Welcome to the Stanford Automata Theory Course

Why Study Automata?

What the Course is About

Why Study Automata?

- A survey of Stanford grads 5 years out asked which of their courses did they use in their job.
- Basics like intro-programming took the top spots, of course.
- But among optional courses, CS154 stood remarkably high.
  - 3X the score for AI, for example.

How Could That Be?

- Regular expressions are used in many systems.
  - E.g., UNIX a.*b.
  - E.g., DTD's describe XML tags with a RE format like person (name, addr, child*).
- Finite automata model protocols, electronic circuits.

How? – (2)

- Context-free grammars are used to describe the syntax of essentially every programming language.
  - Not to forget their important role in describing natural languages.
- And DTD's taken as a whole, are really CFG's.

How? – (3)

- When developing solutions to real problems, we often confront the limitations of what software can do.
  - *Undecidable* things – no program whatever can do it.
  - *Intractable* things – there are programs, but no fast programs.
- Automata theory gives you the tools.

Other Good Stuff

- We’ll learn how to deal formally with discrete systems.
  - Proofs: You never really prove a program correct, but you need to be thinking of why a tricky technique really works.
- We’ll gain experience with abstract models and constructions.
  - Models layered software architectures.

Automata Theory – Gateway Drug

- This theory has attracted people of a mathematical bent to CS, to the betterment of all.
    - Ken Thompson – before UNIX was working on compiling regular expressions.
    - Jim Gray – thesis was automata theory before he got into database systems and made fundamental contributions there.

Course Outline

- Regular Languages and their descriptors:
  - Finite automata, nondeterministic finite automata, regular expressions.
  - Algorithms to decide questions about regular languages, e.g., is it empty?
  - Closure properties of regular languages.

Course Outline – (2)

- Context-free languages and their descriptors:
  - Context-free grammars, pushdown automata.
  - Decision and closure properties.

Course Outline – (3)

- Recursive and recursively enumerable languages.
  - Turing machines, decidability of problems.
  - The limit of what can be computed.
- Intractable problems.
  - Problems that (appear to) require exponential time.
  - NP-completeness and beyond.

Text (Not Required)

- Hopcroft, Motwani, Ullman, *Automata Theory, Languages, and Computation* 3rd Edition.
- Course covers essentially the entire book.

Finite Automata

What Are They?

Who Needs 'em?

An Example: Scoring in Tennis

What is a Finite Automaton?

- A formal system.
- Remembers only a finite amount of information.
- Information represented by its *state*.
- State changes in response to *inputs*.
- Rules that tell how the state changes in response to inputs are called *transitions*.

Why Study Finite Automata?

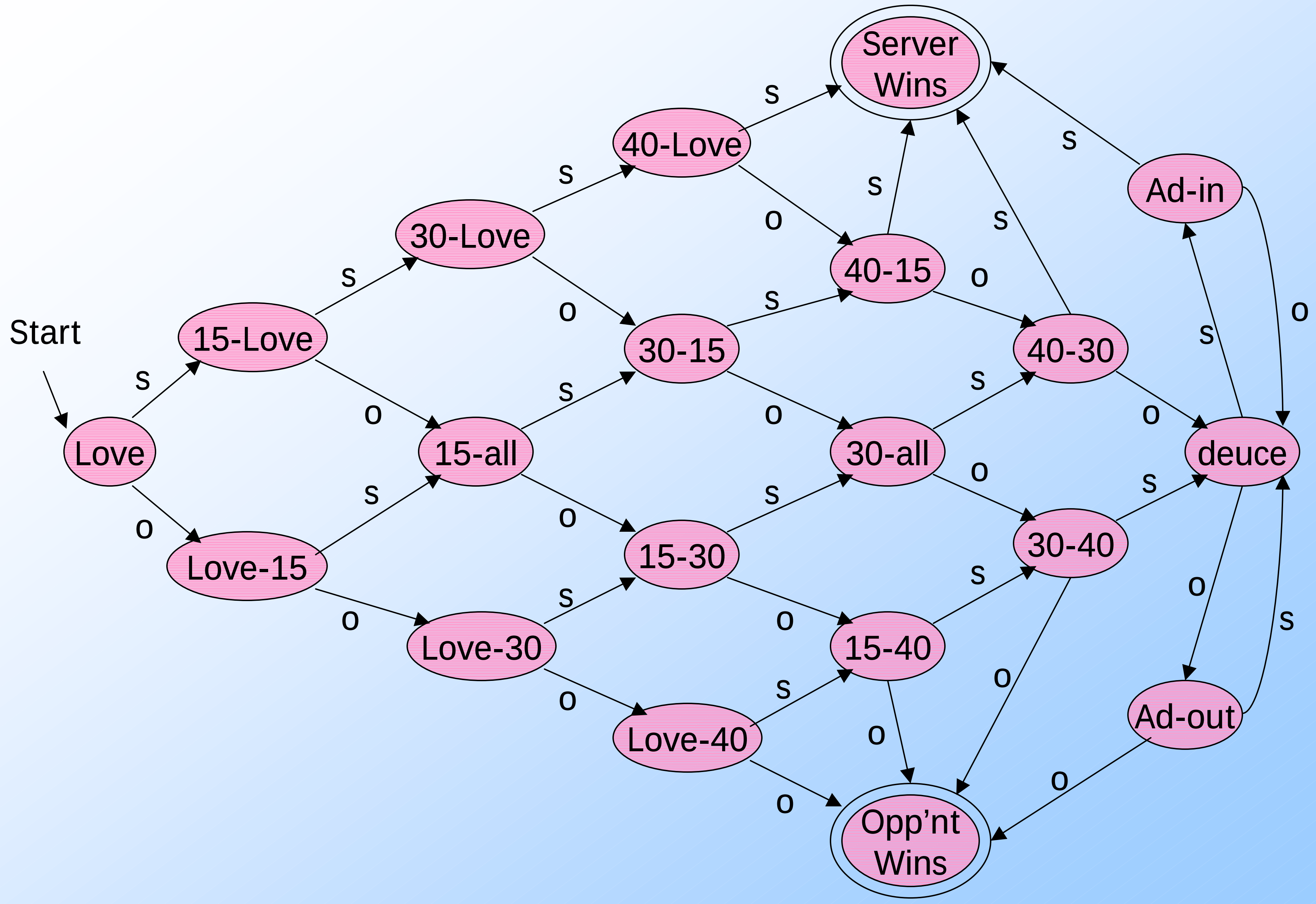
- Used for both design and verification of circuits and communication protocols.
- Used for many text-processing applications.
- An important component of compilers.
- Describes simple patterns of events, etc.

