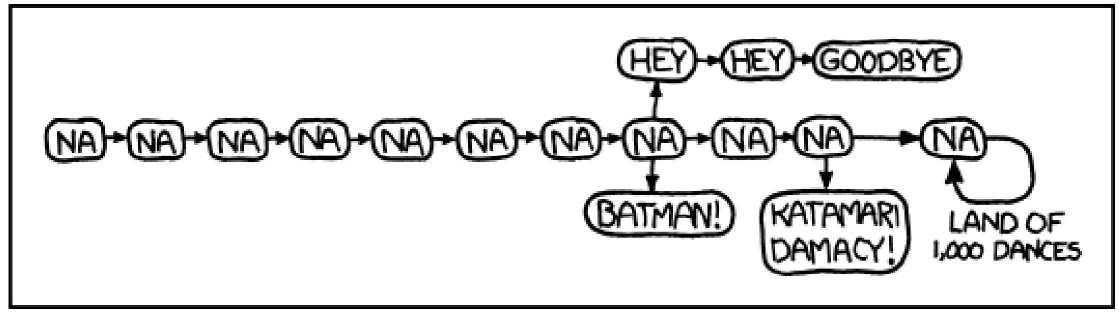
CS 432
Fall 2017

Mike Lam, Professor



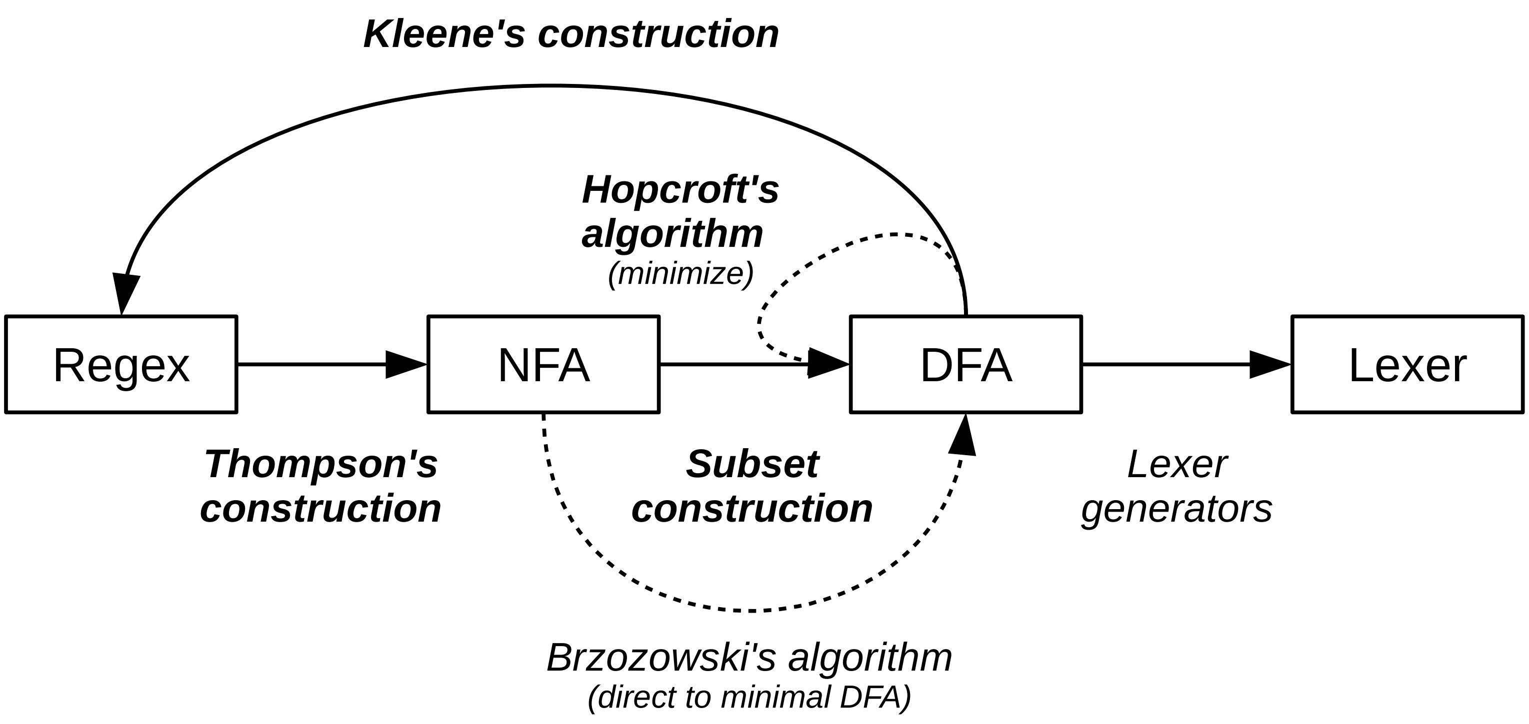
Finite Automata Conversions and Lexing

Finite Automata

- Key result: all of the following have the same expressive power (i.e., they all describe *regular* languages):
  - Regular expressions (REs)
  - Non-deterministic finite automata (NFAs)
  - Deterministic finite automata (DFAs)
- Proof by construction
  - An algorithm exists to convert any RE to an NFA
  - An algorithm exists to convert any NFA to a DFA
  - An algorithm exists to convert any DFA to an RE
  - For every regular language, there exists a minimal DFA
    - Has the fewest number of states of all DFAs equivalent to RE

Finite Automata

- Finite automata transitions:



(dashed lines indicate transitions to a minimized DFA)

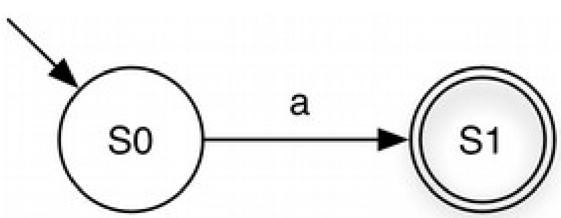
Finite Automata Conversions

- RE to NFA: Thompson's construction
