Compiler Construction

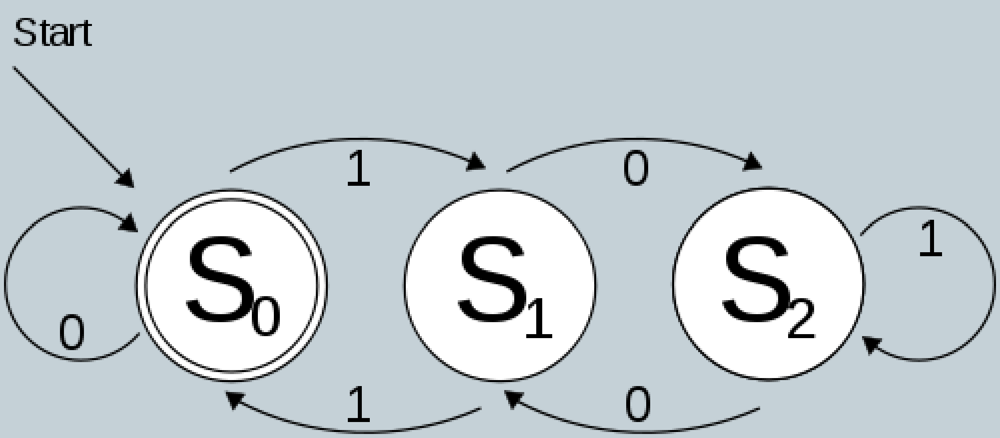
ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Algebra of Regular Expressions

AXIOM	DESCRIPTION
r | s = s | r	| is commutative
r | (s | t) = (r | s) | t	| is associative
(r s) t = r (s t)	concatenation is associative
r (s | t) = r s | r t (s | t) r = s r | t r	concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity element for concatenation
r* = (r | ϵ)*	relation between * and ϵ
r** = r*	* is idempotent

Automata & Language Theory

- Terminology
  - FSA = Finite State Automaton
    - A recognizer that takes an input string and determines whether it's a valid string of the language.



Regexp: (0 | 1(01*0)*1)*

Automata & Language Theory

- Deterministic FSA (DFA)
  - Has in each state at most one action for any given input symbol
- Non-Deterministic FSA (NFA)
  - May have more than one alternative action for the same input symbol per state.
  - Cannot utilize standard single character look-ahead algorithm!

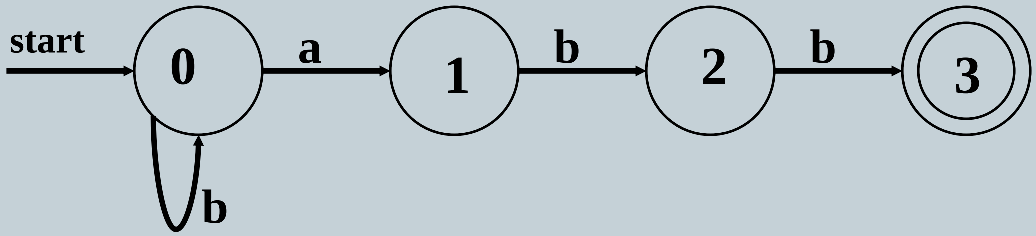
- Bottom Line
  - expressive power(NFA) == expressive power(DFA)
  - Conversion can be automated

Deterministic Finite Automata

A DFA is a mathematical model that consists of :

- **S, a set of *states***
- **Σ, the symbols of the input alphabet**
- ***move*, a transition function.**
  - ***move*(state, symbol) → state**
  - ***move* : $S \times \Sigma \rightarrow S$**
- **A state $s_0 \in S$, the start state**
- **$F \subseteq S$, a set of final or accepting states.**

DFA: Transition table representation of b*abb



$S = \{ 0, 1, 2, 3 \}$

$\Sigma = \{ a, b \}$

$s_0 = 0$

$F = \{ 3 \}$

Move:

input

state	a	b
0	1	0
1	--	2
2	--	3

DFA simulation algorithm

Since DFA move/transition tables don't have any alternative options, DFAs are easily simulated using the following algorithm.

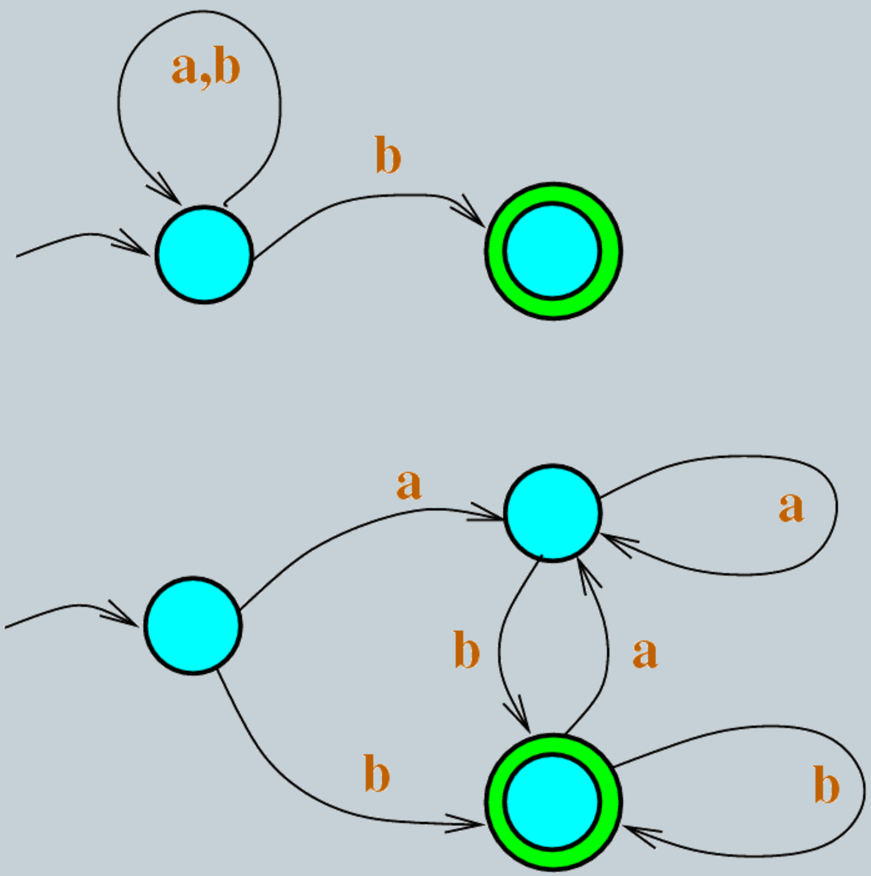
```
state ← s0;
lexeme ← ϵ;
trail ← ϵ;
c ← nextchar();
while c ≠ eof and c ∈ AcceptableCharacters(state) do
  append(trail, c);
  state ← move[state,c];  /* transition table */
  if state ∈ FinalStates then
    lexeme ← concat(lexeme, trail);
    trail ← ϵ;
  endif
  c ← nextchar();
end;
pushbackcharacter(c);
pushback(trail);
```

Non-Deterministic Finite Automata

An NFA is a mathematical model that consists of :

- **S, a set of states**
- **Σ, the symbols of the input alphabet**
- ***move*, a transition function.**
  - ***move*(state, symbol) $\rightarrow \mathcal{P}(S)$**
  - ***move* : $S \times \Sigma \rightarrow \mathcal{P}(S)$**
- **$s_0 \in S$, the start state**
- **$F \subseteq S$, a set of final or accepting states.**

Automata & Language Theory



$\Sigma = \{a,b\}$

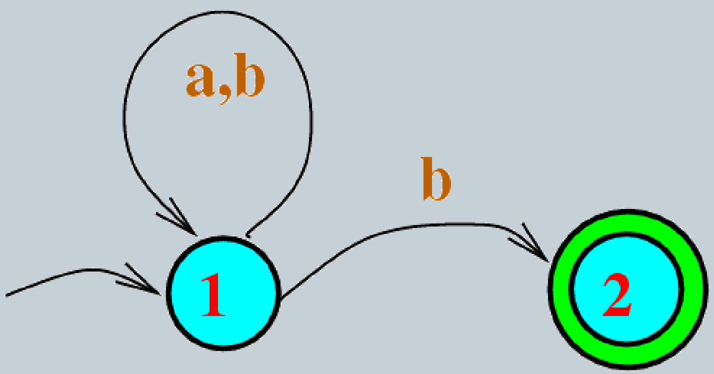
$L = (a+b)^*b$

NFA

DFA

NFA and DFA accepting the same language

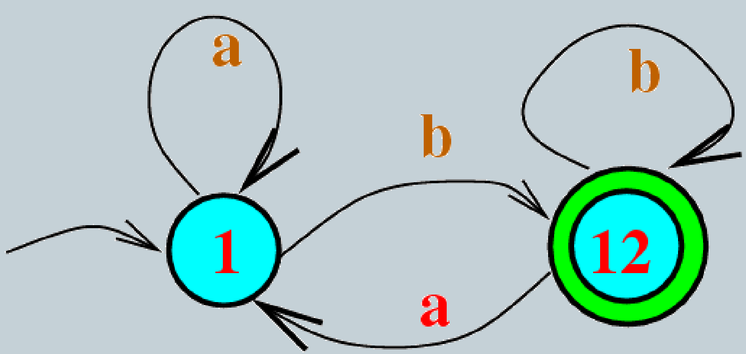
Automata & Language Theory



$\Sigma = \{a,b\}$

$L = (a+b)^*b$

NFA



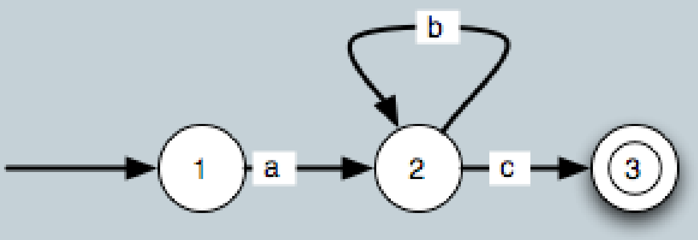
DFA

NFA and minimal DFA accepting the same language

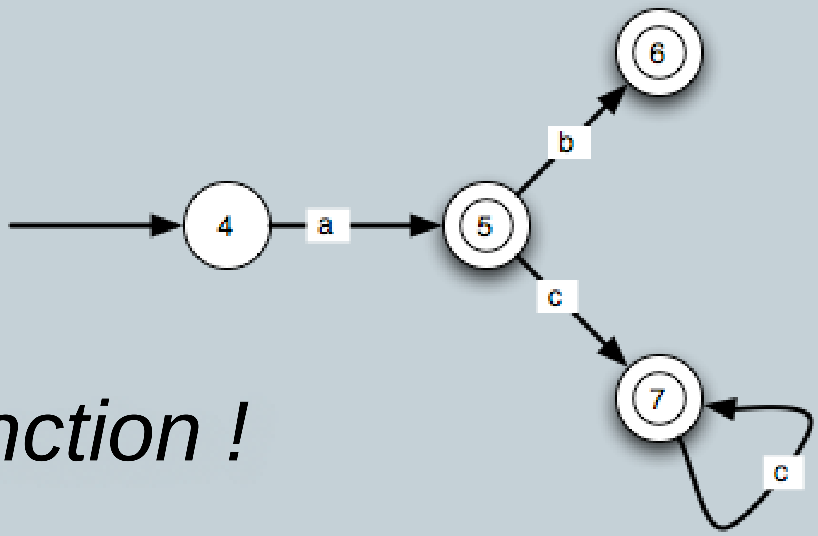
NFA Construction: (a (b*c)) | (a (b |c+)?)

- Automatic construction example

- a(b*c)



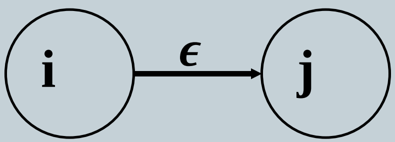
- a(b|c+)?



