University of
Central
Florida

Complexity Theory Formal Languages & Automata Theory

Charles E. Hughes

COT6410 – Spring 2023 Notes

Regular Languages

I Hope This is Mostly Review

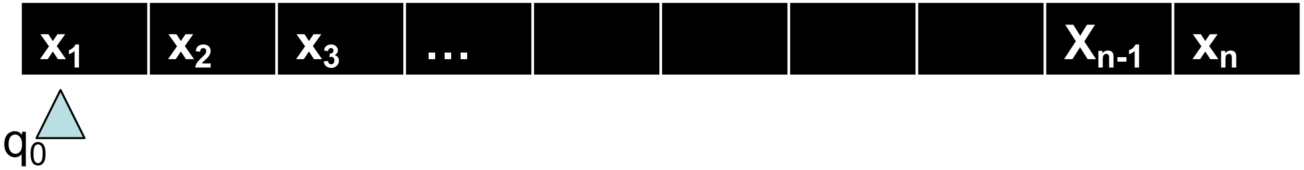
Read Sipser or Aho, Motwani, and Ullman if not old stuff for you

Finite-State Automata

- A Finite-State Automaton (FSA) has only one form of memory, its current state.
- As any automaton has a predetermined finite number of states, this class of machines is quite limited, but still very useful.
- There are two classes: Deterministic Finite-State Automata (DFAs) and Non-Deterministic Finite-State Automata (NFAs)
- We focus on DFAs for now.

Concrete Model of FSA

$A = (Q,\Sigma,\delta,q_0,F)$: Deterministic Final State Automaton (DFA)
$L = L(A)$ is a finite-state (regular) language over finite alphabet Σ
Each **x_i** is a character in Σ
$w = x_1\ x_2 \ldots x_n$ is a string to be tested for membership in **L**



- Blue arrow above represents read head that starts on left.
- **$q_0 \in Q$** (finite state set) is initial state of machine.
- Only action at each step is to change state based on character being read and current state. State change is determined by a transition function δ**: $Q \times \Sigma \rightarrow Q$**.
- Once state is changed, read head moves right.
- Machine stops when head passes last input character.
- Machine accepts a string as a member of **L** if it ends up in a state from Final State set **$F \subseteq Q$**.

Deterministic Finite-State Automata (DFA)

- A deterministic finite-state automaton (DFA) **A** is defined by a 5-tuple
  A = (Q,Σ,δ,q_0,F), where
  - **Q** is a finite set of symbols called the states of A
  - **Σ** is a finite set of symbols called the alphabet of A
  - **δ** is a function from **Q × Σ** into **Q (δ: Q × Σ** → **Q)** called the transition function of **A**
  - **$q_0 \in$Q** is a unique element of **Q** called the start state
  - **F** is a subset of **Q** (**F ⊆ Q**) called the final states (can be empty)

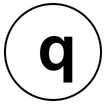
DFA Transitions

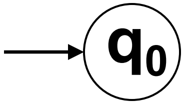
- Given a DFA, $\mathbf{A = (Q,\Sigma,\delta,q_0,F)}$, we can definition the reflexive transitive closure of $\boldsymbol{\delta}$, $\boldsymbol{\delta^*:Q \times \Sigma^* \rightarrow Q}$, by
  - $\boldsymbol{\delta^*(q,\lambda) = q}$ where λ is the string of length 0
    - Some use ϵ rather than λ as symbol for string of length zero
  - $\boldsymbol{\delta^*(q,ax) = \delta^*(\delta(q,a),x)}$, where $\mathbf{a} \in \boldsymbol{\Sigma}$ and $\mathbf{x} \in \boldsymbol{\Sigma^*}$
  - Note that this means
    $\boldsymbol{\delta^*(q,a) = \delta(q,a)}$, where $\mathbf{a} \in \boldsymbol{\Sigma}$ as $\mathbf{a = a}\lambda$
  - Also, if $\boldsymbol{\delta^*(q,x) = p}$ and $\boldsymbol{\delta^*(p,y) = r}$ then $\boldsymbol{\delta^*(q,xy) = r}$
- We also define the transitive closure of δ, δ^+, by
  - $\boldsymbol{\delta^+(q,w) = \delta^*(q,w)}$ when $\mathbf{|w|>0}$ or, equivalently, $\mathbf{w} \in \boldsymbol{\Sigma^+}$
- The function $\boldsymbol{\delta^*}$ describes every step of computation by the automaton starting in some state until it runs out of characters to read

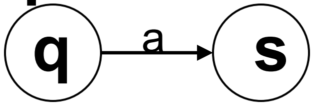
Regular Languages and DFAs

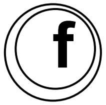
- Given a DFA, **$A = (Q,\Sigma,\delta,q_0,F)$**, we can define the language accepted by **A** as those strings that cause it to end up in a final state once it has consumed the entire string
- Formally, the language accepted by **A** is
  - $\{ w \mid \delta^*(q_0,w) \in F \}$
- We generally refer to this language as ***L*(A)**
- We define the notion of a Regular Language by saying that a language is Regular if and only if it is accepted (recognized) by some DFA

State Diagram

- A finite-state automaton can be described by a state diagram, where
  - Each state is represented by a node labelled with that state, e.g., 

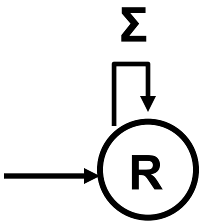
  - The start state has an arc entering it with no source, e.g., 

  - Each transition **δ(q,a) = s** is represented by a directed arc from node **q** to node **s** that is labelled with the letter **a**, e.g., 

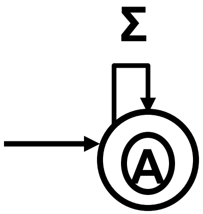
  - Each final state has an extra circle around its node, e.g., 

Really Simple DFAs # 1,2

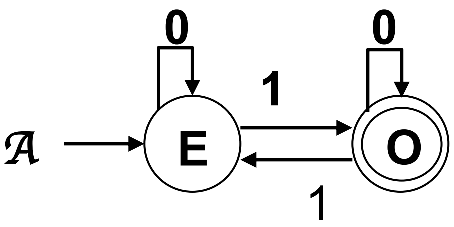
- Accept the Empty Set over **Σ**
  $\mathcal{A}$ **= ({R, Σ,** δ**, R, ∅)**, where δ is defined by



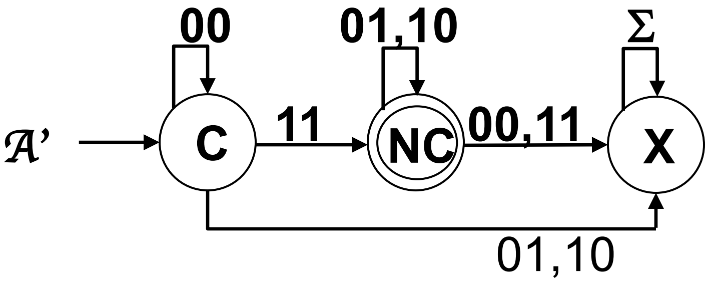
- Accept **Σ***
  $\mathcal{A}$ **= ({A}, Σ,** δ**, A, {A})**, where δ is defined by



Sample DFAs # 3,4



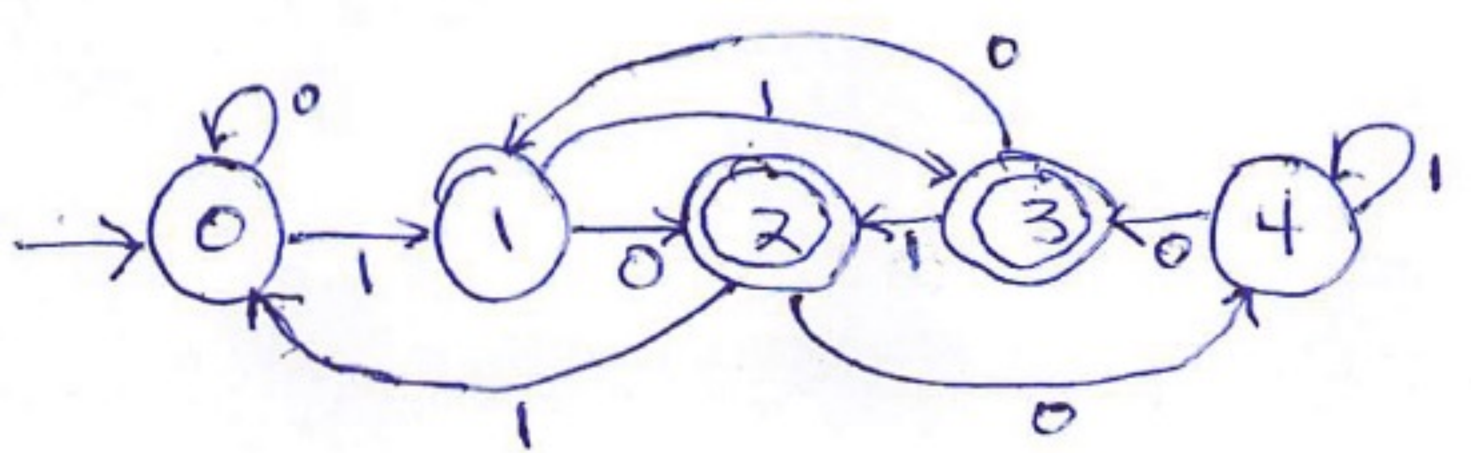
$\mathcal{A}$ = ({E,O}, {0,1}, δ, **E, {O})**, where δ is defined by above diagram.
L($\mathcal{A}$) = { w | w is a binary string of odd parity **}**



$\mathcal{A}$' = ({C,NC,X}, {00,01,10,11}, δ', **C, {NC})**, where δ' is defined by above diagram.
L($\mathcal{A}$') = { w | w is a pair of binary strings where the bottom string is the **2**'s complement of the top one, both read least (lsb) to most significant bit (msb) }

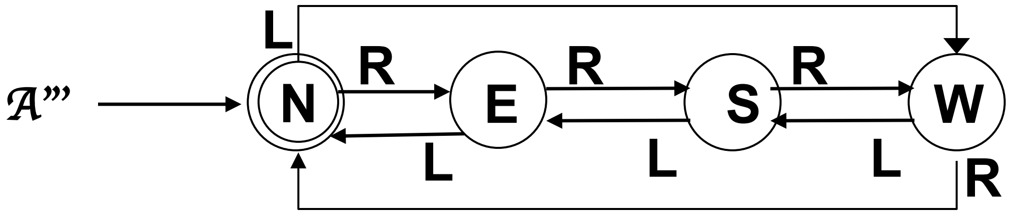
Sample DFA # 5

𝒜”



𝒜” = ({0,1,2,3,4}, {0,1}, δ”, 0, **{2,3})**, where δ” is defined by above diagram. **L(𝒜”) = { w | w** is a binary string that, read left to right (msb to lsb), when interpreted as a decimal number divided by 5, has a remainder of 2 or 3 **}**

Sample DFA # 6



𝒜''' = ({N,E,W,S}, {R,L}, δ''', N, **{N}),** where δ''' is defined by above diagram.
L(𝒜''') = { w | w is a set of commands passed to a sentinel that starts facing North and changes directions R(ight)/clockwise or L(eft)/counterclockwise based on the corresponding input character. w must eventually lead the sentinel back to facing North **}**

State Transition Table

- A finite-state automaton can be described by a state transition table with **|Q|** rows and **|Σ|** columns
- Rows are labelled with state names and columns with input letters
- The start state has some indicator, e.g., a greater than sign (**>q**) and each final state has some indicator, e.g., an underscore (**f̲**)
- The entry in row **q**, column **a**, contains **δ(q,a)**
- In general we will use state diagrams, but transition tables are useful in some cases (state minimization)

Sample DFA # 7

		0	1
→	**0 % 5**	0 % 5	1 % 5
	1 % 5	2 % 5	3 % 5
	2 % 5	4 % 5	0 % 5
Accept State	**3 % 5**	1 % 5	2 % 5
	4 % 5	3 % 5	4 % 5

$\mathcal{A}$’’’ = ({0%5,1%5,2%5,3%5,4%5}, {0,1}, δ’’’, 0, {3%5}), where δ’’’ is defined by above diagram.

L($\mathcal{A}$”) = { w | w is a binary string of length at least 1 being read left to right (msb to lsb) that, when interpreted as a decimal number divided by 5, has a remainder of 3 **}**

Really, this is better done as a state diagram similar to what you saw earlier but have put this up so you can see the pattern.

Sample DFA # 8

	A-Z	a-z	0-9	@#$%^&
⇨ **Empty**	A	a	0	@
A	A	Aa	A0	A@
a	Aa	a	a0	a@
0	A0	a0	0	0@
@	A@	a@	0@	@
Aa	Aa	Aa	Aa0	Aa@
A0	A0	Aa0	A0	A0@
A@	A@	Aa@	A0@	A@
a0	Aa0	a0	a0	a0@
a@	Aa@	a@	a0@	a@
0@	A0@	a0@	0@	0@
Aa0	Aa0	Aa0	Aa0	Aa0@
Aa@	Aa@	Aa@	Aa0@	Aa@
A0@	A0@	Aa0@	A0@	A0@
a0@	Aa0@	a0@	a0@	a0@
Aa0@	Aa0@	Aa0@	Aa0@	Aa0@

This checks a string to see if it's a legal password. In our case, a legal password must contain at least one of each of the following: lower case letter, upper case letter, number, and special character from the following set {**@#$%^&**}. No other characters are allowed

FSAs and Applications

- A synchronous sequential circuit has
  - Binary input lines (input admitted at clock tick)
  - Binary output lines (simple case is one line)
    - 1 accepts; 0 rejects input
  - Internal flip flops (memory) that define state (n flip flops = 2^n states)
  - Simple combinatorial circuits (and, or, not) that combine current state and input to alter internal state
  - Simple combinatorial circuits (and, or, not) that use state to determine output
- Think about FSA to recognize the string PAPAPAT appearing somewhere in a corpus of text, say with a substring PAPAPAPATRICK
- Comments about GREP and Lexical Analysis

Complement of Regular Sets

- Let $\mathbf{A = (Q,\Sigma,\delta,q_0,F)}$ and let $\mathbf{L} = \boldsymbol{L}\mathbf{(A)}$ then
  $\mathbf{w} \notin \boldsymbol{L}\mathbf{(A)}$ iff $\boldsymbol{\delta}^*\mathbf{(q_0,w)} \notin \mathbf{F}$ iff $\boldsymbol{\delta}^*\mathbf{(q_0,w)} \in \mathbf{Q\text{-}F}$
- Simply create new automaton
  $\mathbf{A^C = (Q,\Sigma,\delta,q_0,Q\text{-}F)}$
- $\boldsymbol{L}\mathbf{(A^C) = \{ w \mid \delta^*(q_0,w) \in Q\text{-}F \} =}$
  $\mathbf{\{ w \mid \delta^*(q_0,w) \notin F \} =}$
  $\mathbf{\{ w \mid w \notin} \boldsymbol{L}\mathbf{(A) \}}$
- Choosing the right representation can make a very big difference in how easy or hard it is to prove some property is true

Parallelizing DFAs

- Regular sets can be shown closed under many binary operations using the notion of parallel machine simulation
- Let $\mathbf{A_1 = (Q_1,\Sigma,\delta_1,q_0,F_1)}$ and $\mathbf{A_2 = (Q_2,\Sigma,\delta_2,s_0,F_2)}$ where $\mathbf{Q_1 \cap Q_2 = \emptyset}$
- $\mathbf{B = (Q_1 \times Q_2,\Sigma,\delta_3,<q_0,s_0>,F_3)}$ where
  $\mathbf{\delta_3(<q,s>,a) = < \delta_1(q,a), \delta_2(s,a) >, q \in Q_1, s \in Q_2, a \in \Sigma}$
- Union is $\mathbf{F_3 = (F_1 \times Q_2) \cup (Q_1 \times F_2)}$
- Intersection is $\mathbf{F_3 = F_1 \times F_2}$
  - Can also do by combining union and complement
- Difference is $\mathbf{F_3 = F_1 \times (Q_2 – F_2)}$
  - Can also do by combining intersection and complement
- Exclusive Or is $\mathbf{F_3 = (F_1 \times (Q_2\text{-}F_2)) \cup ((Q_1\text{-}F_1) \times F_2)}$

Reversal of L

- If $\mathbf{x}$ is a string over $\mathbf{\Sigma}$ and $\mathbf{x = a_1\ a_2 \ldots a_n}$,
  then $\mathbf{x^R}$ (x reversed) = $\mathbf{a_n \ldots a_2\ a_1}$
- If **L** is some language, then
  $\mathbf{L^R = \{ x^R \mid x \in L \}}$
- Trying to show if **L** is Regular that $\mathbf{L^R}$ is also Regular, using DFAs is problematic
- Could change start state to final, all final to start states and reverse all arcs
  that is, if $\mathbf{\delta(q,a) = p}$ then $\mathbf{\delta^R(p,a) = q}$, but then the automaton is no longer deterministic

Non-determinism NFA

- A non-deterministic finite-state automaton (NFA) **A** is defined by a 5-tuple
  $A = (Q,\Sigma,\delta,q_0,F)$, where
  - **Q** is a finite set of symbols called the states of **A**
  - **Σ** is a finite set of symbols called the alphabet of **A**
  - **δ** is a function from **$Q \times \Sigma_e$** into **$P(Q) = 2^Q$** ;
    Note: **$\Sigma_e = (\Sigma \cup \{\lambda\})$**
    $\delta: Q \times \Sigma_e \rightarrow P(Q)$ called the transition function of **A**;
    by definition **$q \in \delta(q,\lambda)$**
  - **$q_0 \in Q$** is a unique element of **Q** called the start state
  - **F** is a subset of **Q** (**$F \subseteq Q$**) called the final states

Comments on NFAs

- A state/input (called a discriminant) can lead nowhere, one place or many places in an NFA; moreover, an NFA can jump between states without reading any input symbol
- For simplicity, we often extend the definition of $\boldsymbol{\delta: Q \times \Sigma_e \rightarrow P(Q)}$ to a variant that handles sets of states, where
  $\boldsymbol{\delta: P(Q) \times \Sigma_e \rightarrow P(Q)}$ is defined as
  $\boldsymbol{\delta(S,a) = } \cup_{\boldsymbol{q \in S}} \boldsymbol{\delta(q,a)}$, where $\boldsymbol{a} \in \boldsymbol{\Sigma_e}$
  if $\boldsymbol{S = \emptyset}$, $\cup_{\boldsymbol{q \in S}} \boldsymbol{\delta(q,a) = \emptyset}$

λ-Closure

- Given an NFA, **A = (Q,Σ,δ,q_0,F)**, we can recursively define the **λ-Closure** of **δ**, **λ-Closure:Q** → **P(Q)** by
  - **q** ∈ **λ-Closure(q)**
  - If **s** ∈ **λ-Closure(q)** then **λ-Closure(s)** ⊆ **λ-Closure(q)**
- We can then extend the **λ-Closure** to work on sets so that **λ-Closure:P(Q)** → **P(Q)** is defined by
  - **λ-Closure(S) =** $\cup_{q \in S}$ **λ-Closure(q)** where **S** ⊆ **Q**

NFA Transitions

- Given an NFA, $\mathbf{A = (Q,\Sigma,\delta,q_0,F)}$, we can define the reflexive transitive closure of $\boldsymbol{\delta}$, $\boldsymbol{\delta^*: P(Q) \times \Sigma^* \rightarrow P(Q)}$, by
  - $\boldsymbol{\delta^*(S,\lambda) = \lambda\text{-Closure}(S)}$
  - $\boldsymbol{\delta^*(S,ax) = \delta^*(\lambda\text{-Closure}(\delta(S,a)),x)}$, where $\mathbf{a} \in \boldsymbol{\Sigma}$ and $\mathbf{x} \in \boldsymbol{\Sigma^*}$
    - Note that $\boldsymbol{\delta^*(S,ax) = \cup_{q\in S} \cup_{p\in\lambda\text{-Closure}(\delta(q,a))} \delta^*(p,x)}$, where $\mathbf{a} \in \boldsymbol{\Sigma}$ and $\mathbf{x} \in \boldsymbol{\Sigma^*}$
- We also define the transitive closure of $\boldsymbol{\delta}$, $\boldsymbol{\delta^+}$, by
  - $\boldsymbol{\delta^+(S,w) = \delta^*(S,w)}$ when $\mathbf{|w|>0}$ or, equivalently, $\mathbf{w} \in \boldsymbol{\Sigma^+}$
- The function $\boldsymbol{\delta^*}$ describes every “possible” step of computation by the non-deterministic automaton starting in some state until it runs out of characters to read

NFA Languages

- Given an NFA, **A = (Q,Σ,δ,q_0,F)**, we can define the language accepted by **A** as those strings that allow it to end up in a final state once it has consumed the entire string – here we just mean that there is some accepting path
- Formally, the language accepted by **A** is
  - **$\{ w \mid (\delta^*(\lambda\text{-Closure}(\{q_0\}),w) \cap F) \neq \varnothing \}$**
- Notice that we accept if there is any set of choices of transitions that lead to a final state

Finite-State Diagram

- A non-deterministic finite-state automaton can be described by a finite-state diagram, except
  - We now can have transitions labeled with λ
  - The same letter can appear on multiple arcs from a state **q** to multiple distinct destination states

Equivalence of DFA and NFA

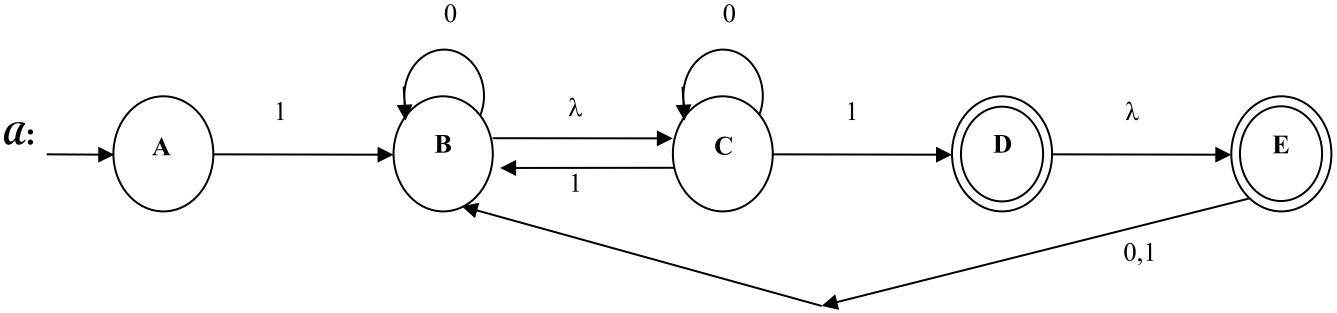
- Clearly every DFA is an NFA except that **δ(q,a) = s** becomes **δ(q,a) = {s}**, so any language accepted by a DFA can be accepted by an NFA.
- The challenge is to show every language accepted by an NFA is accepted by an equivalent DFA. That is, if **A** is an NFA, then we can construct a DFA **A'**, such that ***L*(A') = *L*(A)**.

