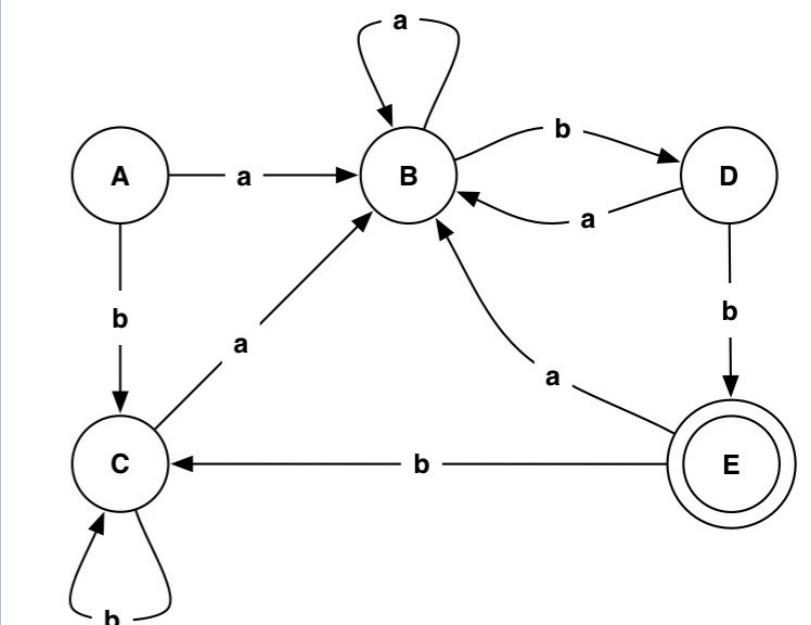


2. Lexical Analysis

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking
for their kind permission to reuse and adapt
the CS132 and CS502 lecture notes.
<http://www.cs.ucla.edu/~palsberg/>
<http://www.cs.purdue.edu/homes/hosking/>



Roadmap



- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
- > Limits of regular languages

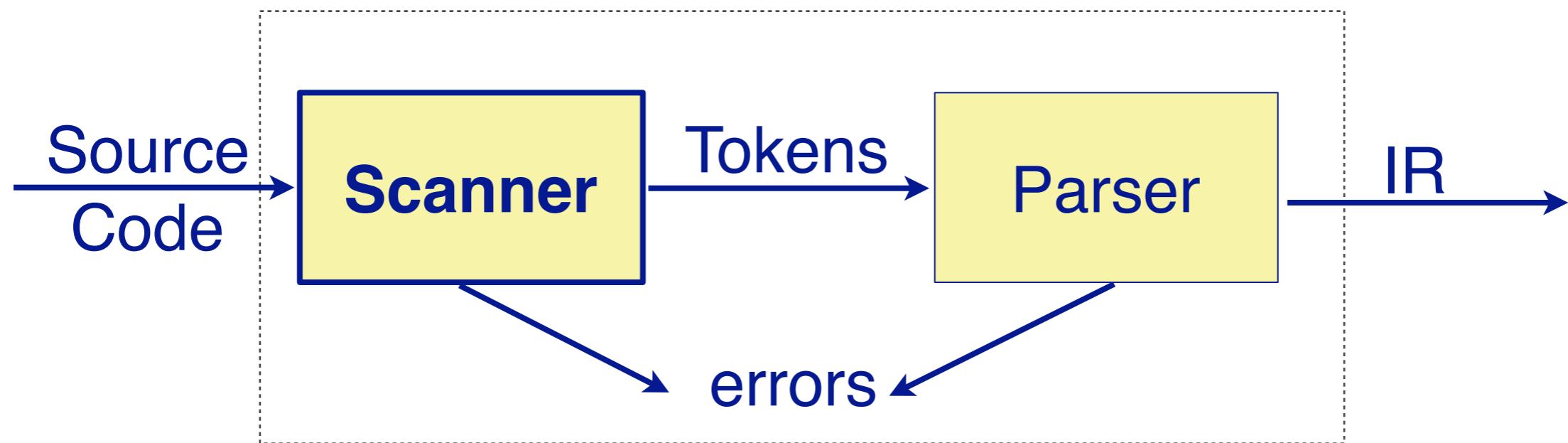
See, *Modern compiler implementation in Java* (Second edition), chapter 2.

Roadmap



- > **Introduction**
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
- > Limits of regular languages

Lexical Analysis



1. Maps sequences of characters to *tokens*
2. Eliminates white space (tabs, blanks, comments etc.)

$x = x + y$	\rightarrow	$<ID, x> <EQ> <ID, x> <PLUS> <ID, y>$
-------------	---------------	---------------------------------------

The string value of a token is a *lexeme*.

How to specify rules for token classification?

A scanner must recognize various parts of the language's syntax

Some parts are easy:

White space

```
<WS> ::= <WS> ''
          | <ws> '\t'
          |
          |   ''
          |
          |   '\t'
```

Keywords and operators

specified as literal patterns: do, end

Comments

opening and closing delimiters: /* ... */

Specifying patterns

Other parts are harder:

Identifiers

alphanumeric followed by k alphanumerics (`_`, `$`, `&`, ...))

Numbers

integers: 0 or digit from 1–9 followed by digits from 0–9

decimals: integer '.' digits from 0–9

reals: (integer or decimal) 'E' (+ or –) digits from 0–9

complex: '(' real ',' real ')')

We need an expressive notation to specify these patterns!

A key issue is ...



Roadmap



- > Introduction
- > **Regular languages**
- > Finite automata recognizers
- > From RE to DFAs and back again
- > Limits of regular languages

Languages and Operations

A *language* is a set of strings

<i>Operation</i>	<i>Definition</i>
Union	$L \cup M = \{ s \mid s \in L \text{ or } s \in M \}$
Concatenation	$LM = \{ st \mid s \in L \text{ and } t \in M \}$
Kleene closure	$L^* = \cup_{i=0,\infty} L^i$
Positive closure	$L^+ = \cup_{i=1,\infty} L^i$

Formally, a language is a set of strings (or “sentences”). We can perform various operations over languages, such as union, concatenation etc.

In the slide, L and M are languages, while s and t are strings. Operations over languages produce new languages by iterating over strings they contain.

The Kleene closure produces all possible concatenations of strings in a language L.

Examples:

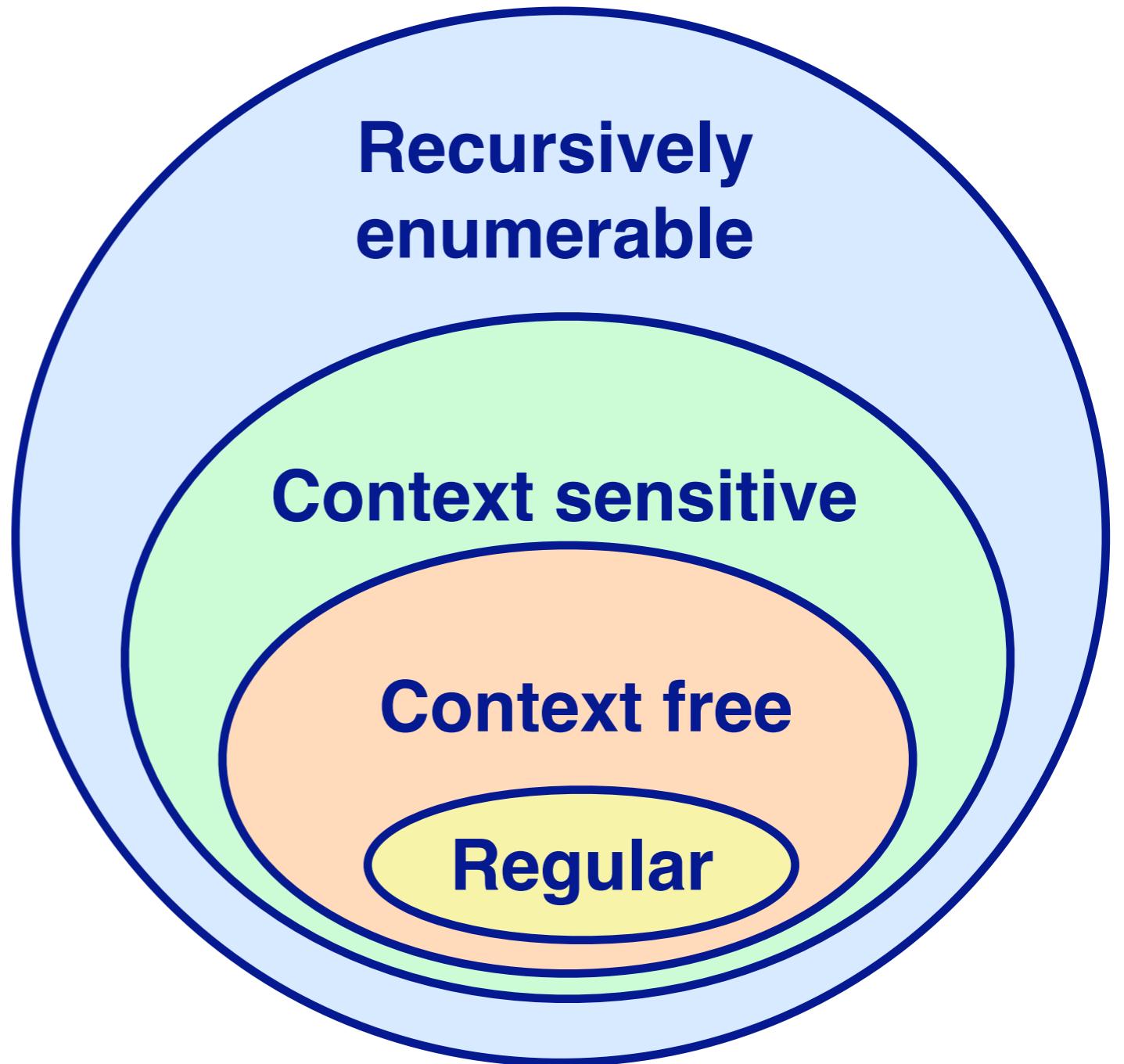
$$L = \{ a, b \}, M = \{ c, d \}$$

$$LM = \{ ac, ad, bc, bd \}$$

$$L^* = \{ ^\wedge, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots \}$$

Production Grammars

- > Powerful formalism for language description
 - Start symbol (S_0)
 - Production rules ($A \rightarrow abA$)
 - Non-terminals (A, B)
 - Terminals (a,b)
- > Rewriting



A common way to specify languages is with the help of *production grammars*. These consist of a set of *rewrite rules* that allow you *generate* all possible strings in a language.

A grammar starts with a *start symbol* S_0 , and consists of a number of rules of the form

$$A \rightarrow abA$$

consisting of *non-terminals*, like S_0 and A , that can be *expanded* using further production rules, and *terminals*, like a and b , that cannot.

By repeated expending terminals using different rules, one can generate all possible strings in the language specified by the grammar.

Detail: The Chomsky Hierarchy

- > **Type 0: $a \rightarrow \beta$**
 - Unrestricted grammars generate *recursively enumerable languages*. Minimal requirement for recognizer: Turing machine.
- > **Type 1: $aA\beta \rightarrow a\gamma\beta$**
 - Context-sensitive grammars generate *context-sensitive languages*, recognizable by linear bounded automata
- > **Type 2: $A \rightarrow \gamma$**
 - Context-free grammars generate *context-free languages*, recognizable by non-deterministic push-down automata
- > **Type 3: $A \rightarrow a$ and $A \rightarrow aB$**
 - Regular grammars generate *regular languages*, recognizable by finite state automata

NB: A is a non-terminal; a, β , γ are strings of terminals and non-terminals

Since compilers need to recognize languages rather than generate them, we need a way to turn a grammar into a recogniser.

The Chomsky Hierarchy (named after Noam Chomsky) formalizes how different constraints over the production rules produce very different classes of languages. Unrestricted grammars (i.e., where the left and right-hand sides of the rules may contain a mix of terminals and non-terminals) are the hardest to parse, and require a Turing machine to recognize them.

Programming languages are mostly context-free (only non-terminals on the left-hand side), with occasionally some context-sensitive features. Typically the tokens of a programming language (i.e., identifiers, strings, comments etc.) are Type 3 and can be recognized by a FSA.

https://en.wikipedia.org/wiki/Chomsky_hierarchy

Grammars for regular languages

Regular grammars generate regular languages

Definition:

In a regular grammar, all productions have one of two forms:

1. $A \rightarrow aA$
2. $A \rightarrow a$

where A is any non-terminal and a is any terminal symbol

These are also called type 3 grammars (Chomsky)

Regular languages can be described by *Regular Expressions*

Regular expressions (RE) over an alphabet Σ :

1. ϵ is a RE denoting the set $\{\epsilon\}$
2. If $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. If r and s are REs denoting $L(r)$ and $L(s)$, then:
 - > $(r) | (s)$ is a RE denoting $L(r) \cup L(s)$
 - > $(r)(s)$ is a RE denoting $L(r)L(s)$
 - > $(r)^*$ is a RE denoting $L(r)^*$

We adopt a *precedence* for operators: *Kleene closure*, then *concatenation*, then *alternation* as the order of precedence.

For any RE r , there exists a grammar g such that $L(r) = L(g)$

Epsilon is the set with the “empty” string. As you can see, we don’t define a^+ (1 or more copies of a) or $[a]$ (optional a) as they can be derived.

Patterns are often specified as regular languages.

Notations used to describe a regular language (or a regular set) include both regular expressions and regular grammars

Examples

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

- > $a \mid b$ denotes $\{a,b\}$
- > $(a \mid b) (a \mid b)$ denotes $\{aa,ab,ba,bb\}$
- > a^* denotes $\{\varepsilon, a, aa, aaa, \dots\}$
- > $(a \mid b)^*$ denotes the set of all strings of a's and b's (including ε)
- > Universit($\ddot{a} \mid ae$)t Bern($e \mid$) ...