Tennis

- Like ping-pong, except you are very tiny and stand on the table.
- *Match* = 3-5 sets.
- *Set* = 6 or more games.

Scoring a Game

- One person serves throughout.
- To win, you must score at least 4 points.
- You also must win by at least 2 points.
- Inputs are s = “server wins point” and o = “opponent wins point.”

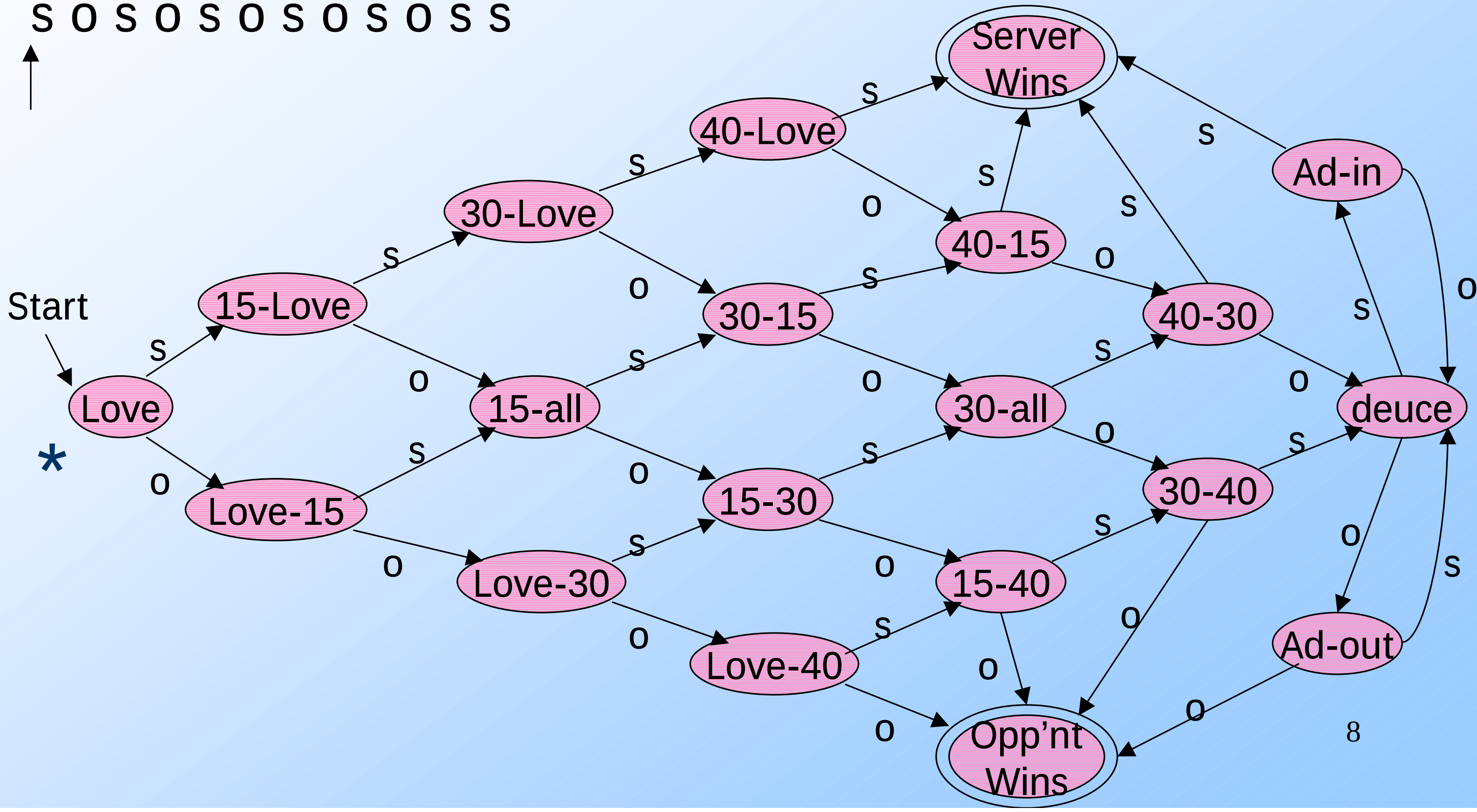

Start
Love
15-Love
Love-15
30-Love
15-all
Love-30
40-Love
30-15
15-30
Love-40
40-15
30-all
15-40
40-30
30-40
deuce
Ad-in
Ad-out
Server Wins
Opp'nt Wins
s
o

Acceptance of Inputs

- Given a sequence of inputs (*input string*), start in the start state and follow the transition from each symbol in turn.
- Input is *accepted* if you wind up in a final (accepting) state after all inputs have been read.

