CMSC 330: Organization of Programming Languages

Regular Expressions and Finite Automata

The Theory Behind r.e.'s

- That's it for the basics of Ruby
  - If you need other material for your project, you'll either see it in discussion section, or you'll need to learn it on your own
- Next up: How do r.e.'s really work?
  - Mixture of a very practical tool (string matching with r.e.'s) and some nice theory
  - A great computer science result

A Few Questions about Regular Expressions

- What does a regular expression represent?
  - Just a set of strings
- What are the basic components of r.e.'s?
  - E.g., we saw that e+ is the same as ee*
- How are r.e.'s implemented?
  - We'll see how to turn a r.e. into a program
- Can r.e.'s represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the regular languages

Some Definitions

- An *alphabet* is a finite set of symbols
  - Usually denoted Σ
- A *string* is a finite sequence of symbols from Σ
  - ε is the empty string ("" in Ruby)
  - $|s|$ is the length of string s
    - $|Hello| = 5$, $|\varepsilon| = 0$
  - Note: $\varnothing$ is the empty set (with 0 elements); $\varnothing \neq \{ \varepsilon \}$
- *Concatenation* is indicated by juxtaposition
  - If s_1 = super and s_2 = hero, then s_1s_2 = superhero
  - Sometimes also written $s_1 \cdot s_2$
  - For any string s, we have $s\varepsilon = \varepsilon s = s$

Languages

- A *language* is a set of strings over an alphabet
- Example: The set of phone numbers over the alphabet Σ = {0, 1, 2, 3, 4, 5, 6, 7, 9, (,), -}
  - Give an example element of this language
  - Are all strings over the alphabet in the language?
  - Is there a Ruby regular expression for this language?
    - Is the Ruby regular expression over the same alphabet?
- Example: The set of all strings over Σ
  - Often written Σ^*

Languages (cont'd)

- Example: The set of all valid Ruby programs
  - Is there a Ruby regular expression for this language?
- Example: The set of strings of length 0 over the alphabet Σ = {a, b, c}
  - $\{s \mid s \in \Sigma^* \text{ and } |s| = 0\} = \{\varepsilon\} \neq \varnothing$

Operations on Languages

- Let Σ be an alphabet and let L, L_1, L_2 be languages over Σ
- Concatenation L_1L_2 is defined as
  - $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
  - Example: $L_1 = \{$"hi", "bye"$\}$, $L_2 = \{$"1", "2"$\}$
    - $L_1L_2 = \{$"hi1", "hi2", "bye1", "bye2"$\}$
- Union is defined as
  - $L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2\}$
  - Example: $L_1 = \{$"hi", "bye"$\}$, $L_2 = \{$"1", "2"$\}$
    - $L_1 \cup L_2 = \{$"hi", "bye", "1", "2"$\}$

Operations on Languages (cont'd)

- Define L^n inductively as
  - $L^0 = \{\varepsilon\}$
  - $L^n = LL^{n-1}$ for $n > 0$
- In other words,
  - $L^1 = LL^0 = L\{\varepsilon\} = L$
  - $L^2 = LL^1 = LL$
  - $L^3 = LL^2 = LLL$
  - ...

Examples of L^n

- Let $L = \{a, b, c\}$
- Then
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{a, b, c\}$
  - $L^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

Operations on Languages (cont'd)

- *Kleene closure* is defined as

  $L^* = \bigcup_{i \in [0..\infty]} L^i$

- In other words...

  L^* is the language (set of all strings) formed by concatenating together zero or more strings from L

Definition of Regexps

- Given an alphabet Σ, the *regular expressions* over Σ are defined inductively as

regular expression	denotes language
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
each element $\sigma \in \Sigma$	$\{\sigma\}$

– ...

Definition of Regexps (cont'd)

- Let A and B be regular expressions denoting languages L_A and L_B, respectively

regular expression	denotes language
AB	L_AL_B
$(A\|B)$	$L_A \cup L_B$
A^*	L_A^*

- There are no other regular expressions for Σ
- We use ()'s as needed for grouping

The Language Denoted by an r.e.

- For a regular expression e, we will write [[e]] to mean the language denoted by e
  - [[a]] = {a}
  - [[(a|b)]] = {a, b}
- If s∈[[re]], we say that re *accepts*, *describes*, or *recognizes* s.