  - Core insight: **inductively** build up NFA using “templates”
  - Core concept: use **null transitions** to build NFA quickly
- NFA to DFA: Subset construction
  - Core insight: DFA nodes represent **subsets** of NFA nodes
  - Core concept: use **null closure** to calculate subsets
- DFA minimization: Hopcroft’s algorithm
  - Core insight: create **partitions**, then keep splitting
- DFA to RE: Kleene's construction
  - Core insight: repeatedly eliminate states by **combining** regexes

Thompson's Construction

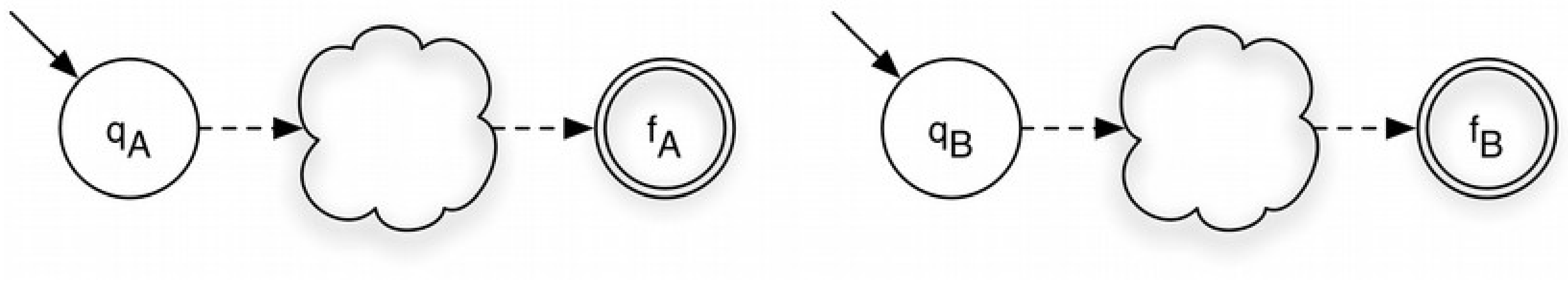
- Basic idea: create NFA inductively, bottom-up
  - Base case:
    - Start with individual alphabet symbols (see below)
  - Inductive case:
    - Combine by adding new states and null/epsilon transitions
    - **Templates** for the three basic operations
  - Invariant:
    - The NFA always has exactly one start state and one accepting state