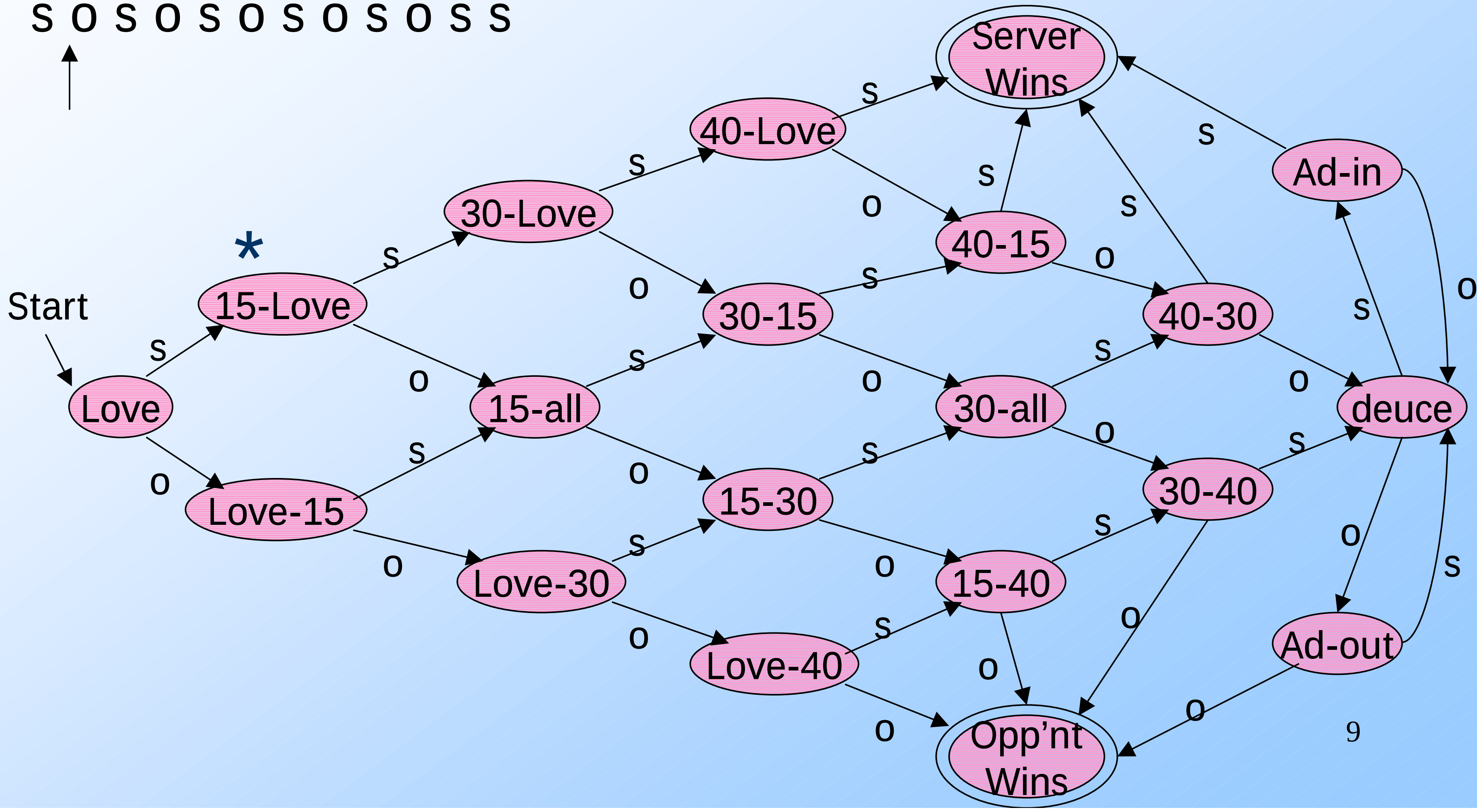
Example: Processing a String

s o s o s o s o s o s o s s



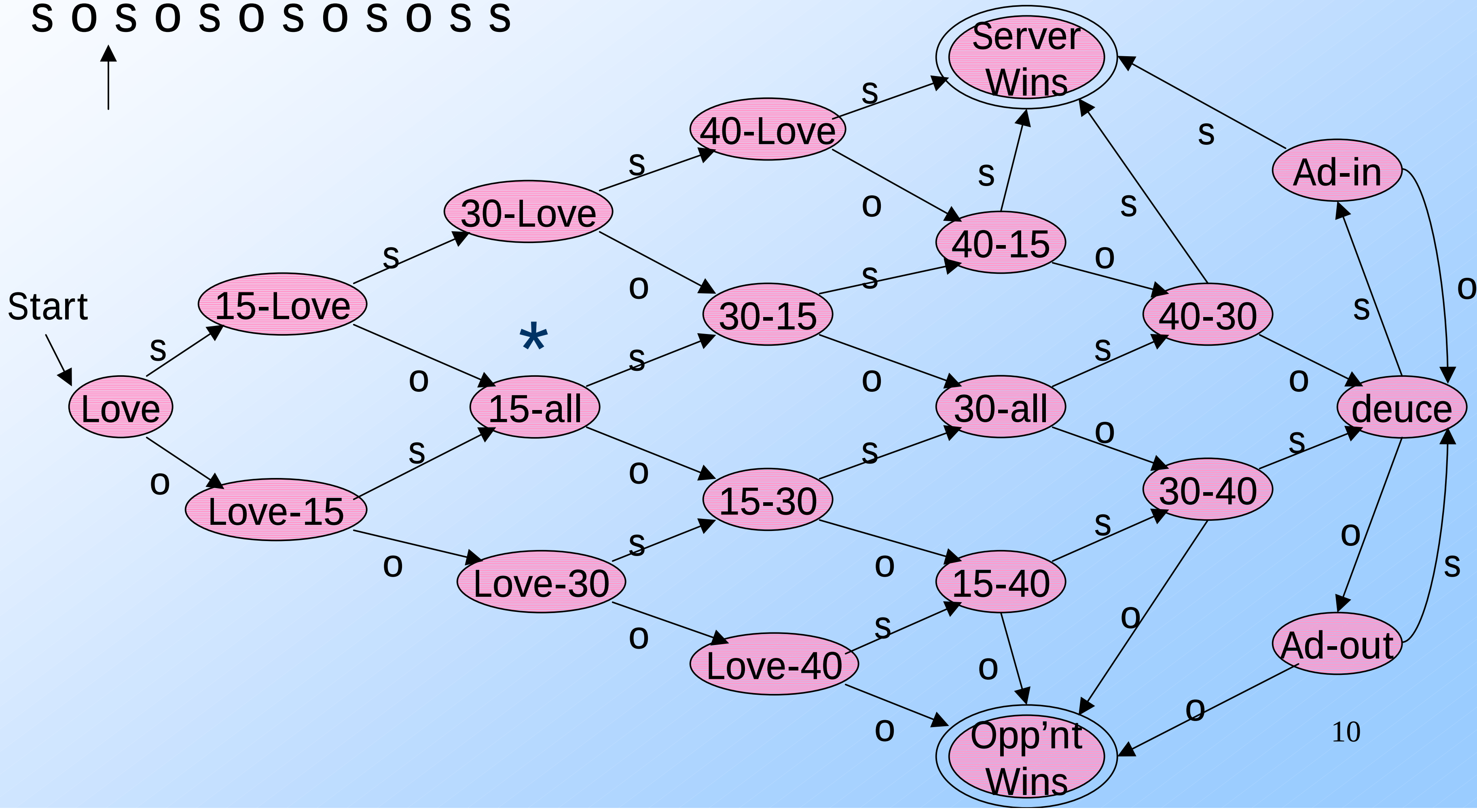