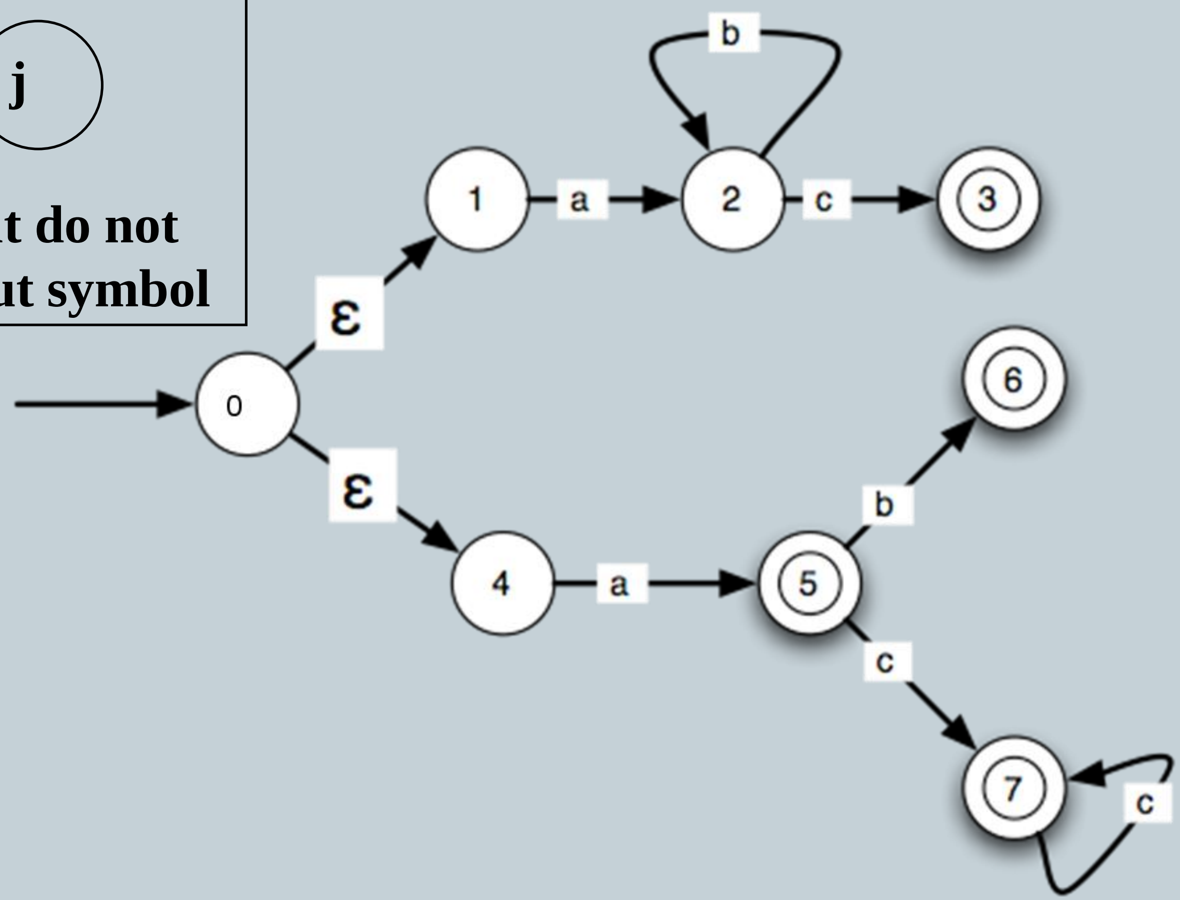
Build a *Disjunction !*

Epsilon-Transitions: (a (b*c)) | (a (b |c+)?)

ϵ move

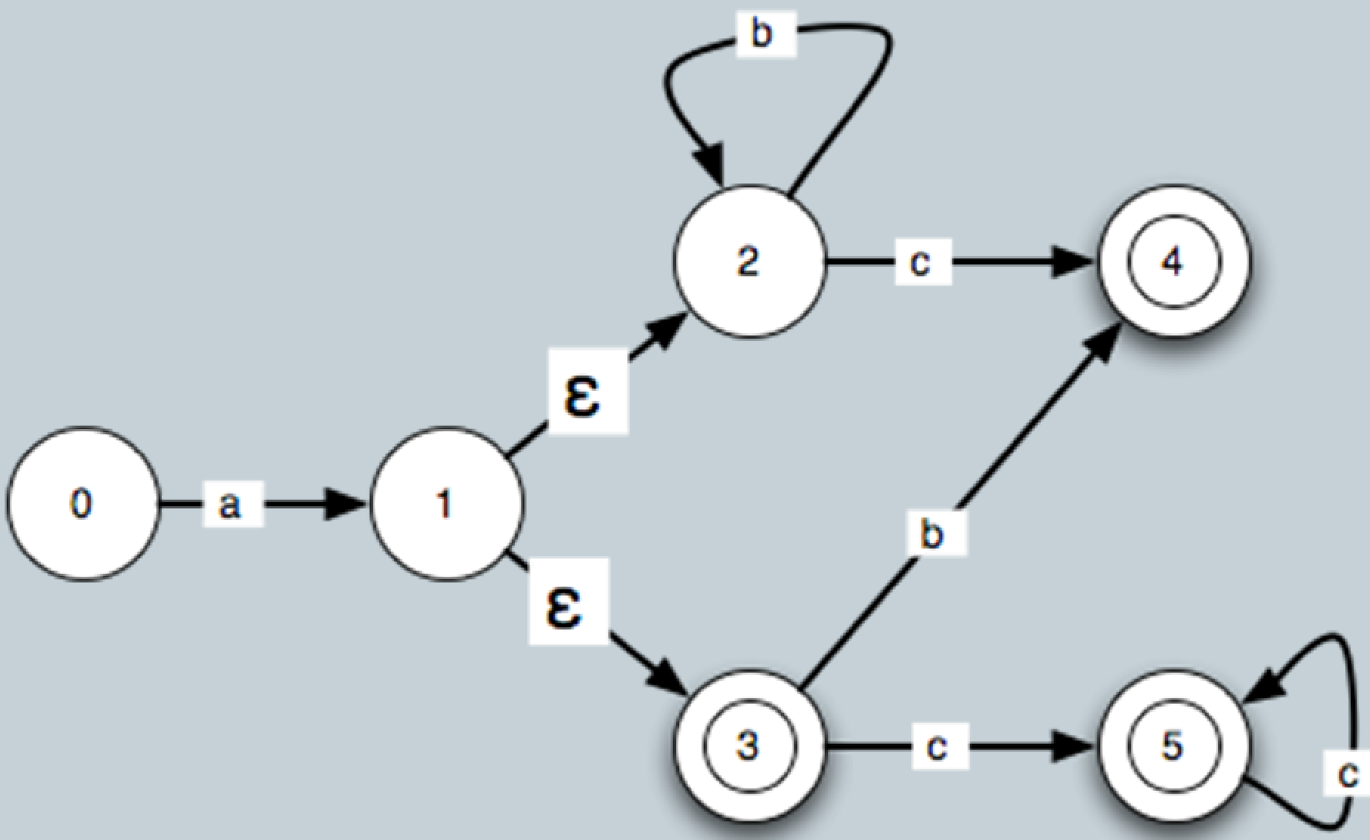


Switch state but do not accept any input symbol



Epsilon-Transitions: (a (b*c)) | (a (b |c+)?)

- A bit smarter is to factor out common prefixes

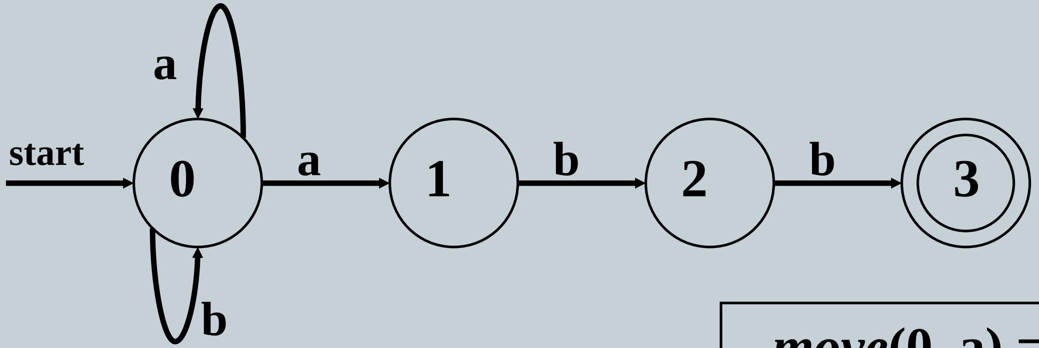


NFA- Regular Expressions & Lexers

- A lexical analyzer is a union of NFAs.
- Each NFA accepts a language token.

A Backtracking NFA?

- **Given an input string, we trace moves**
- **If no more (acceptable) input & in final state, ACCEPT**
- **If no more (acceptable) input & not in final state, backtrack if possible**



Input:
ababb

***move*(0, a) = 1**
***move*(1, b) = 2**
***move*(2, a) = ? (undefined)**

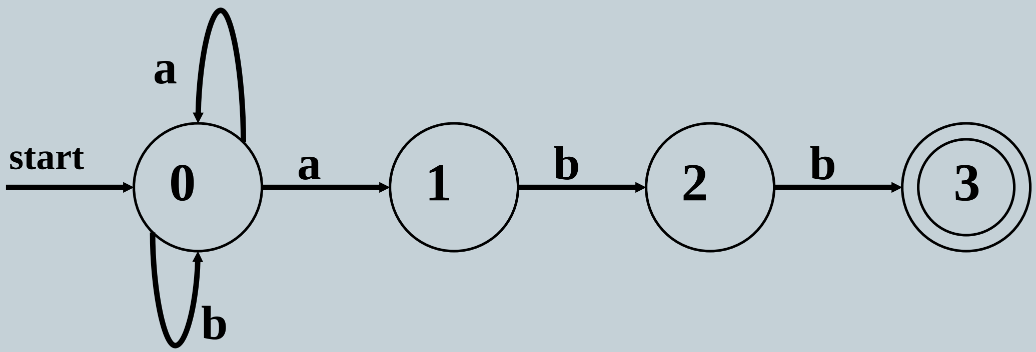
REJECT !

***move*(0, a) = 0**
***move*(0, b) = 0**
***move*(0, a) = 1**
***move*(1, b) = 2**
***move*(2, b) = 3**

ACCEPT !

Even worse...

- Most paths do not result in acceptance!



aaaabb is only accepted along the path :

$0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

BUT… it is not accepted along the paths:

$0 \rightarrow 1 \rightarrow$ fail

$0 \rightarrow 0 \rightarrow 1 \rightarrow$ fail

$0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow$ fail

The NFA “Problem”

- Valid input may need backtracking
  - Can be very inefficient (worst case, exponential time complexity)
  - Morever, it needs (substantial) input buffering
- Solution?
  - Build an equivalent DFA!

The DFA saves the day

- A DFA is an NFA with a few restrictions
  - No epsilon transitions
  - For every state s, there is at most one transition (s,x) for any symbol x in Σ

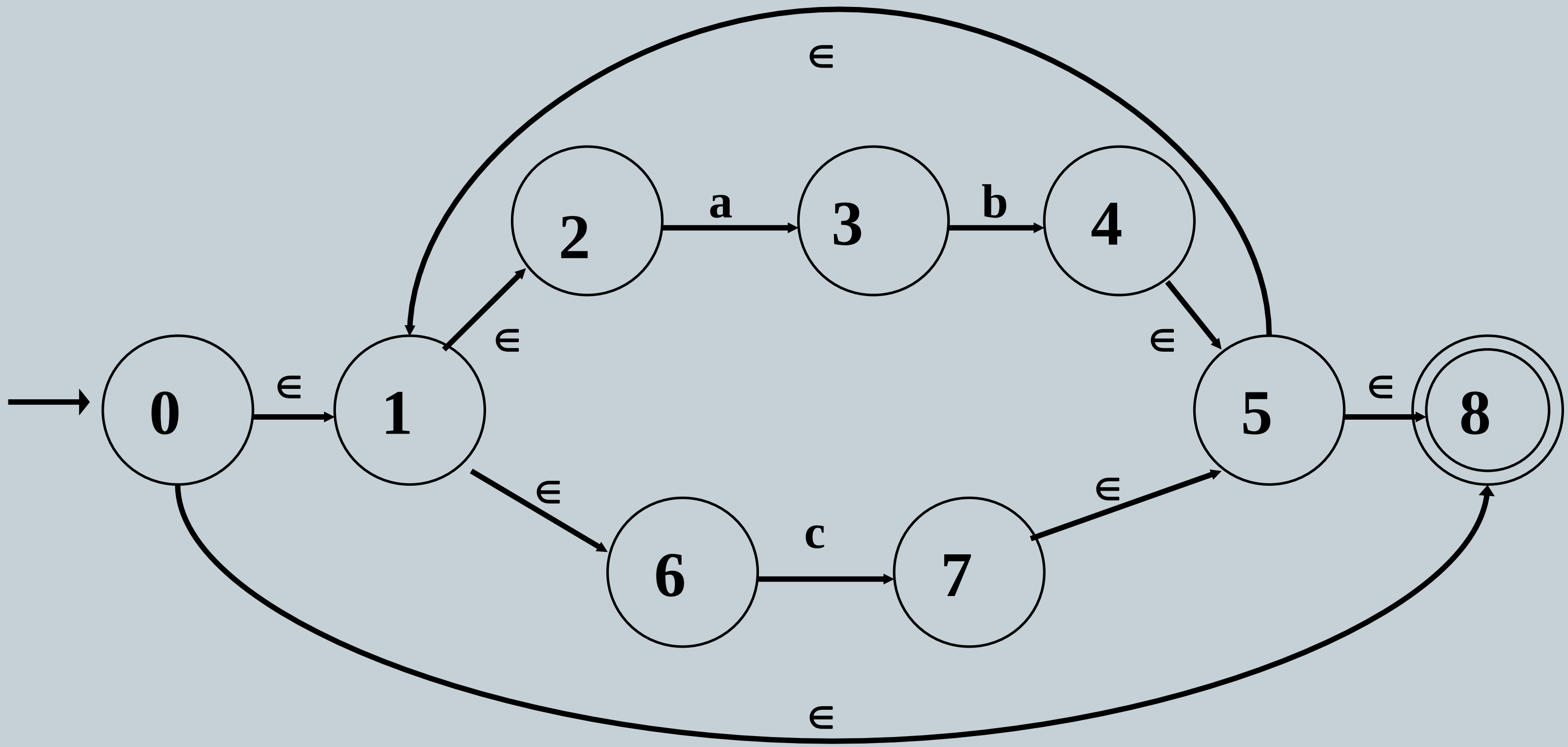
- Why do you want to do this?
  - Easy to implement a DFA with an algorithm!
    - The current character suffices to make decisions
    - No look-ahead required
    - No backtracking (stack/buffer) required
    - Much faster scanner

NFA vs. DFA

- NFA
  - smaller number of states than DFA
  - it requires a backtracking computation.
- DFA
  - larger number of states
  - it requires a *constant* computation for each input symbol.

