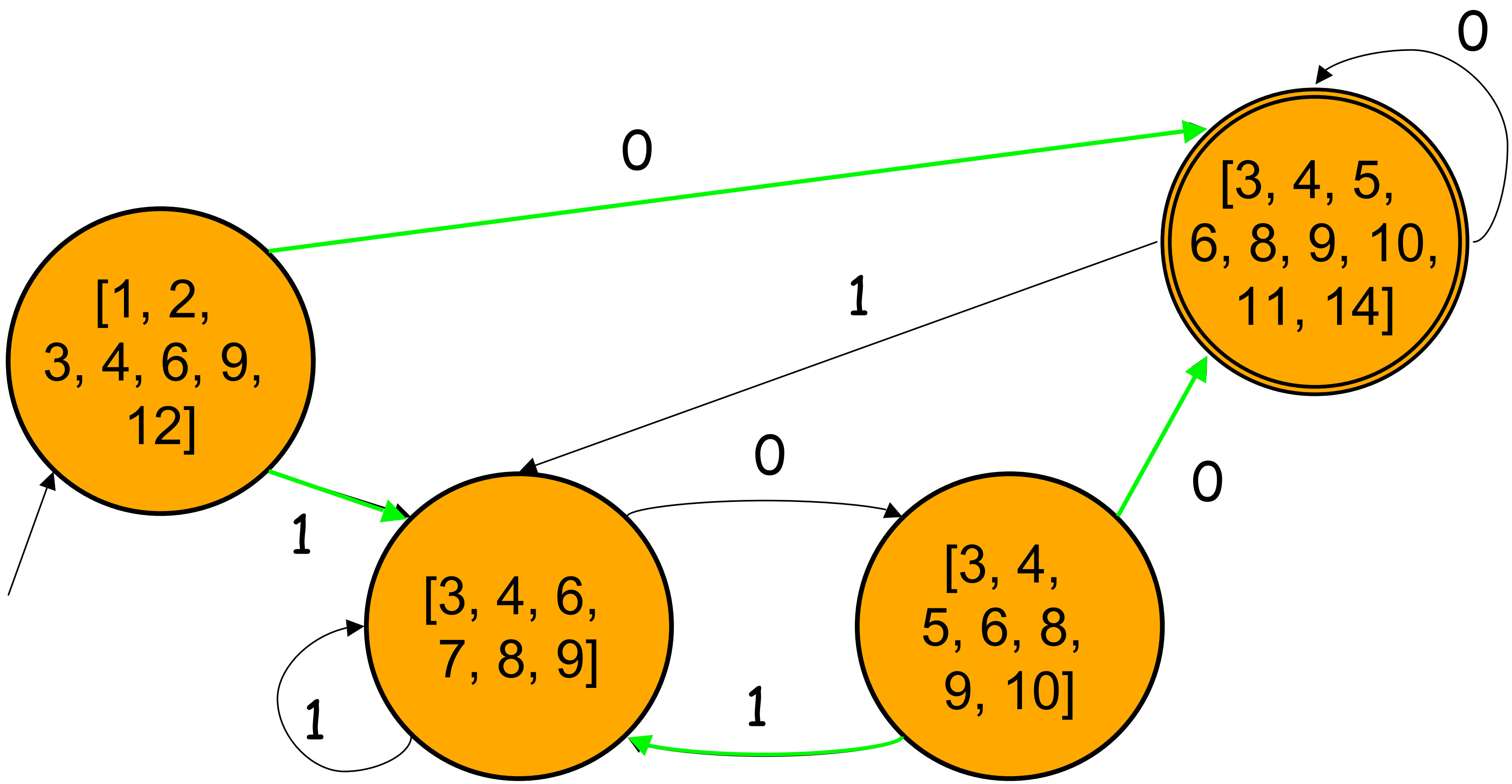
DFA (partial)



