Check-In

Review: Compiler Overview

What is an example of an input to a C compiler that would cause a lexical analysis error?

What is an example of an input to a C compiler that would cause a syntactic analysis error?

Administriva

Housekeeping

Project 1

- Out tonight
- Add Flex rules for our language
- Might want to find a project partner!

Administriva
Housekeeping

What should we call our language?

Surveys (Mostly) Processed

Housekeeping

- We <u>will</u> do flipped Wednesdays
- I'll try to put out some 510 review

EECS 665
COMPILER
CONSTRUCTION

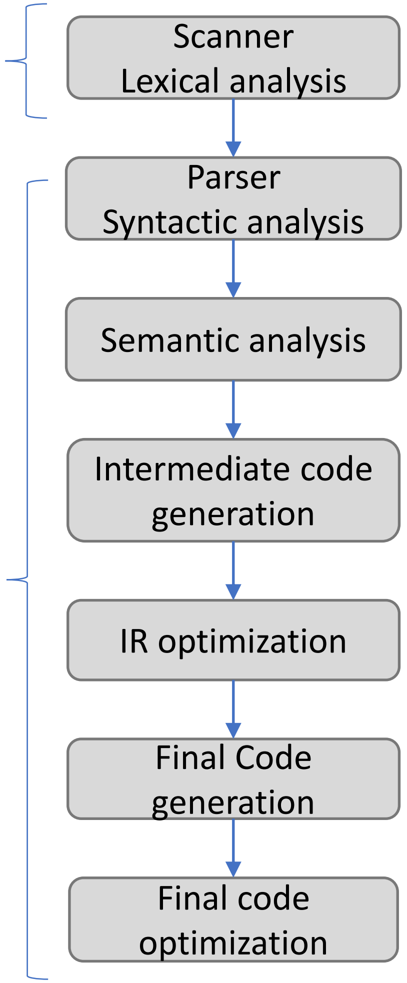
2 – Implementing Scanners

Compiler Construction

Progress Pics

Currently working on lexical analysis concepts

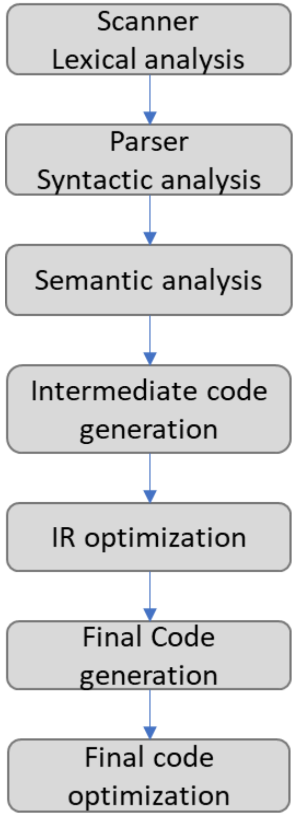
- Convert the character stream from the user into the token stream (the “words” of the programming language)



Last Time

Review: Overview

Introduced a definition of a compiler and its workflow



Used RegExs to *specify* languages of tokens

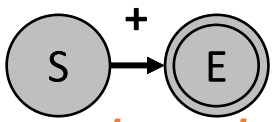
<u>Token</u>
Integer Literal

<u>RegEx</u>
0|(1|2|…|9)(0|1|…|9)*

Claimed RegExs can be automatically translated to *recognize* languages

<u>RegEx</u>
"+"

<u>DFA</u>



In lab we showed a tool that automatically does this translation (Flex)

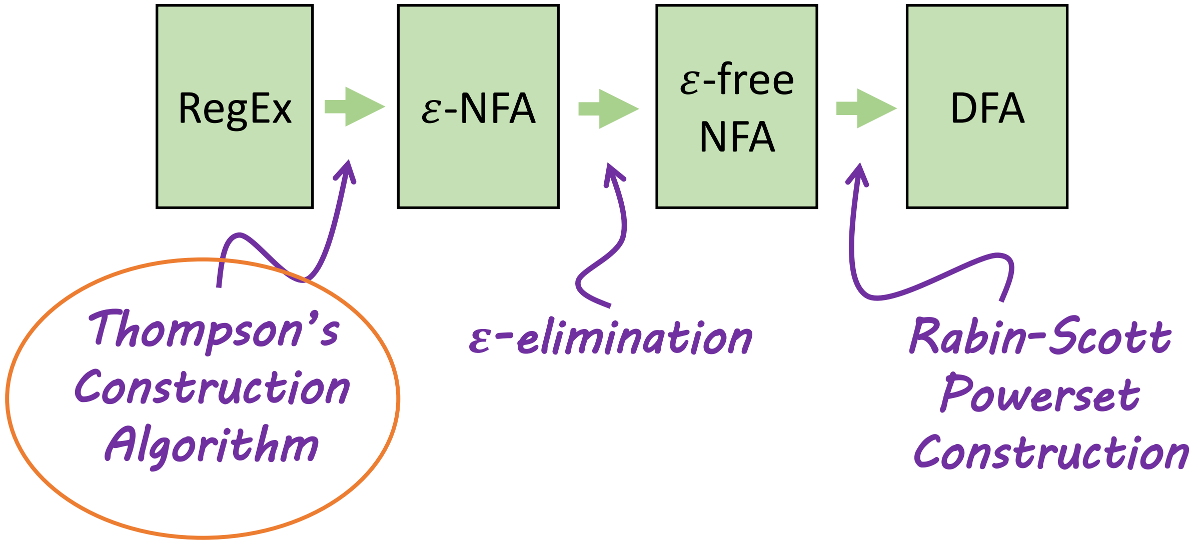
<u>You Should Know</u>

- The phases of the compiler and what each step does
- What errors the various phases catch
- How to specify a token's lexemes with a regex

Lecture Overview

Lecture 2 – Implementing Scanners

Walk through the translation process formally



Key Concept

Thompson's Construction Algorithm

Use an *expression tree*:

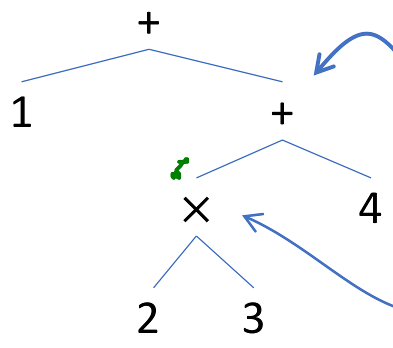
- Leaf: atomic operand
- Branch: operations joining subtrees

Expression Tree Examples

Arithmetic Expression

$1 + 2 \times 3 + 4$

Arithmetic Expression Tree



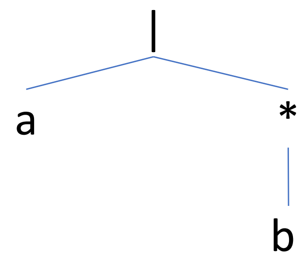
low precedence operators shallow in the tree

NOTE: high precedence operators deep in the tree

RegEx

a|b*

Expression Tree



Thompson's Construction Intuition

Two-Step Process:

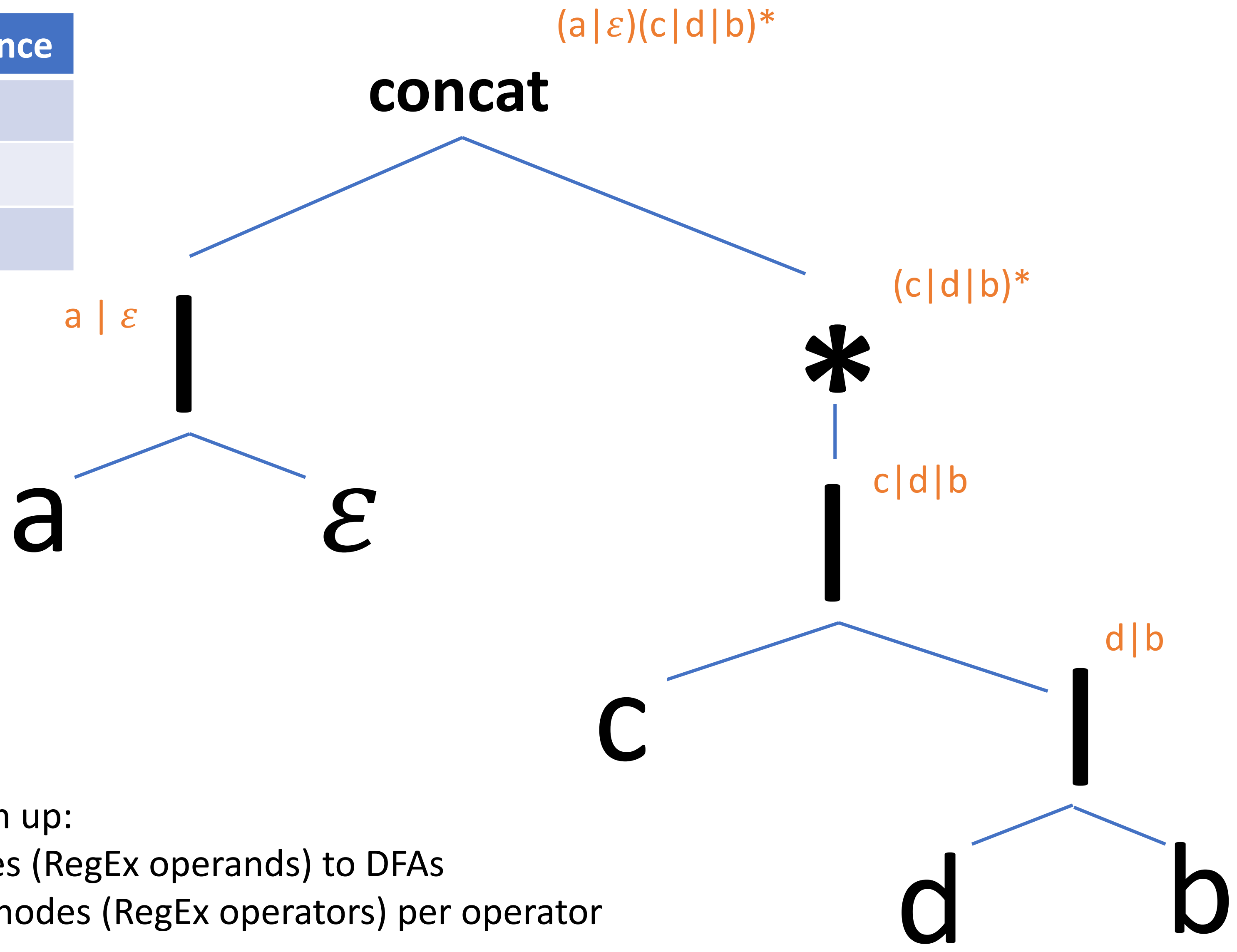
- Break the RegEx down to the simplest units with "obvious" FSMs (i.e. expression tree leaves)
- Combine the sub-FSMs according to operator rules (i.e. expression tree branch rules)

Recombobulate the Regex into an FSM piecewise

Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Op	Precedence
*	High
concat	Medium
|	Low

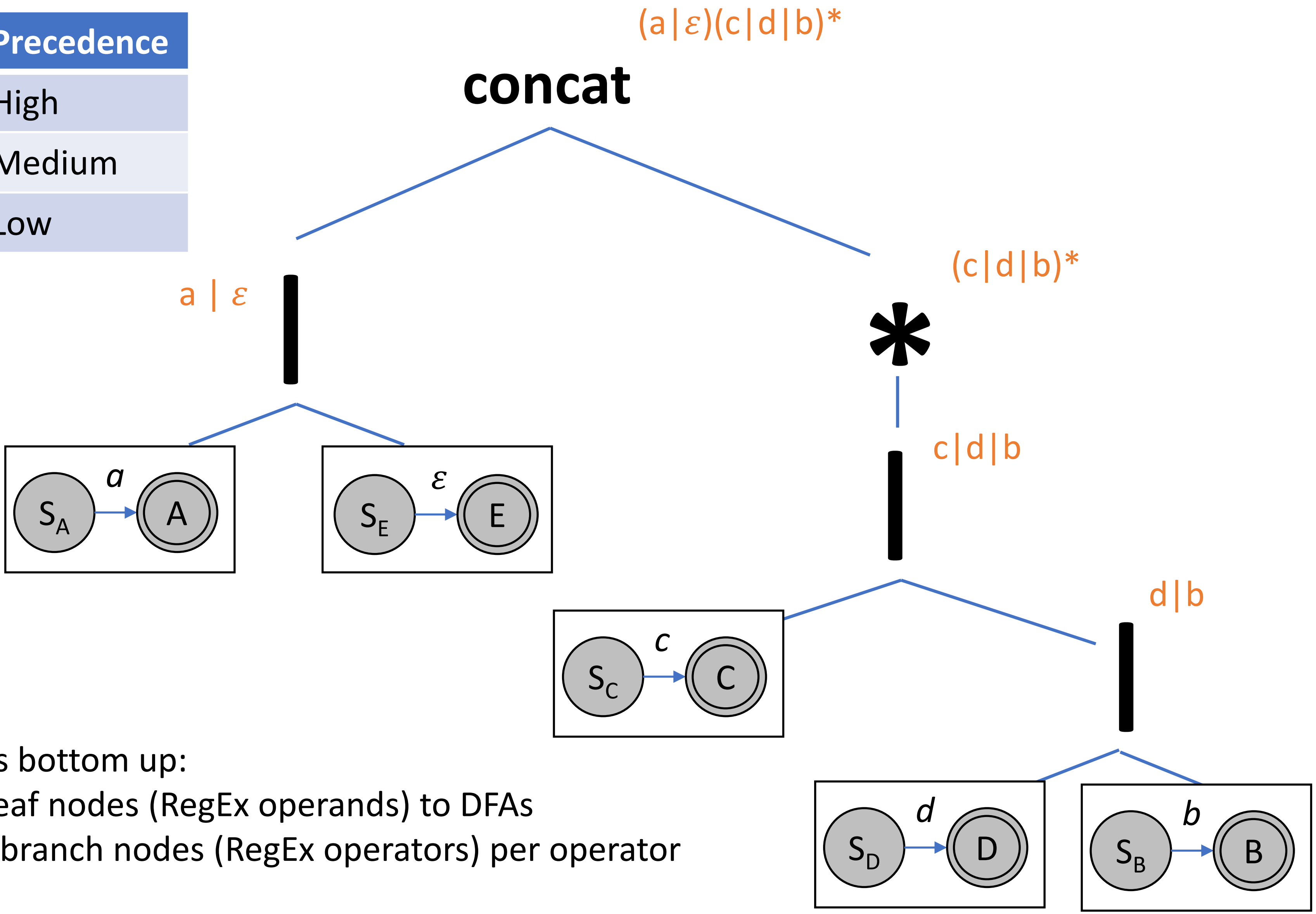


Apply rules bottom up:
- convert leaf nodes (RegEx operands) to DFAs
- combine branch nodes (RegEx operators) per operator

Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Op	Precedence
*	High
concat	Medium
|	Low



