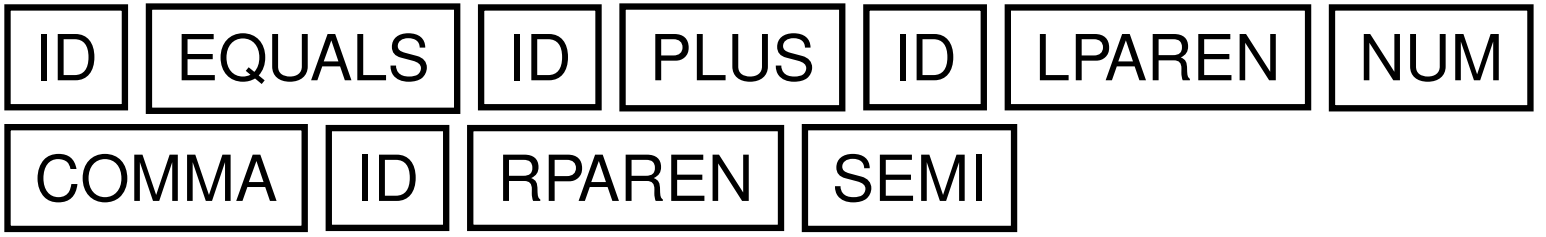
Lexical Analysis (Scanning)

Lexical Analysis (Scanning)

Translates a stream of characters to a stream of tokens

```
f o o ␣ = ␣ a + ␣ bar(2, ␣ q);
```



Token	Lexemes	Pattern
EQUALS	`=`	an equals sign
PLUS	`+`	a plus sign
ID	`a foo bar`	letter followed by letters or digits
NUM	`0 42`	one or more digits

Lexical Analysis

Goal: simplify the job of the parser.

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the the identifer is “supercalifragilisticexpialidocious.”

Parser rules are only concerned with tokens.

Describing Tokens

Alphabet: A finite set of symbols

Examples: { 0, 1 }, { A, B, C, . . . , Z }, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Stephen, $\alpha\beta\gamma$

Language: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), { 1, 11, 111, 1111 }, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let $L = \{ \epsilon$, wo $\}$, $M = \{$ man, men $\}$

Concatenation: Strings from one followed by the other

$LM = \{$ man, men, woman, women $\}$

Union: All strings from each language

$L \cup M = \{\epsilon$, wo, man, men $\}$

Kleene Closure: Zero or more concatenations

$M^* = \{\epsilon, M, MM, MMM, \ldots\} =$
$\{\epsilon$, man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, $\ldots\}$

Regular Expressions over an Alphabet Σ

A standard way to express languages for tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote languages $L(r)$ and $L(s)$,
   - $(r)|(s)$ denotes $L(r) \cup L(s)$
   - $(r)(s)$ denotes $\{tu : t \in L(r), u \in L(s)\}$
   - $(r)^*$ denotes $\cup_{i=0}^{\infty} L^i$ ($L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$)

Regular Expression Examples

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

RE	Language
$a\|b$	$\{a, b\}$
$(a\|b)(a\|b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
$(a\|b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$a\|a^*b$	$\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Specifying Tokens with REs

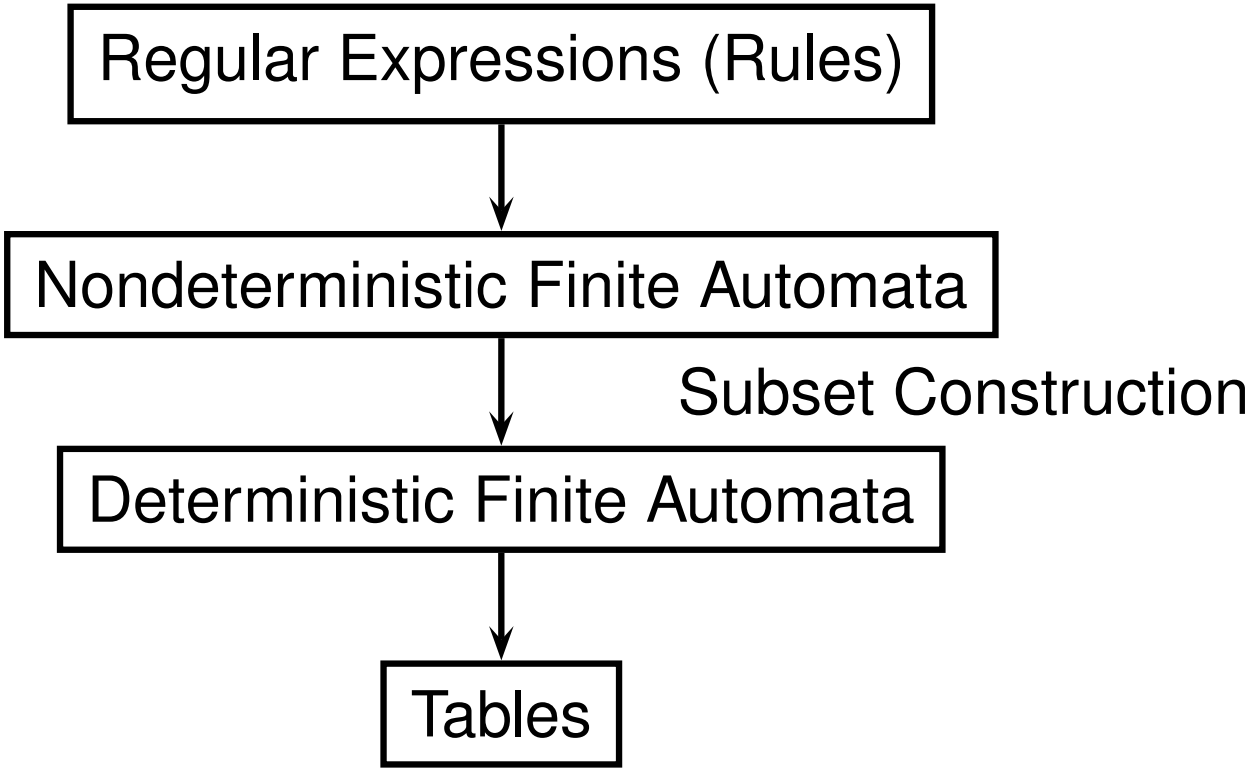
Typical choice: $\Sigma =$ ASCII characters, i.e.,
$\{\sqcup, !, \texttt{"}, \#, \$, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \sim\}$

letters: $A|B|\cdots|Z|a|\cdots|z$

digits: $0|1|\cdots|9$

identifier: **letter** $($ **letter** $|$ **digit** $)^*$

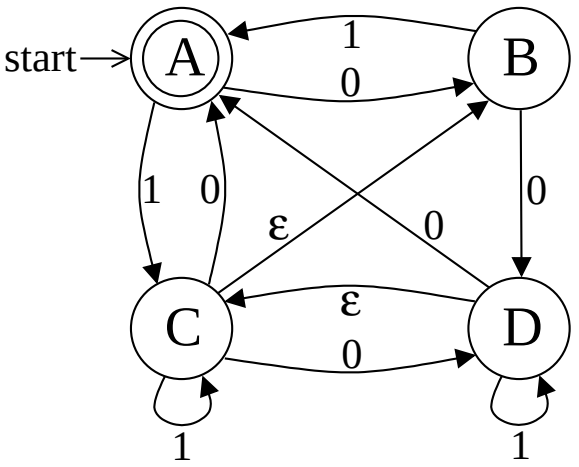
Implementing Scanners Automatically



Nondeterministic Finite Automata

1. Set of states S: $\{$ (A), (B), (C), (D) $\}$
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \rightarrow 2^S$

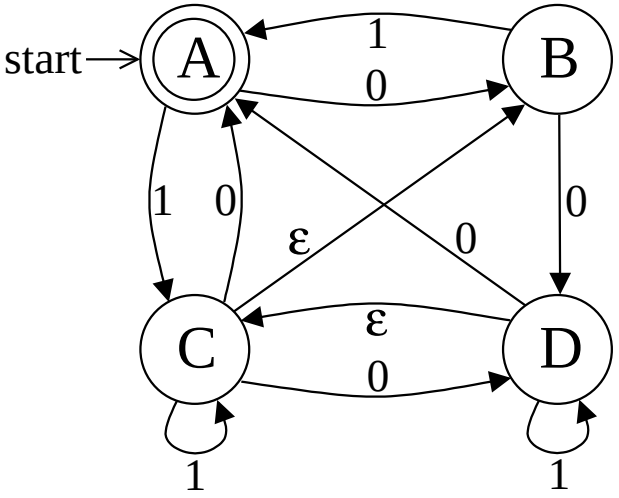
state	ϵ	0	1
A	–	{B}	{C}
B	–	{D}	{A}
C	{B}	{A,D}	{C}
D	{C}	{A}	{D}



4. Start state s_0 : (A)
5. Set of accepting states F: $\{$ (A) $\}$

The Language induced by an NFA

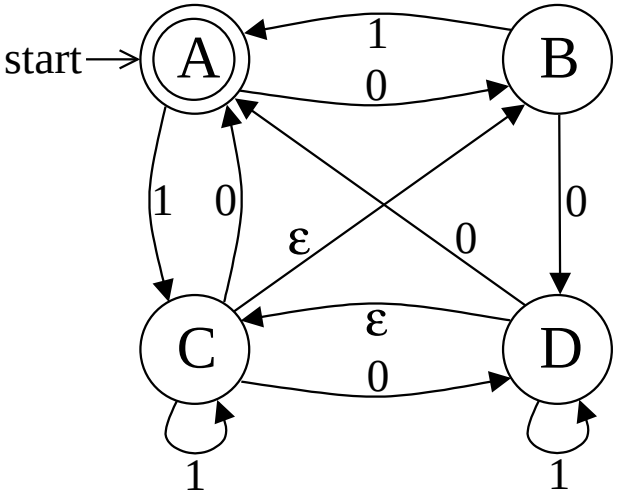
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



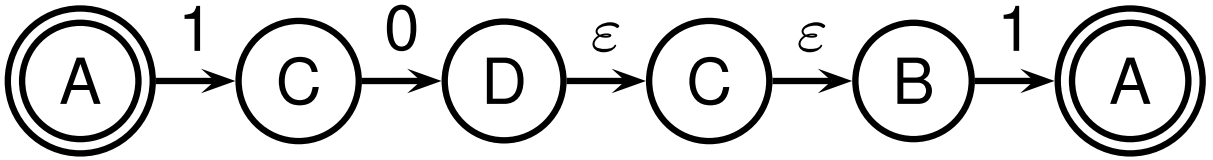
Is the string "101" accepted?