Constructing DFA from NFA

- Let **A = (Q,Σ,δ,q_0,F)** be an arbitrary NFA
- Let **S** be an arbitrary subset of **Q**.
  - Construct the sequence **seq(S)** to be a sequence that contains all elements of **S** in lexicographical order, using angle brackets to indicate a sequence not a set. That is, if **S={q1, q3, q2}** then **seq(S)=<q1,q2,q3>**. If **S=Ø** then **seq(S)=<>**
- Our goal is to create a DFA, **A**', whose state set contains **seq(S)**, whenever there is some w such that **$S=\delta^*(q_0,w)$**
- To make our life easier, we will act as if the states of **A'** are ordered sets, knowing that we really are talking about corresponding sequences

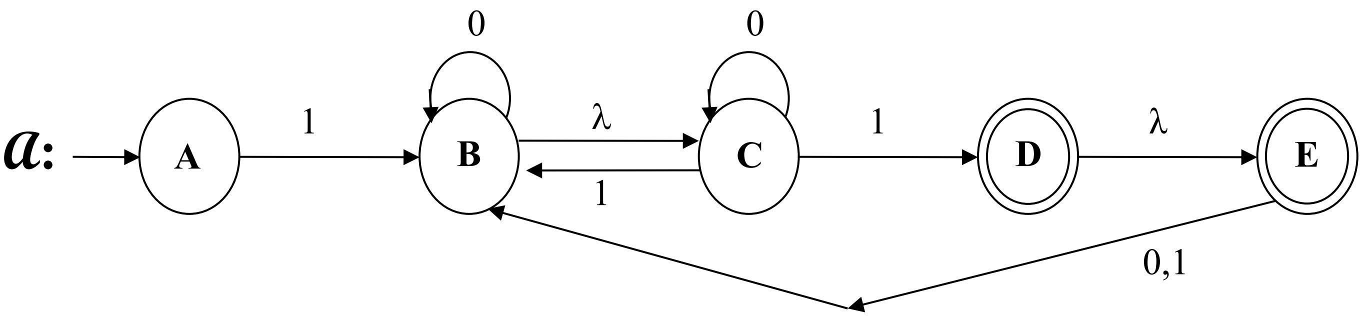
λ-Closure

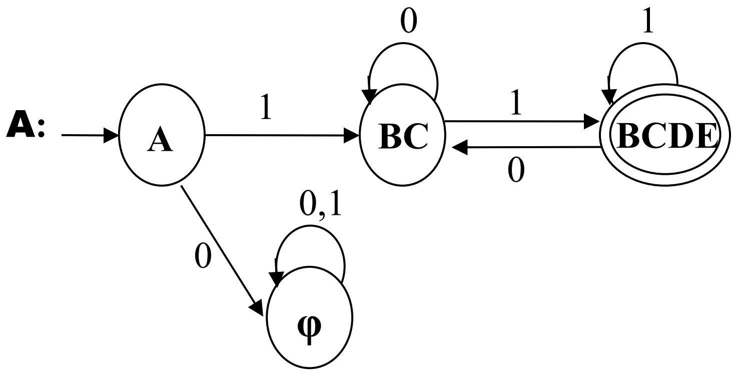
- As before, we define the **λ-Closure** of a state **q** as the set of states one can arrive at from **q**, without reading any additional input.
- Formally **λ-Closure(q) = { t | t ∊ δ*(q,λ) }**
- We can extend this to **S ∈ P(Q)** by
  λ-Closure(S) = { t |t ∊ δ*(q,λ), q ∈ S} = { t |t ∊ λ-Closure(q),q ∈ S}



State	A	B	C	D	E
λ-closure	{ A }	{ B , C }	{ C }	{ D, E }	{ E }

DFA from NFA





Here the DFA has fewer states but, in general, it can have as many as **2ⁿ** states, where the NFA has n states.

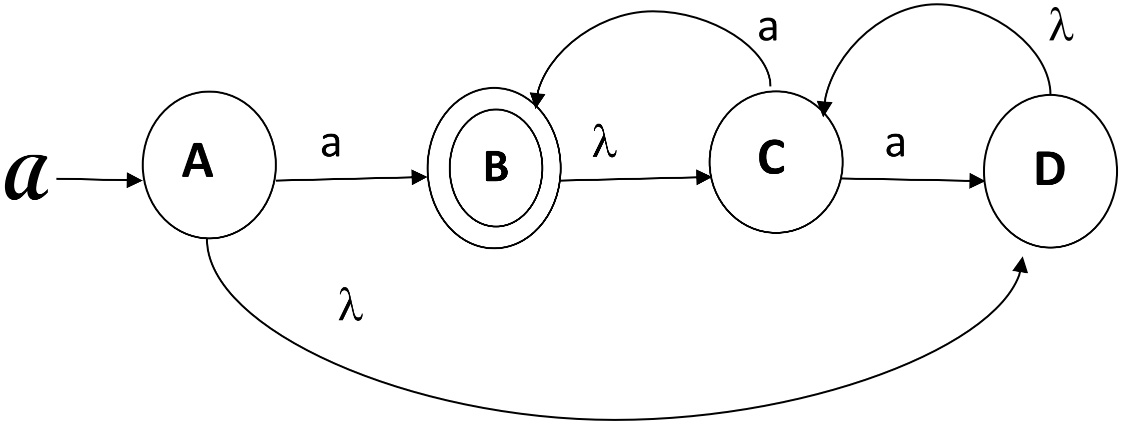
Details of DFA

- Let **$A = (Q,\Sigma,\delta,q_0,F)$** be an arbitrary NFA
- In an abstract sense,
  $A' = (\langle P(Q)\rangle, \Sigma, \delta', \langle\lambda\text{-Closure}(\{q_0\})\rangle, F')$,
  where **$P(Q)$** is the power set of **Q,** but we rarely need so many states ($2^{|Q|}$) and we can iteratively determine those needed by starting at **λ-Closure($\{q_0\}$)** and keeping only states reachable from here
- Define **$\delta'(\langle S\rangle,a) = \langle\lambda\text{-Closure}(\delta(S,a))\rangle =$**
  $\langle\cup_{q\in S}\, \lambda\text{-Closure}(\delta(q,a))\rangle$, where **$a\in\Sigma$, $S \in P(Q)$**
- **$F' = \{\langle S\rangle \in \langle P(Q)\rangle \mid (S \cap F) \neq \emptyset \}$**

Regular Languages and NFAs

- Showing that every DFA can be simulated by an NFA that accepts the same language and every NFA can be simulated by a DFA that accepts the same language proves the following
- A language is Regular if and only if it is accepted (recognized) by some NFA
- We now have two equivalent classes of recognizers for Regular Languages

Simple Exercise: Convert from NFA to DFA



Regular Expressions

Regular Sets

Regular Expressions

- Primitive:
  - Φ denotes {}
  - λ denotes {λ}
  - a where a is in Σ denotes {a}
- Closure:
  - If R and S are regular expressions then so are R ∙ S, R + S and R*, where
    - R ∙ S denotes RS = { xy | x is in R and y is in S }
    - R + S denotes R∪S = { x | x is in R or x is in S }
    - R* denotes R* (defined in page 28 of preliminaries)
- Parentheses are used as needed

Lexical Analysis

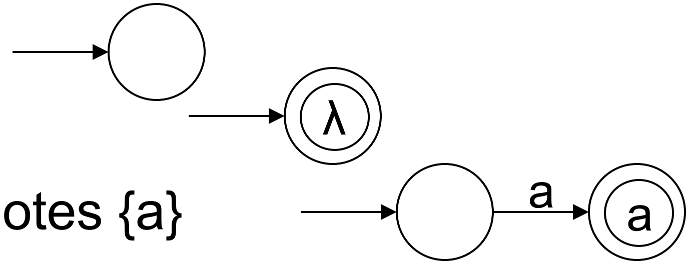
- Consider distinguishing variable names from keywords like
  - IF return(IFSY);
  - INT return(INT);
  - [a-zA-Z]([a-zA-Z0-9_])* return(IDENT);
    - Equivalent to a+b+…+z, etc.
- This really screams for non-determinism
  - With added constraints of finding longest/first match
- Non-deterministic automata typically have fewer states
- However, non-deterministic FSA (NFA) interpretation is not as fast as deterministic

Regular Sets = Regular Languages

- Show every regular expression denotes a language recognized by a finite-state automaton (can do deterministic or non-deterministic)
- Show every Finite-State Automata recognizes a language denoted by a regular expression

Every Regular Set is a Regular Language

- Primitive:
  - Φ denotes { }
  - λ denotes {λ}
  - a where a is in Σ denotes {a}

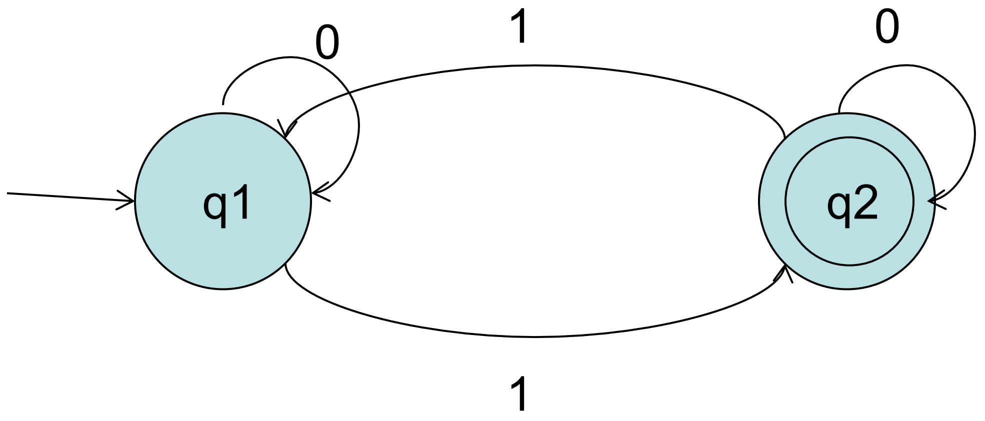


- Closure: (Assume that R's and S's states do not overlap)
  - R · S start with machine for R, add λ transitions from every final state of R's recognizer to start state of S, making final state of S final states of new machine
  - R + S create new start state and add λ transitions from new state to start states of each of R and S, making union of R's and S's final states the new final states
  - R* add λ transitions from each original final state of R back to its start state; keeping original start and making it only final state

Every Regular Language is a Regular Set Using R_{ij}^k

- This is a challenge that can be addressed in multiple ways. but I like to start with the R_{ij}^k approach. Here's how it works.
- Let A = (Q,Σ,δ,q_1,F) be a DFA, where $Q = \{q_1,q_2, \dots , q_n\}$
- $R_{ij}^k = \{w \mid \delta^*(q_i,w) = q_j$, and no intermediate state visited between q_i and q_j, while reading w, has index > k
- Basis: k=0, $R_{ij}^0 = \{ a \mid \delta(q_i,a) = q_j \}$ sets are either Φ, λ, or elements of Σ, or λ + elements of Σ, and so are regular sets
- Inductive hypothesis: Assume R_{ij}^m are regular sets for $0 \le m \le k$, $1 \le i,j \le n$
- Inductive step: k+1, $R_{ij}^{k+1} = (R_{ij}^k + R_{ik+1}^k \cdot (R_{k+1k+1}^k)^* \cdot R_{k+1j}^k)$
- $L(A) = +_{qf\in F} R_{1f}^n$

Convert to RE (Odd Parity)



$R_{11}^0 = \lambda+0$ $\quad R_{12}^0 = 1$ $\quad R_{22}^0 = \lambda+0$ $\quad R_{21}^0 = 1$

$R_{11}^1 = 0^*$ $\quad R_{12}^1 = 0^*1$ $\quad R_{22}^1 = \lambda+0+10^*1$ $\quad R_{21}^1 = 10^*$

$R_{12}^1 = 1+(\lambda+0)(\lambda+0)^*1 = 1+0^*1 = 0^*1$

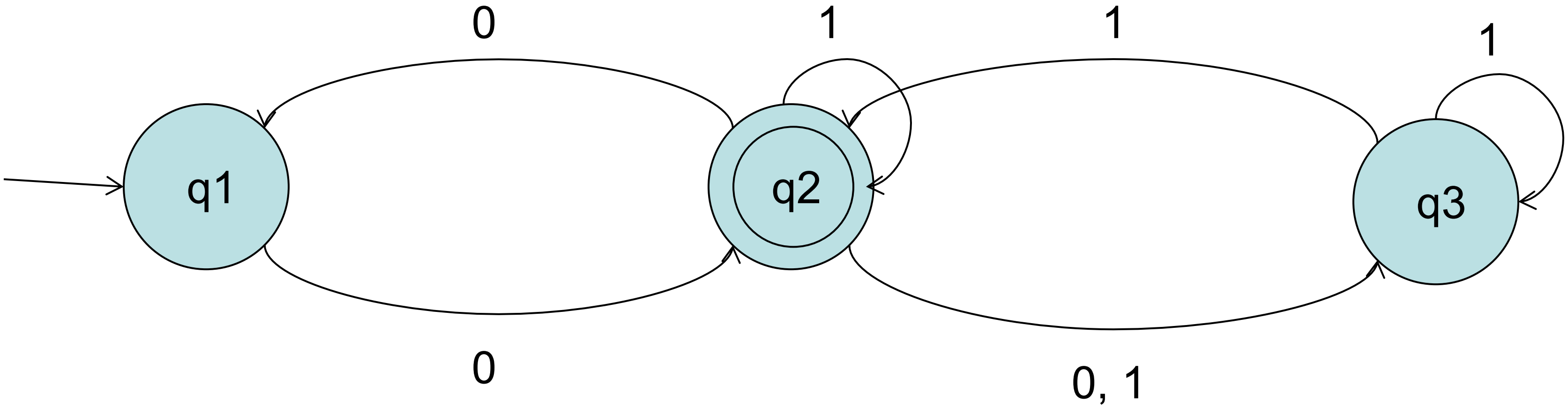
$R_{22}^1 = \lambda+0+1(\lambda+0)^*1 = \lambda+0+10^*1$

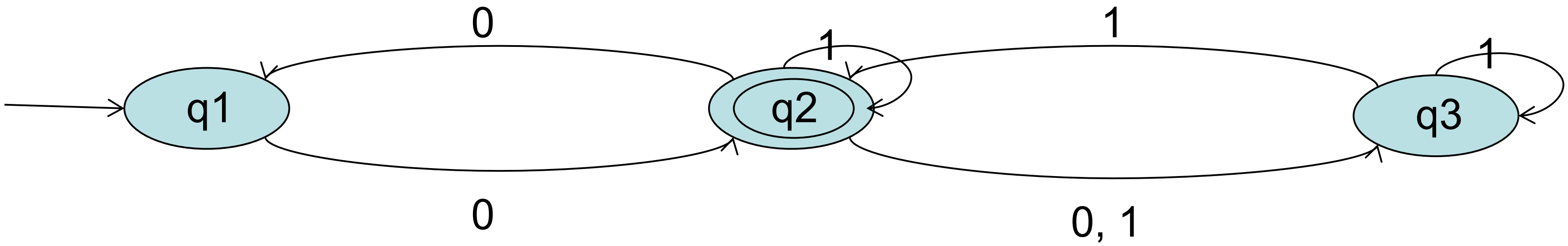
$R_{12}^2 = R_{12}^1 + R_{12}^1 \,(R_{22}^1)^*\, R_{22}^1$

$L = R_{12}^2 = 0^*1+0^*1(\lambda+0+10^*1)^*(\lambda+0+10^*1) = 0^*1(0+10^*1)^*$

Why might a recursive rather than inductive approach be better?

Convert to RE





- $R_{11}^{0}= \lambda$ $R_{12}^{0}= 0$ $R_{13}^{0}= \phi$
- $R_{21}^{0}= 0$ $R_{22}^{0}= \lambda + 1$ $R_{23}^{0}= 0 + 1$
- $R_{31}^{0}= \phi$ $R_{32}^{0}= 1$ $R_{33}^{0}= \lambda + 1$
- $R_{11}^{1}= \lambda$ $R_{12}^{1}= 0$ $R_{13}^{1}= \phi$
- $R_{21}^{1}= 0$ $R_{22}^{1}= \lambda + 1 + 00$ $R_{23}^{1}= 0 + 1$
- $R_{31}^{1} = \phi$ $R_{32}^{1}= 1$ $R_{33}^{1}= \lambda + 1$
- $R_{11}^{2}= \lambda + 0(1+00)^*0$ $R_{12}^{2}= 0(1+00)^*$ $R_{13}^{2}= 0(1+00)^*(0+1)$
- $R_{21}^{2}= (1+00)^*0$ $R_{22}^{2}= (1+00)^*$ $R_{23}^{2}= (1+00)^*(0+1)$
- $R_{31}^{2}= 1(1+00)^*0$ $R_{32}^{2}= 1(1+00)^*$ $R_{33}^{2}= \lambda+1+1(1+00)^*(0+1)$
- $L = R_{12}^{3}=$
  $0(1+00)^* + 0(1+00)^*(0+1)\ (1+1(1+00)^*(0+1))^*\ 1(1+00)^*$

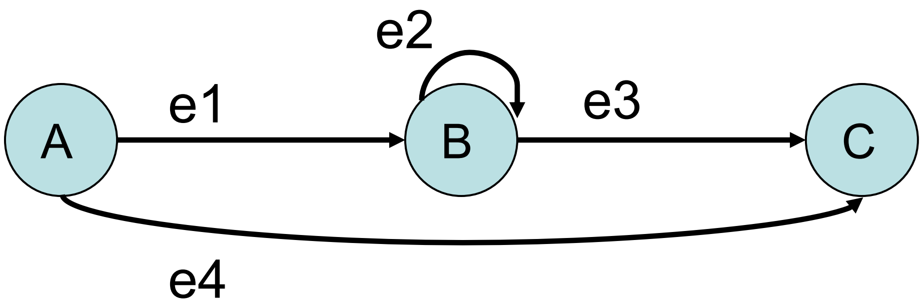
THIS IS GREAT WAY TO GET FORMAL PROOF

State Ripping Concept

- This is like the generalized automata approach you might see in Sipser and other places but with fewer arcs than text. It gets some of its motivation from R_{ij}^{k} approach as well.
- Add a new start state and add a λ–transition to existing start state
- Add a new final state q_f and insert λ–transitions from all existing final states to the new one; make the old final states non-final
- Excluding start and final states, successively pick states to remove
- For each state to be removed, change the arcs of every pair of externally entering and exiting arcs to reflect the regular expression that describes all strings that could result is such a double transition; be sure to account for loops in the state being removed. Also, or (+) together expressions that have the same start and end nodes
- When have just start and final, the regular expression that leads from start to final denotes the associated regular set

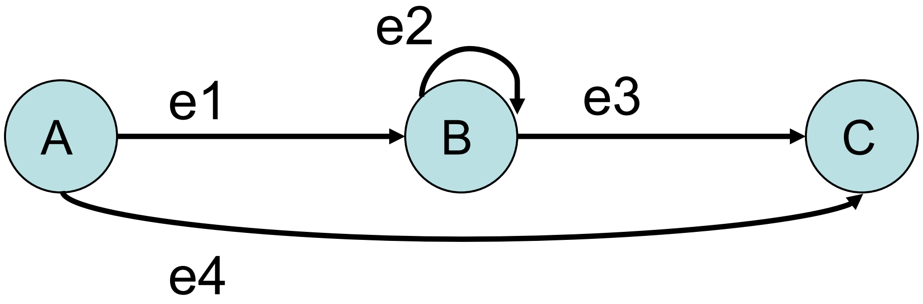
State Ripping Details

- Let B be the node to be removed
- Let e1 be the regular expression on the arc from some node A to some node B (A≠B); e2 be the expression from B back to B (or λ if there is no recursive arc); e3 be the expression on the arc from B to some other node C (C ≠B but C could be A); e4 be the expression from A to C

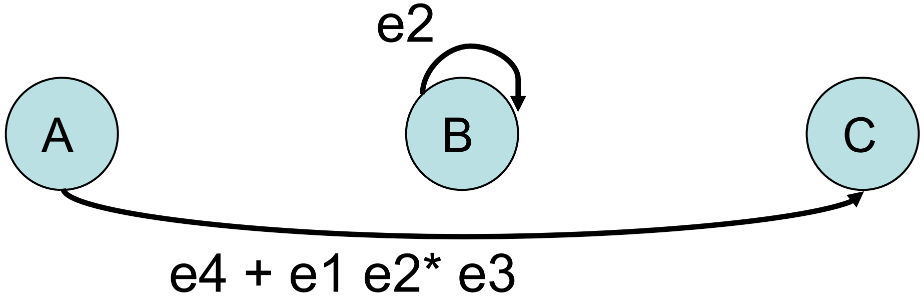


- Note that this just says, what if I allowed the path from A to C to include transitions through B, then what is new regular expression? The form is exactly what we saw in R_{ij}^{k}.

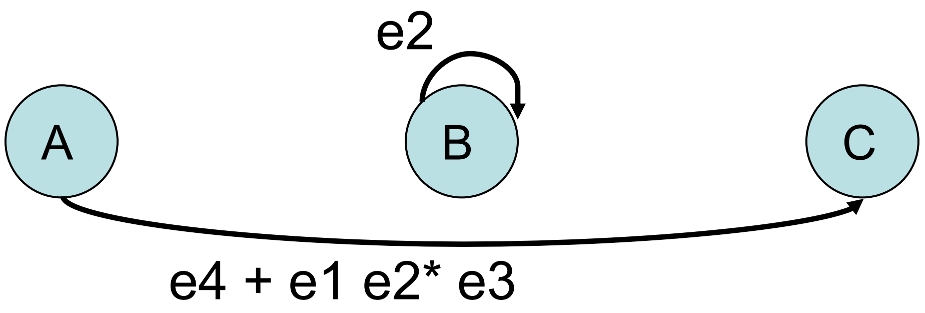
State Ripping Details



- Erase the existing arcs from A to B and A to C, adding a new arc from A to C labelled with the expression
  e4 + e1 e2* e3
- Note that all other arcs associated with A and C are untouched.

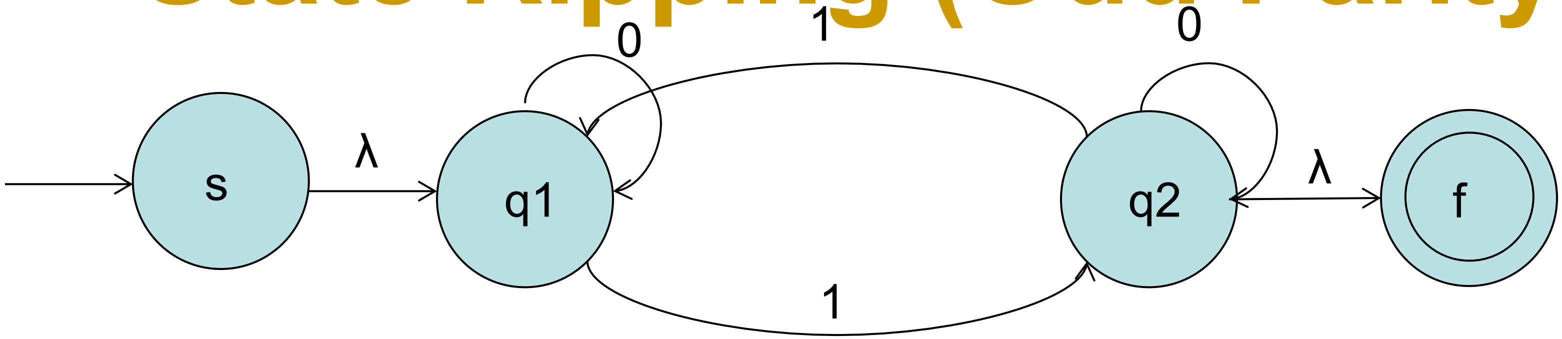


State Ripping Details

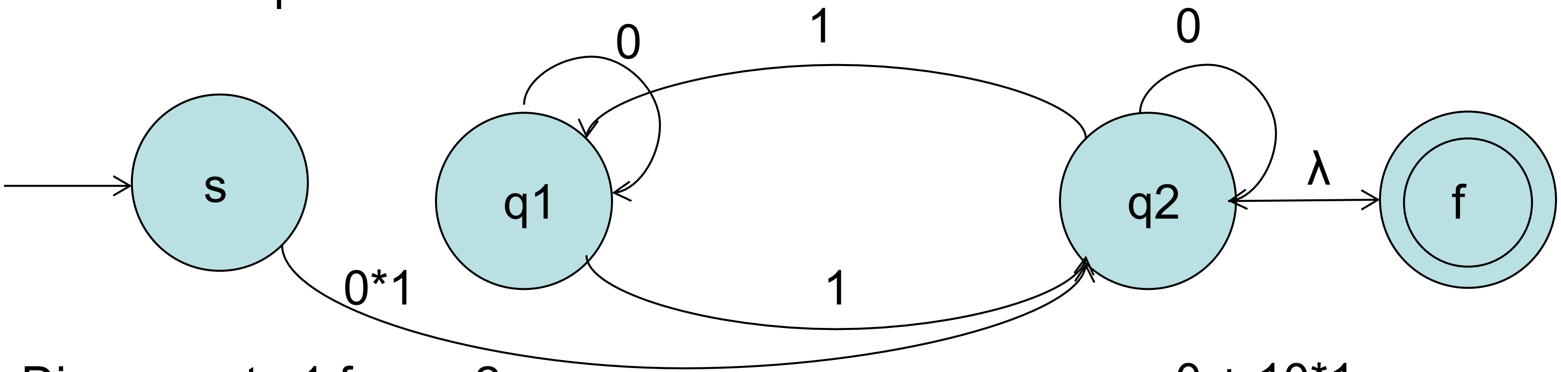


- Do this for all nodes that have edges to B until B has no more entering. edges; at this point remove B and any edges it has to other nodes and itself
- Iterate until all but the start and final nodes remain.
- The expression from start to final describes the regular set that is equivalent to the regular language accepted by the original automaton.
- Note: Your choices of the order of removal make a big difference in how hard or easy this is.