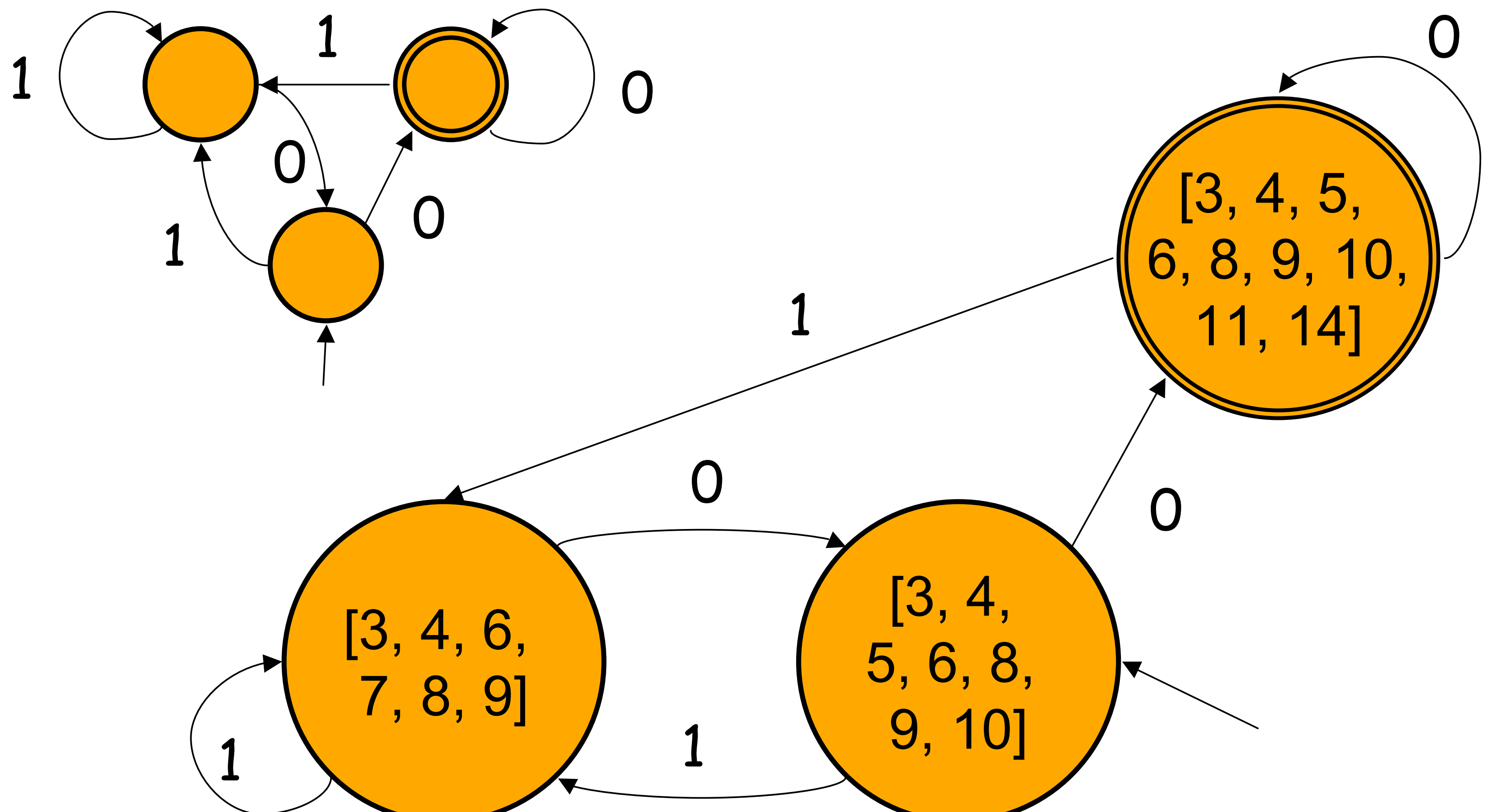
DFA for ((0|1)*00)|0



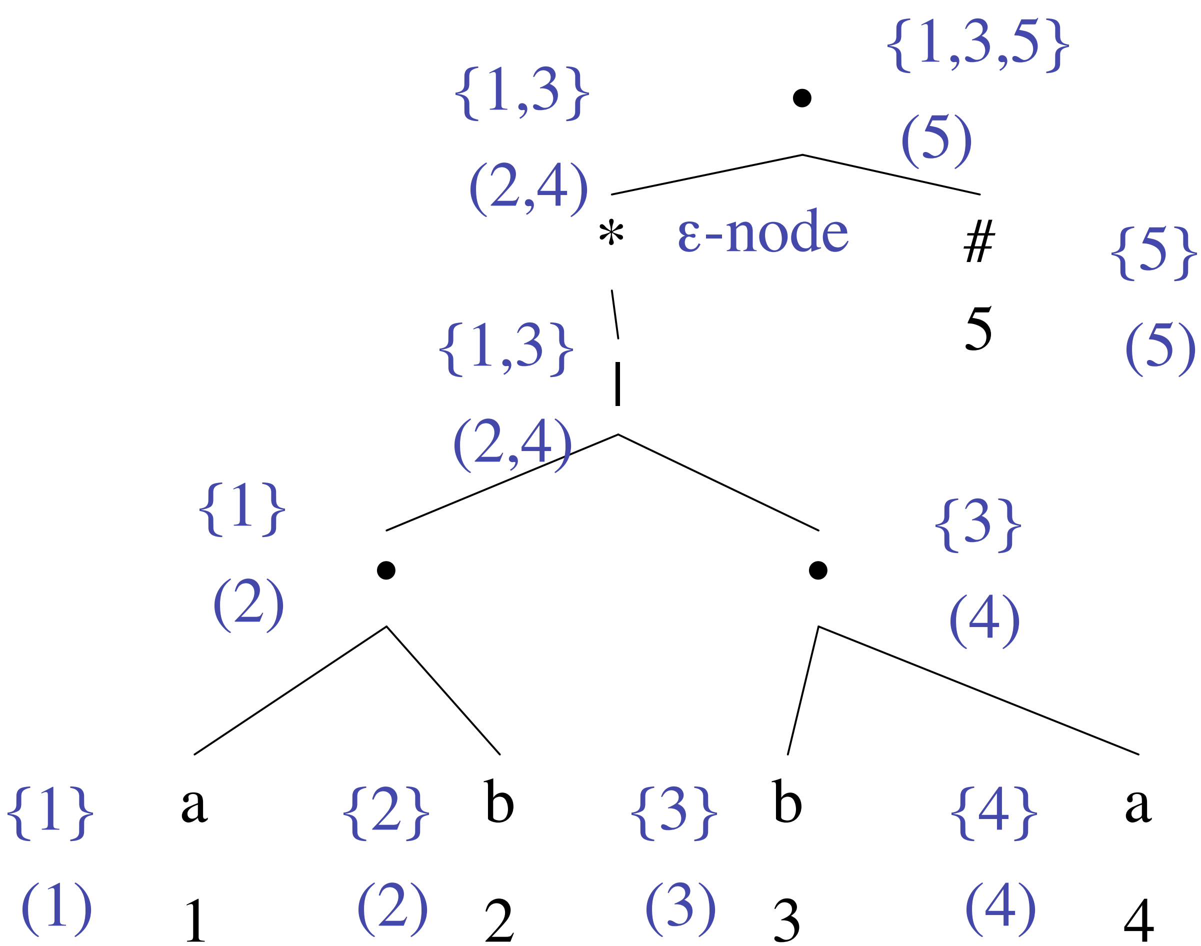
Minimization (I)



Minimization (II)



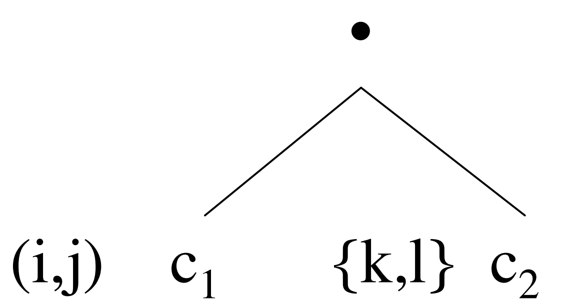
Regexp to DFA: `(ab|ba)*#`



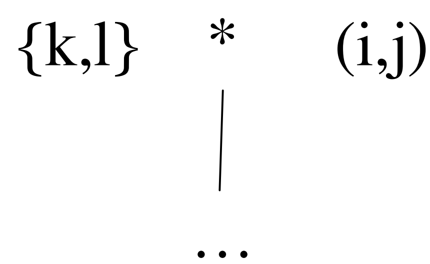
firstpos = {}
lastpos = ()

Regexp to DFA: *followpos*

- *followpos* tells us which positions can follow a position *k*
- There are two rules that use the *firstpos* {} and *lastpos* () information

