# CMSC 330: Organization of Programming Languages

## Finite Automata 2

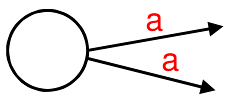


## Types of Finite Automata

- Deterministic Finite Automata (DFA)
  - Exactly one sequence of steps for each string
  - All examples so far
- Nondeterministic Finite Automata (NFA)
  - May have many sequences of steps for each string
  - Accepts if any path ends in final state at end of string
  - More compact than DFA

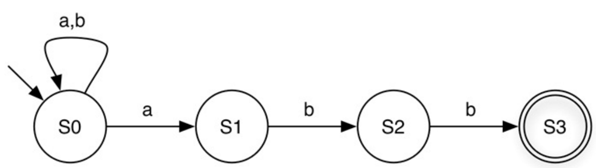
## Comparing DFAs and NFAs

- NFAs can have more than one transition leaving a state on the same symbol



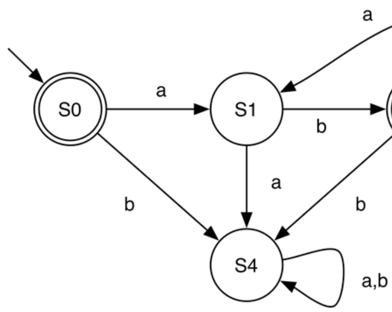

- DFAs allow only one transition per symbol
  - I.e., transition function must be a valid function
  - DFA is a special case of NFA

## NFA for (a|b)*abb




- ba
  - Has paths to either S0 or S1
  - Neither is final, so rejected
- babaabb
  - Has paths to different states
  - One path leads to S3, so accepts string

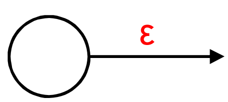
## Another example DFA




- Language?
  - (ab|aba)*

## Comparing DFAs and NFAs (cont.)

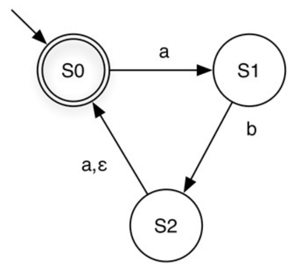
- NFAs may have transitions with empty string label
  - May move to new state without consuming character




ε-transition

- DFA transition must be labeled with symbol
  - DFA is a special case of NFA

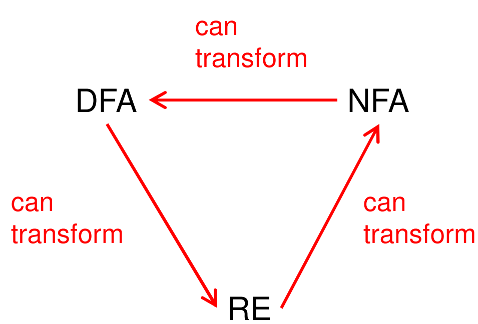
## NFA for (ab|aba)*




- aba
  - Has paths to states S0, S1
- ababa
  - Has paths to S0, S1
  - Need to use ε-transition

## Relating REs to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!



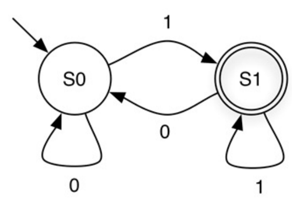
## Formal Definition

- A deterministic finite automaton *(DFA)* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - $\Sigma$ is an alphabet
    - the strings recognized by the DFA are over this set
  - Q is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
    - How many can there be?
  - $\delta : Q \times \Sigma \to Q$ specifies the DFA's transitions
    - What's this definition saying that $\delta$ is?
- A DFA accepts s if it stops at a final state on s

## Formal Definition: Example

- $\Sigma = \{0, 1\}$
- $Q = \{S0, S1\}$
- $q_0 = S0$
- $F = \{S1\}$
-

| $\delta$ | 0 | 1 |
|---|---|---|
| S0 | S0 | S1 |
| S1 | S0 | S1 |



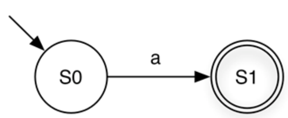

or as { (S0,0,S01),(S0,1,S1),(S1,0,S0),(S1,1,S1) }

## Nondeterministic Finite Automata (NFA)

- An *NFA* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - $\Sigma$ is an alphabet
  - $Q$ is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
  - $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow Q$ specifies the NFA's transitions
    - Transitions on $\varepsilon$ are allowed – can optionally take these transitions without consuming any input
    - Can have more than one transition for a given state and symbol
- An NFA accepts s if there is at least one path from its start to final state on s