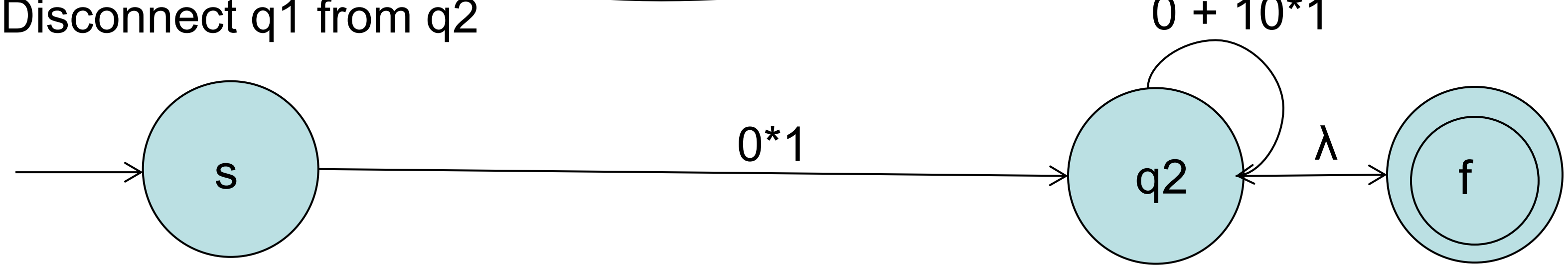
State Ripping (Odd Parity)



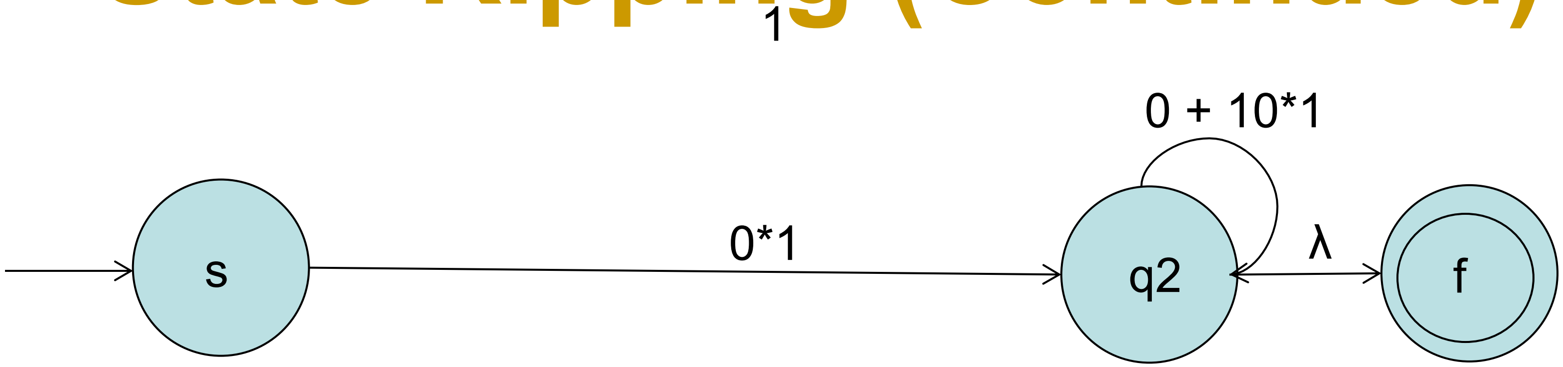
Disconnect q1 from s



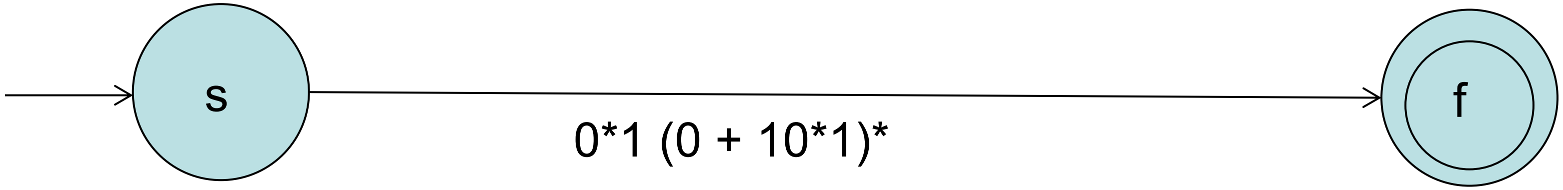
Disconnect q1 from q2



State Ripping (Continued)

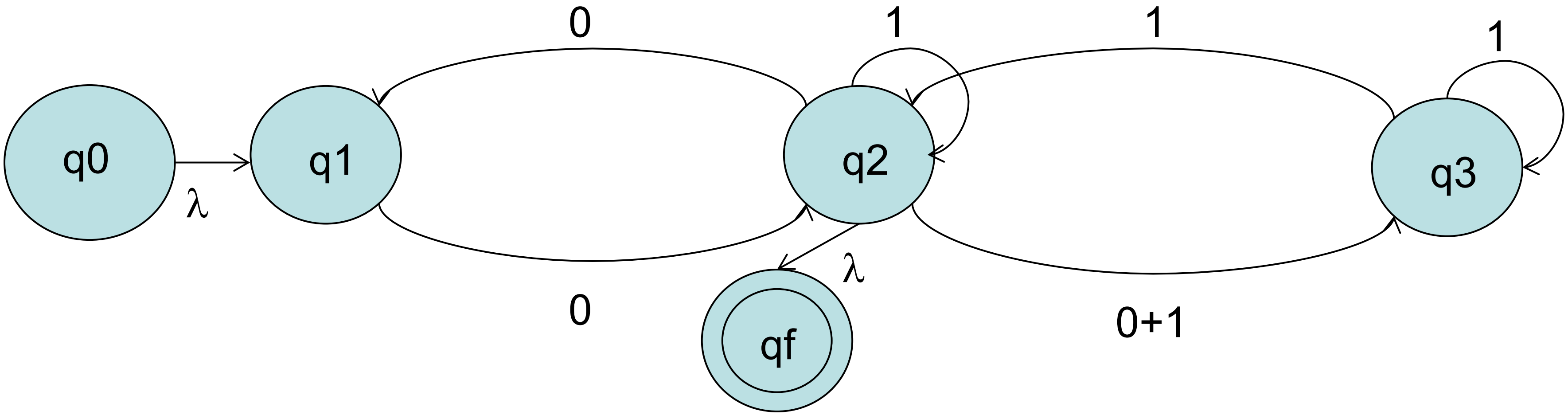


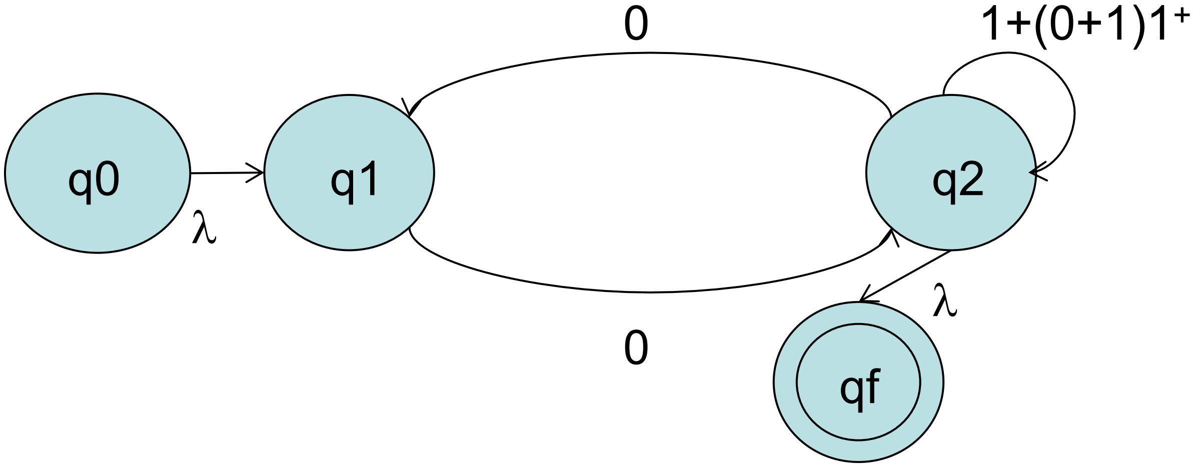
Disconnect s from q2



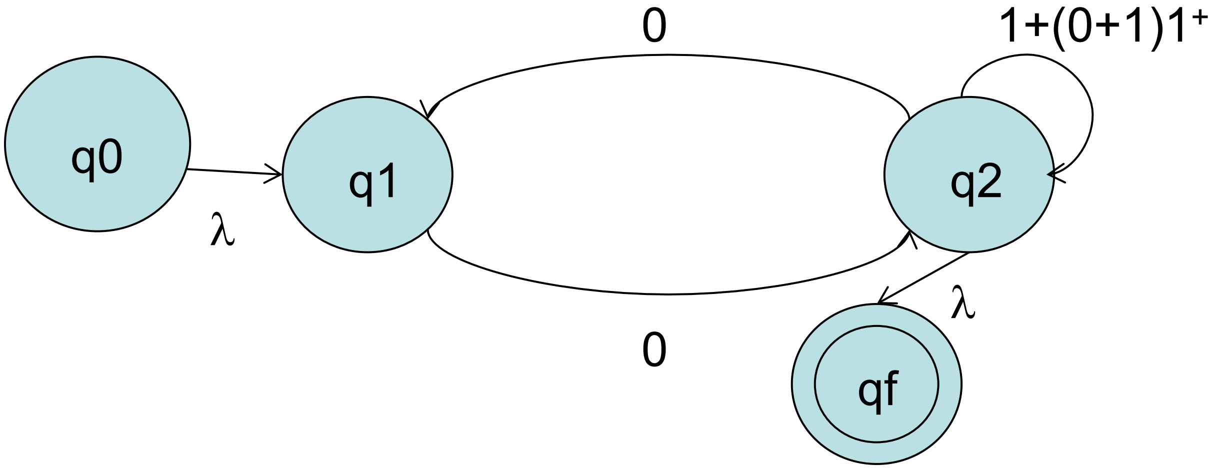
Got same regular expression as we saw with Rijk but what would happen if we ripped q2 and then q1? Try it. The expression will be different, but the set will be the same.

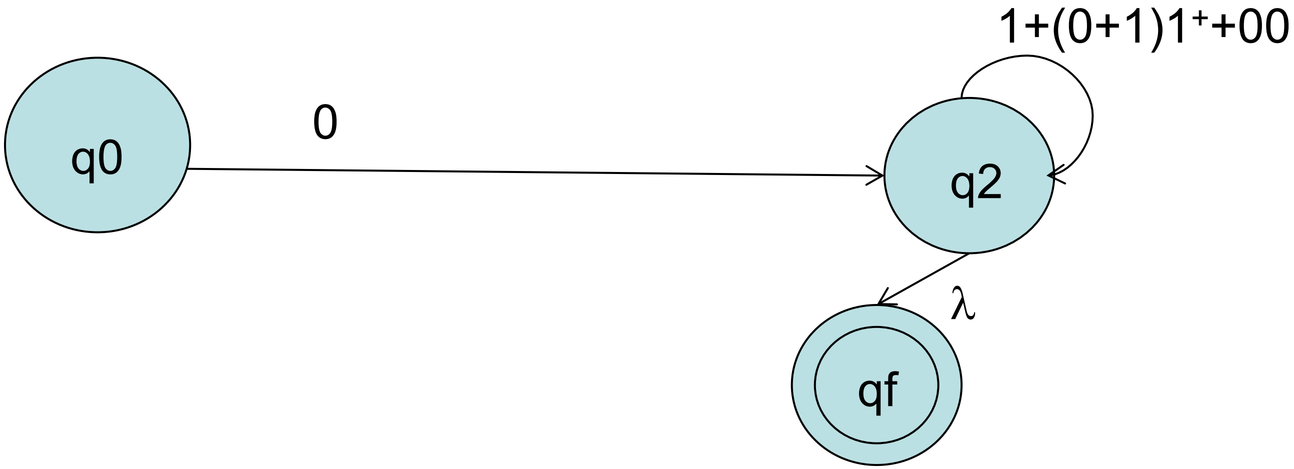
More Complex Case; Rip q3



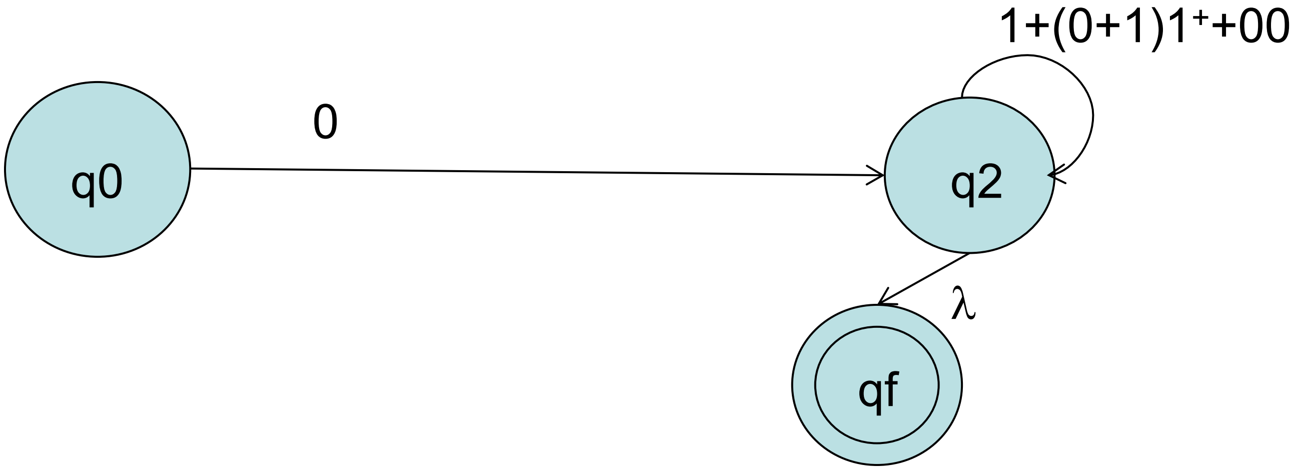


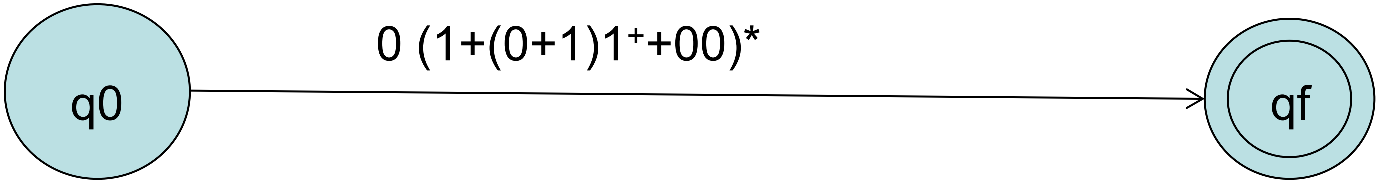
Continued; Rip q1





Continued; Rip q2





$L = 0\ (1+(0+1)1^{+}+00)^{*}$

Regular Equations (Arden)

- Assume that R, Q and P are sets such that P does not contain the string of length zero, and R is defined by
- $R = Q + RP$
- We wish to show that
- $R = QP^*$
- This is called “Arden’s Theorem/Lemma/Rule” (Google it!!)

Show QP* is a Solution

- We first show that QP* is contained in R. By definition, R = Q + RP.
- To see if QP* is a solution, we insert it as the value of R in Q + RP and see if the equation balances.
- $R = Q + QP^*P = Q(\lambda+P^*P) = Q(\lambda+P^+) = QP^*$
- Hence QP* is a solution, but not necessarily the only solution.

Uniqueness of Solution

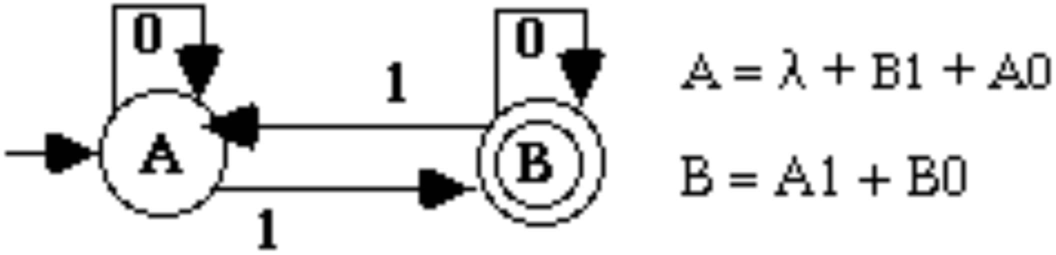
- To prove uniqueness, we show that R is contained in QP*.
- By definition, $R = Q+RP = Q+(Q+RP)P$
- $= Q+QP+RP^2 = Q+QP+(Q+RP)P^2$
- $= Q+QP+QP^2+RP^3$
- ...
- $= Q(\lambda+P+P^2+ \ldots +P^i)+RP^{i+1}$, for all $i>=0$
- Choose any w in R, where $|w| = k$. Then, from above,
- $R = Q(\lambda+P+P^2+ \ldots +P^k)+RP^{k+1}$
- but, since P does not contain the string of length zero, w is not in RP^{k+1}. But then w is in
- $Q(\lambda+P+P^2+ \ldots +P^k)$ and hence w is in QP*.

Reg. Eq. Process

- Let $\mathcal{A}$ = (Q,Σ,δ,q_1,F) be a DFA
- For each pair of states, A,B in Q, where for some input 'a', δ(B,a) = A, include the term Ba in the right-side of the equation for A, that is, A = … + Ba
  This just says that any solution for A must include the solution for B followed by an 'a'.
- If A is the start state, then include λ as one of the terms as well, that is A = λ + …
  This just says that any solution for A must include λ since A is the start state.

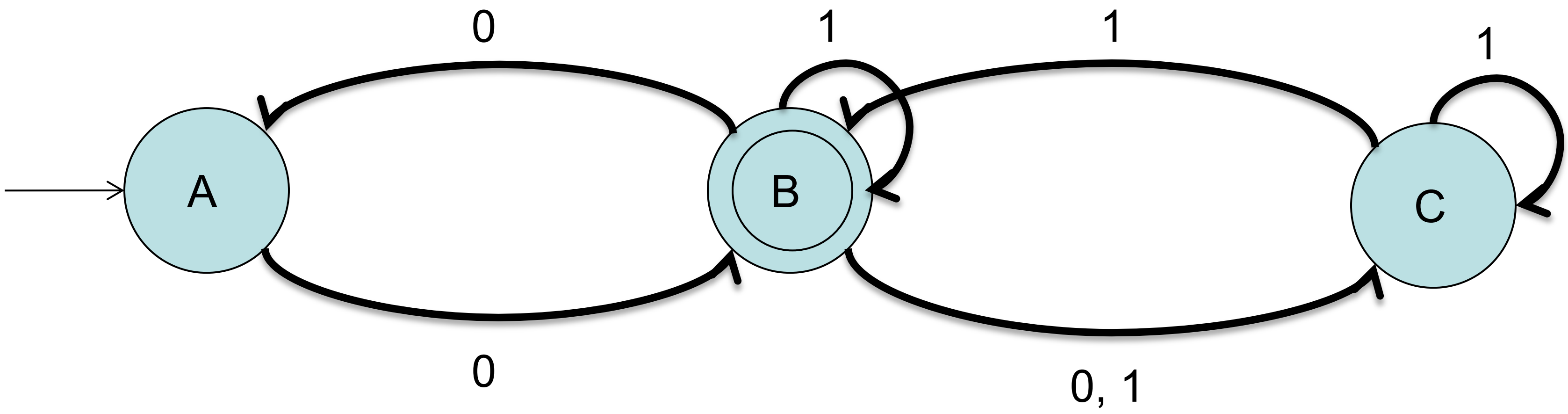
Example

- We use the above to solve simultaneous regular equations. For example, we can associate regular expressions with finite-state automata as follows
- Hence,
- For A, $Q=\lambda+B1$; $P=0$
  $A = QP^* = (\lambda+B1)0^*$
  $= B10^* + 0^*$



- $B = B10^*1 + B0 + 0^*1$
  For B, $Q=0^*1$; $P= B10^*1 + B0 = B(10^*1 + 0)$
- and therefore
- $B = 0^*1(10^*1 + 0)^*$
- Note: This technique fails if there are self lambda transitions.

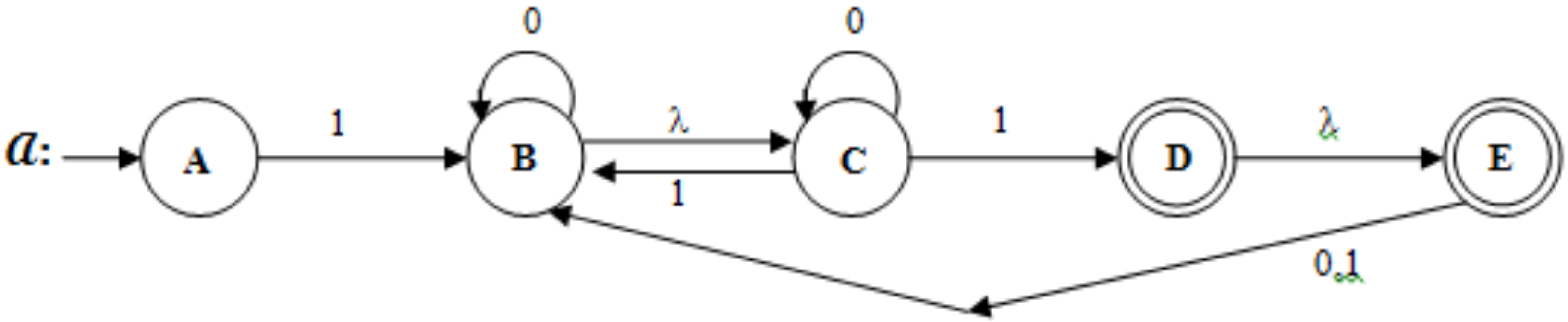
Using Regular Equations



$A = \lambda + B0$
$B = A0 + C1 + B1$
$C = B(0+1) + C1; C = B(0+1)1^*$
$B = 0 + B00 + B(0+1)1^+ + B1$
$B = 0 + B\ (00+(0+1)\ 1^+ + 1); B = 0(00 +(0+1)1^+ + 1)^* = 0\ (1+(0+1)1^++00)^*$

This is same form as with state ripping. It won’t always be so.

