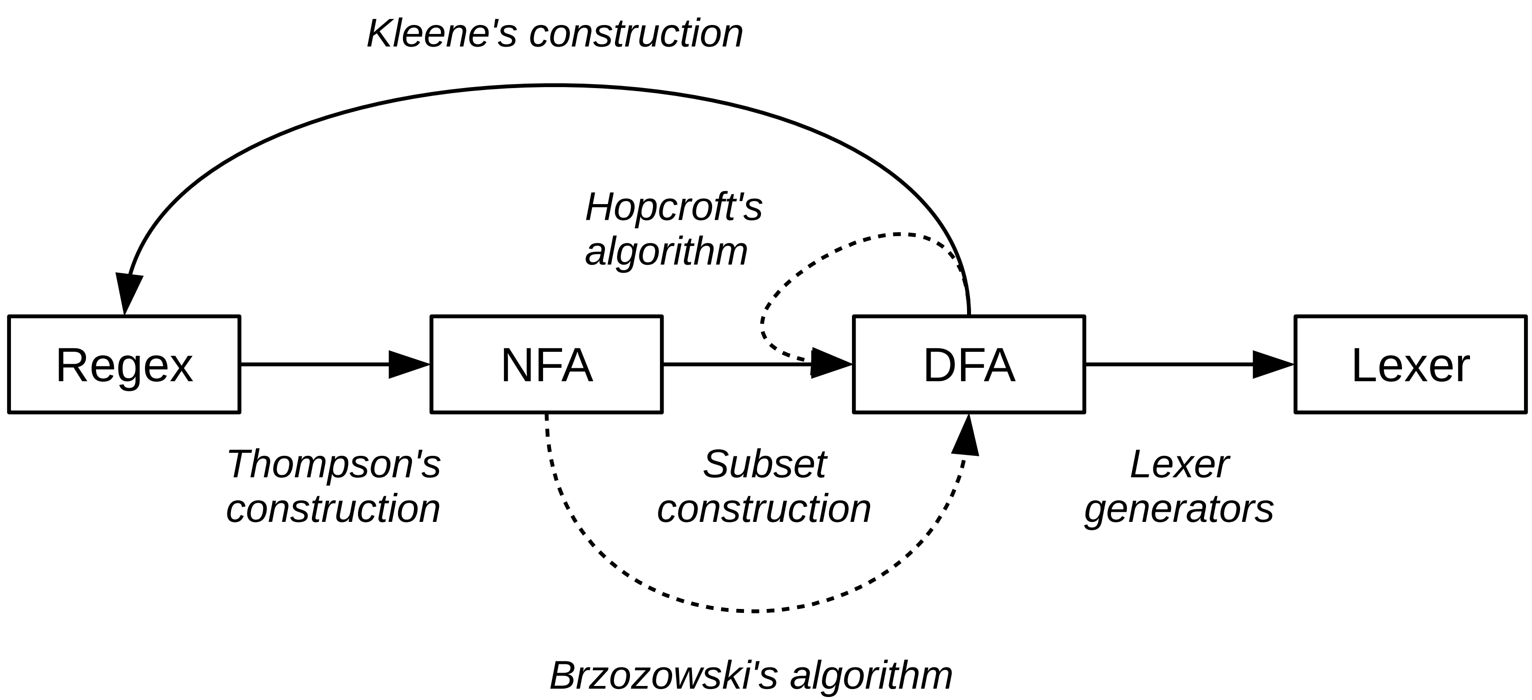
CS 432
Fall 2016

Mike Lam, Professor

Finite Automata Conversions and Lexing

Finite Automata

- Finite automata transitions:

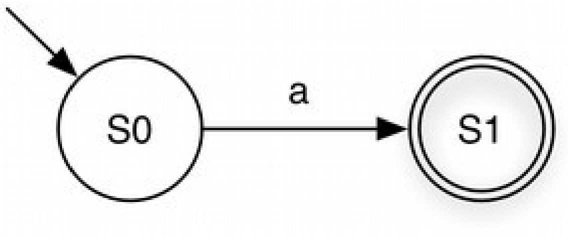


(dashed lines indicate transitions to a minimized DFA)