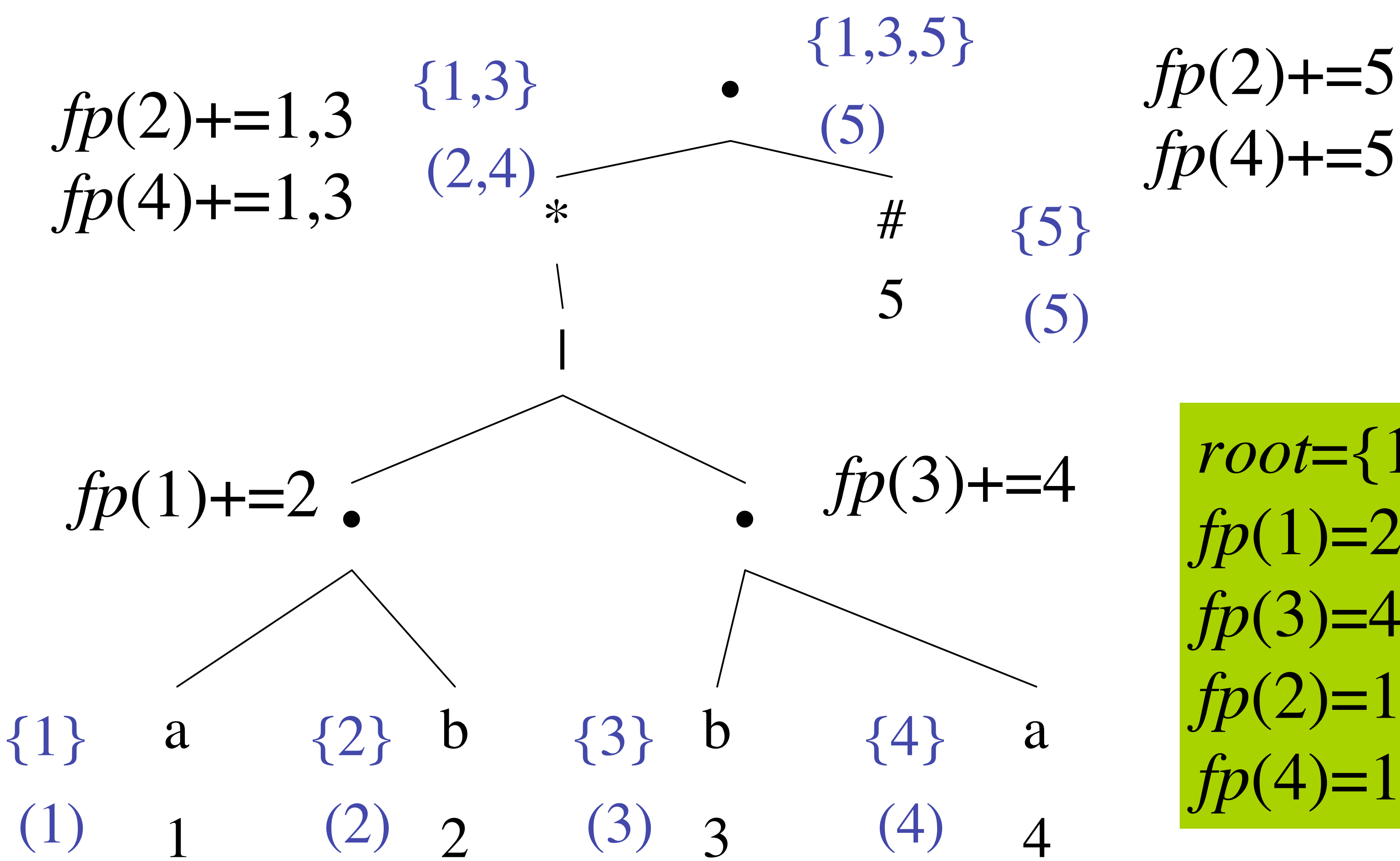


followpos(i)+=k,l
followpos(j)+=k,l



followpos(i)+=k,l
followpos(j)+=k,l

Regexp to DFA: `(ab|ba)*#`

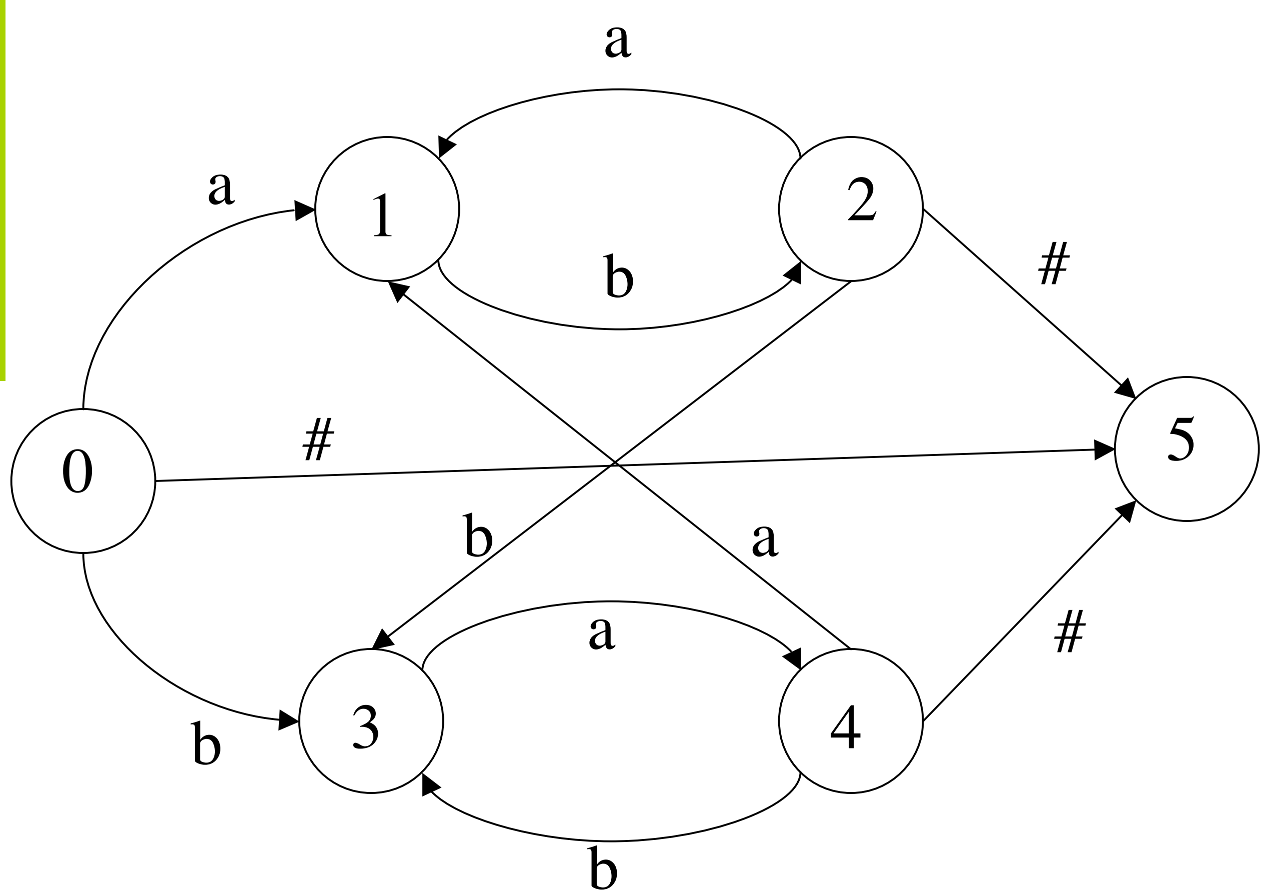


root={1,3,5}
fp(1)=2
fp(3)=4
fp(2)=1,3,5
fp(4)=1,3,5

Regexp to DFA: `(ab|ba)*#`

root={1,3,5}
fp(1)=2
fp(3)=4
fp(2)=1,3,5
fp(4)=1,3,5

1:a
2:b
3:b
4:a
5:#



Regexp to DFA: `(ab|ba)*#`

root={1,3,5}
fp(1)=2
fp(3)=4
fp(2)=1,3,5
fp(4)=1,3,5

{1,3,5} A

A: fp(1),a {2},a B,a

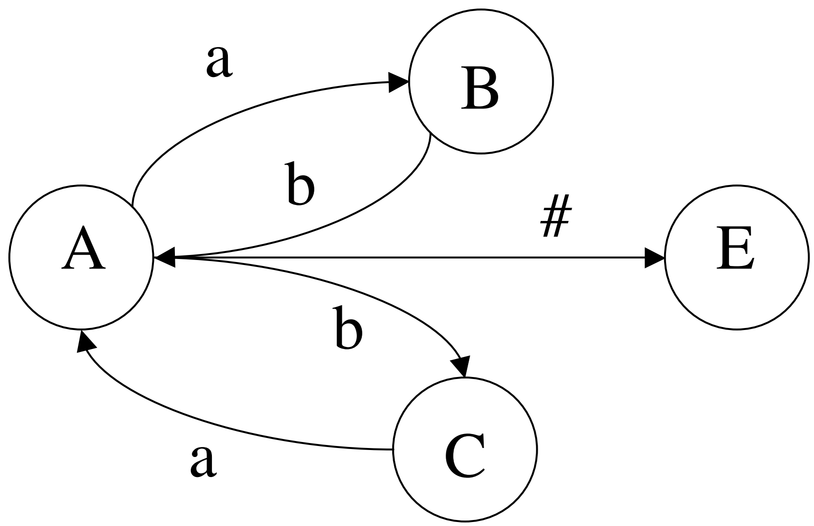
A: fp(3),b {4},b C,b

A: fp(5),# {},# E,#

B: fp(2),b {1,3,5},b A,b

C: fp(4),a {1,3,5},a A,a

1:a
2:b
3:b
4:a
5:#



Equivalence of Regexps

- (R|S)|T == R|(S|T) == R|S|T
- (RS)T == R(ST)
- (R|S) == (S|R)
- R*R* == (R*)* == R* == RR*| ε
- R** == R*
- (R|S)T = RT|ST
- R(S|T) == RS | RT
- (R|S)* == (R*S*)* == (R*S)*R* == (R*|S*)*
- RR* == R*R
- (RS)*R == R(SR)*
- R = R|R = R|ε

Equivalence of Regexps

- 0(10)*1|(01)*
- (01)(01)*|(01)*
- (01)(01)*|(01)(01)*|ε
- (01)(01)*|ε
- (01)*

- (RS)*R == R(SR)*