Algebraic properties of REs

$r s = s r$	is commutative
$r (s t) = (r s) t$	is associative
$r(st) = (rs)t$	concatenation is associative
$r(s t) = rs rt$ $(s t)r = sr tr$	concatenation distributes over
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \epsilon)^*$	ϵ is contained in *
$r^{**} = r^*$	* is idempotent

Examples of using REs to specify lexical patterns

identifiers

letter $\rightarrow (a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z)$

digit $\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

id $\rightarrow \text{letter} (\text{ letter} \mid \text{ digit})^*$

numbers

integer $\rightarrow (+ \mid - \mid \varepsilon) (0 \mid (1 \mid 2 \mid 3 \mid \dots \mid 9) \text{ digit}^*)$

decimal $\rightarrow \text{integer} . (\text{ digit})^*$

real $\rightarrow (\text{ integer} \mid \text{ decimal}) \text{ E } (+ \mid -) \text{ digit}^*$

complex $\rightarrow '(\text{ real }, \text{ real })'$

Roadmap

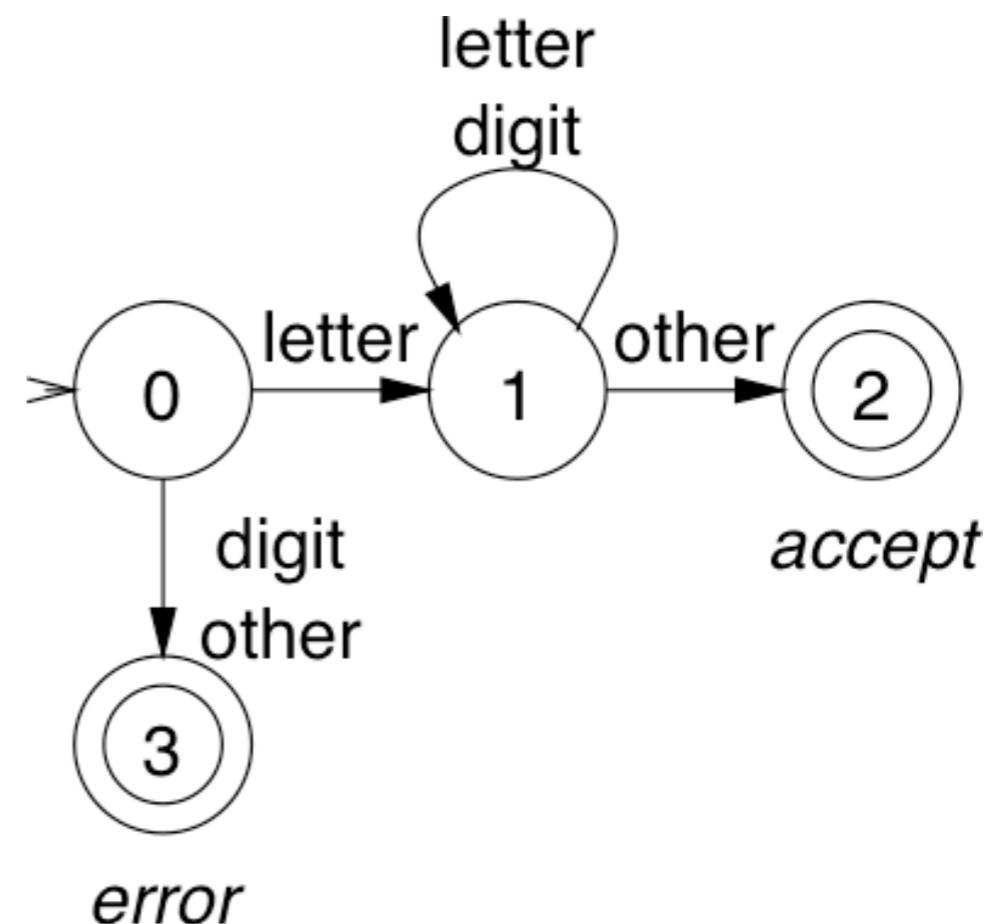


- > Introduction
- > Regular languages
- > **Finite automata recognizers**
- > From RE to DFAs and back again
- > Limits of regular languages

Recognizers

```
letter → (a | b | c | ... | z | A | B | C | ... | Z)  
digit → (0|1|2|3|4|5|6|7|8|9)  
id → letter ( letter | digit )*
```

From a regular expression we can construct a deterministic finite automaton (DFA)



Any regular language can be recognized by a deterministic finite state automaton (DFA).

A finite state automaton (FSA) has a finite number of states, a start state, a number of final states, and labelled transitions between them. An FSA is *deterministic* if, given a state and a label, there is always a *unique transition* to take. In contrast, a non-deterministic finite automation (NFA) may offer a (non-deterministic) choice of transitions.

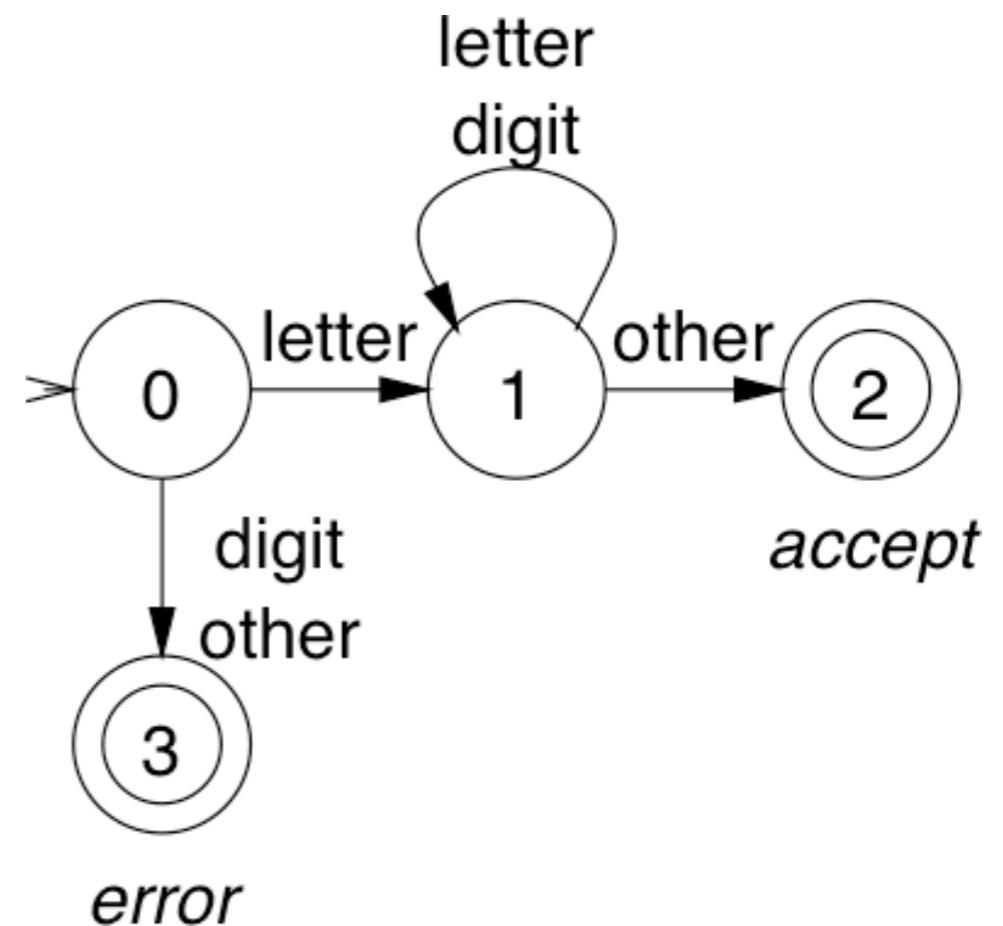
In the example, the start state is 0, the final states are 2 and 3 (leading respectively to acceptance or rejection of the input), and the transitions are all deterministic.

On any given input of a letter, a digit or another character, there is always a unique transition available.

The obvious question now is, given a regular expression, *how can we generate the corresponding DFA?*

Code for the recognizer

```
char ← next_char();
state ← 0;           /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case 1:    /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            break;
        case 2:    /* accept state */
            token_type = identifier;
            done = true;
            break;
        case 3:    /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;
```



Note that the transitions are encoded in the `next_state` matrix

Tables for the recognizer

Two tables control the recognizer

char_class

<i>char</i>	a-z	A-Z	0-9	other
<i>value</i>	letter	letter	digit	other

next_state

	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

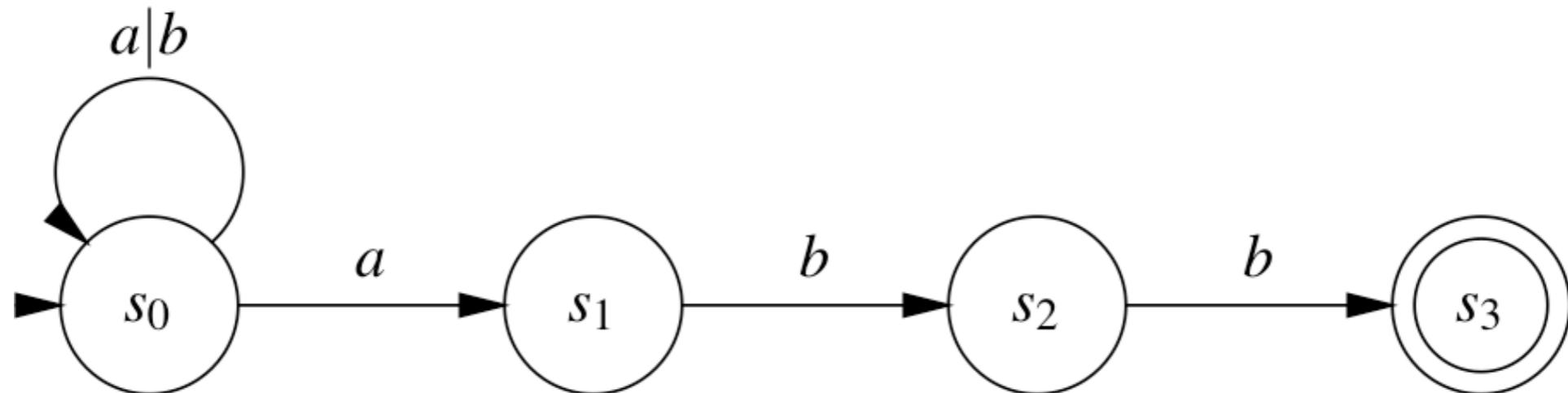
To change languages, we can just change tables

Automatic construction

- > *Scanner generators* automatically construct code from regular expression-like descriptions
 - construct a DFA
 - use *state minimization* techniques
 - emit code for the scanner (table driven or direct code)
- > A key issue in automation is an interface to the parser
- > *lex* is a scanner generator supplied with UNIX
 - emits C code for scanner
 - provides macro definitions for each token (used in the parser)
 - nowadays JavaCC, ANTLR, Bison etc. are more popular

NFA example

What about the RE $(a|b)^*abb$?



State s_0 has multiple transitions on a!

This is a non-deterministic finite automaton

Review: Finite Automata

A non-deterministic finite automaton (**NFA**) consists of:

1. a set of *states* $S = \{ s_0, \dots, s_n \}$
2. a set of *input symbols* Σ (the alphabet)
3. a transition function *move* (δ) mapping state-symbol pairs to sets of states
4. a distinguished *start state* s_0
5. a set of distinguished *accepting (final) states* F

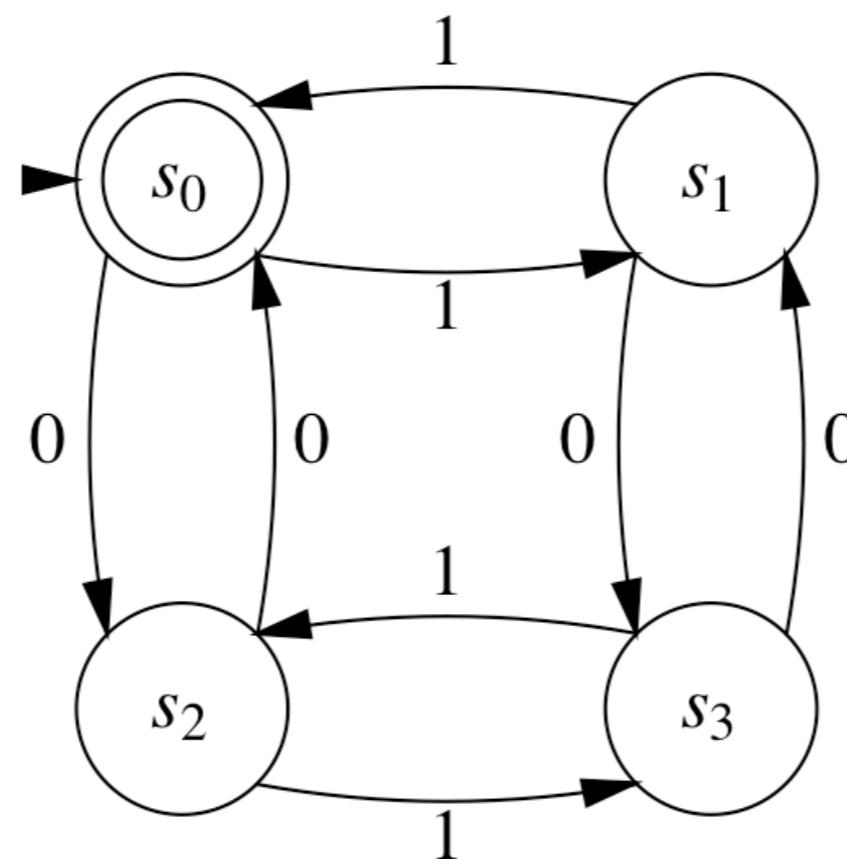
A Deterministic Finite Automaton (**DFA**) is a special case of an NFA:

1. no state has a ϵ -transition, and
2. for each state s and input symbol a , there is at most one edge labeled a leaving s .

A DFA accepts x iff there exists a *unique* path through the transition graph from the s_0 to an accepting state such that the labels along the edges spell x .

DFA example

Example: the set of strings containing an even number of zeros and an even number of ones



The RE is $(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$

Note how the RE walks through the DFA.

The states capture whether there are an even number or odd number of zeroes or ones. This gives 4 possible states.

Note how the RE effectively takes all possible paths through the DFA, always returning back to the start/accepting state: Initially we might just visit states s_1 and s_2 , always returning to s_0 , then we might visit s_3 via s_1 or s_2 , possibly loop back through s_1 or s_2 , return to s_0 , loop again through s_1 and s_2 without visiting s_3 , and then repeat any number of times.