The Language induced by an NFA

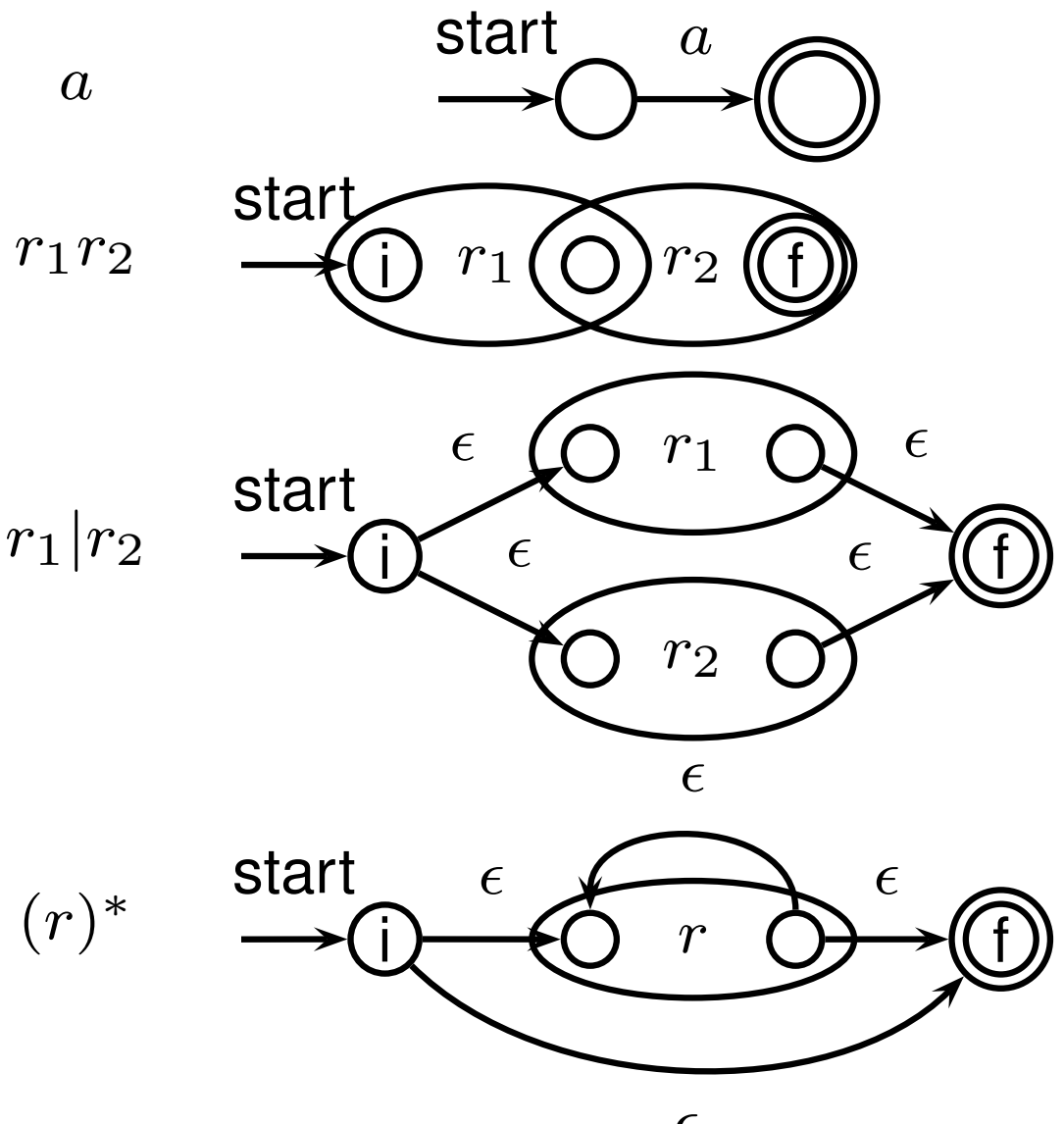
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



Is the string "101" accepted?

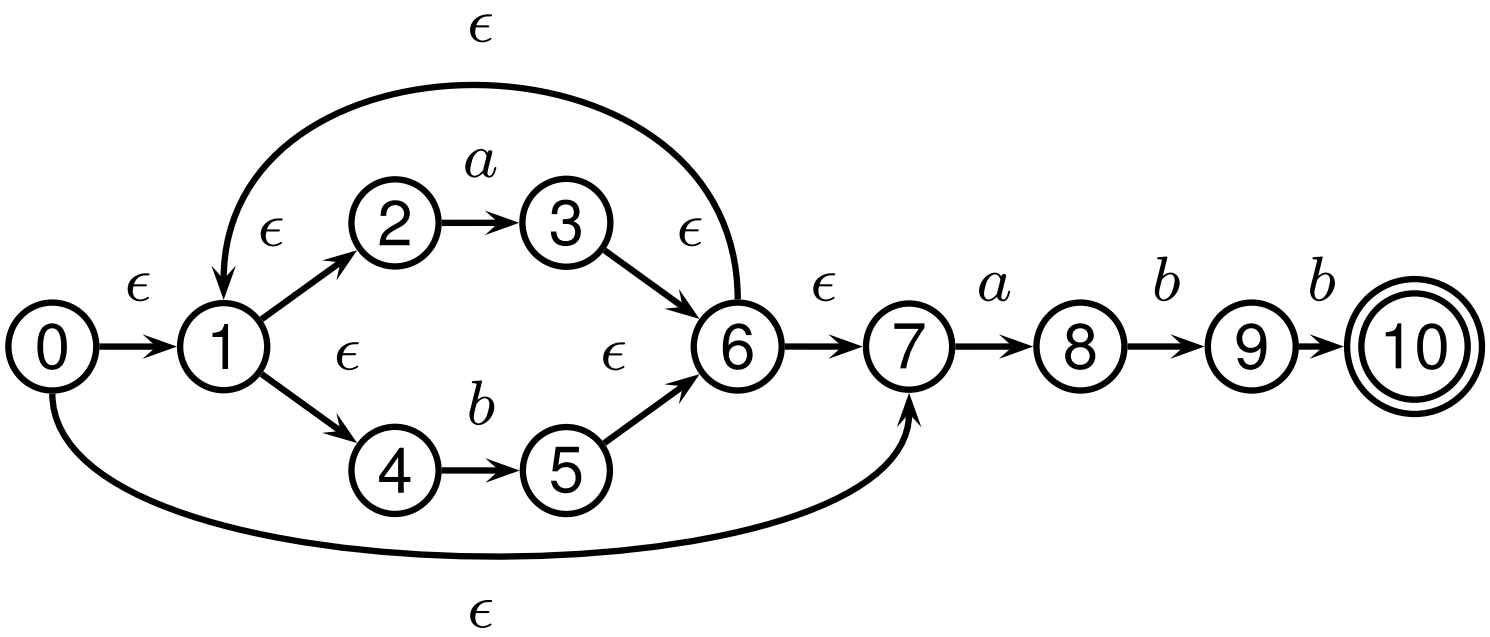


Translating REs into NFAs

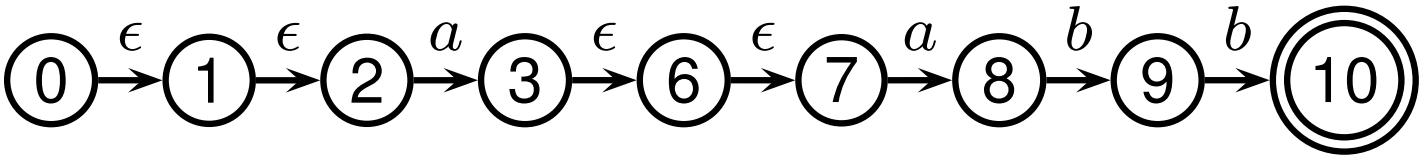


Translating REs into NFAs

Example: translate $(a|b)^*abb$ into an NFA



Show that the string "$aabb$" is accepted.



Simulating NFAs

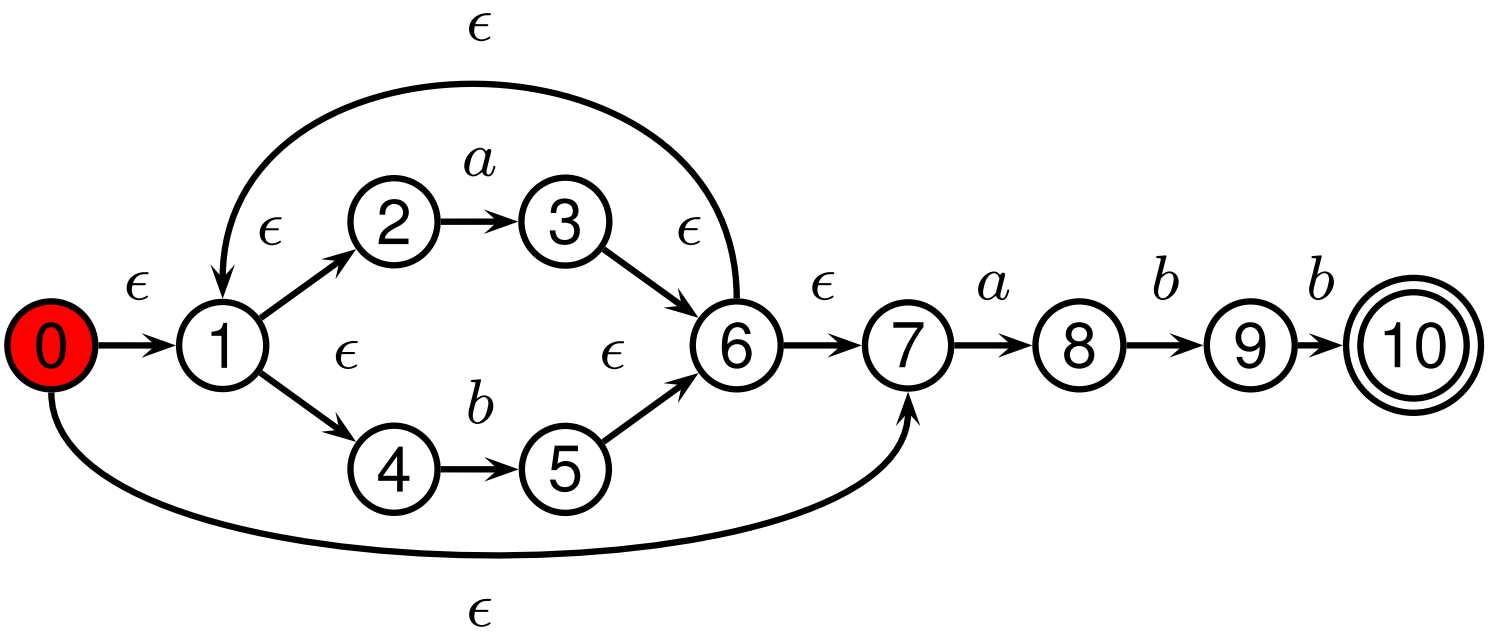
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