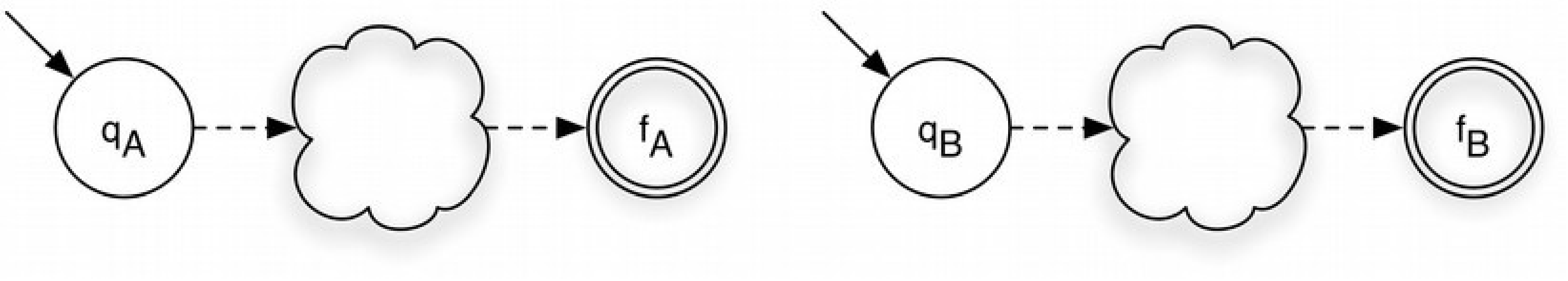
Finite Automata

- RE to NFA: Thompson's construction
  - Core insight: **inductively** build up NFA using “templates”
- NFA to DFA: Subset construction
  - Core insight: DFA node = **subset** of NFA nodes
  - Core concept: use **null closure** to calculate subsets
- DFA minimization: Hopcroft’s algorithm
  - Core insight: create **partitions**, then keep splitting