## Reducing Regular Expressions to NFAs

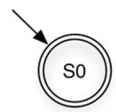
- Goal: Given regular expression e, construct NFA: <e> = $(\Sigma, Q, q_0, F, \delta)$
  - Remember regular expressions are defined recursively from primitive RE languages
  - Invariant: $|F| = 1$ in our NFAs
    - Recall F = set of final states
- Base case: a




<a> = ({a}, {S0, S1}, S0, {S1}, {(S0, a, S1)} )

## Reduction (cont.)

- Base case: ε



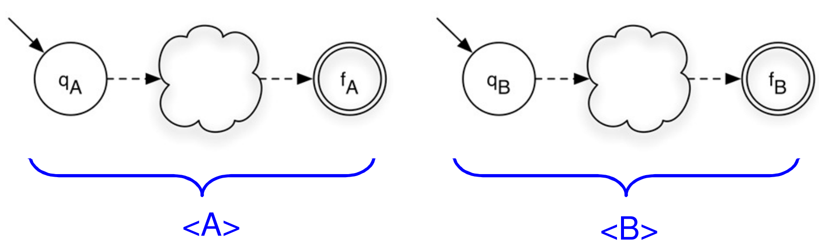

<ε> = (Ø, {S0}, S0, {S0}, Ø)

- Base case: Ø


<Ø> = (Ø, {S0, S1}, S0, {S1}, Ø)

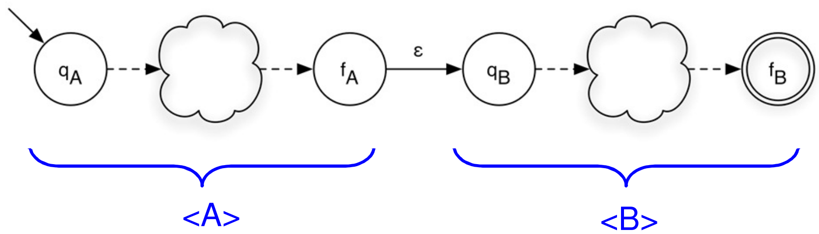
## Reduction: Concatenation

- Induction: AB



## Reduction: Concatenation (cont.)

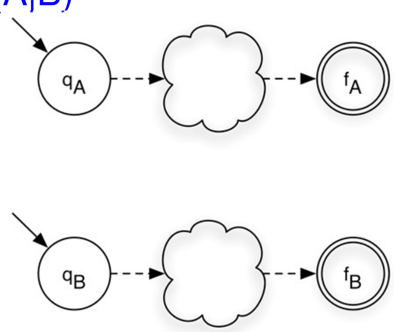
- Induction: AB




- <A> = $(\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- <B> = $(\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
- <AB> = $(\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \{f_B\}, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_B)\})$

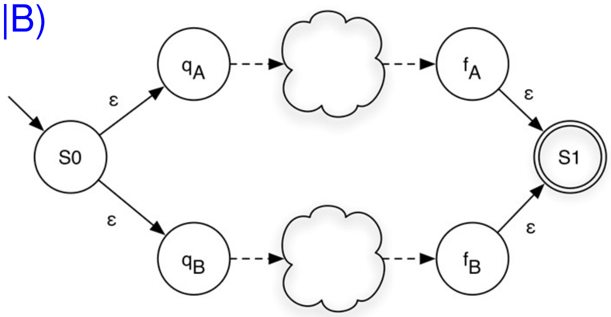
## Reduction: Union

- Induction: (A|B)



## Reduction: Union (cont.)

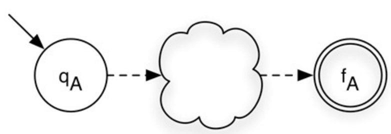
- Induction: (A|B)




  - $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
  - $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
  - $<(A|B)> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0,S1\}, S0, \{S1\}, \delta_A \cup \delta_B \cup \{(S0,\varepsilon,q_A), (S0,\varepsilon,q_B), (f_A,\varepsilon,S1), (f_B,\varepsilon,S1)\})$

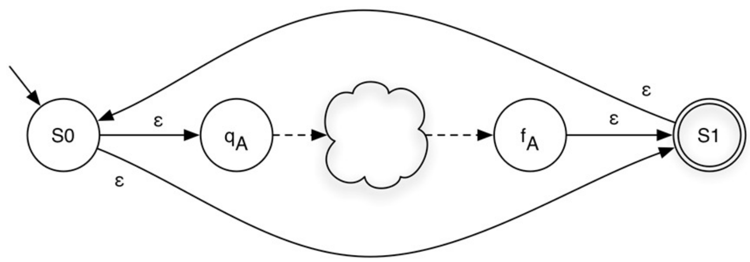
## Reduction: Closure

- Induction: A*



## Reduction: Closure (cont.)

- Induction: A*




- $\langle A \rangle$ = $(\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $\langle A^* \rangle$ = $(\Sigma_A, Q_A \cup \{S0,S1\}, S0, \{S1\},$
  $\delta_A \cup \{(f_A,\varepsilon,S1), (S0,\varepsilon,q_A), (S0,\varepsilon,S1), (S1,\varepsilon,S0)\})$

## Reduction Complexity

- Given a regular expression A of size n...
  Size = # of symbols + # of operations

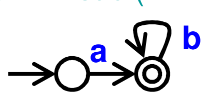
- How many states does <A> have?
  - 2 added for each |, 2 added for each *
  - O(n)
  - That's pretty good!