Use Reg. Eq. to Solve for D + E



A = λ ; B = A1 + C1 + E(0+1) + B0 ; C = B + C0 ; D = C1 ; E = D

C = B0*

D = C1 = B0*1; also, since E = D, E = B0*1

B = A1 + C1 + E(0+1) + B0 = 1 + B0*1 + B0*1(0+1) + B0 = 1 + B0*1(0+1) + B(0*1 + 0)

= 1(0*1(0+1) + 0*1 + 0)*

C = B0* = 1(0*1(0+1) + 0*1 + 0)* 0*

D = C1 = 1(0*1(0+1) + 0*1 + 0)* 0*1 = 1(0*1(0+1+λ) + 0)* 0*1 = 1(0*1(0+1+λ) + 0)* 1

E = D so the language is denoted by 1(0*1(0+1+λ) + 0)* 1

Practice NFAs

- Write NFAs for each of the following
  - $(111 + 000)^+$
  - $(0+1)^* \, 101 \, (0+1)^+$
  - $(1 \, (0+1)^* \, 0) + (0 \, (0+1)^* \, 1)$
- Convert each NFA you just created to an equivalent DFA.

DFAs to REs

- For each of the DFAs you created for the previous page, use ripping of states and then regular equations to compute the associated regular expression. Note: You obviously ought to get expressions that are equivalent to the initial expressions.

State Minimization

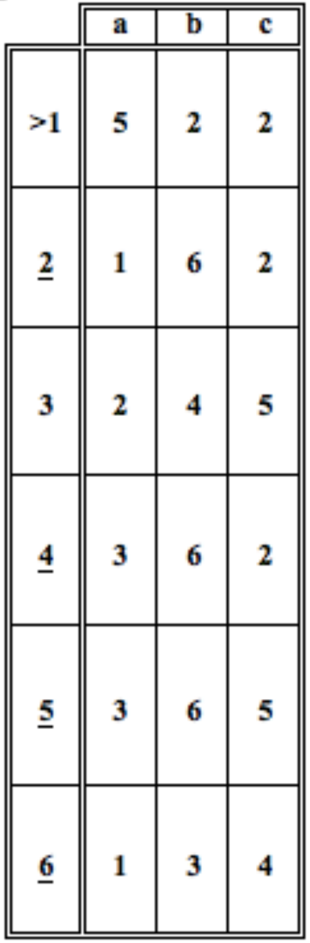
Minimum State DFAs

State Minimization

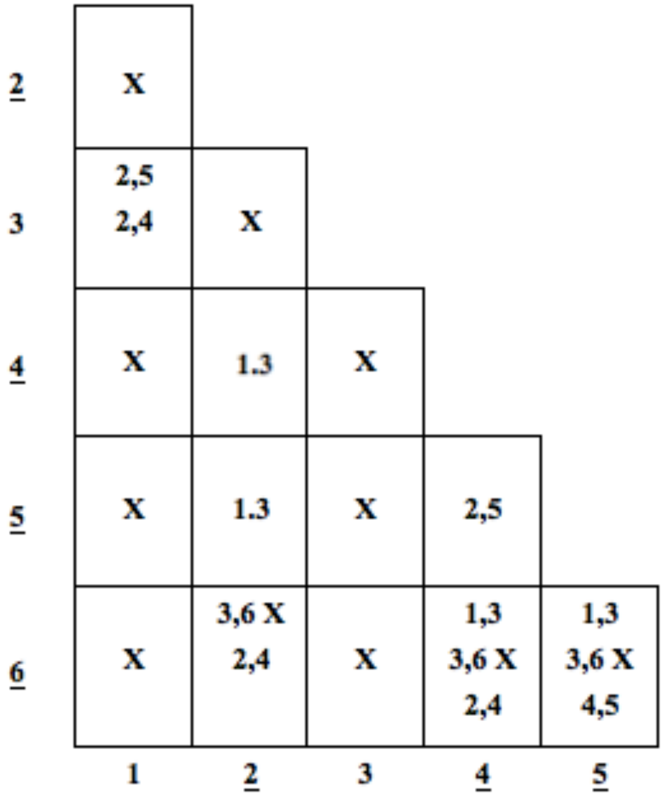
- Sipser text makes it an assignment on Page 299 in Edition 2.
- This is too important to defer, IMHO.
- First step is to remove any state that is unreachable from the start state; a depth first search rooted at start state will identify all reachable states
- One seeks to merge compatible states – states q and s are compatible if, for all strings x, $\delta^*(q,x)$ and $\delta^*(s,x)$ are either both an accepting or both rejecting states
- One approach is to discover incompatible states – states q and s are incompatible if there exists a string x such that one of $\delta^*(q,x)$ and $\delta^*(s,x)$ is an accepting state and the other is not
- There are many ways to approach this but my favorite is to do incompatible states via an n by n lower triangular matrix

Sample Minimization

- This uses a transition table
- Just an X denotes Immediately incompatible
- Pairs are dependencies for compatibility
- If a dependent is incompatible, so are pairs that depend on it
- When done, any not x--ed out are compatible
- Here, new states are <1,3>, <2,4,5>, <6>; <1,3> is start and not accept; others are accept
- Write new diagram

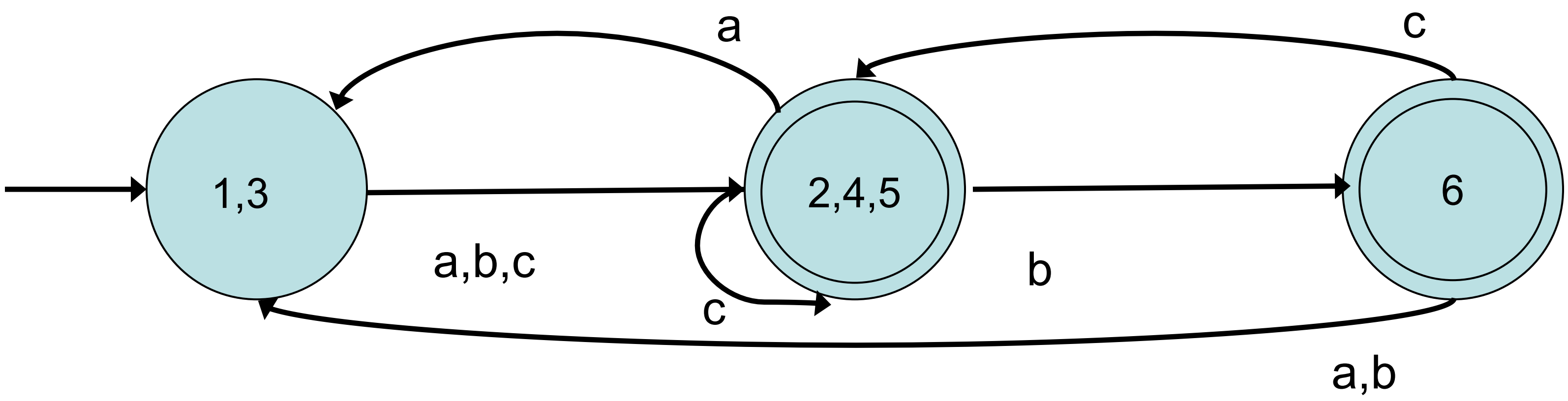


	a	b	c
>1	5	2	2
2	1	6	2
3	2	4	5
4	3	6	2
5	3	6	5
6	1	3	4



2	X				
3	2,5 2,4	X			
4	X	1.3	X		
5	X	1.3	X	2,5	
6	X	3,6 X 2,4	X	1,3 3,6 X 2,4	1,3 3,6 X 4,5
	1	2	3	4	5

Min DFA



Closure Properties

Regular Languages

Reversal of Regular Sets

- It is easier to do this with regular sets than with NFAs
- Let E be some arbitrary expression; E^R is formed by
  - Primitives: $\varnothing^R=\varnothing$ $\lambda^R=\lambda$ $a^R=a$
  - Closure:
    - $(A \cdot B)^R = (B^R \cdot A^R)$
    - $(A + B)^R = (A^R + B^R)$
    - $(A^*)^R = (A^R)^*$
- Challenge: How would you do this with FSA models?
  - Start with DFA; change all final to start states; change start to a final state; and reverse edges (now it's an NFA)
  - Note that this creates multiple start states; can create a new start state with λ-transitions to multiple starts

Substitution

- A substitution is a function, f, from each member, a, of an alphabet, Σ, to a language L_a
- Regular languages are closed under substitution of regular languages (i.e., each L_a is regular)
- Easy to prove by replacing each member of $a \in \Sigma$ in a regular expression for a language L with the regular expression for L_a
- A homomorphism is a substitution where each L_a is a single string

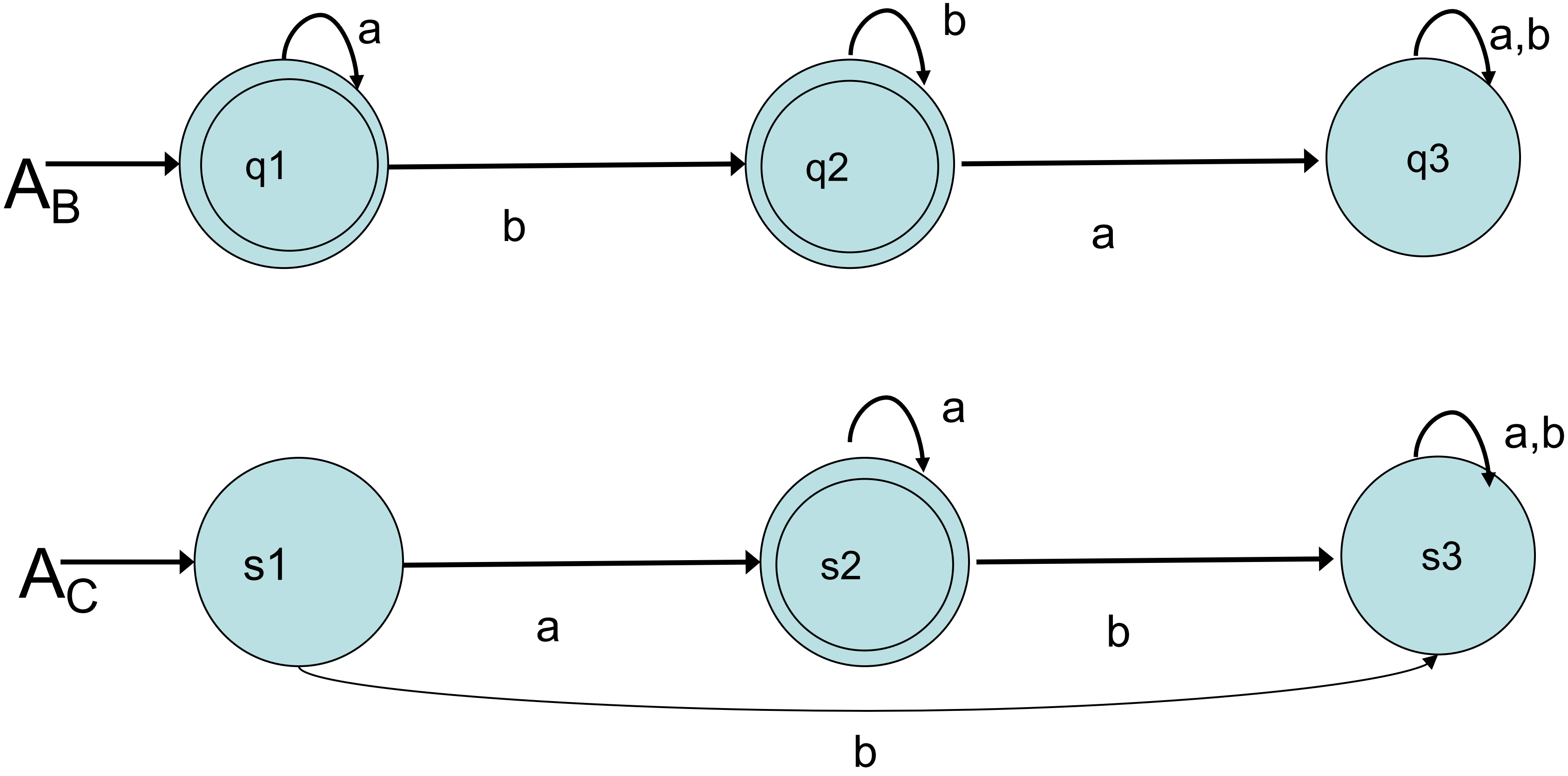
Quotient with Regular Sets

- Quotient of two languages B and C, denoted B/C, is defined as
  $B/C = \{ x \mid \exists y \in C \text{ where } xy \in B \}$
- Let B be recognized by DFA
  $A_B = (Q_B,\Sigma,\delta_B,q_{1B},F_B)$ and C by
  $A_C = (Q_C,\Sigma,\delta_C,q_{1C},F_C)$
- Define the recognizer for B/C by
  $A_{B/C} = (Q_B \cup Q_B \times Q_C,\Sigma,\delta_{B/C},q_{1B}, F_B \times F_C)$
  $\delta_{B/C}(q,a) = \{\delta_B(q,a)\}$ $\quad a \in \Sigma, q \in Q_B$
  $\delta_{B/C}(q,\lambda) = \{<q,q_{1C}>\}$ $\quad q \in Q_B$
  $\delta_{B/C}(<q,p>,\lambda) = \{<\delta_B(q,a),\delta_C(p,a)>\}$ $\quad a \in \Sigma, q \in Q_B, p \in Q_C$
- The basic idea is that we simulate B and then randomly decide it has seen x and continue by looking for y, simulating B continuing after x but with C starting from scratch and both making believe they see the same character at every stage (none actually is seen)

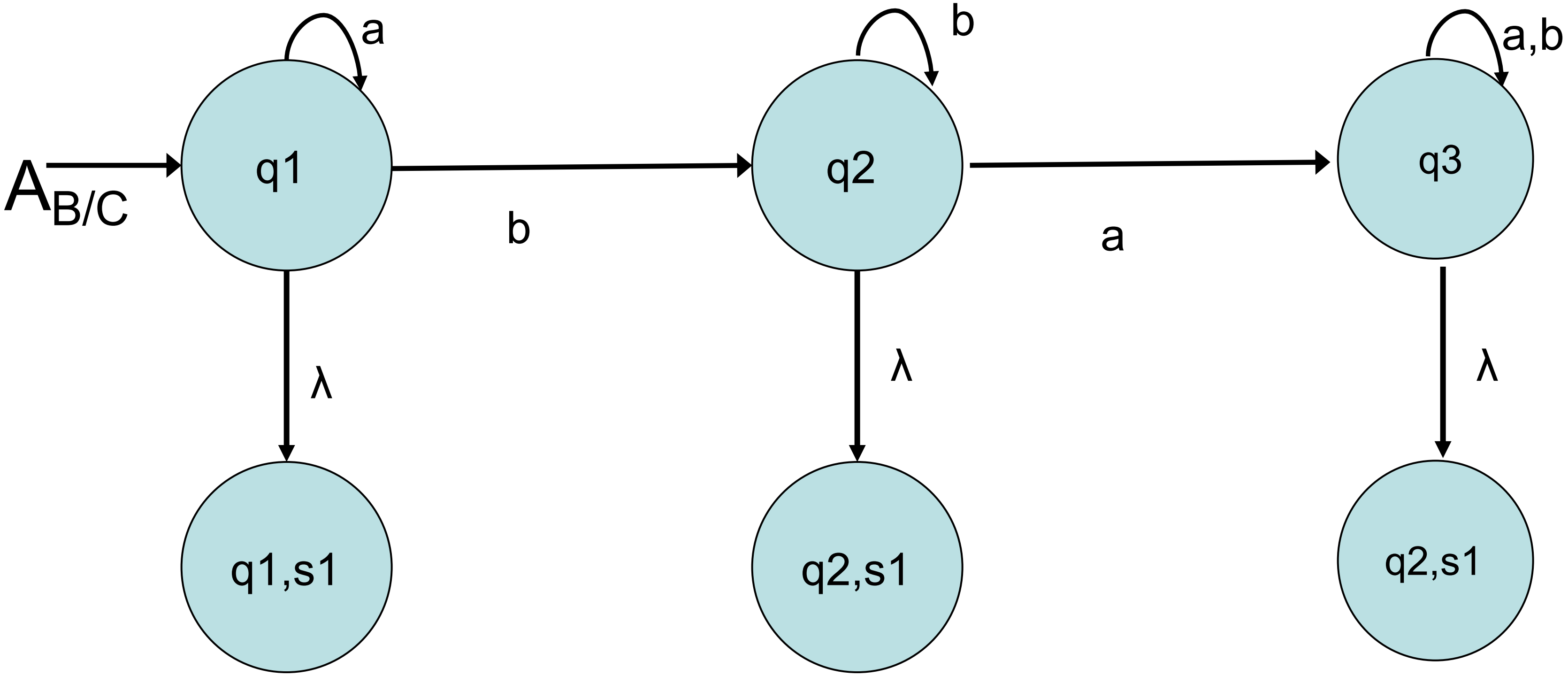
Example of B/C via NFA

- Let $B = a^*b^*$; $C = a^+$
- $B/C = a^*$ as C must remove at least one a and will not apply if there are any b’s
- $A_B = (\{q1,q2,q3\}, \{a,b\}, \delta_B, q1, \{q1,q2\})$
  $\delta_B(q1,a) = q1$; $\delta_B(q1,b) = q2$; $\delta_B(q2,a) = q3$; $\delta_B(q2,b) = q2$;
  $\delta_B(q3,a) = q3$; $\delta_B(q3,b) = q3$
- $A_C = (\{s1,s2,s3\}, \{a,b\}, \delta_C, s1, \{s2\})$
  $\delta_C(s1,a) = s2$; $\delta_C(s1,b) = s3$; $\delta_C(s2,a) = s2$; $\delta_C(s2,b) = s3$;
  $\delta_C(s3,a) = s3$; $\delta_C(s3,b) = s3$
- $A_{B/C} = (\{q1,q2,q3,<q1,s1>,<q1,s2>,<q1,s3>,<q2,s1>>,<q2,s2>,<q2,s3>,$
  $<q3,s1>,<q3,s2>, <q3,s3>\}$ $\{a,b\}$, $\delta_{B/C}$, $q1$, $\{<q1,s2>,<q2,s2>\})$
  $\delta_{B/C}(q,c) = \{\delta_B(q,c)\}$, $q \in \{q1,q2,q3\}$; $c \in \{a,b\}$ // read ’x”
  $\delta_{B/C}(q,\lambda) = \{<q,s1>\}$, $q \in \{q1,q2,q3\}$; // jump to synthesize y
  $\delta_{B/C}(<q,s>,\lambda) = \{<\delta_B(q,c),\delta_C(s,c)>\}$, $c \in \{a,b\}, q \in \{q1,q2,q3\}, s \in \{s1,s2,s3\}$,

Example Worked Out #1

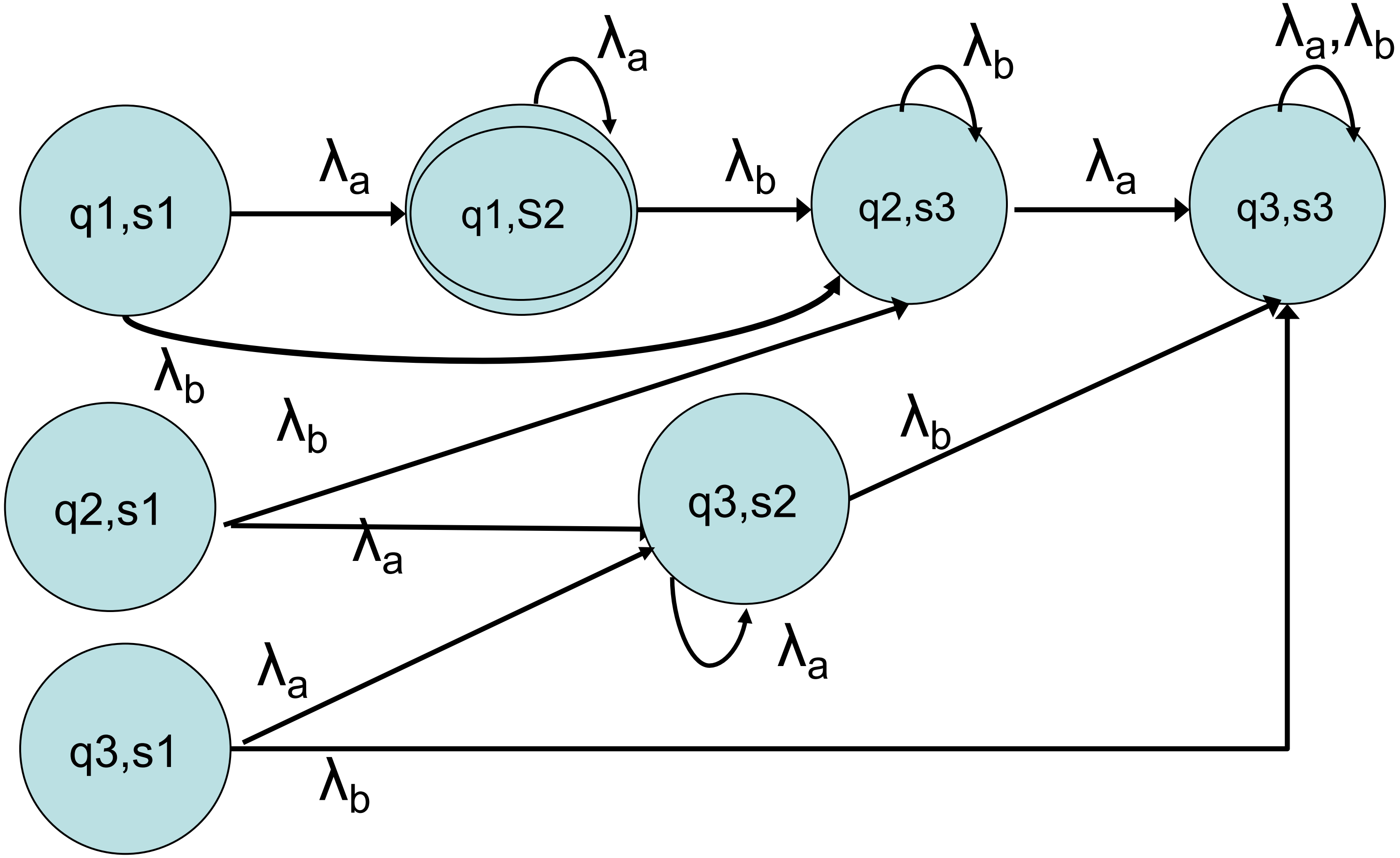


Example Worked Out #2



See next page

Example Worked Out #3

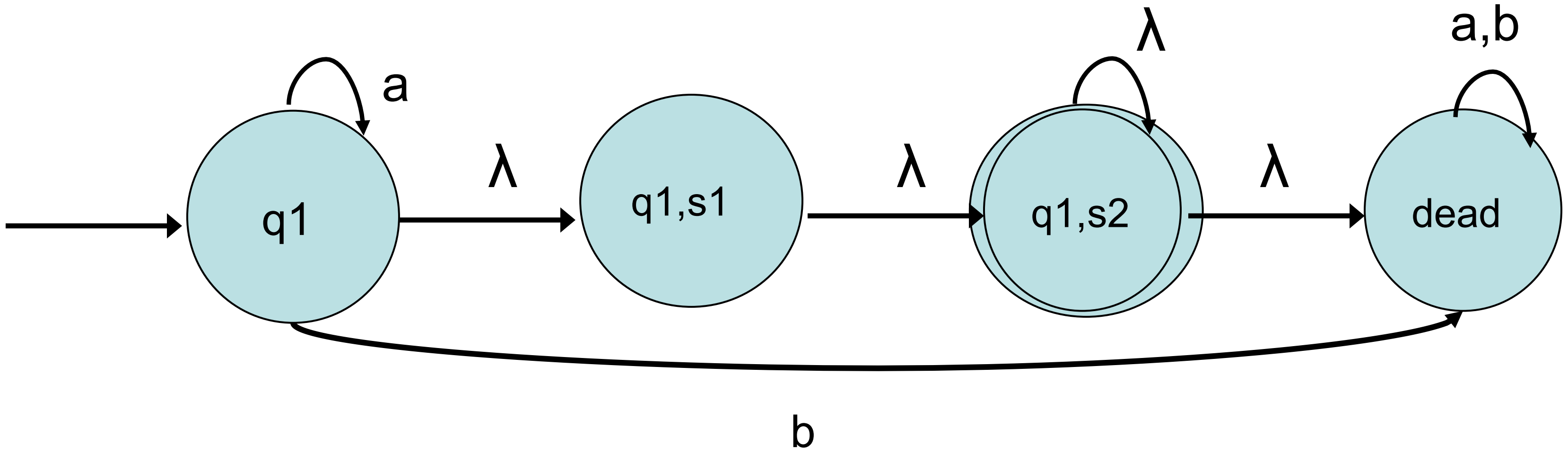


The lambda subscripts indicate the make-believe characters we are”consuming”

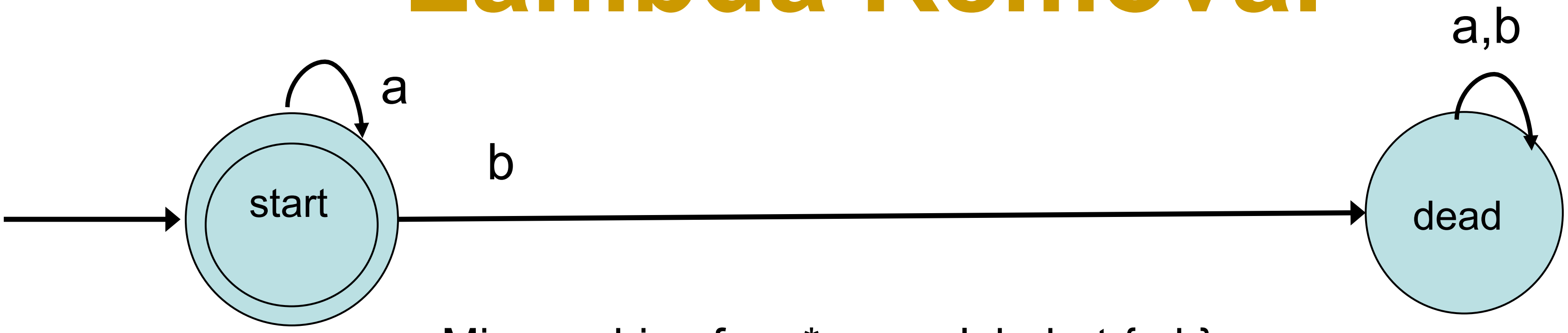
Quick State Reduction

- We had the possibility of 3 + 9 = 12 states
- Only 7 were reachable from start
- Of these, only three can lead to a final state
  q1, <q1,s1>, <q1,s2>
- We will need a dead state for all other paths, so let’s look at what we have, having done some obvious reductions

Reduced NFA



Lambda Removal



Min machine for a* over alphabet {a,b}

Implications of Quotient

- PREFIX(L) = $\{ x \mid \exists y \in \Sigma^* \text{ where } xy \in L \}$ = L / Σ^*
- SUFFIX(L) = $\{ y \mid \exists x \in \Sigma^* \text{ where } xy \in L \}$ = $(L^R / \Sigma^*)^R$
- SUBSTRING(L) = $\{ y \mid \exists x,z \in \Sigma^* \text{ where } xyz \in L \}$ = PREFIX(SUFFIX(L)) = SUFFIX(PREFIX(L))
- So, Regular Languages (Sets) are closed under Prefix, Suffix, and Substring
- But is there a strategy that encompasses quotient and all above and may work for other classes of languages?