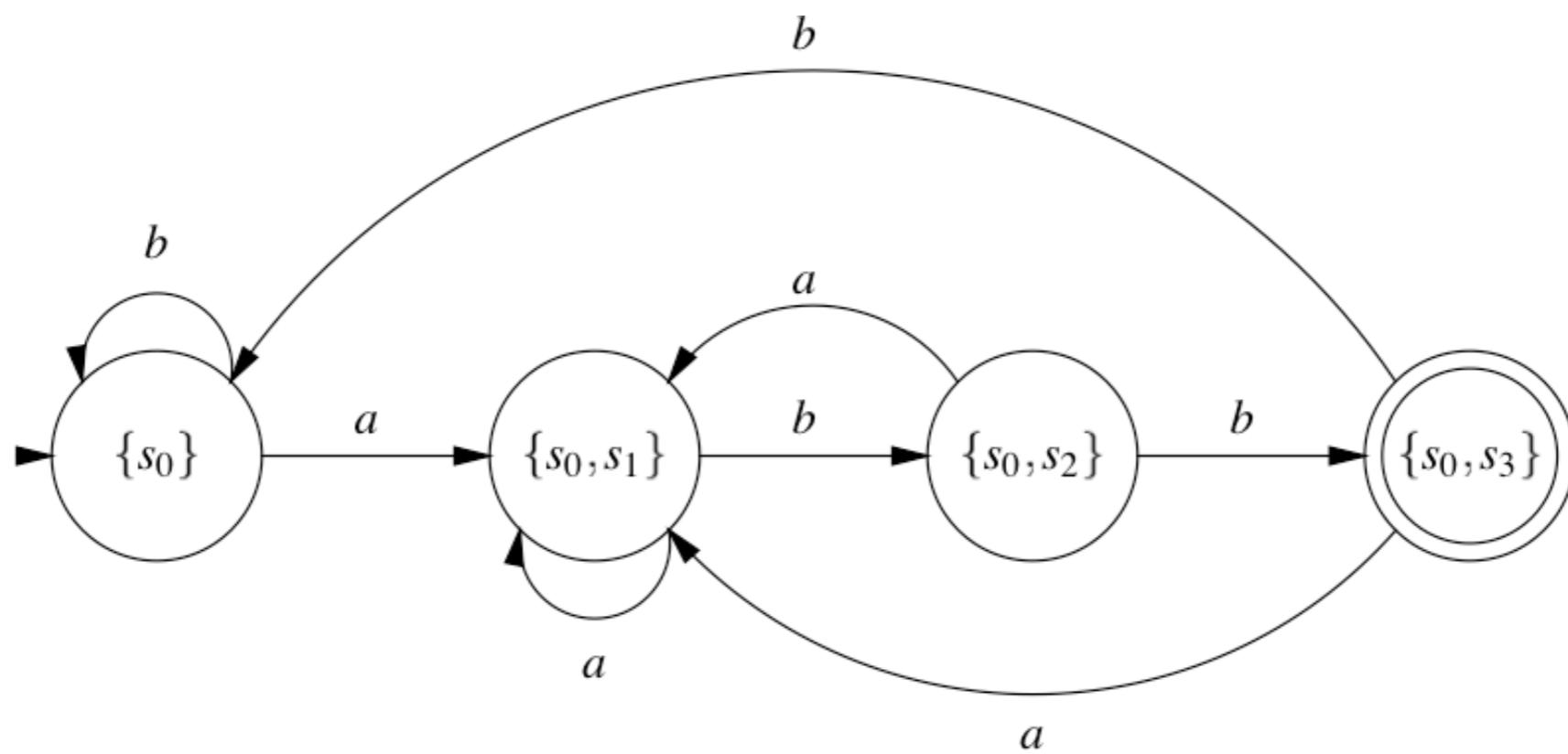
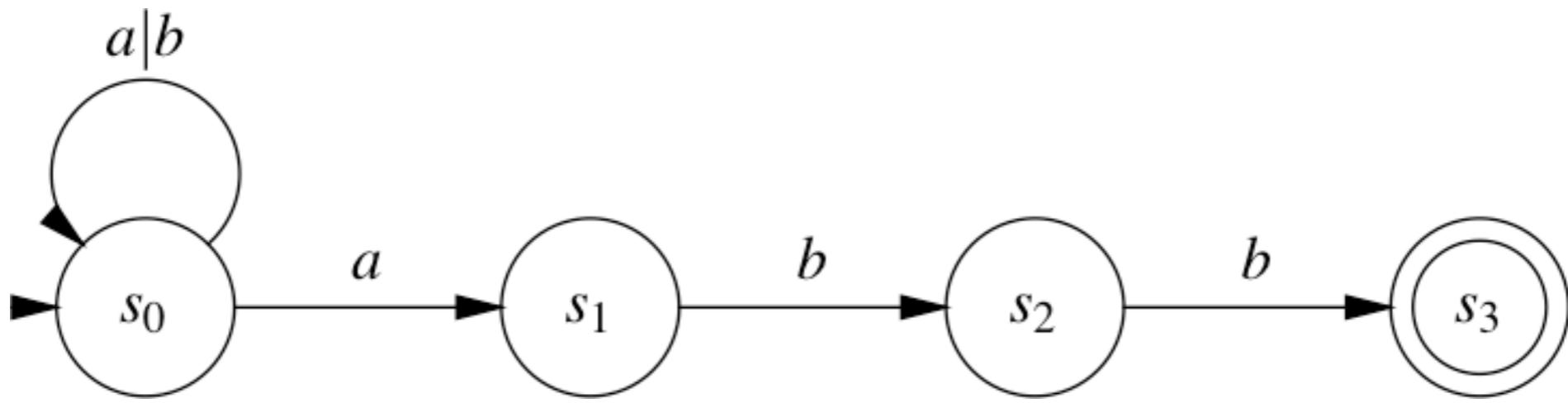
DFAs and NFAs are equivalent

1. DFAs are a subset of NFAs
2. Any NFA can be converted into a DFA, by *simulating sets of simultaneous states*:
 - each DFA state corresponds to a set of NFA states
 - NB: possible exponential blowup

The key idea to converting a NFA to a DFA is to construct a new DFA that simulates taking all possible paths simultaneously whenever there is a non-deterministic choice. The simulator then may be in multiple states at once. Since a DFA must always be in a unique state, the states of the DFA must be all possible subsets of the NFA states.

In theory this could blow up exponentially, but very often only few of these subsets are actually reachable in practice.

NFA to DFA using the subset construction



In the NFA we start in s_0 , so in the DFA we start in $\{s_0\}$. Then we simultaneously follow a to s_0 and s_1 , leading to DFA state $\{s_0, s_1\}$. Alternatively we can follow b back to $\{s_0\}$.

We continue to follow all possible transitions through the NFA, thus *generating only the reachable states* of the DFA.

Although there are 16 possible subsets of states of the NFA, only 4 states are actually reachable, thus avoiding any exponential explosion.

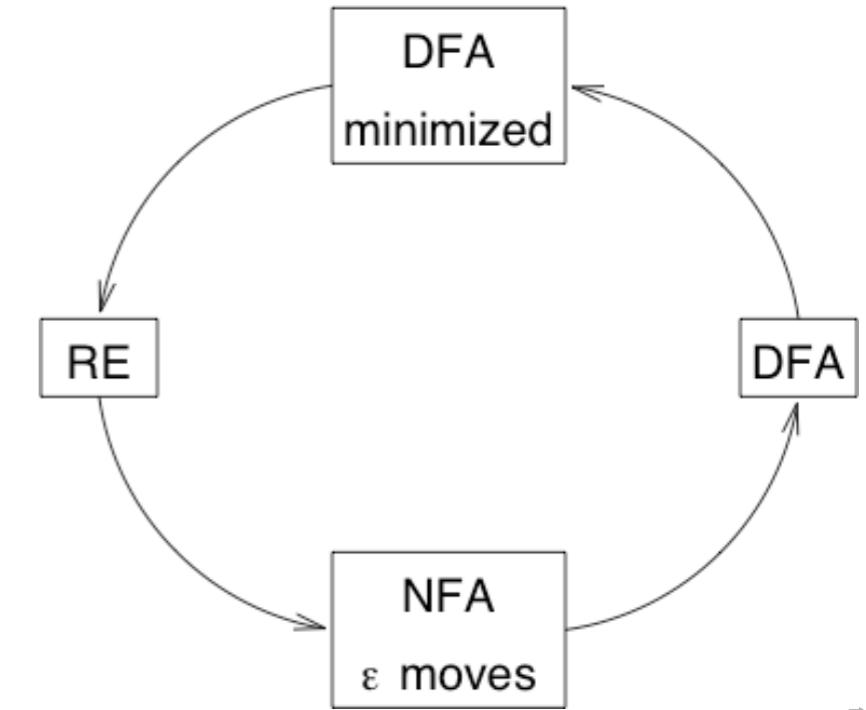
Roadmap



- > Introduction
- > Regular languages
- > Finite automata recognizers
- > **From RE to DFAs and back again**
- > Limits of regular languages

Constructing a DFA from a RE

- > RE → NFA
 - Build NFA for each term; connect with ϵ moves
- > NFA → DFA
 - Simulate the NFA using the subset construction
- > DFA → minimized DFA
 - Merge equivalent states
- > DFA → RE
 - Construct $R_{ij}^k = R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$
 - Or convert via Generalized NFA (GNFA)



Building a DFA from a regular expression requires several steps.

1. We build a NFA from the RE by using *templates* representing the individual subexpressions, and *wiring them together with ε transitions* (i.e., that can be taken silently without consuming input).
2. We convert the NFA to a DFA using the *simulation approach* we have seen earlier.
3. This may generate a DFA with “too many states”, so we apply a *minimization algorithm* that *merges equivalent states*.
4. We close the loop, showing how from a DFA we can construct the equivalent RE. To do this, we *iteratively rewrite the DFA, replacing labels on transitions by RE fragments*, until we end up with a trivial DFA with two states and a transition labeled with the RE that we want.

Roadmap

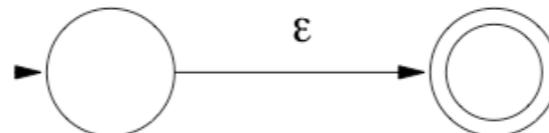
- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
 - > **RE to NFA**
 - > NFA to DFA
 - > DFA to minimized DFA
 - > DFA back to RE
- > Limits of regular languages



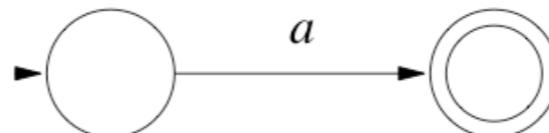
RE to NFA

$N(\varepsilon)$

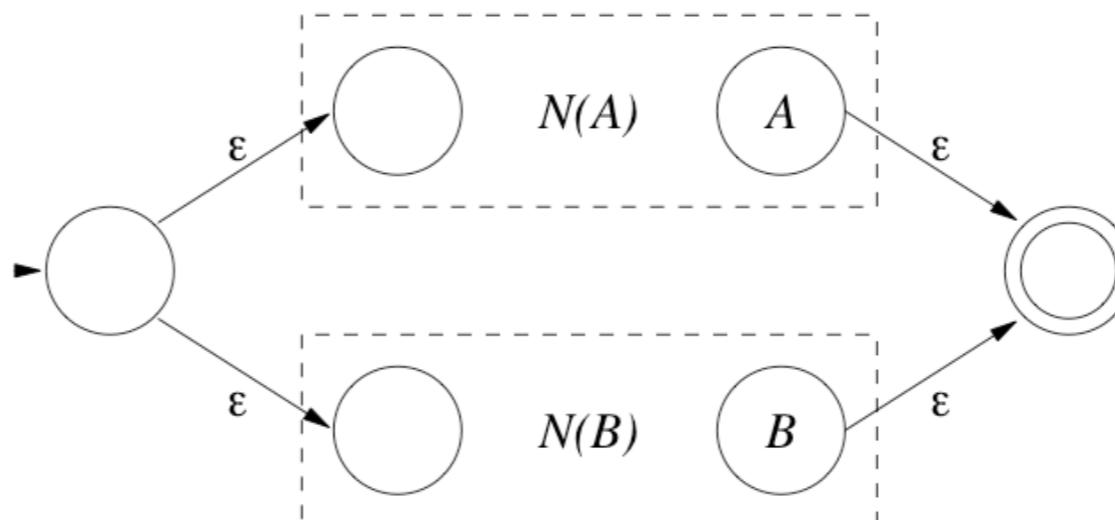
Start State



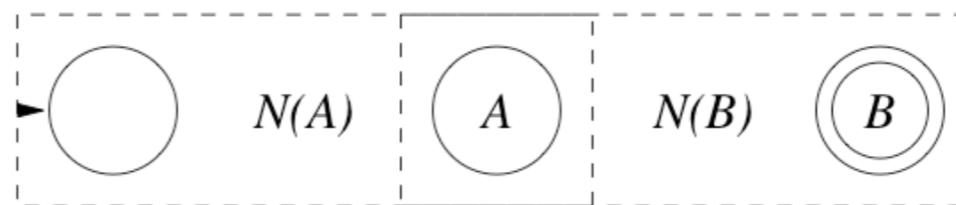
$N(a)$



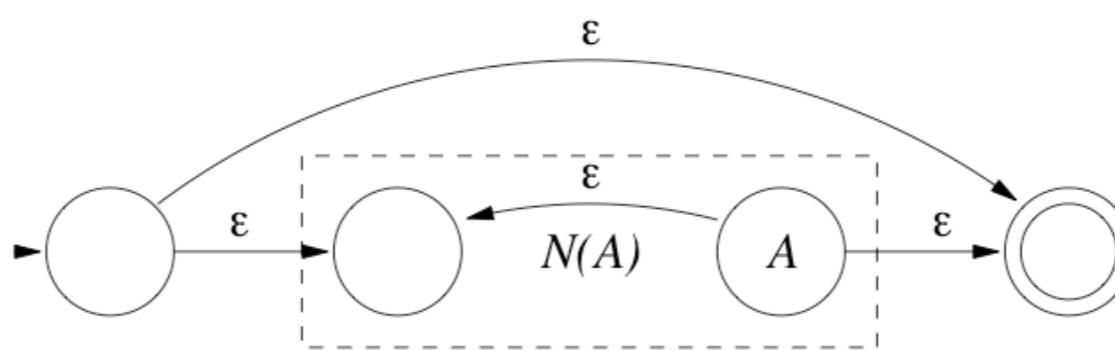
$N(A|B)$



$N(AB)$



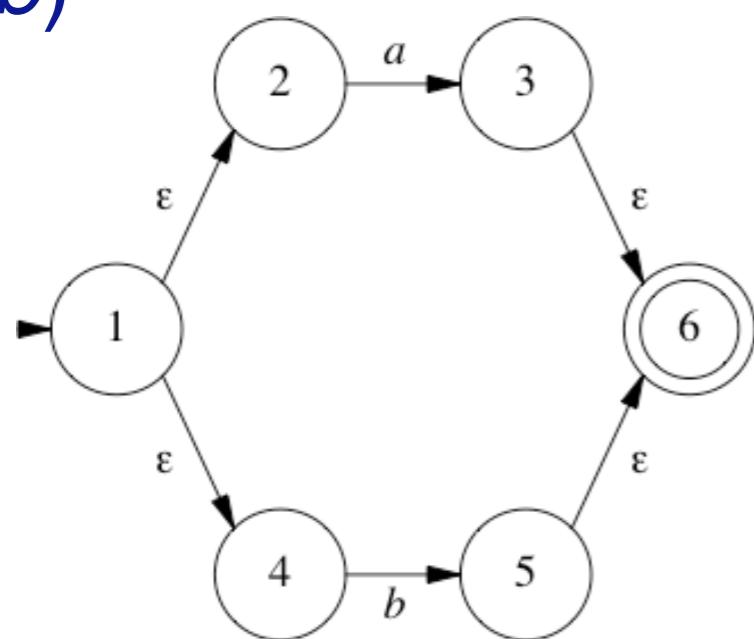
$N(A^*)$



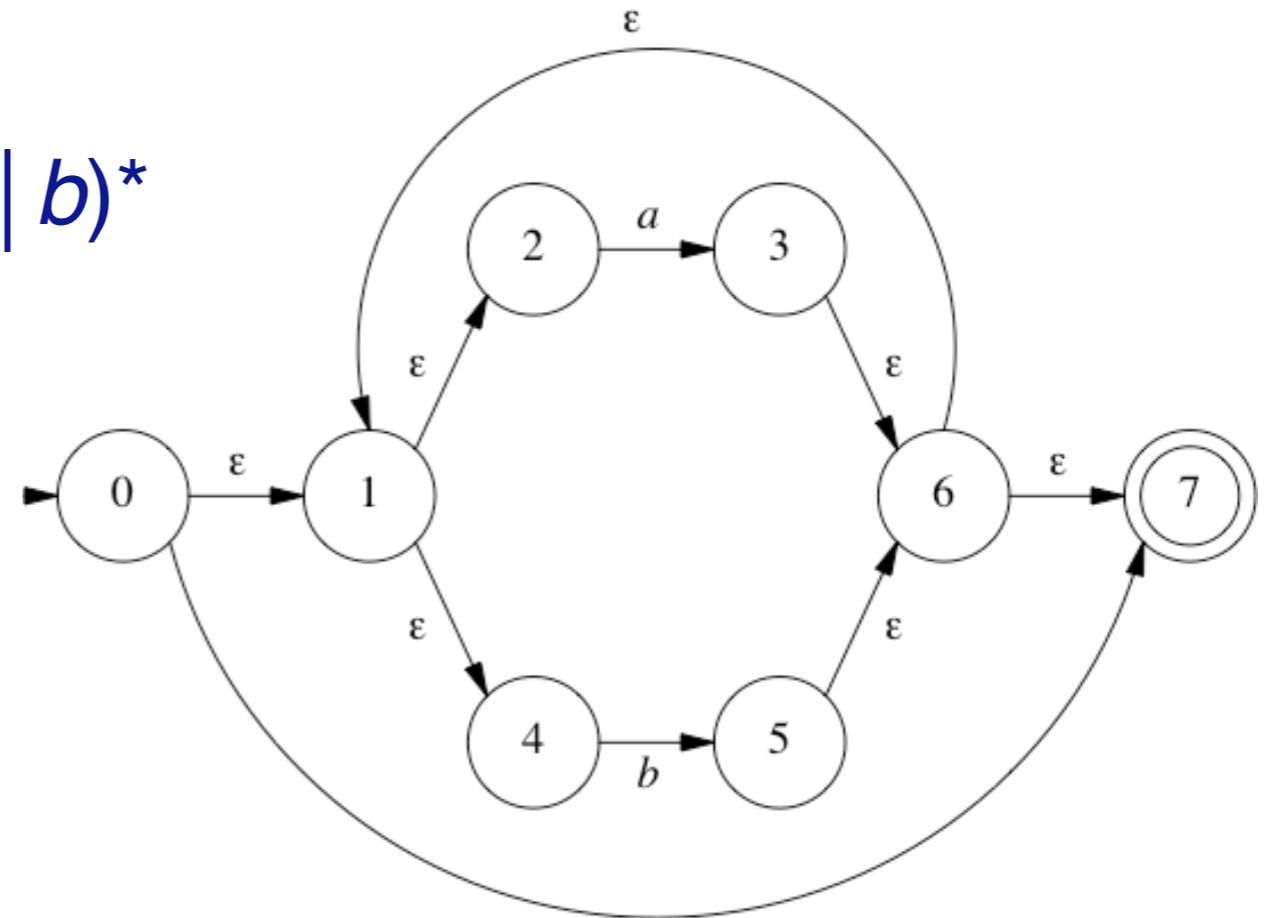
The function N takes as argument an RE and generates an equivalent NFA. N is specified in a recursive rule-based fashion over the syntax of an RE. The dotted regions in the right-hand side represent recursive invocations of N .