Example: Processing a String

s o s o s o s o s o s s



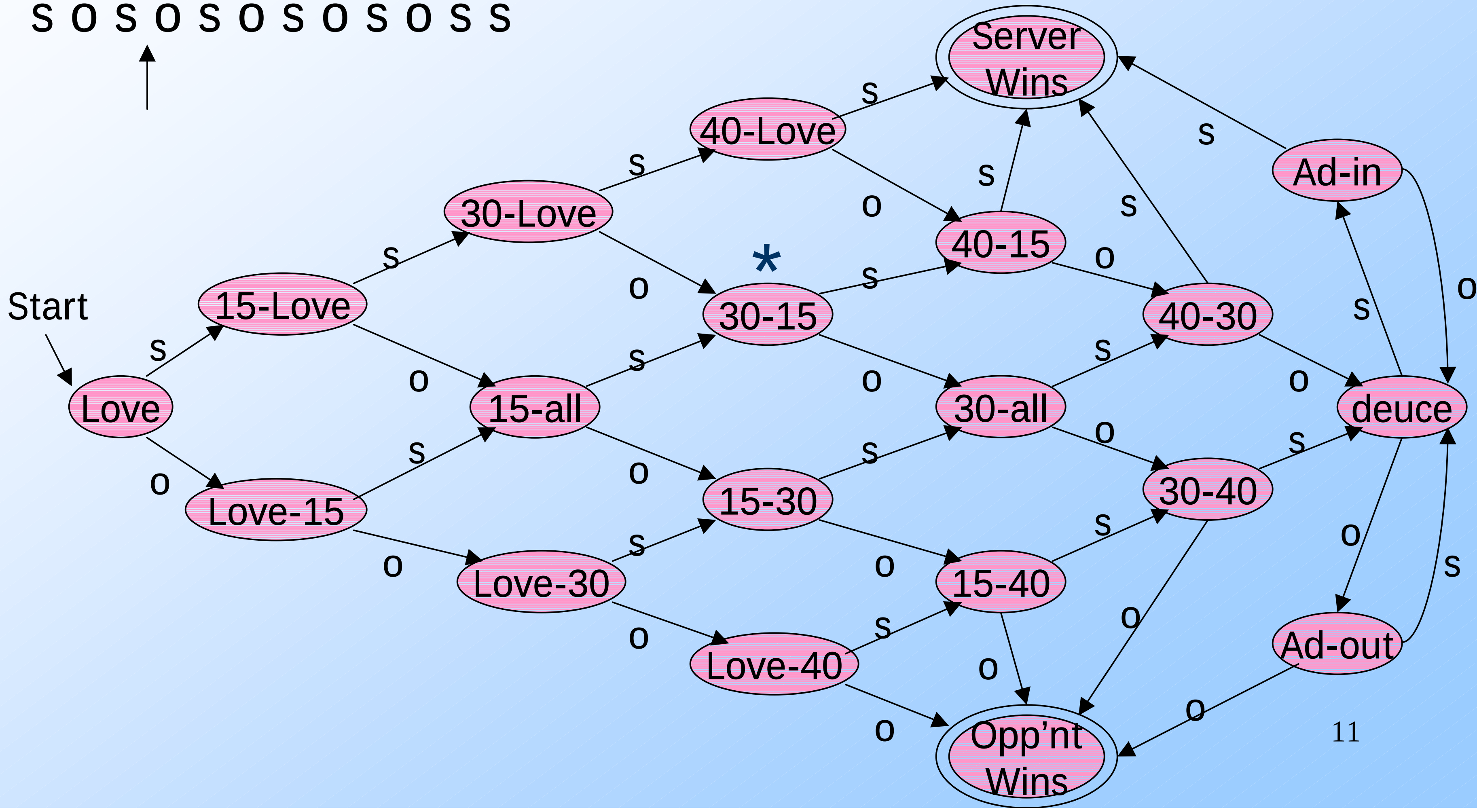
Example: Processing a String