Quotient Again

- Assume some class of languages, $\mathfrak{C}$, is closed under concatenation, intersection with regular and substitution of members of $\mathfrak{C}$, show $\mathfrak{C}$ is closed under Quotient with Regular
- L/R = { x |∃y∈R where xy∈L }, R and L over Σ
  - Define Σ' = { a' | a∈Σ }
  - Let h(a) = a; h(a') = λ where a∈Σ
  - Let g(a) = a' where a∈Σ
  - Let f(a) = {a,a'} where a∈Σ
  - L/R = h(f(L) ∩ (Σ* · g(R)))

Applying Meta Approach

- INIT(L) = PREFIX(L) = { x |∃y∈Σ* where xy∈L }
  - INIT(L) = h(f(L) ∩ (Σ* · g(Σ*)))
  - Also INIT(L) = L / Σ*
- LAST(L) = SUFFIX(L) = { y |∃x∈Σ* where xy∈L }
  - LAST(L) = h(f(L) ∩ (g(Σ*) · Σ*))
- MID(L) = SUBSTRING(L) =
  { y |∃x,z∈Σ* where xyz∈L }
  - MID(L) = h(f(L) ∩ (g(Σ*) · Σ* · g(Σ*)))
- EXTERIOR(L) = { xz |∃y∈Σ* where xyz∈L }
  - EXTERIOR(L) = h(f(L) ∩ (Σ* · g(Σ*) · Σ*))

Substitution Examples

- Consider alphabet {a,b}
- Consider primed version g({a,b}) = {a’,b’}
- Note that g(aba) = a’b’a’
- f(aba) = {aba,aba’,ab’a,ab’a’,a’ba,a’ba’,a’b’a,a’b’a’}
- h(f(aba)) = {aba,ab,aa,a,ba,b,λ}

Back to Quotient

- $f(L) = \{ x_1x_2..x_k \mid a_1a_2..a_k \in L \}$ and each x_i is either a_i or a_i'
- $\Sigma^*g(R) = \{ x\ g(y) \mid x \in \Sigma^* \text{ and } y \in R \} =$
  $\{ xy' \mid x \in \Sigma^* \text{ and } y \in R \}$
- $f(L) \cap \Sigma^*g(R) = \{ xy' \mid xy \in L \text{ and } y \in R \}$
- $h(f(L) \cap \Sigma^*g(R)) = \{ x \mid \exists y \in R \text{ where } xy \in L \}$
  $= L/R$
- Since Regular are closed under substitution, intersection, and concatenation, they are also closed under quotient

Making Life Easy

- The key in proving closure is to always try to identify the “best” equivalent formal model for regular sets when trying to prove a particular property
- For example, how could you even conceive of proving closure under intersection and complement in regular expression notations?
- Note how much easier quotient is when have closure under concatenation, and substitution and intersection with regular languages than showing in FSA notation

Reachable and Reaching

- Reachable*from*(q) = { p | ∃w ∋ δ*(q,w)=p }
  - Just do depth first search from q, marking all reachable states. Works for NFA as well.
- Reaching*to*(q) = { p | ∃w ∋ δ*(p,w)=q }
  - Do depth first search from q, going backwards on transitions, marking all reaching states. Works for NFA as well.

Min and Max

- Min(L) = { w | w∈L and no proper prefix of w is in L } = { w | w∈L and if w=xy, x∈Σ*, y∈Σ^+ then x∉L}
- Max(L) = { w | w∈L and w is not the proper prefix of any word in L } = { w | w∈L and if y∈Σ^+ then wy∉L }
- Examples:
  - Min(0(0+1)*) = {0}
  - Max(0(0+1)*) = {}
  - Min(01 + 0 + 10) = {0,10}
  - Max(01 + 0 + 10) = {01,10}
  - Min({$a^ib^jc^k$ | i ≤ k or j ≤ k}) = {$a^ib^jc^k$ | | i,j ≥0, k = min(i, j)}
  - Max({$a^ib^jc^k$ | i ≤ k or j ≤ k}) = {} because k has no bound
  - Min({$a^ib^jc^k$ | i ≥ k or j ≥ k}) = {λ}
  - Max({$a^ib^jc^k$ | i ≥ k or j ≥ k}) = {$a^ib^jc^k$ | | i,j ≥0, k = max(i, j)}

Regular Closed under Min

- Assume L is regular then Min(L) is regular
- Let L= *L*(A), where A = (Q,Σ,δ,q_0,F) is a DFA with no state unreachable from q_0
- Define $A_{min} = (Q\cup\{dead\},\Sigma,\delta_{min},q_0,F)$, where for $a\in\Sigma$ $\delta_{min}(q,a) = \delta(q,a)$, if $q\in Q\text{-}F$; $\delta_{min}(q,a) = dead$, if $q\in F$; $\delta_{min}(dead,a) = dead$

The reasoning is that the machine A_{min} accepts only elements in L that are not extensions of shorter strings in L. By making it so transitions from all final states in A_{min} go to the new "dead" state, we guarantee that extensions of accepted strings will not be accepted by this new automaton.

Therefore, Regular Languages are closed under Min.

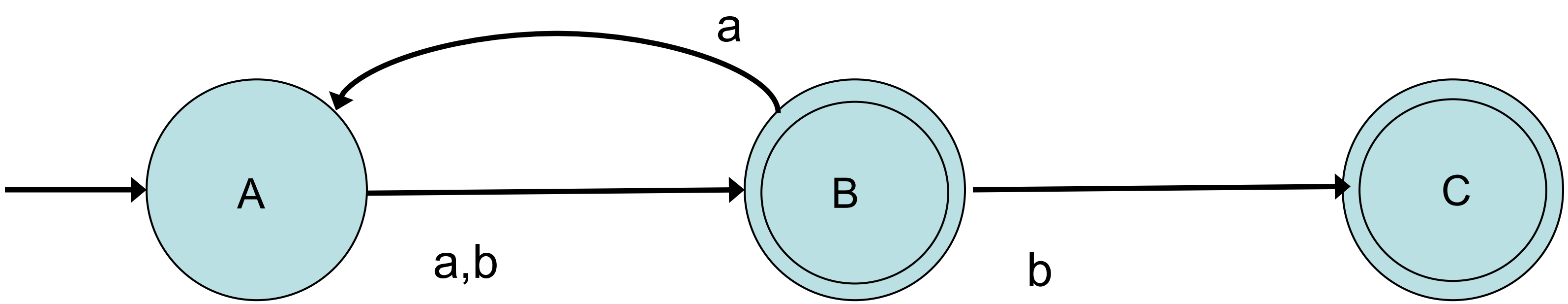
Regular Closed under Max

- Assume L is regular then Max(L) is regular
- Let L= L(A), where A = (Q,Σ,δ,q_0,F) is a DFA with no state unreachable from q_0
- Define $A_{max} = (Q,\Sigma,\delta,q_0,F_{max})$, where
  $F_{max}= \{ f \mid f\in F$ and Reachable*from*$^+(f)\cap F=\Phi \}$
  where Reachable*from*$^+(q) = \{ p \mid \exists w \ni |w|>0$ and $\delta(q,w) = p \}$

The reasoning is that the machine A_{max} accepts only elements in L that cannot be extended. If there is a non-empty string that leads from some final state f to any final state, including f, then f cannot be final in A_{max}. All other final states can be retained.
The inductive definition of Reachable*from*$^+$ is:
1. Reachable*from*$^+(q)$ contains $\{ s \mid$ there exists an element of Σ, a, such that $\delta(q,a) = s \}$
2. If s is in Reachable*from*$^+$ (q) then Reachable*from*$^+$ (q) contains
   $\{ t \mid$ there exists an element of Σ, a, such that $\delta(s,a) = t \}$
3. No other states are in Reachable*from*$^+(q)$

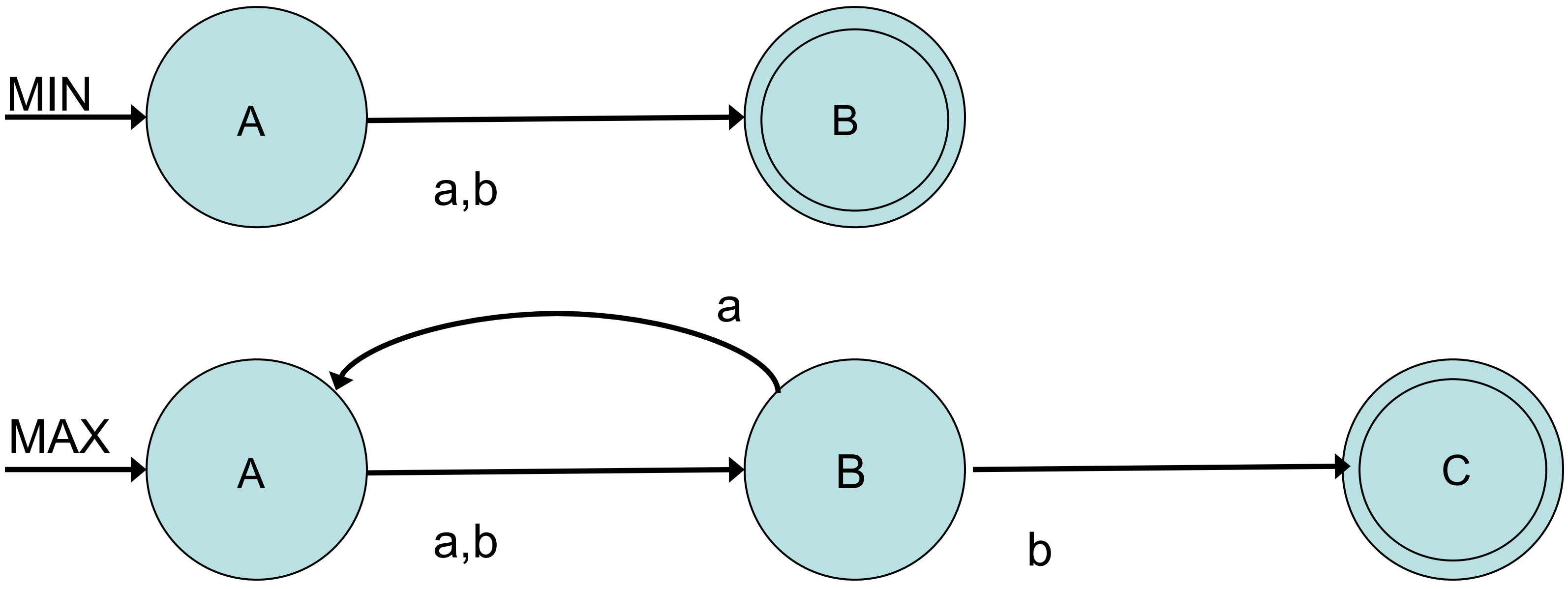
Therefore, Regular Languages are closed under Max.

Regular Expression for L



A = λ+Ba   B = A(a+b)   C = Bb
B = a+b + Ba(a+b) = (a+b)(aa+ab)*
C = (a+b)(aa+ab)*b
L = (a+b)(aa+ab)* (λ+b)
Min(L) = a+b   Max(L) = (a+b)(aa+ab)*b

Min(L) and Max(L)



Min(L) = a+b   Max(L) = (a+b)(aa+ab)*b

Pumping Lemma for Regular Languages

What is not a Regular Language

Pumping Lemma Concept

- Let A = (Q,Σ,δ,q_1,F) be a DFA, where $Q = \{q_1,q_2, \ldots , q_N\}$
- The “pigeon-hole principle” tells us that whenever we visit N+1 or more states, we must visit at least one state more than once (loop)
- Any string, w, of length N or greater leads to us making N transitions after visiting the start state, and so we visit at least one state more than once when reading w

Pumping Lemma For Regular

- Theorem: Let L be regular then there exists an N>0 such that, if w $\in$ L and $|w| \geq N$, then w can be written in the form xyz, where $|xy| \leq N$, $|y|>0$, and for all $i \geq 0$, $xy^iz \in L$
- This means that interesting regular languages (infinite ones) have a very simple self-embedding property that occurs early in long strings

Pumping Lemma Proof

- If L is regular then it is recognized by some DFA, $A=(Q,\Sigma,\delta,q_0,F)$. Let $|Q| = N$ states. For any string w, such that $|w| \geq N$, A must make N+1 state visits to consume its first N characters, followed by |w|-N more state visits.
- In its first N+1 state visits, A must enter at least one state two or more times.
- Let $w = v_1 \ldots v_j \ldots v_k \ldots v_m$, where $m =|w|$, and $\delta(q_0,v_1 \ldots v_j)=\delta(q_0,v_1 \ldots v_k)$, $k > j$, and let this state represent the first one repeated while A consumes w.
- Define $x = v_1 \ldots v_j$, $y = v_{i+1} \ldots v_k$, and $z = v_{k+1} \ldots v_m$. Clearly w=xyz. Moreover, since $k > j$, $|y| > 0$, and since $k \leq N$, $|xy| \leq N$.
- Since A is deterministic, $\delta(q_0,xy)=\delta(q_0,xy^i)$, for all $i \geq 0$.
- Thus, if $w \in L$, $\delta(q_0,xyz) \in F$, and so $\delta(q_0,xy^iz) \in F$, for all $i \geq 0$.
- Consequently, if $w \in L$, $|w|\geq N$, then w can be written in the form xyz, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^iz \in L$.

Lemma's Adversarial Process

- Assume L = $\{a^n b^n \mid n>0\}$ is regular
- P.L.: Provides $N > 0$
  - We CANNOT choose N; that's the P.L.'s job
- Our turn: Choose $a^N b^N \in L$
  - We get to select a string in L
- P.L.: $a^N b^N = xyz$, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^i z \in L$
  - We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose $i = 0$.
  - We have the power here
- P.L: $a^{N-|y|} b^N \in L$; just a consequence of P.L.
- Our turn: $a^{N-|y|} b^N \notin L$; just a consequence of L's structure
- CONTRADICTION, so L is NOT regular

xwx is not Regular (PL)

- **L = { x w x | x,w∈{a,b}+ } :**
- Assume that L is Regular.
- PL: Let N > 0 be given by the Pumping Lemma.
- YOU: Let s be a string, s ∈ L, such that s = $a^N baa^N b$
- PL: Since s ∈ L and |s| ≥ N, s can be split into 3 pieces, s = xyz, such that |xy| ≤ N and |y| > 0 and ∀ i ≥ 0 xy^iz ∈ L
- YOU: Choose i = 2 (**NOTE: for i=0 there is no conflict**)
- PL: xy^2z = xyyz ∈ L
- Thus, $a^{N + |y|}baa^Nb$ would be in L, but this is not so since N+|y| > N
- We have arrived at a contradiction.
- Therefore, L is not Regular.

$a^{Fib(k)}$ is not Regular (PL)

- **L = {$a^{Fib(k)}$ | k>0} :**
- Assume that L is regular
- Let N be the positive integer given by the Pumping Lemma
- Let *s* be a string **s = $a^{Fib(N+3)}$∈ L (**assume skip seeds to get **2, 3, 5, 8, 13, …)**
- Since *s* ∈ L and |s| ≥ N (Fib(N+3)>N in all cases; s is split by PL into xyz, where |xy| ≤ N and |y| > 0 and for all i ≥ 0, xy^iz ∈ L
- We choose i = 2; by PL: xy^2z = xyyz∈ L
- Thus, $a^{Fib(N+3)+|y|}$ would be ∈ L. This means that there is a Fibonacci number between Fib(N+3) and Fib(N+3)+N, but the smallest Fibonacci greater than Fib(N+3) is Fib(N+3)+Fib(N+2) and Fib(N+2)>N
  This is a contradiction; therefore, L is not regular ■
- Note: Using values less than N+3 could be dangerous because N could be 1 and both Fib(2) and Fib(3) are within N of predecessor.

Pumping Lemma Problems

- Use the Pumping Lemma to show each of the following is not regular
  - $\{ 0^m 1^{2n} \mid m \le n \}$
  - $\{ ww^R \mid w \in \{a,b\}^+ \}$
  - $\{ 1^{n^2} \mid n > 0 \}$
  - $\{ ww \mid w \in \{a,b\}^+ \}$

  - What about $\{ wxw^R \mid w,x \in \{a,b\}^+ \}$?

State Minimization

We now want to show, for any
Regular Language R,

the minimum state DFA is unique

Myhill-Nerode Theorem

Myhill-Nerode Theorem

The following are equivalent:

1. L is accepted by some DFA
2. L is the union of some of the classes of a right invariant equivalence relation, R, of finite index.
3. The specific right invariance equivalence relation R_L where x R_L y iff $\forall z$ [xz $\in$ L iff yz $\in$ L]
   has finite index

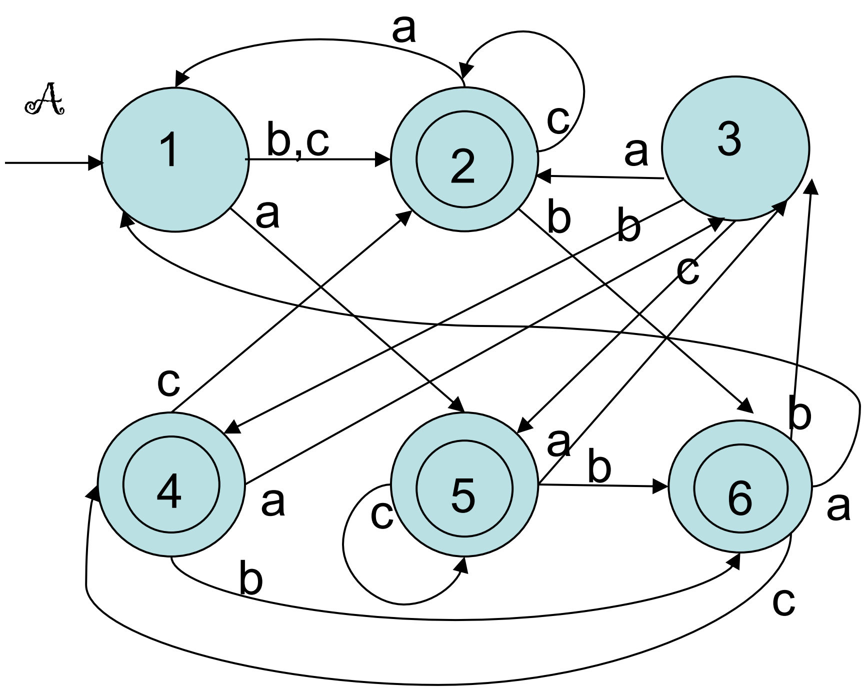
Definition. R is a right invariant equivalence relation iff R is an equivalence relation and $\forall z$ [x R y implies xz R yz].

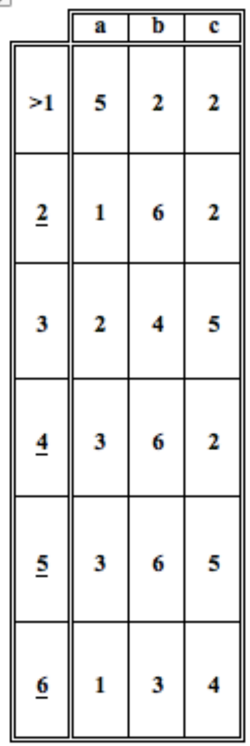
Note: This is only meaningful for relations over strings.

Myhill-Nerode 1 ⇒ 2

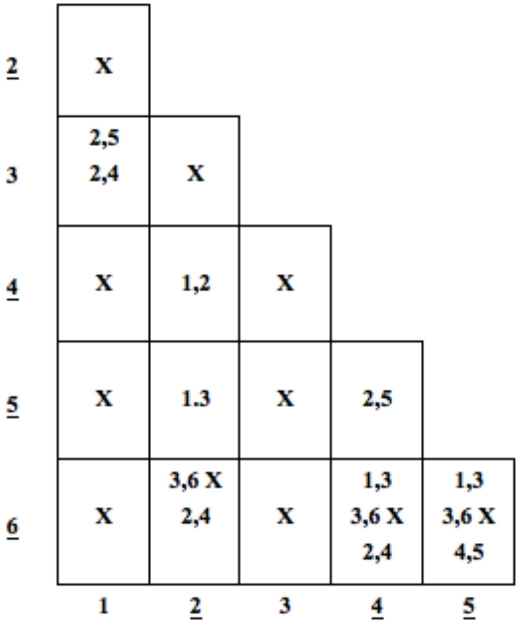
1. Assume L is accepted by some DFA, A = (Q,Σ,δ,q_1,F)
2. Define R_A by x R_A y iff $\delta^*(q_1,x) = \delta^*(q_1,y)$. First, R_A is defined by equality and so is obviously an equivalence relation.
   Clearly if $\delta^*(q_1,x) = \delta^*(q_1,y)$ then $\forall z\ \delta^*(q_1,xz) = \delta^*(q_1,yz)$ because A is deterministic.
   Moreover if $\forall z\ \delta^*(q_1,xz) = \delta^*(q_1,yz)$ then $\delta^*(q_1,x) = \delta^*(q_1,y)$, just by letting z = λ.
   Putting it together x R_A y L iff $\forall z$ xz R_A yz. Thus, R_A is right invariant; its index is |Q| which is finite; and $\mathcal{L}(A) = \cup_{\delta^*(q1,x)\in F}[x]_{RA}$, where $[x]_{RA}$ refers to the equivalence class containing the string x.