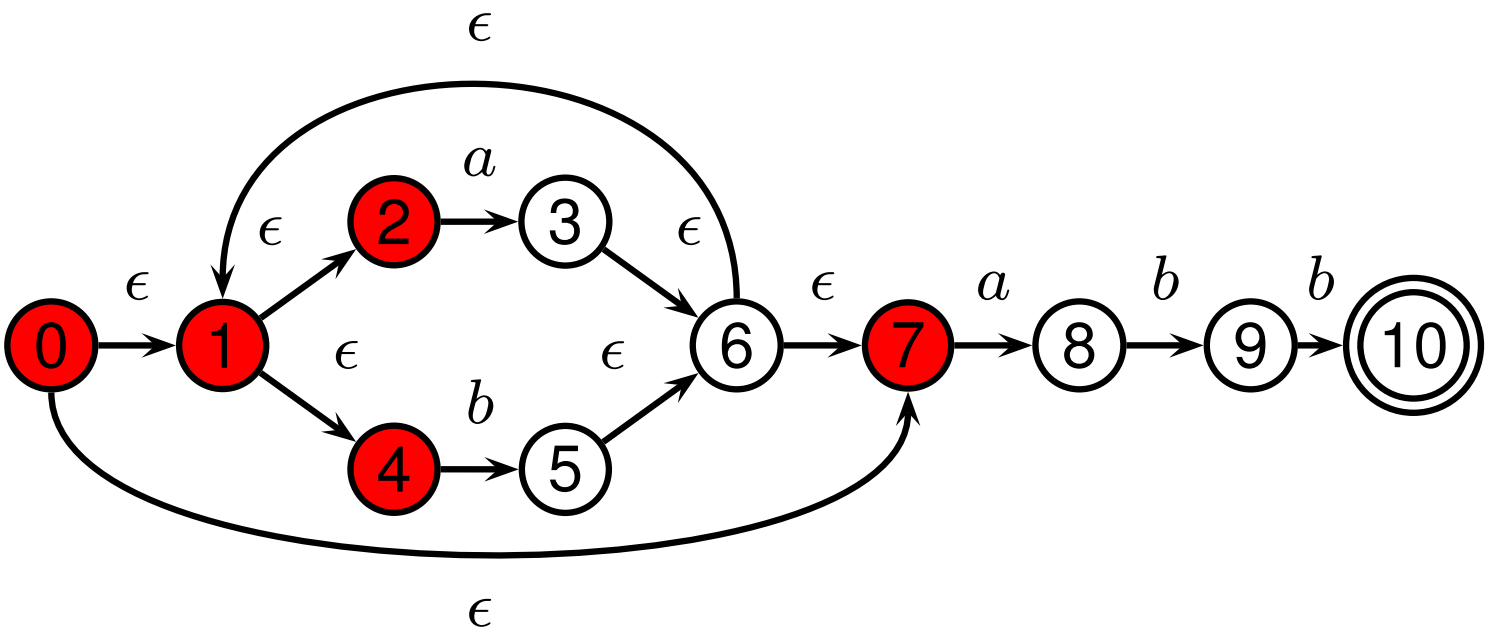
"Two-stack" NFA simulation algorithm:

1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

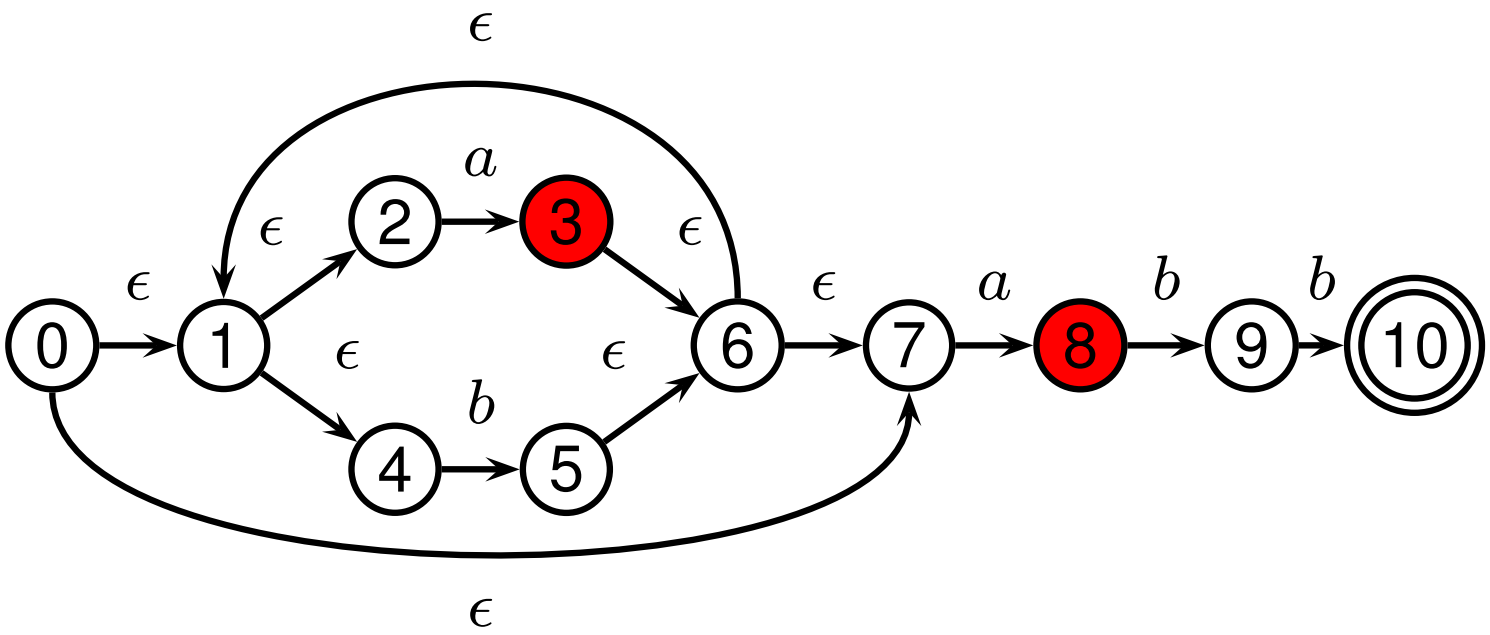
Simulating an NFA: $\cdot aabb$, Start



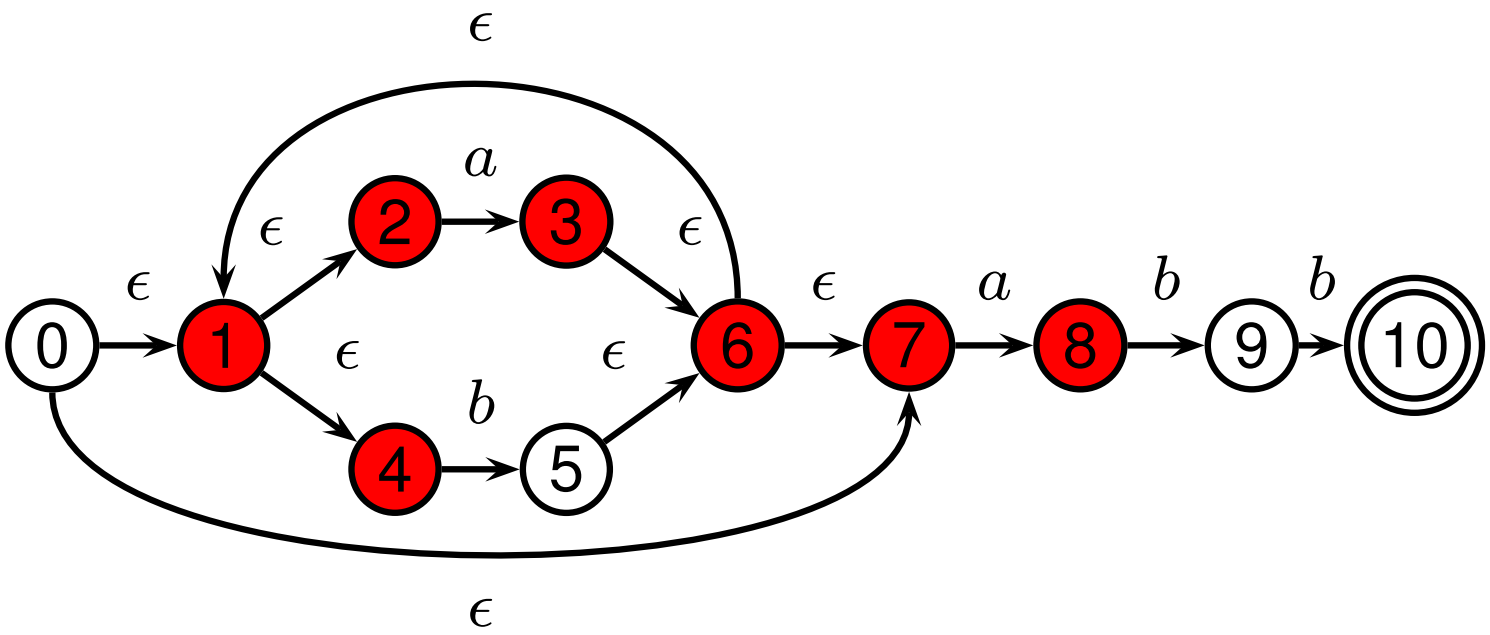
Simulating an NFA: $\cdot aabb$, ϵ-closure