# Practice

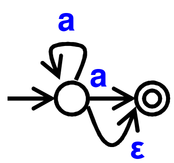
- Draw NFAs for the following regular expressions and languages
  - (0|1)*110*
  - 101*|111
  - all binary strings ending in 1 (odd numbers)
  - all alphabetic strings which come after “hello” in alphabetic order
  - (ab*c|d*a|ab)d

# Recap

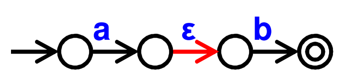
- Finite automata
  - Alphabet, states…
  - $(\Sigma, Q, q_0, F, \delta)$
- Types
  - Deterministic (DFA)



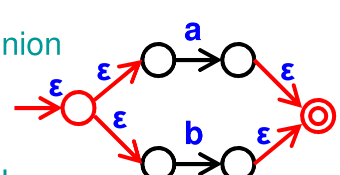

  - Non-deterministic (NFA)



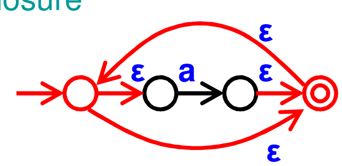

- Reducing RE to NFA
  - Concatenation




  - Union




  - Closure



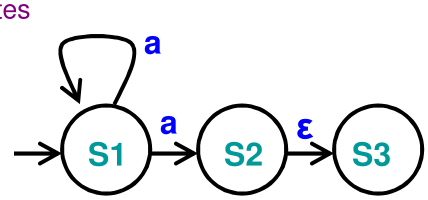
## Next

- Reducing NFA to DFA
  - ε-closure
  - Subset algorithm
- Minimizing DFA
  - Hopcroft reduction
- Complementing DFA
- Implementing DFA



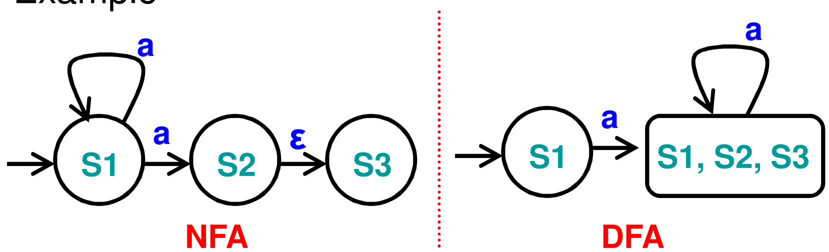
## How NFA Works

- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label
    - ε-transitions
- Example
  - After processing "a"
    - NFA may be in states
      S1
      S2
      S3



## Reducing NFA to DFA

- NFA may be reduced to DFA
  - By explicitly tracking the set of NFA states
- Intuition
  - Build DFA where
    - Each DFA state represents a set of NFA states
- Example






## Reducing NFA to DFA (cont.)

- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states
- Algorithm
  - Input
    - NFA $(\Sigma, Q, q_0, F_n, \delta)$
  - Output
    - DFA $(\Sigma, R, r_0, F_d, \delta)$
  - Using
    - ε-closure(p)
    - move(p, a)

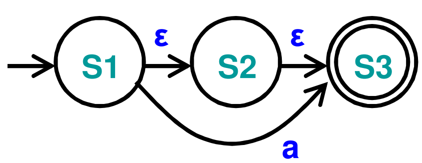
## ε-transitions and ε-closure

- We say $p \xrightarrow{\varepsilon} q$
  - If it is possible to go from state p to state q by taking only ε-transitions
  - If $\exists$ $p, p_1, p_2, \ldots p_n, q \in Q$ such that
    - $\{p,\varepsilon,p_1\} \in \delta$, $\{p_1,\varepsilon,p_2\} \in \delta$, … , $\{p_n,\varepsilon,q\} \in \delta$
- ε-closure(p)
  - Set of states reachable from p using ε-transitions alone
    - Set of states q such that $p \xrightarrow{\varepsilon} q$
    - ε-closure(p) = $\{q \mid p \xrightarrow{\varepsilon} q\}$
  - Note
    - ε-closure(p) always includes p
    - ε-closure( ) may be applied to set of states (take union)