a



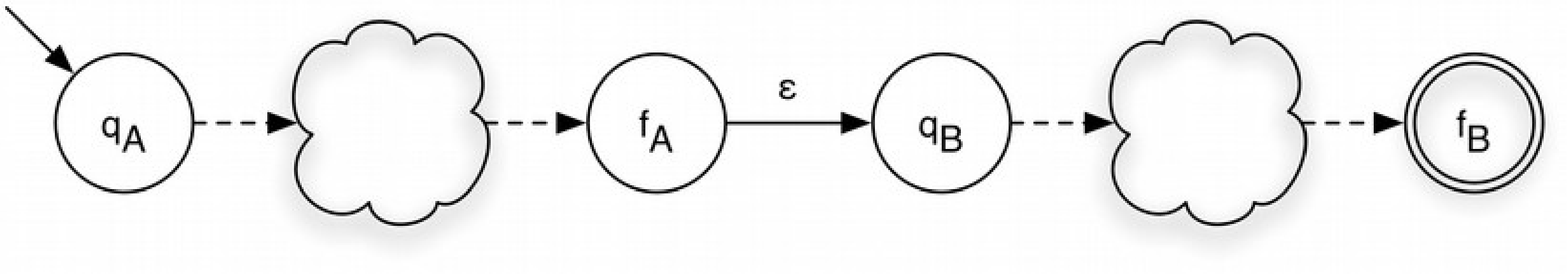
Thompson's: Concatenation

A **B**



Thompson's: Concatenation

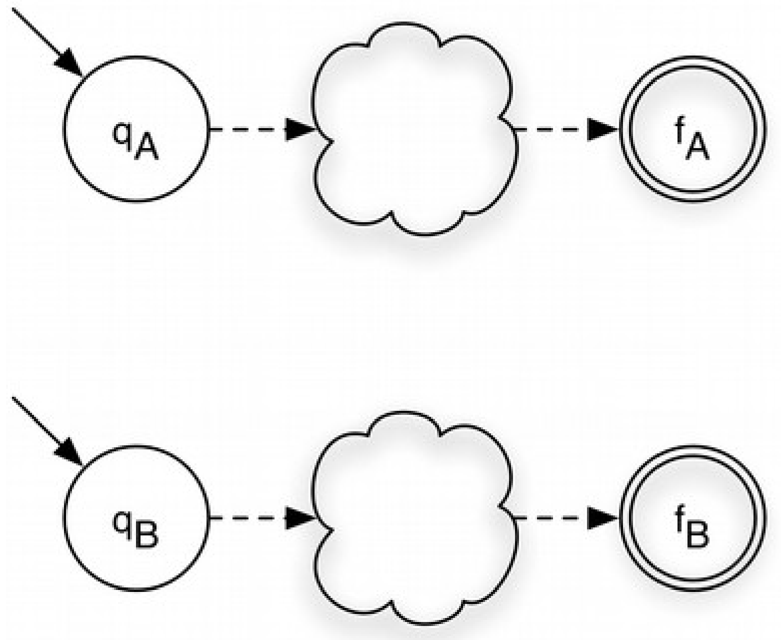
AB



Thompson's: Union

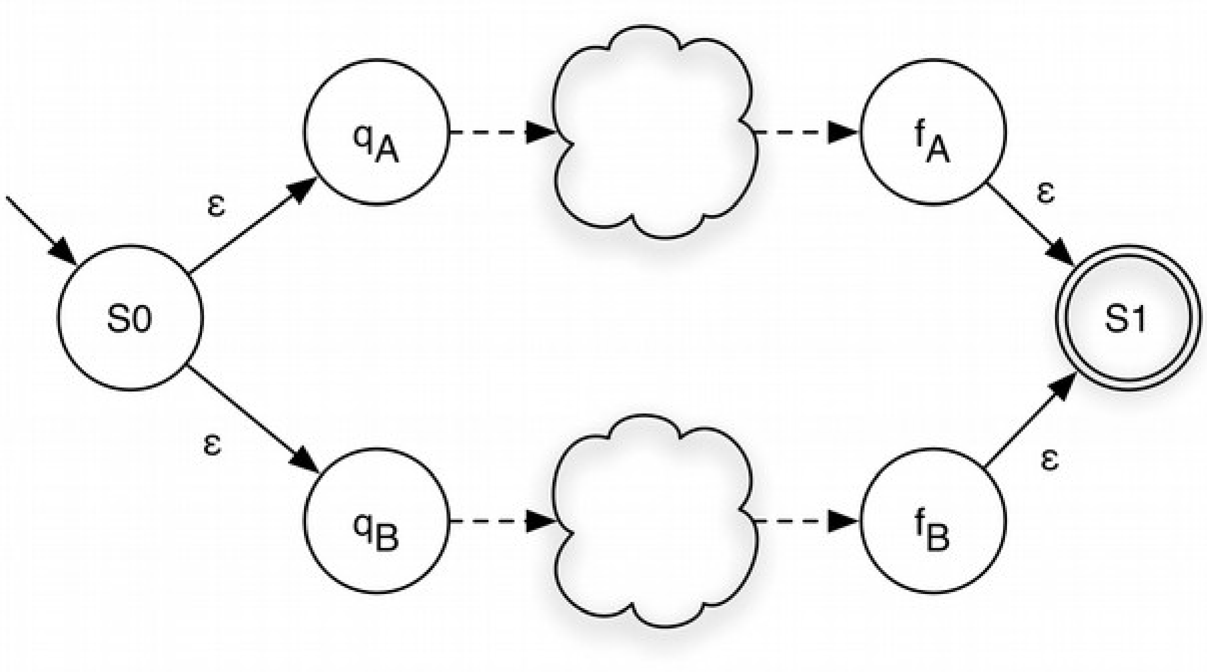
A

B

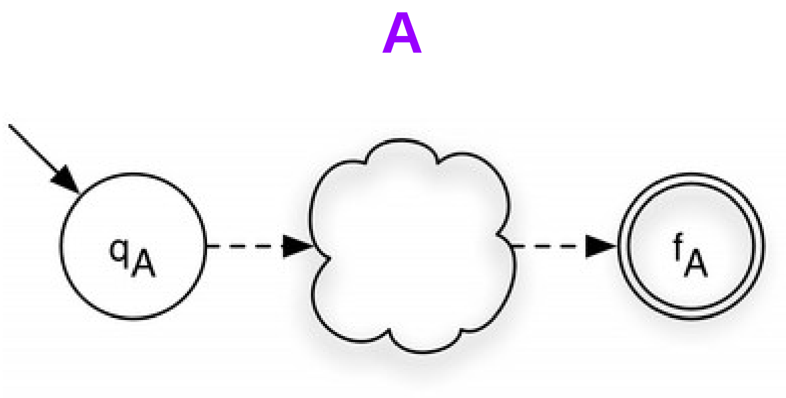


Thompson's: Union

A|B

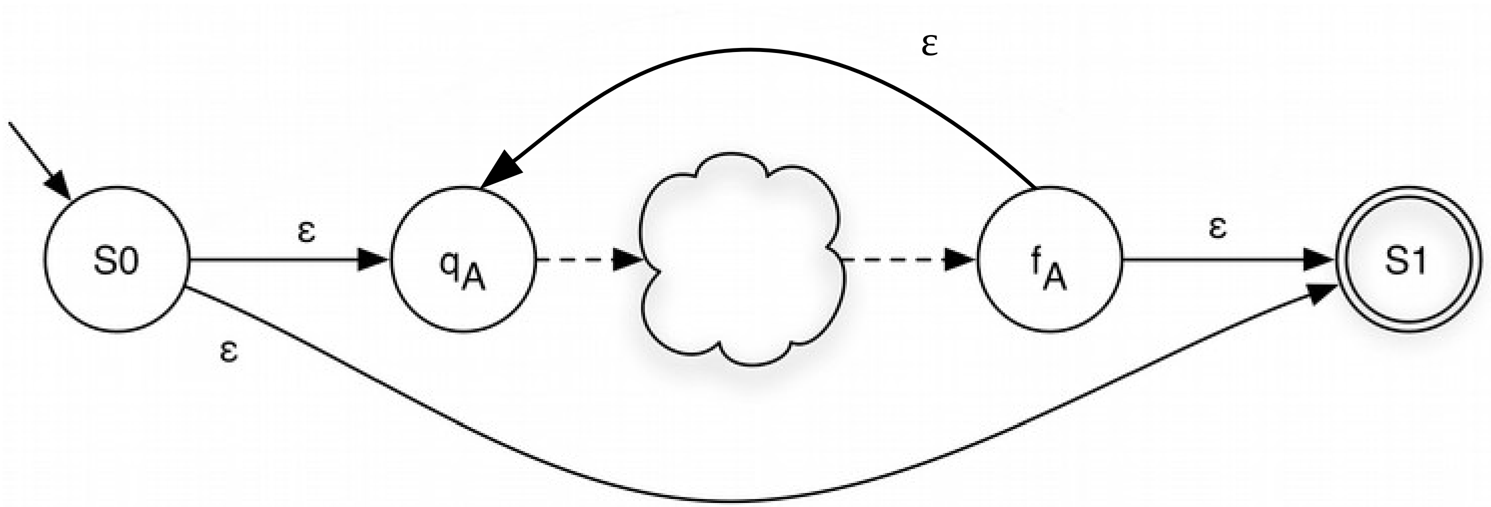


Thompson's: Closure