Example 1

- All strings over Σ = {a, b, c} such that all the a's are first, the b's are next, and the c's last
  - Example: aaabbbbccc but not abcb
- Regexp: a*b*c*
  - This is a valid regexp because...
  - a is a regexp ([[a]] = {a})
  - a* is a regexp ([[a*]] = {ε, a, aa, ...})
  - Similarly for b* and c*
  - So a*b*c* is a regular expression

Which Strings Does a*b*c* Recognize?

aabbbcc
: Yes; aa ∈ [[a*]], bbb ∈ [[b*]], and cc ∈ [[c*]], so entire string is in [[a*b*c*]]

abb
: Yes, abb = abbε, and ε ∈ [[c*]]

ac
: Yes

ε
: Yes

aacbc
: No

abcd
: No -- outside the language

Example 2

- All strings over Σ = {a, b, c}
- Regexp: (a|b|c)*
- Other regular expressions for the same language?
  - (c|b|a)*
  - (a*|b*|c*)*
  - (a*b*c*)*
  - ((a|b|c)*|abc)
  - etc.

Example 3

- All whole numbers containing the substring 330
- Regular expression: (0|1|...|9)*330(0|1|...|9)*
- What if we want to get rid of leading 0's?
- ((1|...|9)(0|1|...|9)*330(0|1|...|9)* | 330(0|1|...|9)*)
- Any other solutions?
- What about all whole numbers not containing the substring 330?
  - Is it recognized by a regexp?

Example 4

- What language does (10|0)*(10|1)* denote?
  - (10|0)*
    - 0 may appear anywhere
    - 1 must always be followed by 0
  - (10|1)*
    - 1 may appear anywhere
    - 0 must always be preceded by 1
  - Put together, all strings of 0's and 1's where every pair of adjacent 0's precedes any pair of adjacent 1's

What Strings are in (10|0)*(10|1)* ?

00101000 110111101

First part in [[(10|0)*]]

Second part in [[(10|1)*]]

Notice that 0010 also in [[(10|0)*]]

But remainder of string is not in [[(10|1)*]]

0010101

Yes

101

Yes

011001

No

Example 5

- What language does this regular expression recognize?
  - ((1|ε)(0|1|...|9) | (2(0|1|2|3))) : (0|1|...|5)(0|1|...|9)
- All valid times written in 24-hour format
  - 10:17
  - 23:59
  - 0:45
  - 8:30

Two More Examples

- (000|00|1)*
  - Any string of 0's and 1's with no single 0's
- (00|0000)*
  - Strings with an even number of 0's
  - Notice that some strings can be accepted more than one way
    - 000000 = 00·00·00 = 00·0000 = 0000·00

Regular Languages

- The languages that can be described using regular expressions are the *regular languages* or *regular sets*
- Not all languages are regular
  - Examples (without proof):
    - The set of palindromes over Σ
    - $\{a^n b^n \mid n > 0\}$ (a^n = sequence of n a's)
- Almost all programming languages are not regular
  - But aspects of them sometimes are (e.g., identifiers)
  - Regular expressions are commonly used in parsing tools

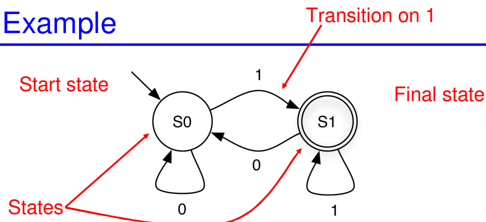
Ruby Regular Expressions

- Almost all of the features we've seen for Ruby r.e.'s can be reduced to this formal definition
  - /Ruby/ – concatenation of single-character r.e.'s
  - /(Ruby|Regular)/ – union
  - /(Ruby)*/ – Kleene closure
  - /(Ruby)+/ – same as (Ruby)(Ruby)*
  - /(Ruby)?/ – same as (ε|(Ruby)) (// is ε)
  - /[a-z]/ – same as (a|b|c|...|z)
  - / [^0-9]/ – same as (a|b|c|...) for a,b,c,... ∈ Σ - {0..9}
  - ^, $ – correspond to extra characters in alphabet

Implementing Regular Expressions

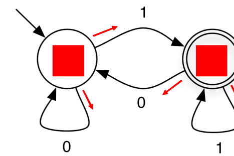
- We can implement regular expressions by turning them into a *finite automaton*
  - A "machine" for recognizing a regular language

Example



- Machine starts in *start* or *initial* state
- Repeat until the end of the string is reached:
  - Scan the next symbol s of the string
  - Take transition edge labeled with s
- The string is *accepted* if the automaton is in a *final* or *accepting* state when the end of the string is reached

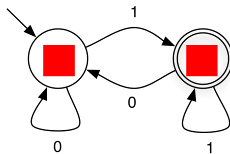
Example



0 0 1 0 1 1

accepted

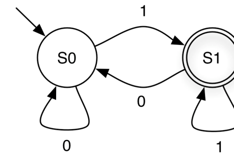
Example



0 0 1 0 1 0

not accepted

What Language is This?