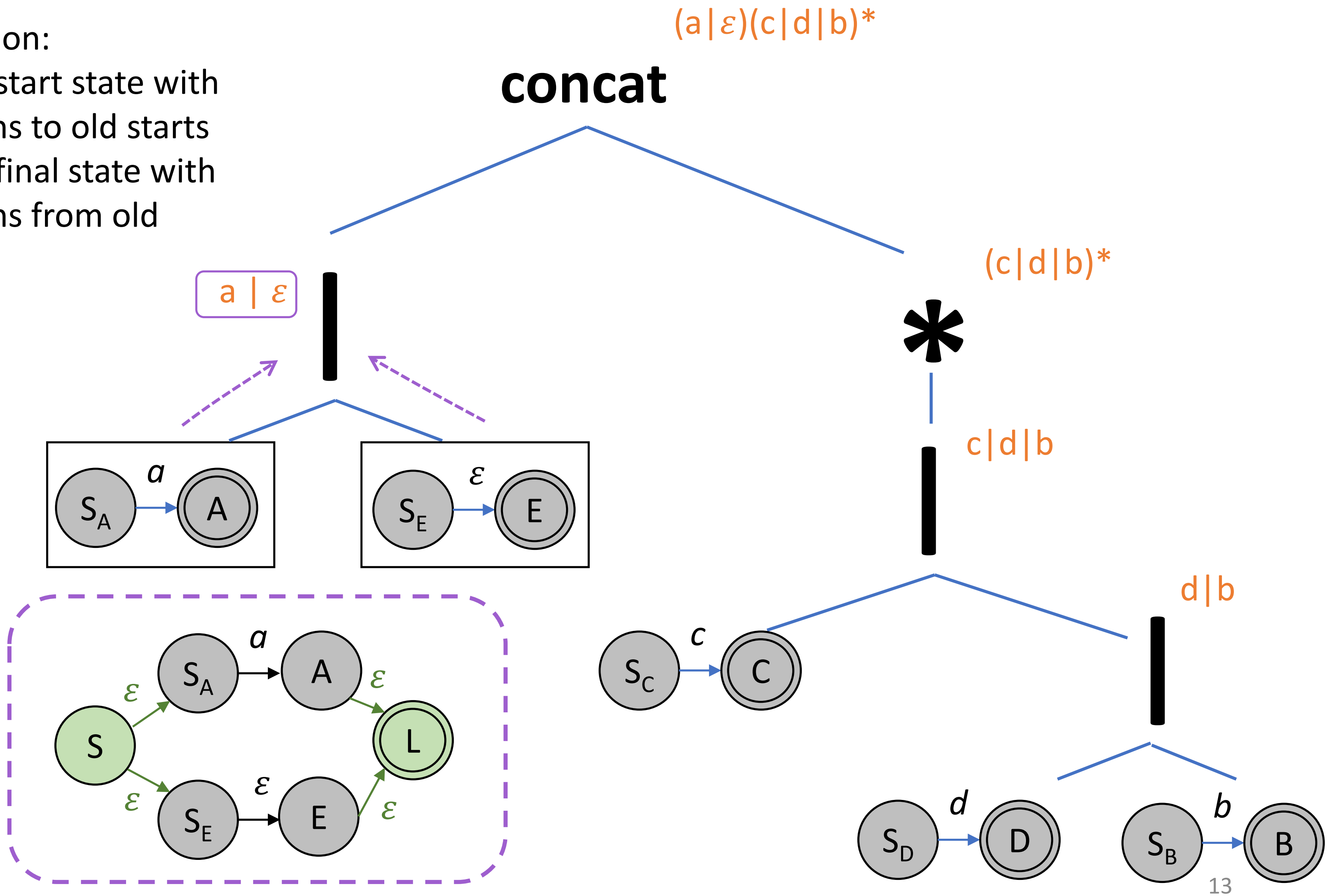
Apply rules bottom up:
- convert leaf nodes (RegEx operands) to DFAs
- combine branch nodes (RegEx operators) per operator

Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Alternation:

- New start state with ε-trans to old starts
- New final state with ε-trans from old finals

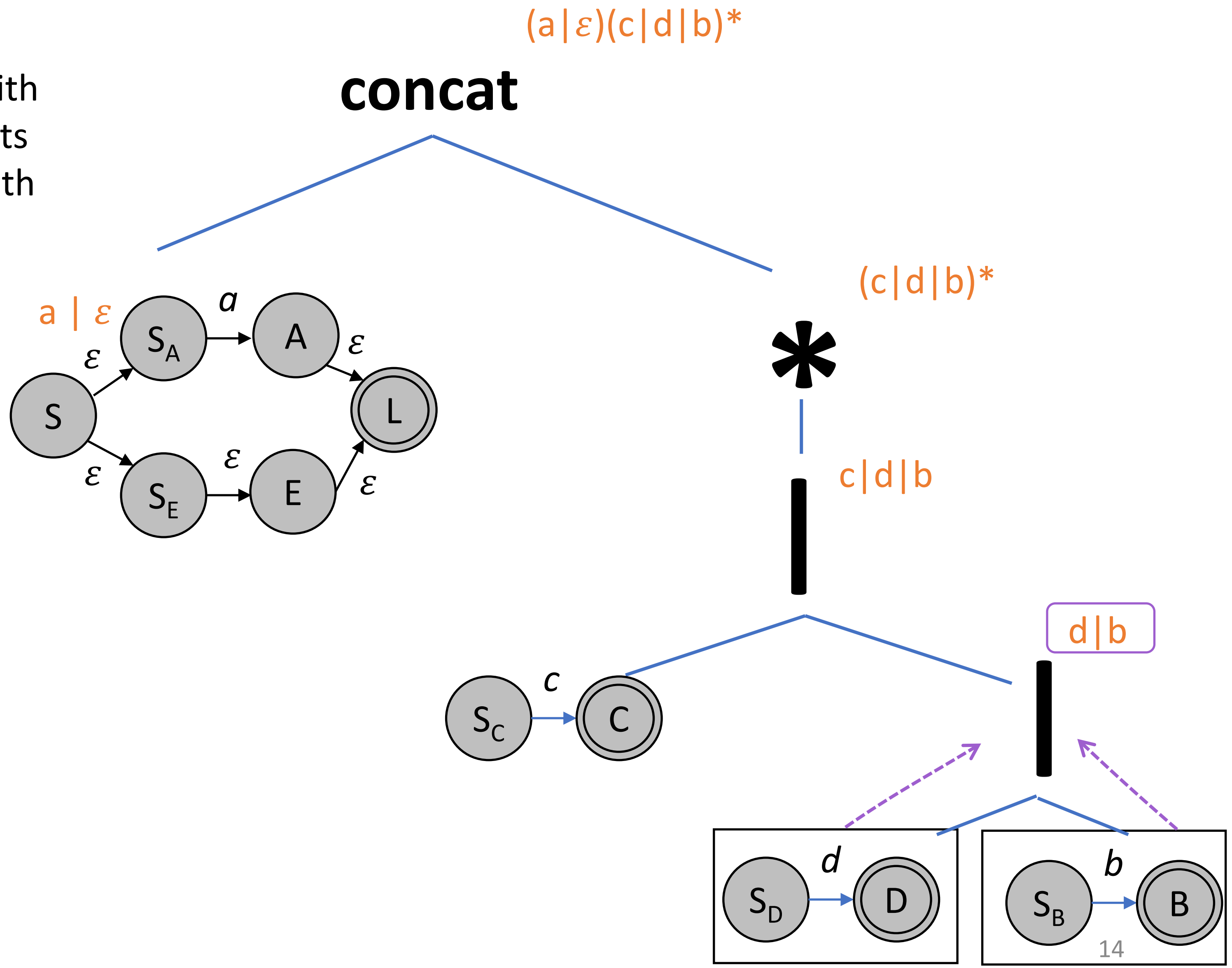


Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Alternation:

- New start state with ε-trans to old starts
- New final state with ε-trans from old finals

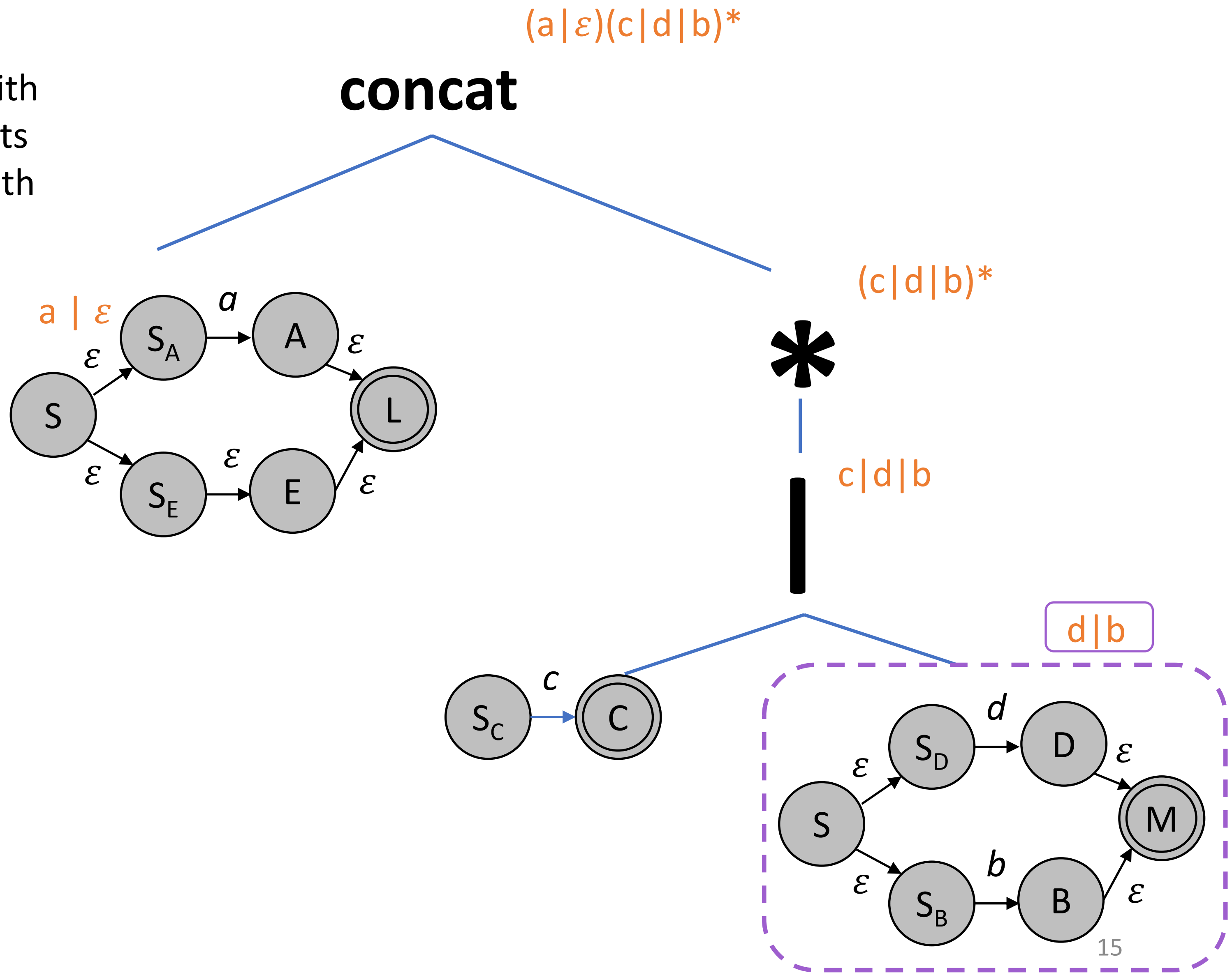


Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Alternation:

- New start state with ε-trans to old starts
- New final state with ε-trans from old finals

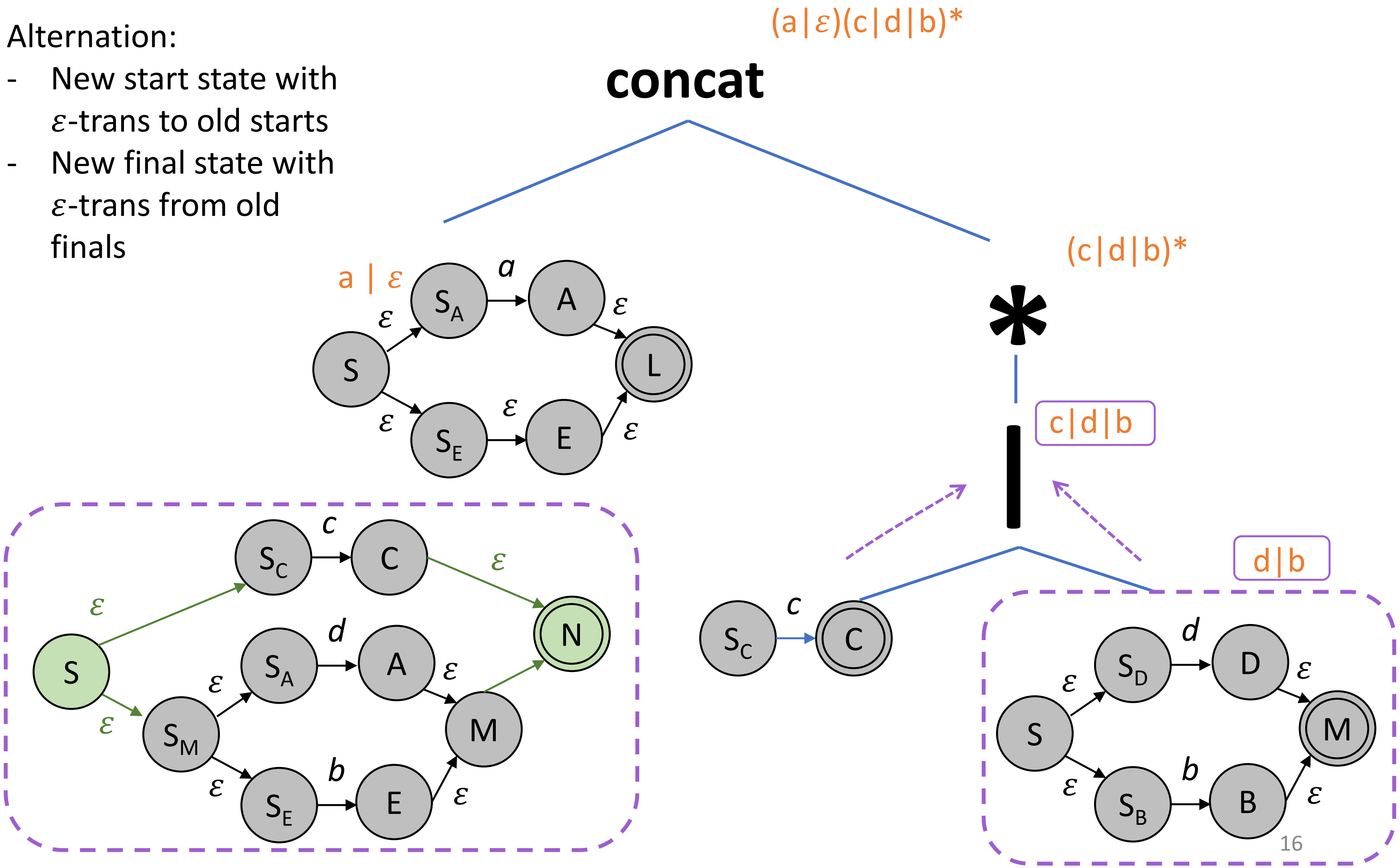


Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Alternation:

- New start state with ε-trans to old starts
- New final state with ε-trans from old finals

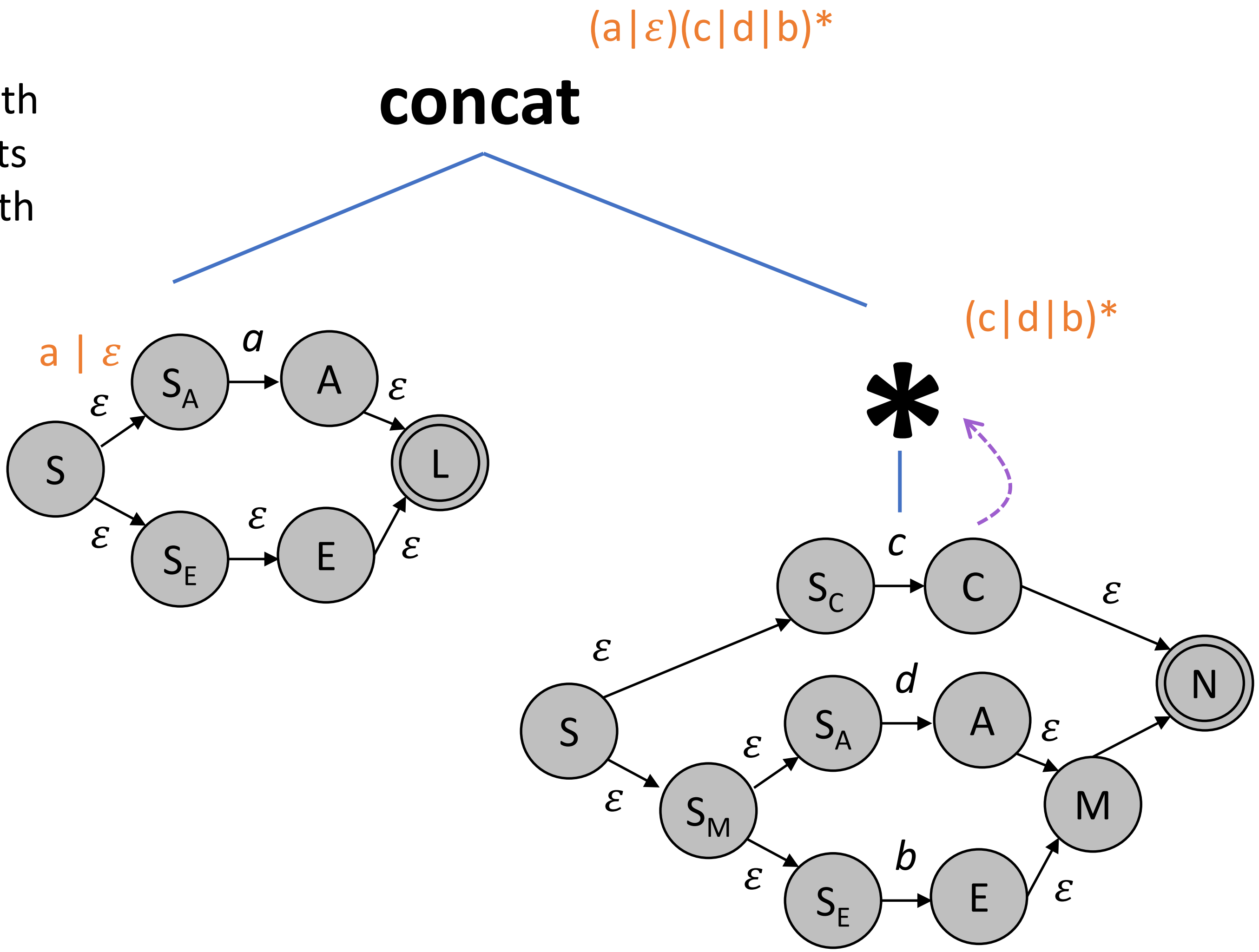


Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Alternation:

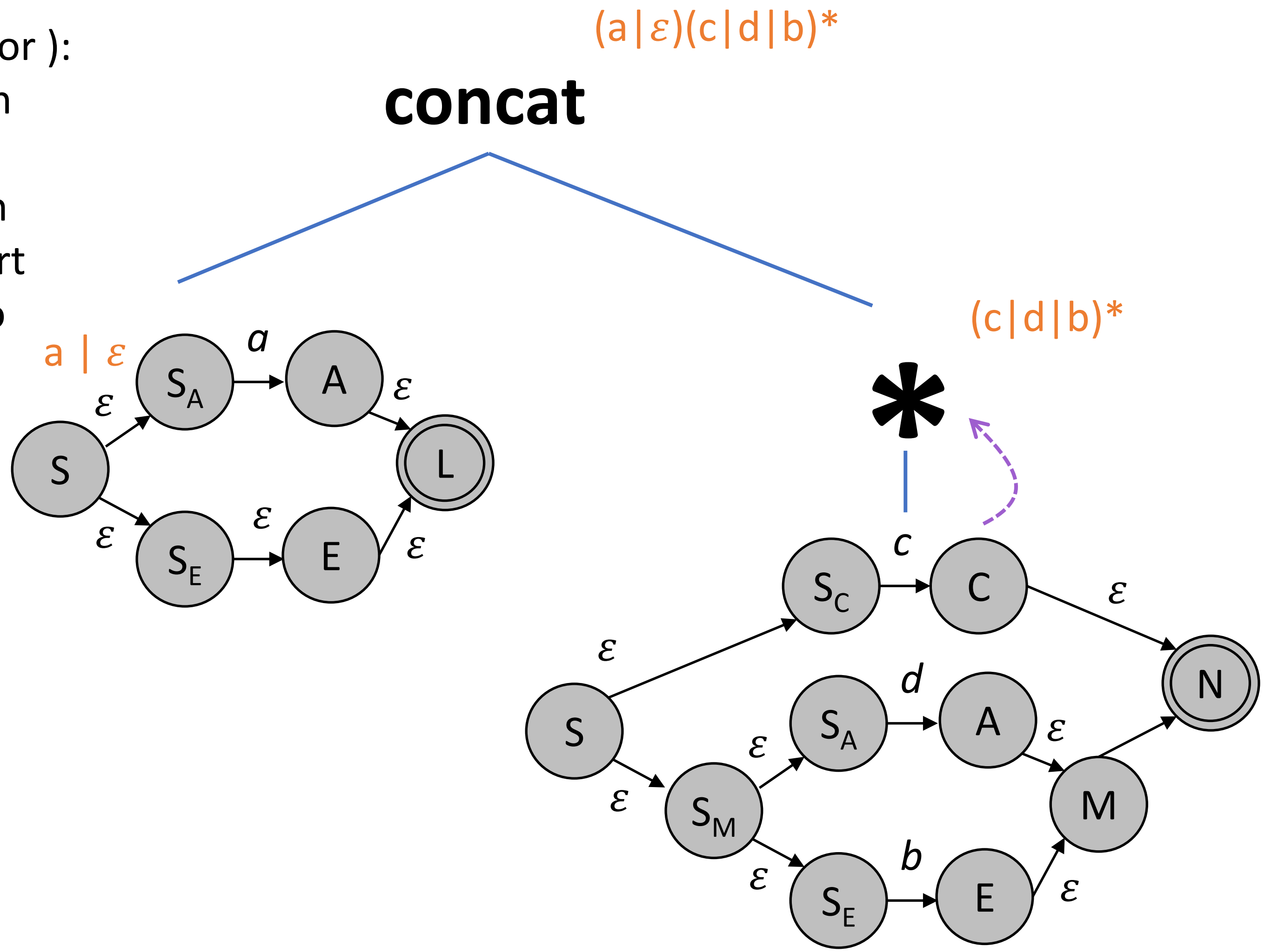
- New start state with ε-trans to old starts
- New final state with ε-trans from old finals



Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

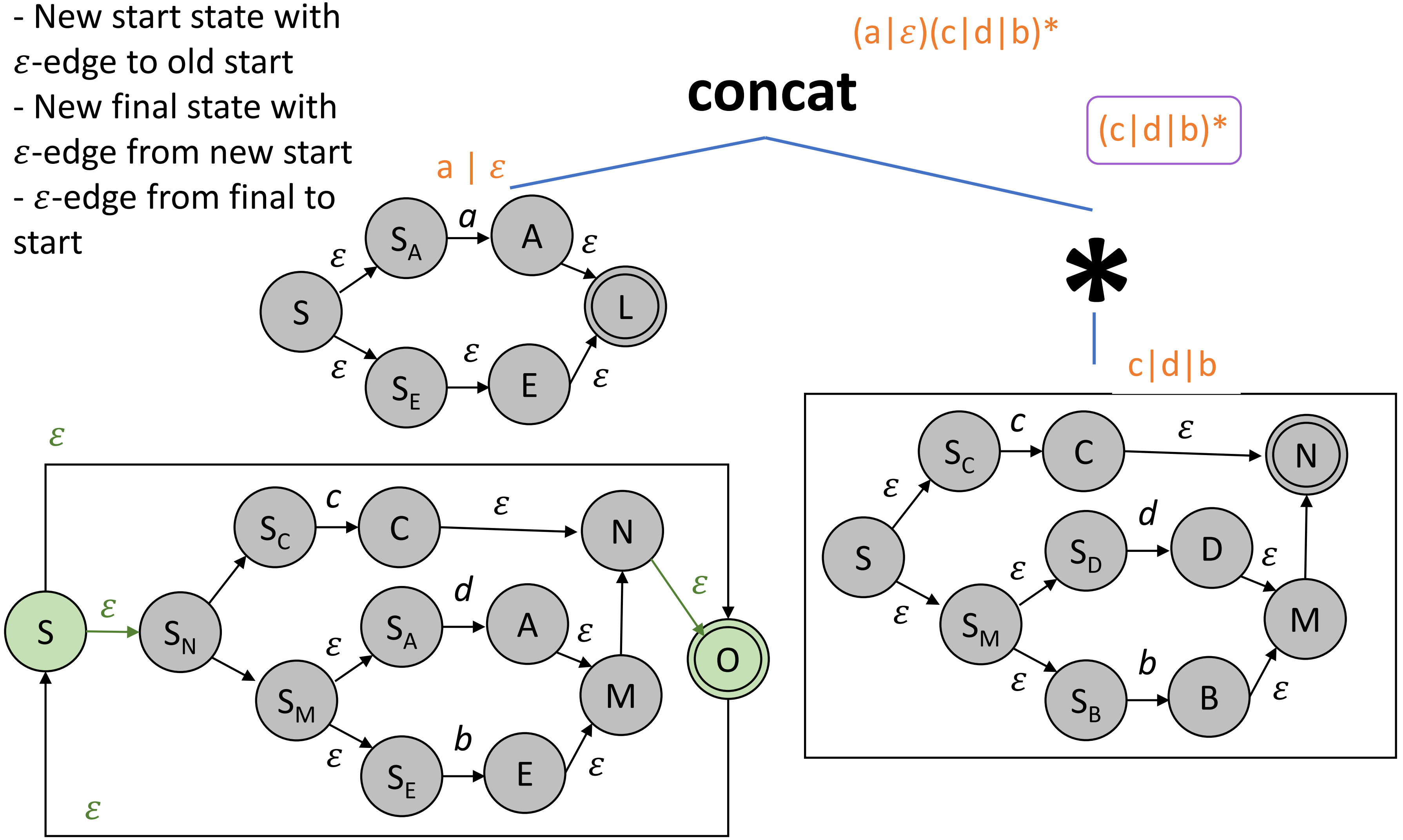
Repetition (* operator):
- New start state with ε-edge to old start
- New final state with ε-edge from new start
- ε-edge from final to start



Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

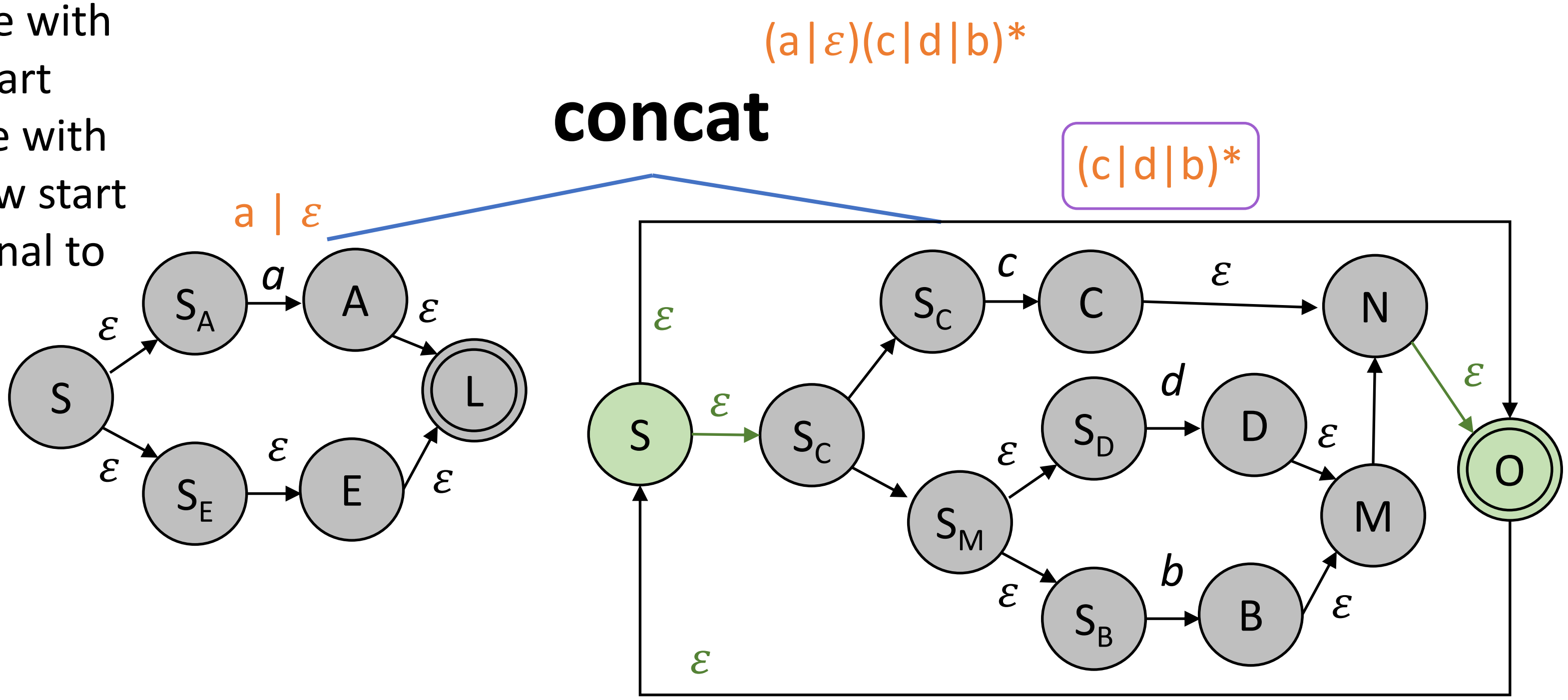
Repetition (* operator):
- New start state with ε-edge to old start
- New final state with ε-edge from new start
- ε-edge from final to start



Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

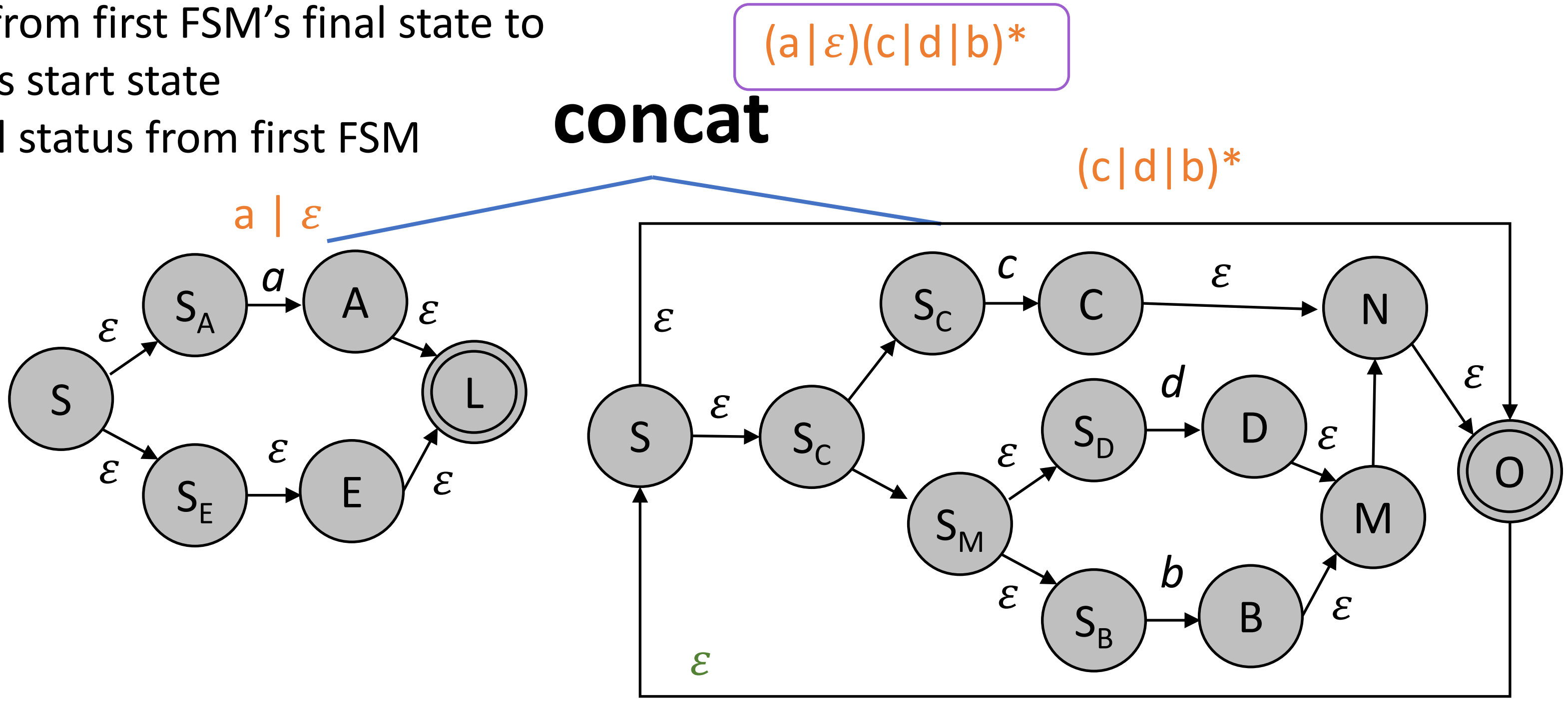
Repetition (* operator):
- New start state with ε-edge to old start
- New final state with ε-edge from new start
- ε-edge from final to start



Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Concatenation:
- Add ε-edge from first FSM's final state to second FSM's start state
- Remove final status from first FSM