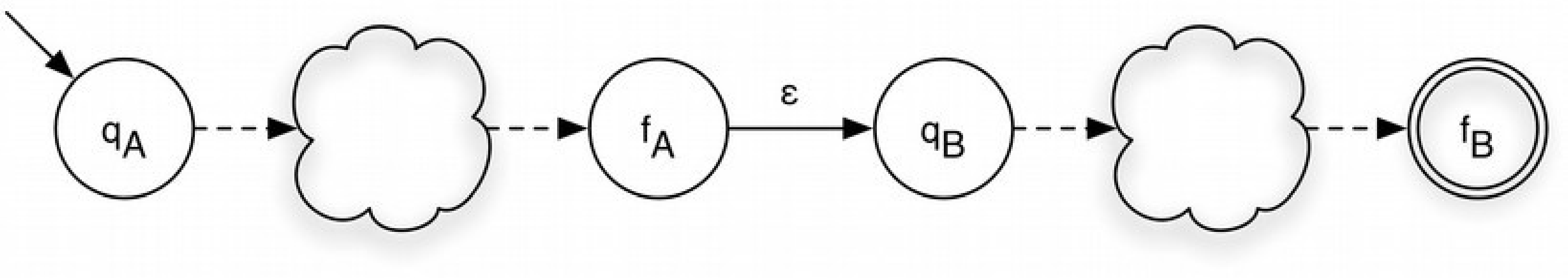
Thompson's: Base case



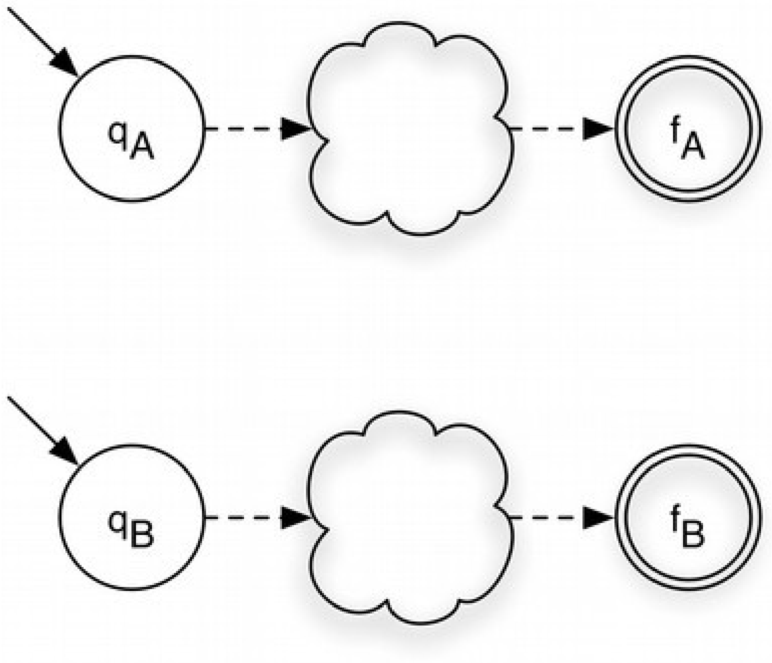
Thompson's: Concatenation



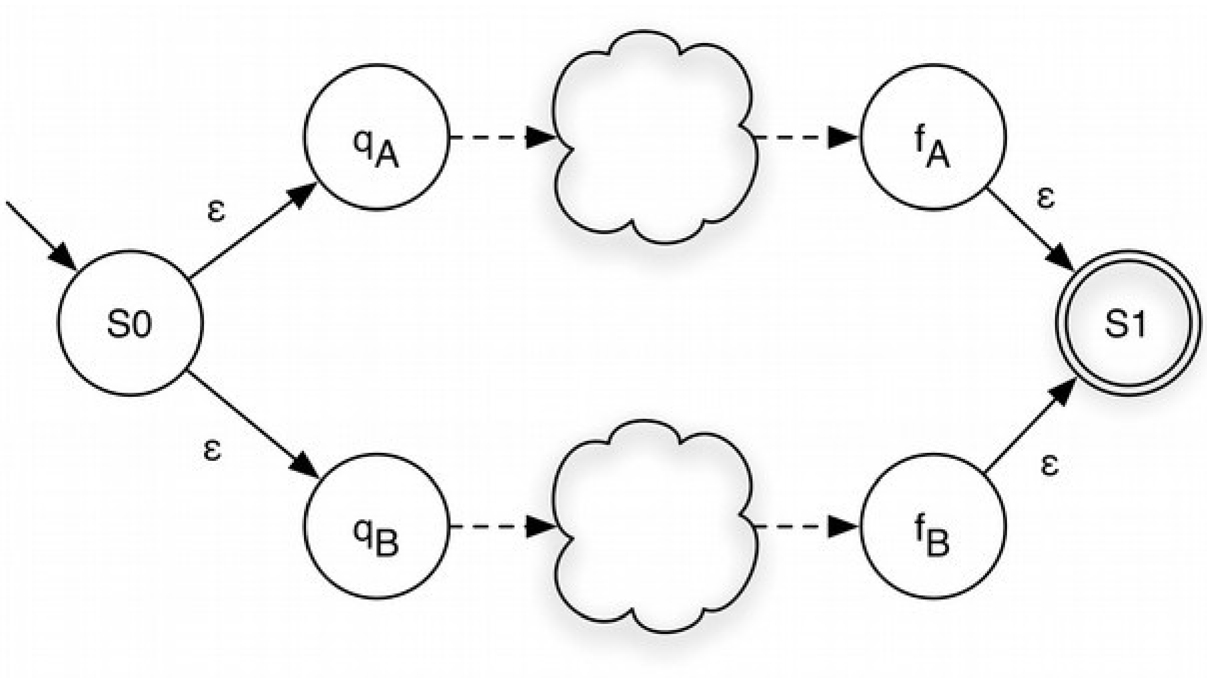
Thompson's: Concatenation



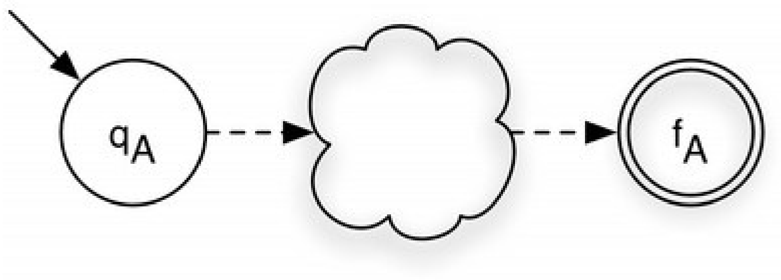
Thompson's: Union