- RS == (RS)
- R* == RR*|**ε**
- R == R|R
- R* == RR*| **ε**

NFA vs. DFA in the wild

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

Lexical Analyzer using DFAs

- Each token is defined using a regexp r_i
- Merge all regexps into one big regexp
  - $R = (r_1 \mid r_2 \mid \ldots \mid r_n)$
- 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. <*print*> and not <*pr*>, <*int*>
- 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

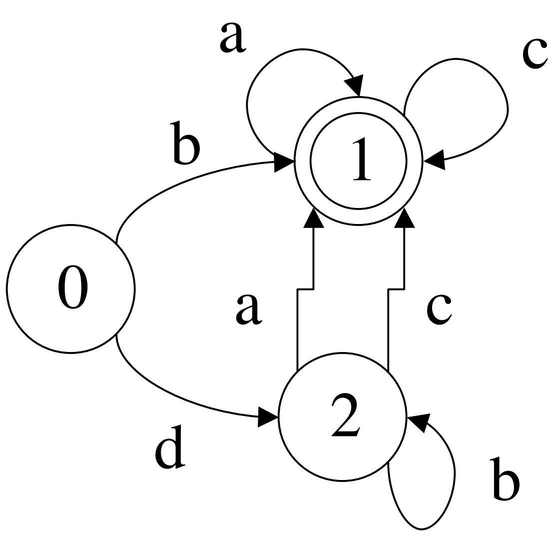
Lexical Analyzer using DFAs

- Alternative method:
  - Organize all the DFAs for each token in an ordered list
  - For input i_1, i_2, …, i_n run all DFAs until some reach a final state (pick the longest match for each DFA)
  - Pick the token for which some DFA could read the longest match in the input,
    - e.g. prefer DFA #8 over all others because it read the input until i_{30} and none of the other DFAs reached i_{30}
  - If two DFAs reach the same input character then pick the one that is listed first in the ordered list

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)

Implementing DFAs



	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

Implementing DFAs

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

			-	1	-	2		
					1	-	1	-
	1	2	1	-				
next	1	2	1	1	1	2	1	-
	0	1	2	3	4	5	6	7
check	2	2	2	0	1	0	1	-

base

0	2
1	4
2	0

nextstate(*s*, *x*) :

L := base[*s*] + *x*

return next[L] **if** check[L] **eq** *s*

Implementing DFAs

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

	-	1	-	2		
			1	-	1	-
-	2	-	-			
-	2	1	1	2	1	-
0	1	2	3	4	5	6

next

-	2	0	1	0	1	-

check

base

0	2
1	3
2	0

-
-
1

default

nextstate(s, x) :
  L := base[s] + x
  return next[L] **if** check[L] **eq** s
  else return *nextstate*(default[s], x)

Summary

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
  – Thompson's Rules
- NFA $\Rightarrow$ DFA
  – Subset constructions
- DFA $\Rightarrow$ minimal DFA
  – Minimization

$\Rightarrow$ Lexical Analyzer (multiple patterns)