## ε-closure: Example 1

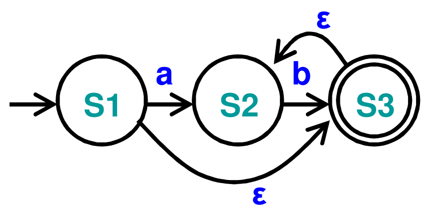
- Following NFA contains
  - S1 $\xrightarrow{\varepsilon}$ S2
  - S2 $\xrightarrow{\varepsilon}$ S3
  - S1 $\xrightarrow{\varepsilon}$ S3




- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2, S3 }
  - ε-closure(S3) = { S3 }
  - ε-closure( { S1, S2 } ) = { S1, S2, S3 } ∪ { S2, S3 }

## ε-closure: Example 2

- Following NFA contains
  - $S1 \xrightarrow{\varepsilon} S3$
  - $S3 \xrightarrow{\varepsilon} S2$
  - $S1 \xrightarrow{\varepsilon} S2$

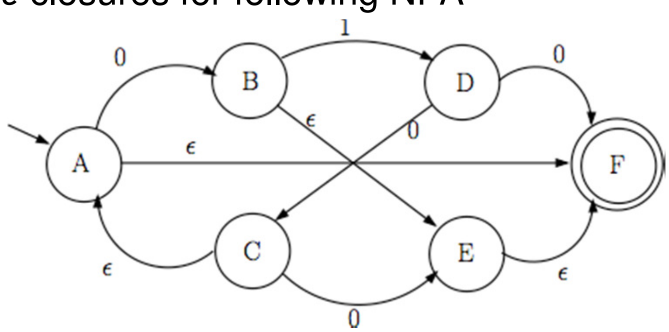



- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2 }
  - ε-closure(S3) = { S2, S3 }
  - ε-closure( { S2,S3 } ) = { S2 } ∪ { S2, S3 }



## ε-closure: Practice

- Find ε-closures for following NFA




- Find ε-closures for the NFA you construct for
  - The regular expression (0|1*)111(0*|1)

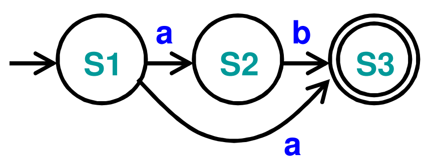
# Calculating move(p,a)

- move(p,a)
  - Set of states reachable from p using exactly one transition on a
    - Set of states q such that {p, a, q} ∈ δ
    - move(p,a) = {q | {p, a, q} ∈ δ}
  - Note move(p,a) may be empty Ø
    - If no transition from p with label a



# move(a,p) : Example 1

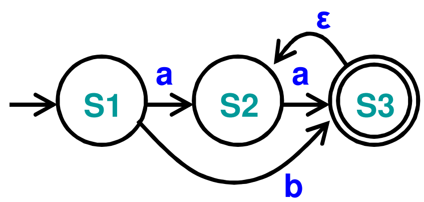
- Following NFA
  - Σ = { a, b }




- Move
  - move(S1, a) = { S2, S3 }
  - move(S1, b) = Ø
  - move(S2, a) = Ø
  - move(S2, b) = { S3 }
  - move(S3, a) = Ø
  - move(S3, b) = Ø

## move(a,p) : Example 2

- Following NFA
  - $\Sigma = \{ a, b \}$




- Move
  - move(S1, a) = { S2 }
  - move(S1, b) = { S3 }
  - move(S2, a) = { S3 }
  - move(S2, b) = Ø
  - move(S3, a) = Ø
  - move(S3, b) = Ø



## NFA → DFA Reduction Algorithm

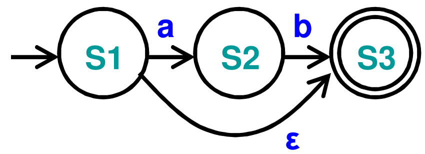
- Input NFA $(\Sigma, Q, q_0, F_n, \delta)$, Output DFA $(\Sigma, R, r_0, F_d, \delta)$
- Algorithm

```
Let r0 = ε-closure(q0), add it to R          // DFA start state
While ∃ an unmarked state r ∈ R              // process DFA state r
    Mark r                                   // each state visited once
    For each a ∈ Σ                           // for each letter a
        Let S = {s | q ∈ r & move(q,a) = s}  // states reached via a
        Let e = ε-closure(S)                 // states reached via ε
        If e ∉ R                             // if state e is new
            Let R = e ∪ R                    // add e to R (unmarked)
        Let δ = δ ∪ {r, a, e}                // add transition r→e
Let Fd = {r | ∃ s ∈ r with s ∈ Fn}           // final if include state in Fn
```