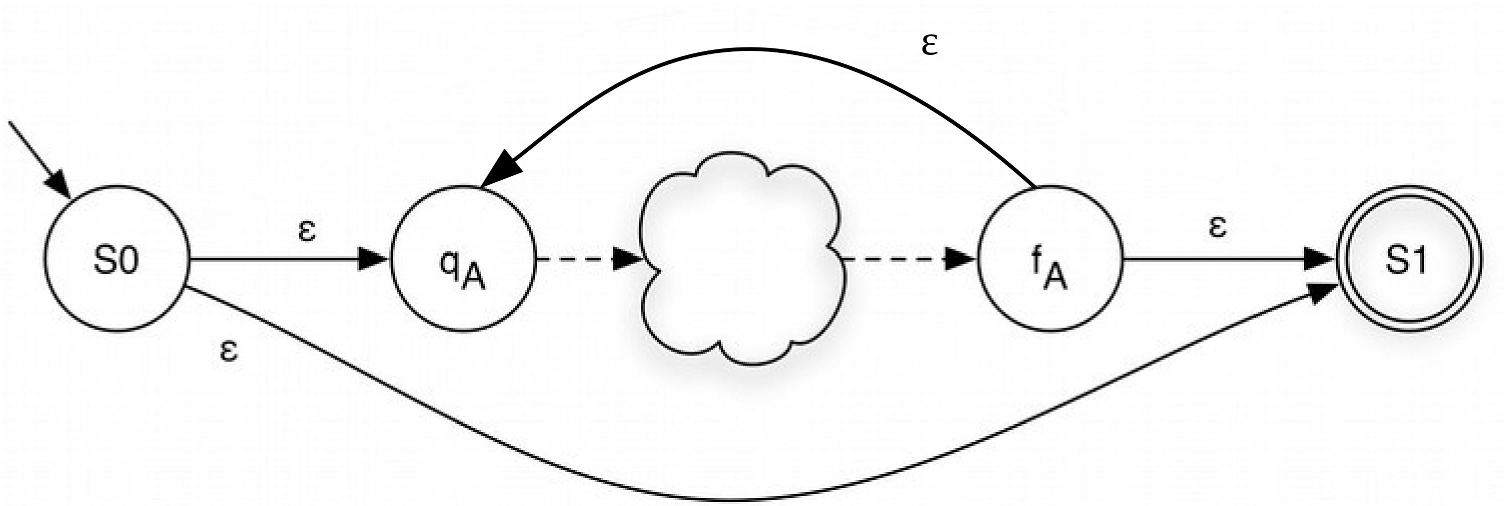
Thompson's: Union



Thompson's: Closure

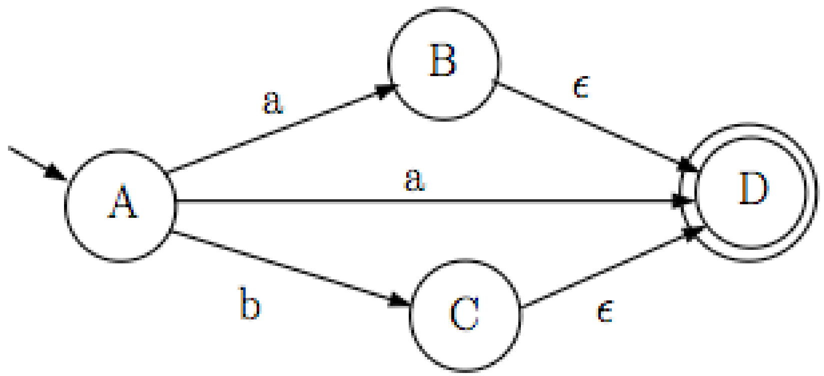


Thompson's: Closure



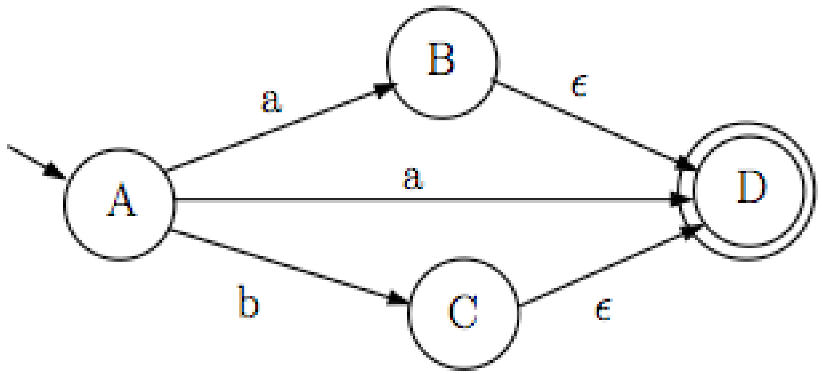
Subset construction

- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - i.e., where can we go “for free?”