- caveat – During the NFA=>DFA conversion, the number of states might explode
  - From N to 2^N states
  - Fortunately:
    - 2^N rarely occurs
    - DFA's can be minimized

NFA to DFA Conversion process



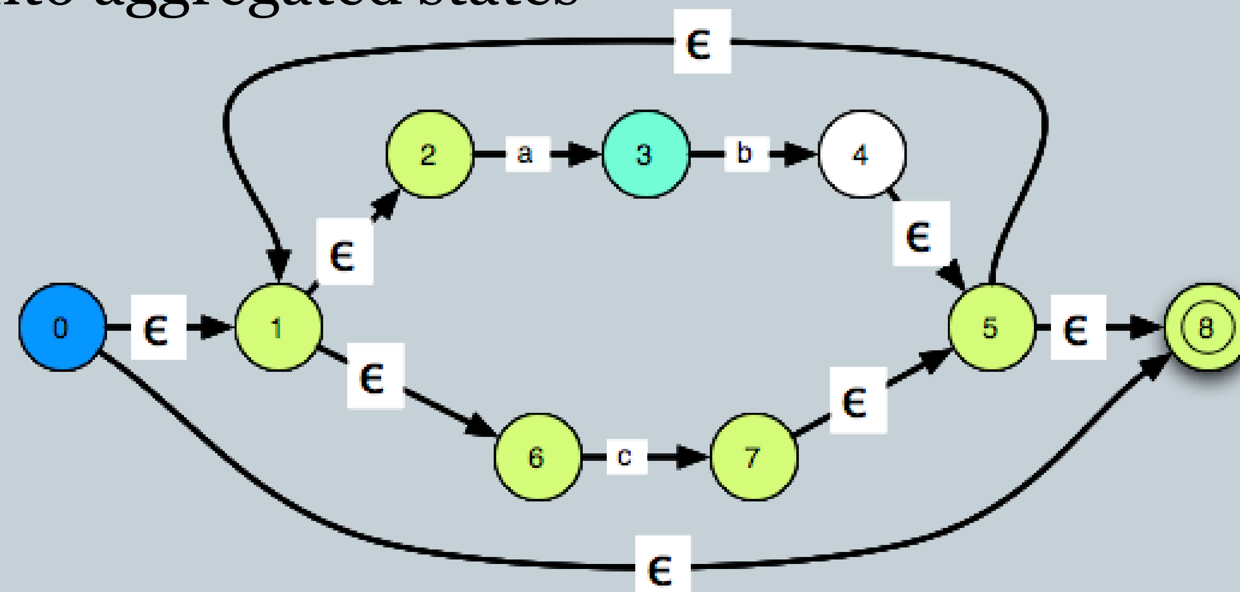
From State 0, where can we move without consuming any input ?

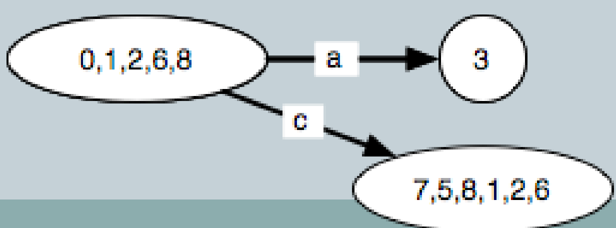
This forms a new state: {0,1,2,6,8}

Which transitions are defined for this new state?

NFA to DFA Conversion

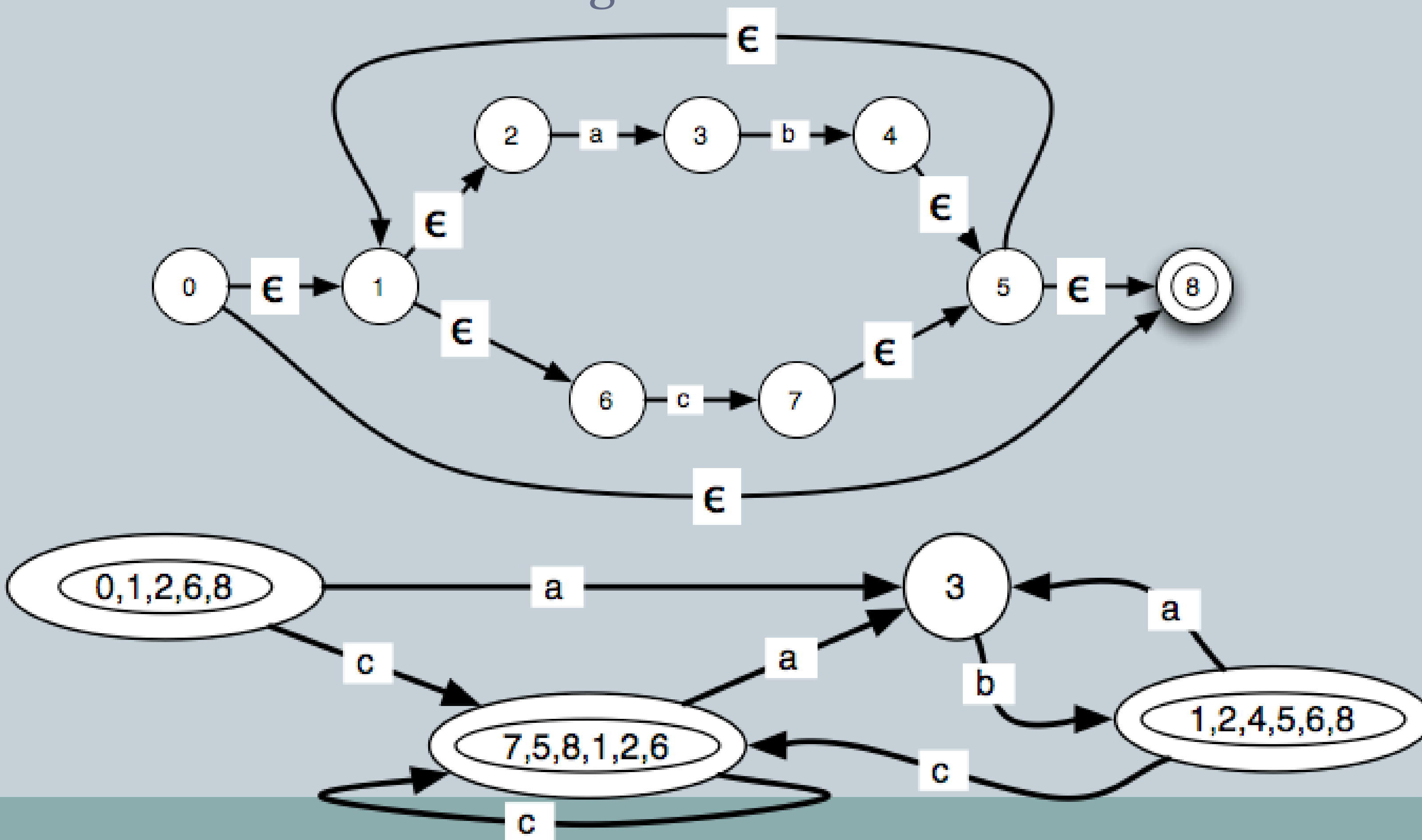
- Merge states that can be reached without consuming any input into aggregated states



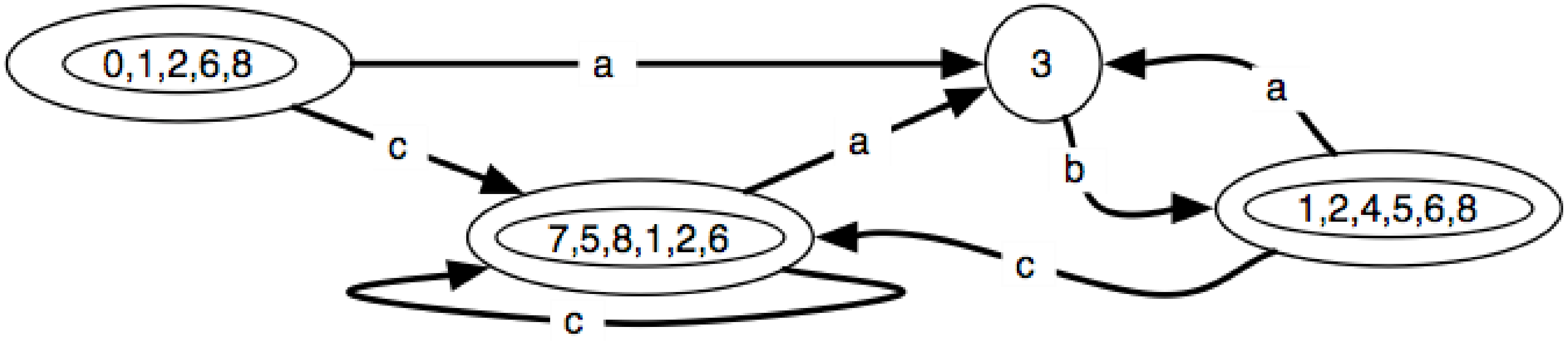


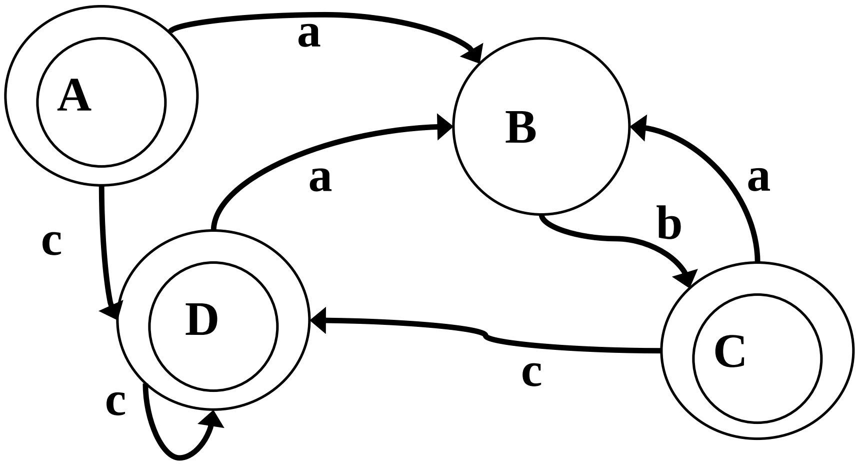
Final States

- An aggregated state is final
  - IFF at least one of its original NFA states was final



The Resulting DFA





NFA->DFA: Algorithm Concepts

NFA N = (S, Σ, s_0, F, MOVE)

These 3 operations are utilized by the conversion algorithm.

$\in$-Closure(s) : s $\in$S

: set of states in S that are reachable from s using $\in$-paths that originate from s.

$\in$-Closure of T : T $\subseteq$ S

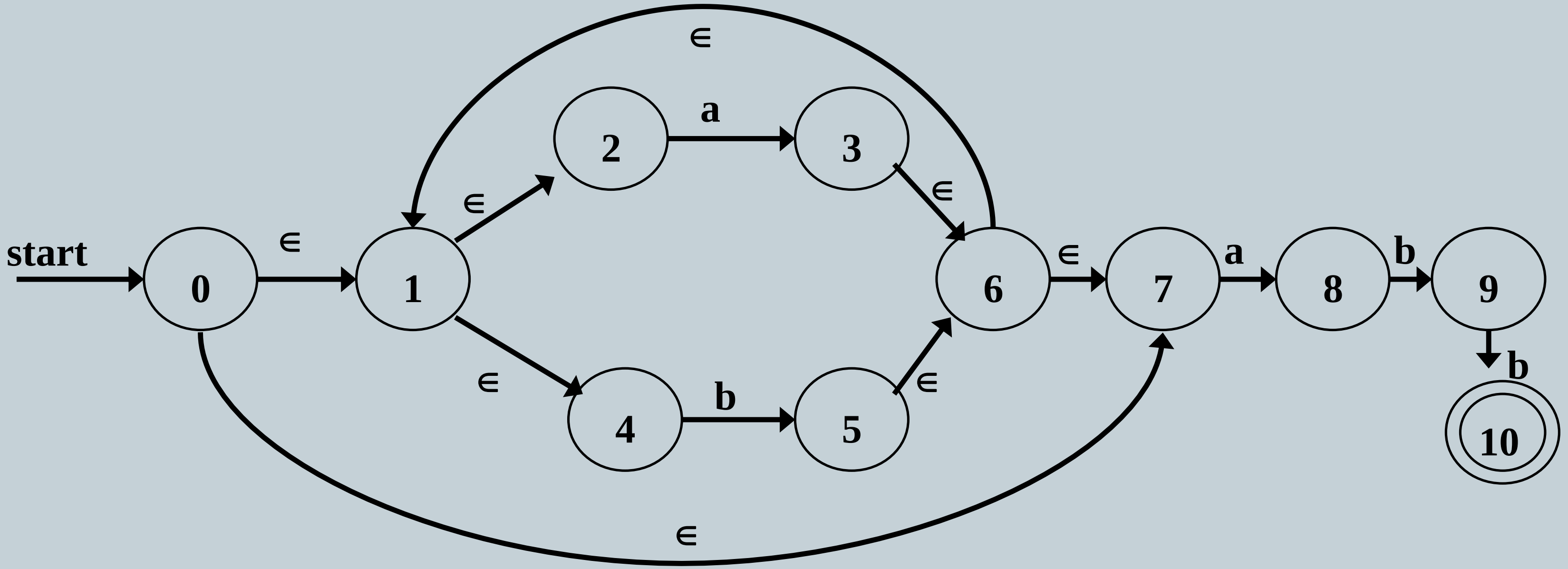
: NFA states reachable from any t $\in$ T using $\in$-paths only.