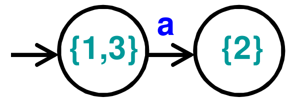
# NFA → DFA Example 1

- Start = ε-closure(S1) = { {S1,S3} }
- R = { {S1,S3} }
- r ∈ R = {S1,S3}
- Move({S1,S3},a) = {S2}
  - e = ε-closure({S2}) = {S2}
  - R = R ∪ {S2} = { {S1,S3}, {S2} }
  - δ = δ ∪ {{S1,S3}, a, {S2}}
- Move({S1,S3},b) = Ø

NFA




DFA

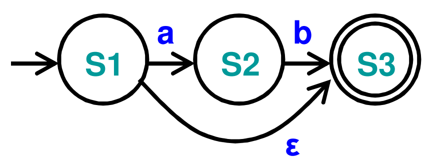




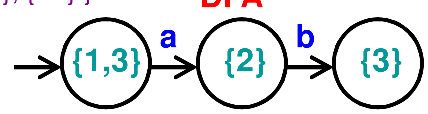

# NFA → DFA Example 1 (cont.)

- R = { {S1,S3}, {S2} }
- r ∈ R = {S2}
- Move({S2},a) = Ø
- Move({S2},b) = {S3}
  - e = ε-closure({S3}) = {S3}
  - R = R ∪ {S3} = { {S1,S3}, {S2}, {S3} }
  - δ = δ ∪ {{S2}, b, {S3}}

NFA

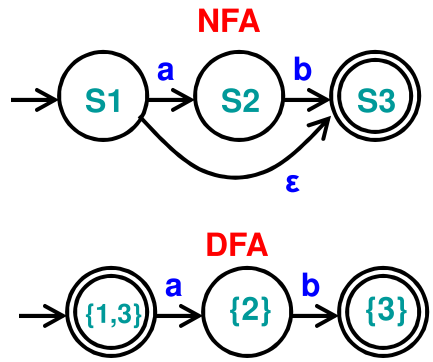



DFA



## NFA → DFA Example 1 (cont.)

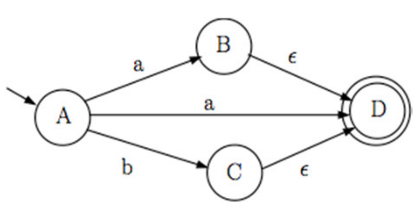
- R = { {S1,S3}, {S2}, {S3} }
- r ∈ R = {S3}
- Move({S3},a) = Ø
- Move({S3},b) = Ø
- $F_d$ = {{S1,S3}, {S3}}
  - Since S3 ∈ $F_n$
- Done!



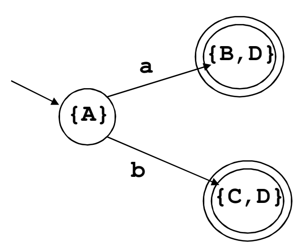



## NFA → DFA Example 2

- NFA



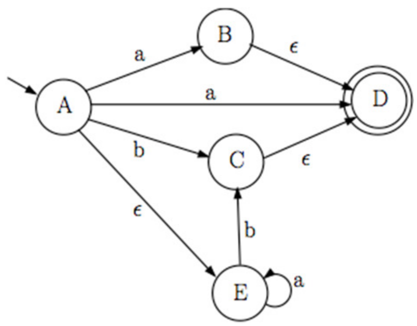

- DFA



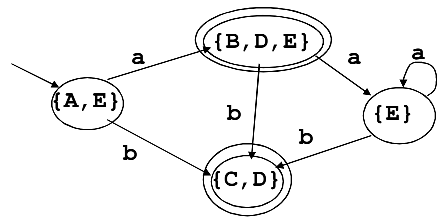

R = { {A}, {B,D}, {C,D} }

# NFA → DFA Example 3

- NFA




- DFA

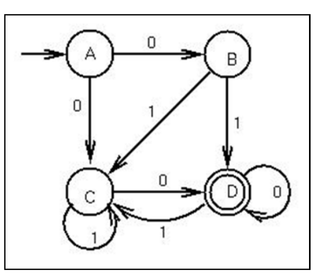


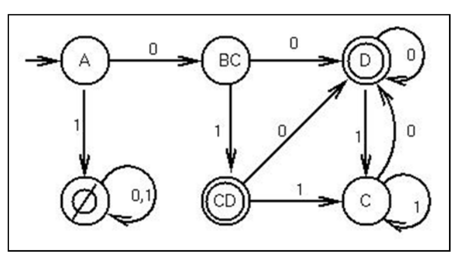

**R = { {A,E}, {B,D,E}, {C,D}, {E} }**



# Equivalence of DFAs and NFAs

- Any string from {A} to either {D} or {CD}
  - Represents a path from A to D in the original NFA




**NFA** **DFA**

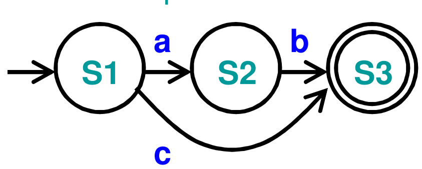
## Equivalence of DFAs and NFAs (cont.)