RE to NFA example: $(a \mid b)^*abb$

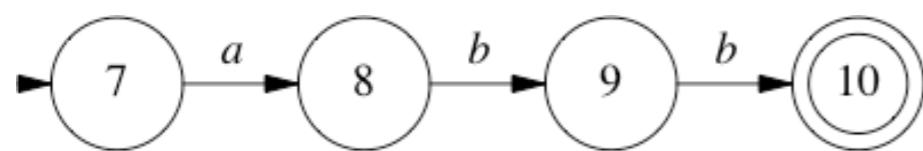
$(a \mid b)$



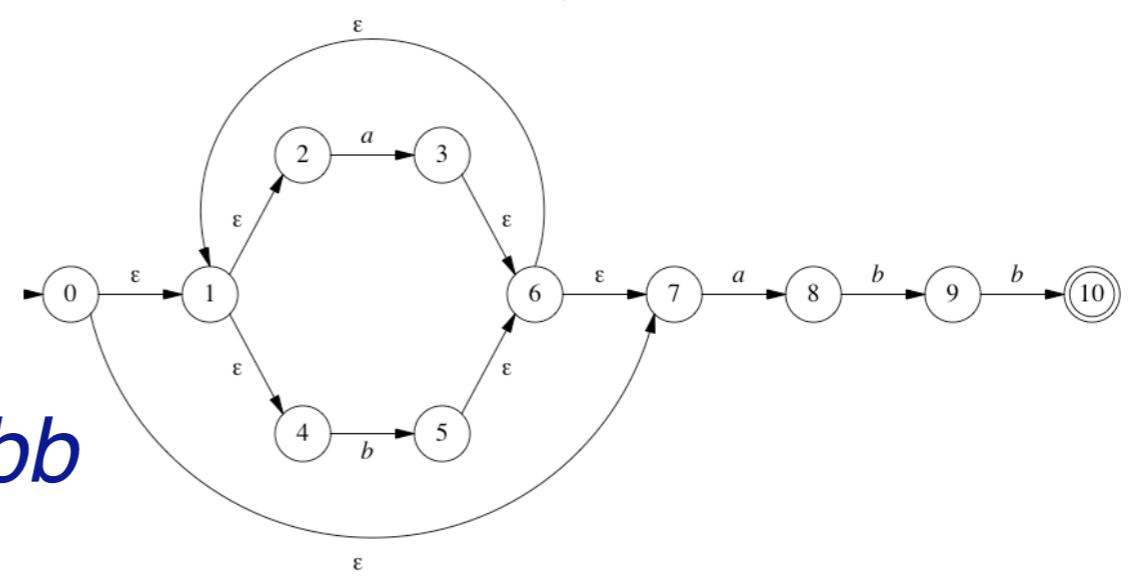
$(a \mid b)^*$



abb



$(a \mid b)^*abb$



To generate a NFA from the RE $(a|b)^*abb$, we first generate $N(a|b)$, combining the templates for $N(A|B)$ and $N(A)$.

Next we apply $N(A^*)$, adding two new states and three ϵ transitions.

We generate $N(abb)$ by combining the $N(A)$ and $N(AB)$ templates, and finally we wire the two parts together by merging the states labeled (7) representing the end of $(a|b)^*$ and the start of abb .

Roadmap

- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
 - > RE to NFA
 - > **NFA to DFA**
 - > DFA to minimized DFA
 - > DFA back to RE
- > Limits of regular languages



NFA to DFA: the subset construction

Input: NFA N

Output: DFA D with states S_D and transitions T_D such that $L(D) = L(N)$

Method: Let s be a state in N and P be a set of states. Use the following operations:

- > ϵ -closure(s) — set of states of N reachable from s by ϵ transitions alone
- > ϵ -closure(P) — set of states of N reachable from some s in P by ϵ transitions alone
- > move(T, a) — set of states of N to which there is a transition on input a from some s in P

add state $P = \epsilon$ -closure(s_0) unmarked to S_D

while \exists unmarked state P in S_D

mark P

for each input symbol a

$U = \epsilon$ -closure(move(P, a))

if $U \notin S_D$

then add U unmarked to S_D

$T_D[P, a] = U$

end for

end while

ϵ -closure(s_0) is the start state of D

A state of D is accepting if it contains an accepting state of N

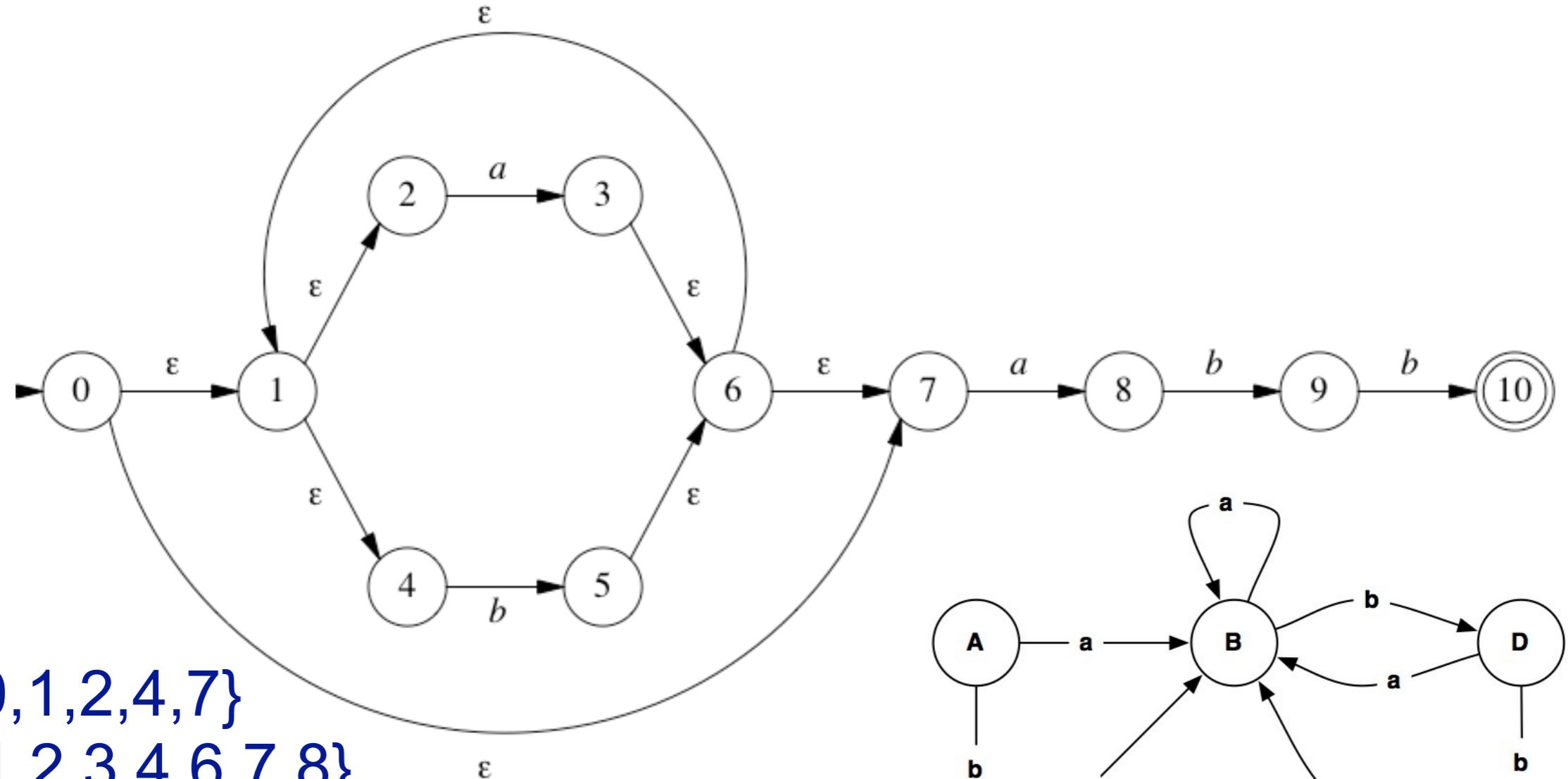
This algorithm simply formalizes the subset construction we saw earlier.

We begin at the start state s_0 of the input NFA N , and we use P to represent the set of states (a “multi-state”) that we reach by simultaneously taking all transitions with the same label. Every time we reach a new multi-state P , we add it to the DFA we are building, D .

We also add P to the set S_D of “unmarked” multi-states that we have yet to explore.

Whenever we take a P to explore, we “mark” it by removing it from S_D . Whenever we reach a “new” multi-state P we add it to D and S_D . When we run out of multi-states, we are done!

NFA to DFA using subset construction: example



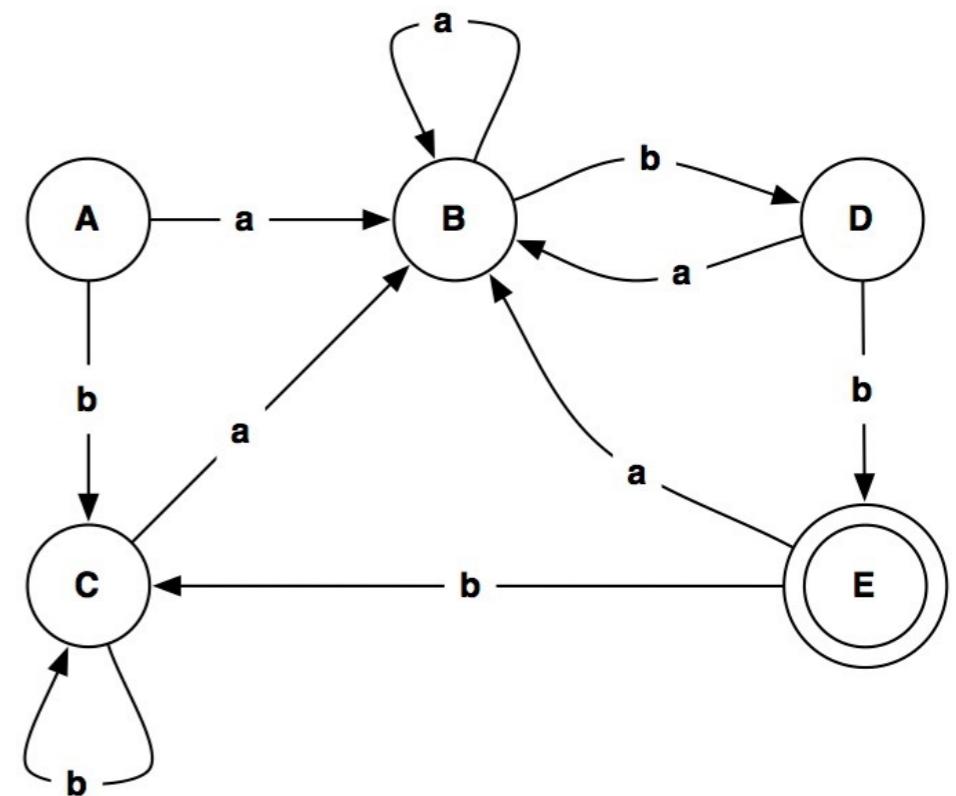
$$A = \{0, 1, 2, 4, 7\}$$

$$B = \{1, 2, 3, 4, 6, 7, 8\}$$

$$C = \{1, 2, 4, 5, 6, 7\}$$

$$D = \{1, 2, 4, 5, 6, 7, 9\}$$

$$E = \{1, 2, 4, 5, 6, 7, 10\}$$



We take as input the NFA at the top and generate the DFA below. From the start state 0, we can take ϵ moves to states 1, 2, 4 and 7, hence our start (multi-)state is $A = \{0,1,2,4,7\}$. We add this state to our DFA.

From A we can take transitions a (from either 2 or 7), or b (from 4). Following a leads us to states 3 or 8. Taking the ϵ -closure gives us $B = \{1,2,3,4,6,7,8\}$.

Following b from A leads us from state 4 to state 5, whose ϵ -closure is $C = \{1,2,4,5,6,7\}$.

We can now “mark” A and continue by exploring B and C. Eventually we reach state E, which includes the end state 10, and so represents a terminal state for the DFA.