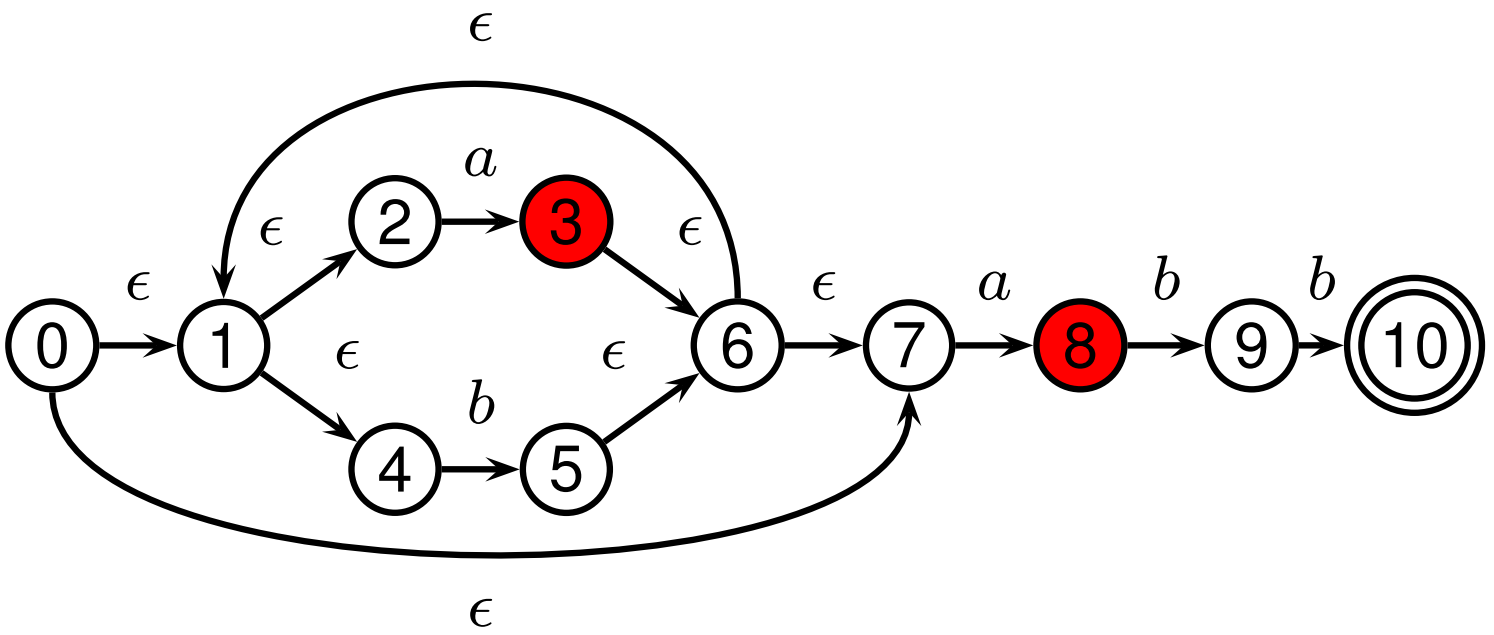
Simulating an NFA: $a{\cdot}abb$



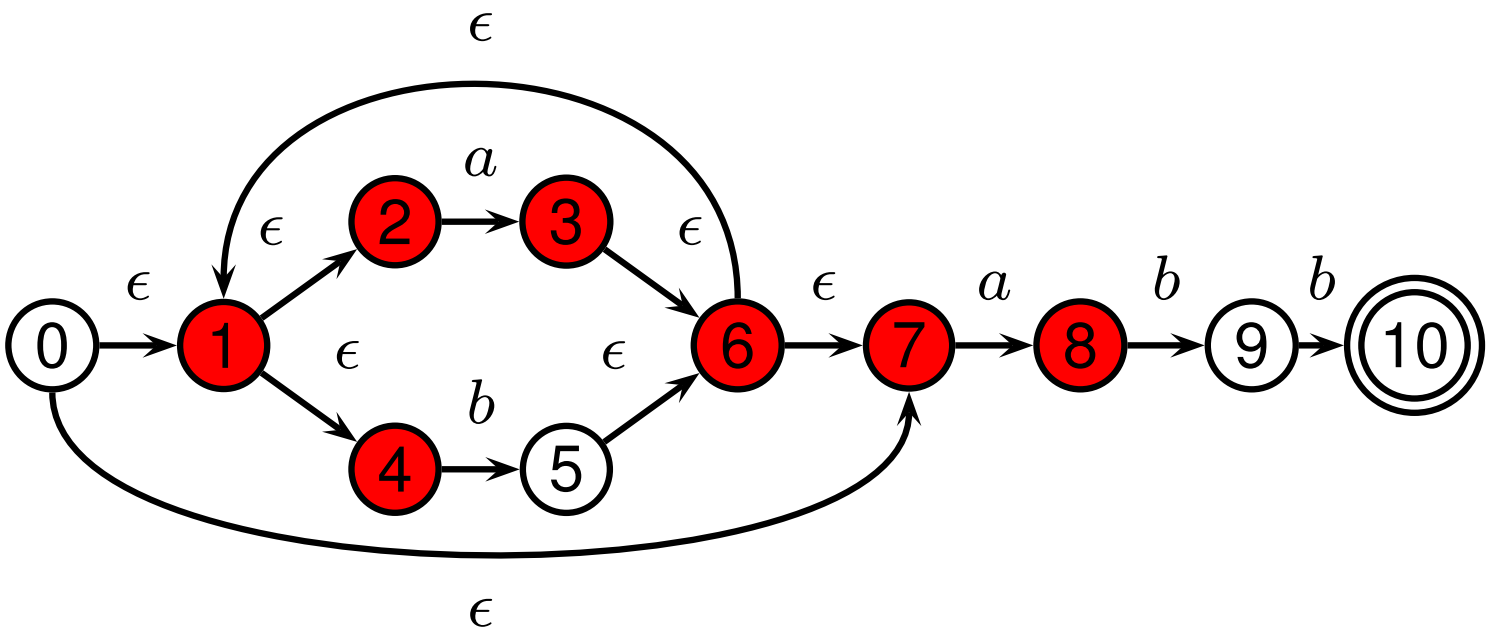
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