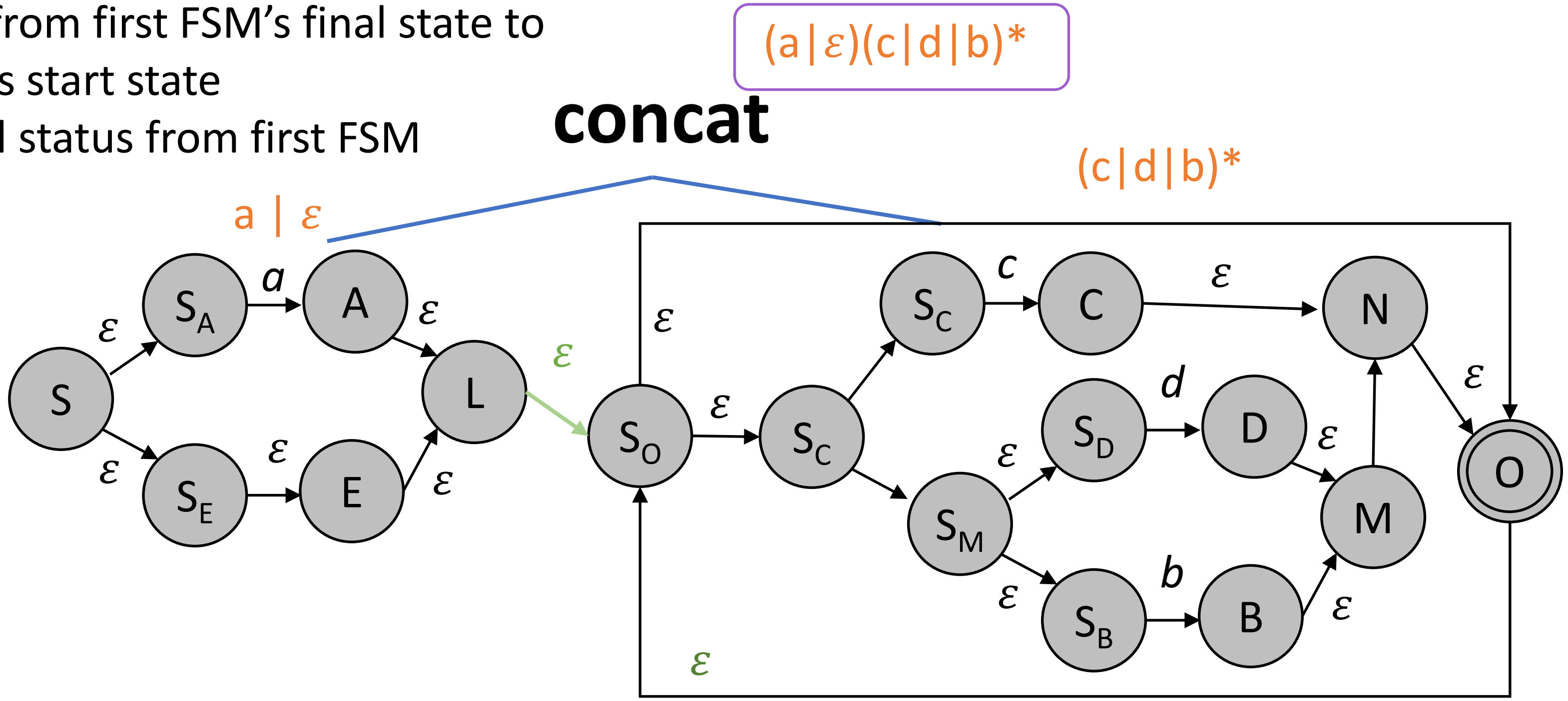


Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

Concatenation:

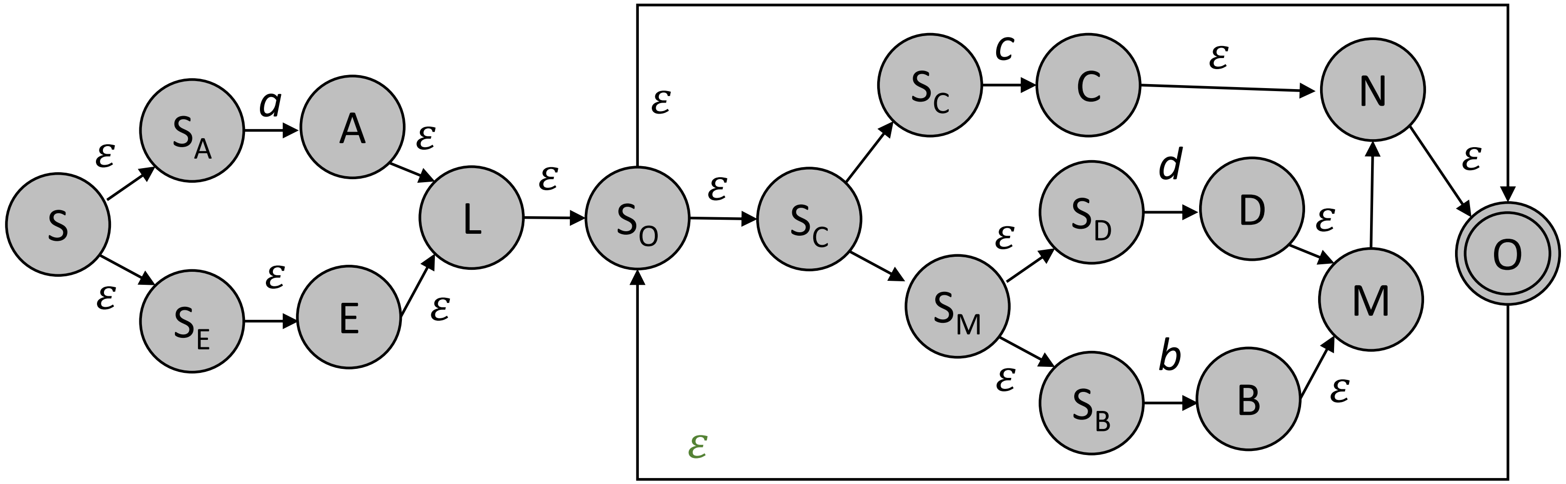
- Add ε-edge from first FSM's final state to second FSM's start state
- Remove final status from first FSM



Thompson's Construction Alg.

Build the RegEx Tree | Replace nodes bottom-up

(a|ε)(c|d|b)*

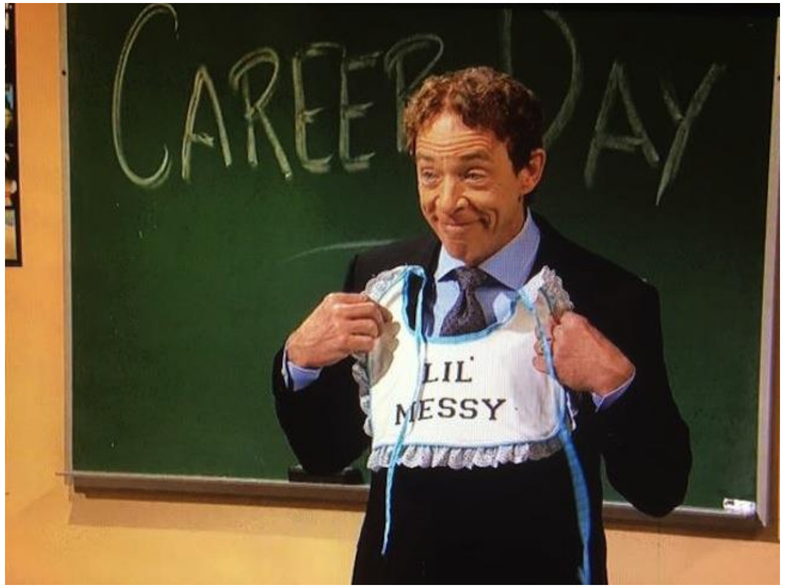


Thompson's Construction: Side-Note

Build the RegEx Tree | Replace nodes bottom-up

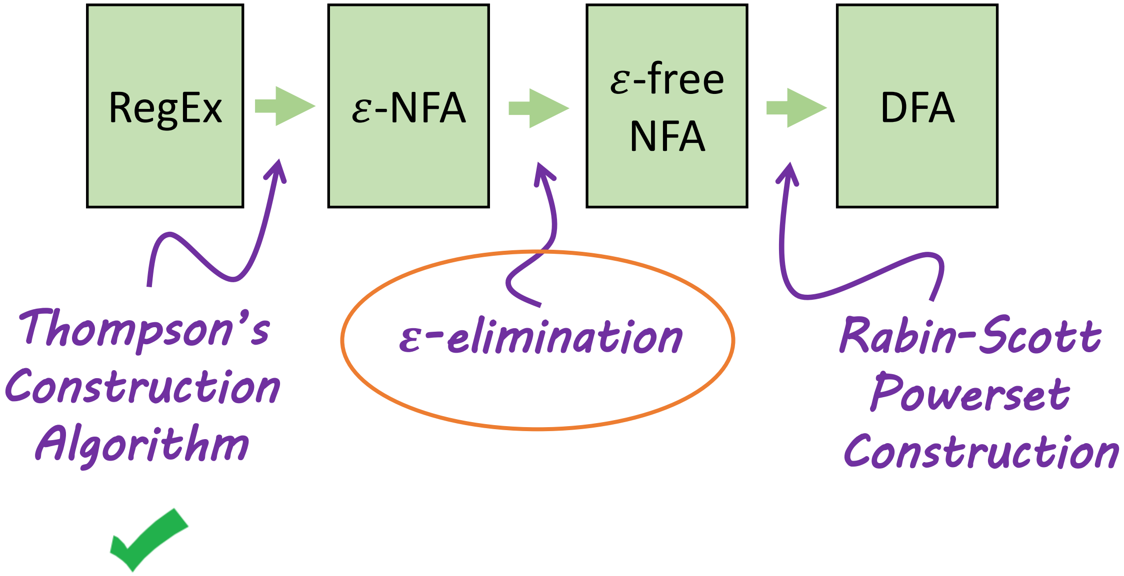
The FSMs produced by Thompsons Construction are a little bit messy!

- Clearly less efficient than what we would do by hand
- Designed for ease of proofs
- In practice, it's easy to minimize FSMs later



From RegEx to DFA

Lecture 2 – Implementing Scanners



Eliminating ε-transitions

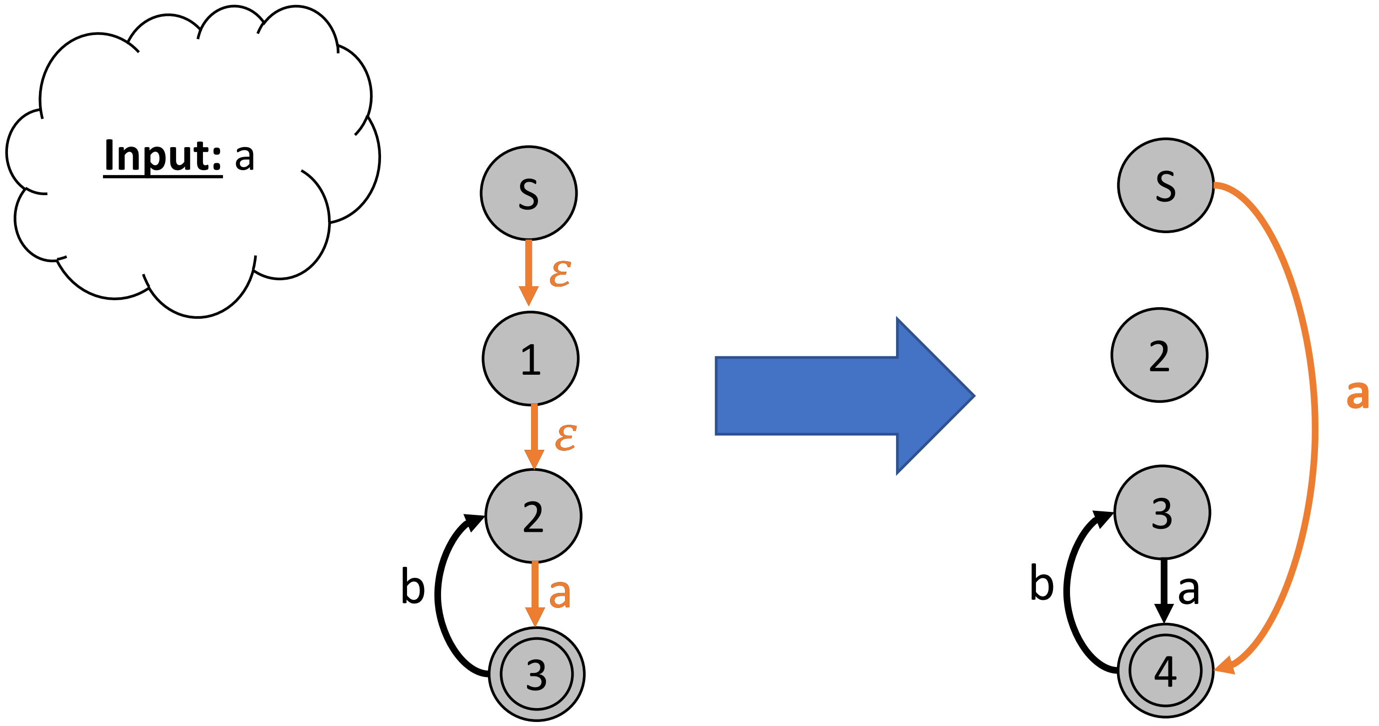
Lecture 2 – Implementing Scanners

Observation: You never see an epsilon in the input

- Consuming a character means taking a “chain” of zero-or-more ε-edges then a real character edge

Algorithm Intuition: *cut out the middleman*

- Replace all “chains” with a direct real-character edge



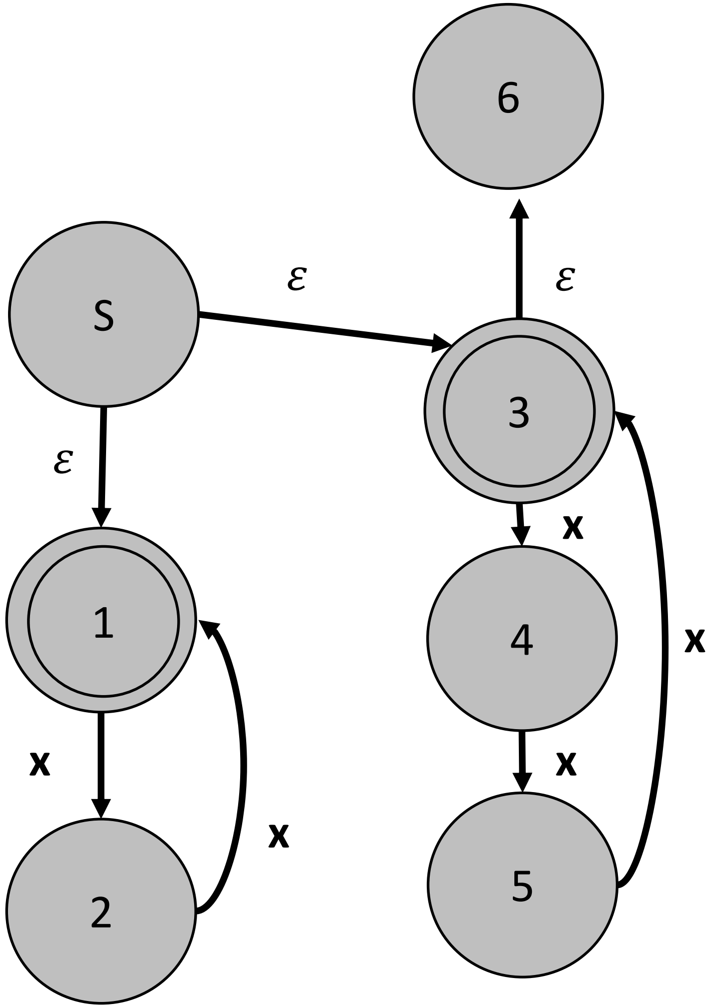
Eliminating ε-transitions

- Compute ε-close(s), the set of states reachable via 0 or more ε-edges from s
- Copy all states from N to an ε-free version, N’
- Put s in F’ if ε-close(s) contains a state in F
- Put s,c → t in δ’ if there is a c-edge to t in ε-close(s)