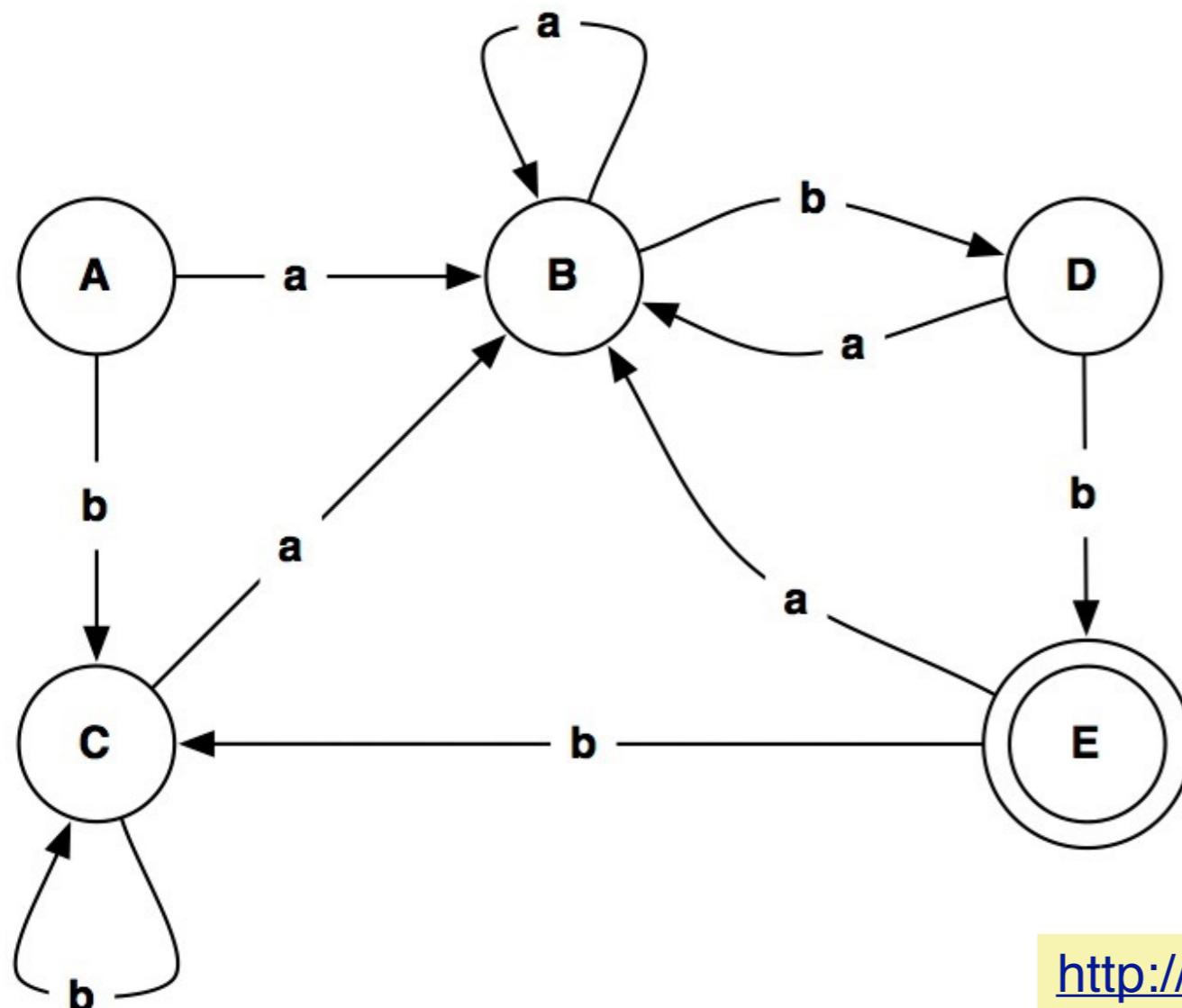
Roadmap

- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
 - > RE to NFA
 - > NFA to DFA
 - > **DFA to minimized DFA**
 - > DFA back to RE
- > Limits of regular languages



DFA Minimization

Theorem: For each regular language that can be accepted by a DFA, there exists a DFA with a minimum number of states.



Minimization approach:
merge *equivalent* states.

States A and C are
indistinguishable, so they
can be merged!

States A and C can be merged because after b^*a we always end up in state B.

This is analogous to the fact that $(a|bb^*a) = b^*a$.

DFA Minimization algorithm

- > Create lower-triangular table DISTINCT , initially blank
- > For every pair of states (p,q) :
 - If p_f is final and q is not, or vice versa
 - $\text{DISTINCT}(p_f, q) = \varepsilon$
- > Loop until no change for an iteration:
 - For every pair of states (p,q) and each symbol a
 - If $\text{DISTINCT}(p, q)$ is blank and $\text{DISTINCT}(\delta(p, a), \delta(q, a))$ is not blank
 - $\text{DISTINCT}(p, q) = a$
- > Combine all states that are not distinct

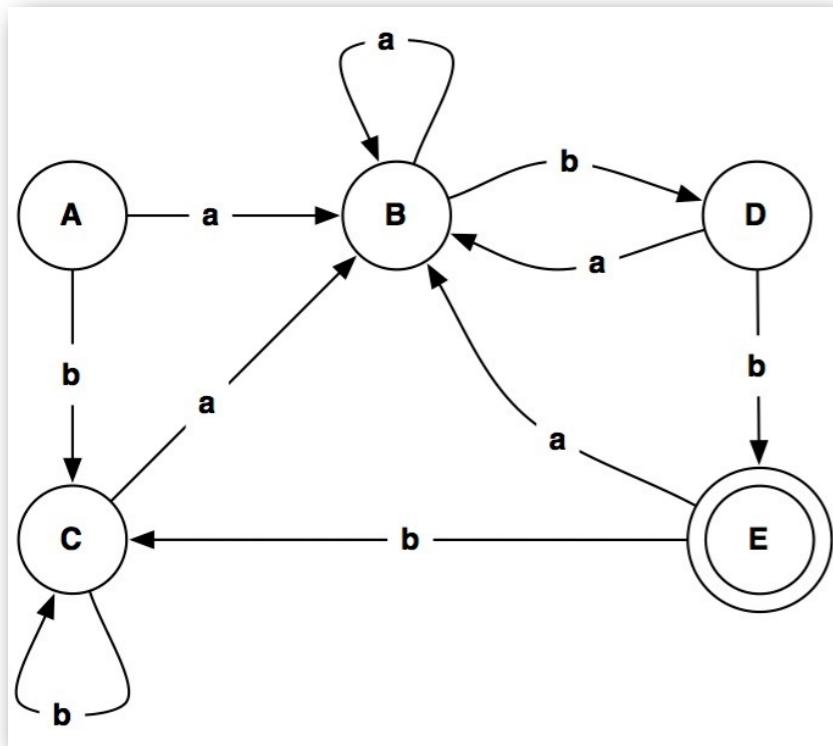
The table DISTINCT (initially blank) records for each state if it is distinct from every other state. Every state is the same as itself, and $\text{DISTINCT}(X,Y) \Leftrightarrow \text{DISTINCT}(Y,X)$, so we only need a lower triangle of the table.

Initially we only know that *the final state p_f is distinct from every other state q* , so we put the label ε to record this in $\text{DISTINCT}(p_f,q)$ for all such q .

We now *work backwards* looking for states with blank fields.

Suppose we *do not yet know if p and q are distinct*, i.e., $\text{DISTINCT}(p,q)$ is blank. Now suppose we can take an α transition from p to $\delta(p,\alpha)$ and from q to $\delta(q,\alpha)$, but $\delta(p,\alpha)$ and $\delta(q,\alpha)$ are DISTINCT, then we *can conclude that p and q are also distinct (since taking the same transition leads to distinct states)*. We record this as $\text{DISTINCT}(p,q) = \alpha$.

Minimization in action



C and A are *indistinguishable*
so can be merged

A	B	C	D	E
A				
B				
C				
D				
E				

A	B	C	D	E
A				
B				
C				
D				
E	ϵ	ϵ	ϵ	ϵ

A	B	C	D	E
A				
B				
C				
D	b	b	b	
E	ϵ	ϵ	ϵ	ϵ

A	B	C	D	E
A	b			
B				
C	b			
D	b	b	b	
E	ϵ	ϵ	ϵ	ϵ

A	B	C	D	E
A	b			
B	b			
C	b			
D	b	b	b	
E	ϵ	ϵ	ϵ	ϵ

E is the final state, so distinct from all other states. We mark all its squares with ε .

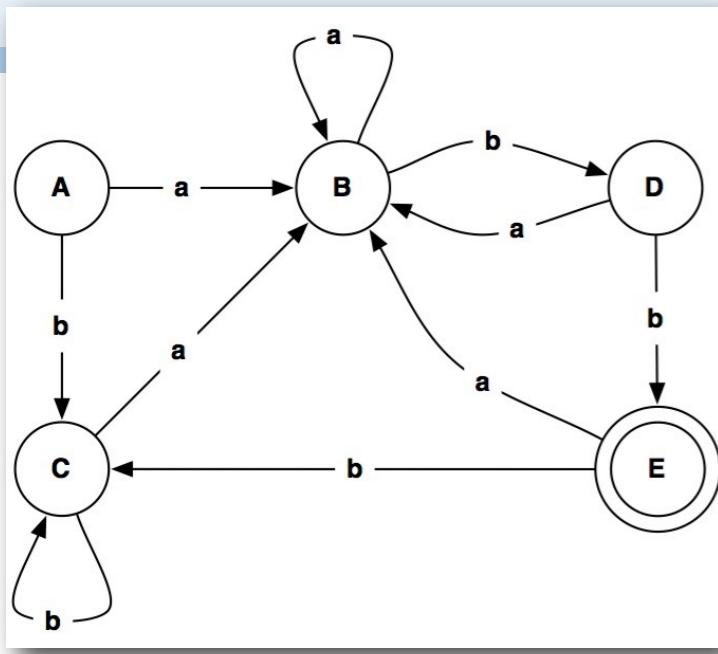
Now we see that D has a b-transition to E and has blank entries, but no other state has b-transitions to E. We therefore mark all its squares with b. (The mark is the “proof” of distinctness: a b-move takes A and C both to C, and B to D. Since C and D are both distinct from E, we know that D is distinct from A, B and C.)

Now we note that B can take a b-move to D, but neither A nor C can, so we mark those squares with b.

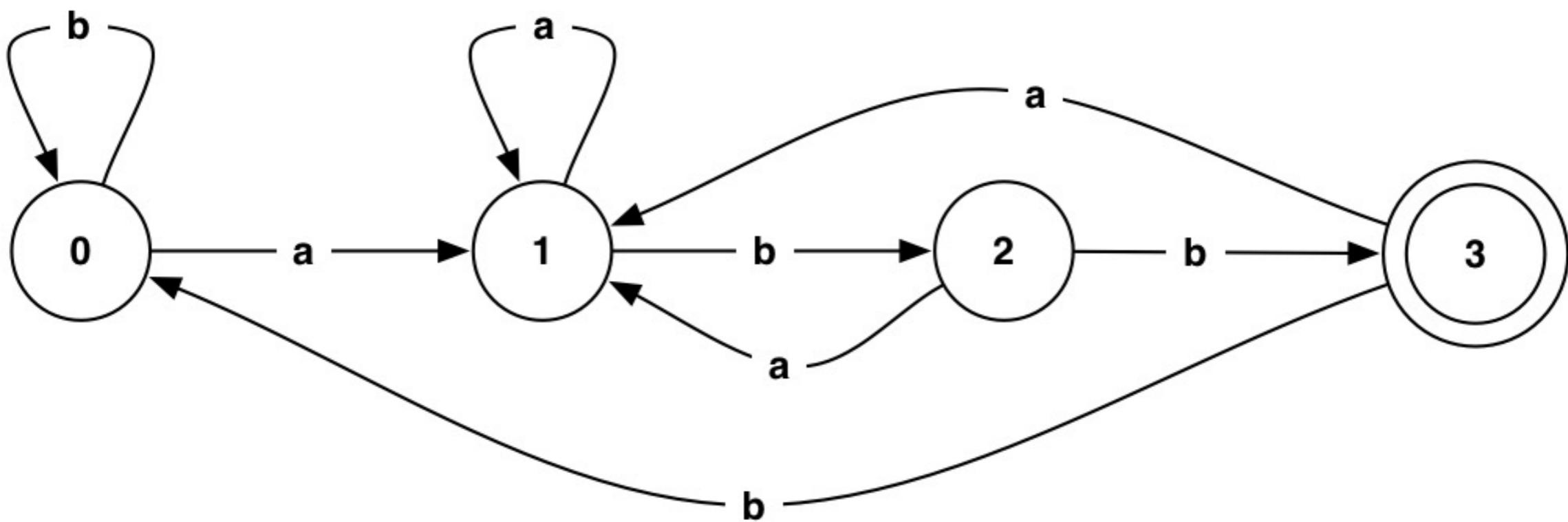
We are left with A and C. An a-move brings both to B and a b-move brings both to C, so we cannot distinguish them.

There are no other blank squares left, so we are done, and A and C can be merged.

DFA Minimization example



It is easy to see that this is in fact the minimal DFA for $(a \mid b)^*abb \dots$



After merging A and C, we get the new DFA below ($A/C=0$, $B=2$, $D=2$, $E=3$).

Actually it is easy to see that this is the minimal DFA:

- Start with the required path abb. This gives us 4 states. Now add the missing arrows.
- Any a transition brings us to state 1, since we must follow with bb.
- Any b not in the path brings us back to state 0, since we must still follow with abb.

Roadmap

- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
 - > RE to NFA
 - > NFA to DFA
 - > DFA to minimized DFA
 - > **DFA back to RE**
- > Limits of regular languages



DFA to RE via GNFA

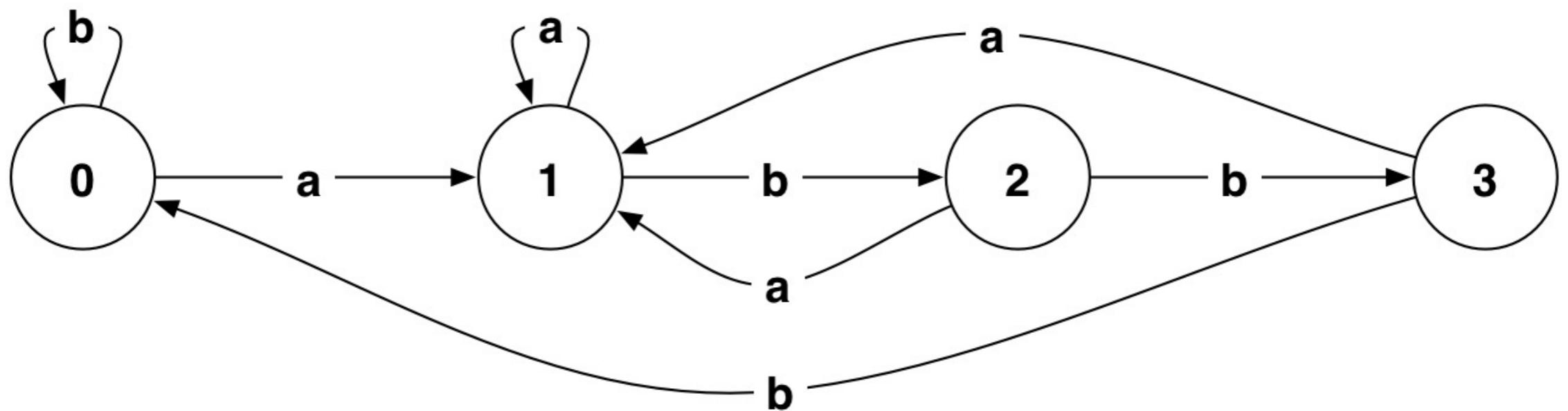
- > A Generalized NFA is an NFA where transitions may have any RE as labels
- > Conversion algorithm:
 1. *Add a new start state and accept state* with ϵ -transitions to/from the old start/end states
 2. *Merge multiple transitions* between two states to a single RE choice transition
 3. *Add empty \emptyset -transitions* between states where missing
 4. *Iteratively “rip out” old states* and replace “dangling transitions” with appropriately labeled transitions between remaining states
 5. *STOP when all old states are gone* and only the new start and accept states remain