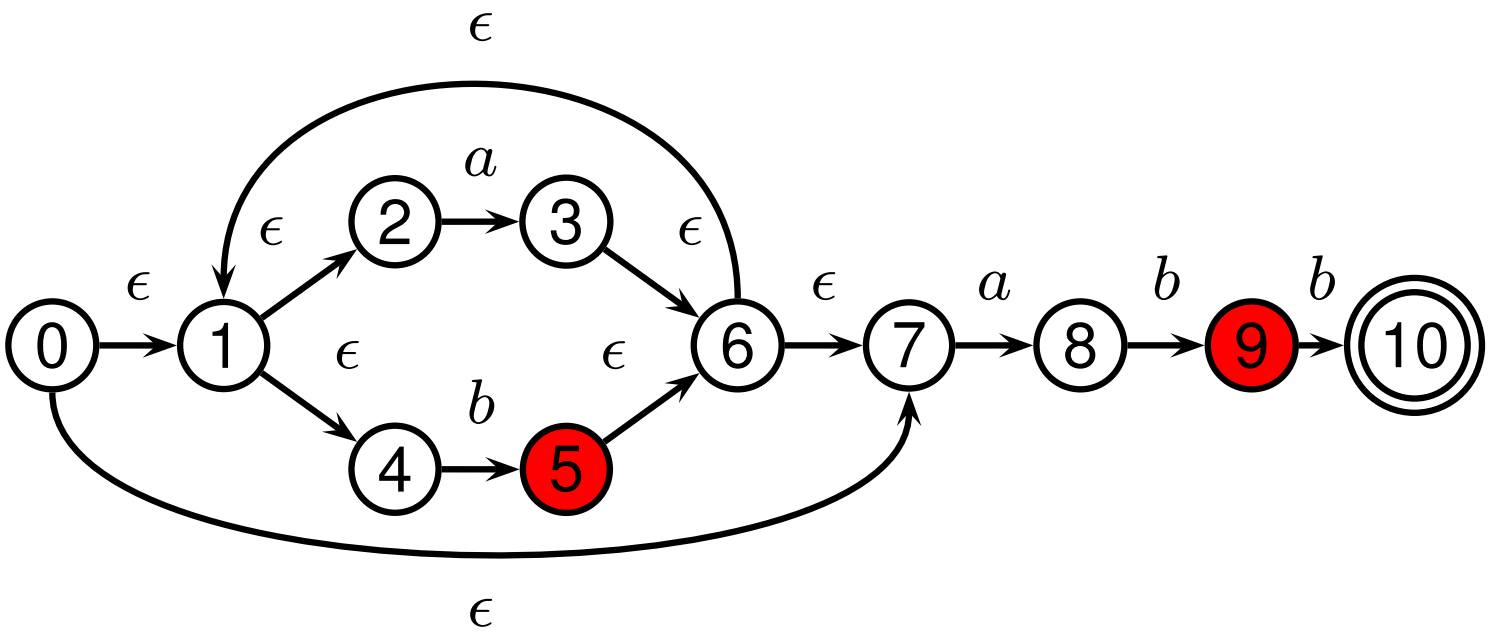
Simulating an NFA: $aa{\cdot}bb$



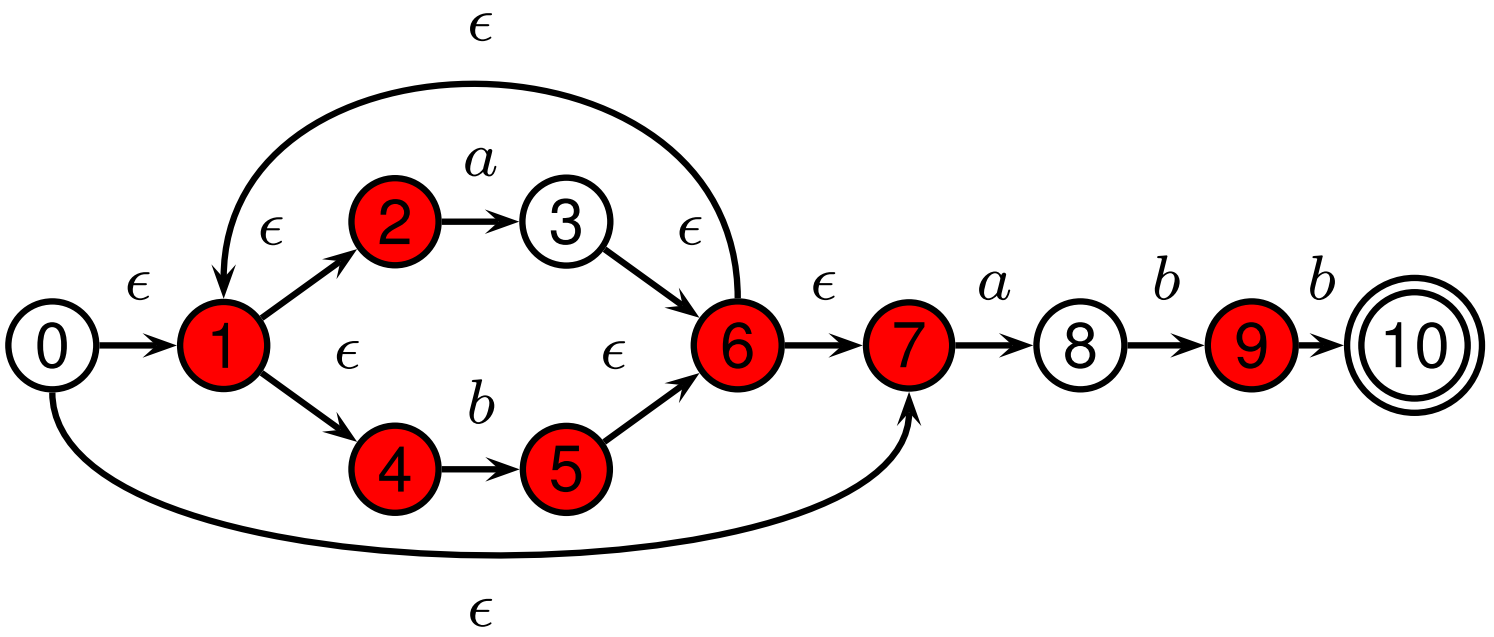
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



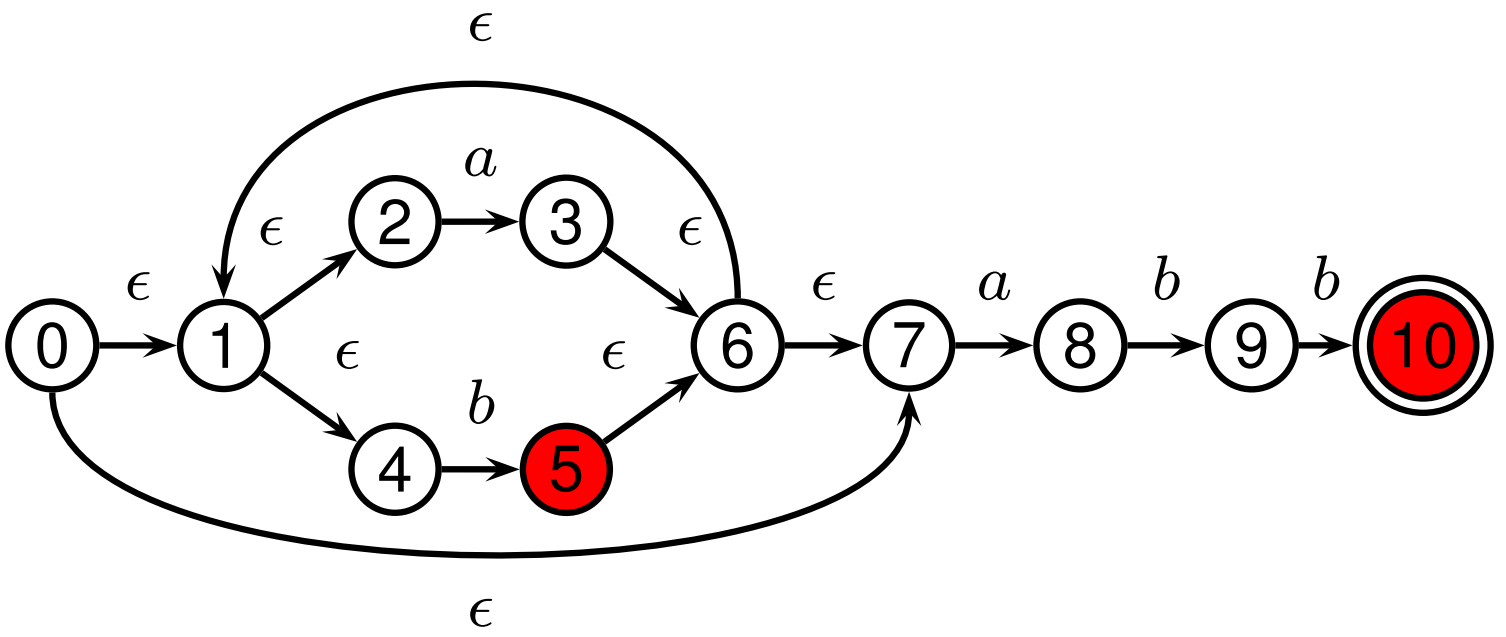
Simulating an NFA: $aab{\cdot}b$



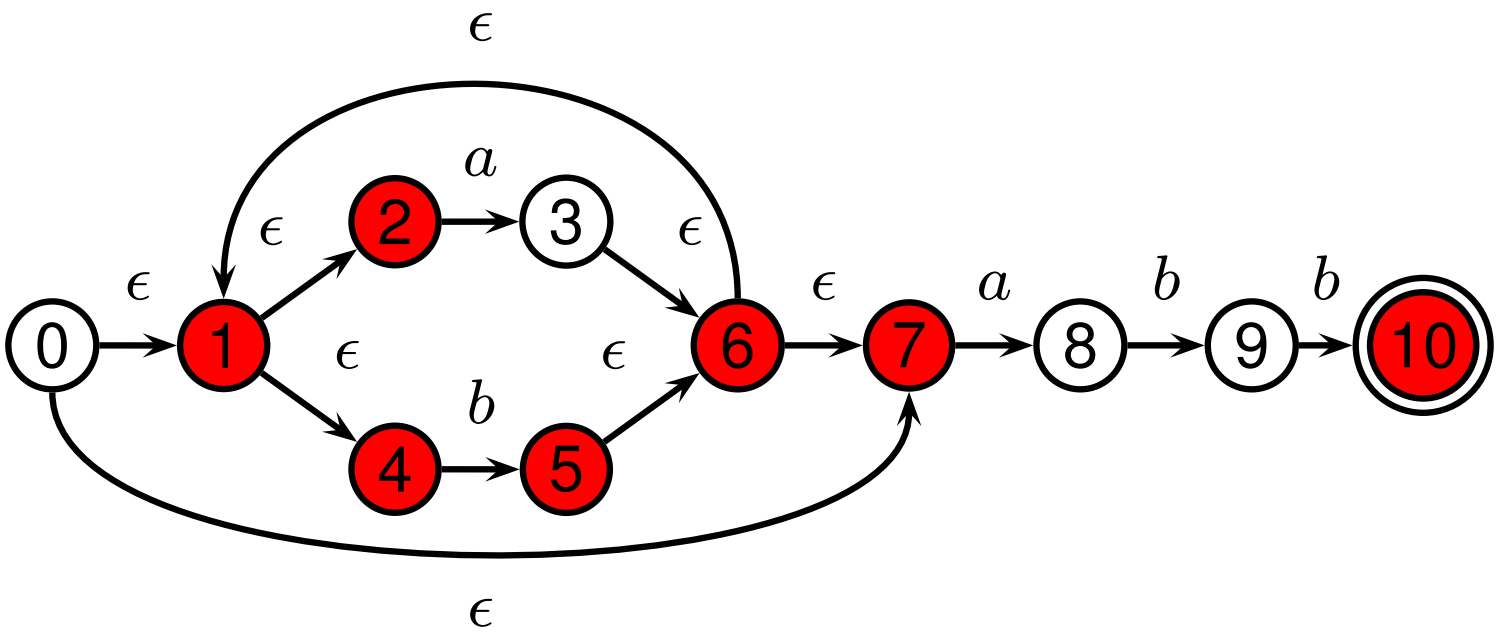
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

