UNIVERSITÄT BERN

2. Lexical Analysis

Prof. O. 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

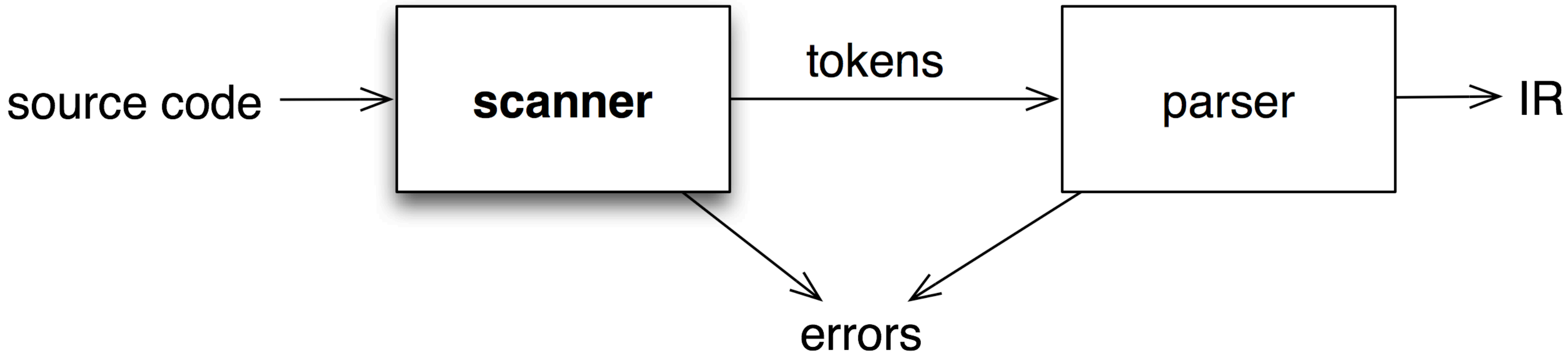
- Regular languages
- Finite automata recognizers
- From regular expressions to deterministic finite automata, and back
- Limits of regular languages

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

Roadmap

- **Regular languages**
- Finite automata recognizers
- From regular expressions to deterministic finite automata, and back
- Limits of regular languages

Scanner



- map characters to ***tokens***

```
x = x + y   →   <id,x> = <id,x> + <id,y>
```

- character string value for a token is a ***lexeme***
- eliminates white space (tabs, blanks, comments *etc.*)
- a key issue is ***speed*** ⇒ use specialized recognizer

Specifying patterns

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 much harder:

Identifiers

alphabetic 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!

Operations on languages

A *language* is a set of strings

Operation	***Definition***
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	$L^+ = \cup_{I=1,\infty} L^i$

Regular expressions describe regular languages

> *Regular expressions over an alphabet Σ:*

1. ε is a RE denoting the set $\{\varepsilon\}$
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) is a RE denoting L(r)
   - (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)*

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.

Examples

identifier

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

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

$id \rightarrow letter\ (\ letter \mid digit\ )^*$

numbers

$integer \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9)\ \ digit^*\ )$

$decimal \rightarrow integer\ .\ (\ digit\ )^*$

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

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

We can use REs to build scanners automatically.

Algebraic properties of REs

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

Examples

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

> a | b denotes {a,b}

> (a | b) (a | b) denotes {aa,ab,ba,bb}

> a* denotes {ε,a,aa,aaa,…}

> (a | b)* denotes the set of all strings of a's and b's (including ε), i.e., (a | b)* = (a* | b*)*

> a | a*b denotes {a,b,ab,aab,aaab,aaaab,…}

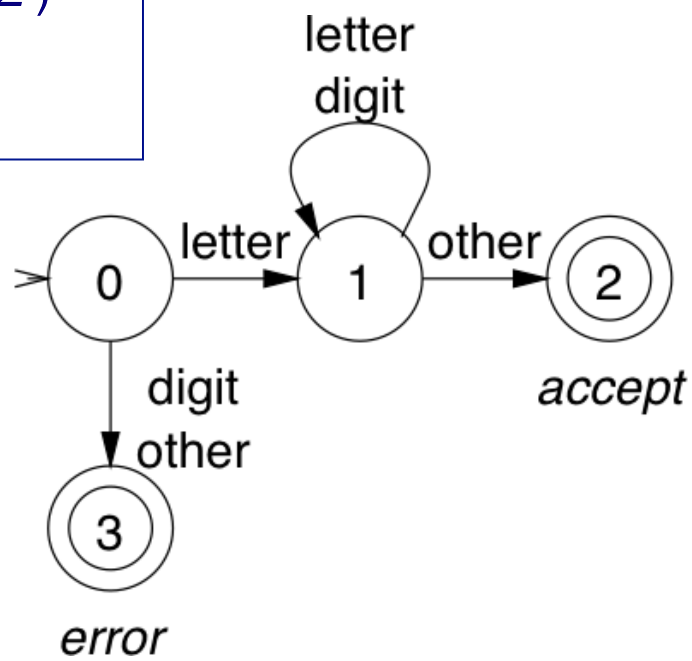
Roadmap

- Regular languages
- **Finite automata recognizers**
- From regular expressions to deterministic finite automata, and back
- Limits of regular languages

Recognizers

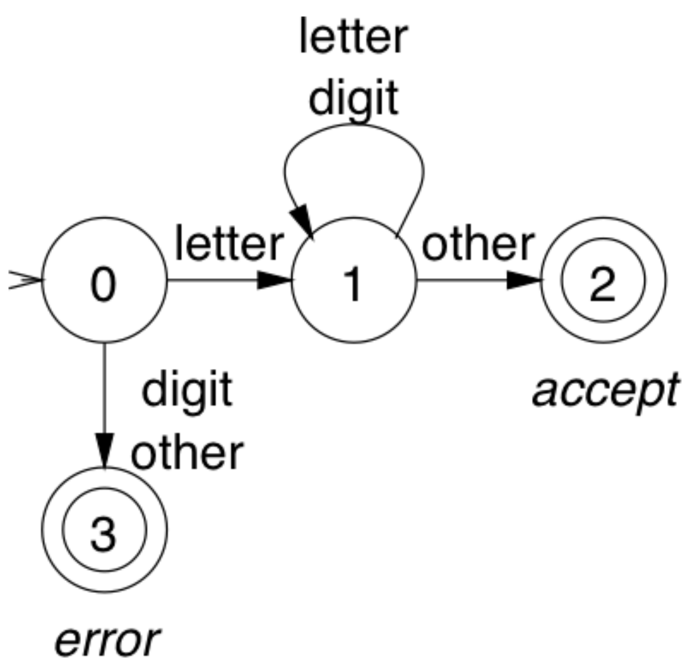
From a regular expression we can construct a *deterministic finite automaton* (DFA)

