CS443: Compiler Construction

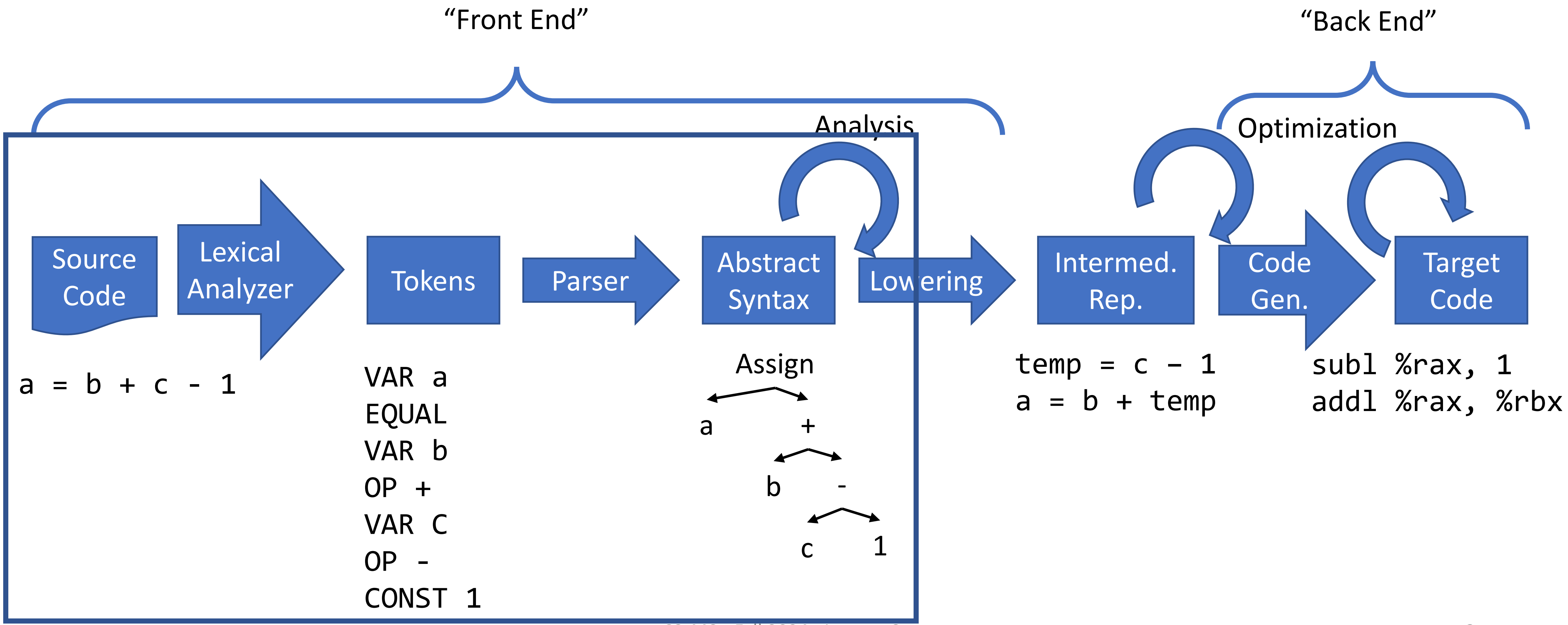
Lecture 2: Lexing, Finite State Machines, Regular Expressions

Announcements

- Project 0 out, due 9/3

- Office hour *locations* swapped this week:
  - Wednesday, 10:30-11:30 SB 218E
  - Thursday, 2-3 Zoom (link in email, I'll also put it on Canvas)

Compilers translate code in phases



Terminology

- *Lexical analysis* “lexing”
- Performed by *lexical analyzer* “lexer”
- Produces stream of *tokens*

Tokens are specified using a *regular* grammar

• Regular expressions R:		
• ε	Empty string	
• abc	Exactly the string abc	Literal
• R_1R_2	R_1 followed by R_2	Concatenation
• $R_1 \mid R_2$	R_1 or R_2	Alternation
• R^*	Zero or more R	Kleene Star
• R^+	One or more R	
• $R?$	Optional R	
• [a-z]	a, b, c, d, …, z	

Regex examples (Alphabet: *a, b*)

Write a regex that recognizes all strings:

• with at least one *a*	(a|b)*a(a|b)*
• where every *a* is immediately followed by a *b*	b*(ab+)*
• beginning and ending in *b*	b|(b(a|b)*b)