DFA, $\mathcal{A}$, Defines RIER, $R_{\mathcal{A}}$ of Finite Index (here 6)





	a	b	c
>1	5	2	2
2	1	6	2
3	2	4	5
4	3	6	2
5	3	6	5
6	1	3	4

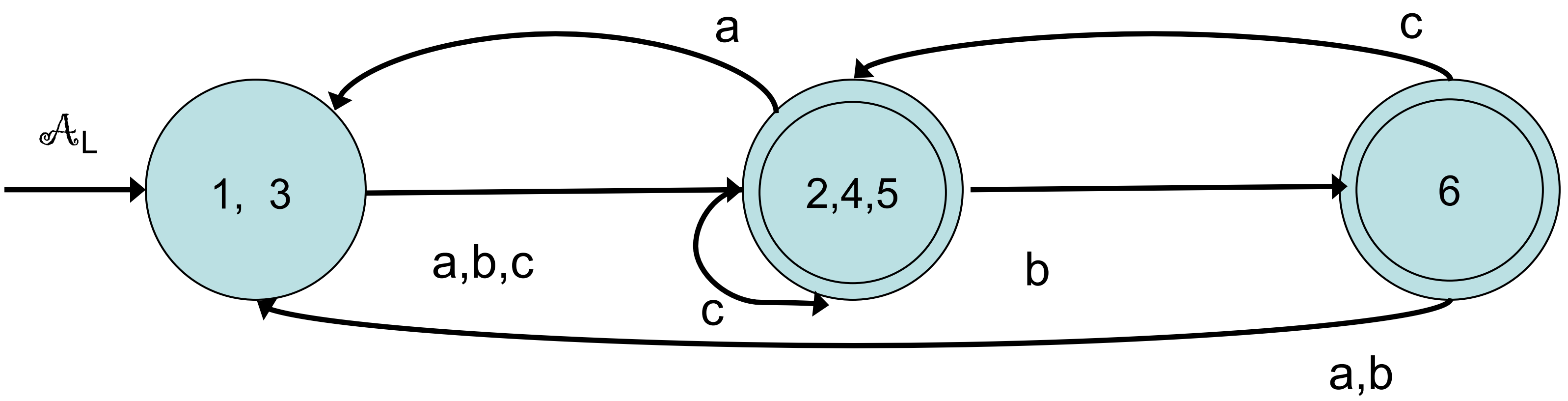


2	X				
3	2,5 2,4	X			
4	X	1,2	X		
5	X	1.3	X	2,5	
6	X	3,6 X 2,4	X	1,3 3,6 X 2,4	1,3 3,6 X 4,5
	1	2	3	4	5

Myhill-Nerode 2 $\Rightarrow$ 3

2. Assume L is the union of some of the classes of a right invariant equivalence relation, R, of finite index.
3. Since x R y iff $\forall$z [xz R yz], R is right invariant and L is the union of some of the equivalence classes, then
   x R y $\Rightarrow$ $\forall$z [xz $\in$ L iff yz $\in$ L] $\Rightarrow$ x R_L y.
   This means that the index of R_L is less than or equal to that of R and so is finite. Note than the index of R_L is then less than or equal to that of any other right invariant equivalence relation, R, of finite index that defines L.

Same Language but Index is 3 This is based on R_L



It is the case that R_L is a refinement of $R_{\mathcal{A}}$ in that x $R_{\mathcal{A}}$ y implies x R_L y. This is true of any relationship for L that is based on the states of some DFA that accepts L.

Thus, since in our first automata abba $R_{\mathcal{A}}$ ac, then abba R_L ac. It is this property that makes the equivalence classes of $\mathcal{A}_L$ be no more than those of $\mathcal{A}$.

Myhill-Nerode 3 ⇒ 1

3. Assume the specific right invariance equivalence relation R_L where x R_L y iff $\forall z$ [$xz \in L$ iff $yz \in L$] has finite index

Define the automaton $A = (Q,\Sigma,\delta,q_1,F)$ by
$Q = \{ [x]_{RL} \mid x \in \Sigma^* \}$
$\delta([x]_{RL},a) = [xa]_{RL}$
$q1 = [\lambda]$
$F = \{ [x]_{RL} \mid x \in L \}$

Note: This is the minimum state automaton, and all others are either equivalent or have redundant indistinguishable states

More Non-Regular

Myhill-Nerode Theorem as Alternative to Pumping Lemma

Use of Myhill-Nerode

- $L = \{a^n b^n \mid n>0\}$ is NOT regular.
- Assume otherwise.
- M-N says that the specific r.i. equiv. relation R_L has finite index, where $x \; R_L \; y$ iff $\forall z \, [\, xz \in L$ iff $yz \in L \,]$.
- Consider the equivalence classes $[a^i b]$ and $[a^j b]$, where $i,j>0$ and $i \neq j$.
- $a^i b b^{i-1} \in L$ but $a^j b b^{i-1} \notin L$ and so $[a^i b]$ is not related to $[a^j b]$ under R_L and thus $[a^i b] \neq [a^j b]$ when $i \neq j$.
- This means that R_L has infinite index.
- Therefore, L is not regular.

xwx is not Regular (MN)

- **L = { x w x | x,w∈ {a,b}+ } :**
- We consider the right invariant equivalence class [a^{i}b], i>0.
- It's clear that a^{i}baa^{i}b is in the language, but a^{k}baa^{i}b is not when k > i.
- This shows that there is a separate equivalence class, [a^{i}b], induced by R$_L$, for each i>0. Thus, the index of R$_L$ is infinite and Myhill-Nerode states that L cannot be Regular.

$a^{Fib(k)}$ is not Regular (MN)

- **$L = \{a^{Fib(k)} \mid k>0\}$:**
- We consider the collection of right invariant equivalence classes $[a^{Fib(j)}]$, $j > 2$.
- It's clear that $a^{Fib(j)}a^{Fib(j+1)}$ is in the language, but $a^{Fib(k)}a^{Fib(j+1)}$ is not when $k>2$ and $k \neq j$ and $k \neq j+2$
- This shows that there is a separate equivalence class $[a^{Fib(j)}]$ induced by R_L, for each $j > 2$.
- Thus, the index of R_L is infinite and Myhill-Nerode states that L cannot be Regular.

Myhill-Nerode and Minimization

- Corollary: The minimum state DFA for a regular language, L, is formed from the specific right invariance equivalence relation R_L, where
  $x \; R_L \; y$ iff $\forall z$ [$xz \in L$ iff $yz \in L$]
- Moreover, all minimum state machines have the same structure as the above, except perhaps for the names of states

What is Regular So Far?

- Any language accepted by a DFA
- Any language accepted by an NFA
- Any language denoted by a Regular Expression
- Any language representing the unique solution to a set of properly constrained regular equations
- Any language, L, that is the union of some of the classes of a right invariant equivalence relation of finite index

What is NOT Regular?

- Well, anything for which you cannot write an accepting DFA or NFA, or a defining regular expression, or a right/left linear grammar (to be discussed shortly), or a set of regular equations, but that's not a very useful statement
- There are two tools we now have that are useful:
  - Pumping Lemma for Regular Languages
  - Myhill-Nerode Theorem

Transducers

Automata with Output

Finite-State Transducers

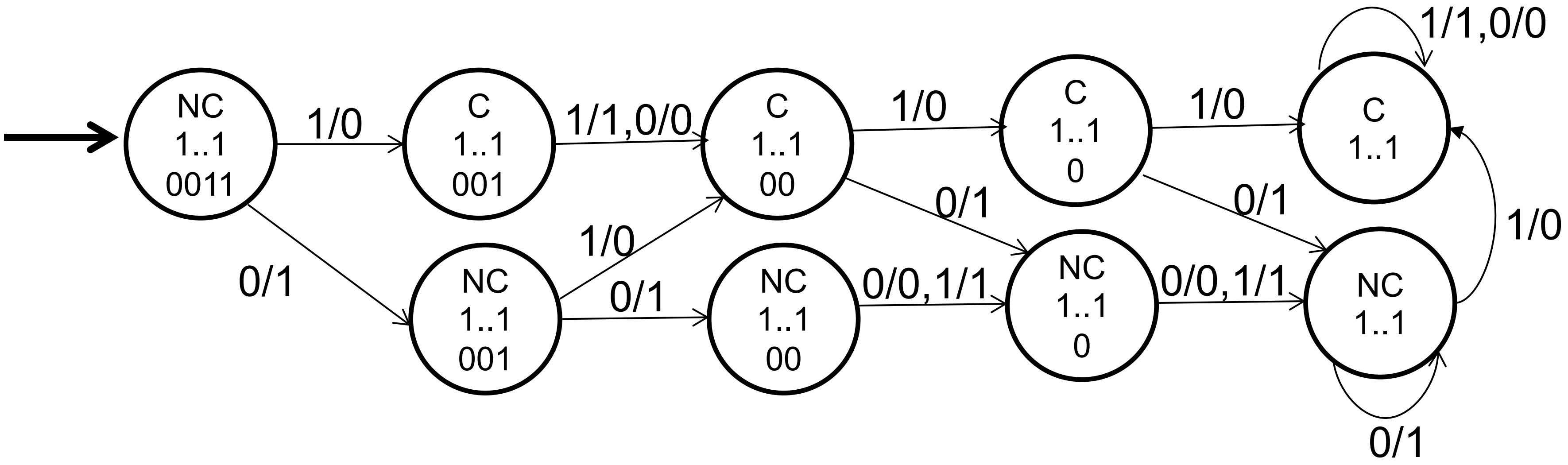
- A transducer is a machine with output
- Mealy Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

    $\gamma: Q \times \Sigma \rightarrow \Gamma$ is the output function
  - Essentially a Mealy Model machine produces a character of output for each character of input it consumes, and it does so on the transitions from one state to the next.
  - A Mealy Model represents a synchronous circuit whose output is triggered each time a new input arrives.

Sample Mealy Model

- Write a Mealy finite-state machine that produces the 2's complement result of subtracting 1101 from a binary input stream (assuming at least 4 bits of input)



Finite-State Transducers

- Moore Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

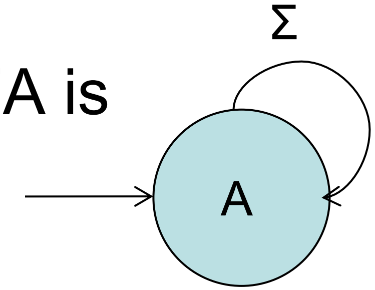
    $\gamma: Q \to \Gamma$ is the output function
  - Essentially a Moore Model machine produced a character of output whenever it enters a state, independent of how it arrived at that state.
  - A Moore Model represents an asynchronous circuit whose output is a steady state until new input arrives.

Summary of Decision and Closure Properties

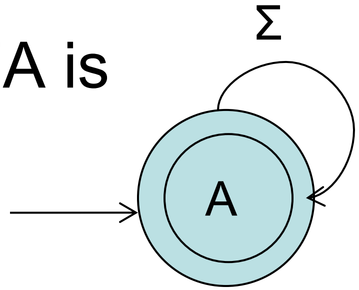
Regular Languages

Decidable Properties

- Membership (just run DFA over string)
- L = Ø: Minimize and see if minimum state DFA is



- L = Σ*: Minimize and see if minimum state DFA is



- Finiteness: Minimize and see if there are no loops emanating on a path to a final state
- Equivalence: Minimize both and see if isomorphic

Closure Properties

- Virtually everything with members of its own class as we have already shown
- Union, concatenation, Kleene *, complement, intersection, set difference, reversal, substitution, homomorphism, quotient with regular sets, Prefix, Suffix, Substring, Exterior, Min, Max and so much more

Regular Languages # 1

- Finite Automata
- Moore and Mealy models: Automata with output.
- Regular operations
- Non-determinism: Its use. Conversion to deterministic FSAs. Formal proof of equivalence.
- Lambda moves: Lambda closure of a state
- Regular expressions
- Equivalence of REs and FSAs.
- Pumping Lemma: Proof and applications.

Regular Languages # 2

- Regular equations: REQs and FSAs.
- Myhill-Nerode Theorem: Right invariant equivalence relations. Specific relation for a language L. Proof and applications.
- Minimization: Why it's unique. Process of minimization. Analysis of cost of different approaches.
- Regular (right linear) grammars, regular languages and their equivalence to FSA languages – Grammars are coming up.

Regular Languages # 3

- Closure properties: Union, concatenation, Kleene *, complement, intersection, set difference, reversal, substitution, homomorphism and quotient with regular sets, Prefix, Suffix, Substring, Exterior.
- Algorithms for reachable states and states that can reach some other chosen states.
- Decision properties: Emptiness, finiteness, equivalence.

Formal Languages

Includes and Expands on
Chapter 2 of Sipser

History of Formal Language

- In 1940s, Emil Post (mathematician) devised rewriting systems as a way to describe how mathematicians do proofs. Purpose was to mechanize them.
- Early 1950s, Noam Chomsky (linguist) developed a hierarchy of rewriting systems (grammars) to describe natural languages.
- Late 1950s, Backus-Naur (computer scientists) devised BNF (a variant of Chomsky's context-free grammars) to describe the programming language Algol.
- 1960s was the time of many advances in parsing. In particular, parsing of context free was shown to be no worse than $O(n^3)$. More importantly, useful subsets were found that could be parsed in O(n).

Grammars

- $G = (V, \Sigma, R, S)$ is a Phrase Structured Grammar (PSG) where
  - V: Finite set of non-terminal symbols
  - Σ: Finite set of terminal symbols ($V \cap \Sigma = \emptyset$)
  - R: finite set of rules of form $\alpha \rightarrow \beta$,
    - α in $(V \cup \Sigma)^* V (V \cup \Sigma)^*$
    - β in $(V \cup \Sigma)^*$
  - S: a member of V called the start symbol
- Right linear restricts all rules to be of forms
  - α in V
  - β of form ΣV, Σ or λ

Derivations

- $x \Rightarrow y$ reads as x derives y iff
  - $x = \gamma\alpha\delta$, $y = \gamma\beta\delta$ and $\alpha \rightarrow \beta$
- $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$
- $\Rightarrow+$ is the transitive closure of $\Rightarrow$
- $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow^* z$ and $z \Rightarrow y$
- Or, $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow z$ and $z \Rightarrow^* y$
- $L(G) = \{ w \mid S \Rightarrow^* w \text{ and } w \in \Sigma^* \}$ is the language generated by G.

Regular Grammars

- Regular grammars are also called right linear grammars
- Each rule of a regular grammar is constrained to be of one of the three forms:

$A \rightarrow \lambda$,	$A \in V$
$A \rightarrow a$,	$A \in V$, $a \in \Sigma$
$A \rightarrow aB$,	$A, B \in V$, $a \in \Sigma$

Example Regular Grammars

G = ({<EVEN>,<ODD>}, {0,1}, R, <EVEN>); R is:

<EVEN> → 0 <EVEN> | 1<ODD>

<ODD> → 1 <EVEN> | 0 <ODD> | λ

L(G) = { w | w ∊ {0,1}* and w has odd parity }

G = ({<0>,<1>,<2>}, {0,1}, R, <0>); R is:

<0> → 0<0> | 1<1>

<1> → 0<2> | 1<0> | λ

<2> → 0<1> | 1<2>

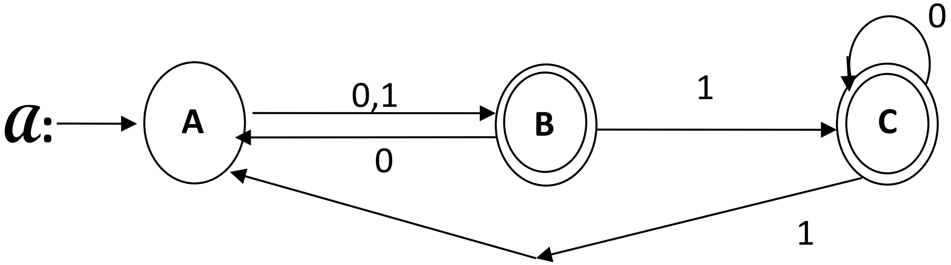
L(G) = { w | w ∊ {0,1}* and "You tell me" }

DFA to Regular Grammar

- Every language recognized by a DFA is generated by an equivalent regular grammar
- Given $A = (Q,\Sigma,\delta,q_0,F)$, $L(A)$ is generated by $G_A = (Q,\Sigma,R,q_0)$ where R contains
  $q \to as$ iff $\delta(q,a) = s$, $a \in \Sigma$
  $q \to \lambda$ iff $q \in F$

Example of DFA to Grammar

- **DFA**



- **Grammar**

G = ({A,B,C}, {0,1), R, A), where **R** is:

A → 0 B | 1 B

B → 0 A | 1 C | λ

C → 0 C | 1 A | λ

Regular Grammar to NFA

- Every language generated by a regular grammar is recognized by an equivalent NFA
- Given G = (V, Σ, R, S), *L*(G) is recognized by $A_G = (V \cup \{f\}, \Sigma, \delta, S, \{f\})$ where δ is defined by
  $\delta(A,a) \subseteq \{B\}$ iff $A \rightarrow aB$
  $\delta(A,a) \subseteq \{f\}$ iff $A \rightarrow a$
  $\delta(A,\lambda) \subseteq \{f\}$ iff $A \rightarrow \lambda$

Example of Grammar to NFA

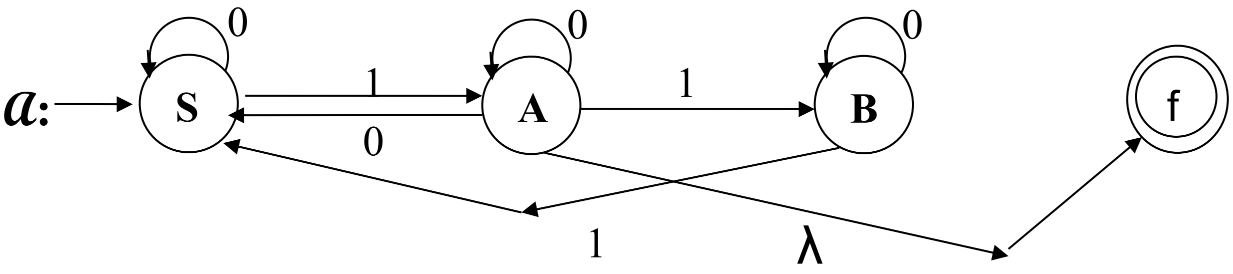
- **Grammar G = ({S,A,B}, {0,1), R, S),** where **R** is:

S → 0 S | 1 A

A → 0 S | 0 A | 1 B | λ

B → 1 S | 0 B

- **NFA (can remove f and make A final)**



What More is Regular?

- Any language, L, generated by a right linear grammar ($A \to a$, $A \to \lambda$, $A \to aB$)
- Any language, L, generated by a left linear grammar ($A \to a$, $A \to \lambda$, $A \to Ba$)
  - Easy to see L is regular as we can reverse these rules and get a right linear grammar that generates L^R, but then L is the reverse of a regular language which is regular
  - Similarly, the reverse L^R of any regular language L is right linear and hence the language itself is left linear

More than One Letter?

- Any language, L, generated by an extended right linear grammar ($A \rightarrow \alpha$, $A \rightarrow \lambda$, $A \rightarrow \alpha B$)
  Any language, L, generated by an extended left linear grammar ($A \rightarrow \alpha$, $A \rightarrow \lambda$, $A \rightarrow B \alpha$)
  where α is a non-zero-length string over the alphabet
- Can just change a rule involving $\alpha = a_1a_2..a_k$, $k> 1$ to a series of k rules
  - One is $A \rightarrow a_1 A'$, where A' is a <u>new</u> symbol
  - If k=2, the other is a_2 or $a_2 B$ depending on whether we had $A \rightarrow \alpha$ or $A \rightarrow \alpha B$
  - If k>2, then repeat above on the new rule involving $a_2a_3..a_k$ (either $A \rightarrow a_2a_3..a_k$ or $A \rightarrow a_2a_3..a_k B$)

Mixing Right and Left Linear

- We can get non-Regular languages if we present grammars that have both right and left linear rules
- To see this, consider G = ({S,T}, Σ, R, S), where R is:
  - $S \rightarrow aT$
  - $T \rightarrow Sb \mid b$
- $L(G) = \{ a^n b^n \mid n > 0 \}$ which is a classic non-regular, context-free language

Context Free Languages

Context Free Grammar

G = (V, Σ, R, S) is a PSG where

Each member of R is of the form

$A \to \alpha$ where α is a strings $(V \cup \Sigma)^*$

Note that the left-hand side (lhs) of a rule is a letter in V;

The right-hand side (rhs) is a string from the combined alphabets

The right-hand side can even be empty (ε or λ)

A context free grammar is denoted as a CFG and the language generated is a Context Free Language (CFL).

A CFL is recognized by a Push Down Automaton (PDA) to be discussed a bit later.