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



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;
```



Tables for the recognizer

Two tables control the recognizer

char_class

char	a-z	A-Z	0-9	other
value	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)

Grammars for regular languages

Regular grammars generate regular languages

Provable fact:

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

Definition:

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

1. A → aA
2. A → a

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

These are also called type 3 grammars (Chomsky)

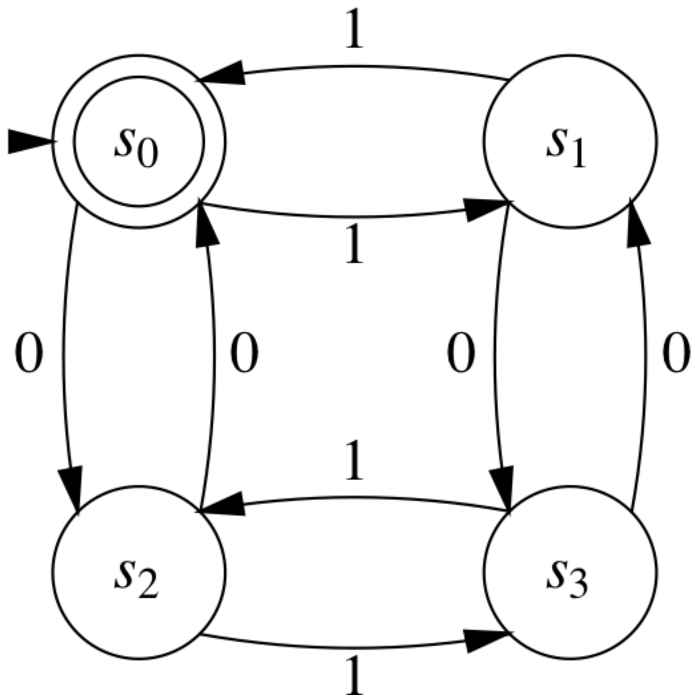
Aside: The Chomsky Hierarchy

- **Type 0: $\alpha \rightarrow \beta$**
  - Unrestricted grammars generate *recursively enumerable languages*, recognizable by Turing machines
- **Type 1: $\alpha A \beta \rightarrow \alpha \gamma \beta$**
  - 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: $A \rightarrow b$ and $A \rightarrow aB$**