Tokens are specified using a *regular* grammar

digit ::= [0-9]

alpha ::= [a-z]

ident ::= alpha (alpha | digit)*

num ::= digit$^+$

ident → IDENT s

num → NUM s

“while” -> WHILE

“+” -> PLUS…

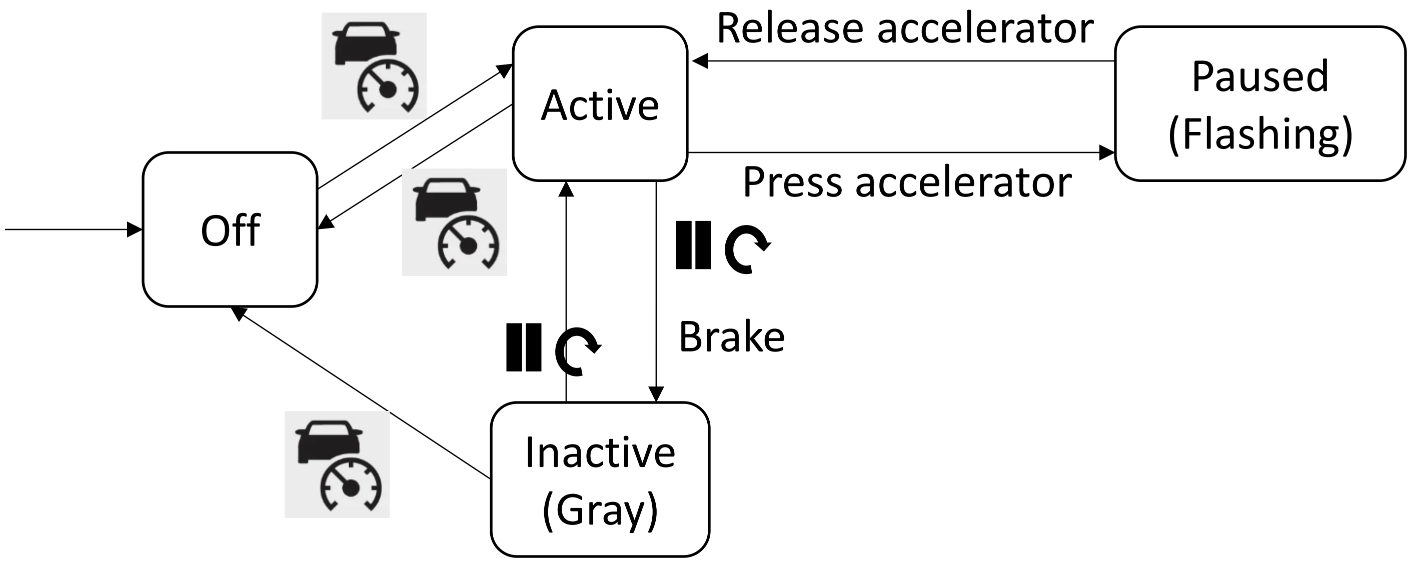
Lexing examples

- while (i < 5) WHILE; LPAREN; IDENT “i”; LT; NUM 5; RPAREN
- while i < 5) WHILE; IDENT “i”; LT; NUM 5; RPAREN
- whole (i < 5) IDENT “whole”; LPAREN; IDENT i; LT; NUM 5; RPAREN

Might be syntax errors during *parsing. Not* errors during lexing.

Regex matching can be done by finite state machines (FSMs)

- Machine can be in one of a *finite number of states*
- Changes state by reading input



Common state machine:
DFA (Deterministic Finite Automaton)

Over an alphabet Σ

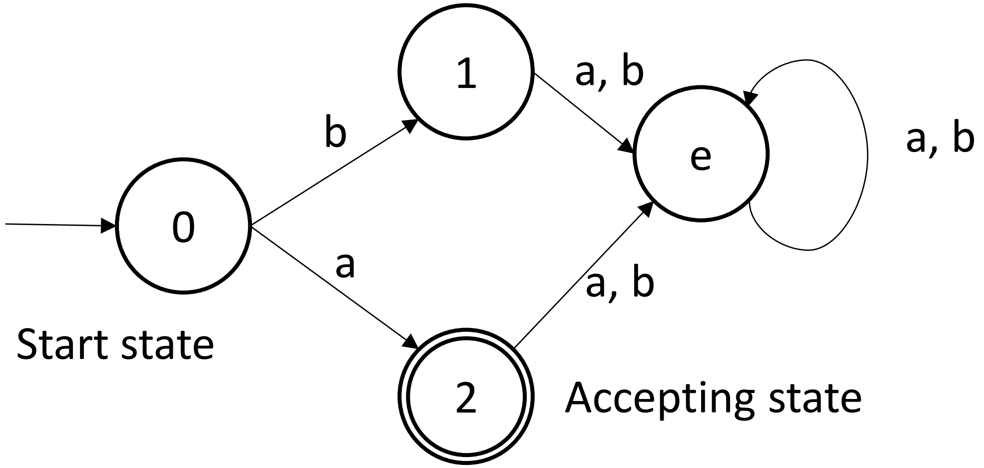
Formal definition: (Q, δ, q_0, F)