***move*(T,a) : T $\subseteq$ S, a$\in\Sigma$**

: Set of states to which there is a transition on input a from some t $\in$ T

Illustrating Conversion – An Example

Start with NFA: **(a | b)*abb**



First we calculate: ∈-closure(0) (i.e., state 0)

∈-closure(0) = {0, 1, 2, 4, 7} (states reachable from 0 via ∈-paths)

Let A={0, 1, 2, 4, 7} be a state of the new DFA

Conversion Example – continued (1)

2nd , we calculate : a : $\in$-closure(*move*(A,a)) and
b : $\in$-closure(*move*(A,b))

<u>a</u> : $\in$-closure(*move*(A,a)) = $\in$-closure(*move*({0,1,2,4,7},a))= $\in$-closure({3,8}) (since *move*(2,a)=3 and *move*(7,a)=8)

We compute: $\in$-closure({3,8}) = {1,2,3,4,6,7,8}
(since $3 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by $\in$-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : $\in$-closure(*move*(A,b)) = $\in$-closure({5}) (since *move*(4,b)=5)

We compute: $\in$-closure({5}) = {1,2,4,5,6,7}
(since $5 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by $\in$-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

Conversion Example – continued (2)

3rd , we calculate transitions for state B on {a,b}

<u>a</u> : ∈-closure(*move(*B,a)) = ∈-closure(*move(*{1,2,3,4,6,7,8},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[B,a] = B.

<u>b</u> : ∈-closure(*move(*B,b)) = ∈-closure(*move(*{1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

<u>a</u> : ∈-closure(*move(*C,a)) = ∈-closure(*move(*{1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

<u>b</u> : ∈-closure(*move(C*,b)) = ∈-closure(*move(*{1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

Conversion Example – continued (3)

5^{th} , we calculate for state D on {a,b}

<u>a</u> : ∈-closure(*move(*D,a)) = ∈-closure(*move(*{1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B
Define Dtran[D,a] = B.

<u>b</u> : ∈-closure(*move(*D,b)) = ∈-closure(*move(*{1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E
Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

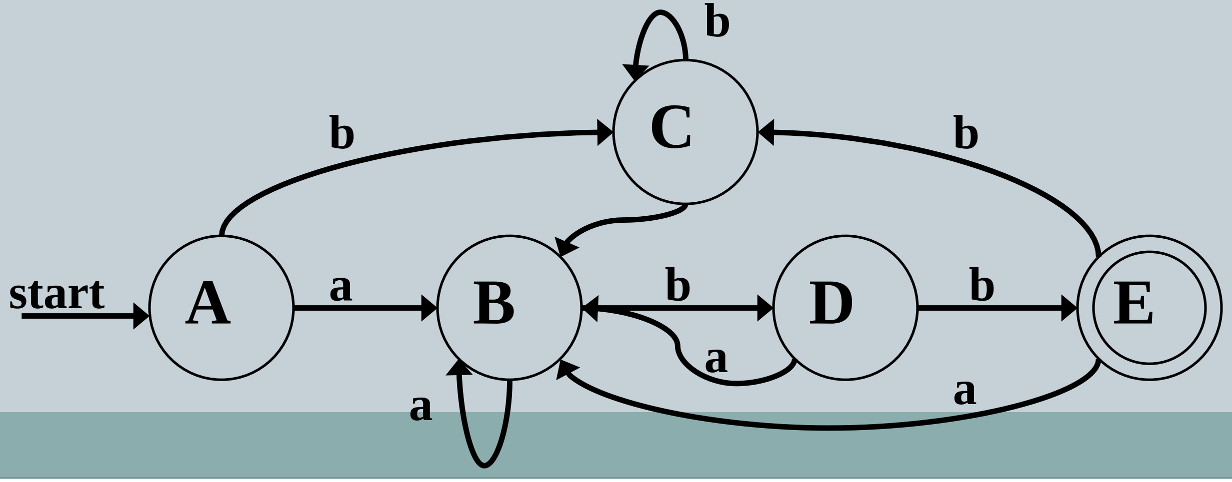
<u>a</u> : ∈-closure(*move(*E,a)) = ∈-closure(*move(*{1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B
Define Dtran[E,a] = B.

<u>b</u> : ∈-closure(*move(E*,b)) = ∈-closure(*move(*{1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C
Define Dtran[E,b] = C.

Conversion Example – continued (4)

This calculation yields the following transition table for the DFA:

State	Input Symbol a	Input Symbol b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Algorithm For Subset Construction

initially, $\in$-closure(s_0) is only (unmarked) state in Dstates;

```
while there is unmarked state T in Dstates do begin
    mark T;
    for each input symbol a do begin
        U := ∈-closure(move(T,a));
        if U is not in Dstates then
            add U as an unmarked state to Dstates;
        Dtran[T,a] := U
    end
end
```

Example: building a lexer

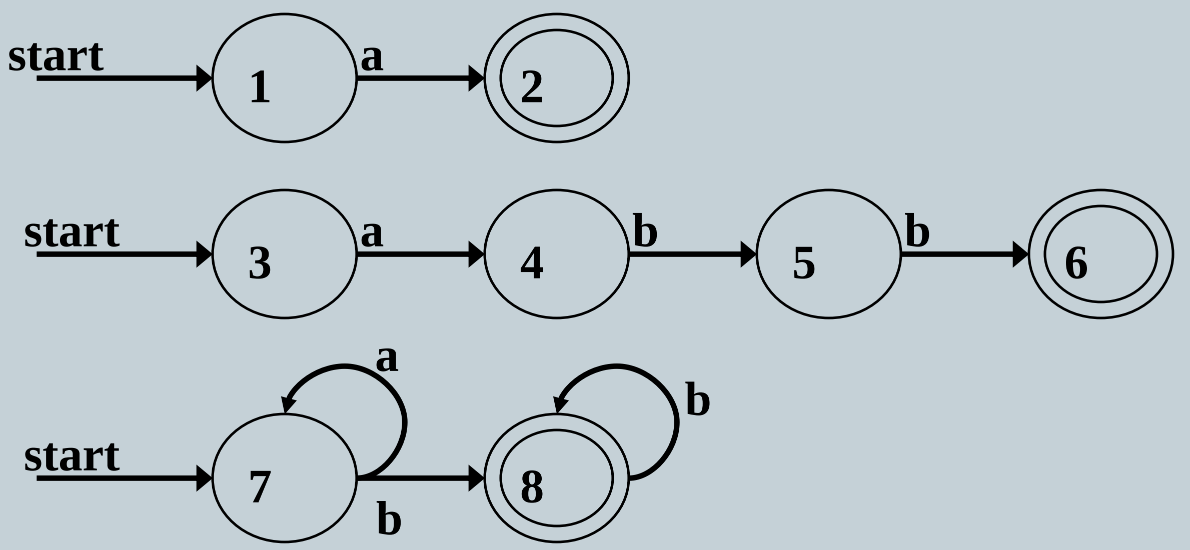
Let : **a**

abb

a*b+

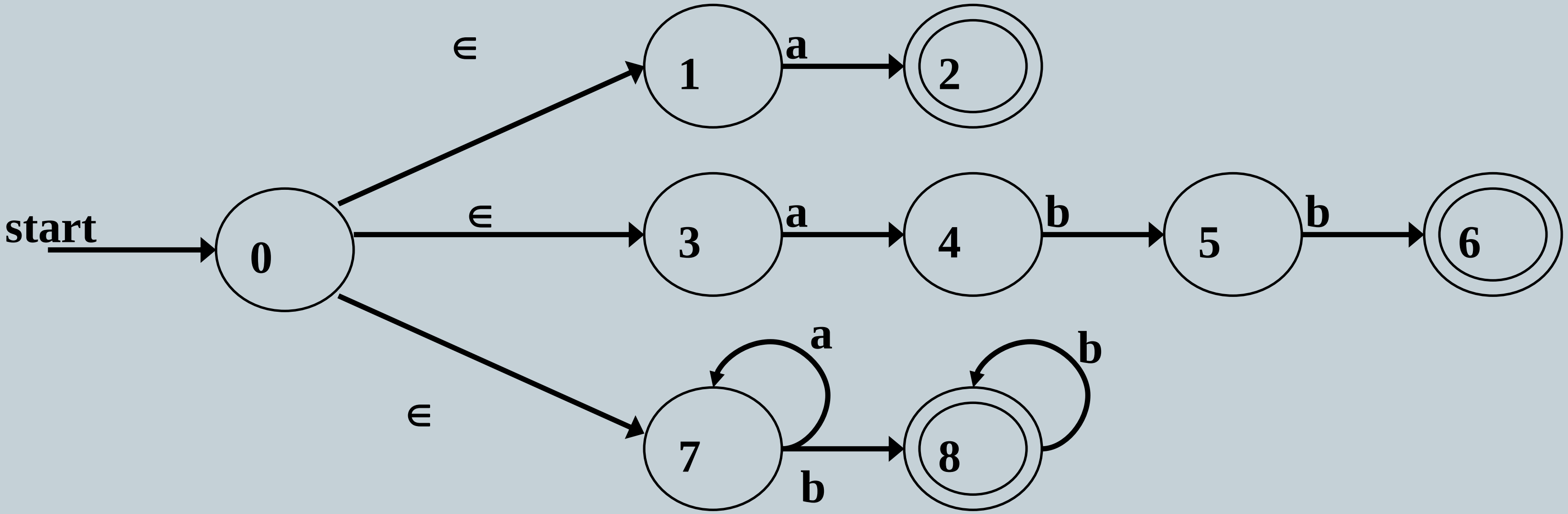
3 patterns

NFA's :



Example – continued(1)

Combined NFA :



Construct DFA : (It has 6 states)

{0,1,3,7}, {2,4,7}, {5,8}, {6,8}, {7}, {8}

Example – continued(2)

NFA->DFA for this example:

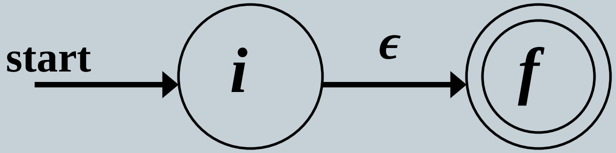
	Input Symbol			
STATE	a	b	Pattern	
{0,1,3,7}	{2,4,7}	{8}	none	
{2,4,7}	{7}	{5,8}	a	
{8}	-	{8}	a^*b^+	
{7}	{7}	{8}	none	
{5,8}	-	{6,8}	a^*b^+	
{6,8}	-	{8}	abb|a*b+	

Regular Expression to NFA Construction

- We now focus on transforming a Reg. Expr. to an NFA
- This construction allows us to take:
  - Regular Expressions (which describe tokens)
  - to an NFA
  - to a DFA (using the power set construction algorithm)
- The construction process is component wise
  - Construction is an example of syntax-directed translation

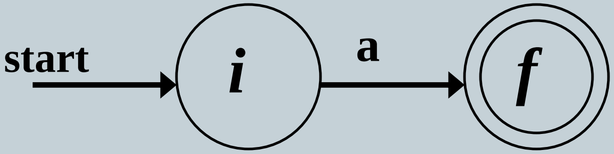
Construction Algorithm : R.E. → NFA

1. For the regular expression ϵ, construct



L(ϵ)