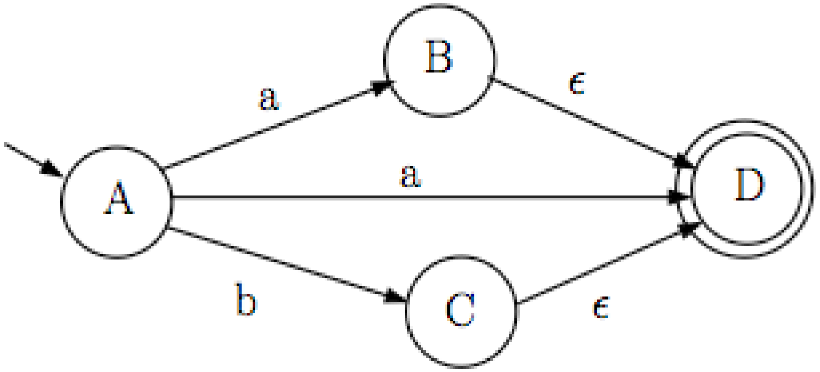
Thompson's: Closure

A*



Subset construction

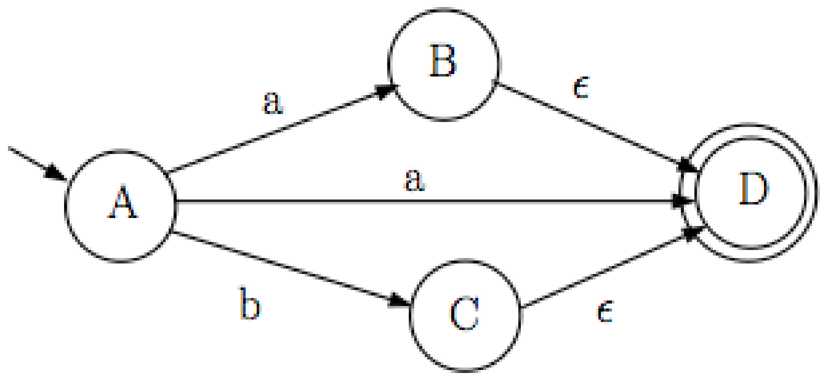
- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” null/epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - i.e., where can we go “for free?”
  - Simulates running all possible paths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



