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

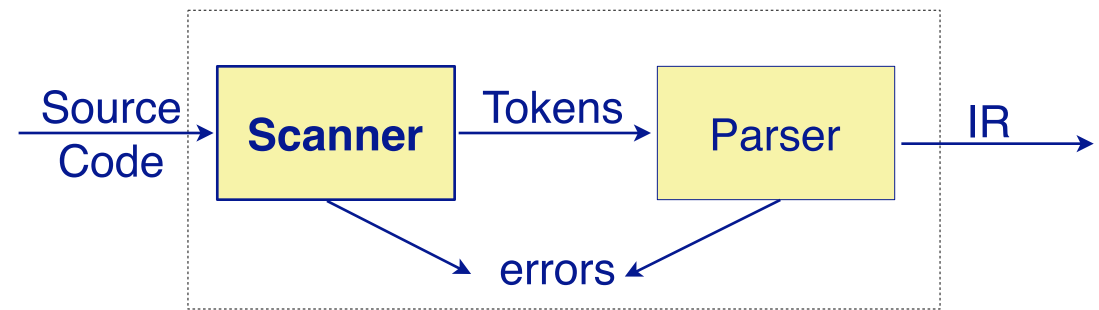
> Introduction
> Regular languages
> Finite automata recognizers
> From RE to DFAs and back again
> Limits of regular languages

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` → `<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

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!

A key issue is ...

why don't we write it by hand?

Roadmap

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

Languages and Operations

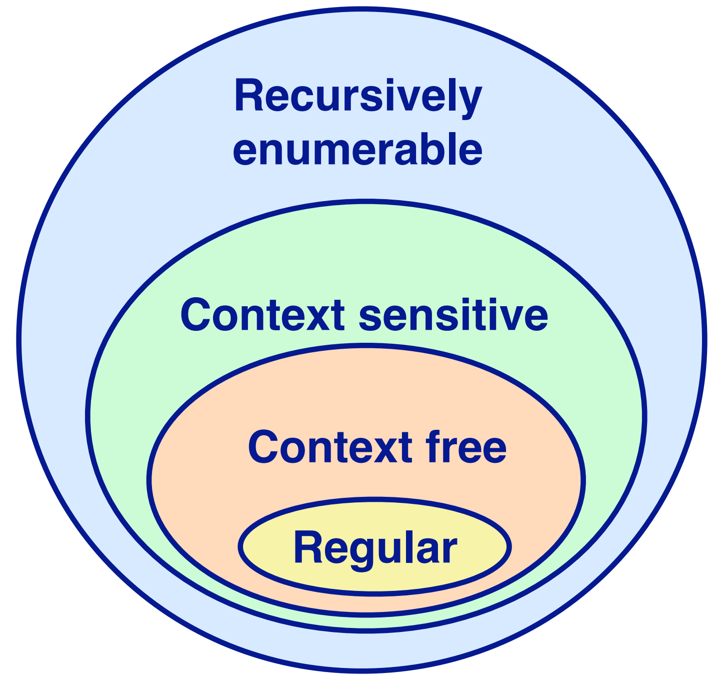
A *language* is a set of strings

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

How do you define a language?
Recognizer.
Production grammar.

Production Grammars

- Powerful formalism for language description
  - Non-terminals (A, B)
  - Terminals (a,b)
  - Production rules (A->abA)
  - Start symbol (S0)
- Rewriting



Detail: The Chomsky Hierarchy

> **Type 0: $\alpha \rightarrow \beta$**

—Unrestricted grammars generate *recursively enumerable languages*. Minimal requirement for recognizer: Turing machine.

> **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: $A \rightarrow a$ 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

Individual identifiers in a classical programming language form a regular language.

The language is on the other hand **context free** most of the time.

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 $\{\varepsilon\}$
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) | (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 (the set with the “empty” string)

As you can see, we don’t define a+ or [a]

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 Σ = {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)
- Universit(ä|ae)t Bern(e|) ...

Σ = Alphabet

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 of using REs to specify lexical patterns

identifiers

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

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

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

numbers

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

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

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

$complex \rightarrow$ '`(`' $real$ '`,`' $real$ '`)`'

Numbers can get much more complicated.

Most programming language tokens can be described with REs.

Roadmap

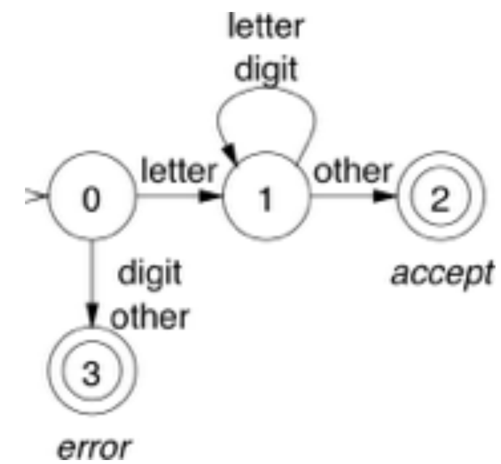
- > Introduction
- > Regular languages
- > **Finite automata recognizers**
- > From RE to DFAs and back again
- > Limits of regular languages

REs are cool for specifying.
FAs are good for implementing REs.

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*)*

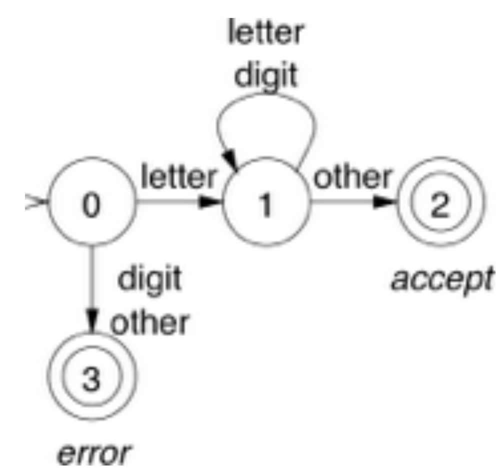
From a regular expression we can construct a *deterministic finite automaton* (DFA)



DFA for recognizing an identifier.
why D? why F? why A?