s o s o s o s o s o s o s s



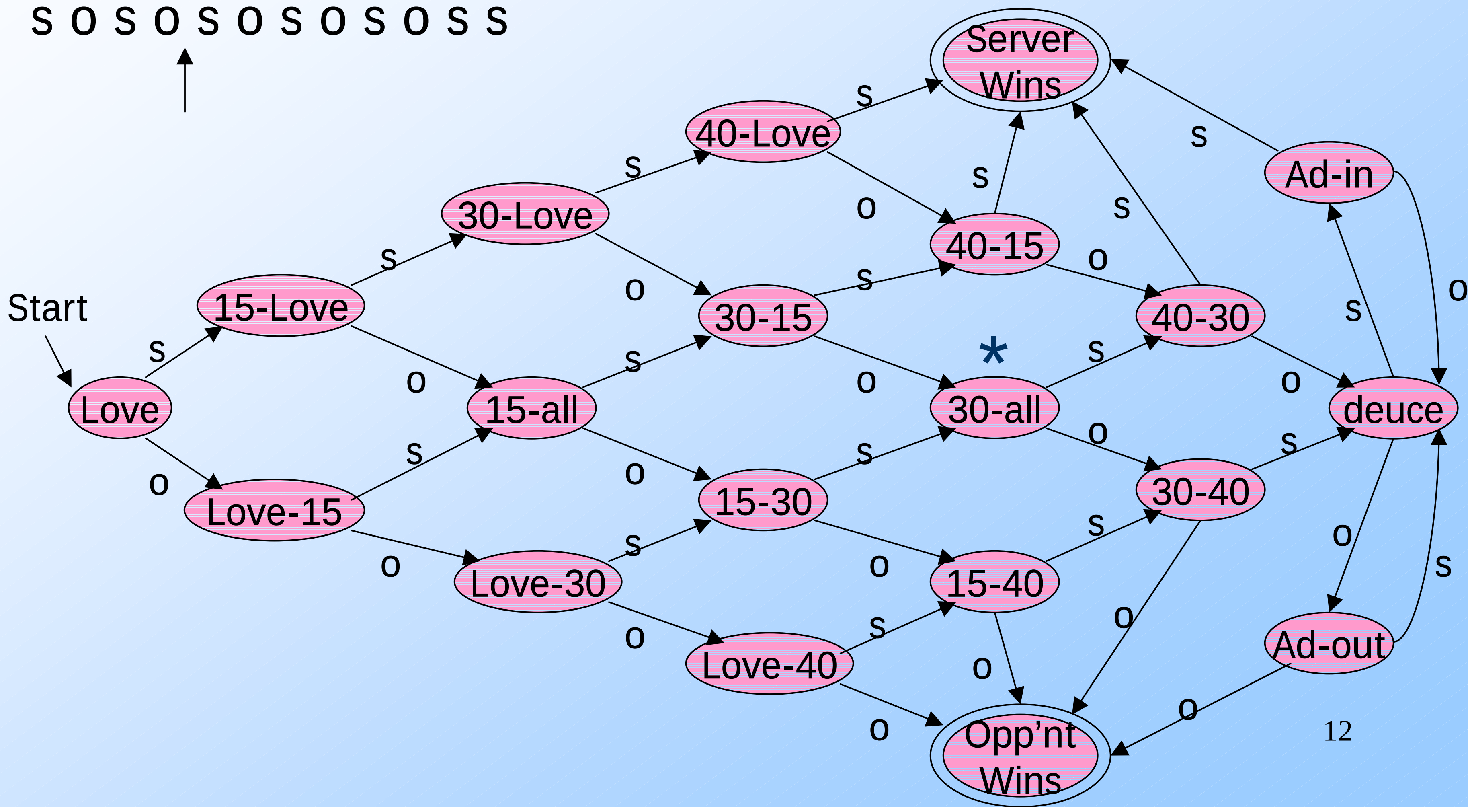
Example: Processing a String

s o s o s o s o s o s s