Example, Step I

Eliminating ε-Transitions

Let ε-close(s) be the set of states reachable via 0 or more ε-edges



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

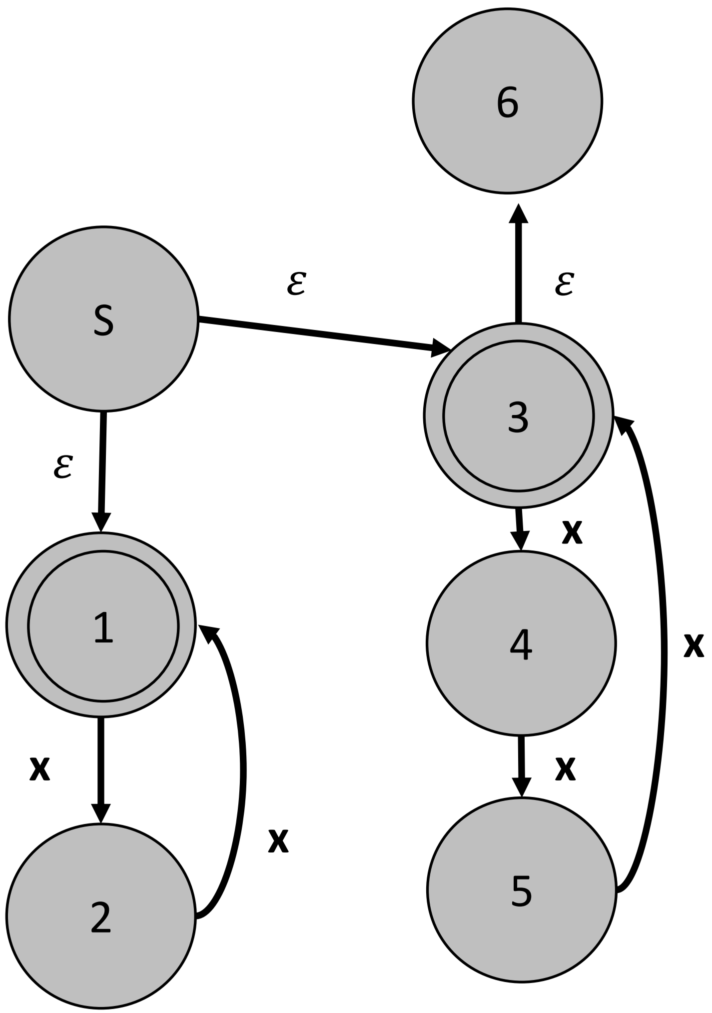
ε-close(5) = {5}

ε-close(6) = {6}

Example, Step II

Eliminating ε-Transitions

Copy all states from N to N’



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

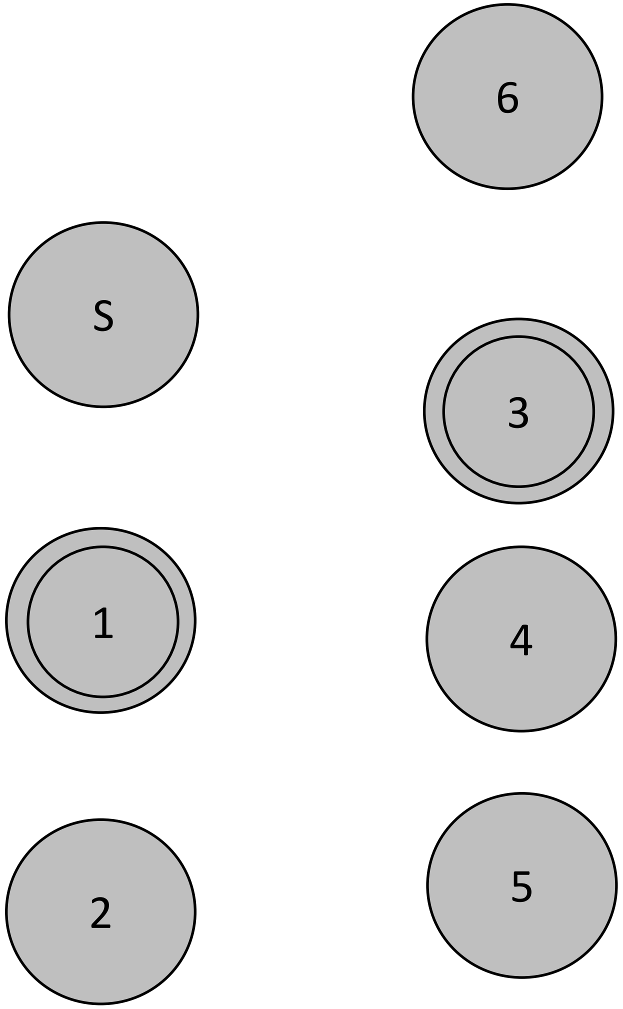
ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

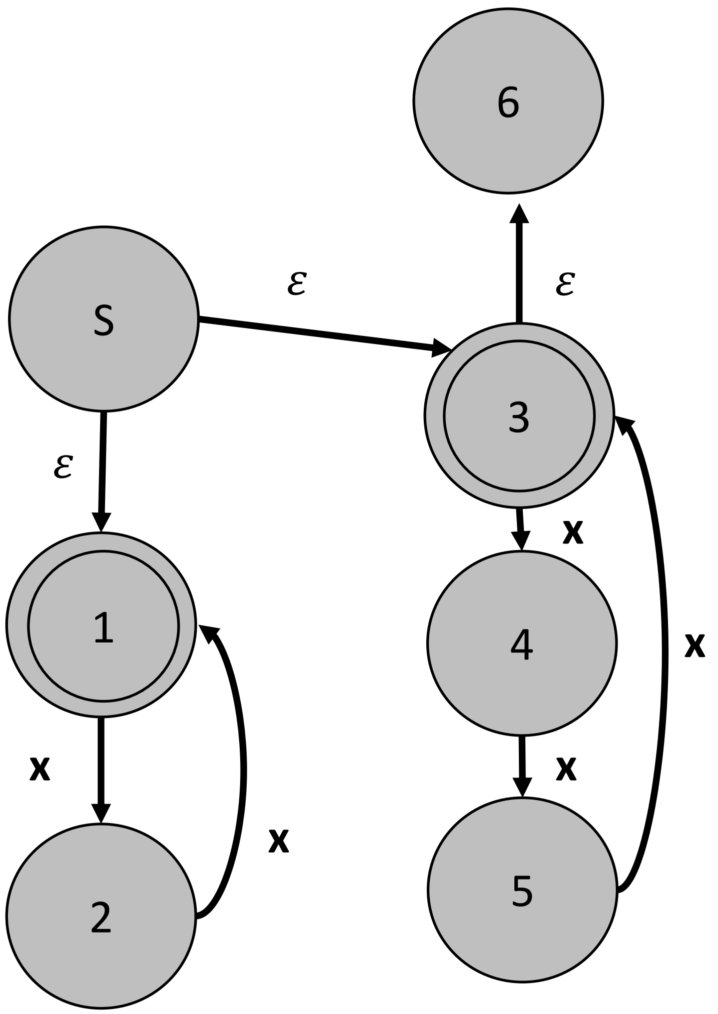
ε-close(5) = {5}

ε-close(6) = {6}



Example, Step III
Eliminating ε-Transitions

Put s in F' if ε-close(s) contains a state in F



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

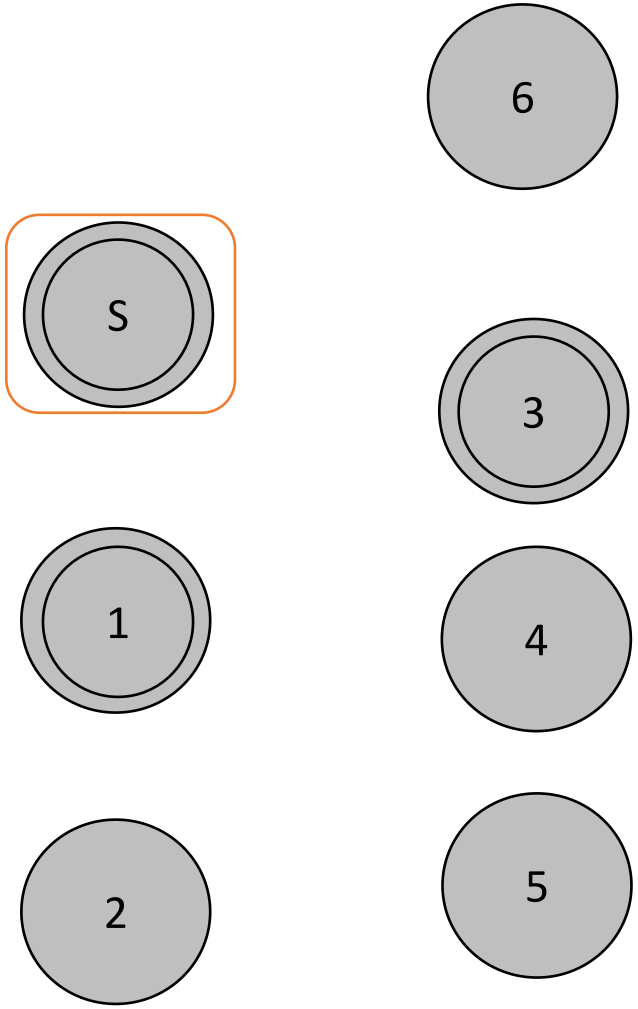
ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

ε-close(5) = {5}

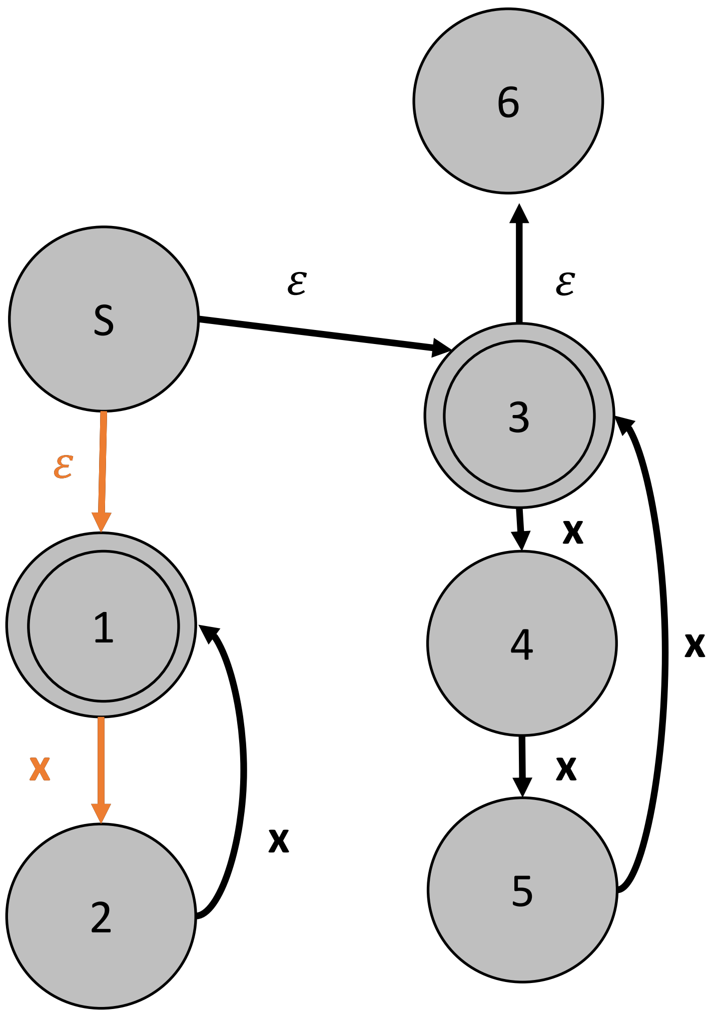
ε-close(6) = {6}



Example, Step IV

Eliminating ε-Transitions

Put s,c $\rightarrow$ t in δ' if there is a c-edge to t in ε-close(s)



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

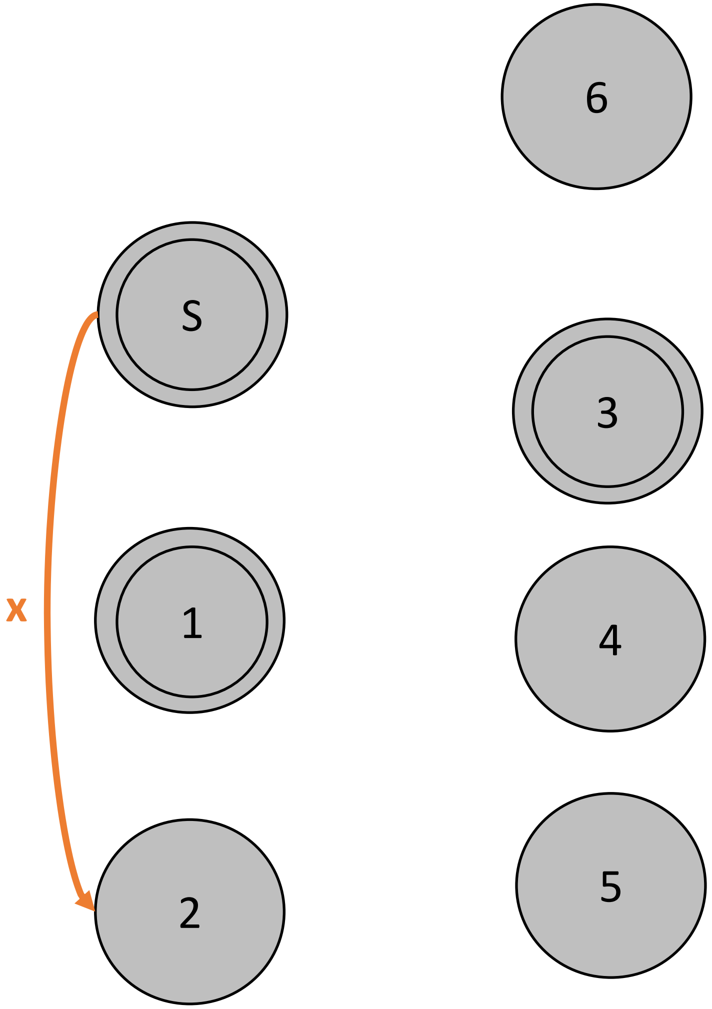
ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

ε-close(5) = {5}

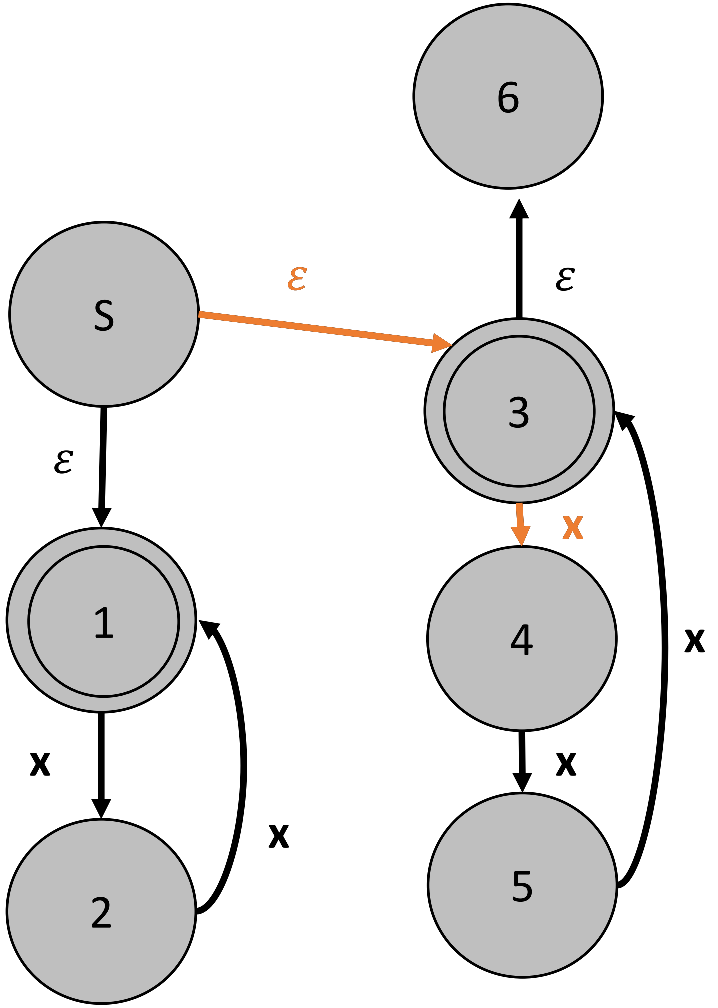
ε-close(6) = {6}



Example, Step IV

Eliminating ε-Transitions

Put s,c $\rightarrow$ t in δ' if there is a c-edge to t in ε-close(s)