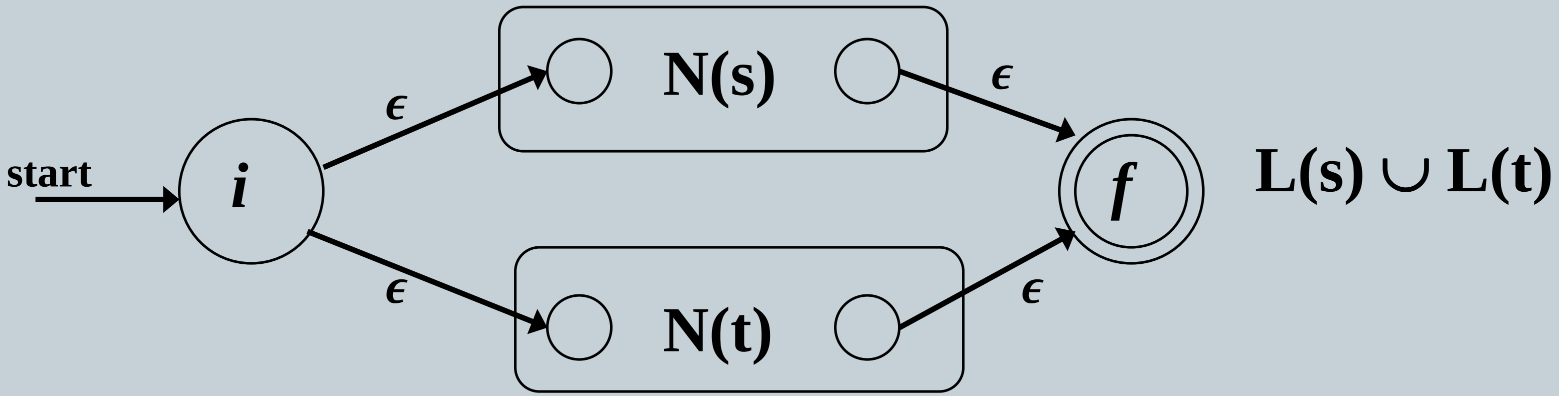
2. For the regular expression a $\in \Sigma$, construct



L(a)

Construction Algorithm : R.E. → NFA

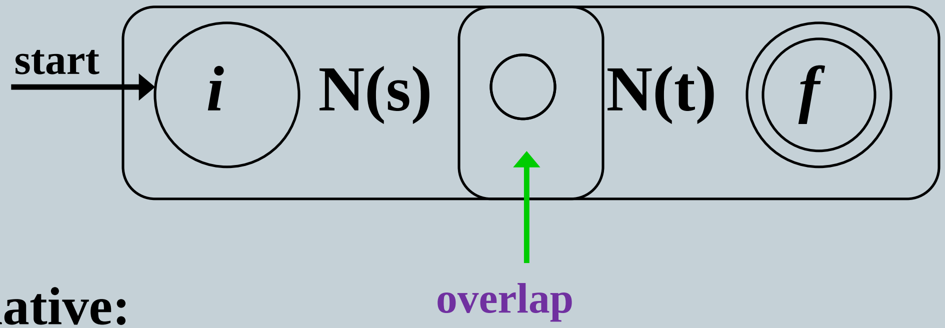
3. If s, t are regular expressions, N(s), N(t) their NFAs then s|t has NFA:



where *i* and *f* are new start / final states, and ϵ-moves are introduced from *i* to the old start states of N(s) and N(t) as well as from all of their final states to *f*.

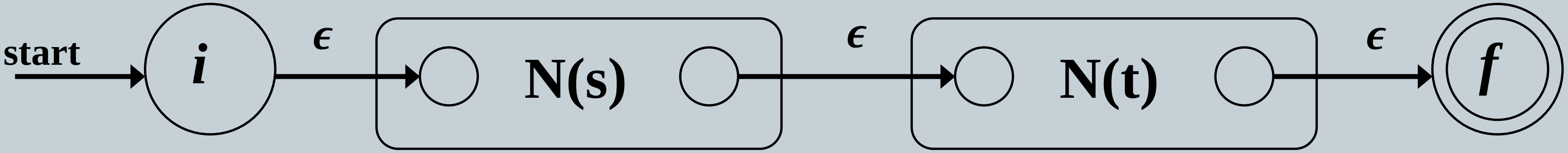
Construction Algorithm : R.E. → NFA

4. If s, t are regular expressions, N(s), N(t) their NFAs then s.t (concatenation) has NFA:



L(s) L(t)

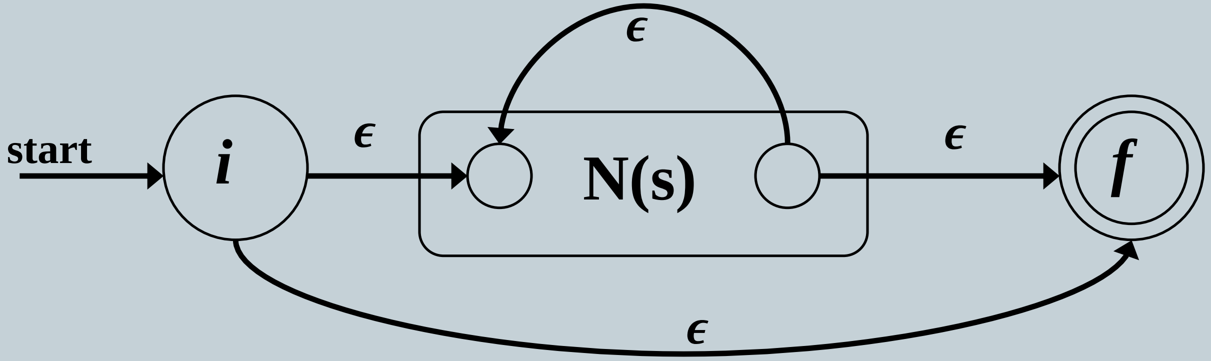
Alternative:



where *i* is the start state of N(s) (or new for the alternative) and *f* is the final state of N(t) (or new). Overlap maps final state of N(s) to start state of N(t).

Construction Algorithm : R.E. → NFA

5. If s is a regular expressions, N(s) its NFA, then s* (Kleene star) has NFA:



where : *i* is a new start state and *f* is a new final state

ϵ-move *i* to *f* (to accept null string)

ϵ-moves *i* to old start, old final(s) to *f*

ϵ-move old final to old start (loop)

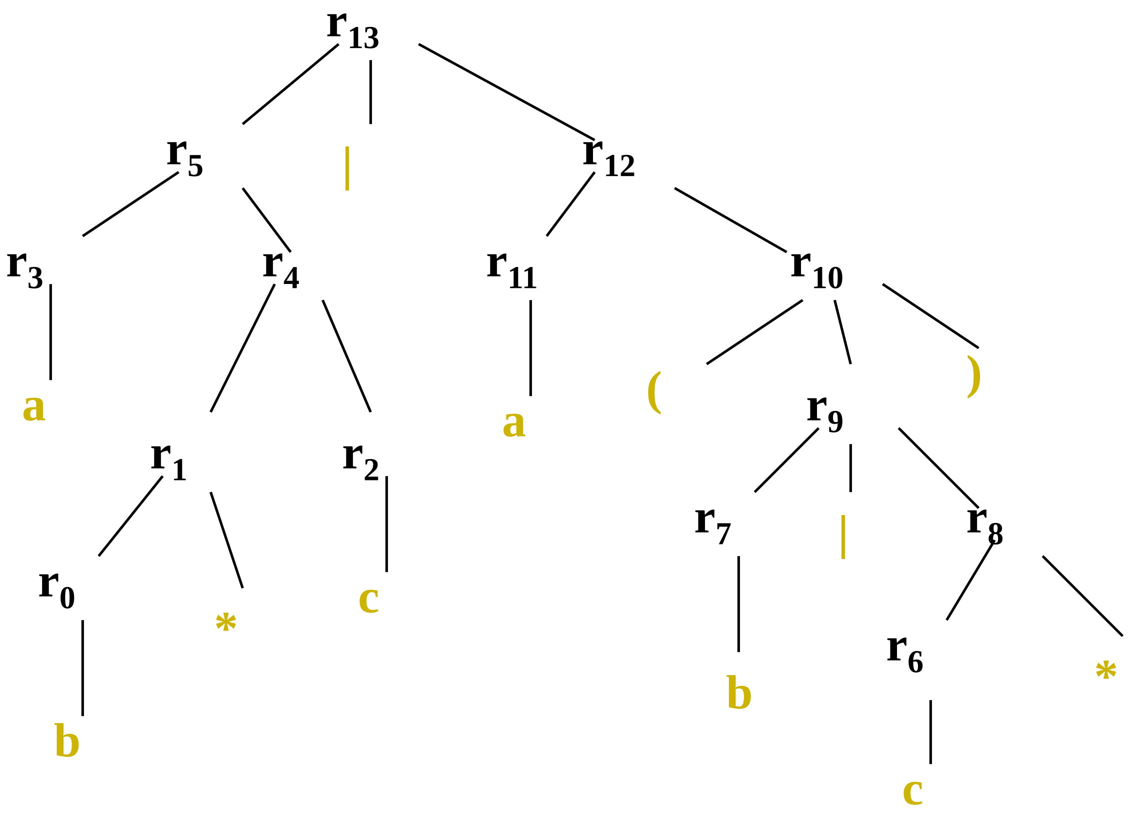
Properties of Construction

Let r be a regular expression, with NFA N(r), then

1. N(r) has at most 2*(#symbols + #connectives of r) states
2. N(r) has exactly one start and one accepting state
3. BEWARE to assign unique names to all states !

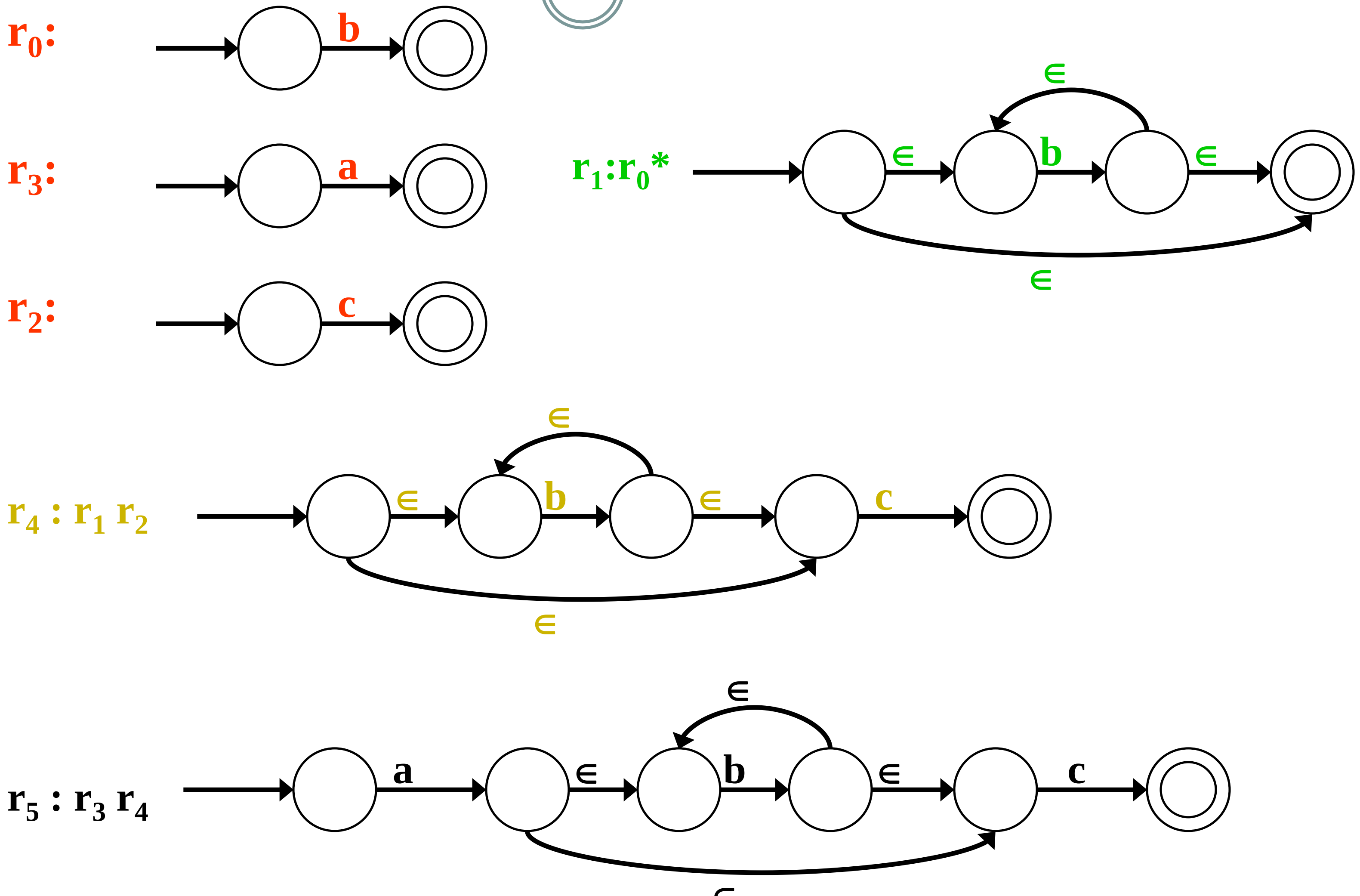
Detailed Example: (ab*c) | (a(b|c*))

Parse Tree for this regular expression:

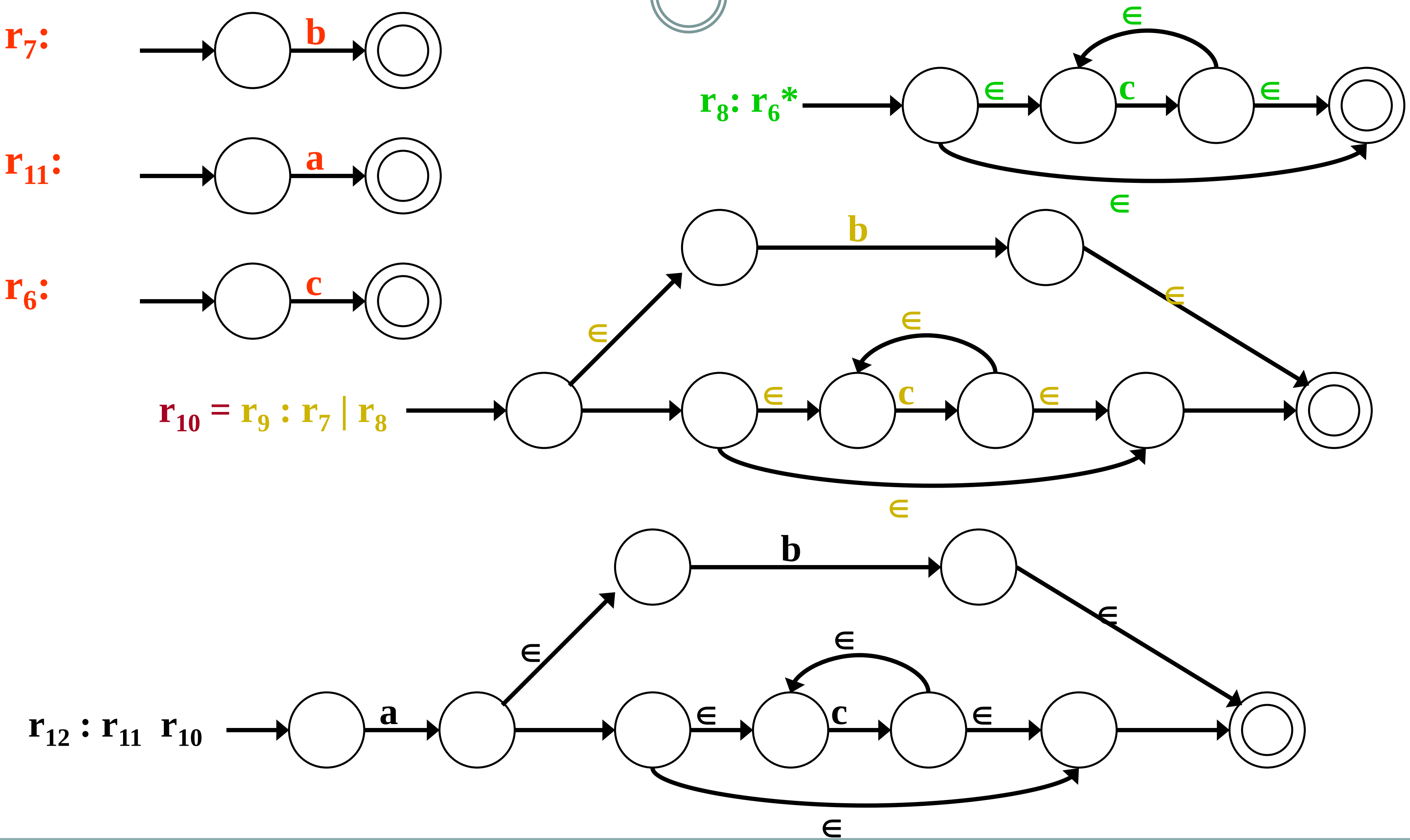


What is the NFA? Let's construct it !

Detailed Example – (ab*c) | (a(b|c*))

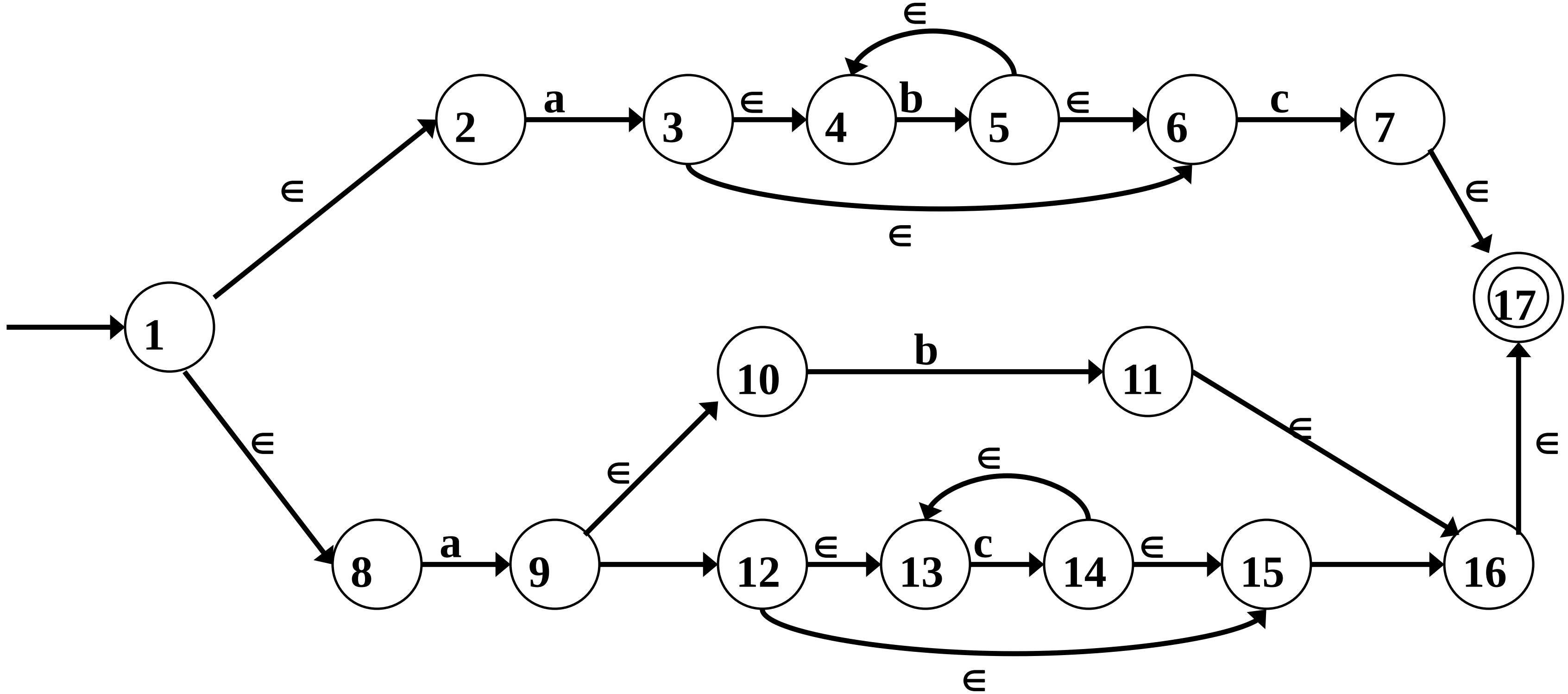


Detailed Example – (ab*c) | (a(b|c*))



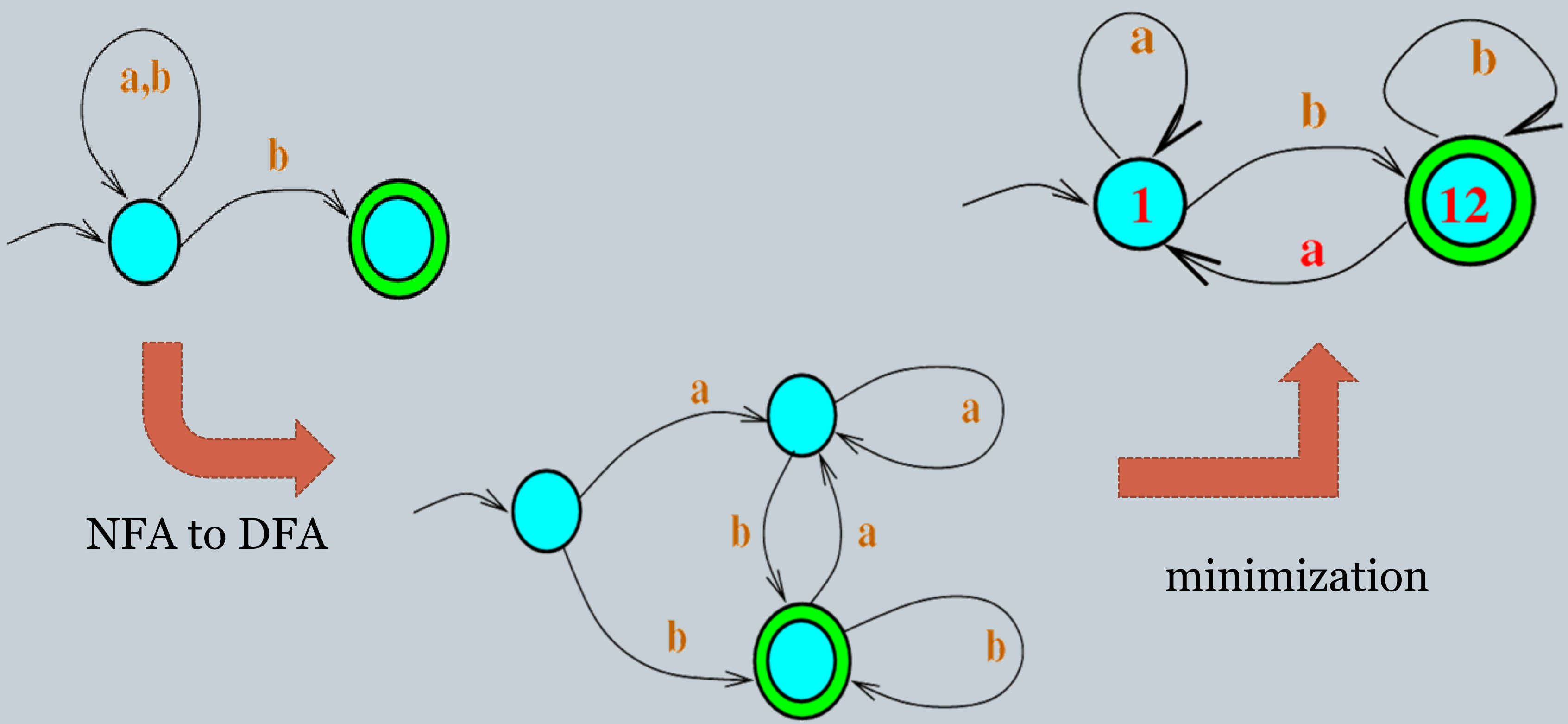
Detailed Example: (ab*c) | (a(b|c*)) Final Step

$\mathbf{r_{13} : r_5 \mid r_{12}}$



Minimizing the number of DFA states

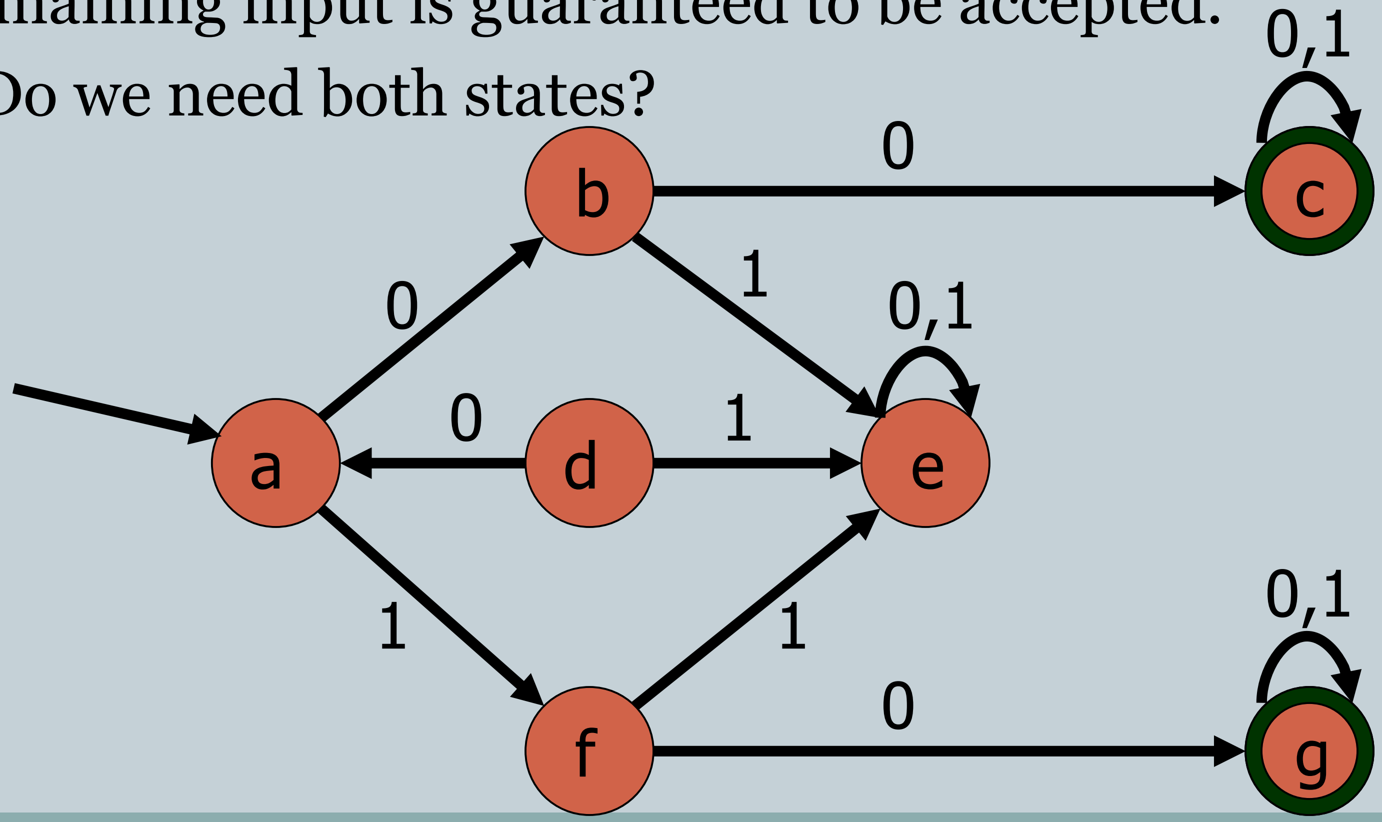
- A DFA is minimal if no DFA with fewer states does the same task