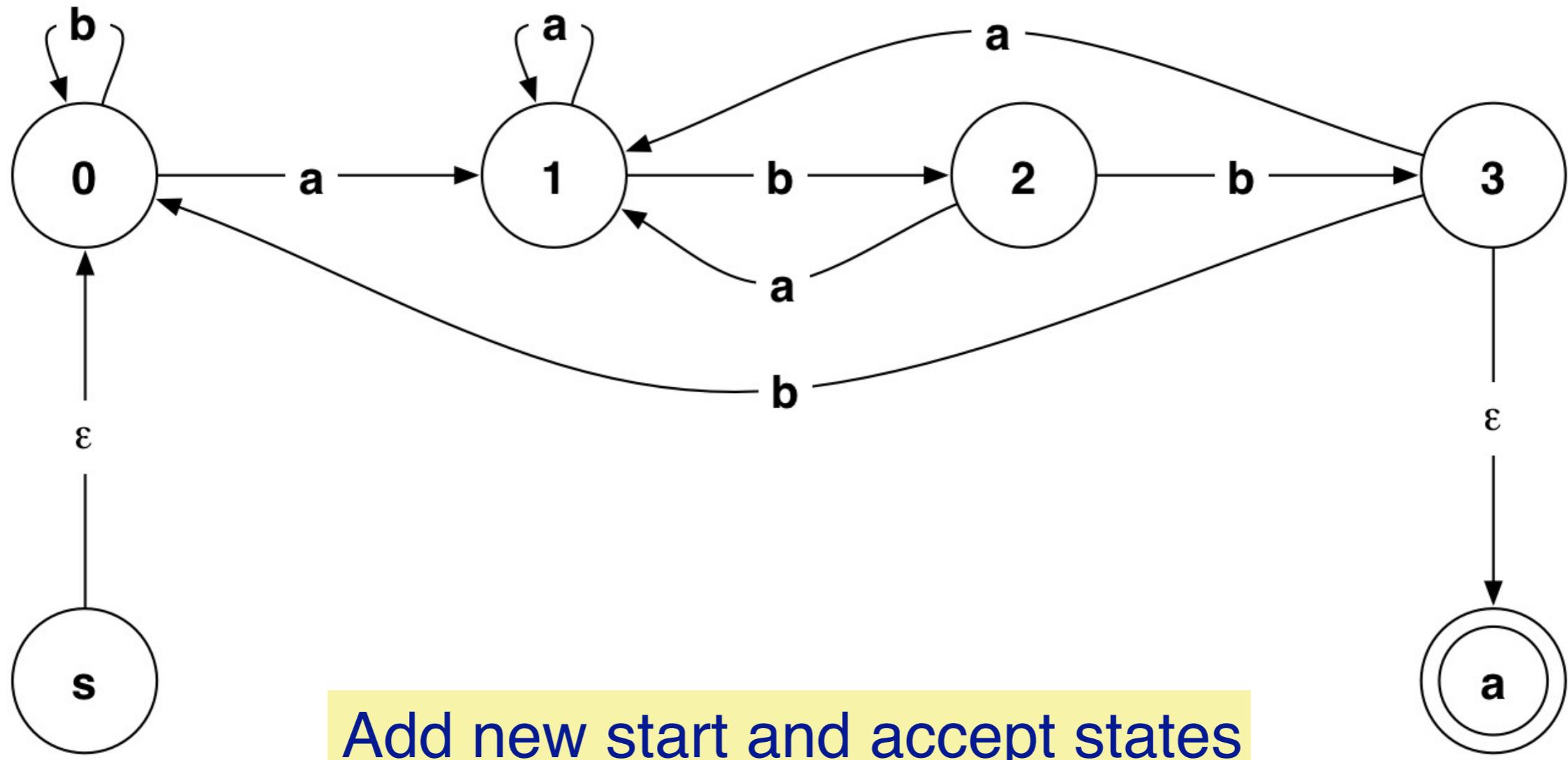
The idea is that we iteratively simplify the GNFA, deleting states, but maintain equivalence by making the transitions more complex. At the end, we have a completely trivial GNFA with only two states, but the transitions (an RE) expresses the whole GNFA.

GNFA conversion algorithm

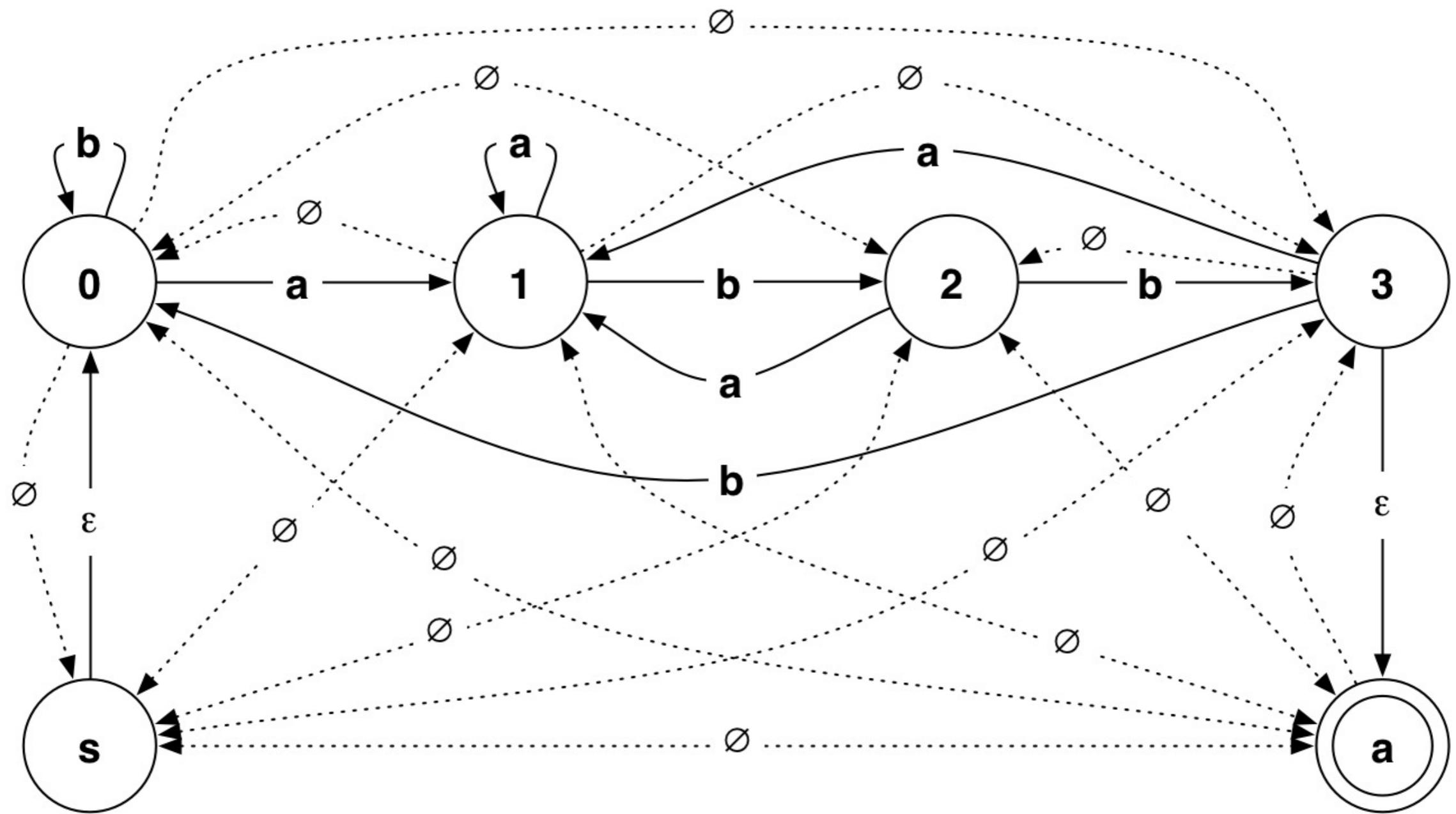
1. Let k be the number of states of G , $k \geq 2$
2. If $k=2$, then RE is the label found between q_s and q_a (start and accept states of G)
3. While $k > 2$, select $q_{rip} \neq q_s$ or q_a
 - $Q' = Q - \{q_{rip}\}$
 - For any $q_i \in Q' - \{q_a\}$ let $\delta'(q_i, q_j) = R_1 R_2^* R_3 \cup R_4$ where:
 $R_1 = \delta'(q_i, q_{rip})$, $R_2 = \delta'(q_{rip}, q_{rip})$, $R_3 = \delta'(q_{rip}, q_j)$, $R_4 = \delta'(q_i, q_j)$
 - Replace G by G'

The initial DFA



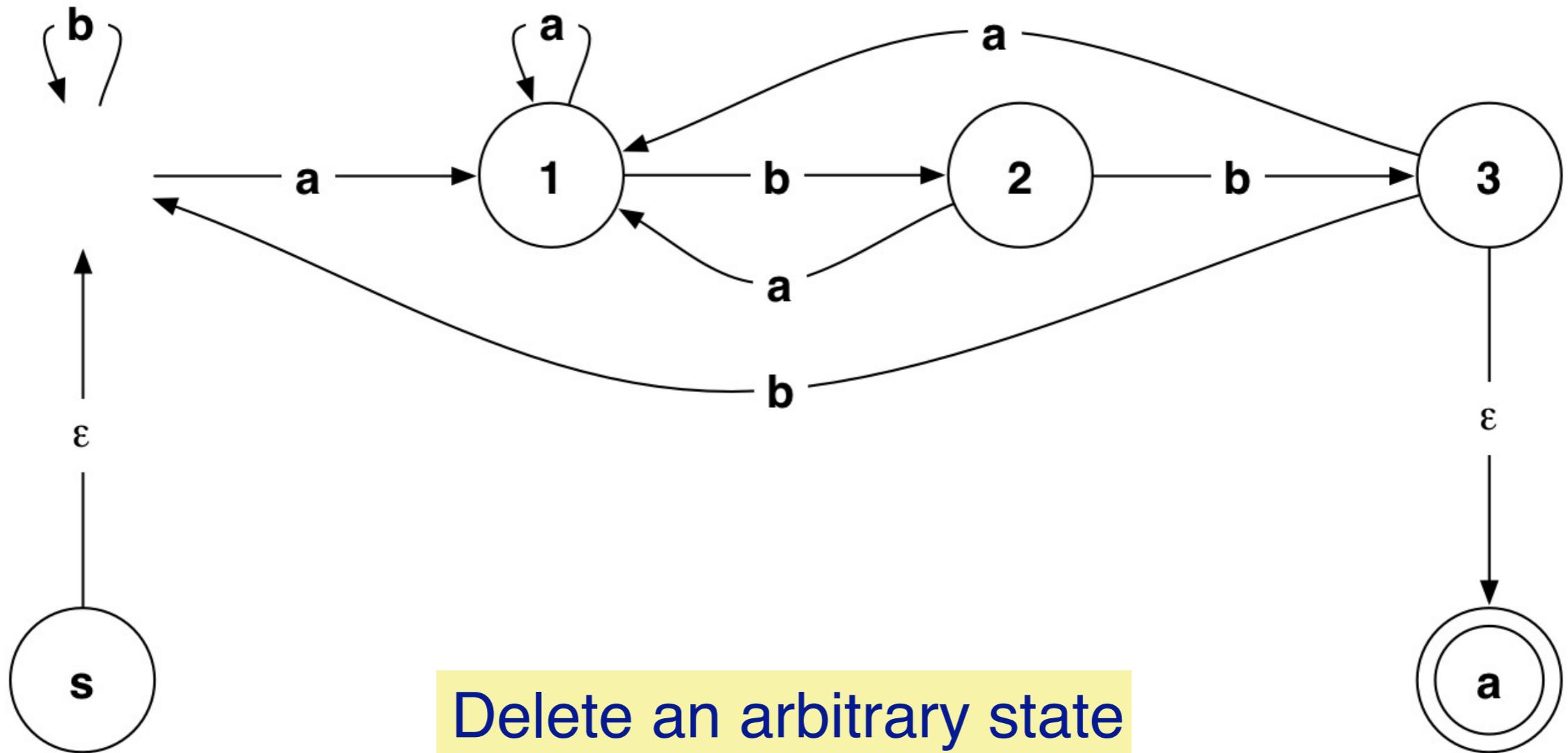


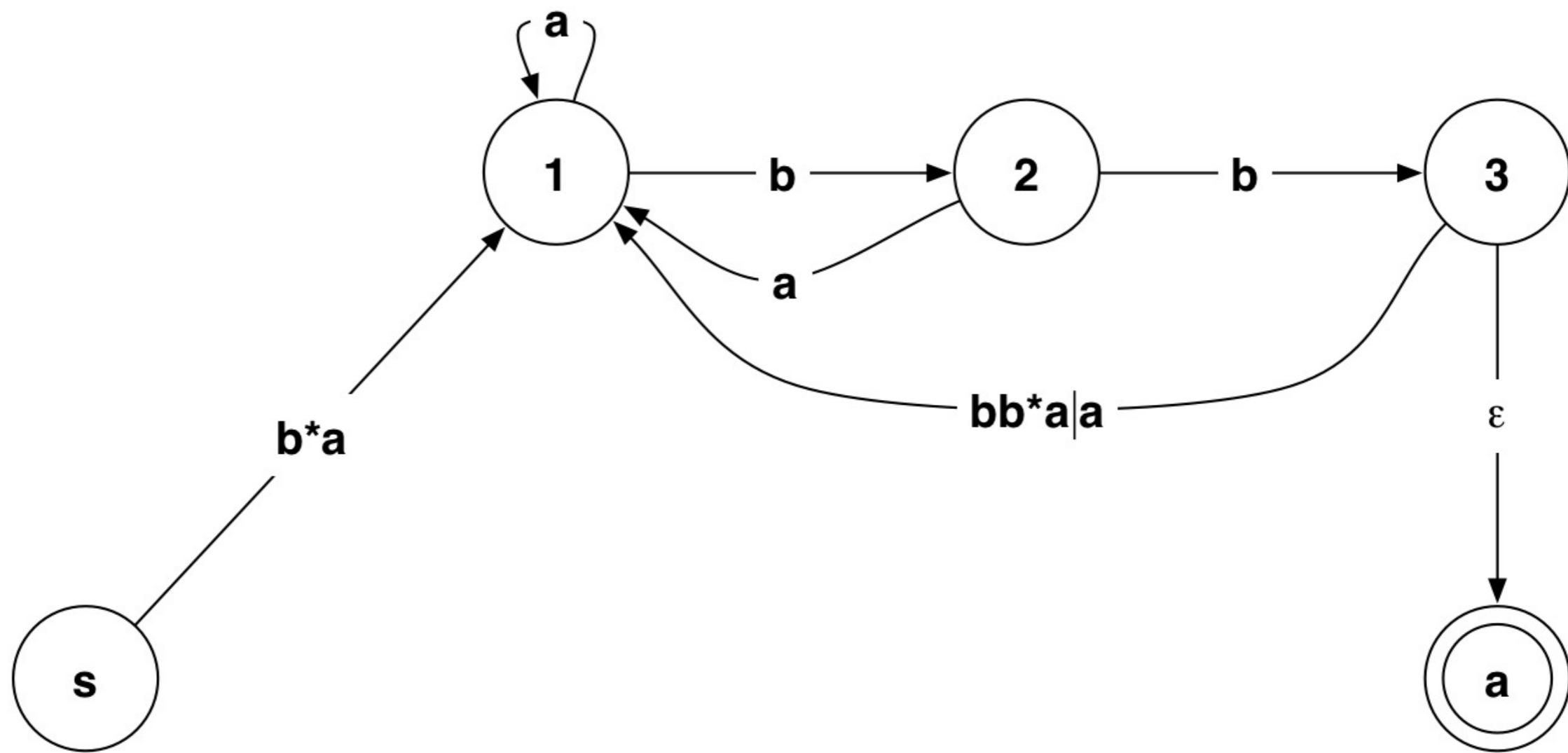
Add new start and accept states



Add missing empty transitions
(we'll just pretend they're there)

An “empty transition” expresses that fact that “you can’t get there from here”. For example, you cannot get from 0 to s.