  - Simulates running all possible paths through the NFA

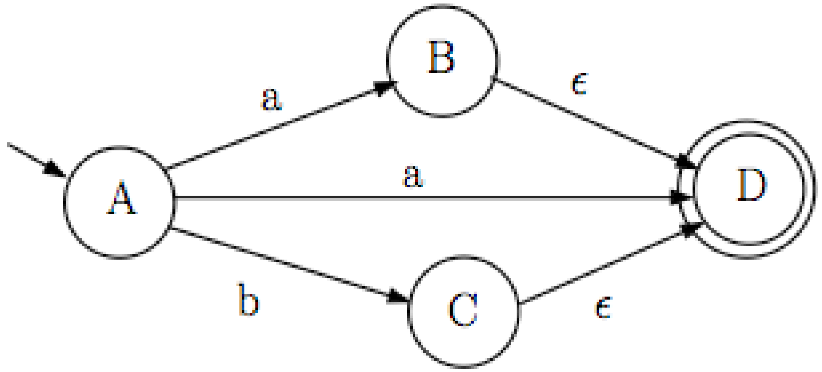


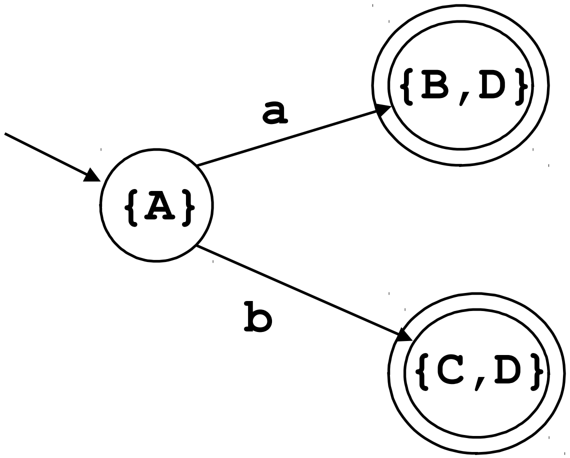
Null closure of A = { A }
Null closure of B = { B, D }
Null closure of C = { C, D }
Null closure of D = { D }