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;
```



I.e., encode the transitions in the next_state matrix

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

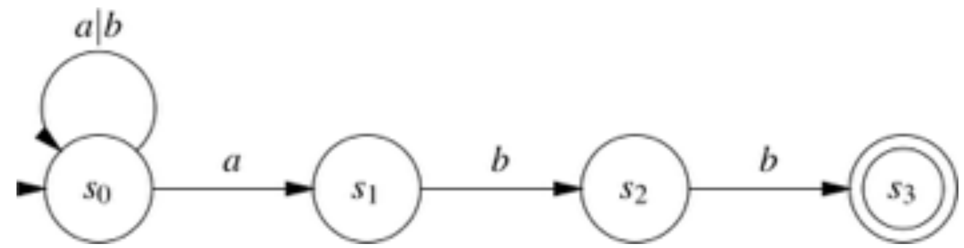
- *Scanner generators* automatically construct code from regular expression-like descriptions
  - —construct a DFA
  - —use *state minimization* techniques
  - —emit code for the scanner (table driven or direct code)

- 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)
  - —nowadays JavaCC is 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

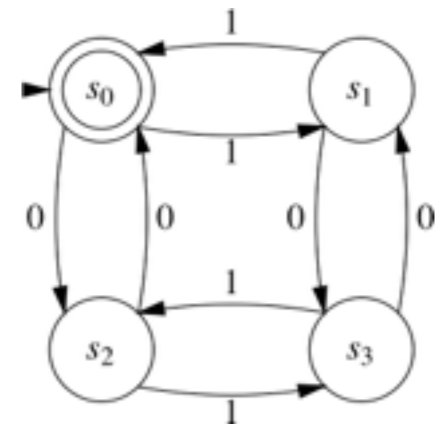
A ***Deterministic Finite Automaton*** (**DFA**) is a special case of an NFA:

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 | 11)*((01 | 10)(00 | 11)*(01 | 10)(00 | 11)*)*

Note how the RE walks through the DFA.

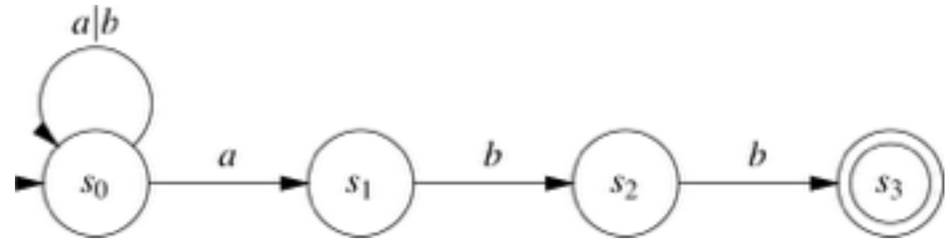