- Q: A set of states
- δ: Transition function ($Q \times \Sigma \to Q$)
- q_0 : Start state
- F: Set of *accepting* states

Total and deterministic – exactly one transition for every state, symbol

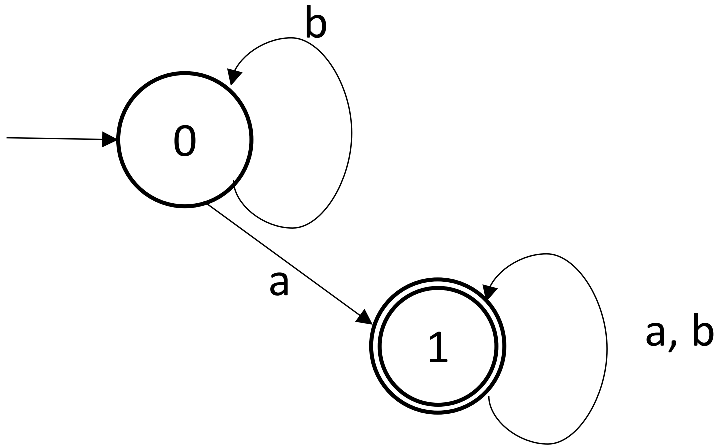
Common state machine:
DFA (Deterministic Finite Automaton)

- ({0, 1, 2, e}, , 0, 2)

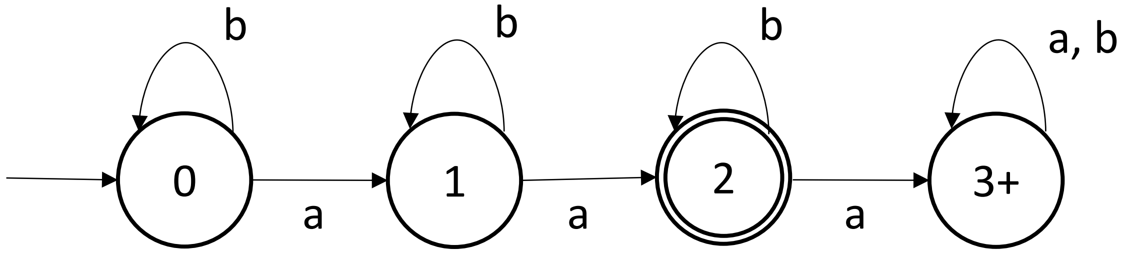


Accepts exactly “a”

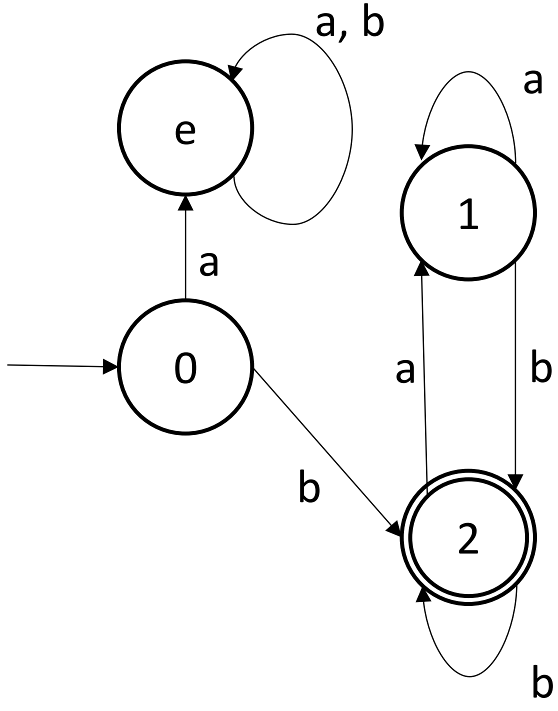
DFA for “at least one a”



DFA for "exactly two *a*s" (Just count the number until we get to 3)