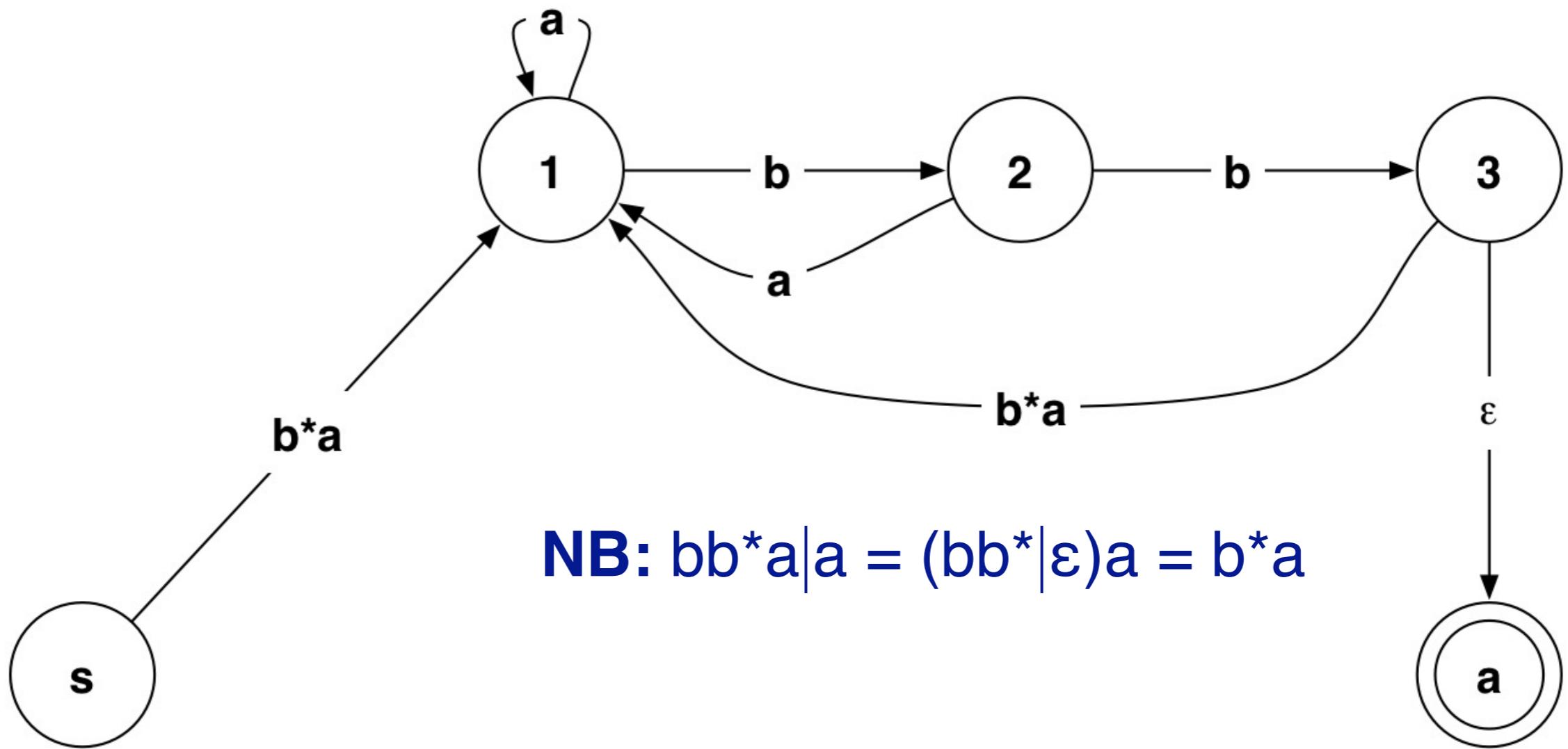


Fix dangling transitions $s \rightarrow 1$ and $3 \rightarrow 1$
 Don't forget to merge the existing transitions!

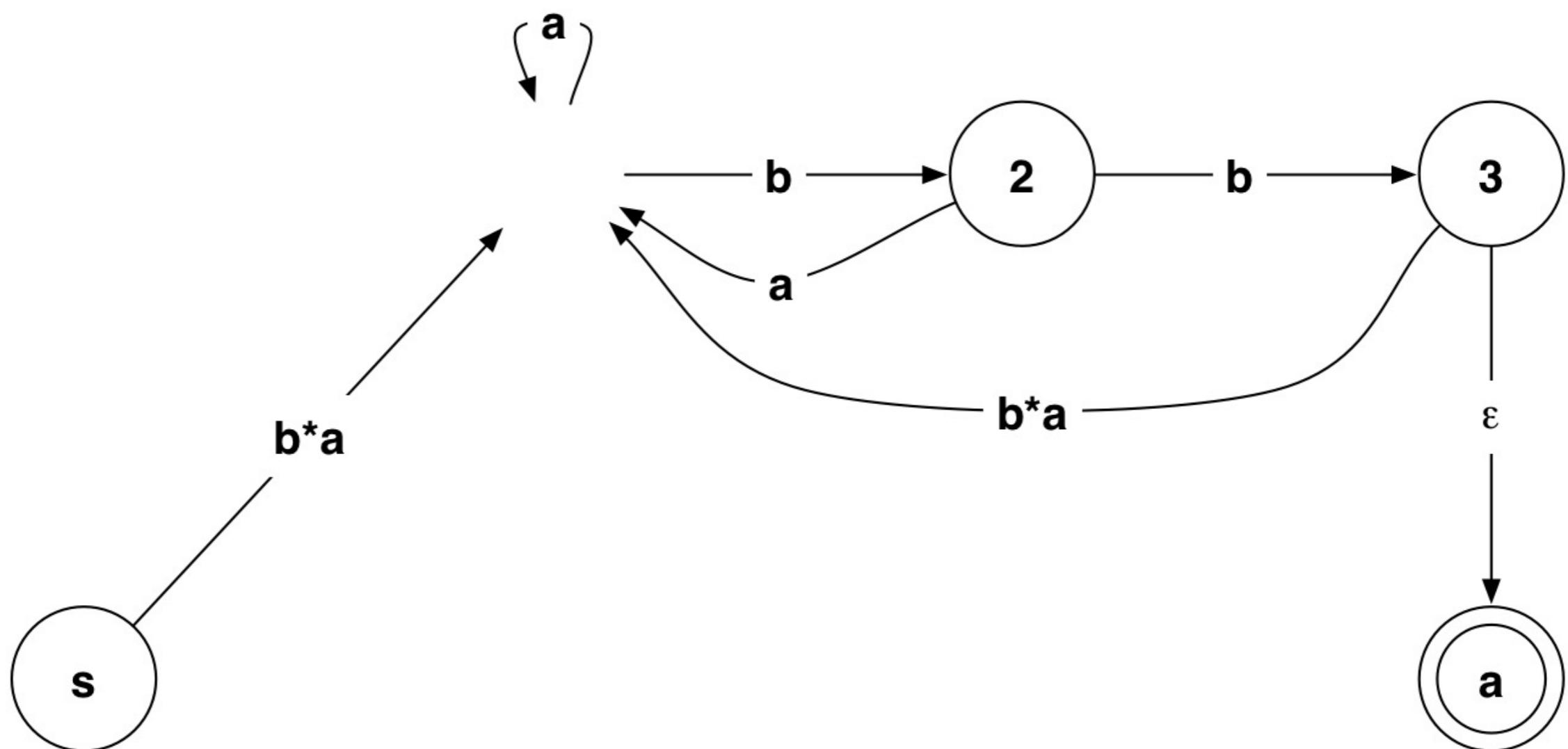
We have to “repair” all transitions that go through the deleted node, in particular from s to 1 and from 3 to 1.

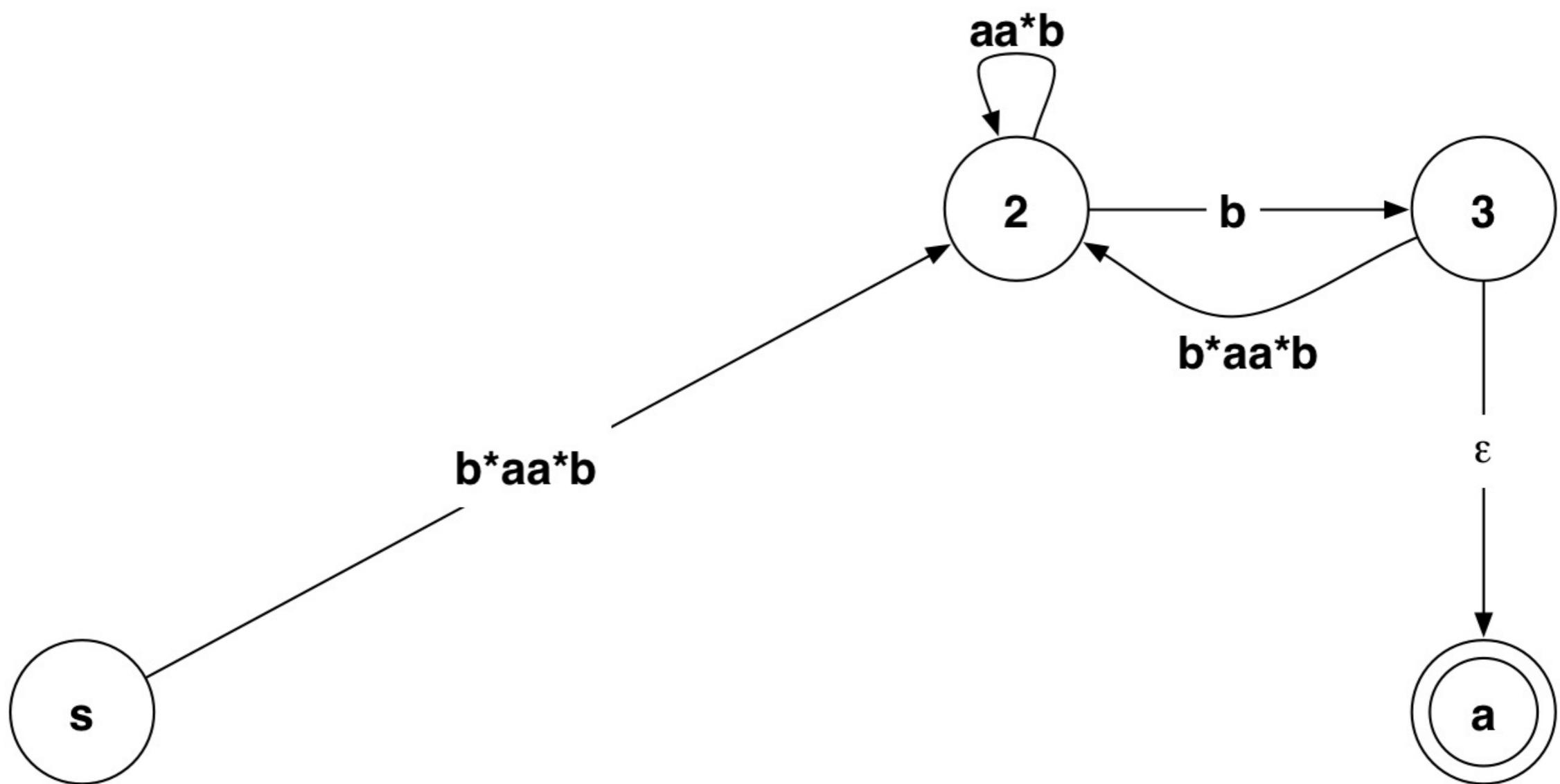
The RE from s to 1 is clearly b^*a .

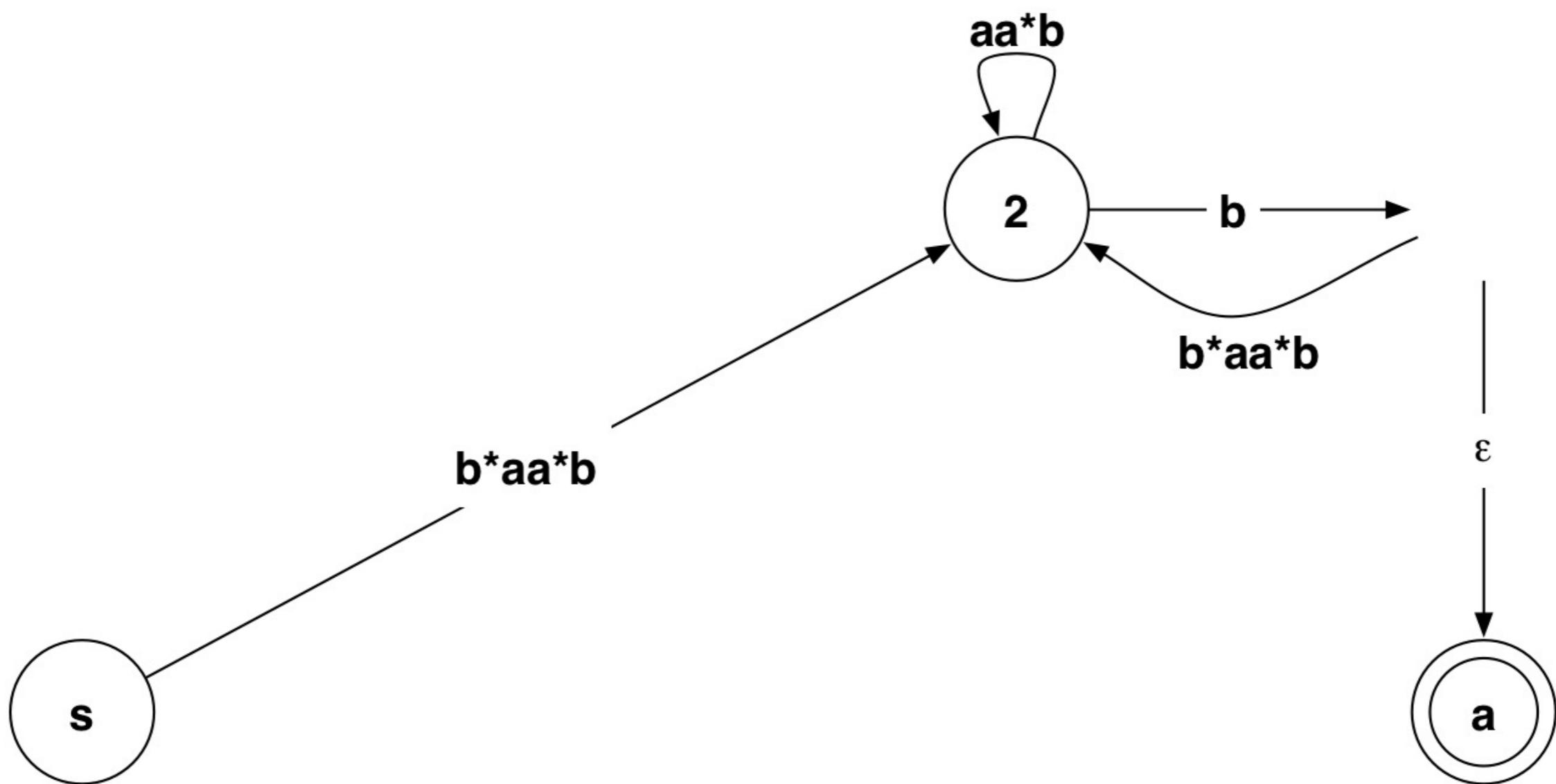
Note that there were two original paths from 3 to 1: the path bb^*a via the deleted state 0, as well as the transition a directly from 3 to 1. Merging these yields $bb^*a|a$ (which is the same as b^*a).