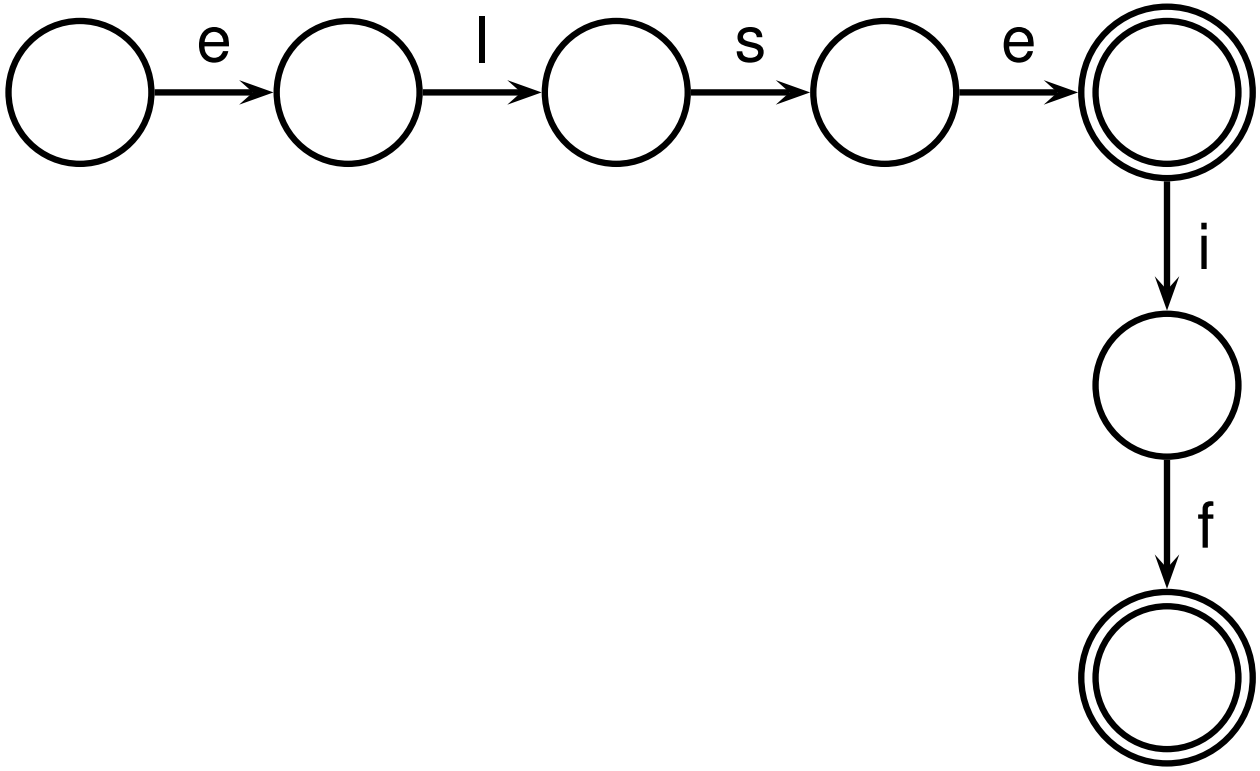
Restricted form of NFAs:

- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

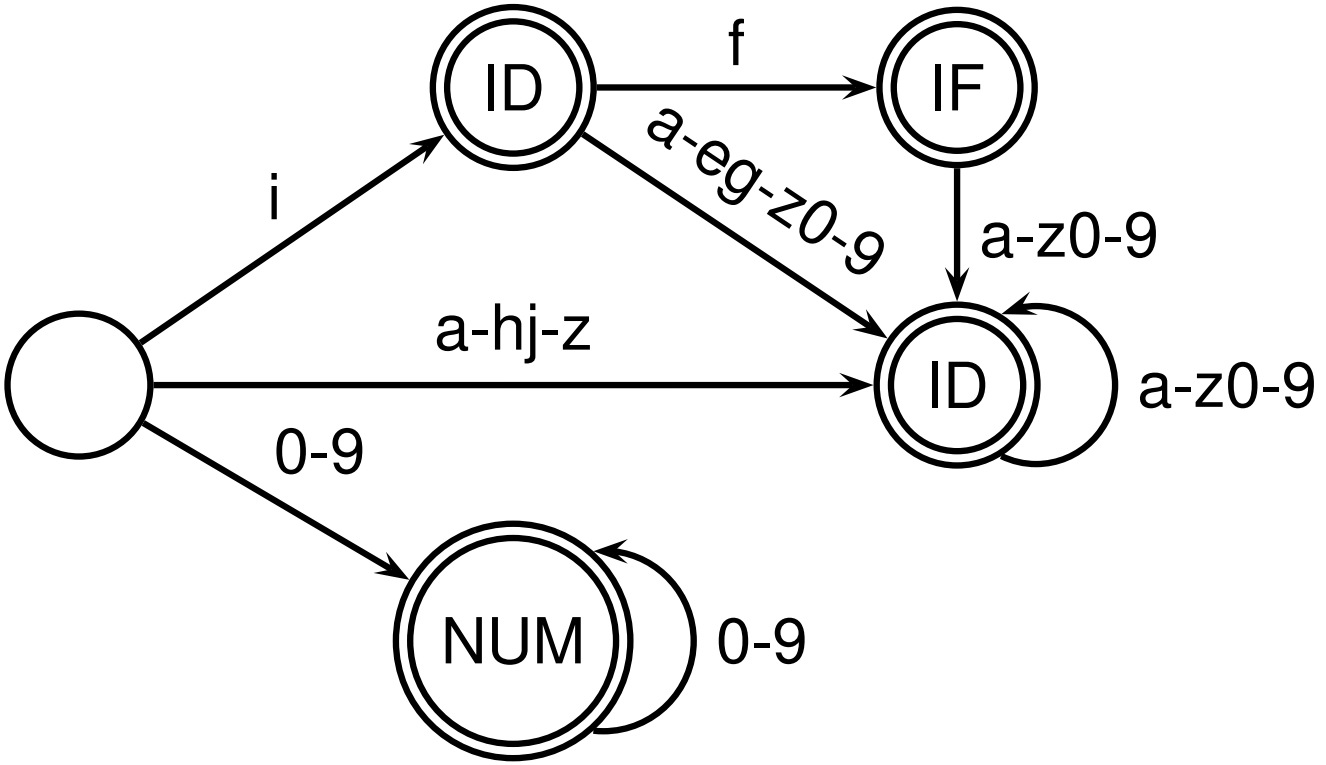
Deterministic Finite Automata

```
ELSE: "else" ;
ELSEIF: "elseif" ;
```



Deterministic Finite Automata

```
IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9')* ;
NUM: ('0'..'9')+ ;
```