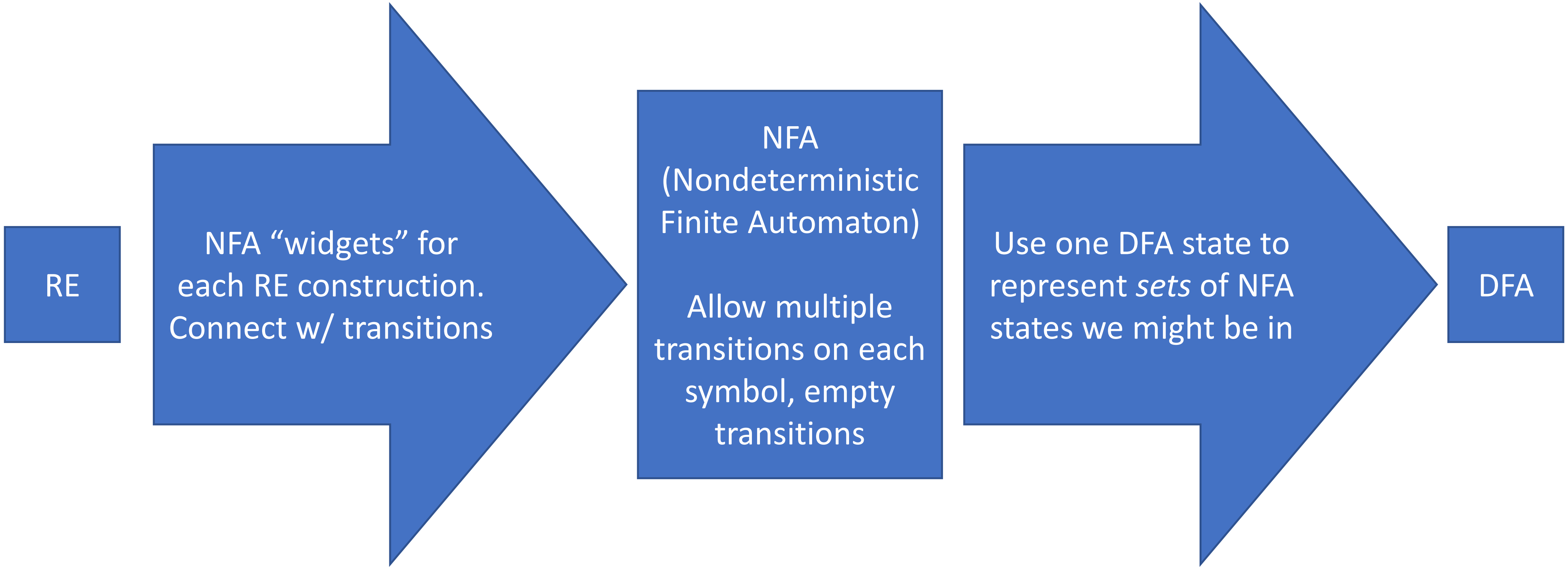


DFA for "beginning and ending with *b*"



Can convert regexes to DFAs-but we don't do it directly

- Full algorithm in Appel, PDB. General idea:



NFAs

Formal definition: (Q, δ, q_0, F)

- Q: A set of states
- δ: Transition function ($Q \times \Sigma \cup \{\varepsilon\}$-> Q)
- q_0 : Start state
- F: Set of *accepting* states