  - Regular grammars generate *regular languages*, recognizable by finite state automata

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

More regular languages

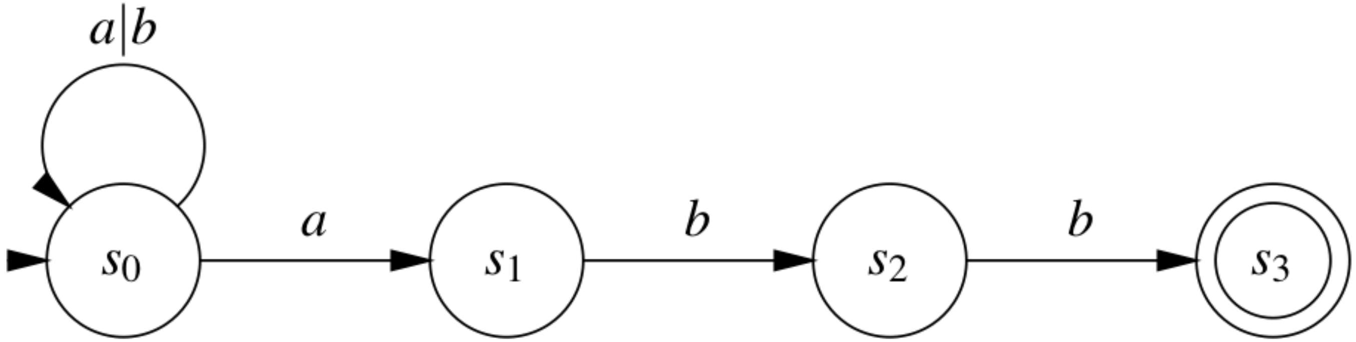
Example: the set of strings containing an even number of zeros and an even number of ones



The RE is (00 | 11)*((01 | 10)(00 | 11)*(01 | 10)(00 | 11)*)*

More regular expressions

*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* 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.

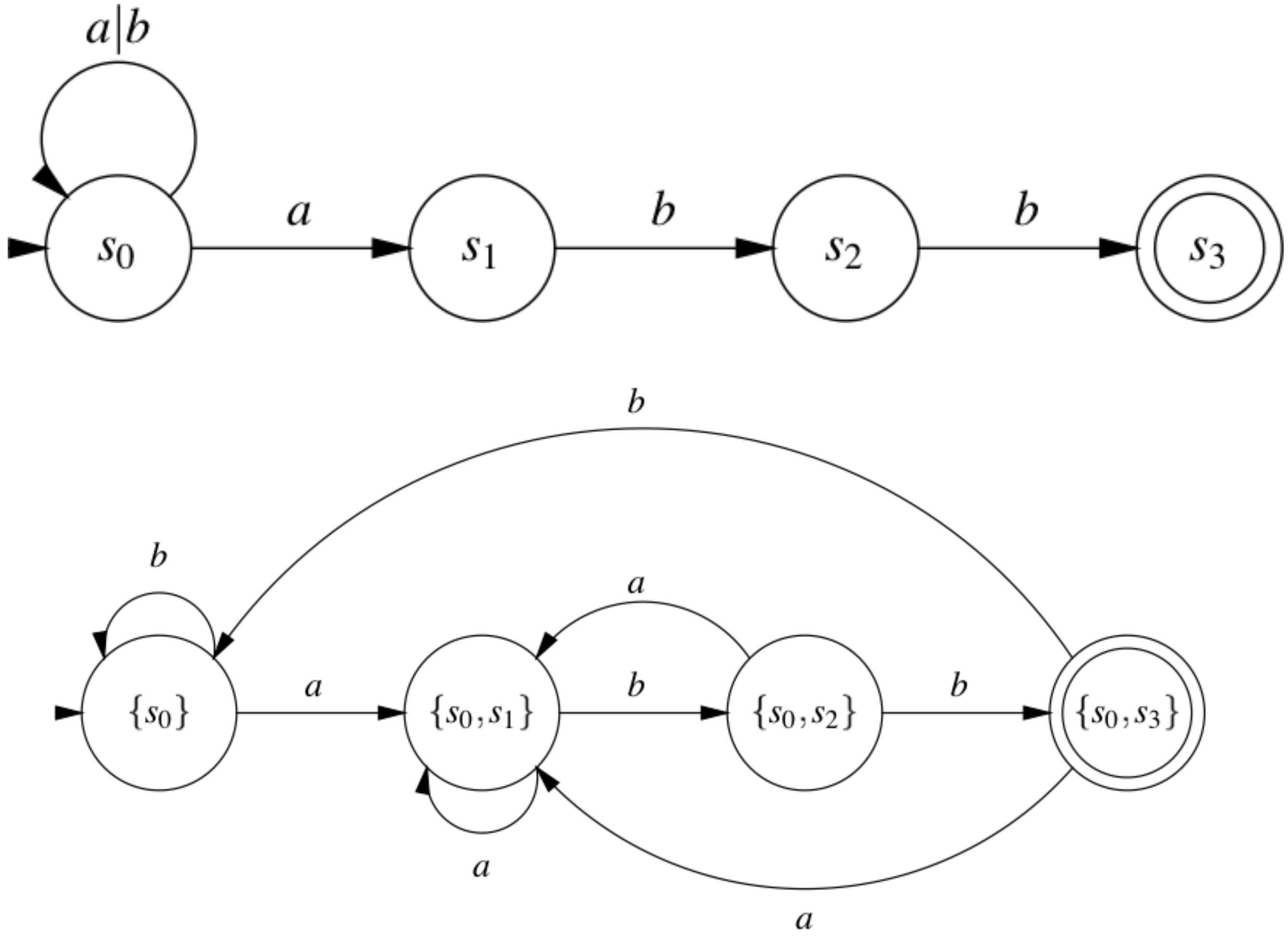
A DFA *accepts x* iff there exists a *unique* path through the transition graph from the s_0 to an accepting state such that the labels along the edges spell x.

DFAs and NFAs are equivalent

1. DFAs are clearly 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

NFA to DFA using the subset construction

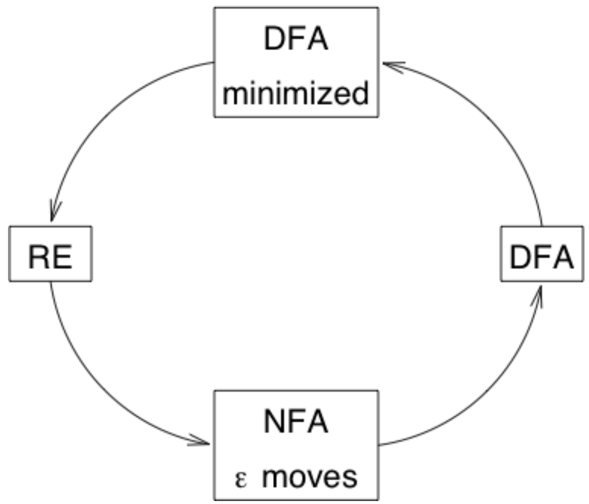


Roadmap

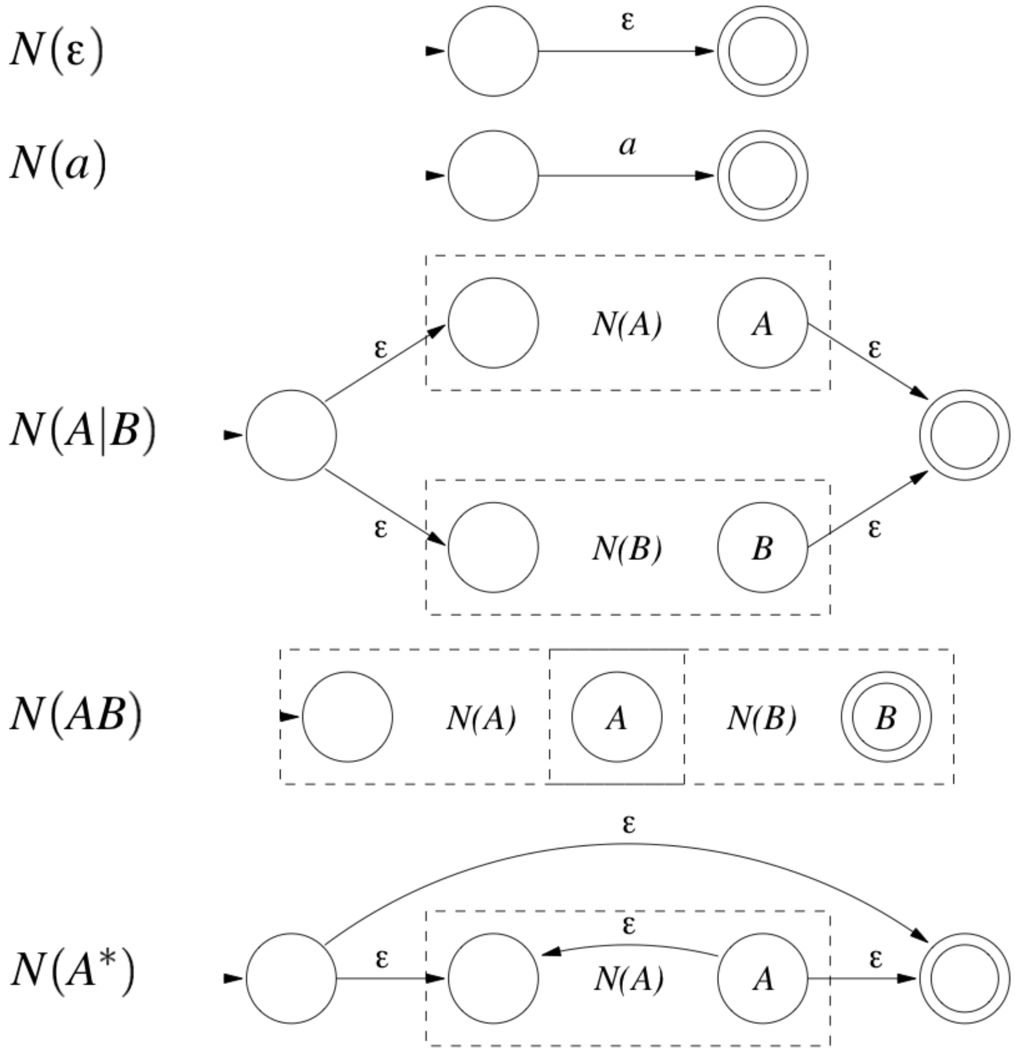
- > Regular languages
- > Finite automata recognizers