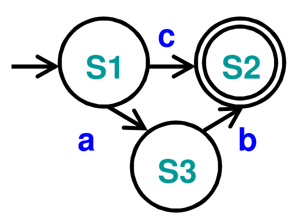
- Can reduce any NFA to a DFA using subset alg.
- How many states in the DFA?
  - Each DFA state is a subset of the set of NFA states
  - Given NFA with n states, DFA may have $2^n$ states
    - Since a set with n items may have $2^n$ subsets
  - Corollary
    - Reducing a NFA with n states may be $O(2^n)$



## Minimizing DFA

- Result from CS theory
  - Every regular language is recognizable by a minimum-state DFA that is unique up to state names
- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
  - Two minimum-state DFAs have same underlying shape



J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," 1971

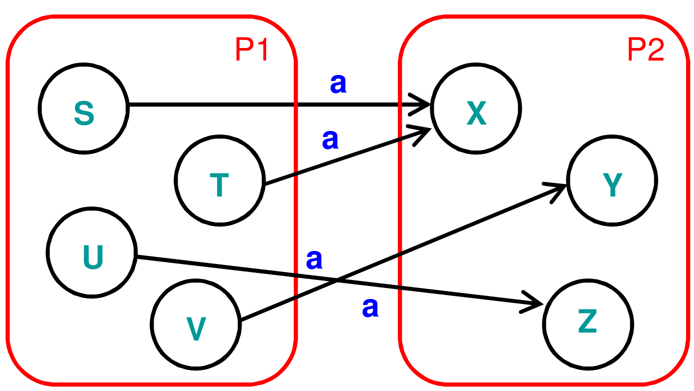
## Minimizing DFA: Hopcroft Reduction

- Intuition
  - Look for states that can be distinguish from each other
    - End up in different accept / non-accept state with identical input
- Algorithm
  - Construct initial partition
    - Accepting & non-accepting states
  - Iteratively refine partitions (until partitions remain fixed)
    - Split a partition if members in partition have transitions to different partitions for same input
      - Two states x, y belong in same partition if and only if for all symbols in Σ they transition to the same partition
  - Update transitions & remove dead states



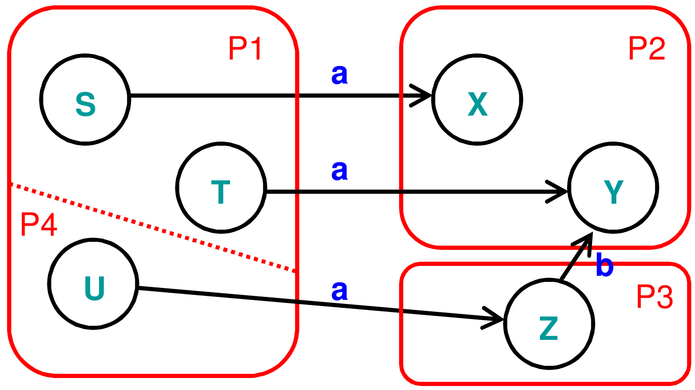
## Splitting Partitions

- No need to split partition {S,T,U,V}
  - All transitions on a lead to identical partition P2
  - Even though transitions on a lead to different states



## Splitting Partitions (cont.)

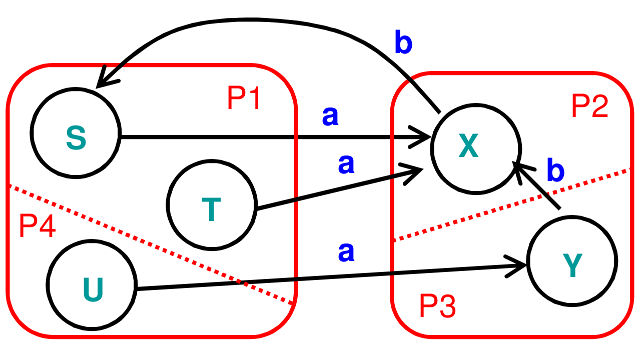
- Need to split partition {S,T,U} into {S,T}, {U}
  - Transitions on a from S,T lead to partition P2
  - Transition on a from U lead to partition P3






## Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition {S,T,U}
  - After splitting partition {X,Y} into {X}, {Y}
  - Need to split partition {S,T,U} into {S,T}, {U}