NB: The states capture whether there are an even number of 0s or 1s => 4 possible states.

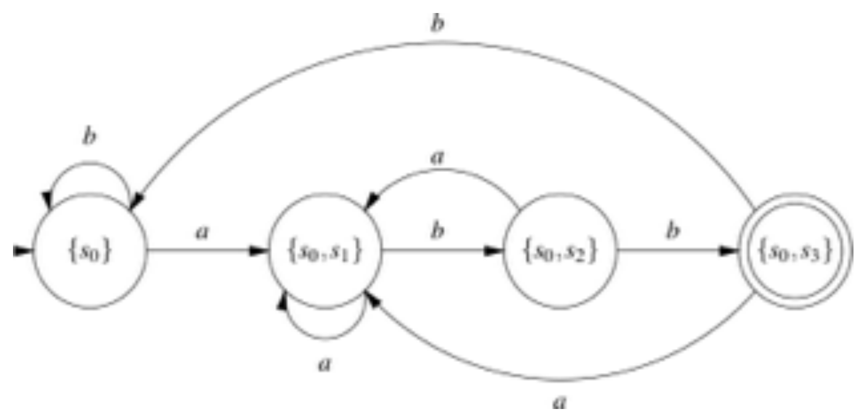
DFAs and NFAs are equivalent

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

NFA to DFA using the subset construction





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

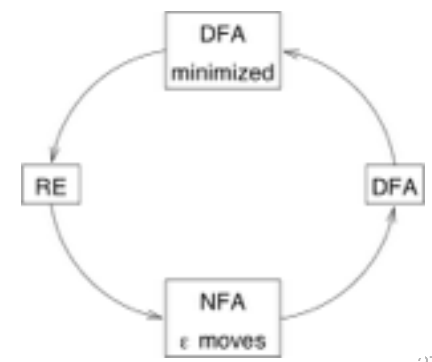
> DFA → minimized DFA

—Merge equivalent states

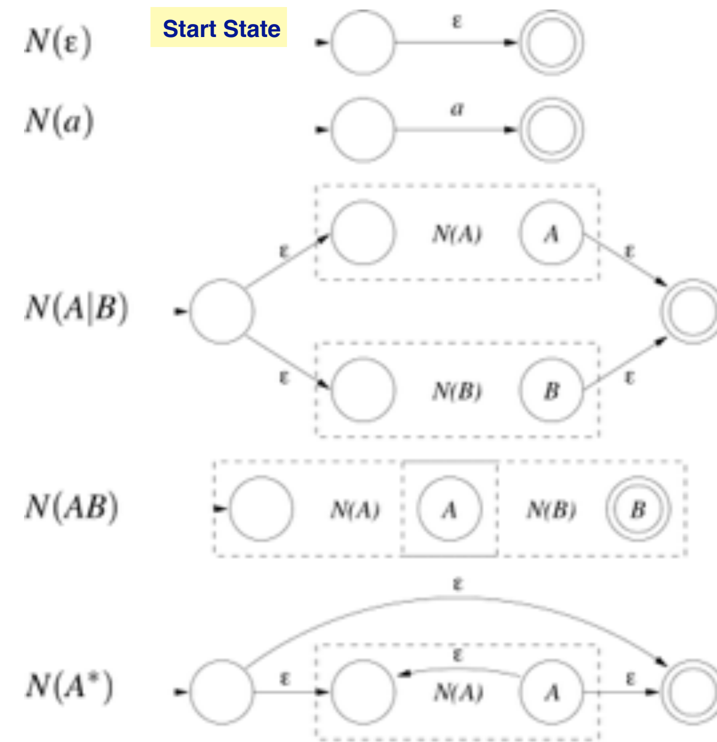
> DFA → RE

—Construct $R^{k}_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$

—Or convert via Generalized NFA (GNFA)



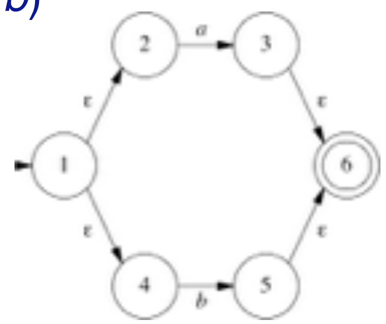
RE to NFA



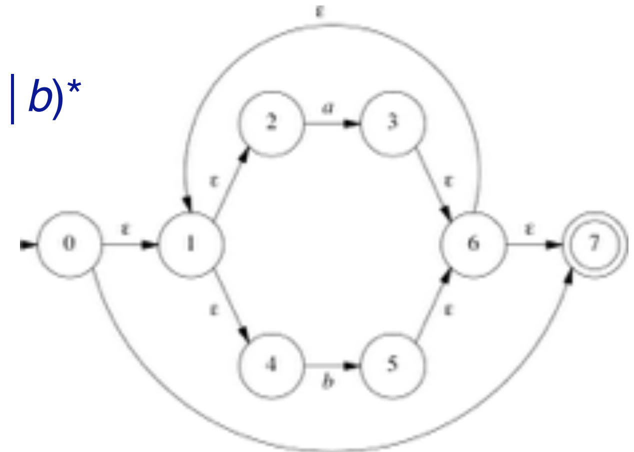
$N(\varepsilon)$
Start State
ε
$N(a)$
a
$N(A|B)$
ε
N(A)
A
ε
ε
N(B)
B
ε
$N(AB)$
N(A)
A
N(B)
B
$N(A^*)$
ε
ε
ε
N(A)
A
ε

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

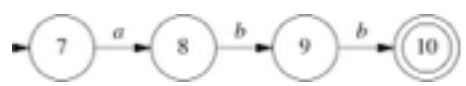
(*a* | *b*)

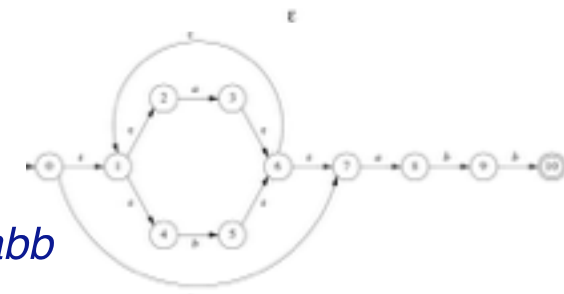


(*a* | *b*)*