Minimizing the number of DFA states

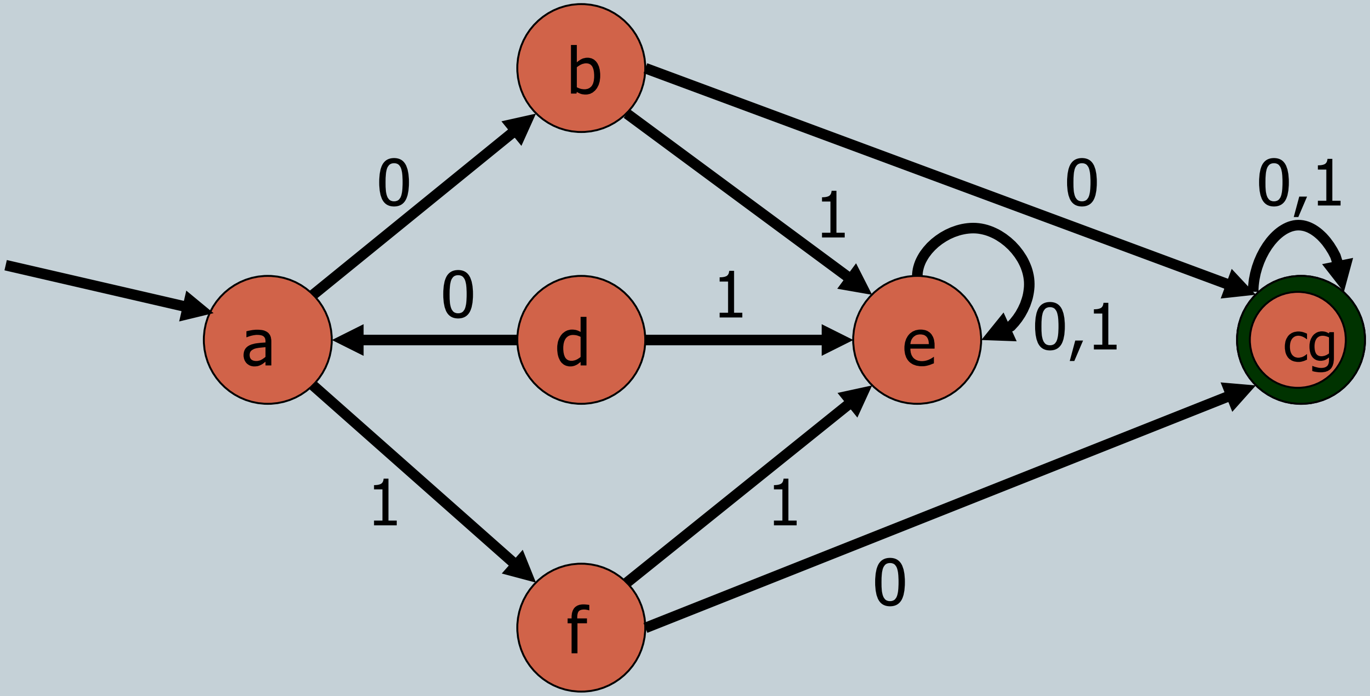
In the following DFA with alphabet {0,1}, consider the accepting states c and g. Once, we reach either of them, remaining input is guaranteed to be accepted.

Q: Do we need both states?



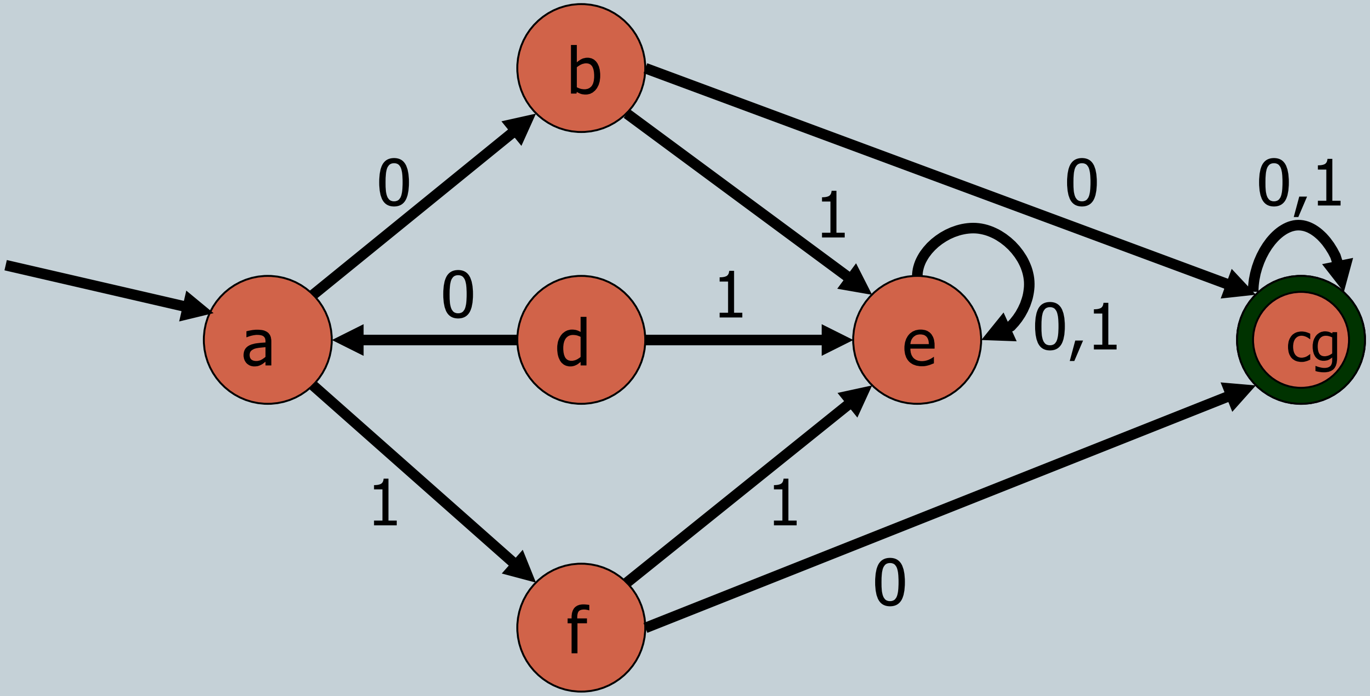
Unify Equivalent States.

A: No, they can be unified as illustrated below.



Unify Equivalent States.

Can any other states be unified because any subsequent string suffixes produce identical results?

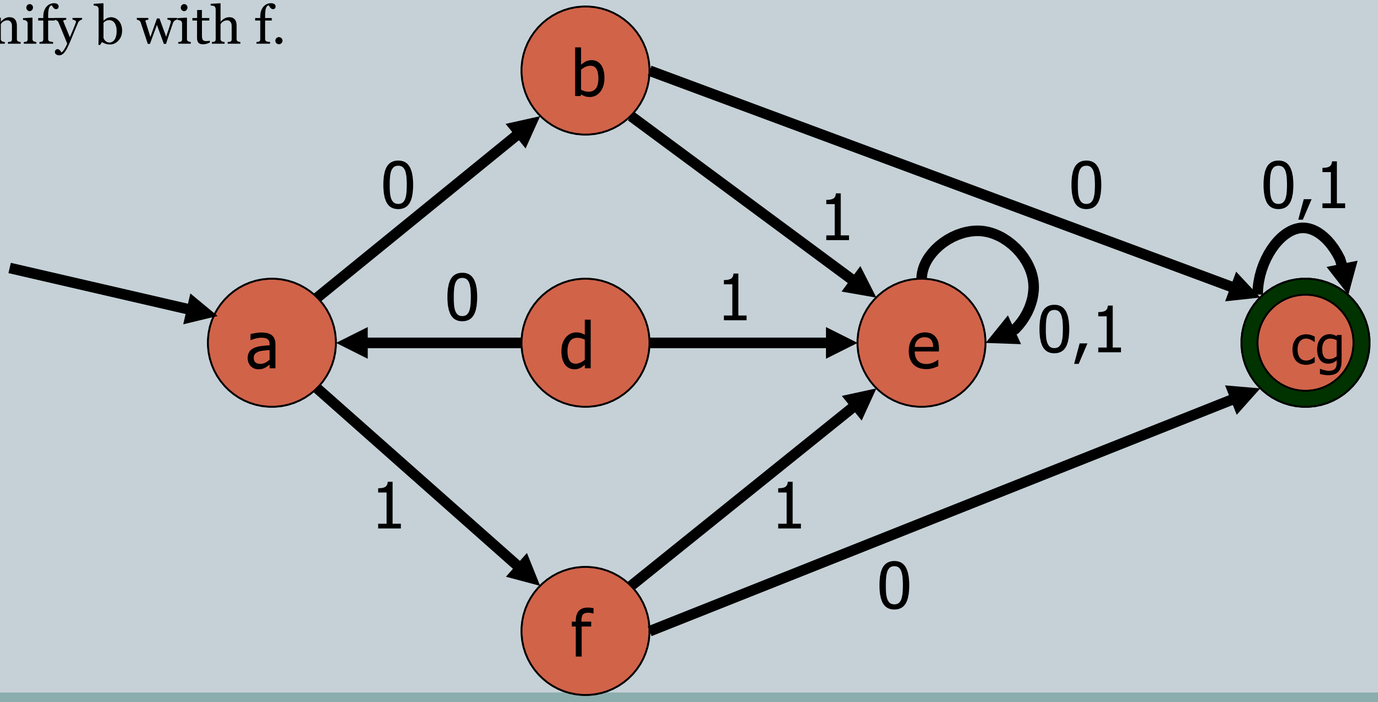


Unify Equivalent States.

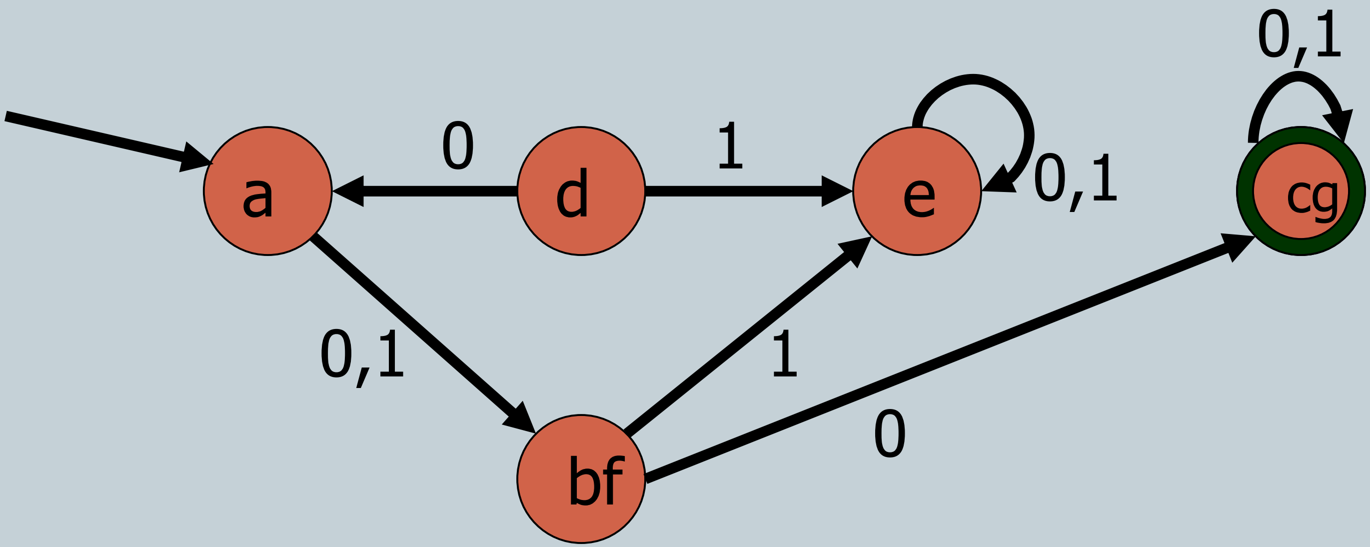
A: Yes, b and f. Notice that if you're in b or f then:

1. if string ends, reject in both cases
2. if next character is 0, forever accept in both cases
3. if next character is 1, forever reject in both cases

So unify b with f.