## DFA Minimization Algorithm (1)

- Input DFA ($\Sigma$, Q, $q_0$, $F_n$, $\delta$), Output DFA ($\Sigma$, R, $r_0$, $F_d$, $\delta$)
- Algorithm

```
Let p0 = Fn, p1 = Q – F          // initial partitions = final, nonfinal states
Let R = { p | p ∈ {p0,p1} and p != Ø }, P = Ø // add p to R if nonempty
While P != R do                   // while partitions changed on prev iteration
    Let P = R, R = Ø              // P = prev partitions, R = current partitions
    For each p ∈ P                // for each partition from previous iteration
        {p0,p1} = split(p,P)      // split partition, if necessary
        R = R ∪ { p | p ∈ {p0,p1} and p != Ø } // add p to R if nonempty
r0 = p ∈ R where q0 ∈ p           // partition w/ starting state
Fd = { p | p ∈ R and exists s ∈ p such that s ∈ Fn } // partitions w/ final states
δ(p,c) = q when δ(s,c) = r where s ∈ p and r ∈ q   // add transitions
```



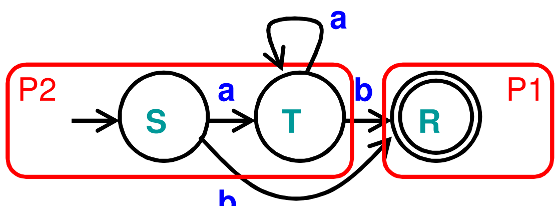
## DFA Minimization Algorithm (2)

- Algorithm for split(p,P)

```
Choose some r ∈ p, let q = p – {r}, m = { } // pick some state r in p
For each s ∈ q                   // for each state in p except for r
    For each c ∈ Σ               // for each symbol in alphabet
        If δ(r,c) = q0 and δ(s,c) = q1 and // q's = states reached for c
            there is no p1 ∈ P such that q0 ∈ p1 and q1 ∈ p1 then
            m = m ∪ {s}          // add s to m if q's not in same partition
Return p – m, m                  // m = states that behave differently than r
                                 // m may be Ø if all states behave the same
                                 // p – m = states that behave the same as r
```

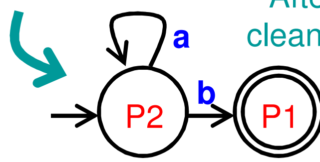
## Minimizing DFA: Example 1

- DFA




- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

After cleanup

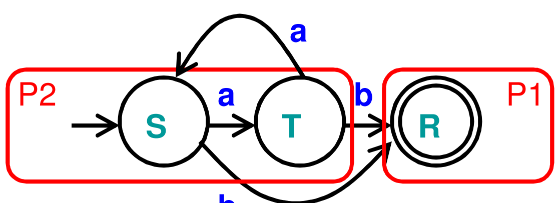



- Split partition? → Not required, minimization done
  - move(S,a) = T → P2
    - move(S,b) = R → P1
  - move(T,a) = T → P2
    - move (T,b) = R → P1



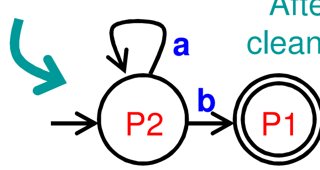
## Minimizing DFA: Example 2

- DFA




- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

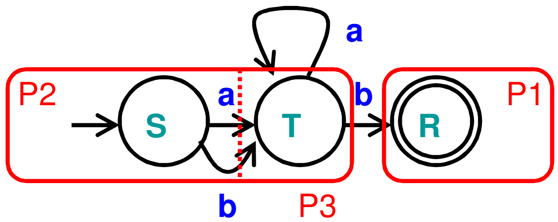
After cleanup




- Split partition? → Not required, minimization done
  - move(S,a) = T → P2
    - move(S,b) = R → P1
  - move(T,a) = S → P2
    - move (T,b) = R → P1

## Minimizing DFA: Example 3

- DFA



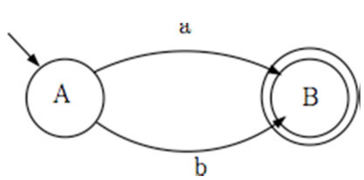

- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2
- Split partition? → Yes, different partitions for B
  - move(S,a) = T → P2
    - move(S,b) = T → P2
  - move(T,a) = T → P2
    - move (T,b) = R → P1

DFA already minimal



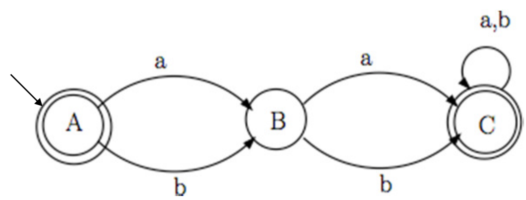
## Complement of DFA

- Given a DFA accepting language L
  - How can we create a DFA accepting its complement?
  - Example DFA
    - $\Sigma = \{a,b\}$