abb





(*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

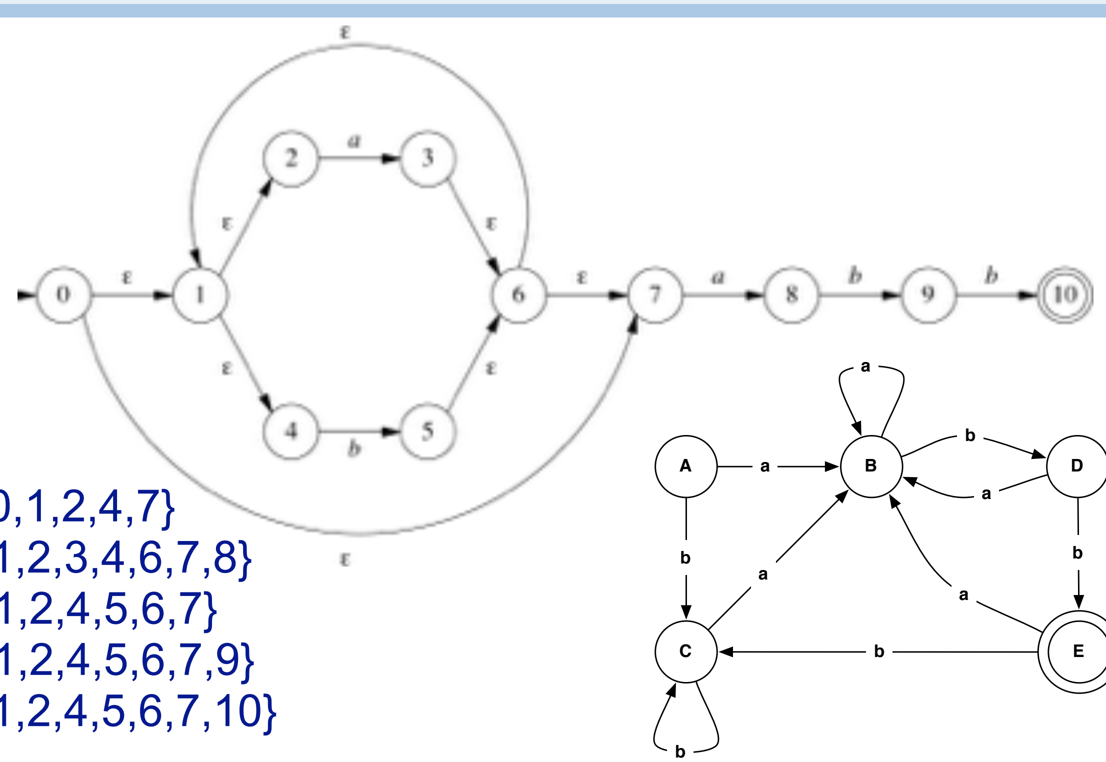
while ∃ unmarked state P in S_D
  mark P
  for each input symbol a
    U = ε-closure(move(P,a))
    if U $\notin$ S_D
    then add U unmarked to S_D
    T_D[P,a] = U
  end for
end while

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

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

Renamed some terms from Palsberg/Hosking slide

NFA to DFA using subset construction: example

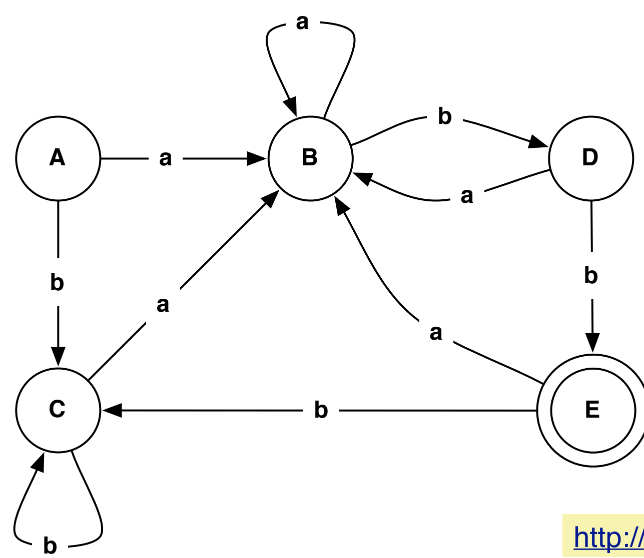


A = {0,1,2,4,7}
B = {1,2,3,4,6,7,8}
C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}
E = {1,2,4,5,6,7,10}

Are NFAs more powerful than DFAs?
Fewer states and easier to construct!
But the transformation is not minimal.

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

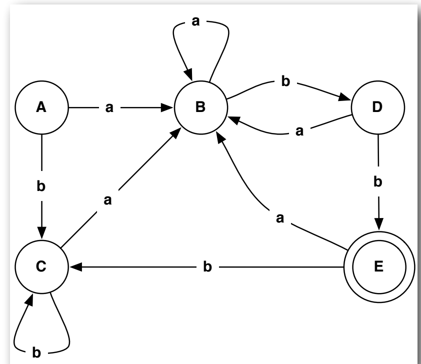
After b*a we always end up in state B.