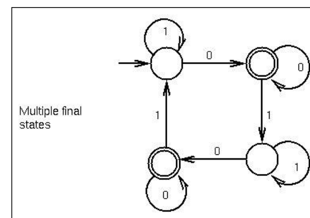
- All strings over {0, 1} that end in 1
- What is a regular expression for this language?
  - (0|1)*1

Formal Definition

- A *deterministic finite automaton (DFA)* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - Σ 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?
  - $\delta : Q \times \Sigma \rightarrow Q$ specifies the DFA's transitions
    - What's this definition saying that δ is?

More on DFAs

- An FSA can have more than one final state:

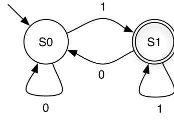


- A string is accepted as long as there is at least one path to a final state

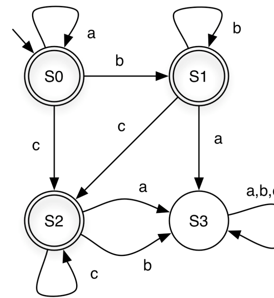
Our Example, Formally

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

δ	0	1
S0	S0	S1
S1	S0	S1



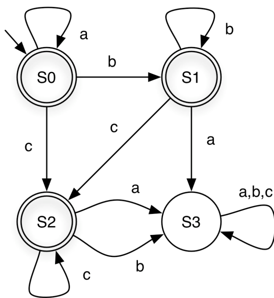
Another Example



string	state at end	accepts ?
aabcc	S2	Y
acc	S2	Y
bbc	S2	Y
aabbb	S1	Y
aa	S0	Y
ε	S0	Y
acba	S3	N

(a,b,c notation shorthand for three self loops)

Another Example (cont'd)

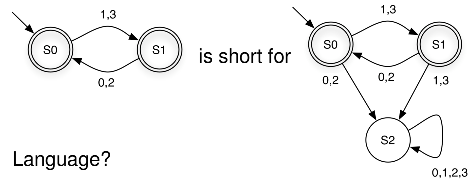


What language does this DFA accept? a*b*c*

S3 is a *dead state* – a nonfinal state with no transition to another state

Shorthand Notation

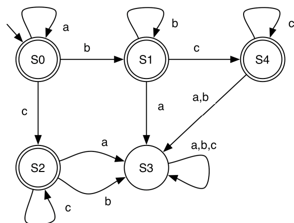
- If a transition is omitted, assume it goes to a dead state that is not shown



Language?

Strings over {0,1,2,3} with alternating even and odd digits, beginning with odd digit

What Lang. Does This DFA Accept?

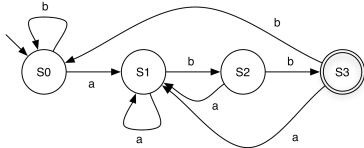


a*b*c* again, so DFAs are not unique

Nondeterministic Finite Automata (NFA)

- An *NFA* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - Σ 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
    - There may be 0, 1, or many
  - $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ specifies the NFA's transitions
    - Transitions on ε 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

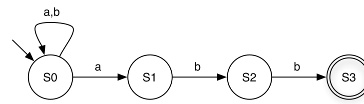
Example DFA



- S0 = "Haven't seen anything yet"
- S1 = "Last symbol seen was an a"
- S2 = "Last two symbols seen were ab"
- S3 = "Last three symbols seen were abb"

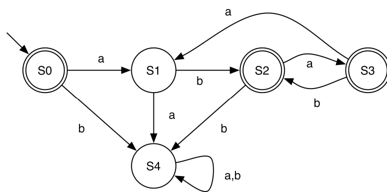
- Language?
- (a|b)*abb

NFA for (a|b)*abb



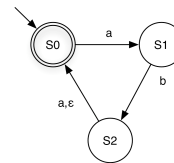
- ba
  - Has paths to either S0 or S1
  - Neither is final, so rejected
- babaabb
  - Has paths to different states
  - One leads to S3, so accepted