Classic CFLs

L1 = { a^n b^n | n ≥ 0 }

G = ({S}, {a,b}, R, S) is a CFG where R is:

$S \rightarrow a\ S\ b \mid \lambda$

L2 = { w w^R | w ∊ {a,b}* }

G = ({S}, {a,b}, R, S) is a CFG where R is:

$S \rightarrow a\ S\ a \mid b\ S\ b \mid \lambda$

L3 = { w | w ∊ {a,b}* and the number of a's is the same as b's}

G = ({S}, {a,b}, R, S) is a CFG where R is:

$S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \lambda$

Culd also do $S \rightarrow S\ a\ S\ b\ S \mid S\ b\ S\ a\ S \mid \lambda$

More CFLs

G_i = ({S}, {a,b}, R_i, S) is a CFG where:

R_1: S $\rightarrow$ a S b | a | a S	L_1 = { a^m b^n | m >n }
R_2: S $\rightarrow$ a S a | b S b | λ | a | b	L_2 = { w | w is a palindrome over {a,b} }

Sample "Useful" CFG

Example of a grammar for a small language:

G = ({<program>, <stmt-list>, <stmt>, <expression>},
{begin, end, ident, ;, =, +, -}, R, <program>) where R is

<program>	→ begin <stmt-list> end
<stmt-list>	→ <stmt>; | <stmt> ; <stmt-list>
<stmt>	→ ident = <expression>
<expression>	→ ident + ident | ident - ident | ident

Here "ident" is a token return from a scanner, as are "begin", "end", ";", "=", "+", "-"

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1	**<assgn>**	**→ <id> = <expr>**
R2	**<id>**	**→ a | b | c**
R3	**<expr>**	**→ <id> + <expr>**
R4		**| <id> * <expr>**
R5		**| (<expr>)**
R6		**| <id>**

The statement a = b * (a + c)
Is generated by the leftmost derivation:

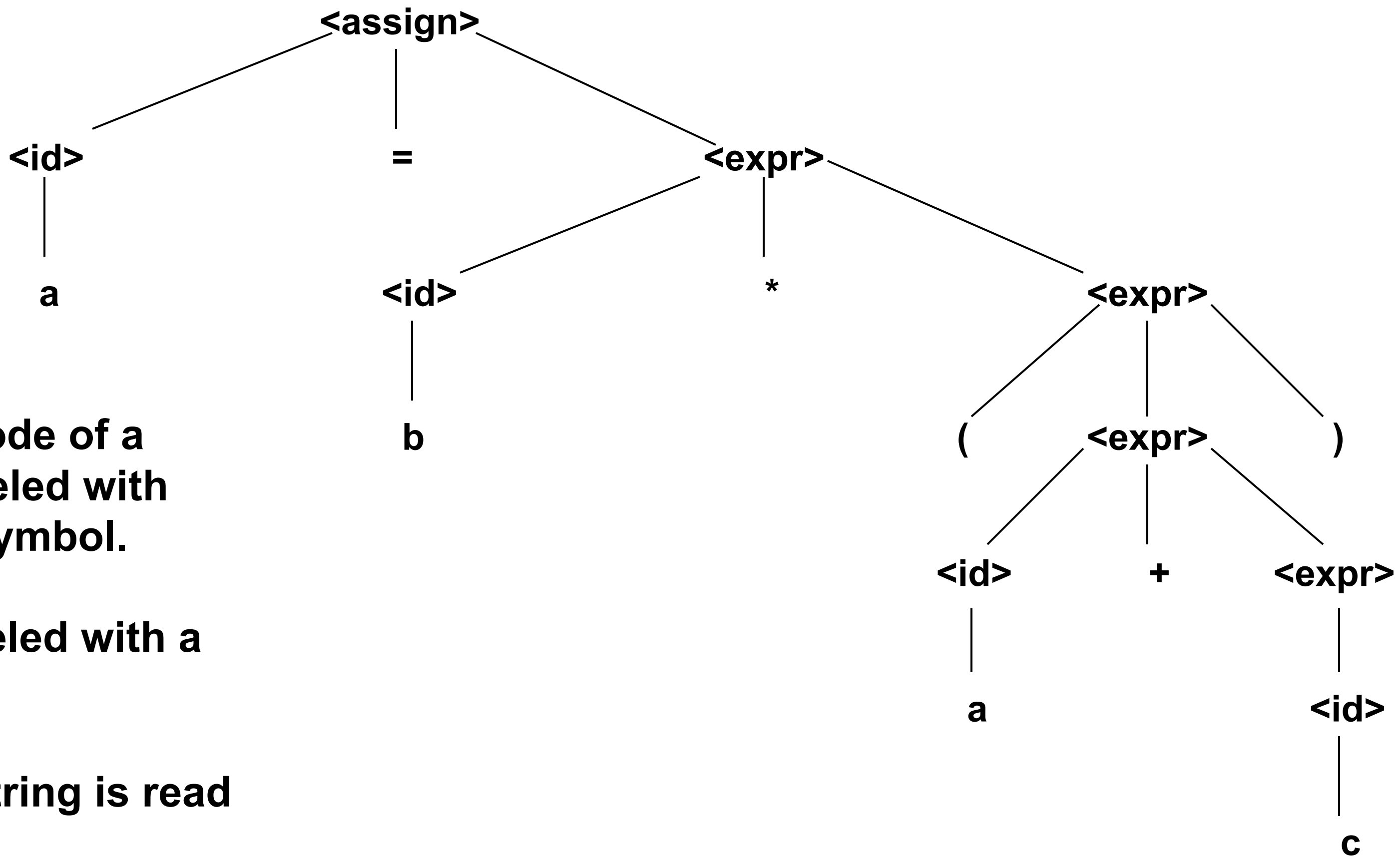
<assgn>	**⇒ <id> = <expr>**	**R1**
	⇒ a = <expr>	**R2**
	⇒ a = <id> * <expr>	**R4**
	⇒ a = b * <expr>	**R2**
	⇒ a = b * (<expr>)	**R5**
	⇒ a = b * (<id> + <expr>)	**R3**
	⇒ a = b * (a + <expr>)	**R2**
	⇒ a = b * (a + <id>)	**R6**
	⇒ a = b * (a + c)	**R2**

In a **leftmost derivation** in that only the leftmost non-terminal is replaced
This is odd as it treats expression parse as **right to left associativity even without parentheses used here**

Parse Trees

A parse tree is a graphical representation of a derivation

For instance, the parse tree for the statement a = b * (a + c) is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

The generated string is read left to right

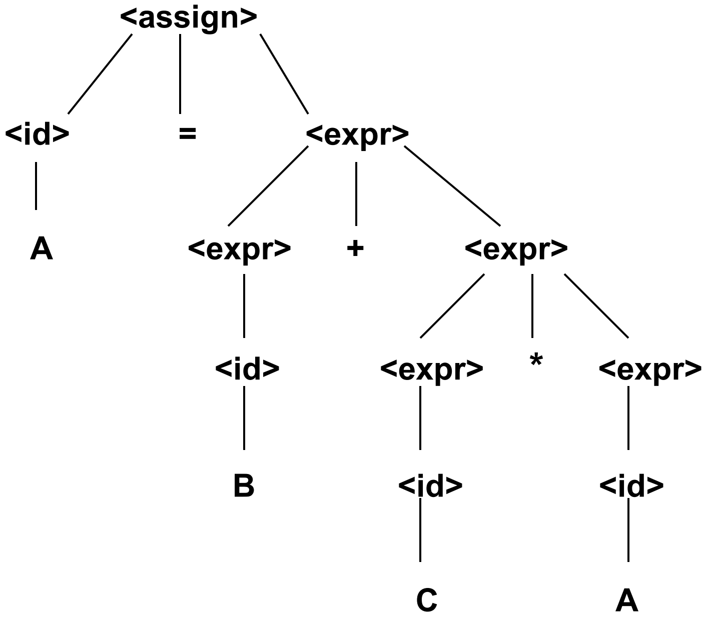
Ambiguity

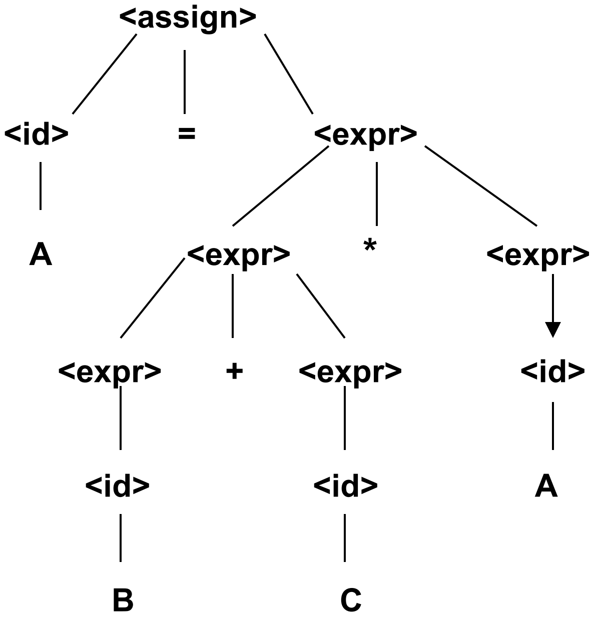
A grammar that generates a sentence for which there are two or more distinct parse trees is said to be "ambiguous"

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression a = b + c * a

```
<assgn> → <id> = <expr>
<id>     → a | b | c
<expr>   → <expr> + <expr>
          |  <expr> * <expr>
          |  ( <expr> )
          | <id>
```

Ambiguous Parse





This grammar generates two parse trees for the same expression.

If a language structure has more than one parse tree, the semantic meaning of the structure cannot be determined uniquely.

Precedence

Operator precedence:

If an operator is generated lower in the parse tree, it indicates that the operator has precedence over the operator generated higher up in the tree.

An unambiguous grammar for expressions:

```
 <assign> → <id> = <expr>
  <id>       → a | b | c
  <expr>    → <expr> + <term>
               |  <term>
  <term>    → <term> * <factor>
               |   <factor>
  <factor>  →  ( <expr> )
                | <id>
```

This grammar indicates the usual precedence order of multiplication and addition operators.

This grammar generates unique parse trees independently of doing a rightmost or leftmost derivation

Left (right)most Derivations

Leftmost derivation:

```
<assgn> → <id> = <expr>
        → a = <expr>
        → a = <expr> + <term>
        → a = <term> + <term>
        → a = <factor> + <term>
        → a = <id> + <term>
        → a = b + <term>
        → a = b + <term> *<factor>
        → a = b + <factor> * <factor>
        → a = b + <id> * <factor>
        → a = b +  c  * <factor>
        → a = b +  c  * <id>
        → a = b +  c  *  a
```

Rightmost derivation:

```
<assgn> ⇒ <id> = <expr>
        ⇒ <id> = <expr> + <term>
        ⇒ <id> = <expr> + <term> *<factor>
        ⇒ <id> = <expr> + <term> *<id>
        ⇒ <id> = <expr> + <term> *  a
        ⇒ <id> = <expr> + <factor> *  a
        ⇒ <id> = <expr> + <id> *  a
        ⇒ <id> = <expr> + c  *  a
        ⇒ <id> = <term> + c  *  a
        ⇒ <id> = <factor> + c  *  a
        ⇒ <id> = <id> + c  *  a
        ⇒ <id> =  b + c  * a
        ⇒ a = b +  c  *  a
```

Ambiguity Test

- A Grammar is Ambiguous if there are two distinct parse trees for some string
- Or, two distinct leftmost derivations
- Or, two distinct rightmost derivations
- Some languages are inherently ambiguous, but many are not
- Unfortunately (to be shown later) there is no systematic (algorithmic) test for ambiguity of an arbitrary context free grammars

Unambiguous Grammar

When we encounter ambiguity, we try to rewrite the grammar to avoid ambiguity.

The ambiguous expression grammar:

<expr> → <expr> <op> <expr> | id | int | (<expr>)

<op> → + | - | * | /

Can be rewritten as:

<expr> → <term> | <expr> + <term> | <expr> - <term>

<term> → <factor> | <term> * <factor> | <term> / <factor>.

<factor> → id | int | (<expr>)

Parsing Problem

The parsing Problem: Take a string of symbols in a language (tokens) and use a grammar for that language to construct the parse tree or report that the sentence is syntactically incorrect.

For correct strings:

Sentence + grammar → parse tree

For a compiler, a sentence is a program:

Program + grammar → parse tree

Types of parsers:

Top-down aka predictive (recursive descent parsing)

Bottom-up aka shift-reduce

Inherent Ambiguity

- There are some CFLs that are inherently ambiguous and others for which we may just have carelessly written an ambiguous grammar.
- We will see later in course that it is not possible to inspect an arbitrary CFG and determine if it is unambiguous.
- However, parsers must be unambiguous to avoid semantic ambiguity.

Not All is Lost

- Just because we cannot determine ambiguity of a grammar does not mean we cannot have a subclass of grammars that are guaranteed to be unambiguous and that can be used to generate precisely the set of unambiguous CFLs.
- Note the distinction between the class of unambiguous CFGs and unambiguous CFLs.
  - Every CFL has an infinite number of CFGs
  - Some of the CFGs for an unambiguous CFL are unambiguous; some are not
  - Every unambiguous CFL has some grammars that are in forms that can be recognized as unambiguous and are the bases of parsers that run in linear time

LR(k) and LL(k) Grammars

- An LL(k) grammar is a grammar that can drive a top-down parse by always making the right parsing decision with just k tokens of lookahead.
- An LR(k) grammar is a grammar that can drive a bottom-up parse by always making the right parsing decision with just k tokens of lookahead.

LL(k) Grammars

- LL means we read the input from left-to-right using a leftmost derivation with a correct decision requiring just k tokens of lookahead.
- There is an algorithm to determine, for any given k, whether an arbitrary CFG is LL(k).
- LL(k+1) grammars can generate languages that cannot be generated by LL(k) ones.
- Lim $k \to \infty$ LL(k) gets all unambiguous CFLs.
- All programming languages you work with are LL(1) so long as we cheat and use a symbol table.
- LL parsers hate left recursion

LR(k) Grammars

- LR means we read the input from left-to-right using a rightmost derivation run in reverse with a correct decision requiring just k tokens of lookahead.
- There is an algorithm to determine, for any given k, whether an arbitrary CFG is LR(k).
- LR(1) grammars are sufficient to generate any and all unambiguous CFLs.
- All programming languages you work with are LR(1) so long as we cheat and use a symbol table.
- LR parsers hate right (tail) recursion.

Removing Left Recursion if doing Top Down

Given left recursive and non left recursive rules

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

Can view as

$A \rightarrow (\beta_1 \mid \ldots \mid \beta_m) (\alpha_1 \mid \ldots \mid \alpha_n)^*$

Star notation is an extension to normal notation with obvious meaning

Now, it should be clear this can be done right recursively as

$A \rightarrow \beta_1 B \mid \ldots \mid \beta_m B$

$B \rightarrow \alpha_1 B \mid \ldots \mid \alpha_n B \mid \lambda$

Left to Right Recursive Expressions

Grammar: Expr → Expr + Term | Term
Term → Term * Factor | Factor
Factor → (Expr) | Int

Fix: Expr → Term ExprRest
ExprRest → + Term ExprRest | λ
Term → Factor TermRest
TermRest → * Factor TermRest | λ
Factor → (Expr) | Int

Removing Right Recursion if doing Bottom Down

Given left recursive and non left recursive rules

$A \rightarrow \alpha_1 A \mid \ldots \mid \alpha_n A \mid \beta_1 \mid \ldots \mid \beta_m$

Can view as

$A \rightarrow (\alpha_1 \mid \ldots \mid \alpha_n)^* (\beta_1 \mid \ldots \mid \beta_m)$

Star notation is an extension to normal notation with obvious meaning

Now, it should be clear this can be done right recursively as

$A \rightarrow B\ \beta_1 \mid \ldots \mid B\ \beta_m$

$B \rightarrow B\ \alpha_1 \mid \ldots \mid B\ \alpha_n \mid \lambda$

Bottom Up vs Top Down

- Bottom-Up: Two stack operations (shift/reduce)
  - Shift (move input symbol to stack)
  - Reduce (replace top of stack α with A, when $A \rightarrow \alpha$)
  - Challenge is when to do shift or reduce and what reduce to do.
    - Can have both kinds of conflict (shift-reduce, reduce-reduce)
- Top-Down: (predictive)
  - If top of stack is terminal
    - If same as input, read and pop
    - If not, we have an error
  - If top of stack is a non-terminal A
    - Replace A with some α, when $A \rightarrow \alpha$
    - Challenge is what A-rule to use

Recursive Descent Parsing

Recursive Descent parsing uses recursive procedures to model the parse tree to be constructed. The parse tree is built from the top down, trying to construct a left-most derivation.

Beginning with ***start*** symbol, for each non-terminal (syntactic class) in the grammar a procedure which parses that syntactic class is constructed.

Consider the expression grammar:

E → T E’
E’ → **+** T E’ | **λ**
T → F T’
T’ → ***** F T’ | **λ**
F → **(** E **)** | **id**

The following procedures can parse strings top-down in this language:

Recursive Descent Example

```
Procedure E
  begin { E }
    call T
    call E’
    print (“ E found ”)
  end { E }

Procedure E’
  begin { E’ }
    If token = “+” then
      begin { IF }
        print (“ + found “)
        Get next token
        call T
        call E’
      end { IF }
    print (“ E’ found “)
  end { E’ }
```

```
Procedure T
  begin { T }
    call F
    call T’
    print (“ T found ”)
  end { T }

Procedure T’
  begin { T’ }
    If token = “ * ” then
      begin { IF }
        print (“ * found “)
        Get next token
        call F
        call T’
      end { IF }
    print (“ T’ found “)
  end { T’ }
```

```
Procedure F
  begin { F }
    case token is
    “(“:
       print (“ (  found ”)
       Get next token
       call E
       if token = “)” then
         begin { IF }
           print (“ ) found”)
           Get next token
            print (“ F found “)
         end { IF }
       else
         call ERROR
    “id“:
       print (“ id found ”)
       Get next token
       print (“ F found “)
   otherwise:
       call ERROR
  end { F }
```

Reduced CFG

- No Nullable ($A \rightarrow \lambda$) unless λ is in language; if so, we can have $S \rightarrow \lambda$, provided S appears on no rhs
- No chain (unit) rules ($A \rightarrow B$)
- No non-productive non-terminal symbols (variables); a variable, A, is productive if $A \Rightarrow+ w$ for some $w \in \Sigma^*$
- No useless symbols; a symbol is useless is it never appears in a syntactic form that is derivable from the start symbol

Nullable Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set $\text{Nullable}(G) = \{A \mid A \Rightarrow^* \lambda\}$
- Nullable(G) is computed as follows

```
Nullable(G) ⊇ { A | A → λ }
Repeat
    Nullable(G) ⊇ { B | B → α and α ∈ Nullable* }
until no new symbols are added
```

Removal of λ-Rules

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set Nullable(G)
- Remove all λ-rules
- For each rule of form $B \rightarrow \alpha A\beta$ where A is nullable, add in the rule $B \rightarrow \alpha\beta$
- The above has the potential to greatly increase the number of rules and add unit rules
  (those of form $B \rightarrow C$, where $B,C \in V$)
- If S is nullable, add new start symbol S_0, as new start state, plus rules $S_0, \rightarrow \lambda$ and $S_0 \rightarrow \alpha$, where $S \rightarrow \alpha$

Chains (Unit Rules)

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ-rules removed
- For $A \in V$, $\mathrm{Chain}(A) = \{ B \mid A \Rightarrow^* B, B \in V \}$
- Chain(A) is computed as follows

```
$\mathrm{Chain}(A) \supseteq \{ A \}$
Repeat
    $\mathrm{Chain}(A) \supseteq \{ C \mid B \rightarrow C \text{ and } B \in \mathrm{Chain}(A) \}$
until no new symbols are added
```

Removal of Unit-Rules

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules removed, except perhaps from start symbol
- Compute Chain(A) for all A∈V
- Create the new grammar G = (V, Σ, R, S) where R is defined by including for each A∈V, all rules of the form $A \rightarrow \alpha$, where $B \rightarrow \alpha \in R$, $\alpha \notin V$ and $B \in$ Chain(A)
  Note: A∈Chain(A) so all its non-unit-rules are included

Non-Productive Symbols

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules and unit-rules removed
- Non-productive non-terminal symbols never lead to a terminal string (not productive)
- Productive(G) is computed by
  Productive(G) $\supseteq$ { A | A $\rightarrow \alpha$, $\alpha \in \Sigma^*$ }
  Repeat
  Productive(G) $\supseteq$ { B | B $\rightarrow \alpha$, $\alpha \in (\Sigma \cup \text{Productive})^*$ }
  until no new symbols are added
- Keep only those rules that involve productive symbols
- If no rules remain, grammar generates nothing

Unreachable Symbols

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules, unit-rules and non-productive symbols removed
- Unreachable symbols are ones that are inaccessible from start symbol
- We compute the complement (Useful)
- Useful(G) is computed by

```
Useful(G) ⊇ { S }
Repeat
    Useful(G) ⊇ { C | B → αCβ, C∈V∪Σ, B∈ Useful(G) }
until no new symbols are added
```

- Keep only those rules that involve useful symbols
- If no rules remain, grammar generates nothing

Chomsky Normal Form

- Each rule of a reduced CFG whose rules are constrained to be of one of the three forms:

  $A \rightarrow a, \qquad A \in V, a \in \Sigma$

  $A \rightarrow BC, \qquad A,B,C \in V$

- If the language contains λ then we allow

  $S \rightarrow \lambda$

  and constrain non-terminating rules to be

  $A \rightarrow BC, \qquad A \in V, \quad B,C \in (V - \{S\})$

CFG to CNF

- Let G = (V, Σ, R, S) be arbitrary reduced CFG
- Define G'=(V ∪ { <a> | a∈Σ }, Σ, R, S)
- Add the rules <a> → a, for all a ∈ Σ
- For any rule, A → α, $|\alpha| > 1$, change each terminal symbol, a, in α to the non-terminal <a>
- Now, for each rule A → BCα, $|\alpha| > 0$, introduce the new non-terminal B<Cα>, and replace the rule A → BCα with the rule A → B<Cα> and add the rule <Cα> → Cα
- Iteratively apply the above step until all rules are in CNF

Example of CNF Conversion

Starting Grammars

- L = { a^i b^j c^k | i=j or j=k }
- G = ({S,A,<B=C>,C,<A=B>}, {a,b}, R, S)
- R:
  - S → A | C
  - A → a A | <B=C>
  - <B=C> → b <B=C> c | λ
  - C → C c | <A=B>
  - <A=B> → a <A=B> b | λ

Remove Null Rules

- Nullable = {<B=C>, <A=B>, A, C, S}
  - S’ → S | λ // S’ added to V
  - S → A | C
  - A → a A | a |<B=C>
  - <B=C> → b <B=C> c | b c
  - C → C c | c | <A=B>
  - <A=B> → a <A=B> b | ab

Remove Unit Rules

- Chains= {[S’:S’,S,A,C,<A=B>,<B=C>],[S:S,A,C,<A=B>,<B=C>], [A:A,<B=C>],[C:C,<B=C>],[<B=C>:<B=C>], [<A=B>:<A=B>]}
  - S’ → λ | aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - S → aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - A → aA | a | b<B=C>c | bc
  - <B=C> → b<B=C>c | bc
  - C → Cc | c | a<A=B>b | ab
  - <A=B> → a<A=B>b | ab

Remove Useless Symbols

- All non-terminal symbols are productive (lead to terminal string)

- S is useless as it is unreachable from S' (new start).
- All other symbols are reachable from S'

Normalize rhs as CNF

- S’ → λ | <a>A | a | <b><<B=C><c>> | <b><c> | C<c> | c | <a><<A=B><b>> | <a><b>
- A → <a>A | a |<b><<B=C><c>> | <b><c>
- <B=C> → <b><<B=C><c>> | <b><c>
- C → C<c> | c | <a><<A=B><b>> | <a><b>
- <A=B> → <a> <<A=B><b>> | <a><b>
- <<B=C><c>> → <B=C><c>
- <<A=B><b>> → <A=B><b>
- <a> → a
- <b> → b
- <c> → c

CKY (Cocke, Kasami, Younger) O(N^3) PARSING

Dynamic Programming

To solve a given problem, we solve small parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution.

The Parsing problem for arbitrary CFGs was elusive, in that its complexity was unknown until the late 1960s. In the meantime, theoreticians developed notion of simplified forms that were as powerful as arbitrary CFGs. The one most relevant here is the Chomsky Normal Form – CNF. It states that the only rule forms needed are:

A →	BC	where B and C are non-terminals
A →	a	where a is a terminal

This is provided the string of length zero is not part of the language.

CKY (Bottom-Up Technique)

```
Let the input string be a sequence of $n$ letters $a_1$ ... $a_n$.
Let the grammar contain $r$ terminal and nonterminal symbols $R_1$ ... $R_r$,
Let $R_1$ be the start symbol.
Let P[n,n] be an array of Sets over {1,…n}. Initialize all elements of P to empty ({}).
For each col = 1 to n
    For each unit production $X \rightarrow a_i$, set add X to P[1,col].
For each row = 2 to n
    For each col = 1 to n-row+1
        For each row2 = 1 to row-1
            if $B \in$ P[row2,col] and $C \in$ P[row-row2,col+row2]  and A -> B C then
                add A to P[row,col]
If $R_1 \in$ P[n,n] is true then $a_1$ ... $a_n$ is member of language
else $a_1$ ... $a_n$ is not a member of language
```

CKY Parser

Present the **CKY** recognition matrix for the string **abba** assuming the Chomsky Normal Form grammar, **G = ({S,A,B,C,D,E}, {a,b}, R, S)**, specified by the rules **R**:

S →	**AB | BA**	
A →	**CD | a**	
B →	**CE | b**	
C →	**a | b**	
D →	**AC**	
E →	**BC**	

	a	**b**	**b**	**a**
1	A,C	B,C	B,C	A,C
2	S,D	E	S,E	
3	B	B		
4	S,E			

2nd CKY Example

E → E F | M E | P E | a
F → M F | P F | M E | P E
P → +
M → –

	a	–	a	+	a	–	a
1	E	M	E	P	E	M	E
2		E, F		E, F		E, F	
3	E		E		E		
4		E, F		E, F			
5	E		E				
6		E, F					
7	E						

Pumping Lemma for Context Free Languages

What is not a CFL

CFL Pumping Lemma Concept

- Let L be a context free language then there is CNF grammar G = (V, Σ, R, S) such that *L*(G) = L.
- As G is in CNF all its rules that allow the string to grow are of the form A → BC, and thus growth has a binary nature.
- Any sufficiently long string z in L will have a parse tree that must have deep branches to accommodate z's growth.
- Because of the binary nature of growth, the width of a tree with maximum branch length k at its deepest nodes is at most 2^k; moreover, if the frontier of the tree is all terminals, then the string so produced is of length at most 2^{k-1}; since the last rule applied for each leaf is of the form A → a.
- Any terminal branch in a derivation tree of height > |V| has more than |V| internal nodes labelled with non-terminals. The "pigeonhole principle" tells us that whenever we visit |V| +1 or more nodes, we must use at least one variable label more than once. This creates a self-embedding property that is key to the repetition patterns that occur in the derivation of sufficiently long strings.