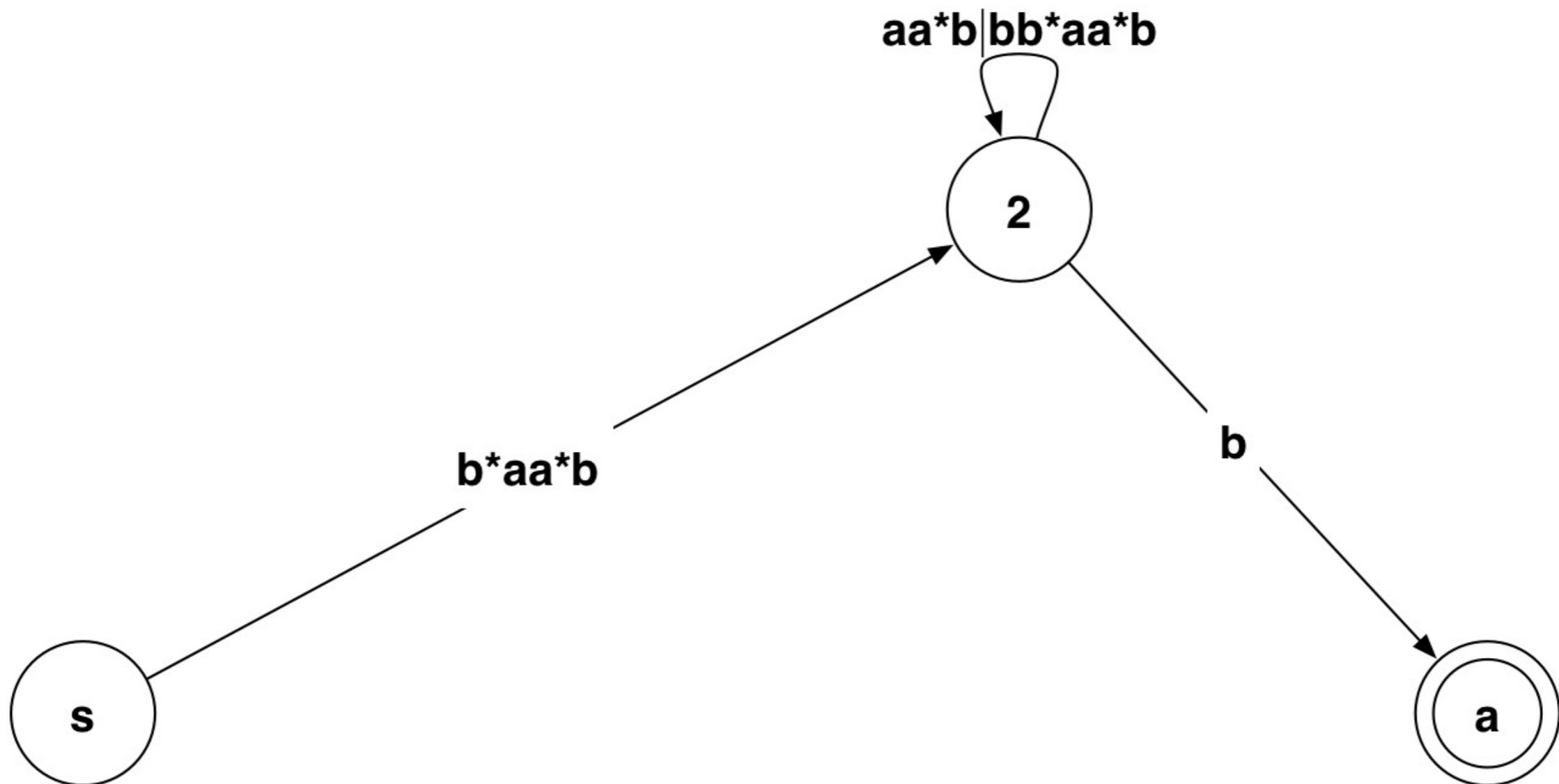


Simplify the RE Delete another state

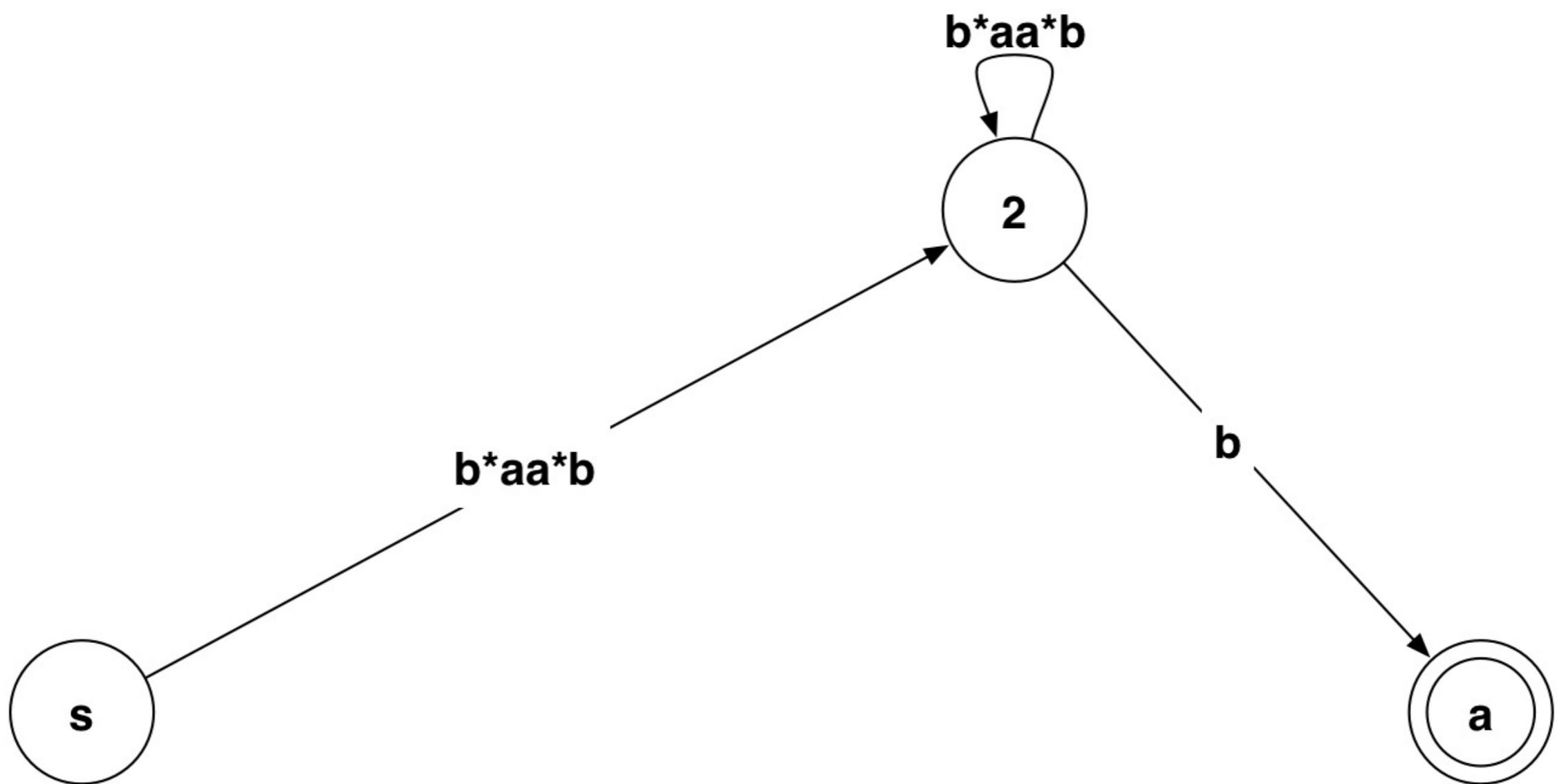


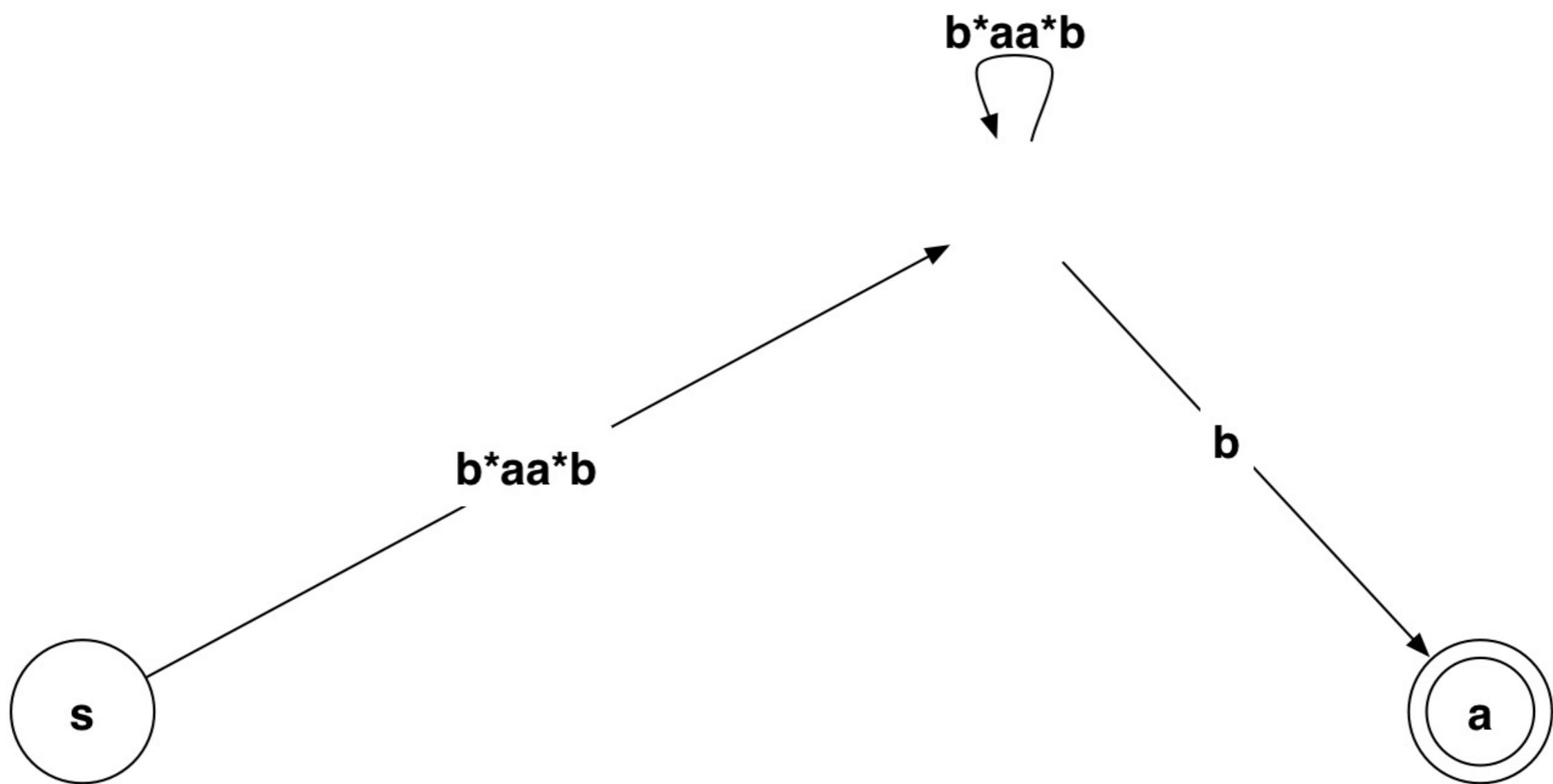




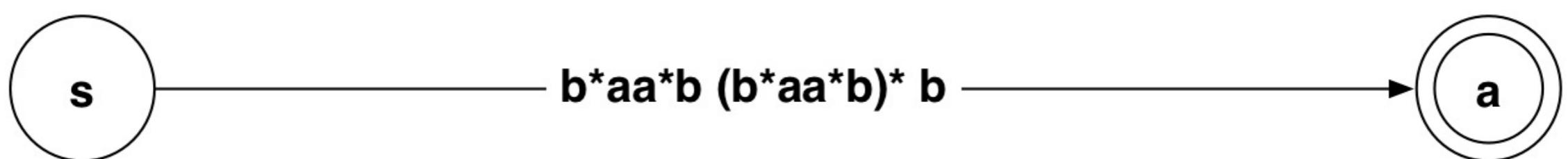


NB: $aa^*b|bb^*aa^*b = (\epsilon|bb^*)aa^*b = b^*aa^*b$





Hm ... not what we expected



$b^*aa^*b (b^*aa^*b)^* b = (a|b)^*abb ?$

- > *We can rewrite:*
 - $b^*aa^*b (b^*aa^*b)^* b$
 - $b^*a^*ab (b^*a^*ab)^* b$
 - $(b^*a^*ab)^* b^*a^* abb$

- > *But does this hold?*
 - $(b^*a^*ab)^* b^*a^* = (a|b)^*$

We can show that the minimal DFAs for these REs are isomorphic ...

Proof: Split any string in $(a|b)^*$ by occurrences of ab . This will match $(Xab)^*X$, where X does not contain ab . X is clearly b^*a^* .

Proof #2 (by @grammarware):

$$\begin{aligned}(b^*a^*ab)^*b^*a^* &= (b^*a^+b)^*b^*a^* \\&= b^*(a^+b^+)^*a^* \\&= b^*(b^*|(a^+b^+)^*)a^* \\&= b^*(b^*|(a^+b^+)^*|a^*)a^* \\&= b^*(a|b)^*a^* \\&= (a|b)^*a^* \\&= (a|b)^*\end{aligned}$$

Roadmap



- > Introduction
- > Regular languages
- > Finite automata recognizers
- > From RE to DFAs and back again
- > **Limits of regular languages**

Limits of regular languages

Not all languages are regular!

One cannot construct DFAs to recognize these languages:

$$L = \{ p^k q^k \}$$

$$L = \{ w c w^r \mid w \in \Sigma^*, w^r \text{ is } w \text{ reversed} \}$$

In general, DFAs cannot count!

However, one *can* construct DFAs for:

- Alternating 0's and 1's:
 $(\epsilon \mid 1)(01)^*(\epsilon \mid 0)$
- Sets of pairs of 0's and 1's
 $(01 \mid 10)^+$

So, what is hard?

Certain language features can cause problems:

- > Reserved words
 - PL/I had no reserved words
 - if then then then = else; else else = then
- > Significant blanks
 - FORTRAN and Algol68 ignore blanks
 - do 10 i = 1,25
 - do 10 i = 1.25
- > String constants
 - Special characters in strings
 - Newline, tab, quote, comment delimiter
- > Finite limits
 - Some languages limit identifier lengths
 - Add state to count length
 - FORTRAN 66 — 6 characters(!)

How bad can it get?

```
1      INTEGERFUNCTIONA
2      PARAMETER(A=6,B=2)
3      IMPLICIT CHARACTER*(A-B)(A-B)
4      INTEGER FORMAT(10),IF(10),D09E1
5      100  FORMAT(4H)=(3)
6      200  FORMAT(4 )=(3)
7      D09E1=1
8      D09E1=1,2
9          IF(X)=1
10         IF(X)H=1
11         IF(X)300,200
12      300      CONTINUE
13      END
14      C      this is a comment
15      $ FILE(1)
16      END
```

*Compiler needs context
to distinguish variables
from control constructs!*

Example due to Dr. F.K. Zadeck of IBM Corporation

What you should know!

- ❖ *What are the key responsibilities of a scanner?*
- ❖ *What is a formal language? What are operators over languages?*
- ❖ *What is a regular language?*
- ❖ *Why are regular languages interesting for defining scanners?*
- ❖ *What is the difference between a deterministic and a non-deterministic finite automaton?*
- ❖ *How can you generate a DFA recognizer from a regular expression?*
- ❖ *Why aren't regular languages expressive enough for parsing?*

Can you answer these questions?

- ❖ *Why do compilers separate scanning from parsing?*
- ❖ *Why doesn't NFA → DFA translation normally result in an exponential increase in the number of states?*
- ❖ *Why is it necessary to minimize states after translation a NFA to a DFA?*
- ❖ *How would you program a scanner for a language like FORTRAN?*



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>