Another example DFA



- Language?
- (ab|aba)*

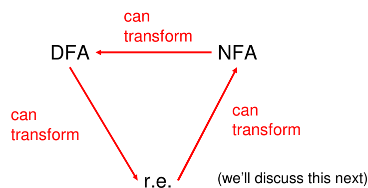
NFA for (ab|aba)*



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

Relating R.E.'s to DFAs and NFAs

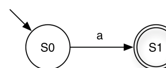
- Regular expressions, NFAs, and DFAs accept the same languages!



Reducing Regular Expressions to NFAs

- Goal: Given regular expression e, construct NFA <e> = $(\Sigma, Q, q_0, F, \delta)$
  - Remember r.e. defined recursively from primitive r.e. languages
  - Invariant: $|F| = 1$ in our NFAs

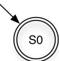
- Base case: a



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

Reduction (cont'd)

- Base case: ε



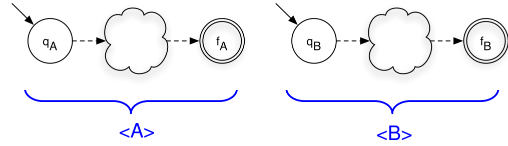
$\langle\varepsilon\rangle = (\varepsilon, \{S0\}, S0, \{S0\}, \varnothing)$

- Base case: ∅

$\langle\varnothing\rangle = (\varnothing, \{S0, S1\}, S0, \{S1\}, \varnothing)$

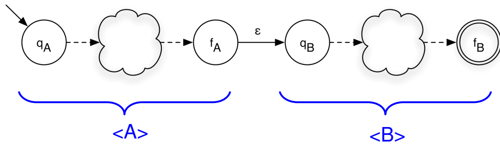
Reduction (cont'd)

- Induction: AB



Reduction (cont'd)

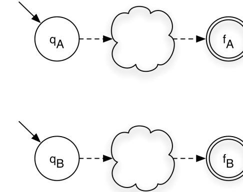
- Induction: AB



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

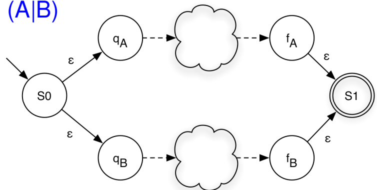
Reduction (cont'd)

- Induction: (A|B)



Reduction (cont'd)

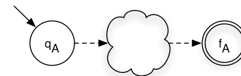
- Induction: (A|B)



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

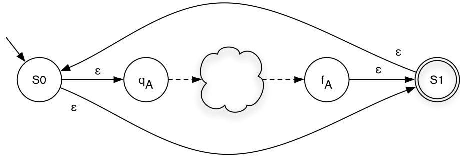
Reduction (cont'd)

- Induction: A*



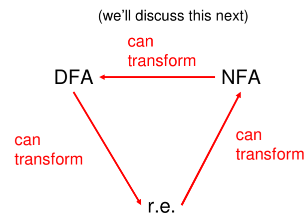
Reduction (cont'd)

- Induction: A*



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

Relating R.E.'s to DFAs and NFAs

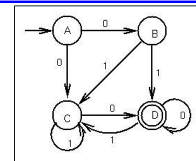


Reduction Complexity

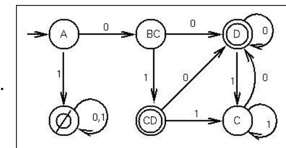
- Given a regular expression A of size n...
  Size = # of symbols + # of operations
- How many states does <A> have?
  – O(n)
  – That's pretty good!
- NFA to DFA reduction
  – Intuition: Build DFA where each DFA state represents a set of NFA states
  – Given NFA with n states, DFA may have 2^n states
  – Not so good, since DFAs are what we can implement easily

Equivalence of DFAs and NFAs

- Let subsets of states be states in DFA
- Keep track of which subset you can be in

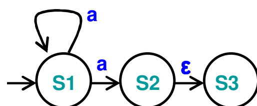


- Any string from {A} to either {D} or {CD} represents a path from A to D in the original NFA.