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

Subset construction

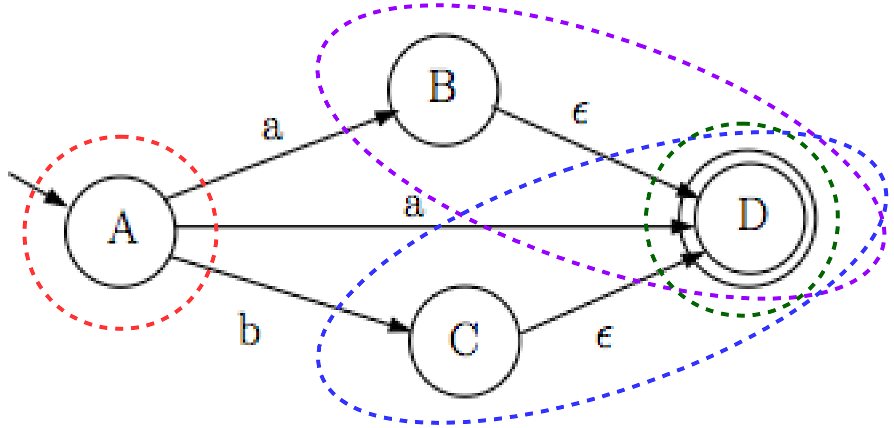
- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” null/epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - i.e., where can we go “for free?”
  - Simulates running all possible paths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