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

Distinguish final state from all others. Then take single steps to check what is distinguishable. The intuition:
- if one state is final and the other not, then they are clearly distinct
- otherwise, for every (state, state, symbol) tuple we see whether the ∂ is in DISTINCT

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	

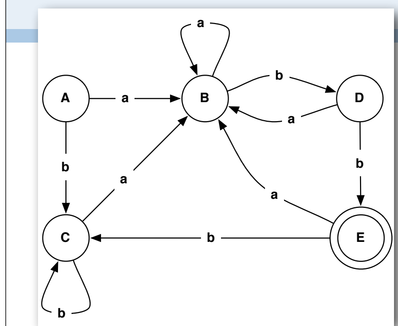
	A	B	C	D	E
A					
B					
C					
D	b	b	b		
E	ε	ε	ε	ε	

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

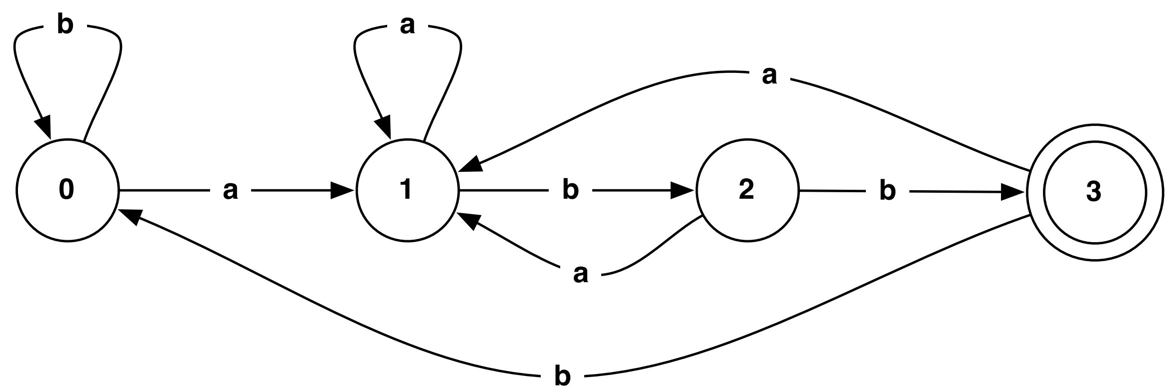
	A	B	C	D	E
A					
B	b				
C		b			
D	b	b	b		
E	ε	ε	ε	ε	

0. initial state. 1. E is final, so different from others.
2. Only a “b” step from D leads to non-blank space.
3. B can make a “b” step to D, so differs from A and C.
4. A and C are indistinguishable. (An “a” takes both to B and “b” takes both to C.)

DFA Minimization example