Unify Equivalent States.

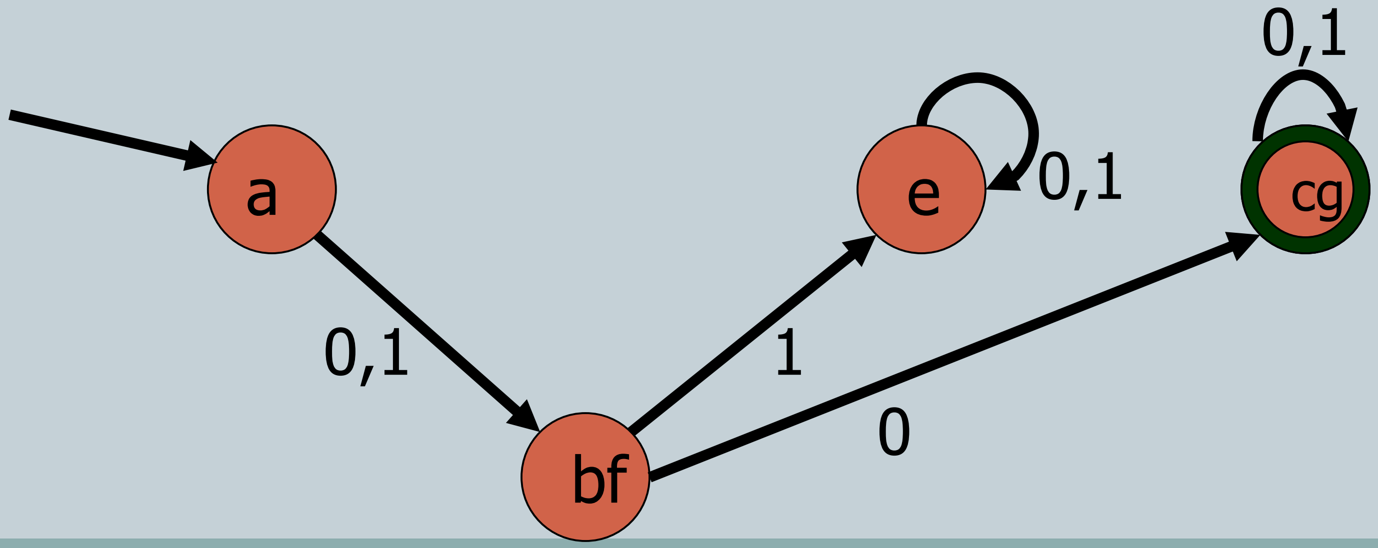


Any other ways to simplify the automaton?

Useless States

Yes, get rid of d.

Getting rid of unreachable ***useless states*** doesn't affect the accepted language.



DFA Minimization

DEF: An automaton is ***irreducible*** if

- it contains no useless states, and
- no two distinct states are equivalent.

The goal of minimization algorithm is to create irreducible automata from arbitrary ones.

Brzozowski's algorithm for Minimization

Idea: use the NFA->DFA subset construction algorithm twice

- For an automaton *N*
  - Let *reverse(N)* be the NFA constructed by making the initial state final, introducing an initial state which is the union of the original final states, and reversing all arcs.
  - Let *dfa(N)* be the DFA that results from applying the NFA to DFA conversion to *N*.
  - Let *reachable(N)* be an automaton for *N that is obtained* after removing all states that are not reachable from the initial state.

- Then,

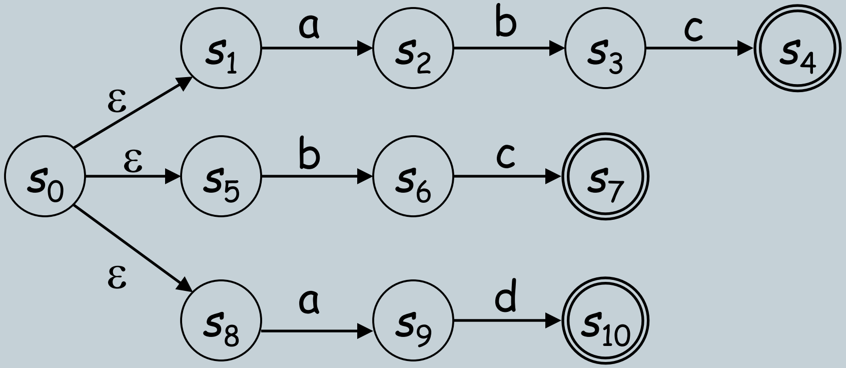
reachable(dfa(reverse[reachable(dfa(reverse(N))]))

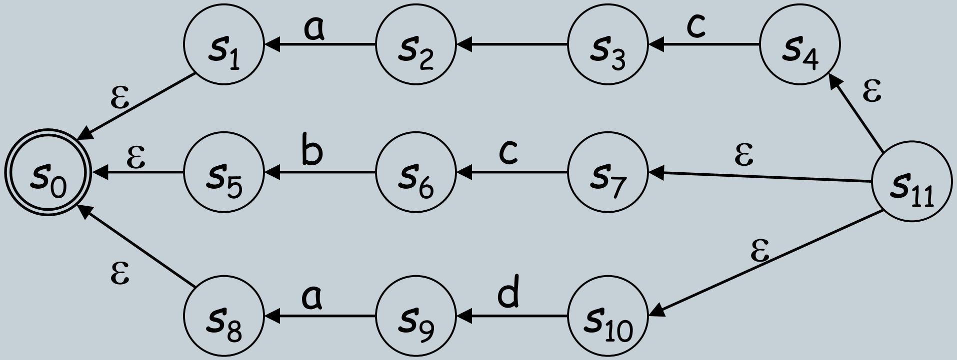
is the minimal DFA that implements *N* [Brzozowski, 1962]

This result is not intuitive, but it is proven in the above mentioned paper. We will not discuss the proof here.

Brzozowski's algorithm for Minimization

Step 1: construction of *reversed NFA*

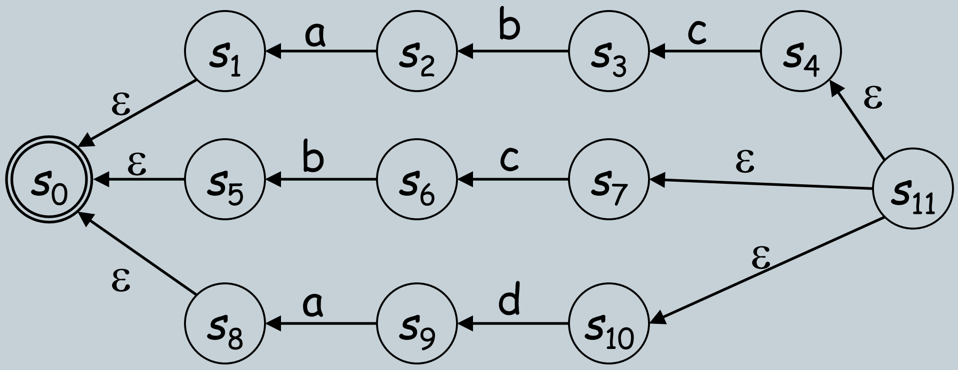




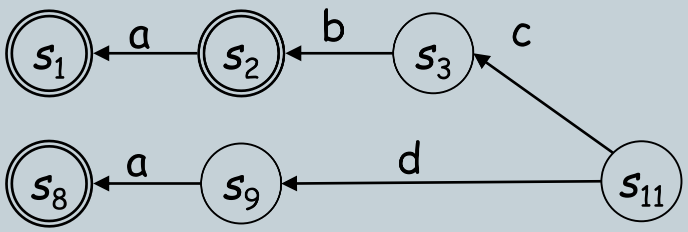
Reversed NFA

Brzozowski's algorithm for Minimization

Step 2: DFA subset construction algorithm on *reverse(NFA)*



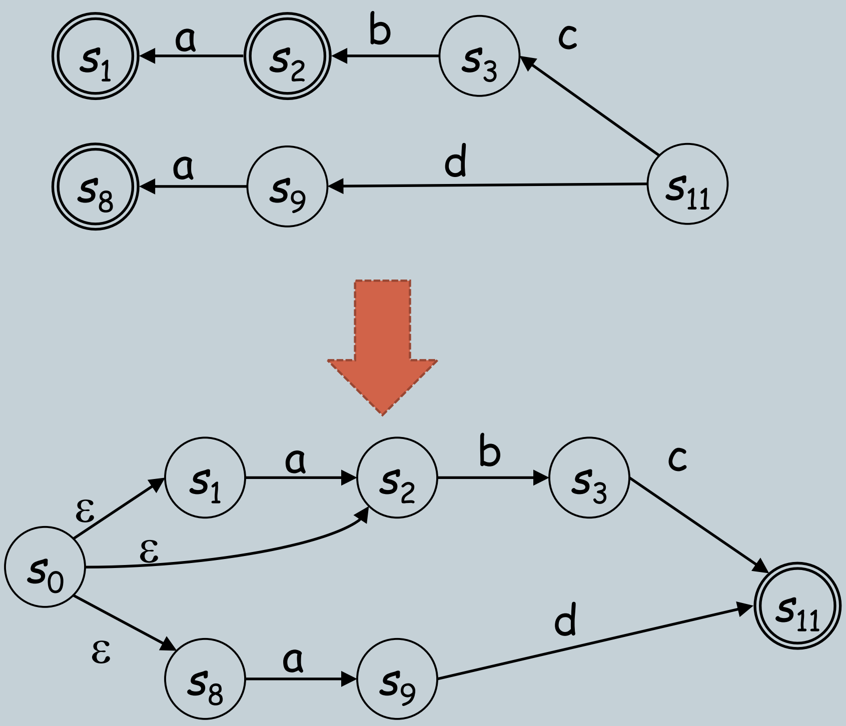
Reversed NFA



dfa(reverse(NFA))

Brzozowski's algorithm for Minimization

Step 3: reverse it again



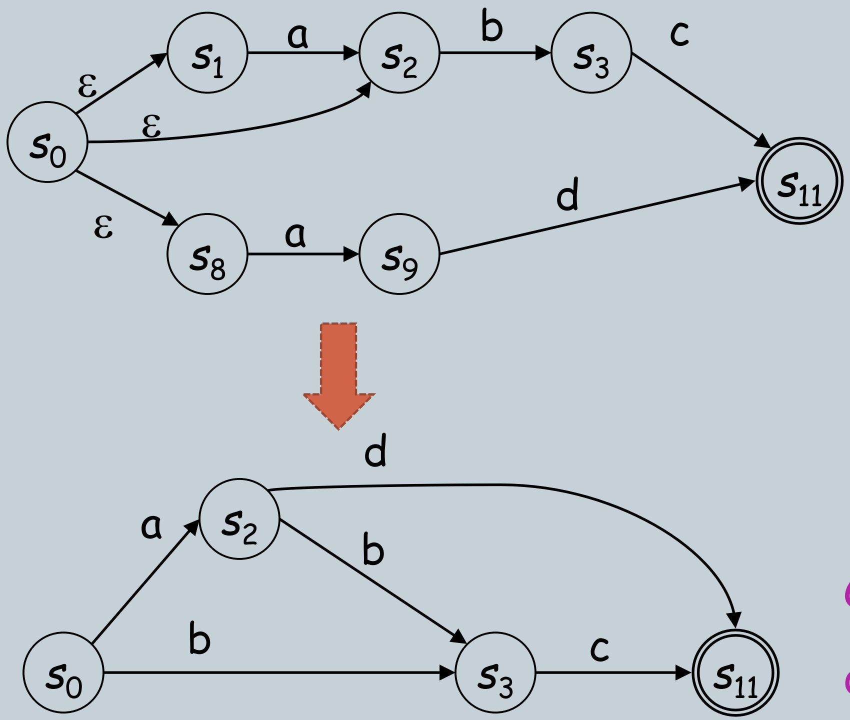
dfa(reverse(NFA))

Reverse it, again:

reverse(dfa(reverse(NFA)))

Brzozowski's algorithm for Minimization

Step 4: again DFA subset construction algorithm



Minimal DFA

Convert to DFA again:

dfa(reverse(dfa(reverse(NFA))))

And remove unreachable states (none in this example)

Summary

If we can

- Specify tokens with Regular Expressions

then, we (or better, Flex) can build a scanner by

- Creating an NFA for the recognition of each token
- Build a big NFA by union of the token NFAs
- Turn the big NFA into a DFA
- Minimize the DFA
- Scan with the obtained DFA

Next Topic

PARSING