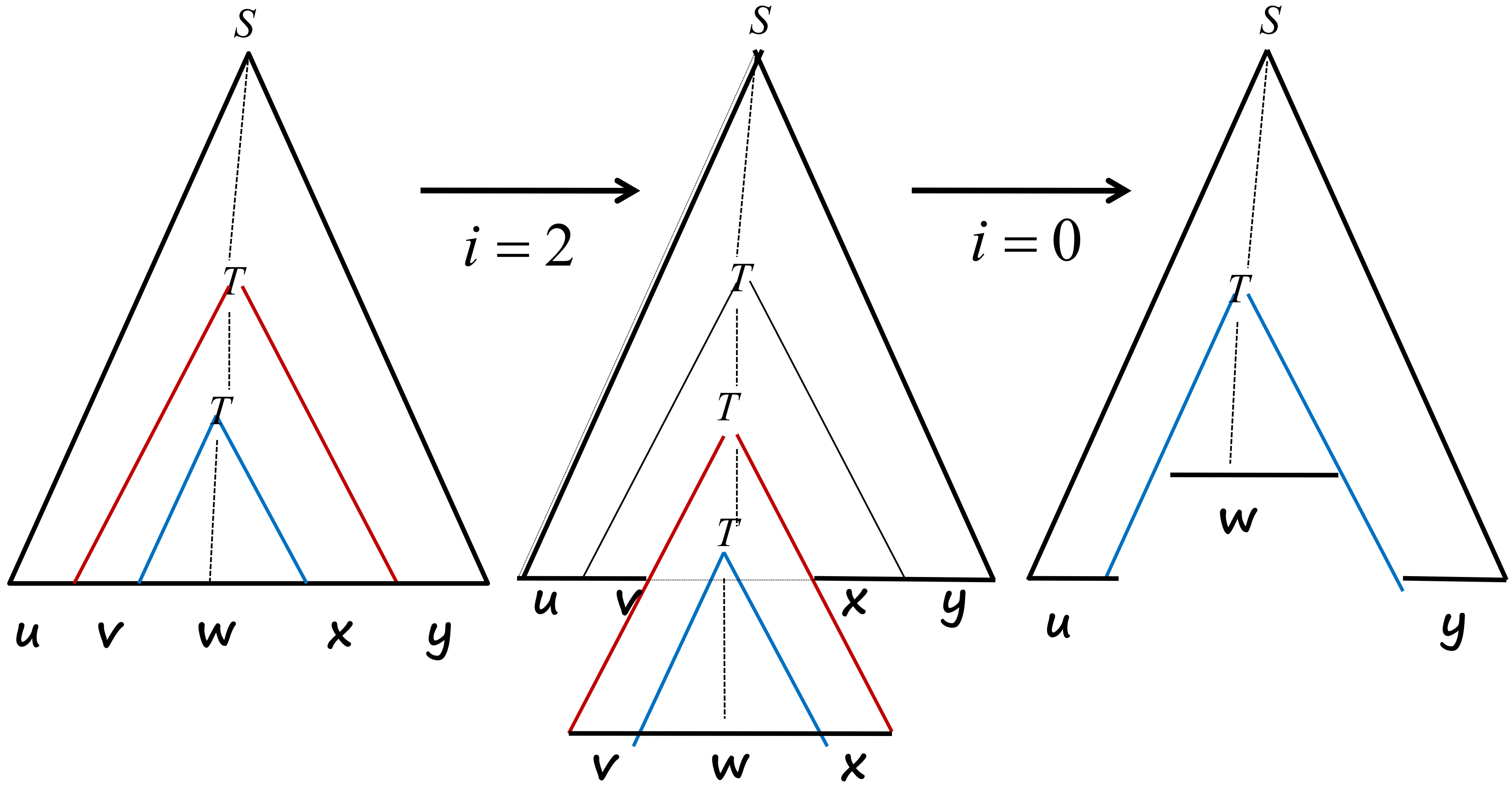
Pumping Lemma For CFL

- Let L be a CFL then there exists an N>0 such that, if $z \in L$ and $|z| \geq N$, then z can be written in the form uvwxy, where $|vwx| \leq N$, $|vx|>0$, and for all $i \geq 0$, $uv^iwx^iy \in L$.
- This means that interesting context free languages (infinite ones) have a self-embedding property that is symmetric around some central area, unlike regular where the repetition has no symmetry and occurs at the start.

Pumping Lemma Proof

- If L is a CFL then it is generated by some CNF grammar, G = (V, Σ, R, S). Let |V| = k. For any string z, such that $|z| \geq N = 2^k$, the derivation tree for z based on G must have a branch with at least k+1 nodes labelled with variables from G.
- By the PigeonHole Principle at least two of these labels must be the same. Let the first (starting from frontier) repeated variable be T and consider the last two instances of T on this path.
- Let z = uvwxy, where $S \Rightarrow^* uTy \Rightarrow^* uvTxy \Rightarrow^* uvwxy$
- Clearly, then, we know $S \Rightarrow^* uTy$; $T \Rightarrow^* vTx$; and $T \Rightarrow^* w$
- But then, we can start with $S \Rightarrow^* uTy$; repeat $T \Rightarrow^* vTx$ zero or more times; and then apply $T \Rightarrow^* w$.
- But then, $S \Rightarrow^* uv^iwx^iy$ for all $i \geq 0$, and thus $uv^iwx^iy \in L$, for all $i \geq 0$.
- Why is $|vx| > 0$? Why is $|vwx| \leq N$? Note there are no unit rules and there are no other repetitions past the first of these T's.

Visual Support of Proof



Lemma's Adversarial Process

- Assume L = $\{a^n b^n c^n \mid n>0\}$ is a CFL
- P.L.: Provides N>0   We CANNOT choose N; that's the P.L.'s job
- Our turn: Choose $a^N b^N c^N \in L$ We get to select a string in L
- P.L.: $a^N b^N c^N = uvwxy$, where $|vwx| \le N$, $|vx|>0$, and for all $i \ge 0$, $uv^iwx^iy \in L$ We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose i=0.   We have the power here
- P.L: Two cases:
  (1) vx contains some a's and maybe some b's. Because $|vwx| \le N$, it cannot contain c's if it has a's. i=0 erases some a's but we still have N c's so $uwy \notin L$
  (2) vx contains no a's. Because $|vx|>0$, vx contains some b's or c's or some of each. i=0 erases some b's and/or c's but we still have N a's so $uwy \notin L$
- CONTRADICTION, so L is NOT a CFL

Second Example: PL for CFL

- Assume L = { ww | w ∈ {a,b}$^+$ } is a CFL
- P.L.: Provides N>0 We CANNOT choose N; that's the P.L.'s job
- Our turn: Choose $a^Nb^Na^Nb^N \in L$ We get to select a string in L
- P.L.: $a^Nb^Na^Nb^N$ = uvwxy, where |vwx| ≤ N, |vx|>0, and for all i≥0, $uv^iwx^iy \in L$ We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose i=0. We have the power here
- P.L: Two cases:
  (1) vx contains some a's and maybe some b's. Because |vwx| ≤ N, it cannot contain a's from both parts involving a's. i=0 erases at least one a from one sequence of a's but we still have N a's in the other, so uwy∉L
  (2) vx contains no a's, then it must contain b's only. Because |vx| >0 and |vwx| ≤ N, it erases some b's from just one sequence of b's but we still have N b's in the other portion so uwy∉ L
- CONTRADICTION, so L is <u>NOT</u> a CFL

Non-Closure

- Intersection ($\{ a^n b^n c^n \mid n \geq 0 \}$ is not a CFL)
  $\{ a^n b^n c^n \mid n \geq 0 \} =$
  $\{ a^n b^n c^m \mid n,m \geq 0 \} \cap \{ a^m b^n c^n \mid n,m \geq 0 \}$
  Both of the above are CFLs
- Complement
  If closed under complement, then would be closed under Intersection as
  $A \cap B = \sim(\sim A \cup \sim B)$

Max and Min of CFL

- Consider the two operations max and min on languages, where
  - max(L) = { x | x ∈ L and, for no non-null y does xy ∈ L } and
  - min(L) = { x | x ∈ L and, for no proper prefix of x, y, does y ∈ L }
- Describe the languages produced by max and min. for each of :
  - L1 = { $a^i b^j c^k$ | k ≤ i or k ≤ j } CFL
    - max(L1) = { $a^i b^j c^k$ | k =max(i, j) } Non-CFL
    - min(L1) = { λ } (string of length 0) Regular
  - L2 = { $a^i b^j c^k$ | k ≥ i or k ≥ j } CFL
    - max(L2) = { } (empty) Regular
    - min(L2) = { $a^i b^j c^k$ | k =min(i, j) } Non-CFL
- max(L1) shows CFL not closed under max
- min(L2) shows CFL not closed under min

Complement of ww

- Let L = { ww | w ∈ $\{a,b\}^+$ }. L is not a CFL
- Consider L's complement, it must be of form xayx'by' or xbyx'ay', where |x|=|x'| and |y|=|y'| or else it's an odd length string
- The hard part above reflects that this language contains even length items with one "transcription error"
- It seems hard to write a CFG but it's all a matter of how you view it
- We don't care about what precedes or follows the errors so long as the lengths are right
- Thus, we can view above as xax'yby' or xbx'y'ay', where |x|=|x'| and |y|=|y'|
- The grammar for this has rules
  S → AB | BA | <ODD>; A → XAX | a ; B → XBX | b
  <ODD> → X | XX <ODD>; X → a | b

Solvable CFL Problems

- Let L be an arbitrary CFL generated by CFG G with start symbol S then the following are all decidable
  - Is w in L?
    Run CKY
    If S in final cell, then w∈L
  - Is L empty (non-empty)?
    Reduce G
    If no rules left, then empty
  - Is L finite (infinite)?
    Reduce G
    Run DFS(S)
    If no loops, then finite

Push Down Automata

CFL Recognizers

Formalization of PDA

- $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
- Q is finite set of states
- Σ is finite input alphabet
- Γ is finite set of stack symbols
- $\delta : Q \times \Sigma_e \times \Gamma_e \rightarrow 2^{Q \times \Gamma^*}$ is transition function
  - Note: Can limit stack push to Γ_e but it's equivalent!!
- $Z_0 \in \Gamma$ is an optional initial symbol on stack
- $F \subseteq Q$ is final set of states and can be omitted for some notions of a PDA

Notion of ID for PDA

- An instantaneous description for a PDA is [q, w, γ] where
  - q is current state
  - w is remaining input
  - γ is contents of stack (leftmost symbol is top)
- Single step derivation is defined by
  - [q,ax,Zα] |— [p,x,$\beta\alpha$] if δ(q,a,Z) contains (p,β)
- Multistep derivation (|—*) is the reflexive transitive closure of single step.

Language Recognized by PDA

- Given A = $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
  there are three senses of recognition
- By final state
  $L(A) = \{w | [q_0,w,Z_0] \vdash^* [f,\lambda,\beta]\}$, where $f \in F$
- By empty stack
  $N(A) = \{w | [q_0,w,Z_0] \vdash^* [q,\lambda,\lambda]\}$
- By empty stack and final state
  $E(A) = \{w | [q_0,w,Z_0] \vdash^* [f,\lambda,\lambda]\}$, where $f \in F$

Top Down Parsing by PDA

- Given G = (V, Σ, R, S), define
  A = ({q}, Σ, Σ∪V, δ, q, S, ϕ)
- $\delta(q,a,a) = \{(q,\lambda)\}$ for all $a \in \Sigma$
- $\delta(q,\lambda,A) = \{(q,\alpha) \mid A \rightarrow \alpha \in R$ (guess) $\}$
- $N(A) = \mathit{L}(G)$

- Has just one state, so is essentially stateless, except for stack content

Example Top Down Parsing by PDA

E → E + T | T

T → T * F | F

F → (E) | Int

- δ(q,+,+) = {(q,λ)}, δ(q,*,*) = {(q,λ)},
- δ(q,Int,Int) = {(q,λ)},
- δ(q,(,() = {(q,λ)}, δ(q,),)) = {(q,λ)}
- δ(q,λ,E) = {(q,E+T), (q,T)}
- δ(q,λ,T) = {(q,T*F), (q,F)}
- δ(q,λ,F) = {(q,(E)), (q,Int)}

Bottom Up Parsing by PDA

- Given G = (V, Σ, R, S), define
  A = ({q,f}, Σ, Σ∪V∪{\$}, δ, q, \$, {f})
- $\delta(q,a,\lambda) = \{(q,a)\}$ for all $a \in \Sigma$, SHIFT
- $\delta(q,\lambda,\alpha^R) \supseteq \{(q,A)\}$ if $A \rightarrow \alpha \in R$, REDUCE
  Cheat: looking at more than top of stack
- $\delta(q,\lambda,S) \supseteq \{(f,\lambda)\}$
- $\delta(f,\lambda,\$) = \{(f,\lambda)\}$, ACCEPT
- E(A) = *L*(G)
- Could also do $\delta(q,\lambda,S\$)\supseteq\{(q,\lambda)\}$, N(A) = *L*(G)

Example Bottom Up Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Int$

- $\delta(q,+,\lambda)=\{(q,+)\}$, $\delta(q,*,\lambda)=\{(q,*)\}$, $\delta(q,Int,\lambda)=\{(q,Int)\}$, $\delta(q,(,\lambda)=\{(q,()\}$, $\delta(q,),\lambda)=\{(q,))\}$
- $\delta(q,\lambda,T+E) = \{(q,E)\}$, $\delta(q,\lambda,T) \supseteq \{(q,E)\}$
- $\delta(q,\lambda,F*T) \supseteq \{(q,T)\}$, $\delta(q,\lambda,F) \supseteq \{(q,T)\}$
- $\delta(q,\lambda,)E() \supseteq \{(q,F)\}$, $\delta(q,\lambda,Int) \supseteq \{(q,F)\}$
- $\delta(q,\lambda,E) \supseteq \{(f,\lambda)\}$
- $\delta(f,\lambda,\$) = \{(f,\lambda)\}$
- $E(A) = \mathit{L}(G)$

Challenge

- Use the two recognizers on some sets of expressions like
  - 5 + 7 * 2
  - 5 * 7 + 2
  - (5 + 7) * 2

Converting a PDA to CFG

- Sipser has one approach; here is another
- Let A = (Q, Σ, Γ, δ, q_0, Z, F) accept L by empty stack and final state
- Define A' = (Q$\cup$\{q_0',f\}, Σ, $\Gamma\cup$\{\$\}, δ', q_0', \$, \{f\}) where
  - δ'(q_0', λ, \$) = \{(q_0, PUSH(Z)) or in normal notation \{(q_0, Z\$)\}
  - δ' does what δ does but only uses PUSH and POP instructions, always reading top of stack
    Note1: we need to consider using the \$ for cases of the original machine looking at empty stack, when using λ for stack check. This guarantees we have top of stack until very end.
    Note2: If original adds stuff to stack, we do pop, followed by a bunch of pushes.
  - We add (f, λ) = (f, POP) to δ'(q_f, λ, \$) whenever q_f is in F, so we jump to a fixed final state.
- Now, wlog, we can assume our PDA uses only POP and PUSH, has just one final state and accepts by empty stack and final state. We will assume the original machine is of this form and that its bottom of stack is \$.
- Define G = (V, Σ, R, S) where
  - V = \{S\} $\cup$ \{ <q, X, p> | q,p $\in$ Q, X $\in$ Γ \}
  - R on next page

Rules for PDA to CFG

- R contains rules as follows:
  $S \to \langle q_0,\$,f\rangle$ where $F = \{f\}$
  meaning that we want to generate w whenever
  $[q_0,w,\$] \vdash\!\!\!\!—^{*}[f,\lambda,\lambda]$
- Remaining rules are:
  $\langle q,X,p\rangle \to a\langle s,Y,t\rangle\langle t,X,p\rangle$
  whenever $\delta(q,a,X) \supseteq \{(s,\mathrm{PUSH}(Y))\}$
  $\langle q,X,p\rangle \to a$
  whenever $\delta(q,a,X) \supseteq \{(p,\mathrm{POP})\}$
- Want $\langle q,X,p\rangle \Rightarrow^{*} w$ when $[q,w,X] \vdash\!\!\!\!—^{*}[p,\lambda,\lambda]$

Closure Properties

Context Free Languages

Intersection with Regular

- CFLs are closed under intersection with Regular sets
  - To show this we use the equivalence of CFGs generative power with the recognition power of PDAs (shown later).
  - Let $A_0 = (Q_0, \Sigma, \Gamma, \delta_0, q_0, \$, F_0)$ be an arbitrary PDA
  - Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an arbitrary DFA
  - Define $A_2 = (Q_0 \times Q_1, \Sigma, \Gamma, \delta_2, <q_0,q_1> \$, F_0 \times F_1)$ where

    $\delta_2(<q,s>, a, X) \supseteq \{(<q',s'>, \alpha)\}, a\in\Sigma\cup\{\lambda\}, X\in\Gamma$ iff
    $\delta_0(q, a, X) \supseteq \{(q', \alpha)\}$ and $\delta_1(s,a) = s'$ (if $a=\lambda$ then $s' = s$).
  - Using the definition of derivation, we see that

    $[<q_0,q_1>, w, \$] \vdash^* [<t,s>, \lambda, \beta]$ in A_2 iff
    $[q_0, w, \$] \vdash^* [t, \lambda, \beta]$ in A_0 and
    $[q_1, w] \vdash^* [s, \lambda]$ in A_1

    But then $w\in L(A_2)$ iff $t\in F_0$ and $s\in F_1$ iff $w \in L(A_0)$ and $w \in L(A_1)$

Substitution

- CFLs are closed under CFL substitution
  - Let $G=(V,\Sigma,R,S)$ be a CFG
  - Let f be a substitution over Σ such that
    - $f(a) = L_a$ for $a \in \Sigma$
    - $G_a = (V_a,\Sigma_a,R_a,S_a)$ is a CFG that produces L_a.
    - No symbol appears in more than one of V or any V_a
  - Define $G_f = (V \cup_{a\in\Sigma}V_a, \cup_{a\in\Sigma}\Sigma_a, R' \cup_{a\in\Sigma}R_a, S)$
    - $R' = \{ A \to g(\alpha)$ where $A \to \alpha$ is in $R \}$
    - $g: (V\cup\Sigma)^* \to (V \cup_{a\in\Sigma}S_a)^*$
    - $g(\lambda) = \lambda$; $g(B) = B$, $B \in V$; $g(a) = S_a$, $a \in \Sigma$
    - $g(\alpha X) = g(\alpha)\ g(X)$, $|\alpha| > 0$, $X \in V\cup\Sigma$
  - Claim, $f(L(G)) = L(G_f)$, and so CFLs closed under substitution and homomorphism.

More on Substitution

- Consider G'$_f$. If we limit derivations to the rules R' = { $A \to g(\alpha)$ where $A \to \alpha$ is in R } and consider only sentential forms over the $\cup_{a \in \Sigma} S_a$, then $S \Rightarrow^* S_{a1}\ S_{a2}\ \ldots\ S_{an}$ in G' iff $S \Rightarrow^*$ a1 a2 … an in G iff a1 a2 … an $\in$ *L*(G). But, then w $\in$ *L*(G) iff f(w) $\in$ *L*(G_f) and, thus, f(*L*(G)) = *L*(G_f).
- Given that CFLs are closed under union, substitution, homomorphism and intersection with regular sets, we can recast previous proofs to show that CFLs are closed under
  - Prefix, Suffix, Substring, Quotient with Regular Sets
- Later we will show that CFLs are not closed under Quotient with CFLs.

Context Sensitive

Will revisit on Complexity Theory

Context Sensitive Grammar

G = (V, Σ, R, S) is a PSG where

Each member of R is a rule whose right side is no shorter than its left side.

The essential idea is that rules are length preserving, although we do allow $S \rightarrow \lambda$ so long as S never appears on the right-hand side of any rule.

A context sensitive grammar is denoted as a CSG and the language generated is a Context Sensitive Language (CSL).

The recognizer for a CSL is a Linear Bounded Automaton (LBA), a form of Turing Machine (soon to be discussed), but with the constraint that it is limited to moving along a tape that contains just the input surrounded by a start and end symbol.

Phrase Structured Grammar

We previously defined PSGs. The language generated by a PSG is a Phrase Structured Language (PSL) but is more commonly called a recursively enumerable (re) language. The reason for this will become evident a bit later in the course.

The recognizer for a PSL (re language) is a Turing Machine, a model of computation we will soon discuss.

CSG Example#1

$L = \{ a^n b^n c^n \mid n>0 \}$

G = ({A,B,C}, {a,b,c}, R, A) where R is

$A \rightarrow aBbc \mid abc$

$B \rightarrow aBbC \mid abC$

Note: $A \Rightarrow aBbc \Rightarrow^n a^{n+1}(bC)^n\, bc$ // n>0

$Cb \rightarrow bC$ // Shuttle C over to a c

$Cc \rightarrow cc$ // Change C to a c

Note: $a^{n+1}(bC)^n\, bc \Rightarrow^* a^{n+1}b^{n+1}c^{n+1}$

Thus, $A \Rightarrow^* a^n b^n c^n$, n>0

CSG Example#2

$L = \{ ww \mid w \in\{0,1\}^+ \}$

G = ({S,A,X,Z,<0>,<1>}, {0,1}, R, S) where R is

S → 00 | 11 | 0A<0> | 1A<1>

A → 0AZ | 1AX | 0Z | 1X

Z0 → 0Z	Z1 → 1Z	// Shuttle Z (for owe zero)
X0 → 0X	X1 → 1X	// Shuttle X (for owe one)
Z<0> → 0<0>	Z<1> → 1<0>	// New 0 must be on rhs of old 0/1’s
X<0> → 0<1>	X<1> → 1<1>	// New 1 must be on rhs of old 0/1’s
<0> → 0		// Guess we are done
<1> → 1		// Guess we are done

Side Commentary

These are slides that will not be discussed directly but summarize side comments I have made

Lexical Analysis

In earlier discussions I have alluded to the fact that compilers have two early phases – lexical and syntax analysis.

Lexical analysis typically is driven by regular expressions that specify keywords, operators, special symbols, and identifiers. The job of lexical analysis is to identify and characterize these components, called tokens or lexemes, so syntax analysis can ignore individual characters, treating categories as terminals, e.g., **age** and **height** are both identifiers, but **while** is a keyword, and both * and / are multiplicative operators.

Syntax Analysis

As noted, syntax analysis depends on lexical analysis to acquire the tokens associated with an input stream (a presumed program in some source language).

While lexical analysis has no concept of what tokens mean when they are contextually laid out, syntax analysis understands the structure of a valid program. Its job is to check for syntax errors (bad structural combinations) and semantic issues related to type mismatch, use before definition, or function signature errors – these are possible due to the use of a symbol table even though not checkable based on pure grammatical analysis.

Order Analysis

Lexical analysis can be done with a DFA and so this is an O(N) process.

The cost of syntax analysis using a CFG is bounded above by O($N^{2.37}$) due to CKY but we don't use arbitrary CFGs because we are focused on unambiguous CFLs and can use either an LR(1) (any unambiguous CFL) or LL(1) (any useful unambiguous CFL) grammar. This gets us O(N) parsing but requires the use of a symbol table to get around context sensitive issues.

It seems we could go to Context Sensitive Grammars (CSGs) to handle typing, etc., but that might put us in (2^N) territory, so we avoid this approach in practice.