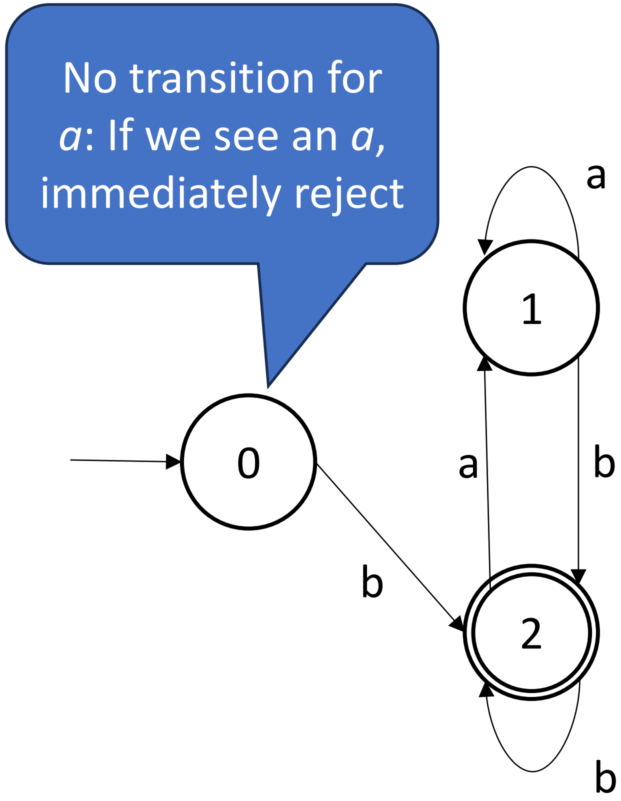
Silent transitions

~~Total and deterministic – exactly one transition for every state, symbol~~

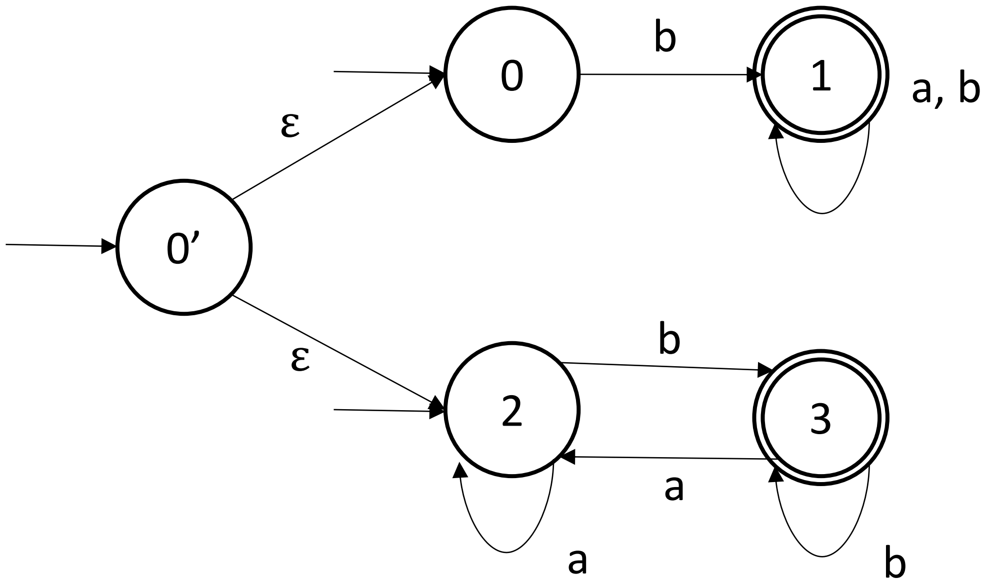
Zero or more transitions

Accept if there's a way to reach a final state ("try all paths at once")

NFA for "beginning and ending with *b*"

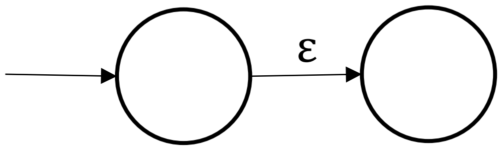


NFA for “beginning **or** ending with *b*”

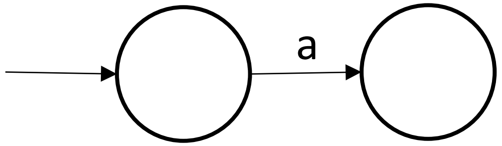


Convert Regexes to NFAs with “widgets”