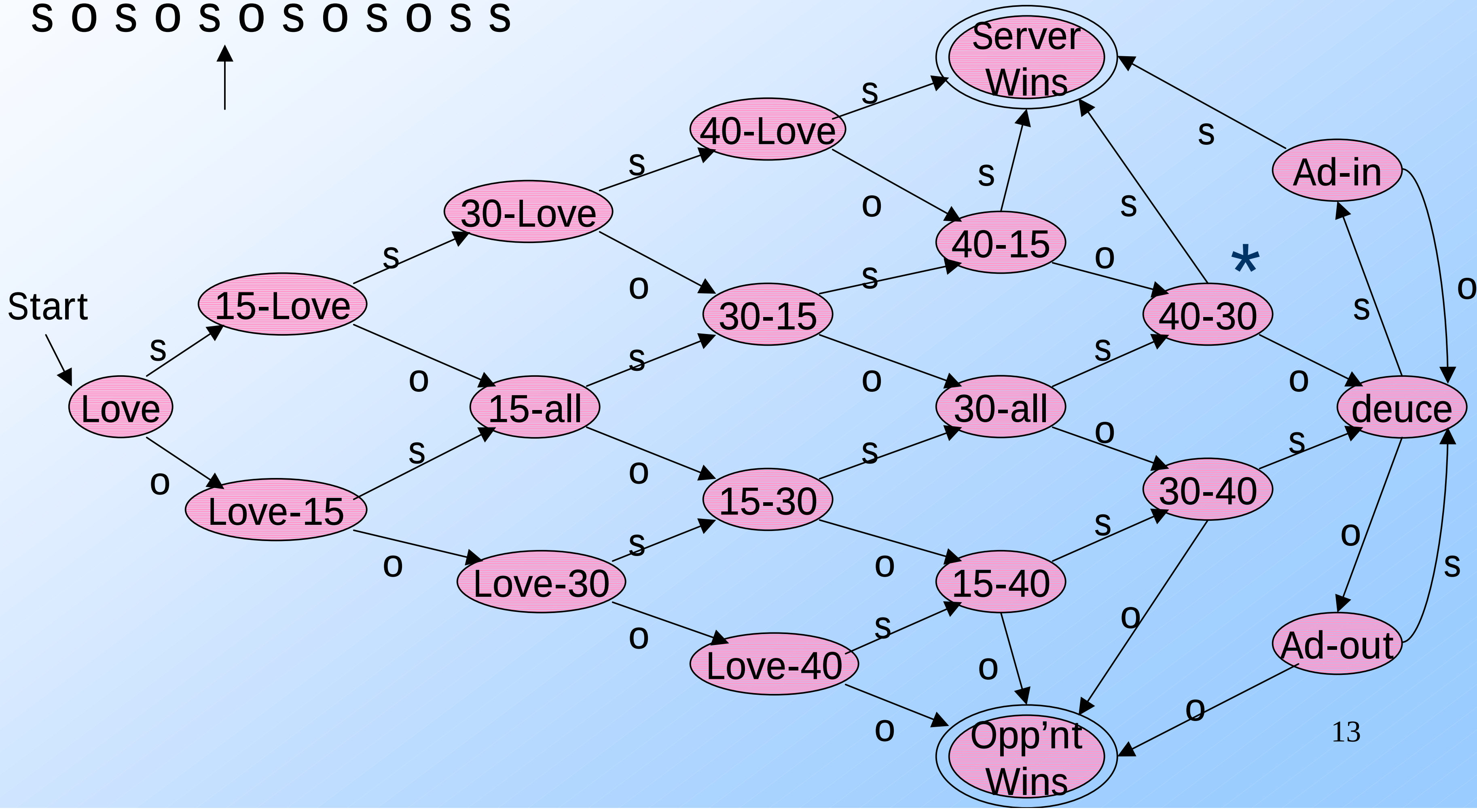
Example: Processing a String

s o s o s o s o s o s s



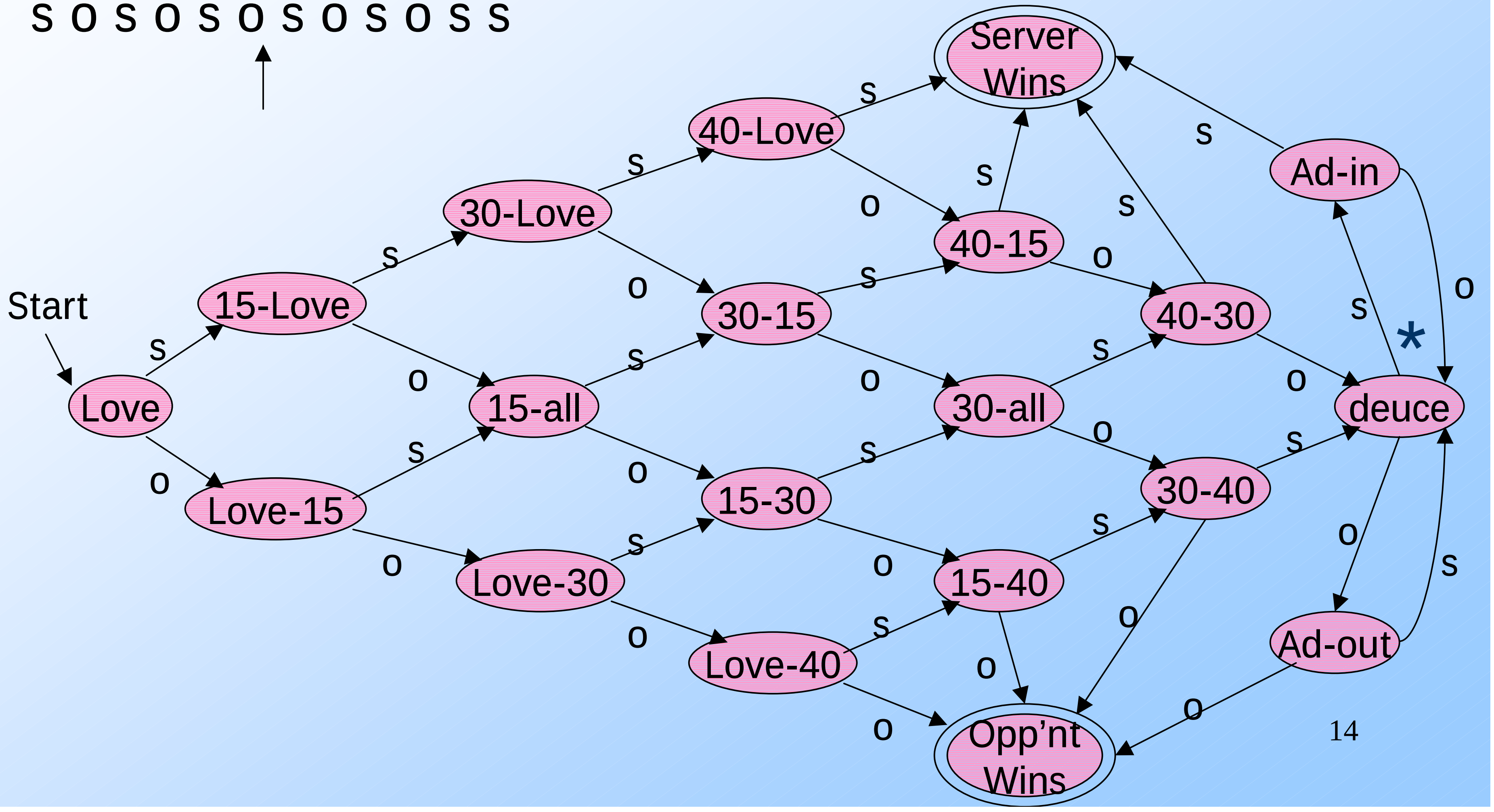