- > **From regular expressions to deterministic finite automata, and back**
- > Limits of regular languages

Constructing a DFA from a regular expression

- 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 $R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$
  - Or convert via Generalized NFA (GNFA)

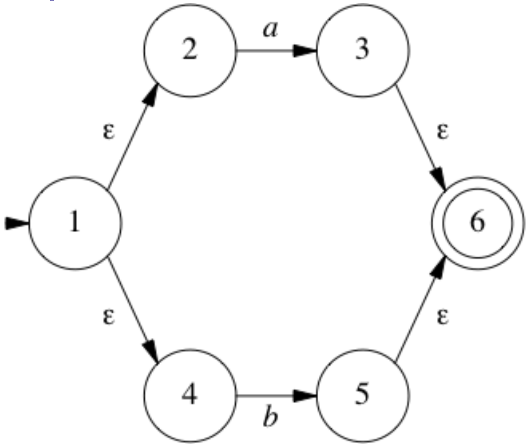


RE to NFA

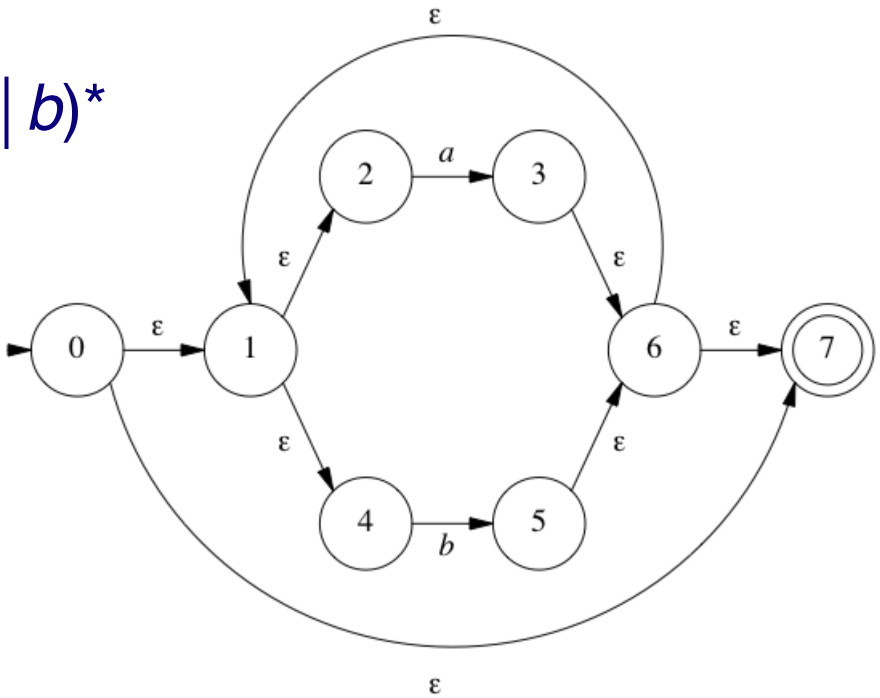


RE to NFA example: (*a*|*b*)**abb*

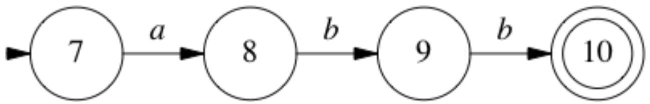
(*a*|*b*)



(*a*|*b*)*



abb



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 = ε-closure(s_0) unmarked to S_D

while ∃ unmarked state P in S_D

  mark P

  for each input symbol a

    U = ε-closure(move(P,a))

    if U $\notin$ S_D

    then add U unmarked to S_D

    T_D[T,a] = U

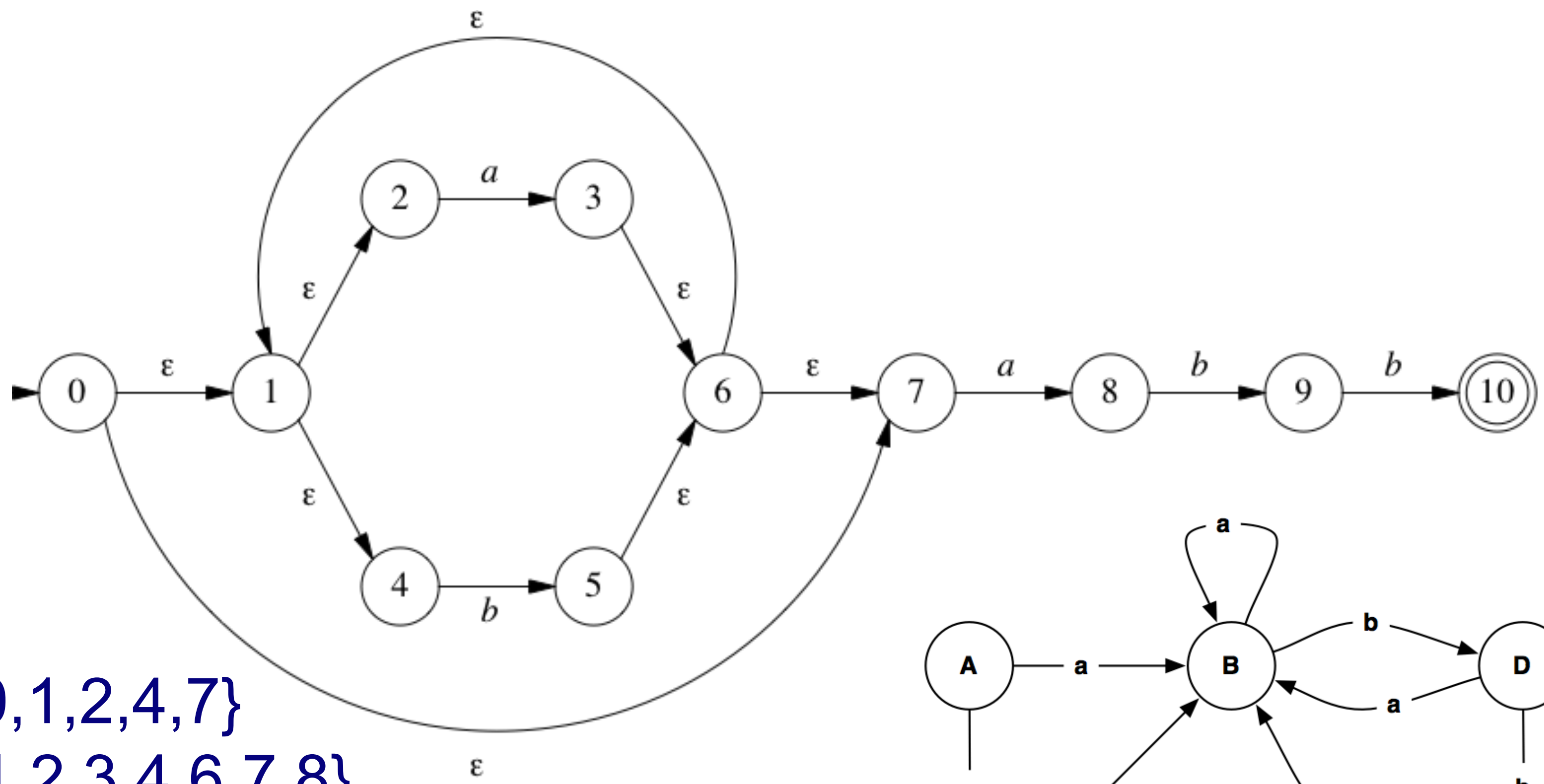
  end for

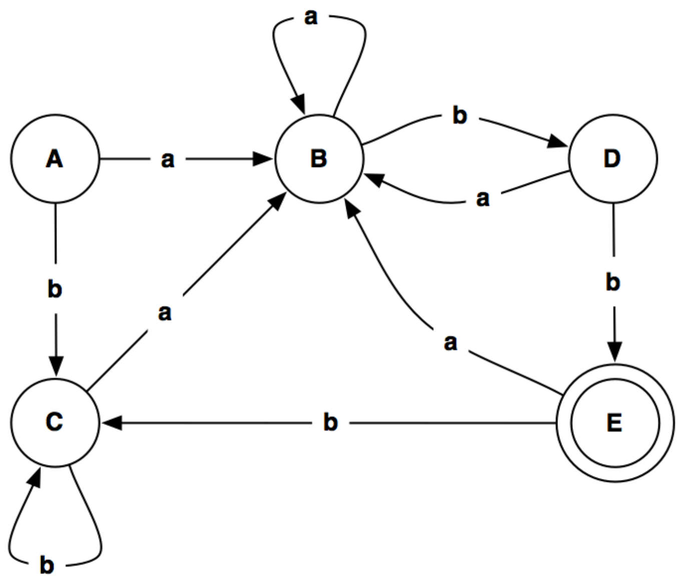
end while

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

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