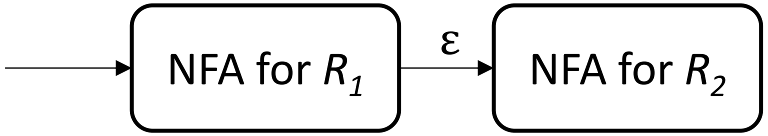
- ε



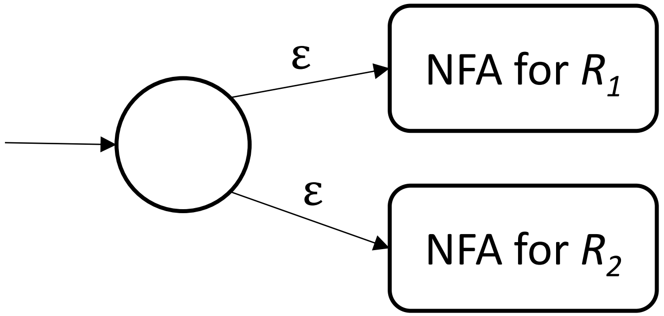
- a



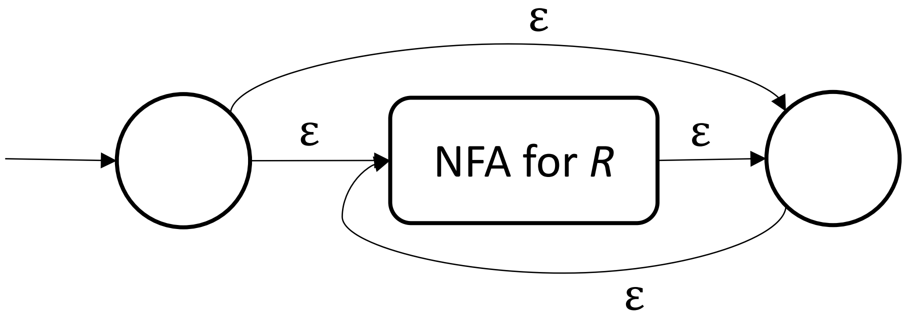
- R_1R_2



- $R_1 \mid R_2$

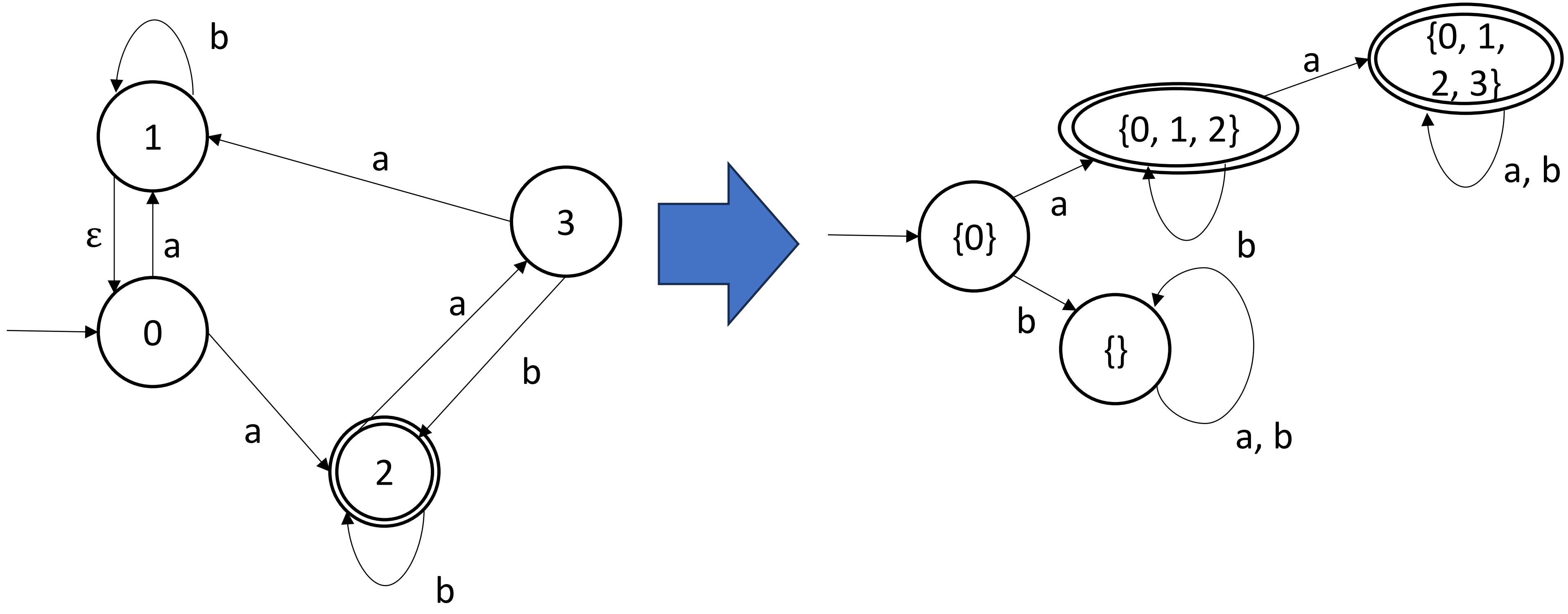


- R^*



Convert NFAs to DFAs with “subset construction”

- DFA states = *sets of NFA states*
- Final DFA states = contain >= 1 final NFA state



DFAs/regexes can’t do everything

- Make a Regex/DFA for “open and close parens match”

- Impossible!
- Proof sketch: Need different states for (, ((, (((, …
- “DFAs can’t count”