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

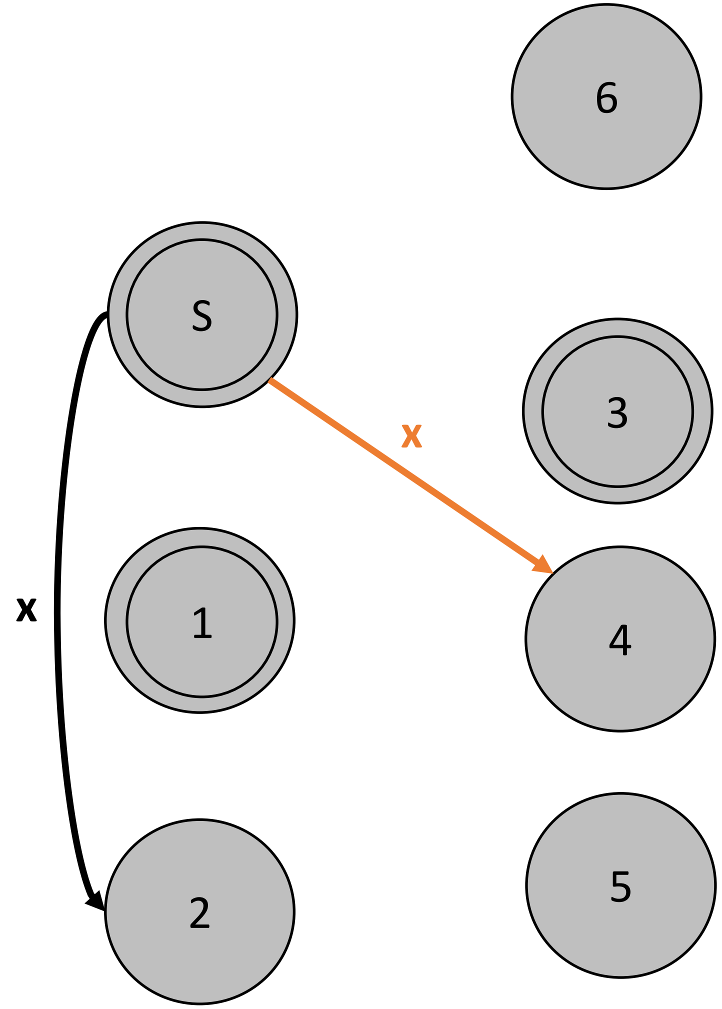
ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

ε-close(5) = {5}

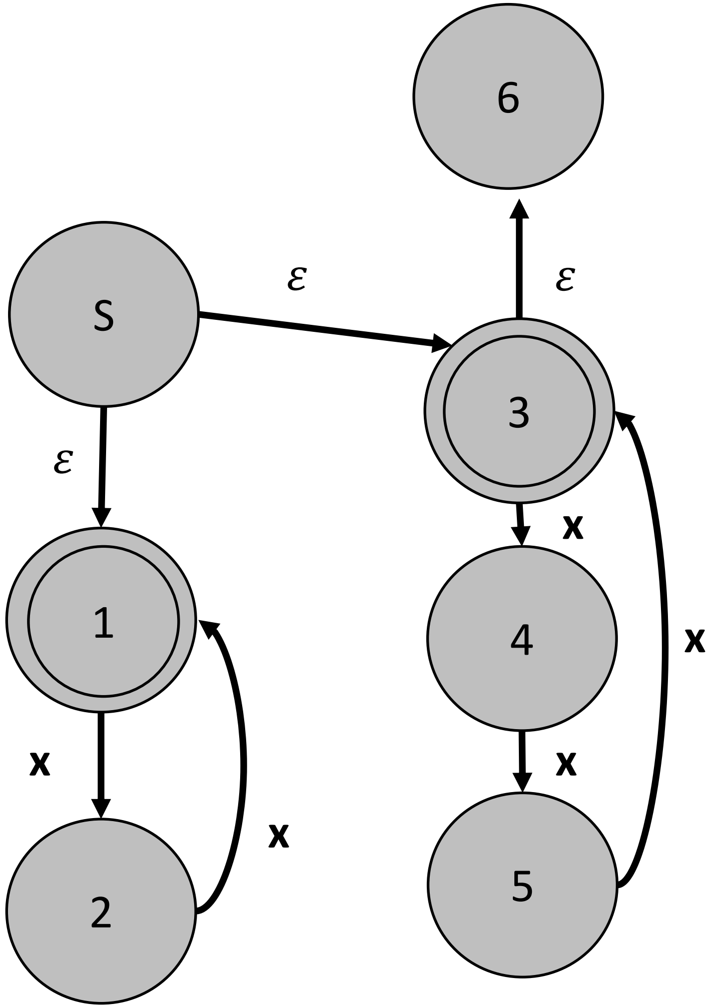
ε-close(6) = {6}



Example, Step IV

Eliminating ε-Transitions

Put s,c → t in δ' if there is a c-edge to t in ε-close(s)



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

ε-close(1) = {1}

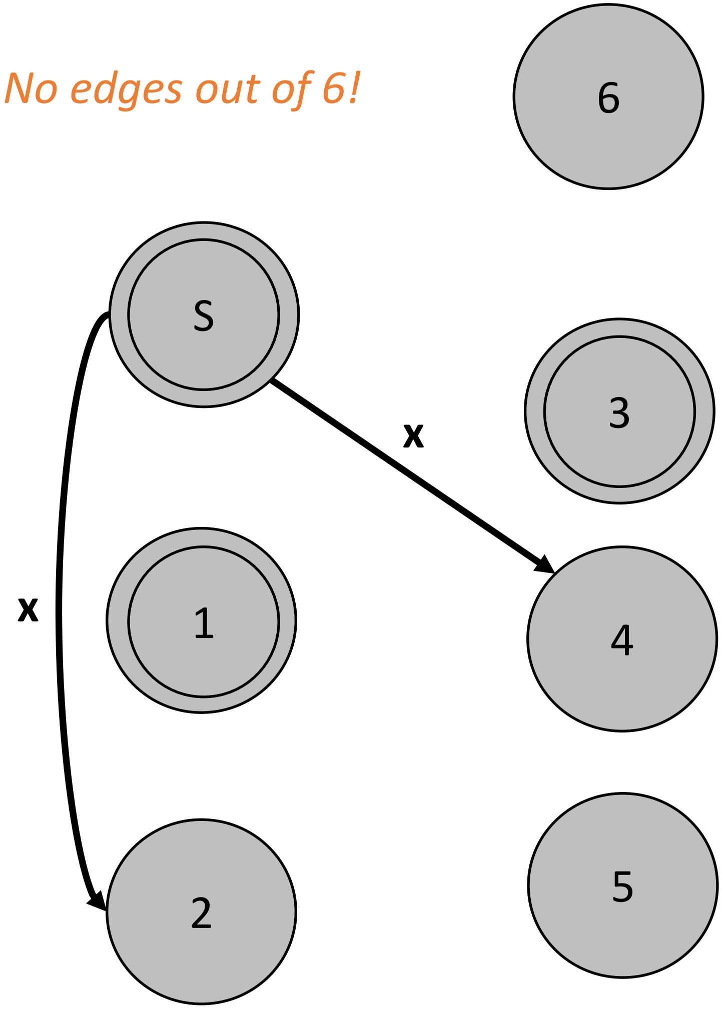
ε-close(2) = {2}

ε-close(4) = {4}

ε-close(5) = {5}

ε-close(6) = {6}

No edges out of 6!

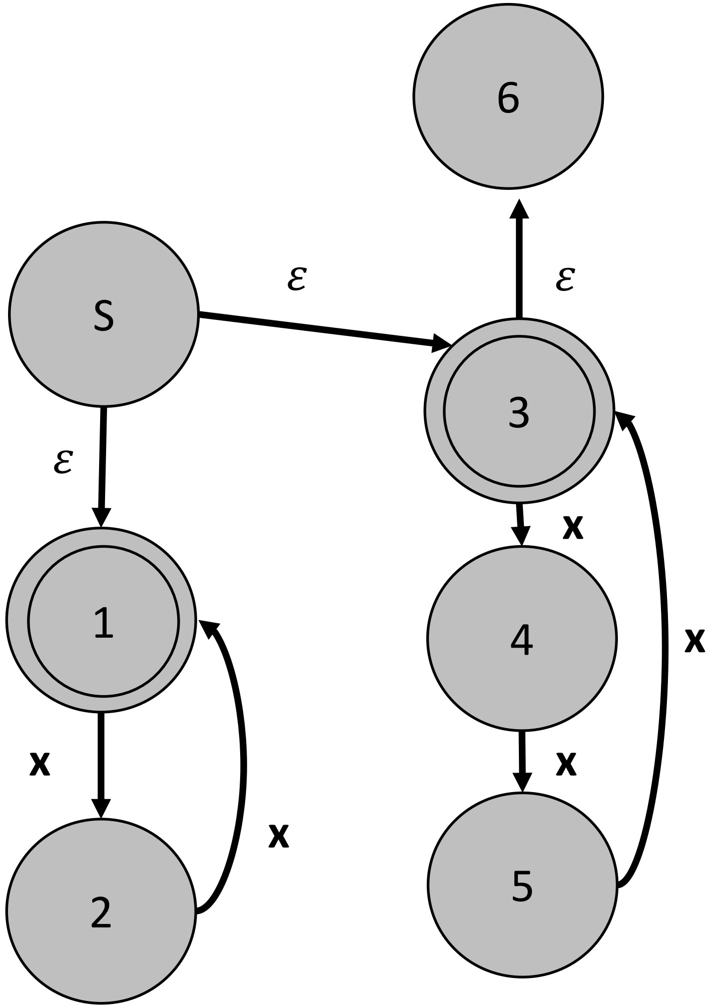


Example, Step IV

Eliminating ε-Transitions

Put s,c → t in δ' if there is a c-edge to t in ε-close(s)

Note: this definition necessarily preserves all original non-ε edges



ε-close(S) = {S,1,3,6}

ε-close(3) = {3, 6}

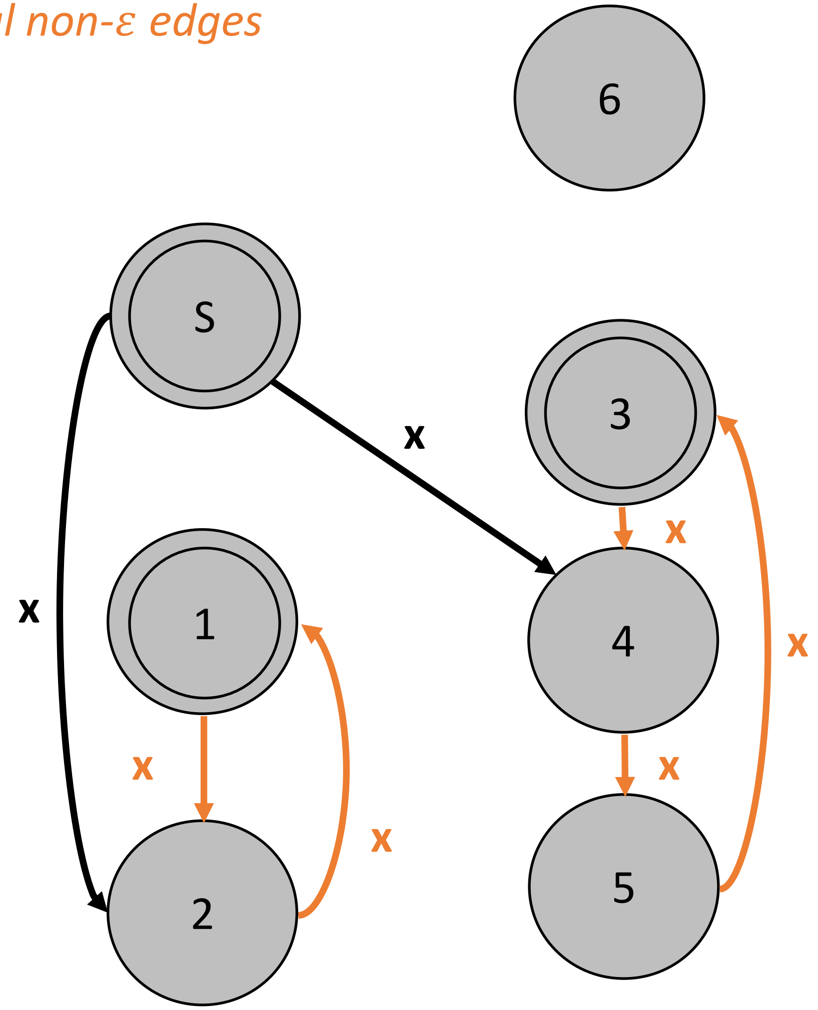
ε-close(1) = {1}

ε-close(2) = {2}

ε-close(4) = {4}

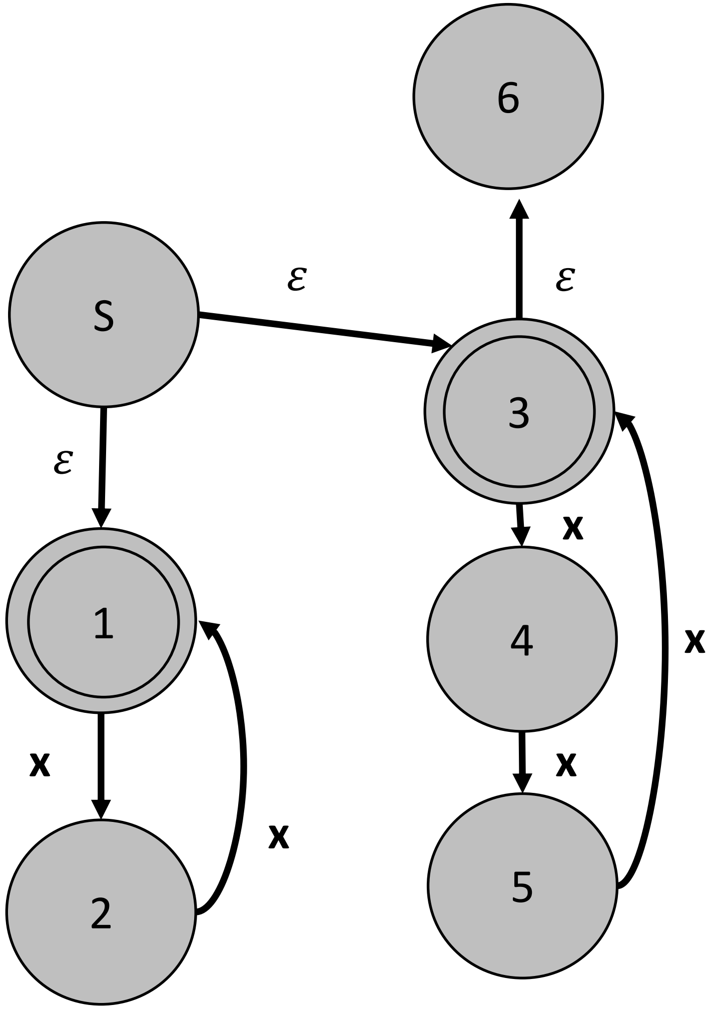
ε-close(5) = {5}

ε-close(6) = {6}



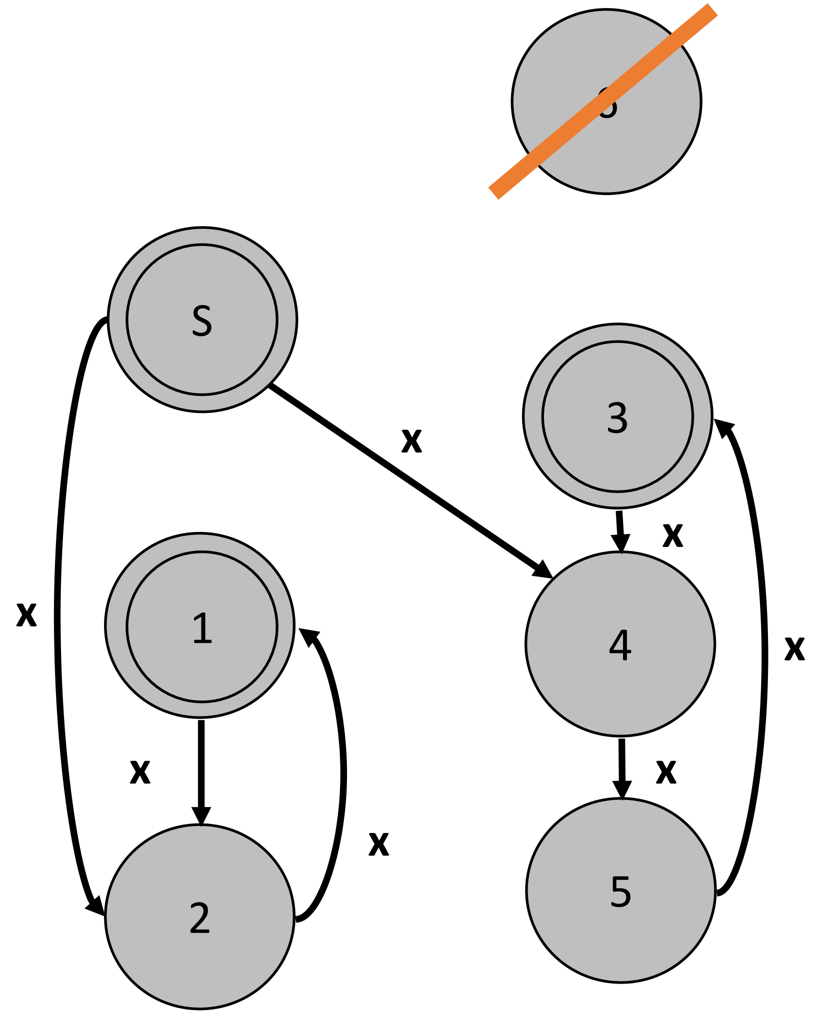
Example, Done!