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



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

Start with the 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 follow with abb.

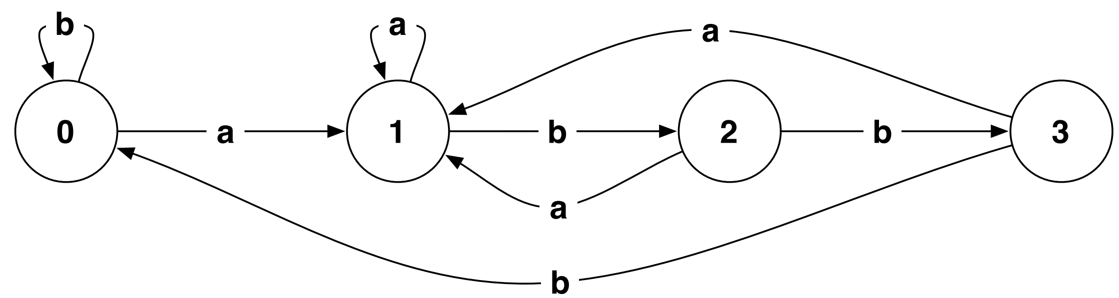
DFA to RE via GNFA

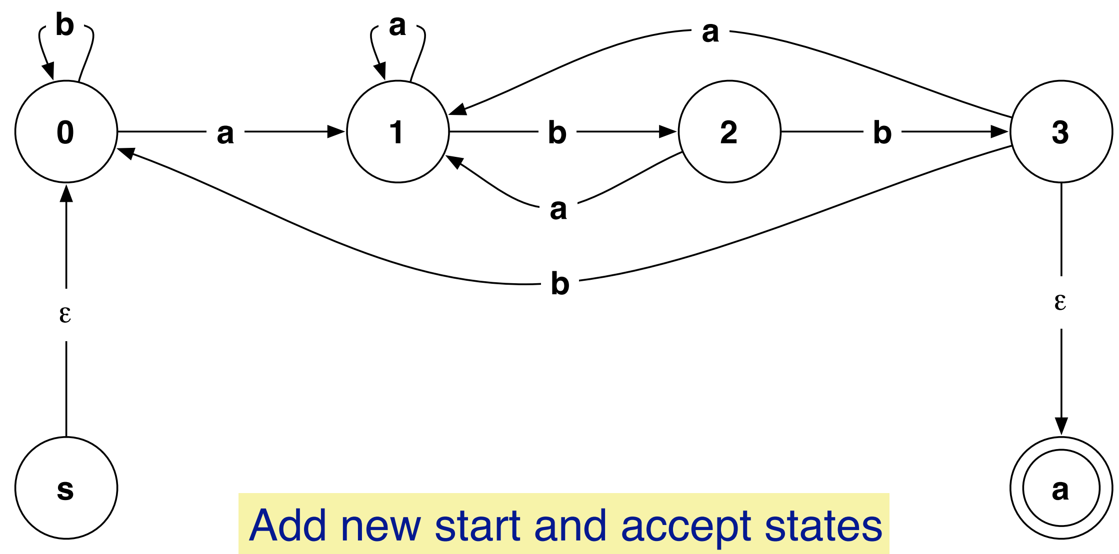
- A *Generalized NFA* is an NFA where transitions may have any RE as labels
- Conversion algorithm:
  1. *Add a new start state and accept state* with ε-transitions to/from the old start/end states
  2. *Merge multiple transitions* between two states to a single RE choice transition
  3. *Add empty ∅–transitions*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

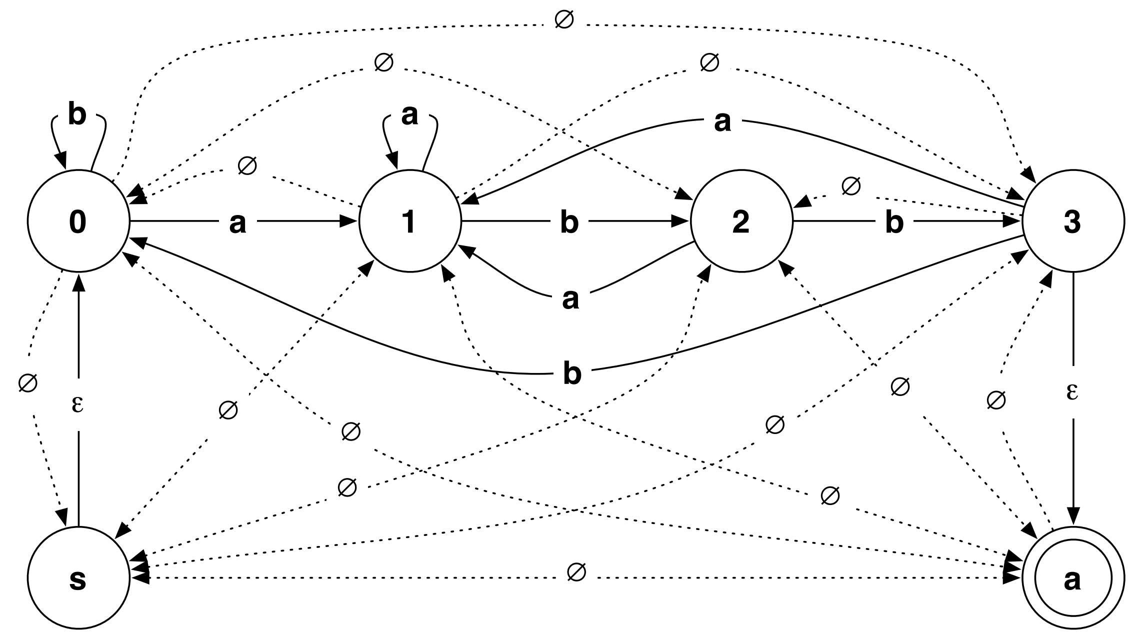
1. Let *k* be the number of states of G, $k \geq 2$
2. If $k=2$, then RE is the label found between q_s and q_a (start and accept states of G)
3. While $k>2$, select $q_{rip} \neq q_s$ or q_a
   - $Q' = Q - \{q_{rip}\}$