## Complement of DFA (cont.)

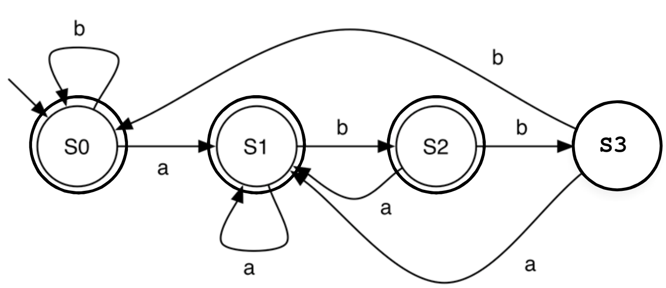
- Algorithm
  - Add explicit transitions to a dead state
  - Change every accepting state to a non-accepting state & every non-accepting state to an accepting state
- Note this only works with DFAs
  - Why not with NFAs?





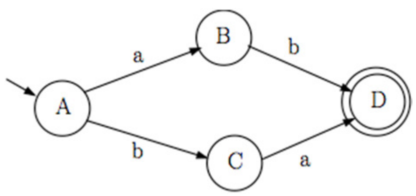

## Practice

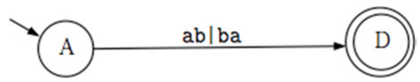
Make the DFA which accepts the complement of the language accepted by the DFA below.



## Reducing DFAs to REs

- General idea
  - Remove states one by one, labeling transitions with regular expressions
  - When two states are left (start and final), the transition label is the regular expression for the DFA



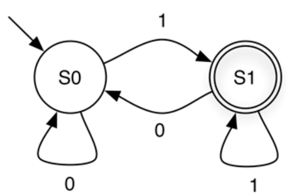



## Relating REs to DFAs and NFAs

- Why do we want to convert between these?
  - Can make it easier to express ideas
  - Can be easier to implement

# Implementing DFAs

It's easy to build a program which mimics a DFA




```
cur_state = 0;
while (1) {

  symbol = getchar();

  switch (cur_state) {

    case 0: switch (symbol) {
              case '0':  cur_state = 0; break;
              case '1':  cur_state = 1; break;
              case '\n': printf("rejected\n"); return 0;
              default:   printf("rejected\n"); return 0;
            }
            break;

    case 1: switch (symbol) {
              case '0':  cur_state = 0; break;
              case '1':  cur_state = 1; break;
              case '\n': printf("accepted\n"); return 1;
              default:   printf("rejected\n"); return 0;
            }
            break;

    default: printf("unknown state; I'm confused\n");
             break;
  }
}
```



# Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

given components ($\Sigma$, Q, $q_0$, F, $\delta$) of a DFA:
let q = $q_0$
while (there exists another symbol s of the input string)
  q := $\delta$(q, s);
if q $\in$ F then
  accept
else reject

- q is just an integer
- Represent $\delta$ using arrays or hash tables
- Represent F as a set

## Run Time of DFA

- How long for DFA to decide to accept/reject string s?
  - Assume we can compute $\delta(q, c)$ in constant time
  - Then time to process s is $O(|s|)$
    - Can't get much faster!
- Constructing DFA for RE A may take $O(2^{|A|})$ time
  - But usually not the case in practice
- So there's the initial overhead
  - But then processing strings is fast

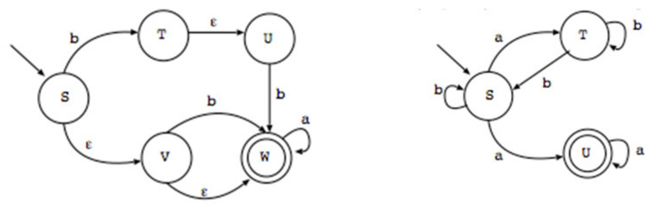


## Regular Expressions in Practice

- Regular expressions are typically "compiled" into tables for the generic algorithm
  - Can think of this as a simple byte code interpreter
  - But really just a representation of $(\Sigma, Q_A, q_A, \{f_A\}, \delta_A)$, the components of the DFA produced from the RE
- Regular expression implementations often have extra constructs that are non-regular
  - I.e., can accept more than the regular languages
  - Can be useful in certain cases
  - Disadvantages
    - Nonstandard, plus can have higher complexity

## Practice

- Convert to a DFA




- Convert to an NFA and then to a DFA
  - (0|1)*11|0*
  - Strings of alternating 0 and 1
  - aba*|(ba|b)



## Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA
- Equivalence of RE, NFA, DFA
  - RE → NFA
    - Concatenation, union, closure
  - NFA → DFA
    - ε-closure & subset algorithm
- DFA
  - Minimization, complement
  - Implementation