Eliminating ε-Transitions



=

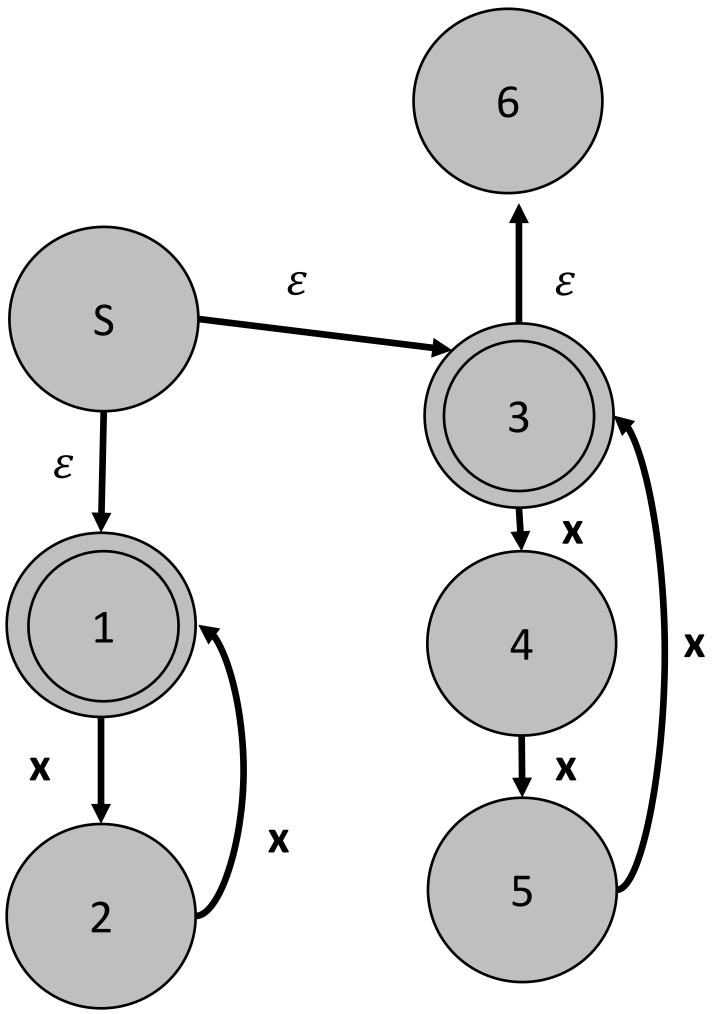
Can also remove unreachable "useless" state

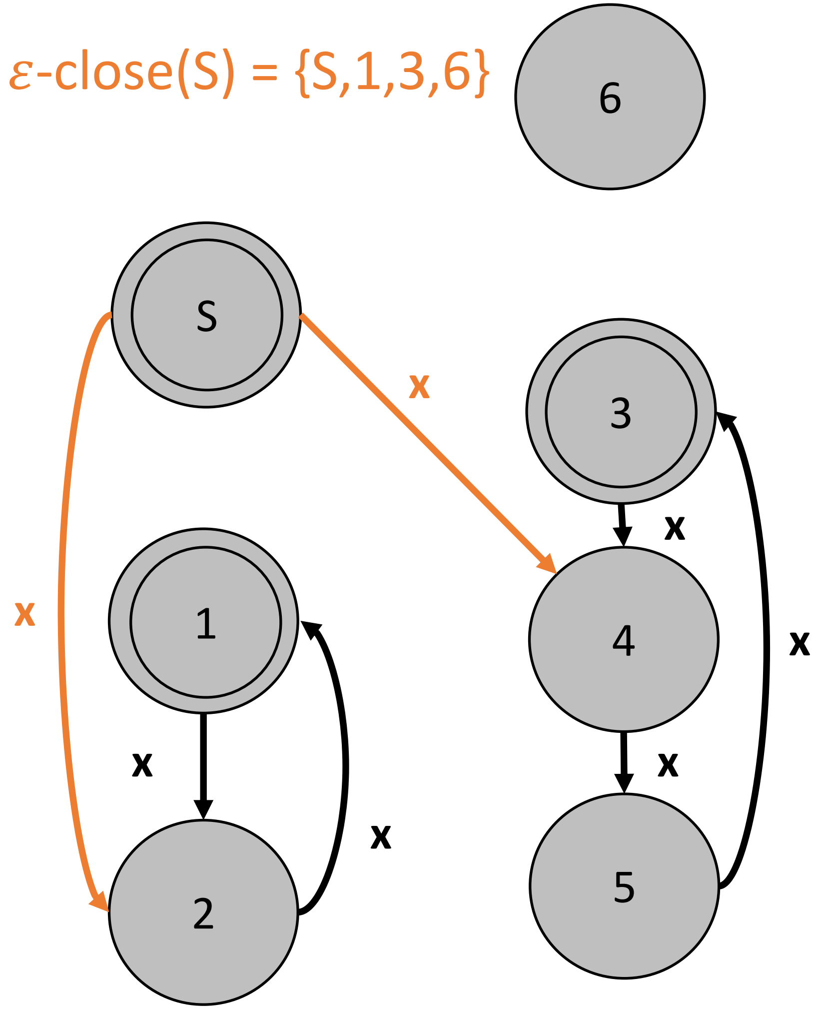


Example, Step IV

Eliminating ε-Transitions

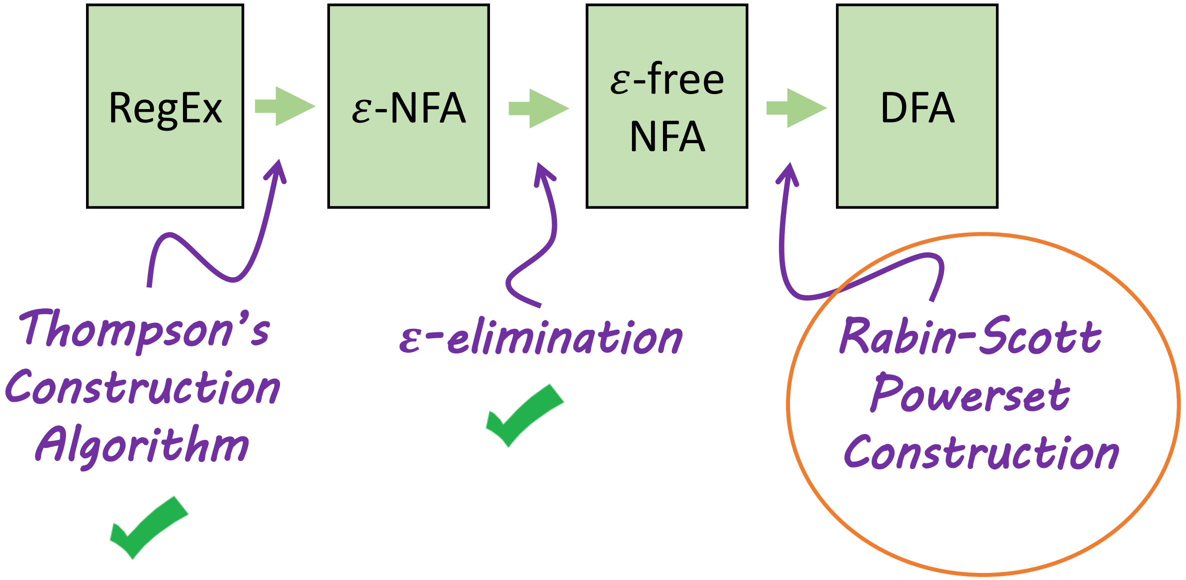
Put s,c → t in δ' if there is a c-edge to t in ε-close(s)





From RegEx to DFA

Lecture 2 – Implementing Scanners



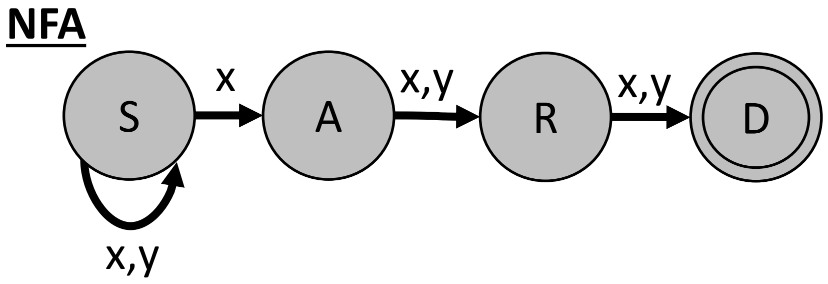
Recall: NFA Matching Procedure

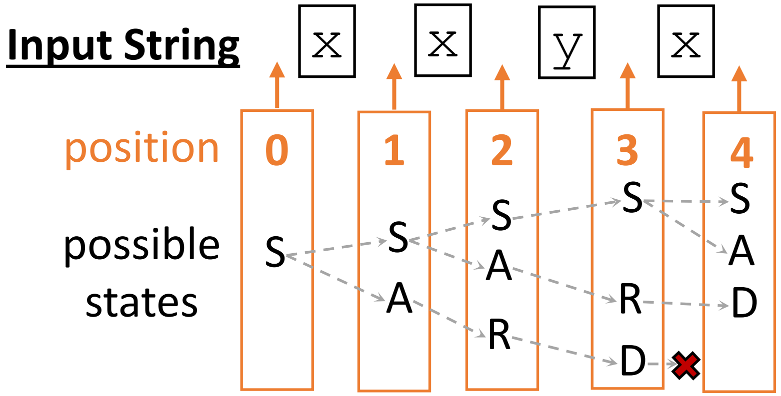
Rabin-Scott Powerset construction

- NFA can "choose" which transition to take
  - Always moves to states that leads to acceptance (if possible)
- Simulate <u>set</u> of states the NFA *could* be in
  - If any state in the ending set is final, string accepted

S,x = {S,A}	R,x = {D}
S,y = {S}	R,y = {D}
A,x = {R}	D,x = {}
A,y = {R}	D,y = {}

Successor <u>set</u> of states

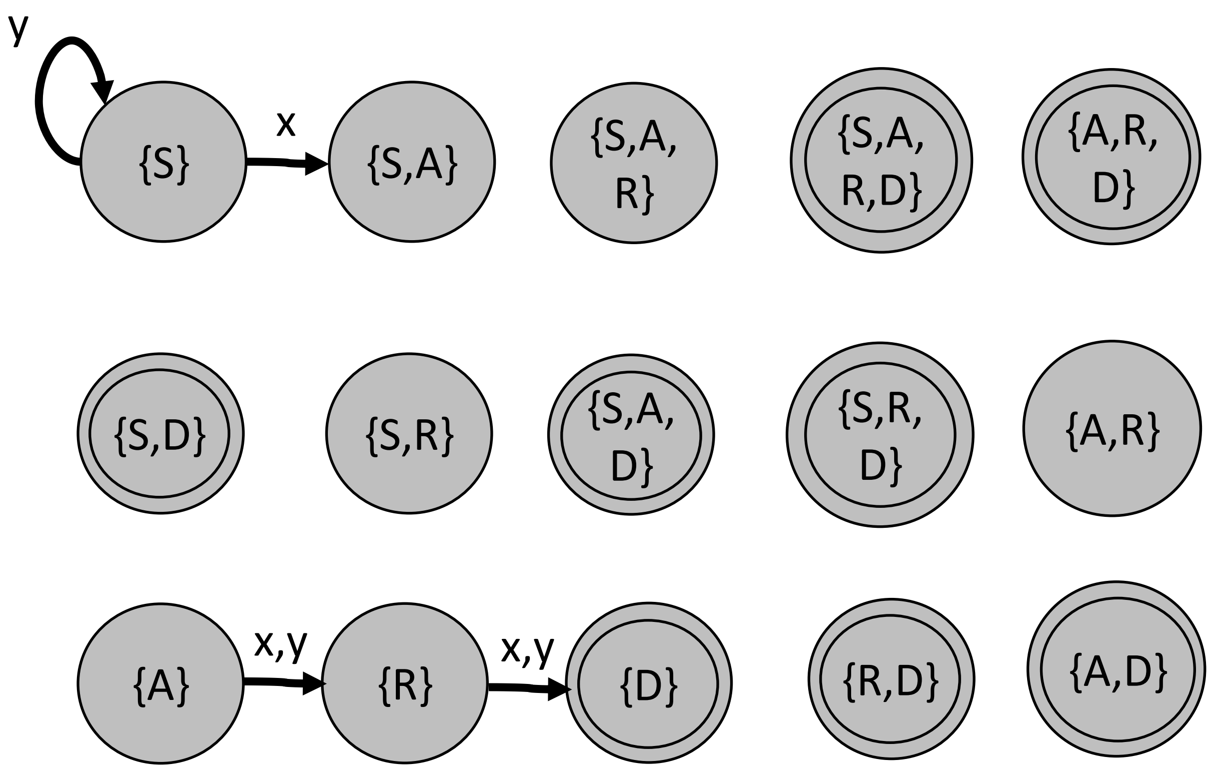




From Successors to Powerset DFA

Rabin-Scott Powerset Construction

S,x = {S,A}	A,x = {R}	R,x = {D}	D,x = {}
S,y = {S}	A,y = {R}	R,y = {D}	D,y = {}



From Successors to Powerset DFA

Rabin-Scott Powerset Construction

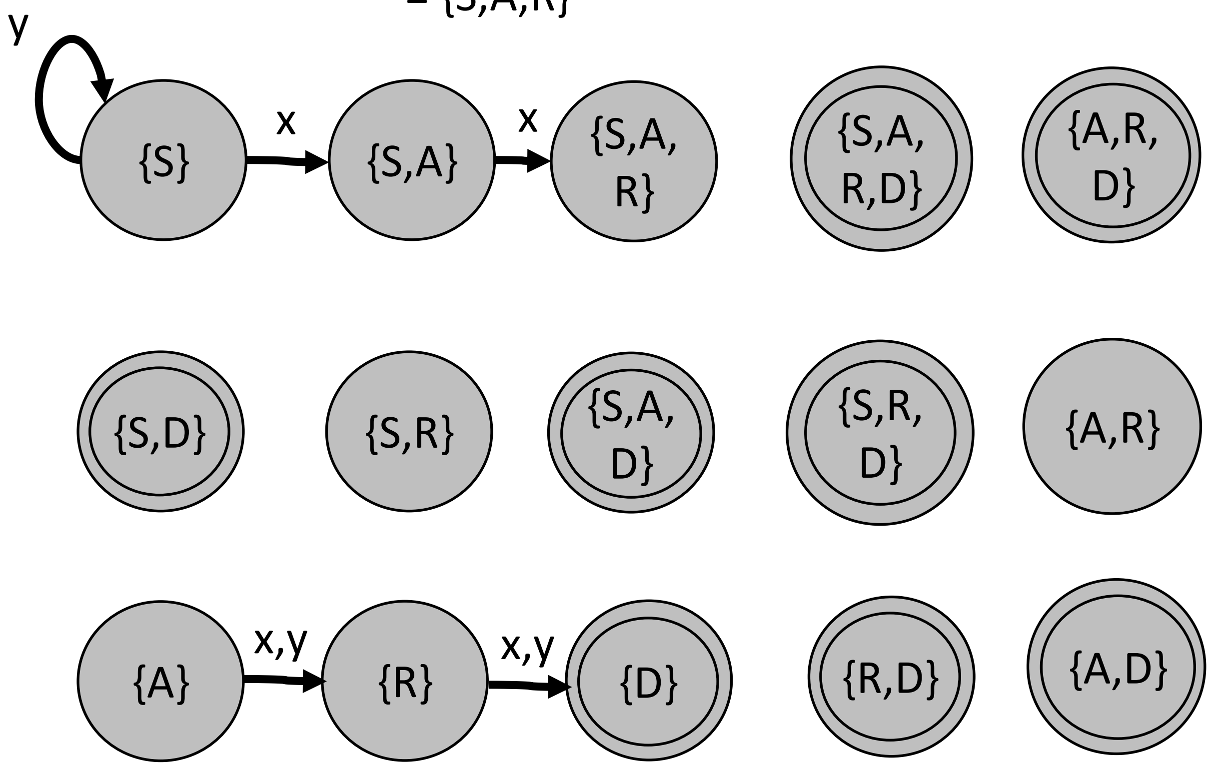
S,x = {S,A} A,x = {R} R,x = {D} D,x = {}