Subset Example

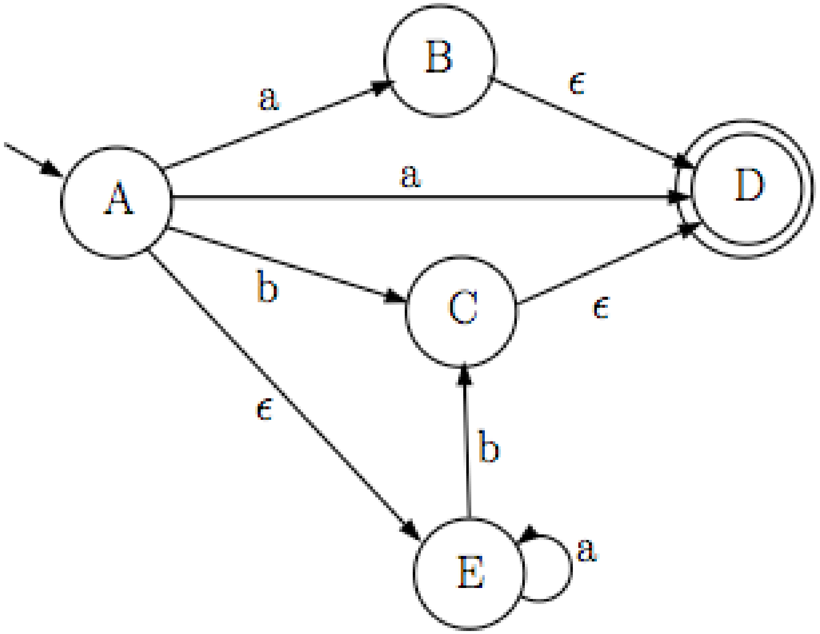


Subset Example

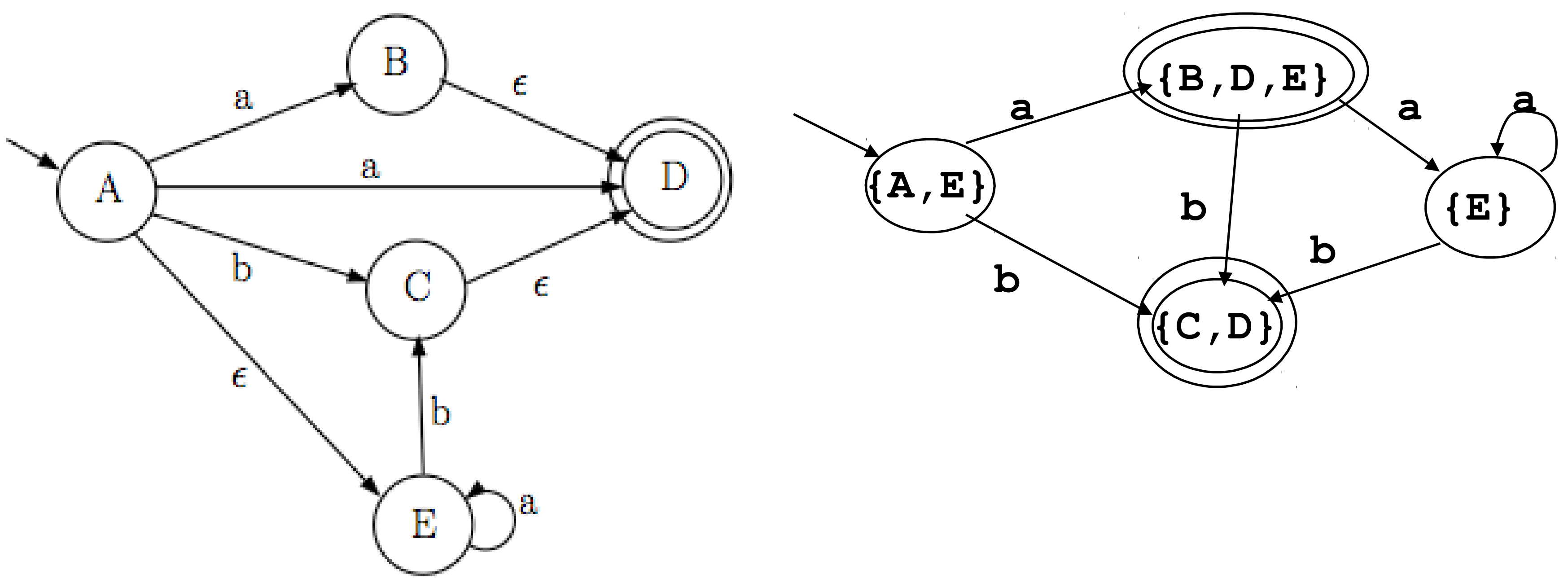




Subset Example

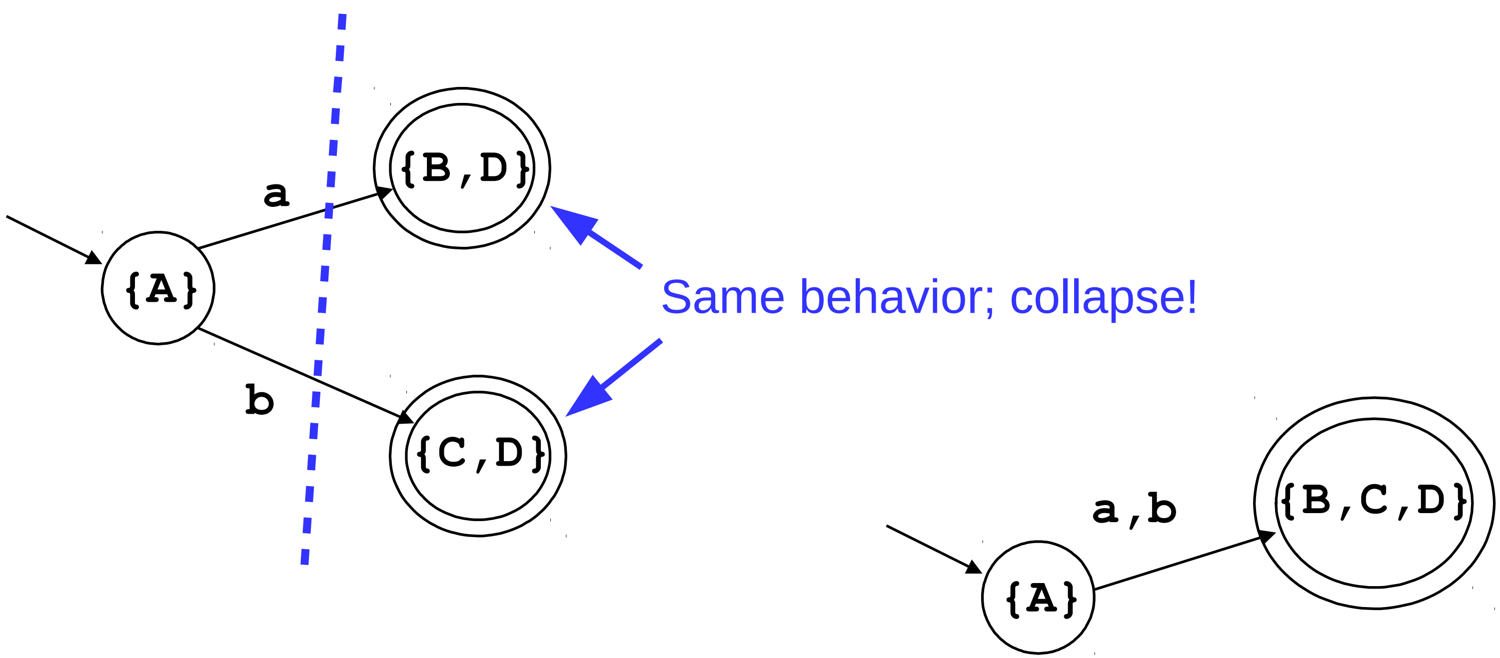


Subset Example



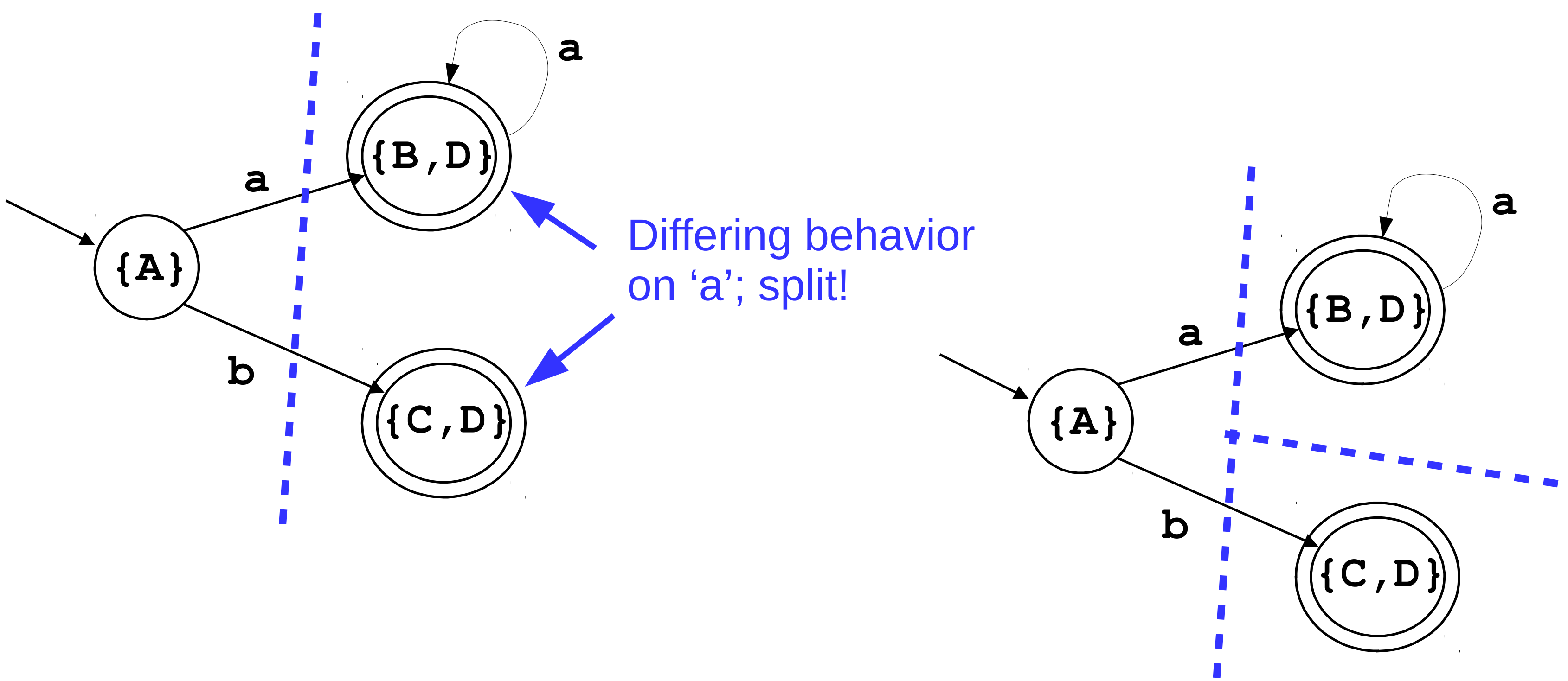
Hopcroft’s DFA Minimization

- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with differing behaviors
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn’t
- When done, collapse partitions to a single state



Hopcroft’s DFA Minimization

- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with differing behaviors
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn’t
- When done, collapse partitions to a single state



Discussion Questions

- How long does it take to...
  - Build an NFA?
  - Run an NFA?
  - Build a DFA?