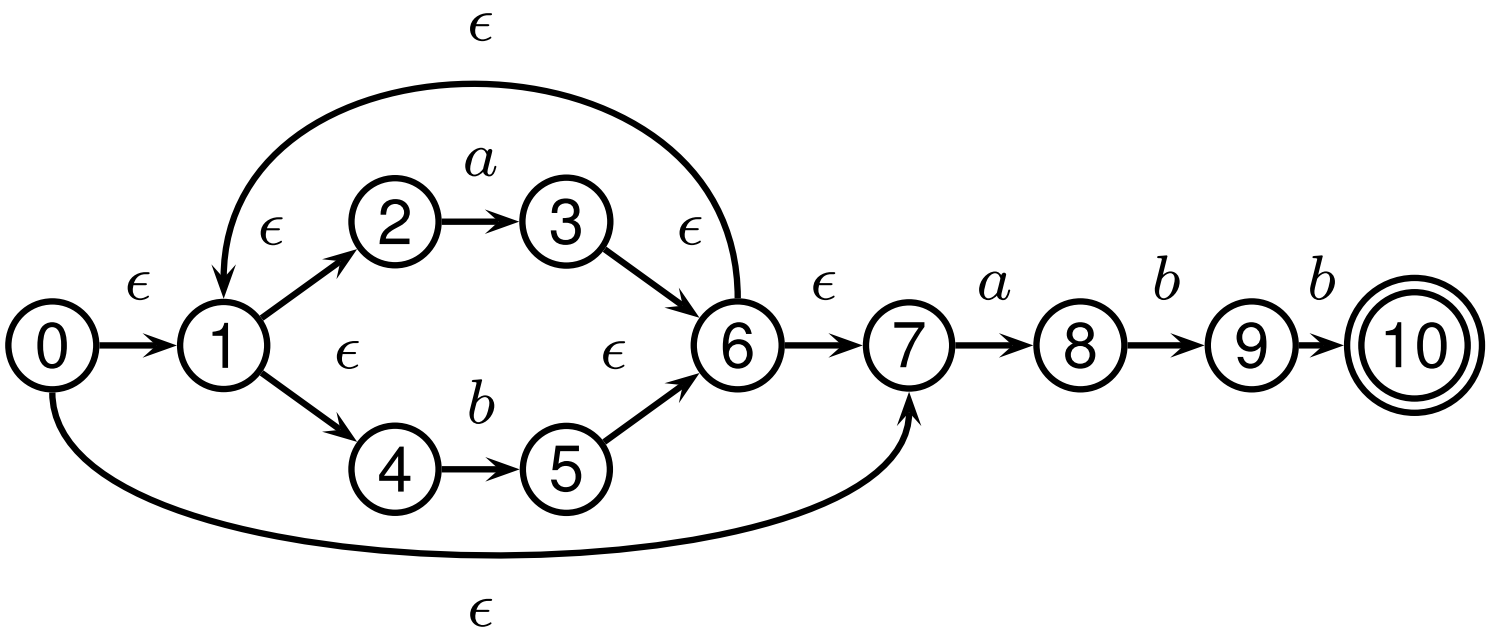
Building a DFA from an NFA

Subset construction algorithm

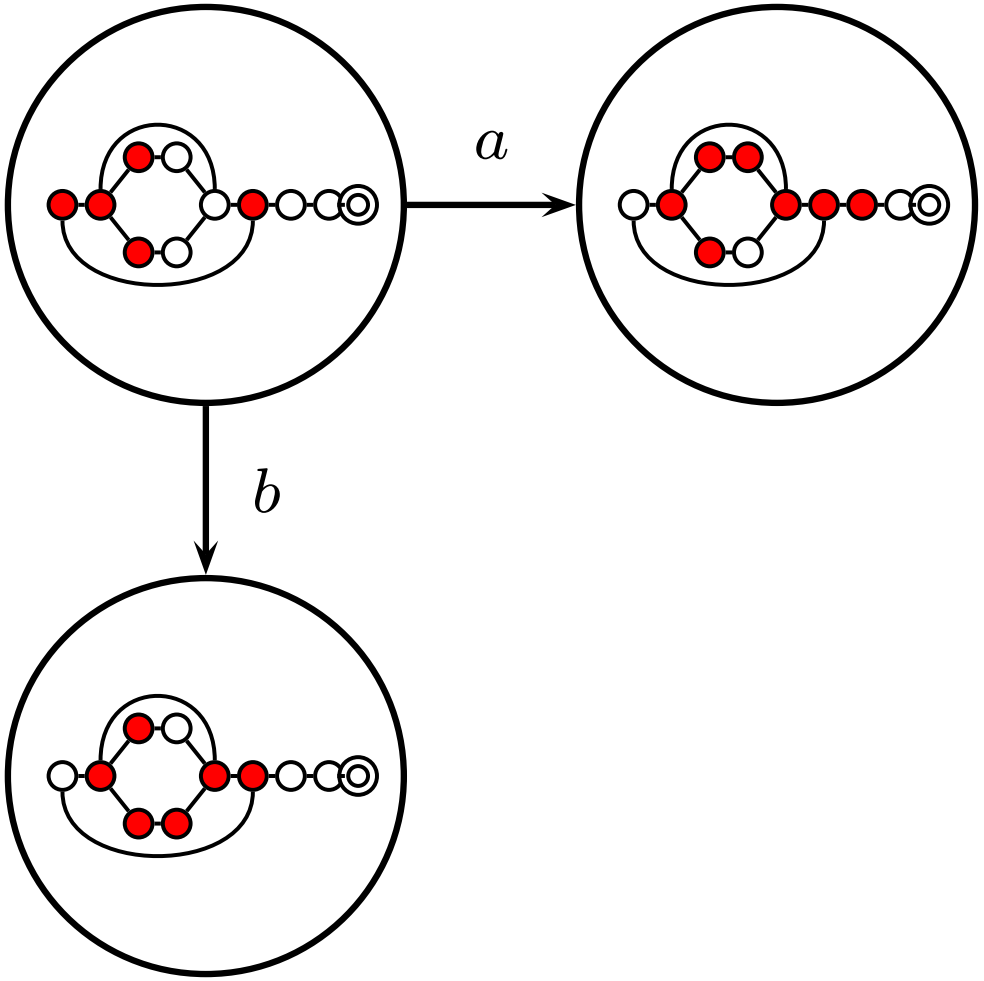
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

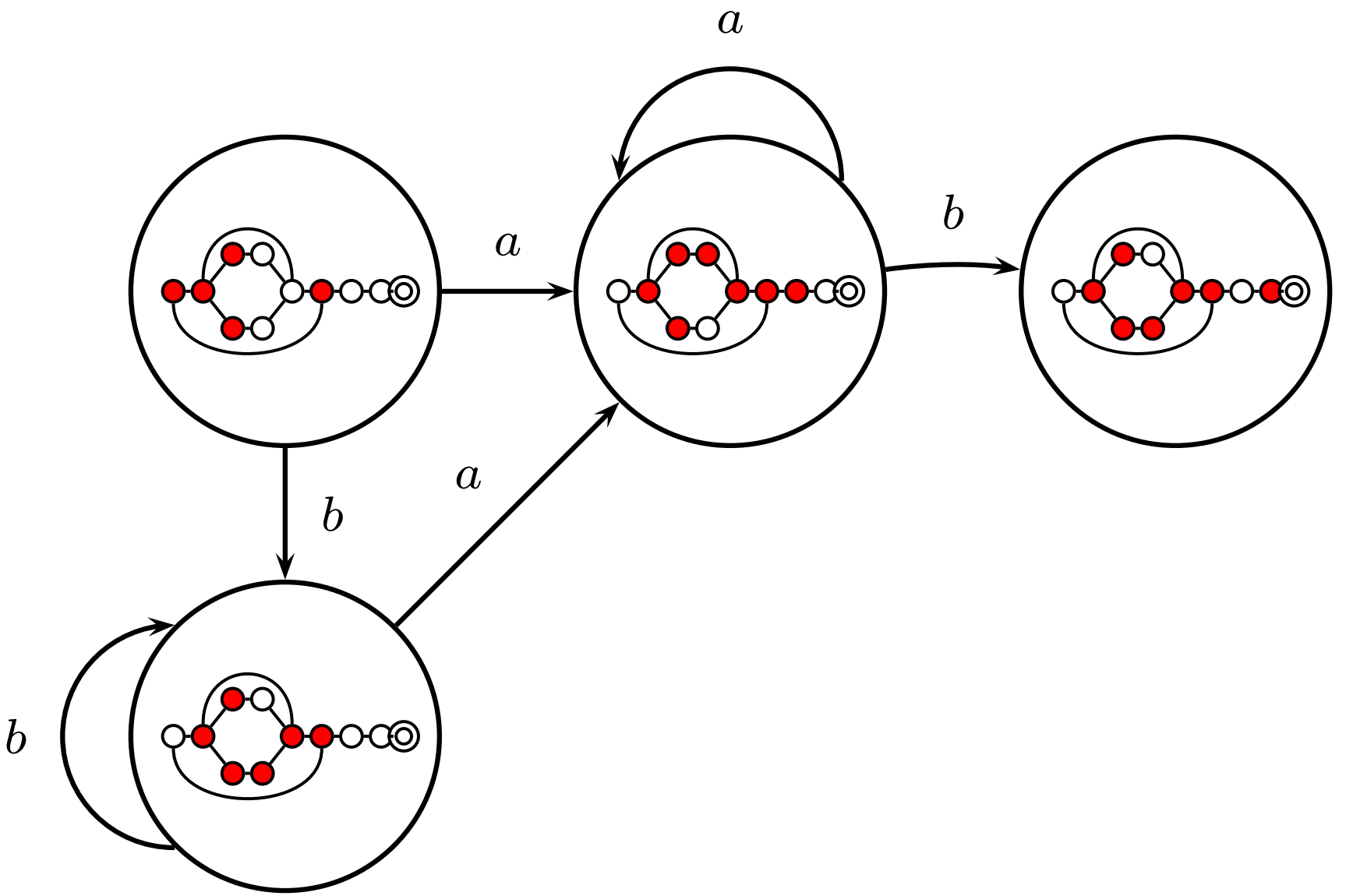
Subset construction for $(a|b)^*abb$



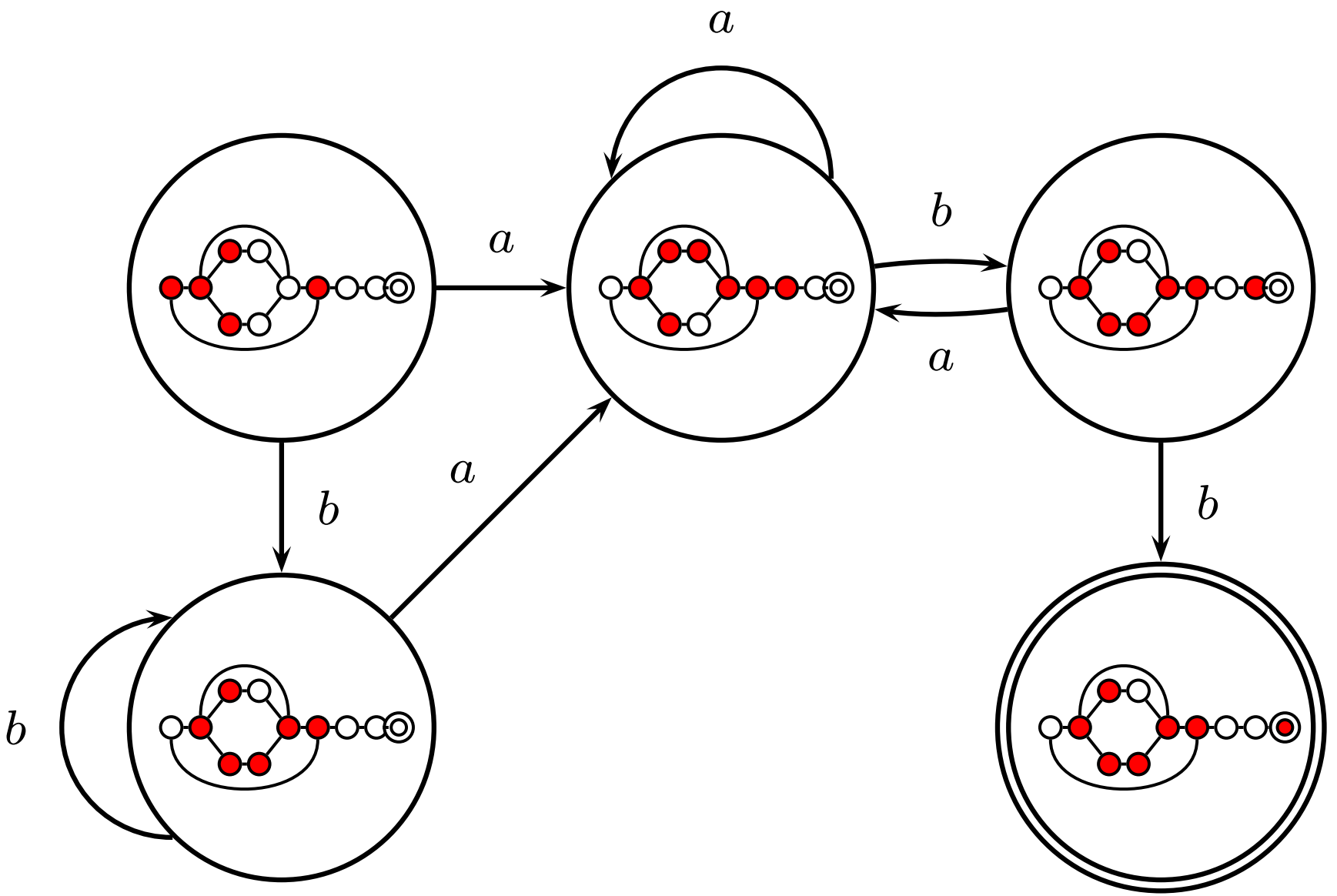
Subset construction for $(a|b)^*abb$ (1)