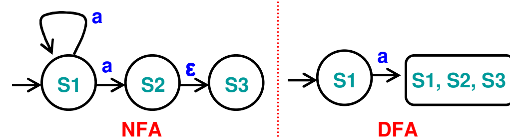
How NFA Works

- 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.)

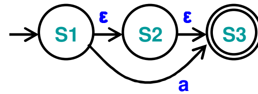
- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states
- Algorithm
  - Input
    - NFA (Σ, Q, q_0, F_n, δ)
  - Output
    - DFA (Σ, R, r_0, F_d, δ)
  - Using
    - ε-closure(p)
    - move(p, a)

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, … 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 | 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

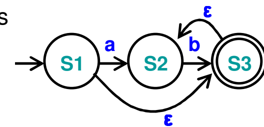
- Following NFA contains
  - S1 $\xrightarrow{\varepsilon}$ S2
  - S2 $\xrightarrow{\varepsilon}$ S3
  - S1 $\xrightarrow{\varepsilon}$ S3



- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2, S3 }
  - ε-closure(S3) = { S3 }
  - ε-closure({ S1, S2 }) = { S1, S2, S3 } $\cup$ { S2, S3 }

ε-closure: Example 2

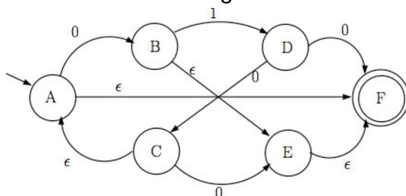
- Following NFA contains
  - S1 $\xrightarrow{\varepsilon}$ S3
  - S3 $\xrightarrow{\varepsilon}$ S2
  - S1 $\xrightarrow{\varepsilon}$ S2



- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2 }
  - ε-closure(S3) = { S2, S3 }
  - ε-closure({ S2,S3 }) = { S2 } $\cup$ { S2, S3 }

ε-closure: Practice

- Find ε-closures for following NFA



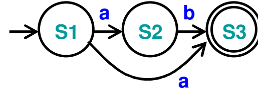
- Find ε-closures for the NFA you construct for
  - The regular expression (0|1*)111(0*|1)

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} $\in \delta$
    - move(p,a) = {q | {p, a, q} $\in \delta$}
  - Note move(p,a) may be empty Ø
    - If no transition from p with label a

move(a,p) : Example 1

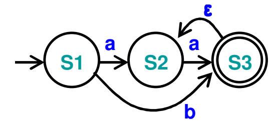
- Following NFA
  - Σ = { a, b }



- 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
  - Σ = { a, b }



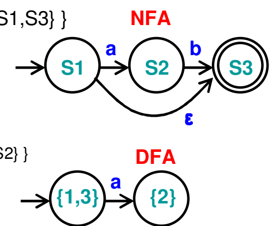
- Move
  - move(S1, a) = { S2 }
  - move(S1, b) = { S3 }
  - move(S2, a) = { S3 }
  - move(S2, b) = Ø
  - move(S3, a) = Ø
  - move(S3, b) = Ø

NFA → DFA Reduction Algorithm

- Input NFA (Σ, Q, q_0, F_n, δ), Output DFA (Σ, R, r_0, F_d, δ)
- Algorithm
  - Let r_0 = ε-closure(q_0), 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 F_d = {r | ∃ s ∈ r with s ∈ F_n} // final if include state in F_n

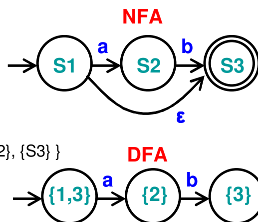
NFA → DFA Example 1

- 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



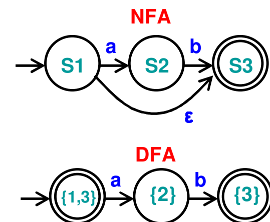
NFA → DFA Example 1 (cont.)

- 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}}



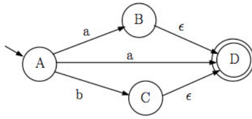
NFA → DFA Example 1 (cont.)

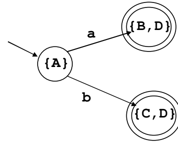
- R = { {S1,S3}, {S2}, {S3} }
- r ∈ R = {S3}
- Move({S3},a} = Ø
- Move({S3},b} = Ø
- F_d = {{S1,S3}, {S3}}
  - Since S3 ∈ F_n
- Done!