NFA to DFA using subset construction: example

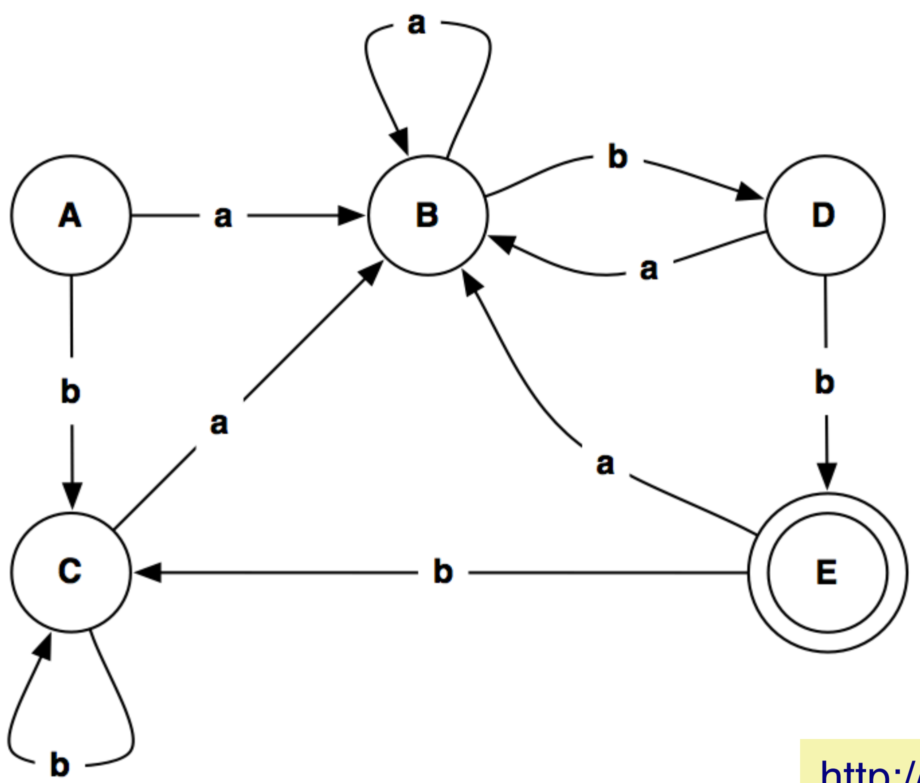




A = {0,1,2,4,7}
B = {1,2,3,4,6,7,8}
C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}
E = {1,2,4,5,6,7,10}

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!

http://en.wikipedia.org/wiki/DFA_minimization

DFA Minimization algorithm

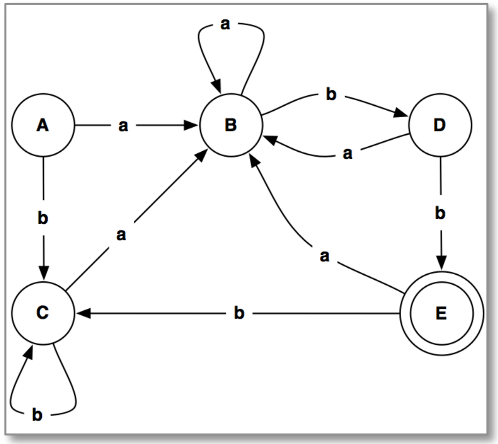
> Create lower-triangular table DISTINCT, initially blank

> For every pair of states (*p*,*q*):
  - If p is final and q is not, or vice versa
    - *DISTINCT(p,q) = ε*

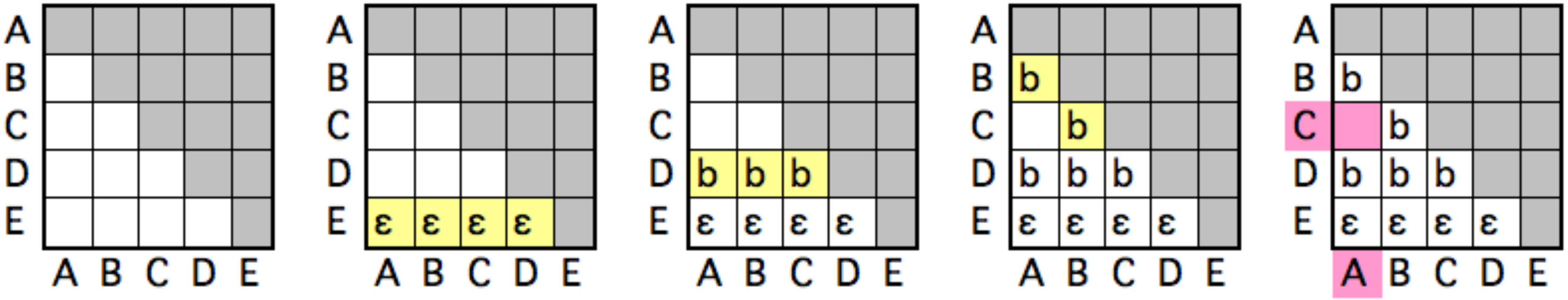
> Loop until no change for an iteration:
  - For every pair of states (*p*,*q*) and each symbol α
    - *If DISTINCT(p,q) is blank and DISTINCT(δ(p,α), δ(q,α)) is not blank*
      - DISTINCT(*p*,*q*) = α

> Combine all states that are not distinct

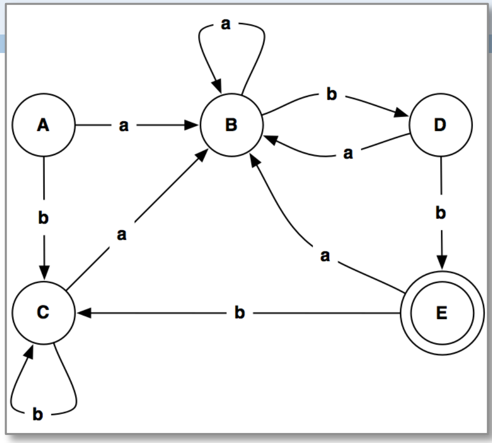
Minimization in action



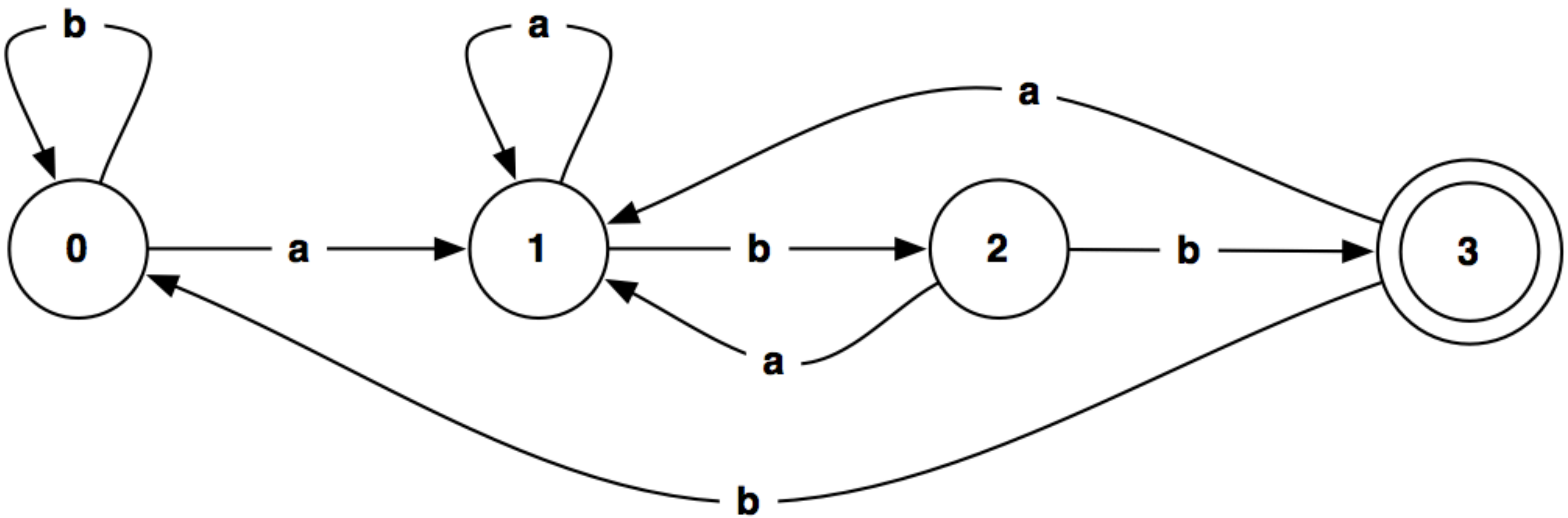
C and A are *indistinguishable* so can be merged