   - For any $q_i \in Q' - \{q_a\}$ **let** $\boldsymbol{\delta'(q_i,q_j) = R_1 R_2^*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



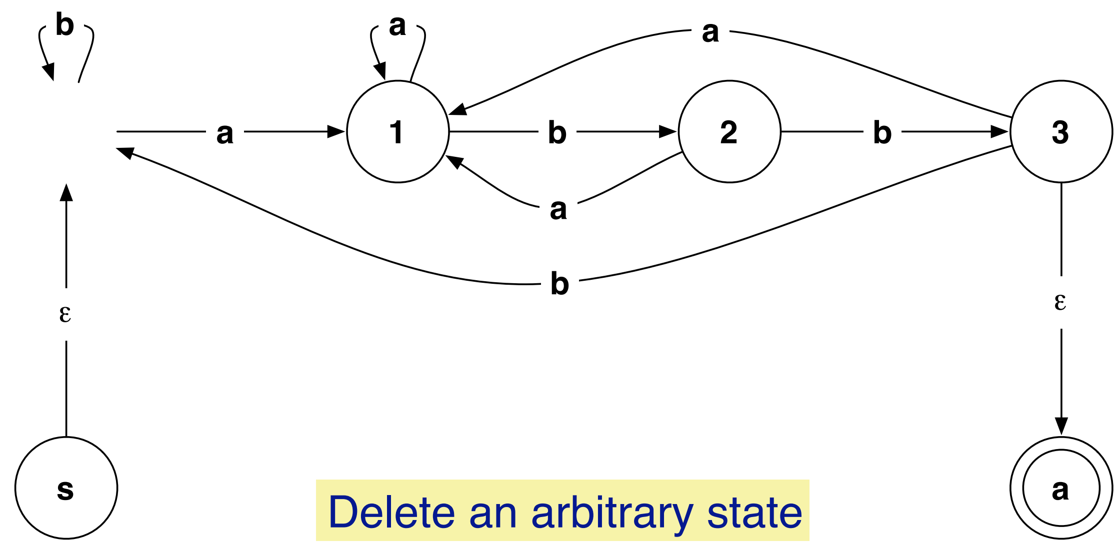


Add new start and accept states

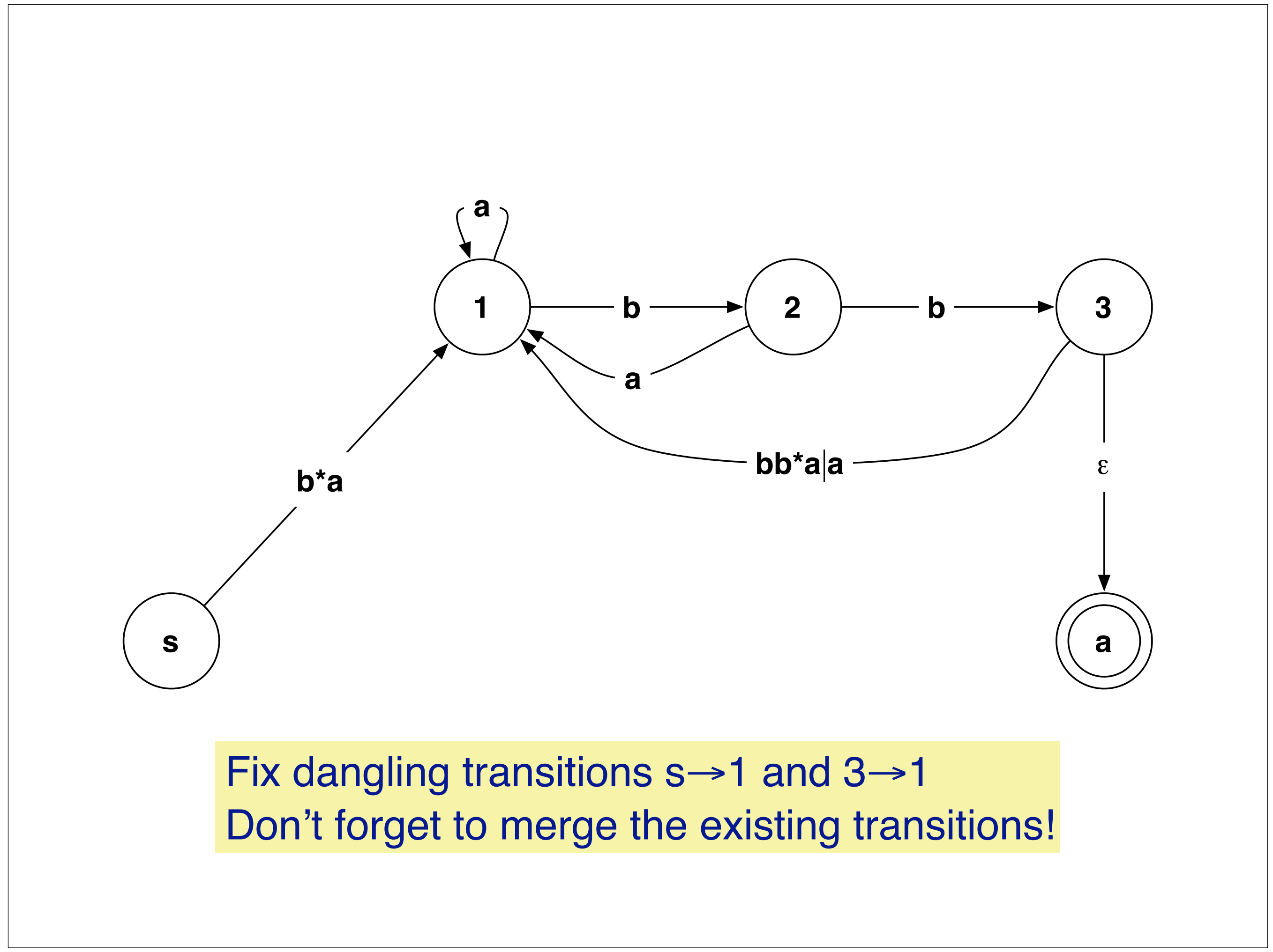


Add missing empty transitions
(we'll just pretend they're there)

This means “you can't get there from here”

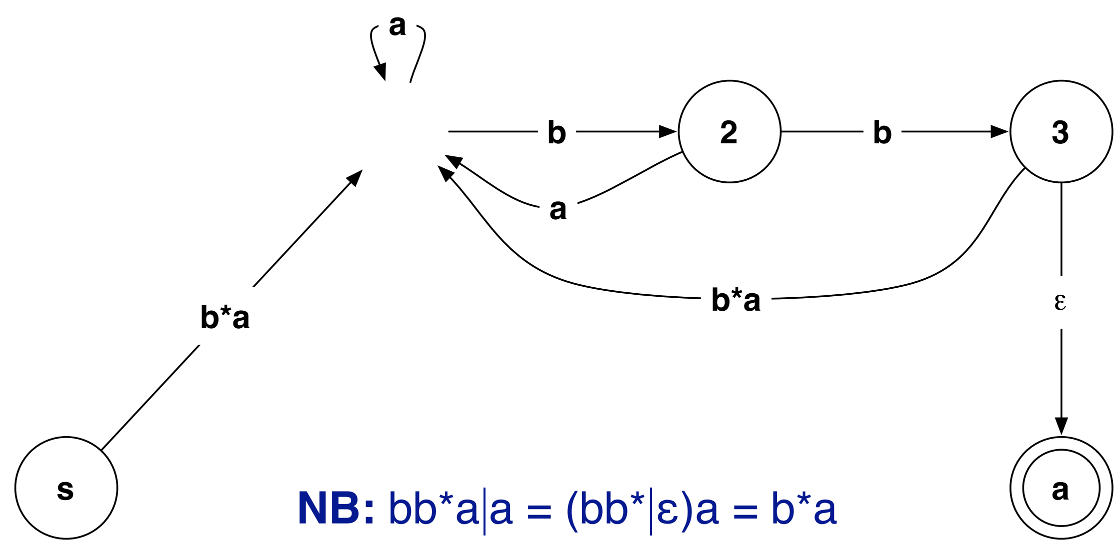


Delete an arbitrary state

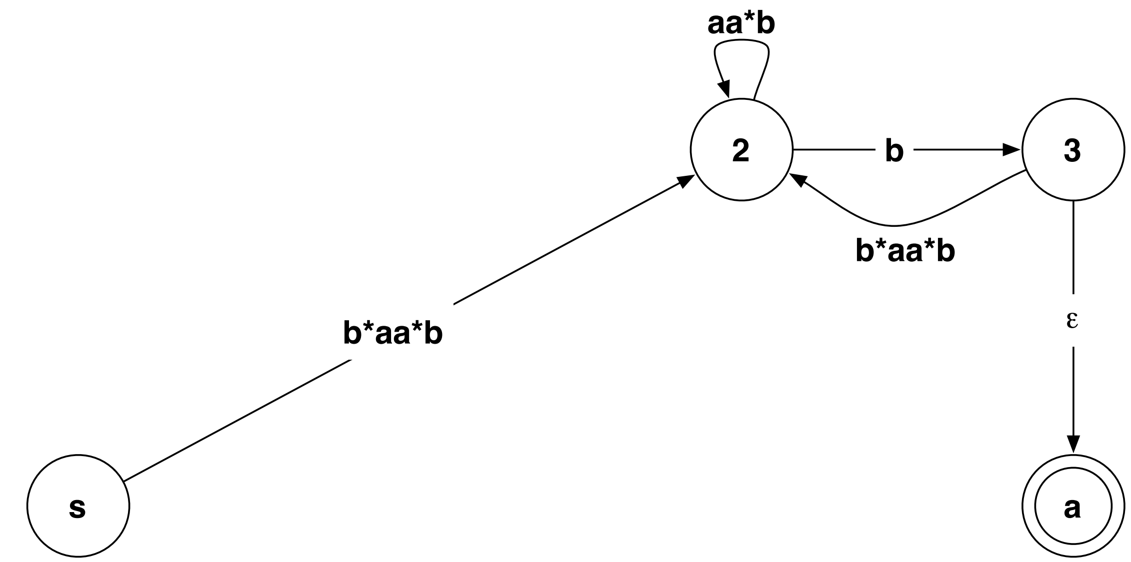


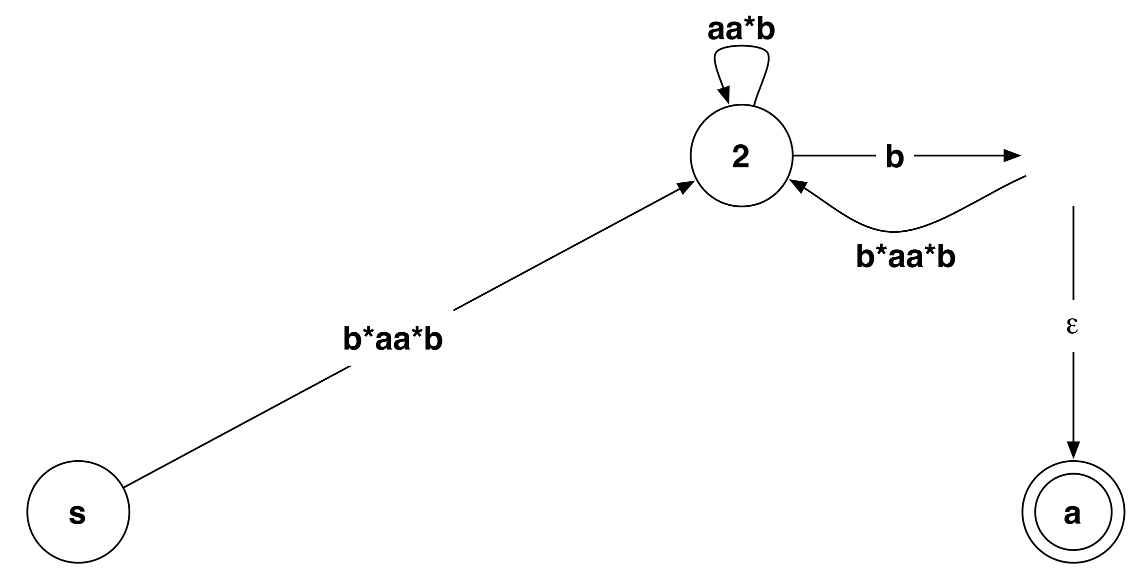
NB: The path from (3) to (1) merges the old path bb*a from (3)->(0)->(1) and the path a from (3)->(1).

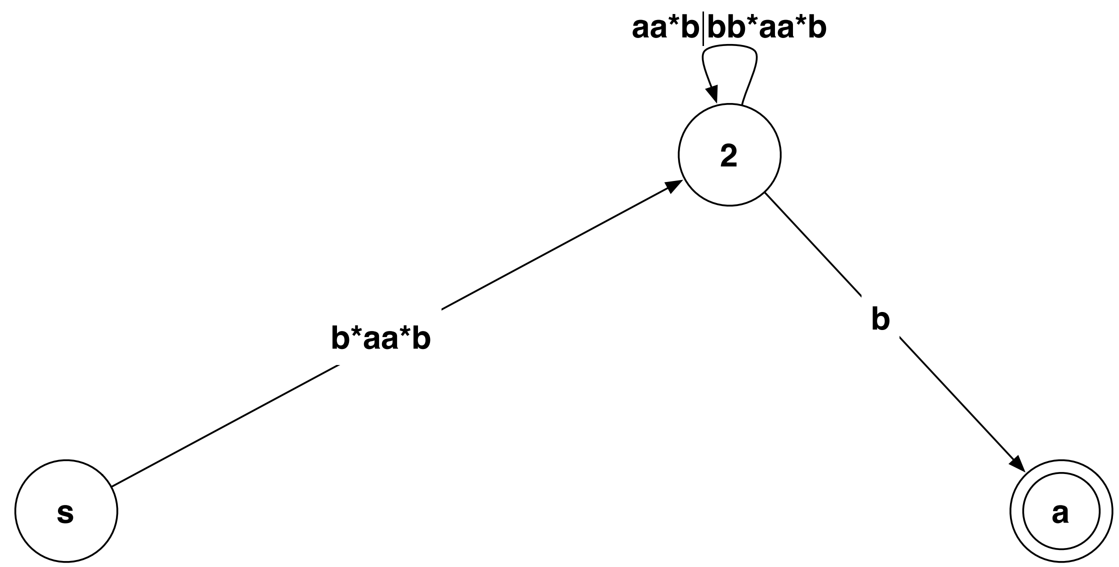
Simplify the RE
Delete another state



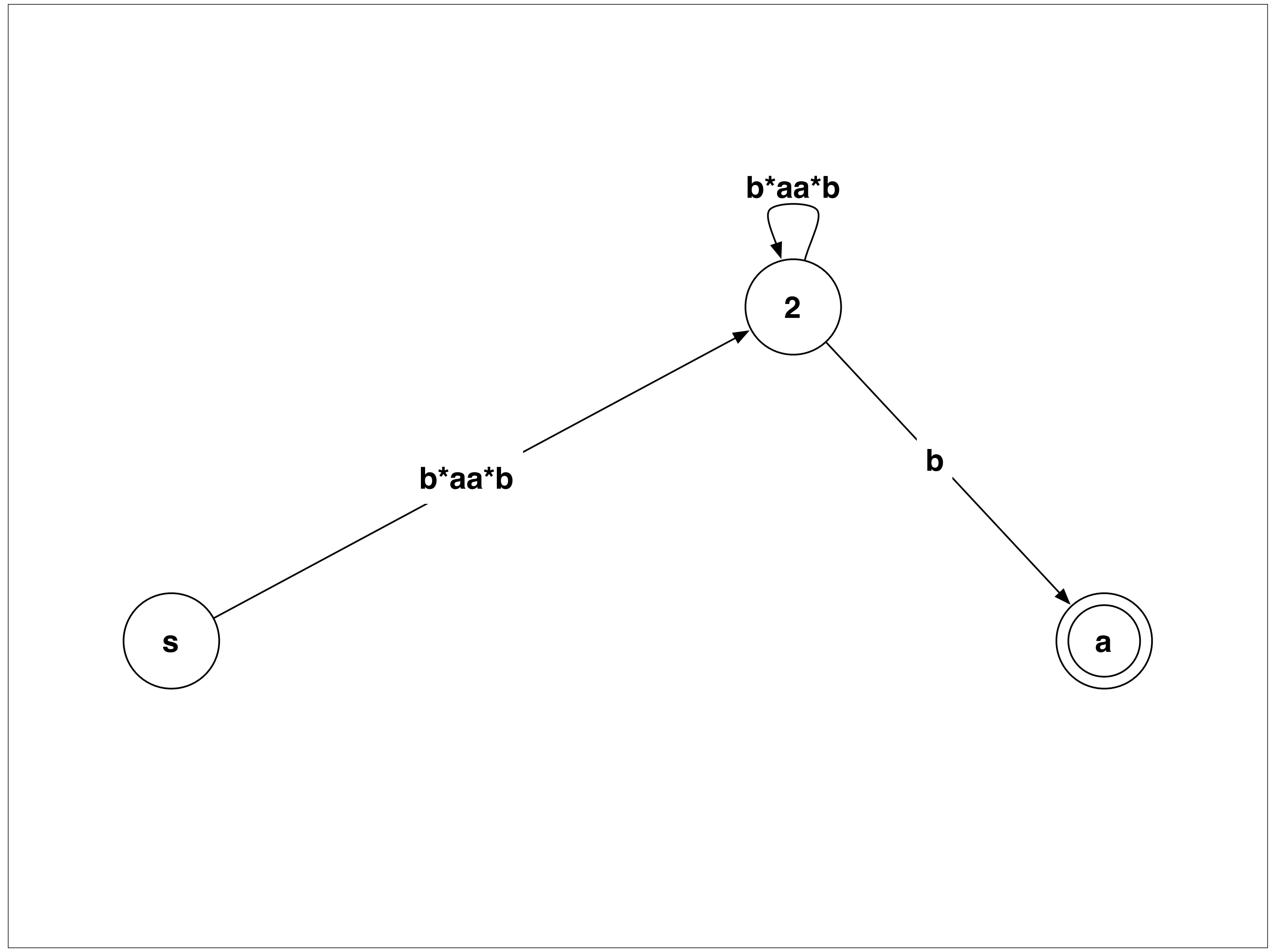
NB: bb*a|a = (bb*|ε)a = b*a

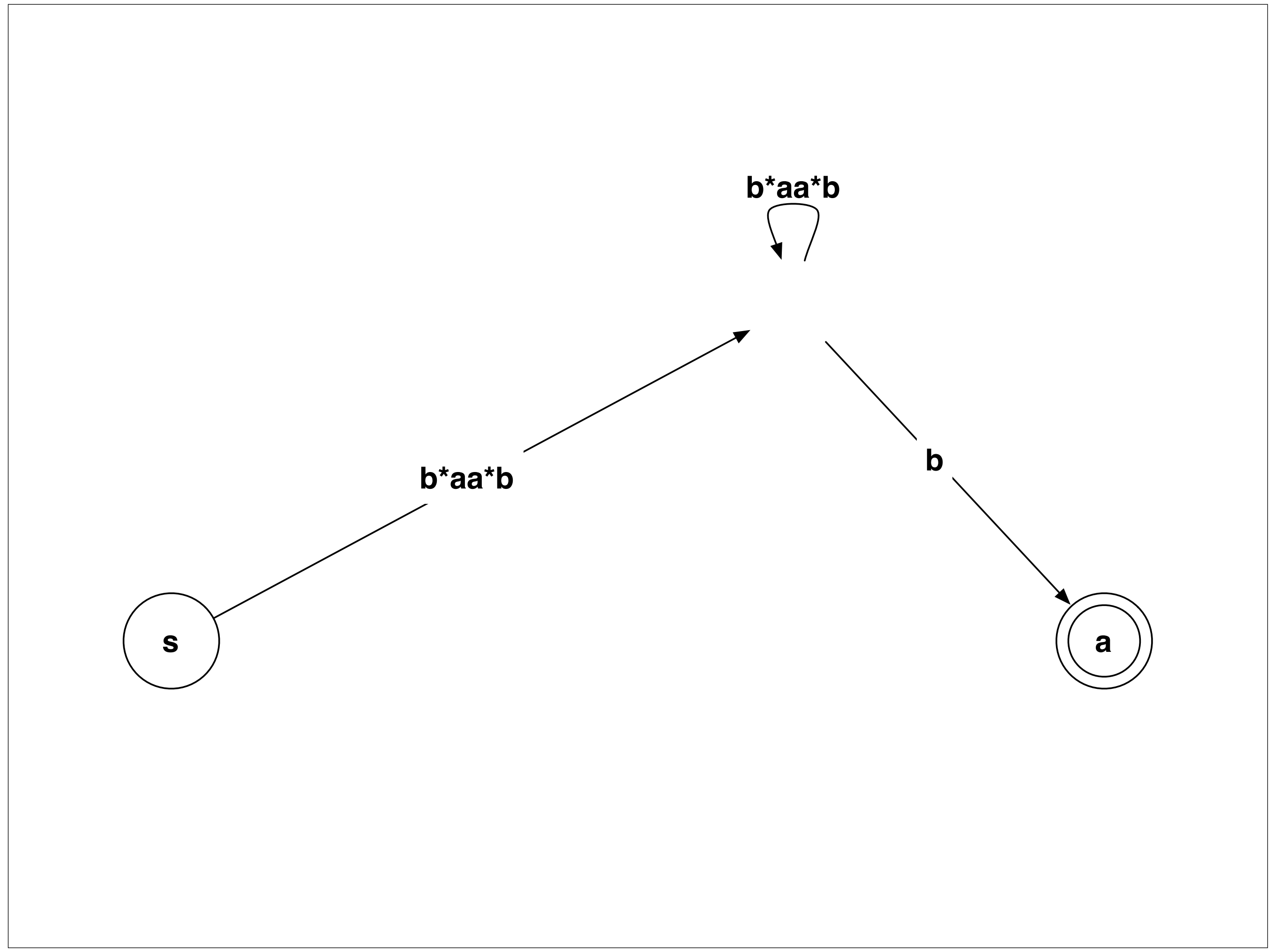

aa*b
2
b
3
b*aa*b
ε
b*aa*b
s
a


aa*b
2
b
b*aa*b
b*aa*b
ε
s
a



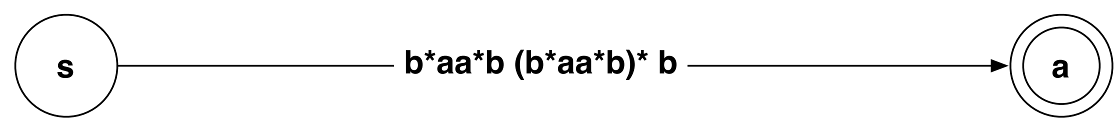
NB: aa*b|bb*aa*b = (ε|bb*)aa*b = b*aa*b


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



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

Hm … not what we expected



Note that b*aa*b = b*a*ab
And so b*aa*b (b*aa*b)* b = (b*a*ab)* b*a*abb
It remains to be shown that (b*a*ab)* b*a* = (a|b)* …

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*. QED

Proof #2 by @grammarware: $(b^*a^*ab)^*b^*a^* = (b^*a^+b)^*b^*a^* = b^*(a^+b^+)^*a^* = b^*(b^*|(a^+b^+)^*)a^* = b^*(b^*|(a^+b^+)^*|a^*)a^* = b^*(a|b)^*a^* = (a|b)^*a^* = (a|b)^*$

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:

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

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
```

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

Compiler needs context to distinguish variables from control constructs!

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?*