  - Run a DFA?

Efficiency Concerns

- Thompson's construction
  - At most two new states and four transitions per regex character
  - Thus, a linear space increase with respect to the # of regex characters
  - Constant # of operations per increase means linear time as well
- NFA execution
  - Proportional to both NFA size and input string size
  - Must track multiple simultaneous “current” states
- Subset construction
  - Potential exponential state space explosion
  - A *n*-state NFA could require up to 2^n DFA states
  - However, this rarely happens in practice
- DFAs execution
  - Proportional to input string size only (only track a single “current” state)

NFA/DFA complexity

- NFAs build quicker (linear) but run slower
  - Better if you will only run the FA a few times
- DFAs build slower (worst case exponential) but run faster
  - Better if you will run the FA many times

	NFA	**DFA**
Build time	$O(m)$	$O(2^m)$
Run time	$O(m \times n)$	$O(n)$

m = length of regular expression
n = length of input string

Lexers

- Auto-generated
  - Table-driven: generic scanner, auto-generated tables
  - Direct-coded: hard-code the tables into the scanner
  - Common tools: lex/flex and similar
- Hand-coded
  - Better I/O performance (i.e., buffering)
  - More efficient interfacing w/ other phases

Handling Keywords

- Issue: keywords are identifiers
- Option 1: Embed into NFA/DFA
  - Separate regex for keywords
  - Easier/faster for generated scanners
- Option 2: Use lookup table
  - Scan as identifier then check for a keyword
  - Easier for hand-coded scanners