Example: Processing a String

s o s o s o s o s o s s



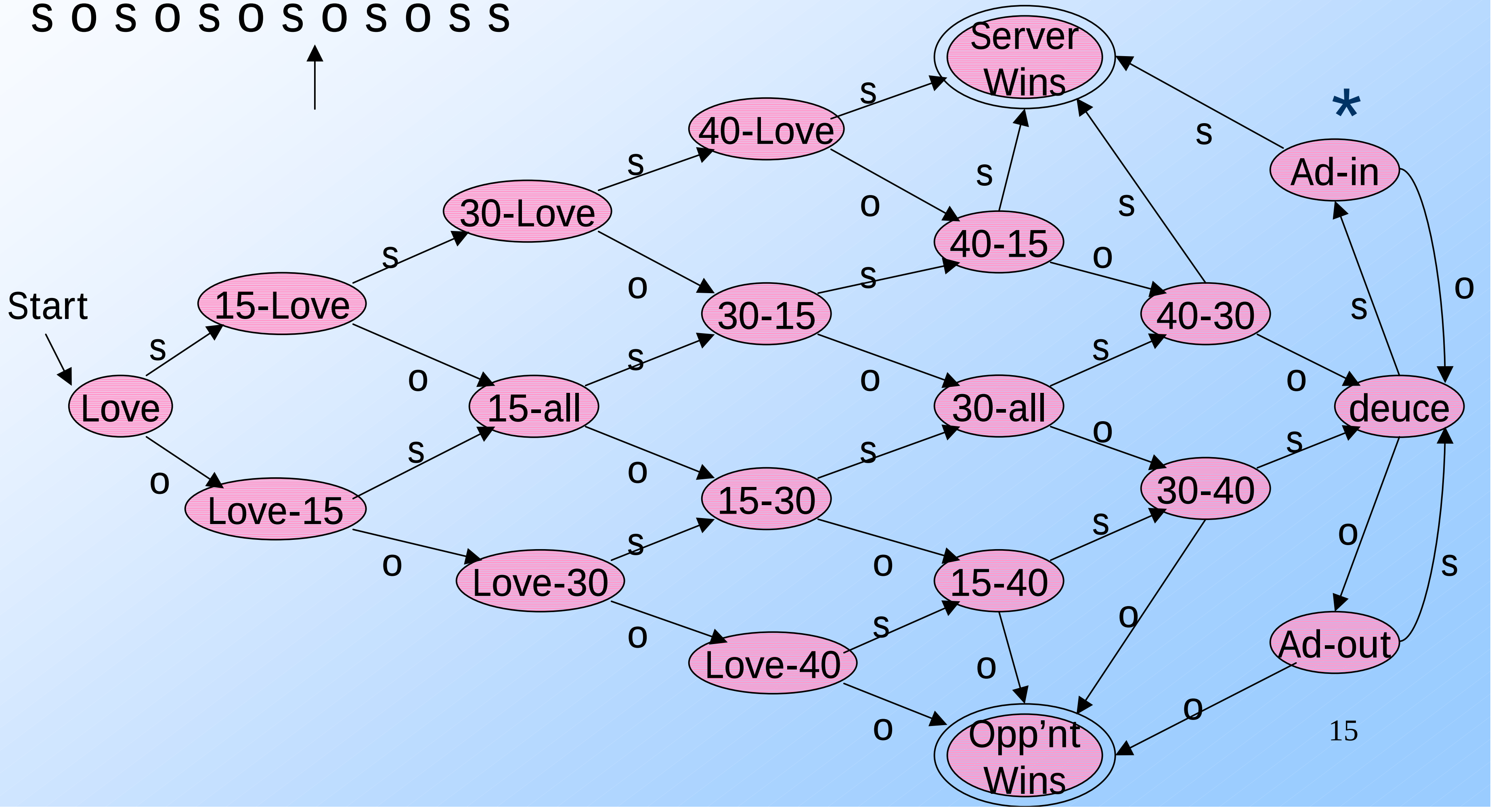
Example: Processing a String