S,y = {S} A,y = {R} R,y = {D} D,y = {}

{S,A}, x = S,x U A,x

= {S,A} U {R}

= {S,A,R}

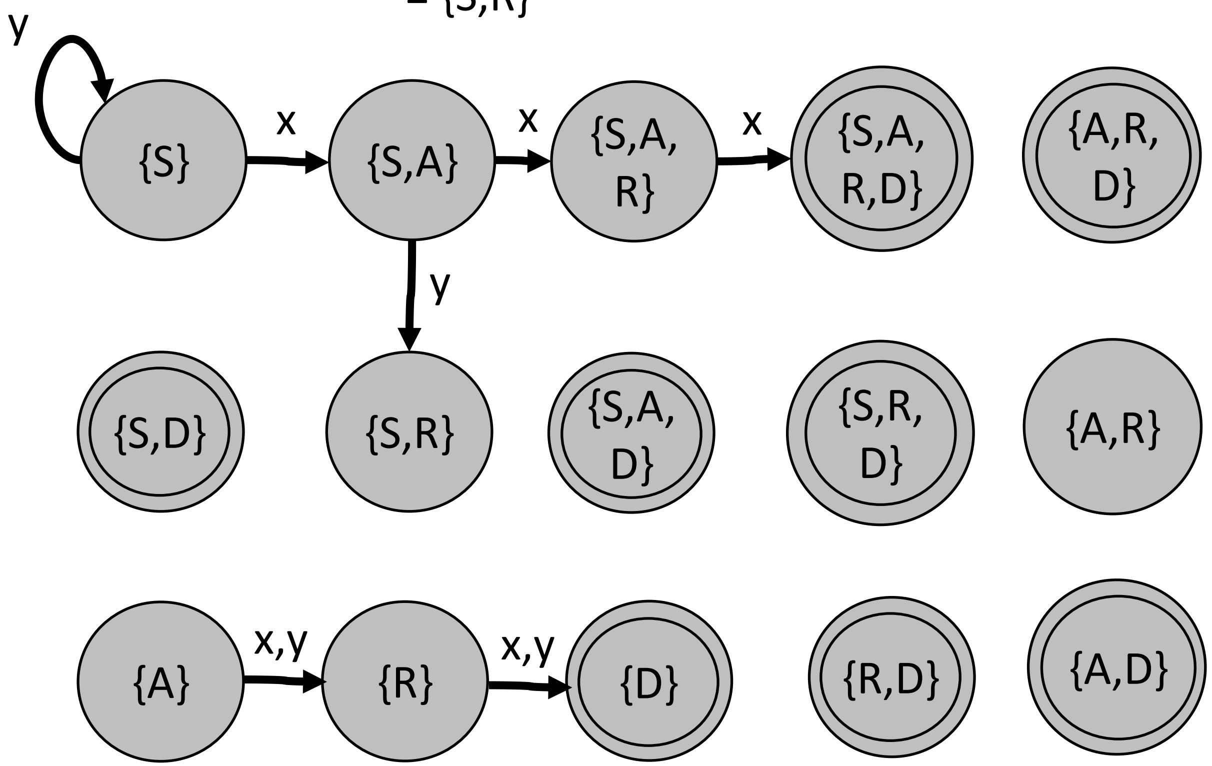


From Successors to Powerset DFA

Rabin-Scott Powerset Construction

S,x = {S,A}	A,x = {R}	R,x = {D}	D,x = {}
S,y = {S}	A,y = {R}	R,y = {D}	D,y = {}

{S,A}, y = S,y U A,y
= {S} U {R}
= {S,R}

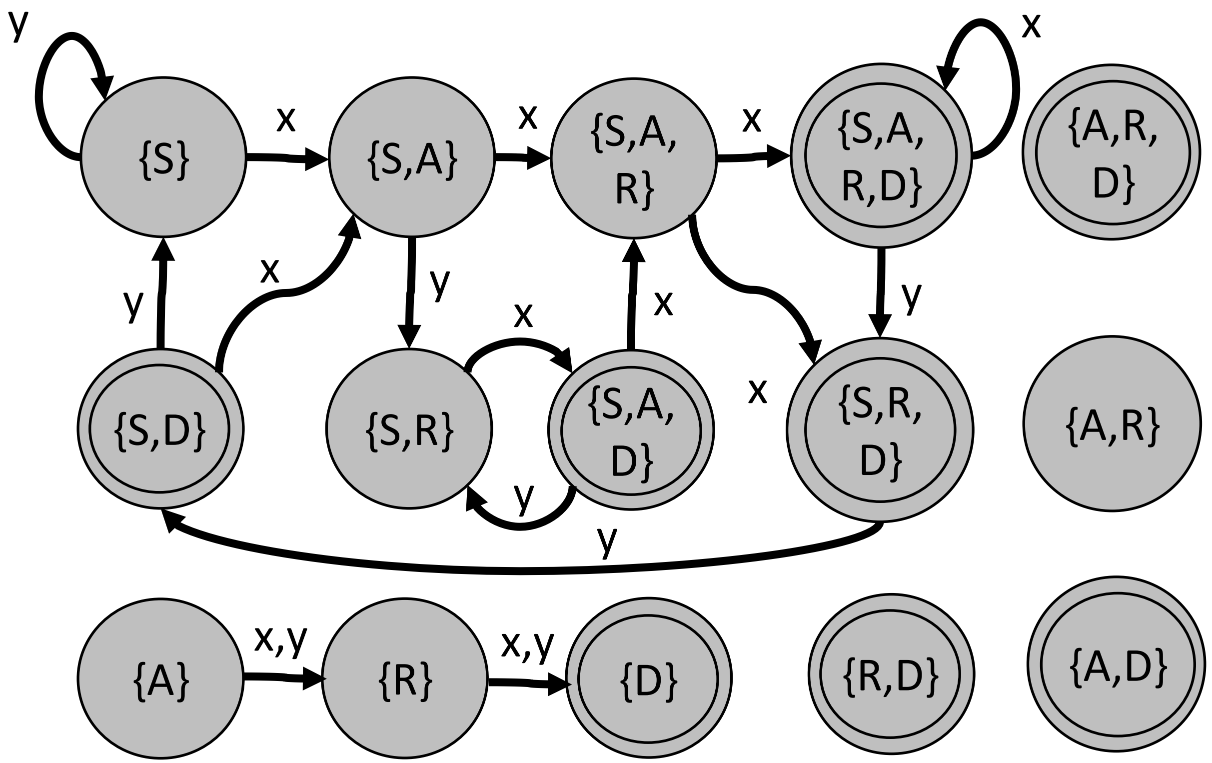


From Successors to Powerset DFA

Rabin-Scott Powerset Construction

S,x = {S,A}   A,x = {R}   R,x = {D}   D,x = {}

S,y = {S}   A,y = {R}   R,y = {D}   D,y = {}

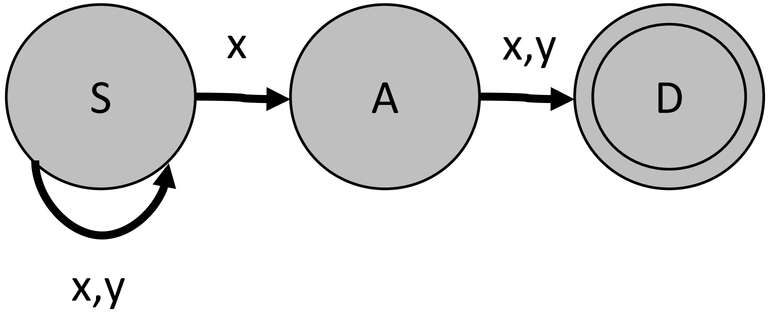


Exponential State Count

Rabin-Scott Powerset Construction

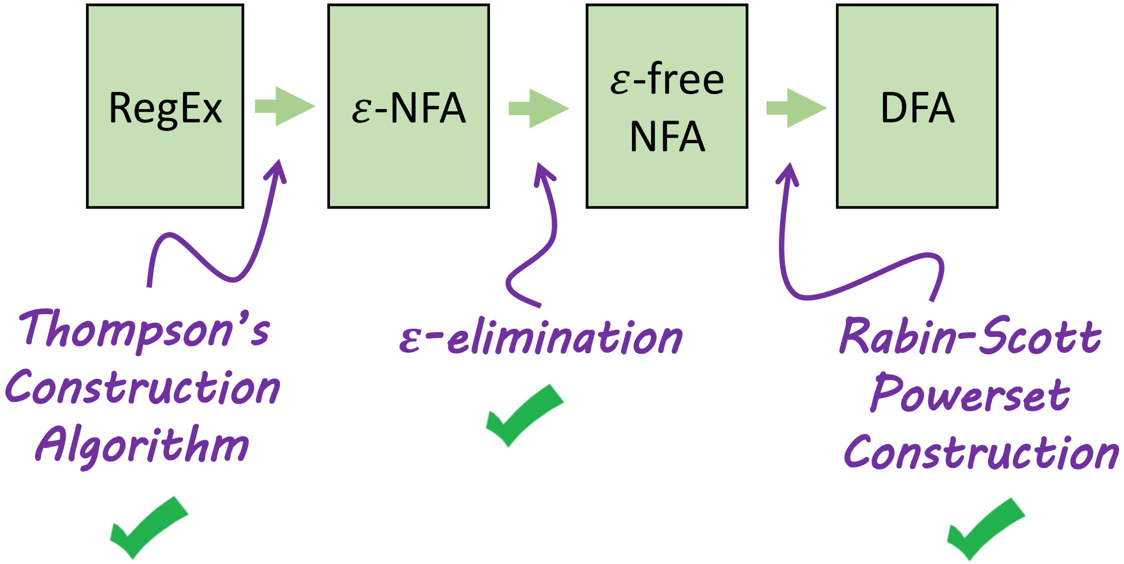
- How may states might the DFA have?
  - $2^{|Q|}$
- Why $2^{|Q|}$?

S	A	D	
0	0	0	{}
0	0	1	{D}
0	1	0	{A}
1	0	0	{S}
0	1	1	{A,D}
1	1	0	{S,A}
1	0	1	{S,D}
1	1	1	{S,A,D}



From RegEx to DFA

Lecture 2 – Implementing Scanners



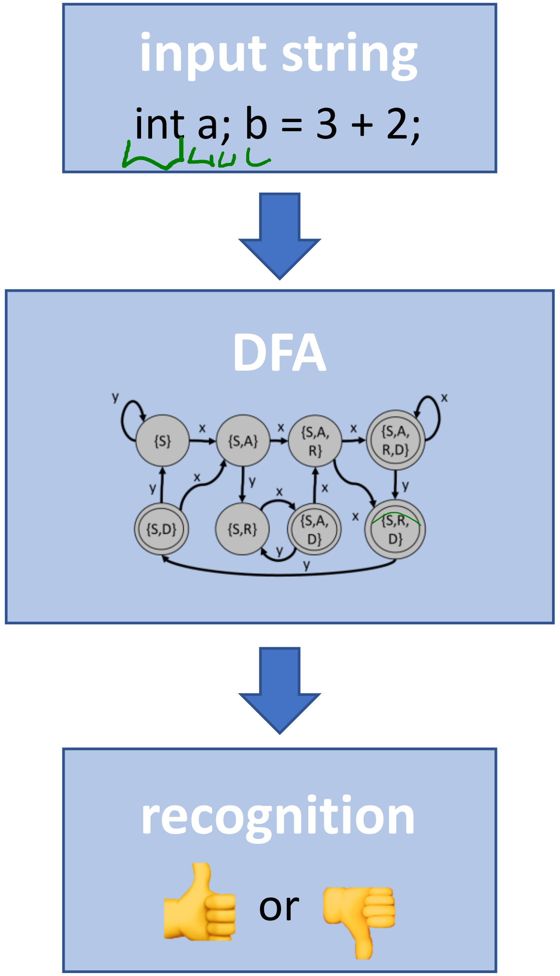
DONE!

… or are we?

DFA ≠ Tokenizer

Limitations

- Finite automata only check for language membership of a string (recognition)
- The Scanner needs to
  - Break the input into many different tokens
  - Know what characters comprise the token

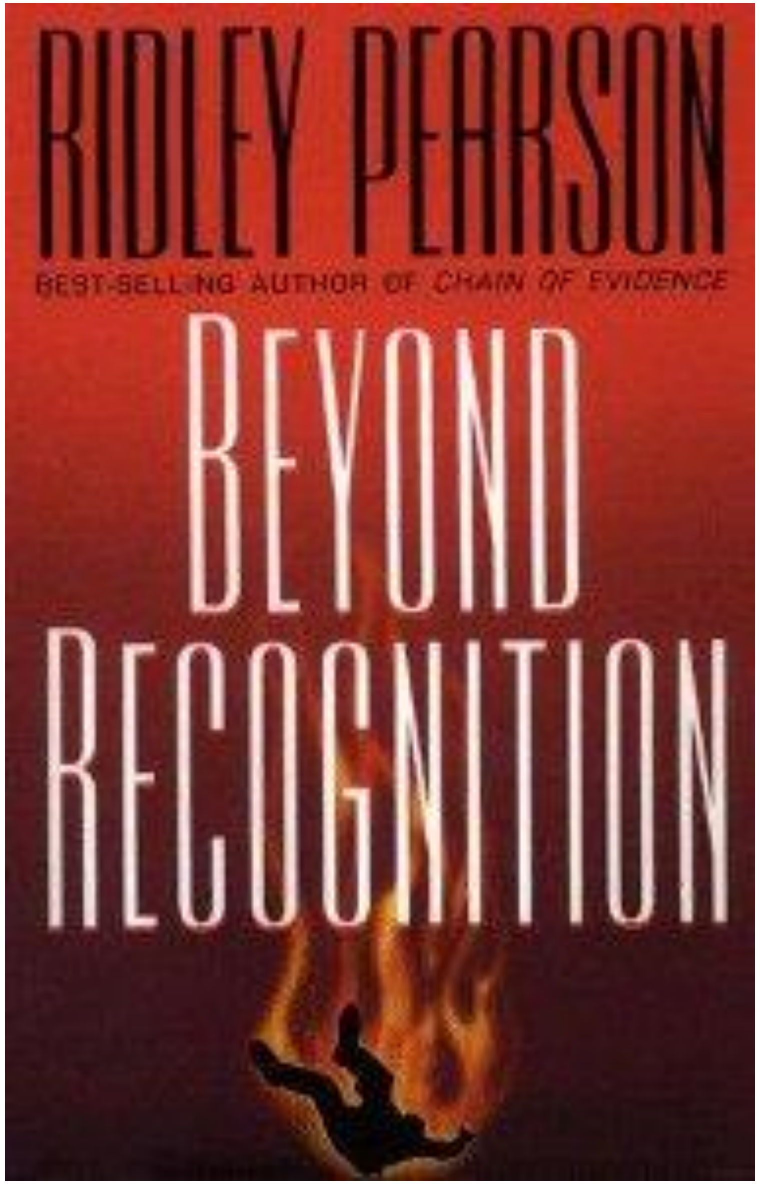


DFA ≠ Tokenizer

Limitations

- Finite automata only check for language membership of a string (recognition)
- The Scanner needs to
  - Break the input into many different tokens
  - Know what characters comprise the token

We need to go… *beyond recognition*



Next Time

Lecture 3 Preview