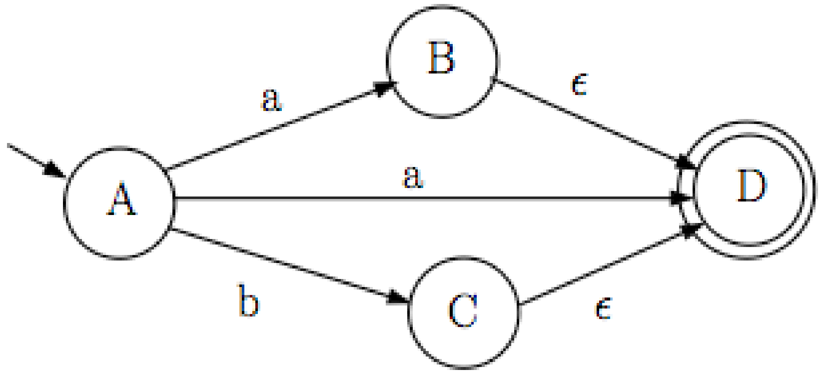
Subset construction

- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” null/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

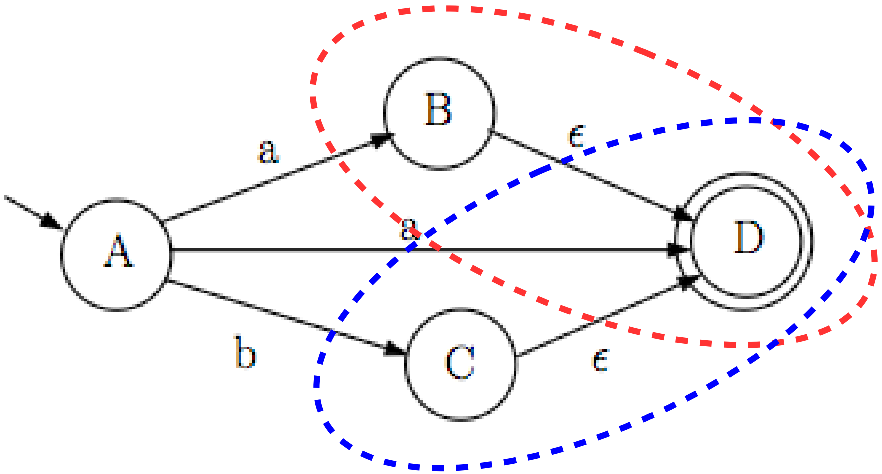


Null closure of A = { A }
Null closure of B = { B, D }
Null closure of C = { C, D }
Null closure of D = { D }