DFA Minimization example



It is easy to see that this is in fact the minimal DFA for (*a* | *b*)**abb* …



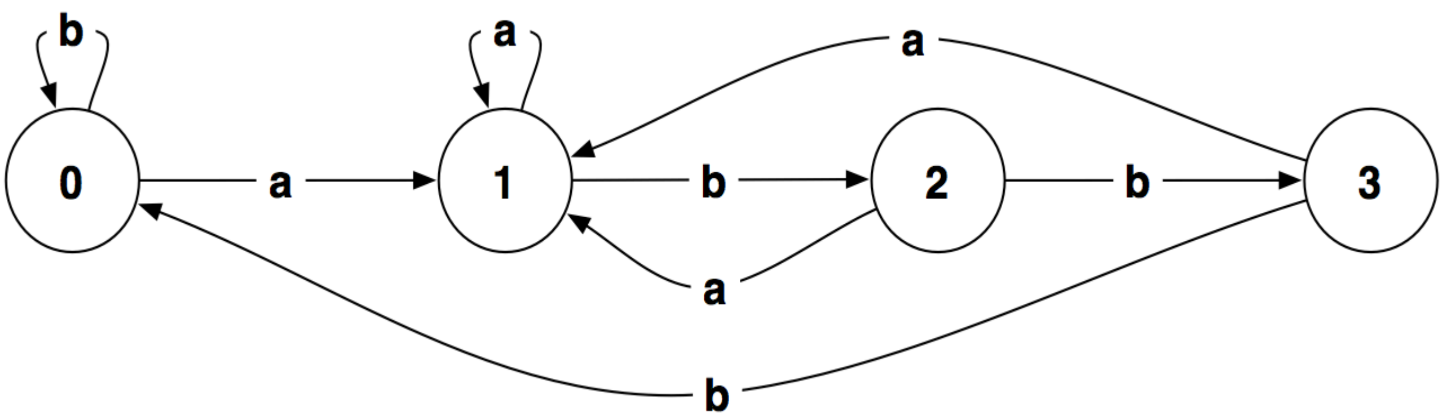
DFA to RE via GNFA

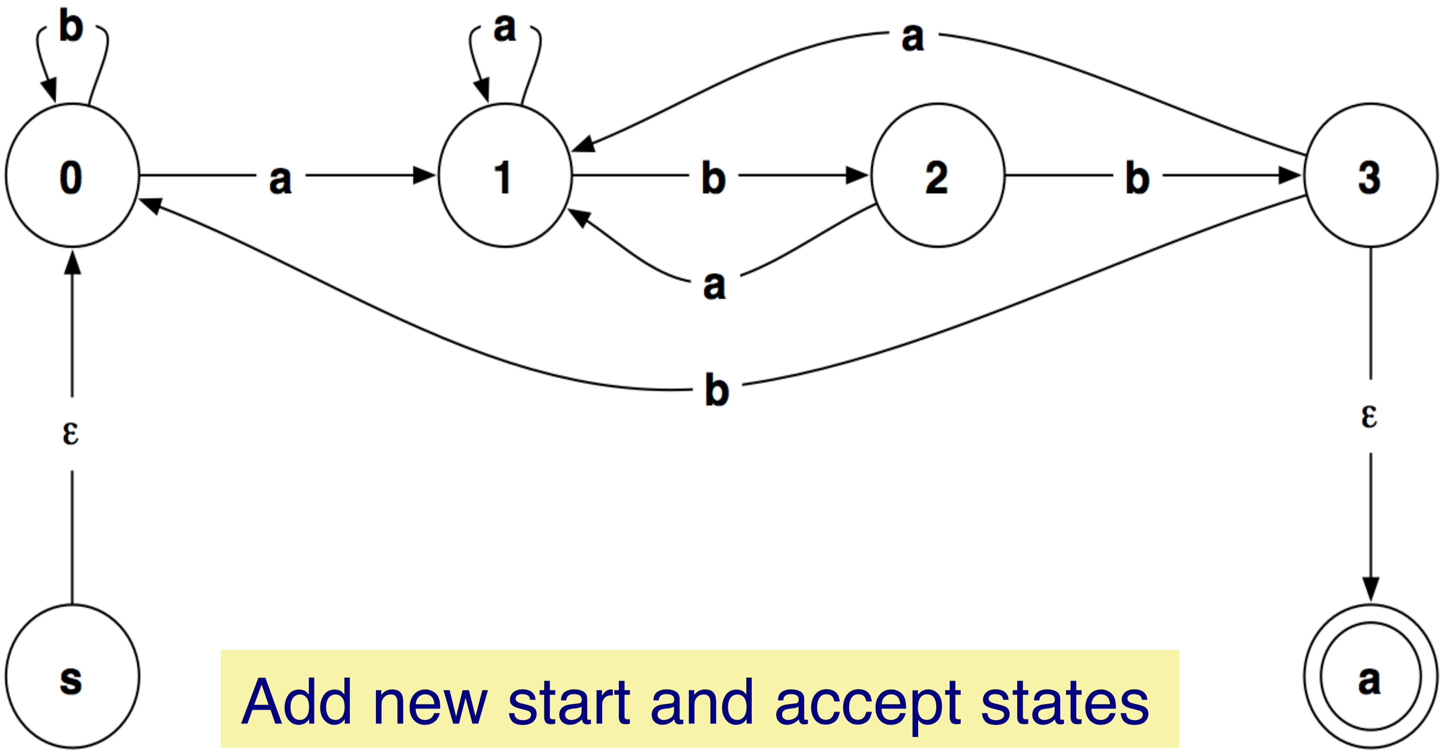
- A *Generalized NFA* is an NFA where transitions may have any RE as labels
- Conversion algorithm:
  1. *Add a new start state and accept state* with ε-transitions to/from the old start/end states
  2. *Merge multiple transitions* between two states to a single RE choice transition
  3. *Add empty ∅–transitions* 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

GNFA conversion algorithm

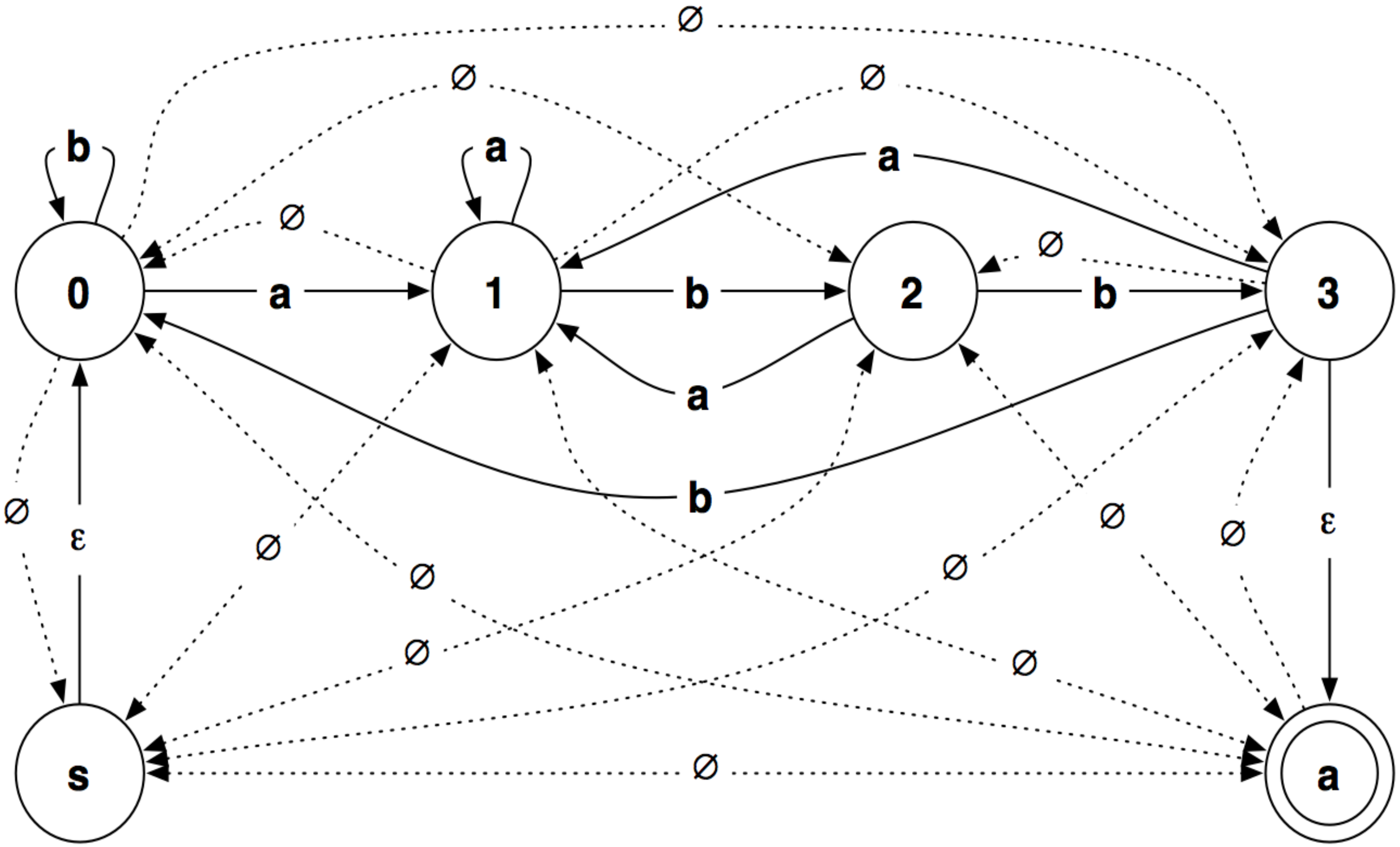
1. Let *k* be the number of states of G, $k \geq 2$
2. If $k=2$, then RE is the label found between q_s and q_a (start and accept states of G)
3. While $k>2$, select $q_{rip} \neq q_s$ or q_a
   — $Q' = Q - \{q_{rip}\}$
   — For any $q_i \in Q' - \{q_a\}$ **let $\delta'(q_i,q_j) = R_1\ R_2^*\ R_3 \cup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