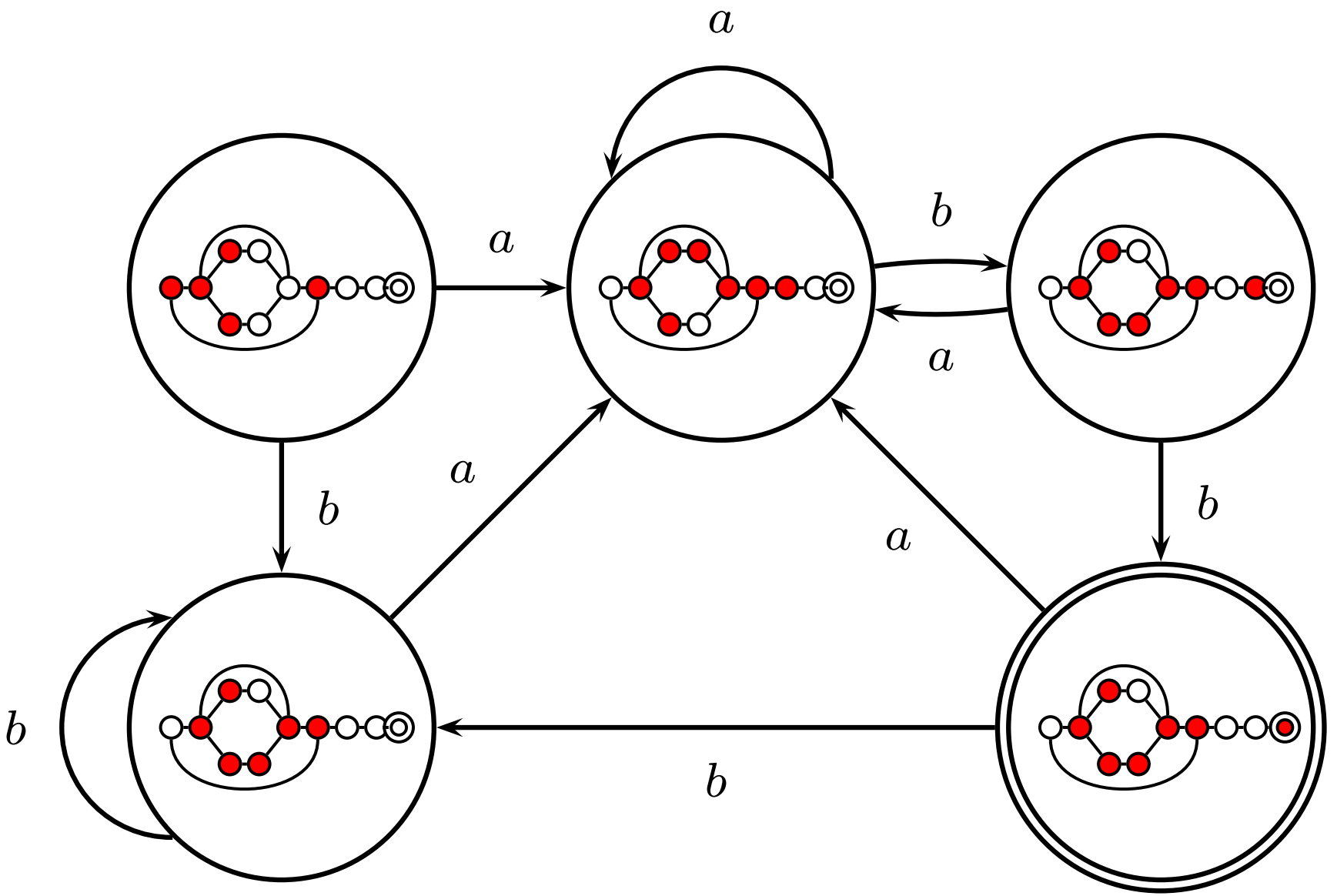
Subset construction for $(a|b)^*abb$ (2)



Subset construction for $(a|b)^*abb$ (3)



Subset construction for $(a|b)^*abb$ (4)



Subset Construction

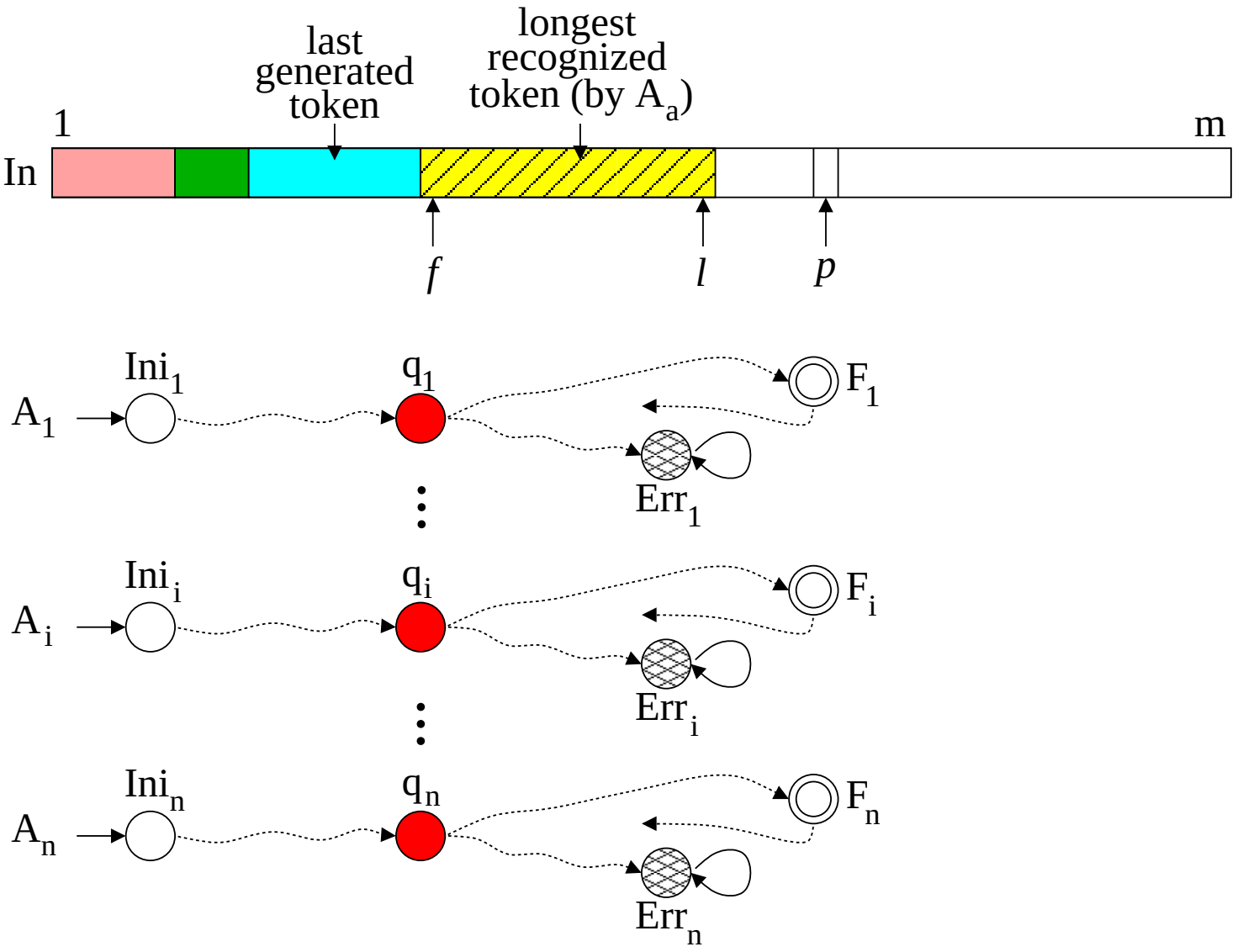
An DFA can be exponentially larger than the corresponding NFA.

n states versus 2^n

Tools often try to strike a balance between the two representations.

ANTLR uses a different technique.

An algorithm for scanning



$p = f = 1; l = 0; \forall i : q_i = \mathit{Ini}_i;$
while $p \leq m$ **do**
  $\forall i : q_i = \delta(q_i, \mathit{In}[p]); p = p + 1;$ **//State transition**
  if $\exists i : q_i \in F_i$ **then**
    $l = p - 1;\ a =$ *smallest* i *such that* $q_i \in F_i;$
  elseif $\forall i : q_i = \mathit{Err}_i$ **then**
    if $l \geq f$ **then**
      generate token of type a *with word* $\mathit{In}[f \ldots l]$;
      $p = f = l + 1; l = f - 1;$
      $\forall i : q_i = \mathit{Ini}_i;$
    else
      Lexical error
    endif
  endif
endwhile
if $l < f$ **then** *Lexical error* **endif**

The ANTLR Compiler Generator

Language and compiler for writing compilers

Running ANTLR on an ANTLR file produces Java source files that can be compiled and run.

ANTLR can generate

- Scanners (lexical analyzers)
- Parsers
- Tree walkers

An ANTLR File for a Simple Scanner

```
class  ExprLexer : public antlr4::Lexer;

LPAREN : '('  ;          // Rules for puctuation
RPAREN : ')'  ;
STAR : '*'  ;
PLUS : '+'  ;
SEMI : ';'  ;
fragment                 // Can only be used as a sub-rule
DIGIT : '0'..'9'  ;      // Any character between 0 and 9
INT : (DIGIT)+ ;         // One or more digits

WS : (' ' | '\t' | '\n'| '\r')     // Whitespace
     -> skip ;                     // Action: ignore
```

ANTLR Specifications for Scanners

Rules are names starting with a capital letter.

A character in single quotes matches that character.

```
LPAREN : '(' ;
```

A string in single quotes matches the string

```
IF : 'if' ;
```

A vertical bar indicates a choice:

```
OP : '+' | '-' | '*' | '/' ;
```

ANTLR Specifications

Question mark makes a clause optional.

```
PERSON : ('wo')? 'm' ('a'|'e') 'n' ;
```

(Matches man, men, woman, and women.)

Double dots indicate a range of characters:

```
DIGIT : '0'..'9';
```

Asterisk and plus match "zero or more," "one or more."

```
ID : LETTER (LETTER | DIGIT)* ;
NUMBER : (DIGIT)+ ;
```