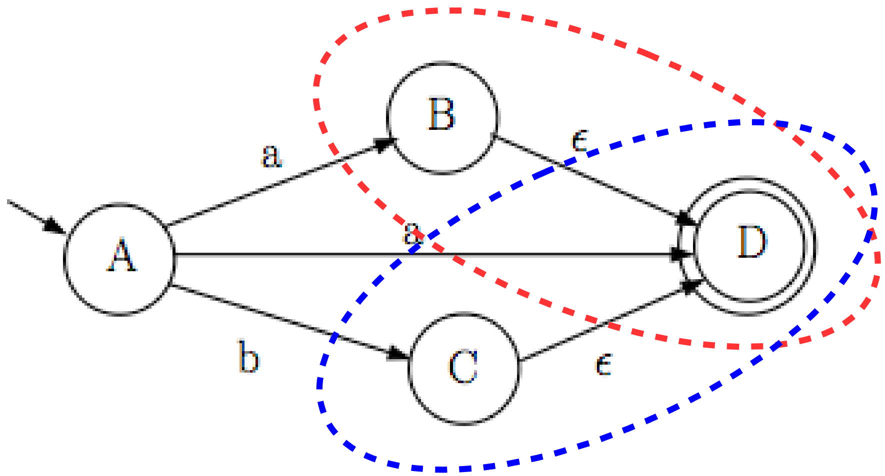
Subset Example

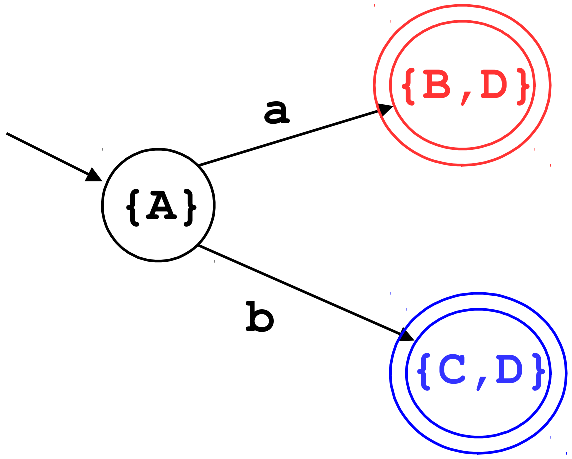


Subset Example



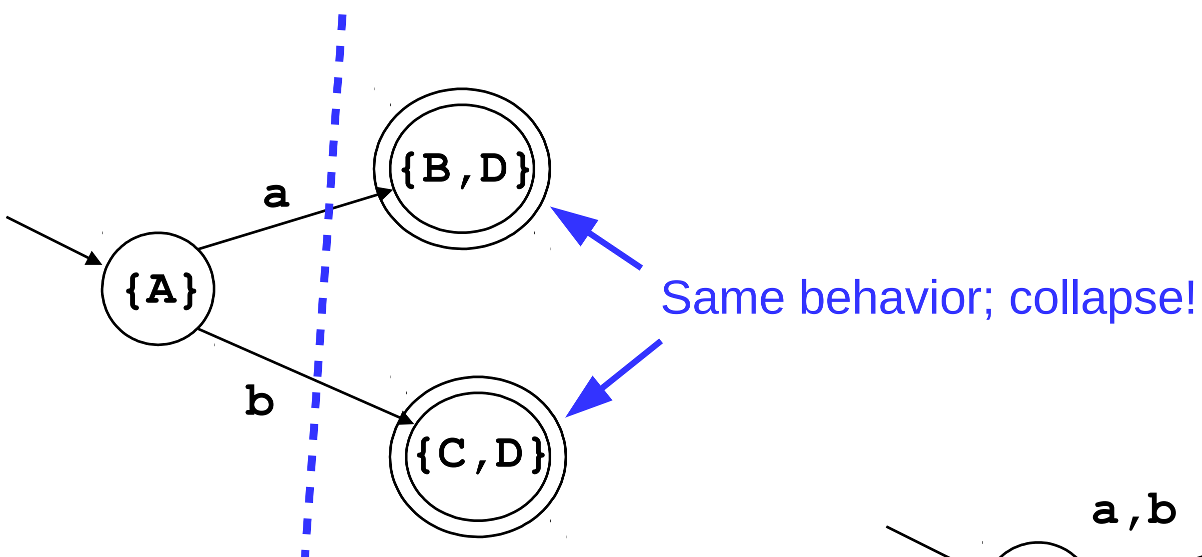
Subset Example

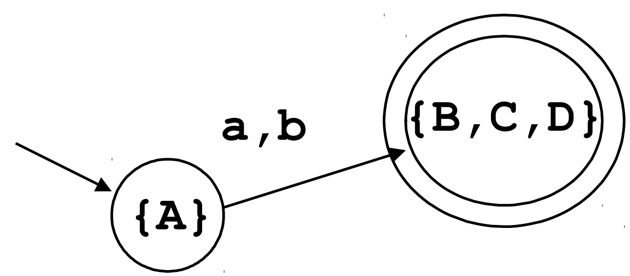




Hopcroft’s DFA Minimization

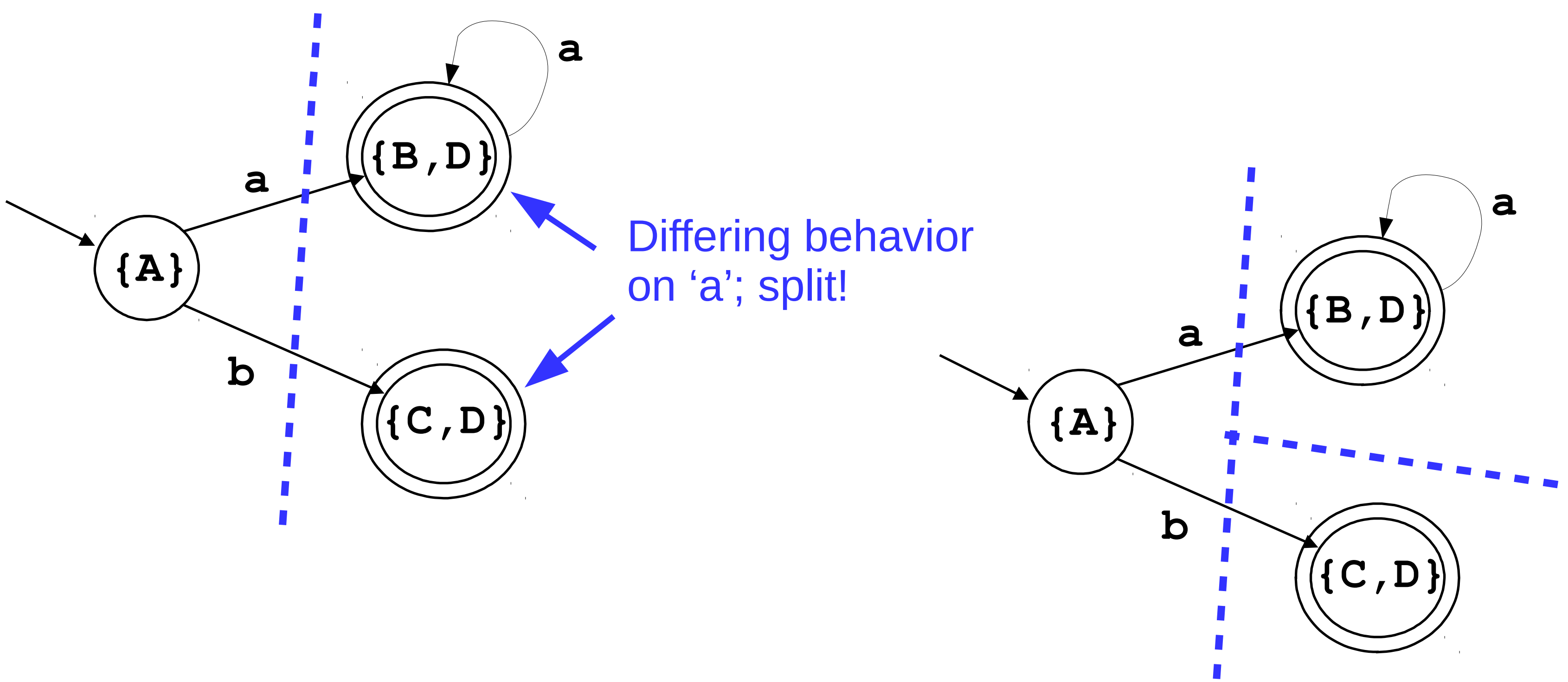
- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with differing behaviors
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn’t
- When done, each partition becomes a single state





Hopcroft's DFA Minimization

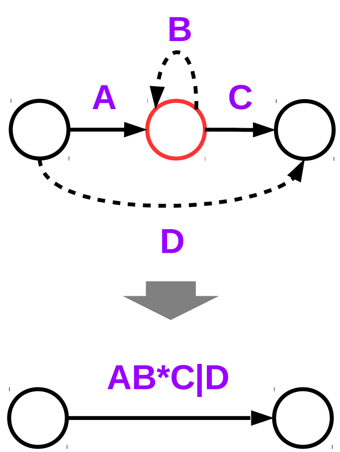
- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with differing behaviors
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn't
- When done, each partition becomes a single state



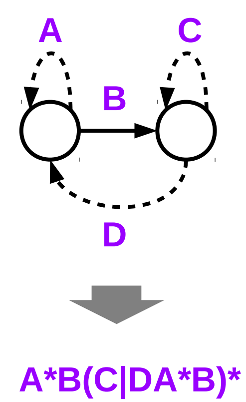
Kleene's Construction

- Replace edge labels with REs
  - "a" → "a" and "a,b" → "a|b"
- Eliminate states by combining REs
  - See pattern below; apply pairwise around each state to be eliminated
  - Repeat until only one or two states remain
- Build final RE
  - One state with "A" self-loop → "A*"
  - Two states: see pattern below

Eliminating states:



Combining final two states:



NFA/DFA complexity

- What are the time and space requirements to...
  - Build an NFA?
  - Run an NFA?
  - Build a DFA?
  - Run a DFA?

NFA/DFA complexity

- 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
  - Or if you need features that are difficult to implement with DFAs
- DFAs build slower but run faster (linear)
  - 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 transitions using jumps
  - 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 valid 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
  - (Thus, this is probably easier for P2)