NFA → DFA Example 2

- NFA
- DFA

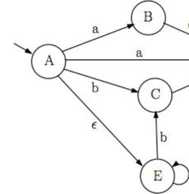


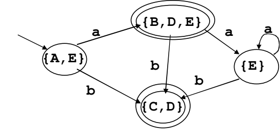


R = { {A}, {B,D}, {C,D} }

NFA → DFA Example 3

- NFA
- DFA

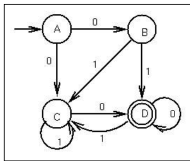


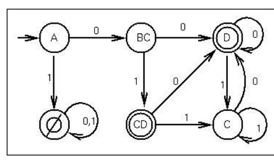


R = { {A,E}, {B,D,E}, {C,D}, {E} }

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





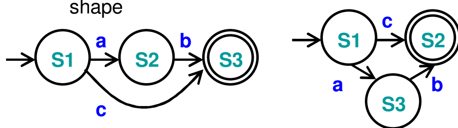
NFA **DFA**

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



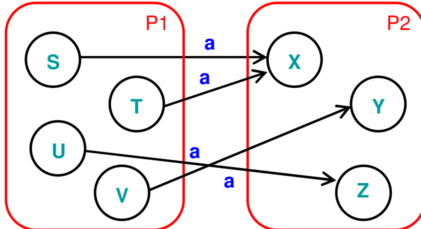
Minimizing DFA: Hopcroft Reduction

- 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

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

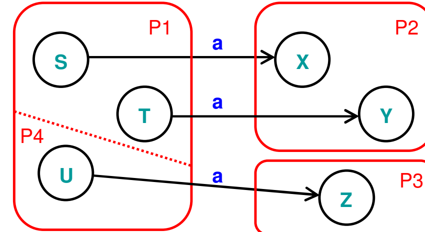
Splitting Partitions

- 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



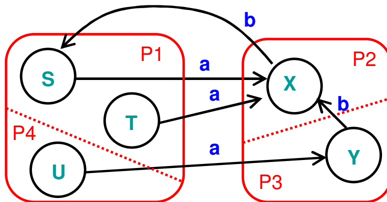
Splitting Partitions (cont.)

- 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 R 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}



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(T,a) = T → P2
  - move(S,b) = R → P1
  - 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(T,a) = S → P2
  - move(S,b) = R → P1
  - move (T,b) = R → P1

Minimizing DFA: Example 3

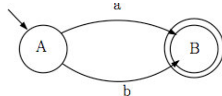
- DFA
- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

DFA already minimal

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

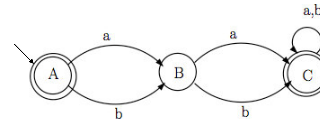
Complement of DFA

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



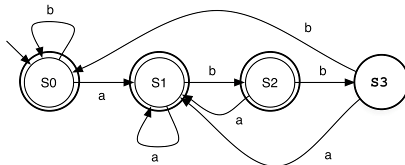
Complement of DFA (cont.)

- 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.

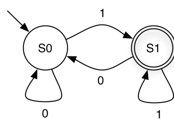


Relating REs to DFAs and NFAs

- 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)

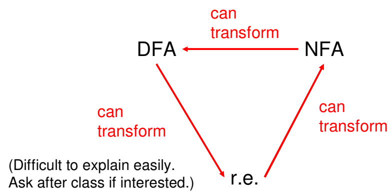
Alternatively, use generic table-driven DFA

```
given components (Σ, Q, q0, F, δ) of a DFA:
let q = q0
while (there exists another symbol s of the input string)
   q := δ(q, s);
if q∈F then
   accept
else reject
```

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

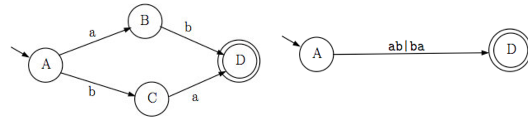
Relating R.E.'s to DFAs and NFAs

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



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



Run Time of Algorithm

- Given a string s, how long does algorithm take to decide whether s is accepted?
  - Assume we can compute $\delta(q0, c)$ in constant time
  - Then the time per string s to determine acceptance is $O(|s|)$
  - Can't get much faster!
- But recall that constructing the DFA from the regular expression A may take $O(2^{|A|})$ time
  - But this is usually not the case in practice
- So there's the initial overhead, but then accepting 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 r.e.
- 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