The initial NFA

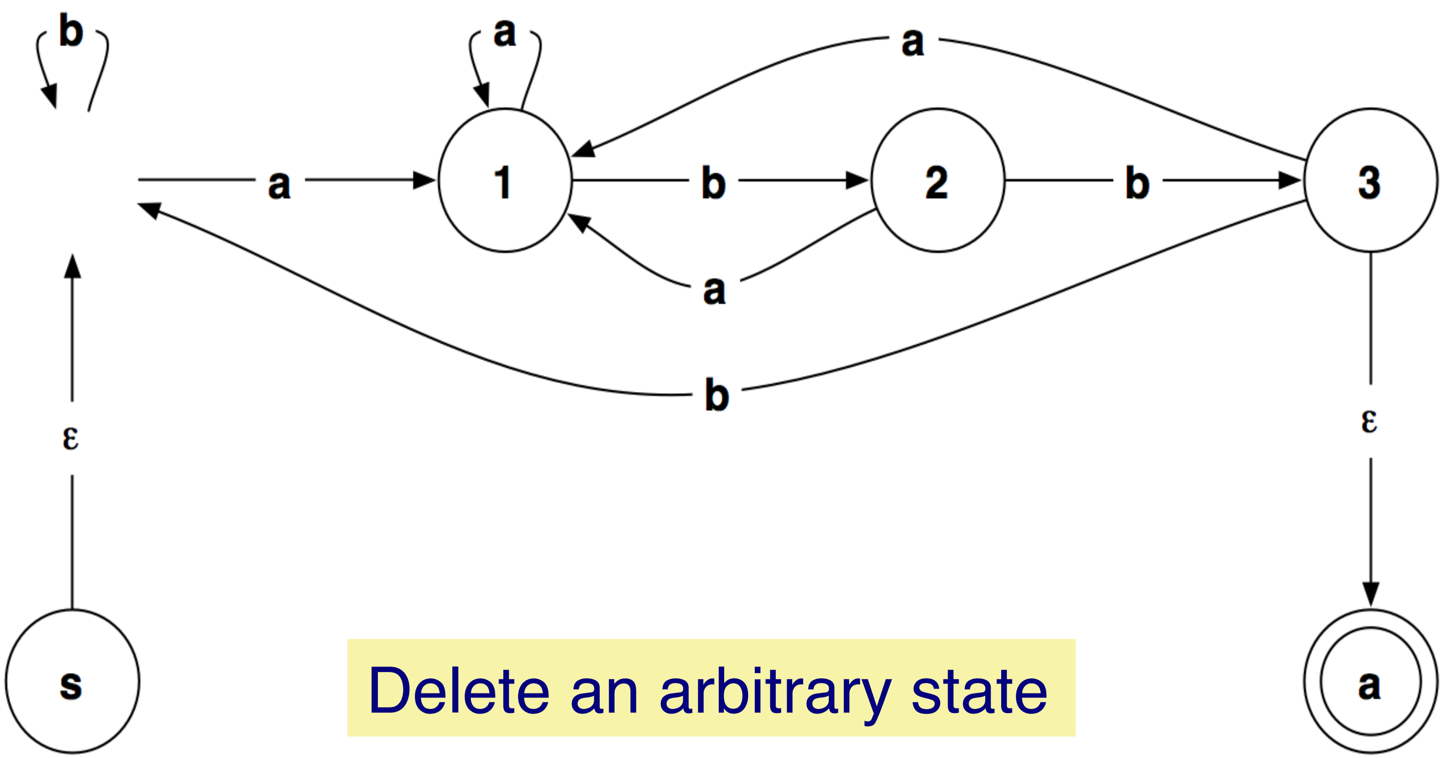




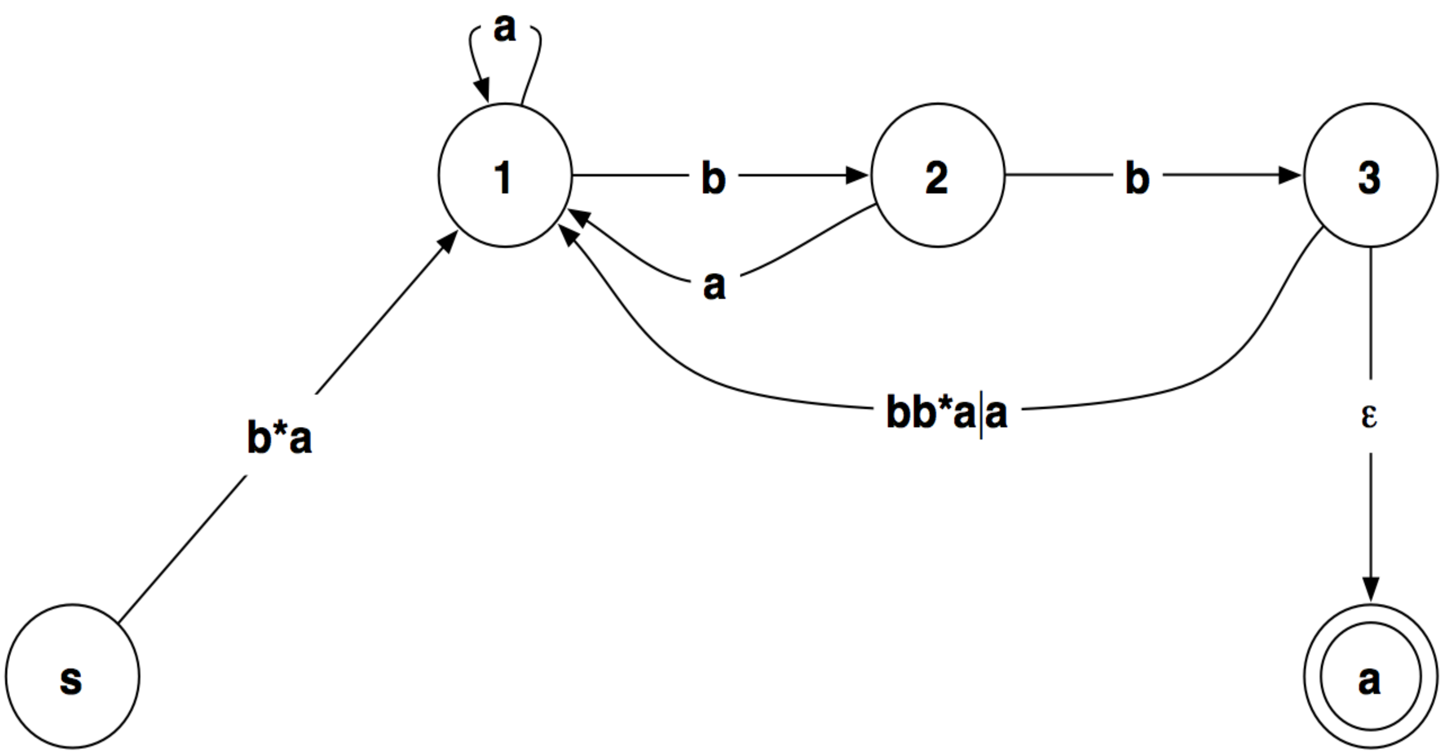
Add new start and accept states



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

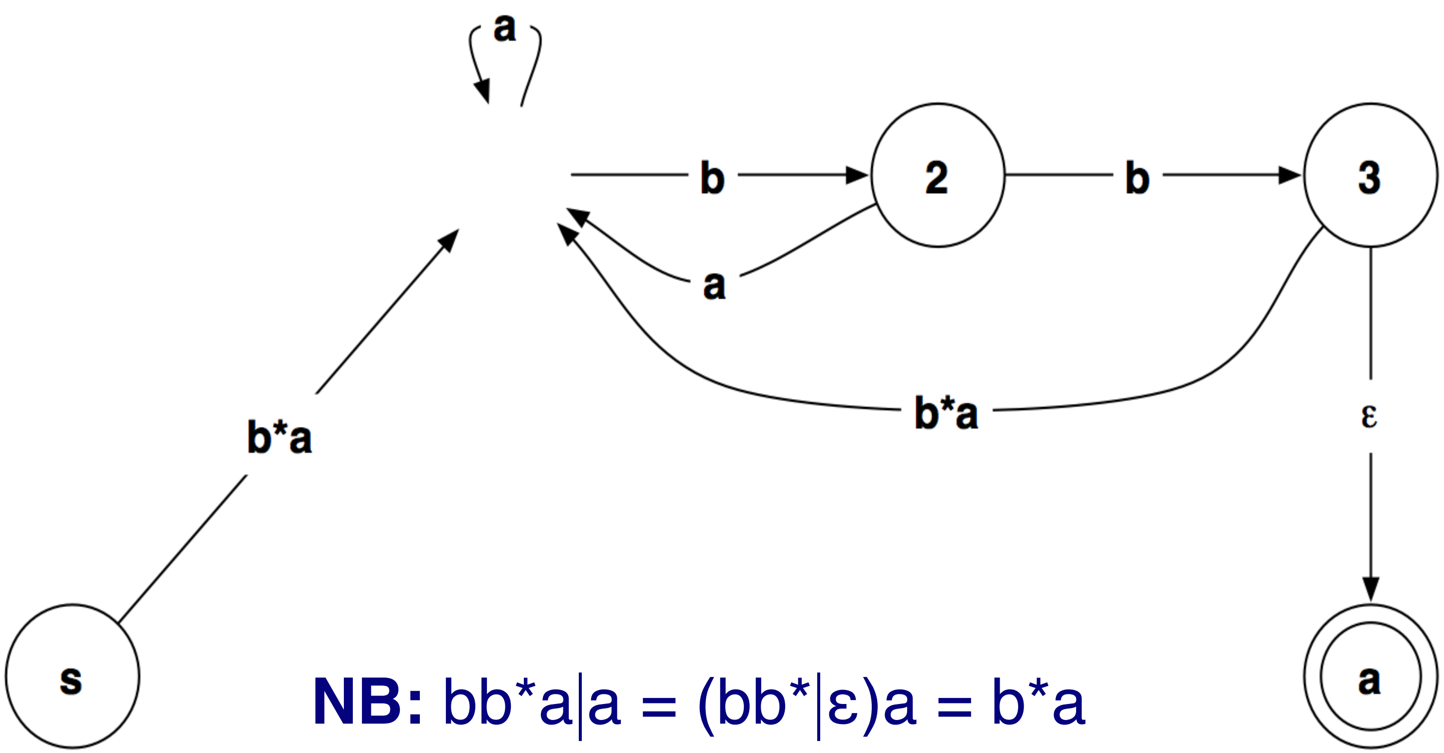


Delete an arbitrary state

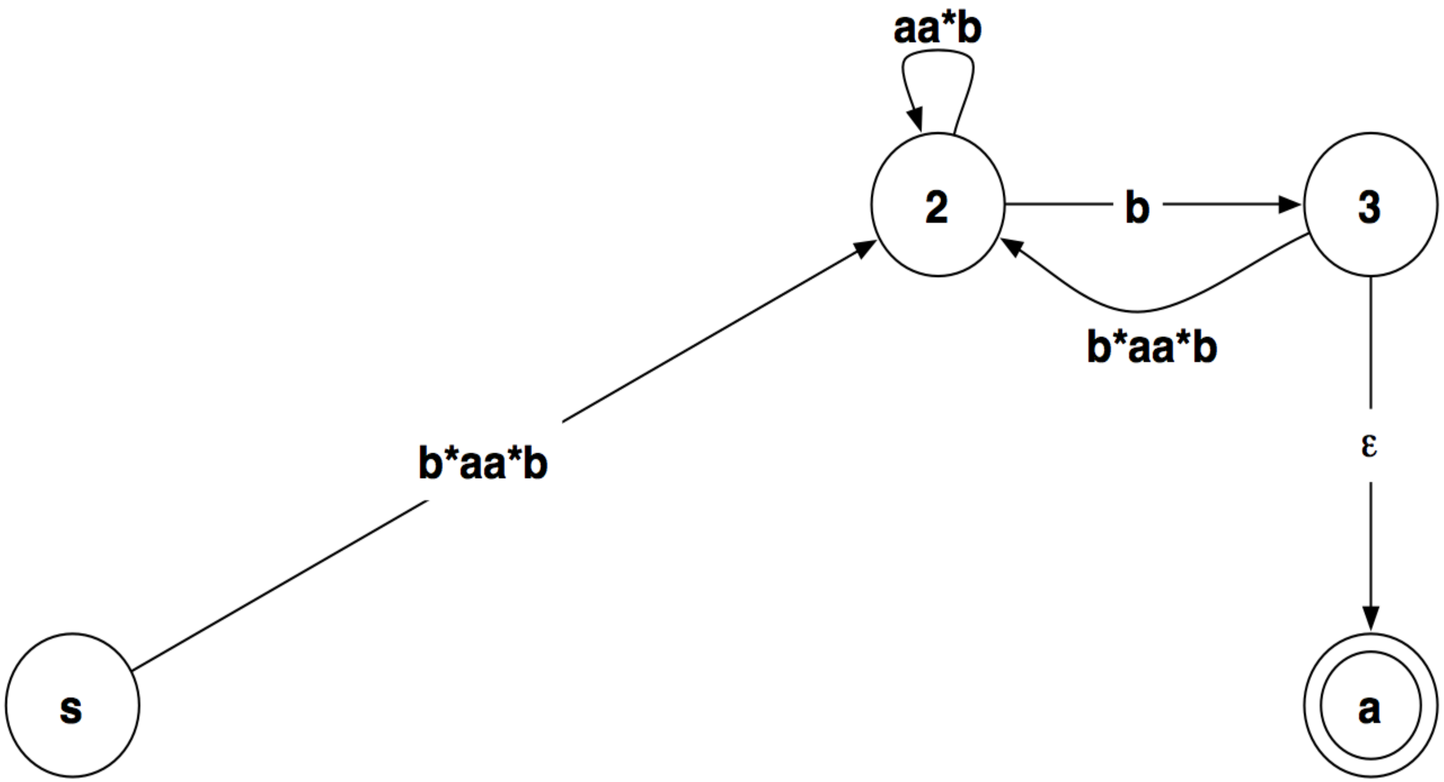


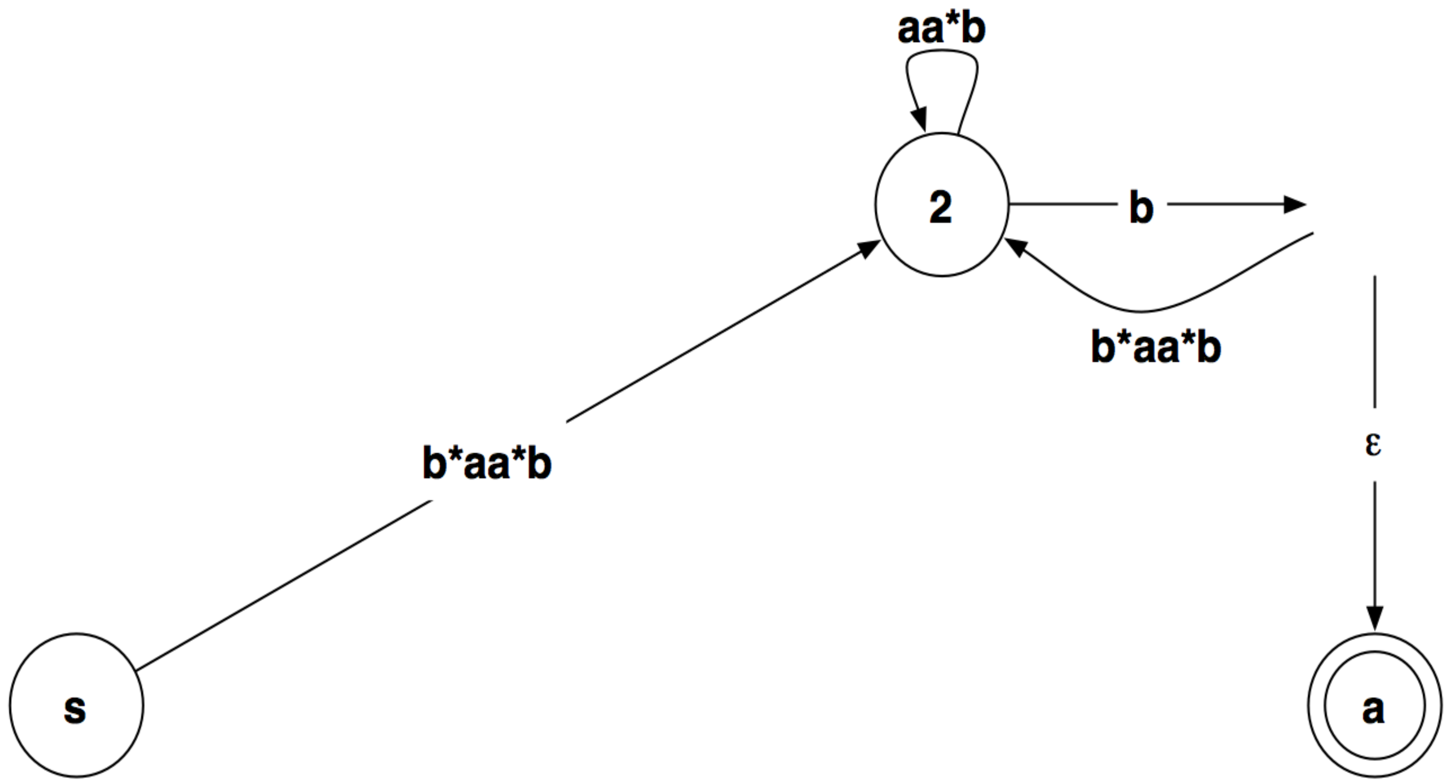
Fix dangling transitions s→1 and 3→1
Don’t forget to merge the existing transitions!

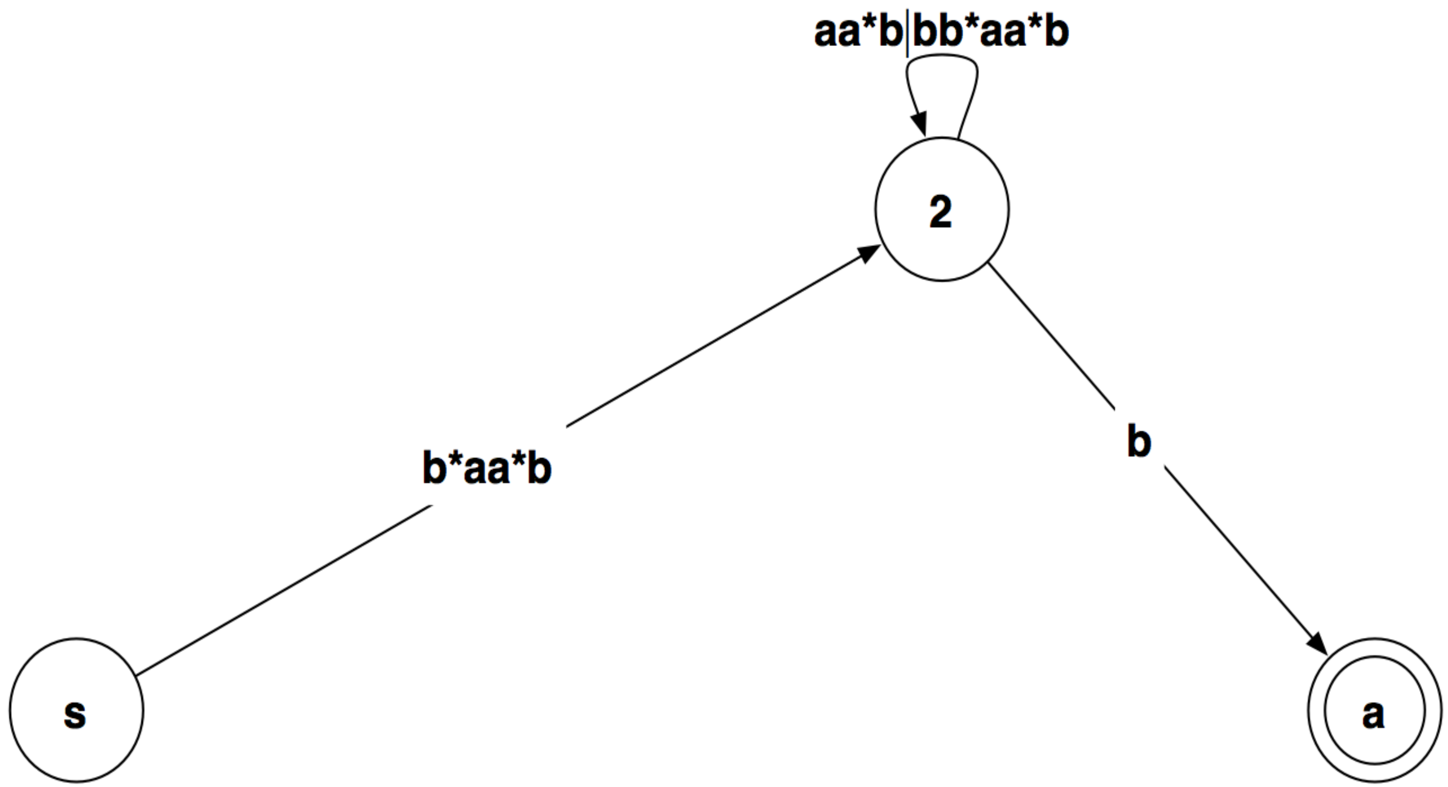
Simplify the RE
Delete another state

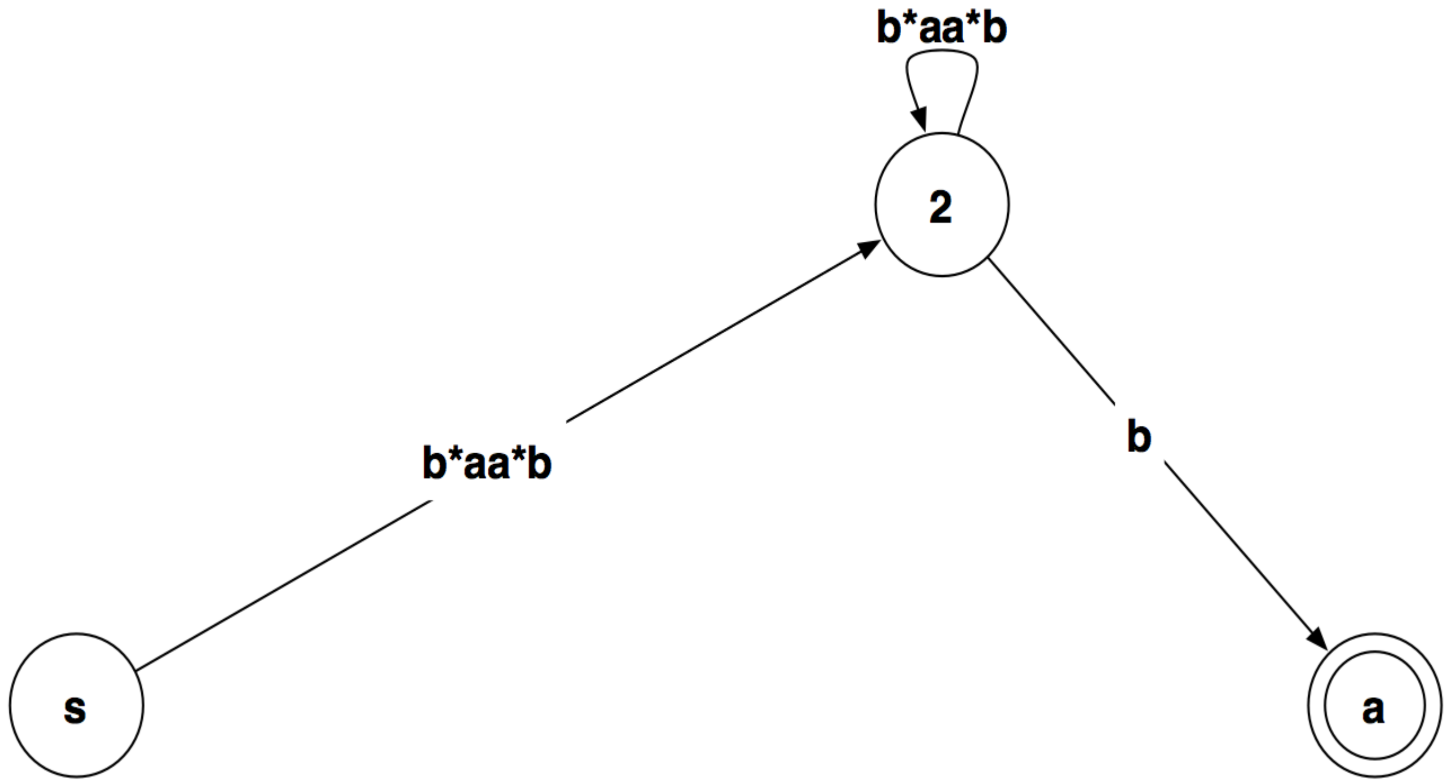


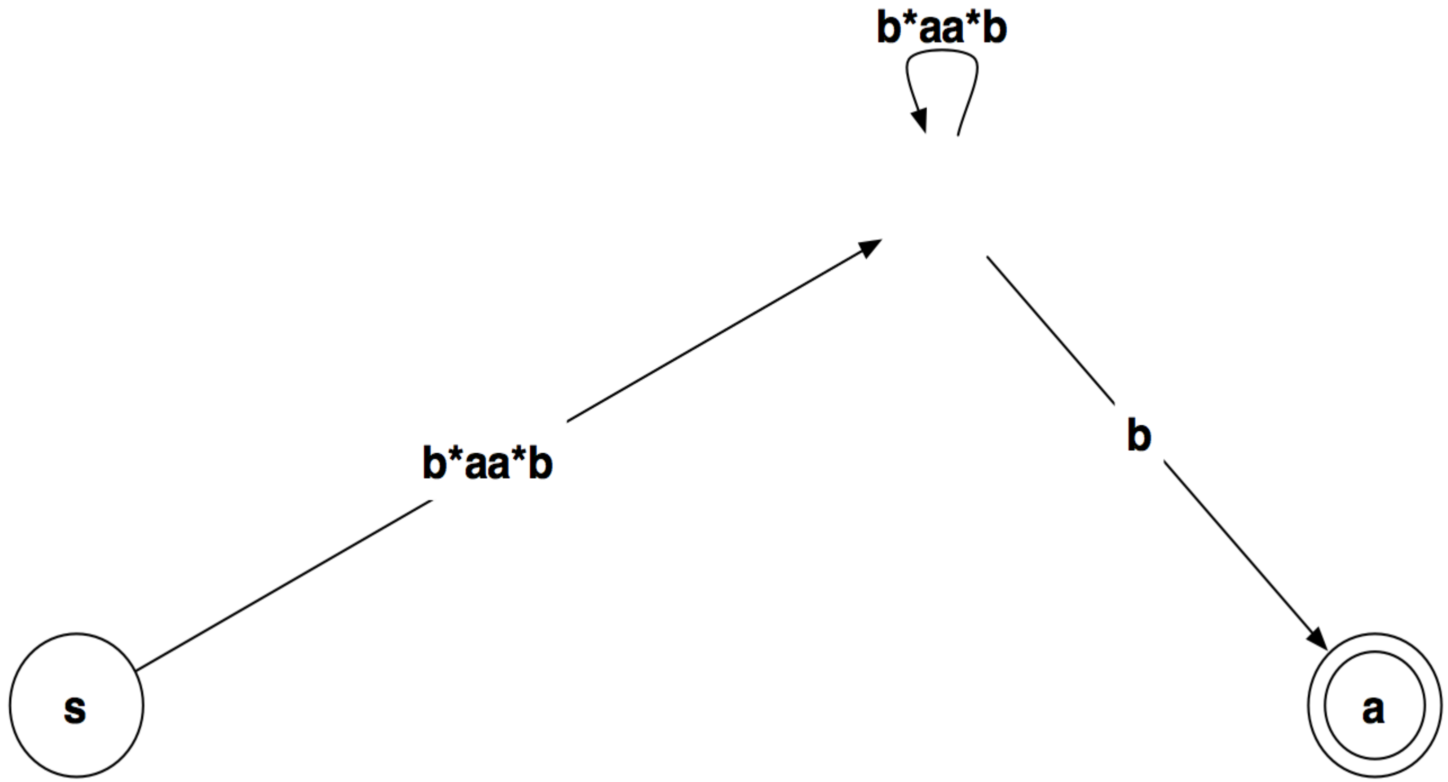
NB: bb*a|a = (bb*|ε)a = b*a


aa*b
2
b
3
b*aa*b
b*aa*b
ε
s
a

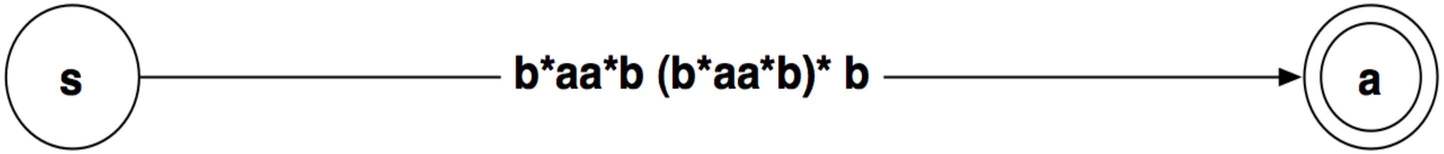

aa*b
2
b
b*aa*b
b*aa*b
ε
s
a


aa*b|bb*aa*b
2
b*aa*b
b
s
a


b*aa*b
2
b*aa*b
b
s
a


b*aa*b
b*aa*b
b
s
a

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 …

Roadmap

- Regular languages
- Finite automata recognizers
- From regular expressions to deterministic finite automata, and back
- **Limits of regular languages**

Limits of regular languages

Not all languages are regular!

One cannot construct DFAs to recognize these languages:

$L = \{ p^k q^k \}$
$L = \{ wcw^r \mid w \in \Sigma^*, w^r \text{ is w } \textit{reversed} \}$

In general, DFAs cannot count!

However, one *can* construct DFAs for:

- Alternating 0's and 1's:
  $(\varepsilon \mid 1)(01)^*(\varepsilon \mid 0)$
- 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?

```
            INTEGERFUNCTIONA
            PARAMETER(A=6,B=2)
            IMPLICIT CHARACTER*(A-B)(A-B)
            INTEGER FORMAT(10),IF(10),DO9E1
      100   FORMAT(4H)=(3)
      200   FORMAT(4 )=(3)
            DO9E1=1
            DO9E1=1,2
               IF(X)=1
               IF(X)H=1
               IF(X)300,200
      300      CONTINUE
            END
      C     this is a comment
          $ FILE(1)
            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?*

License

`http://creativecommons.org/licenses/by-sa/3.0/`