s o s o s o s o s o s s



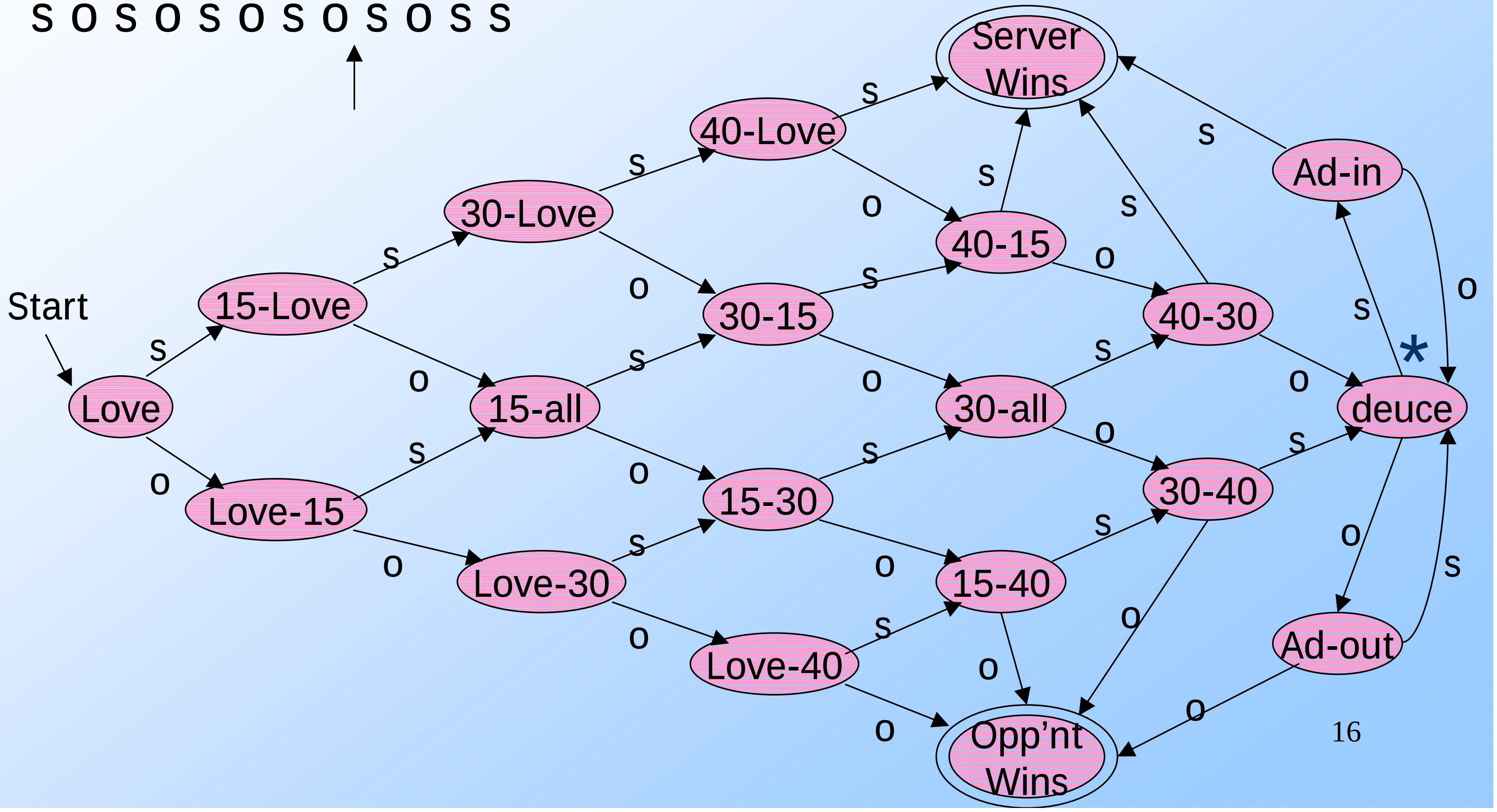
Example: Processing a String

s o s o s o s o s o s s



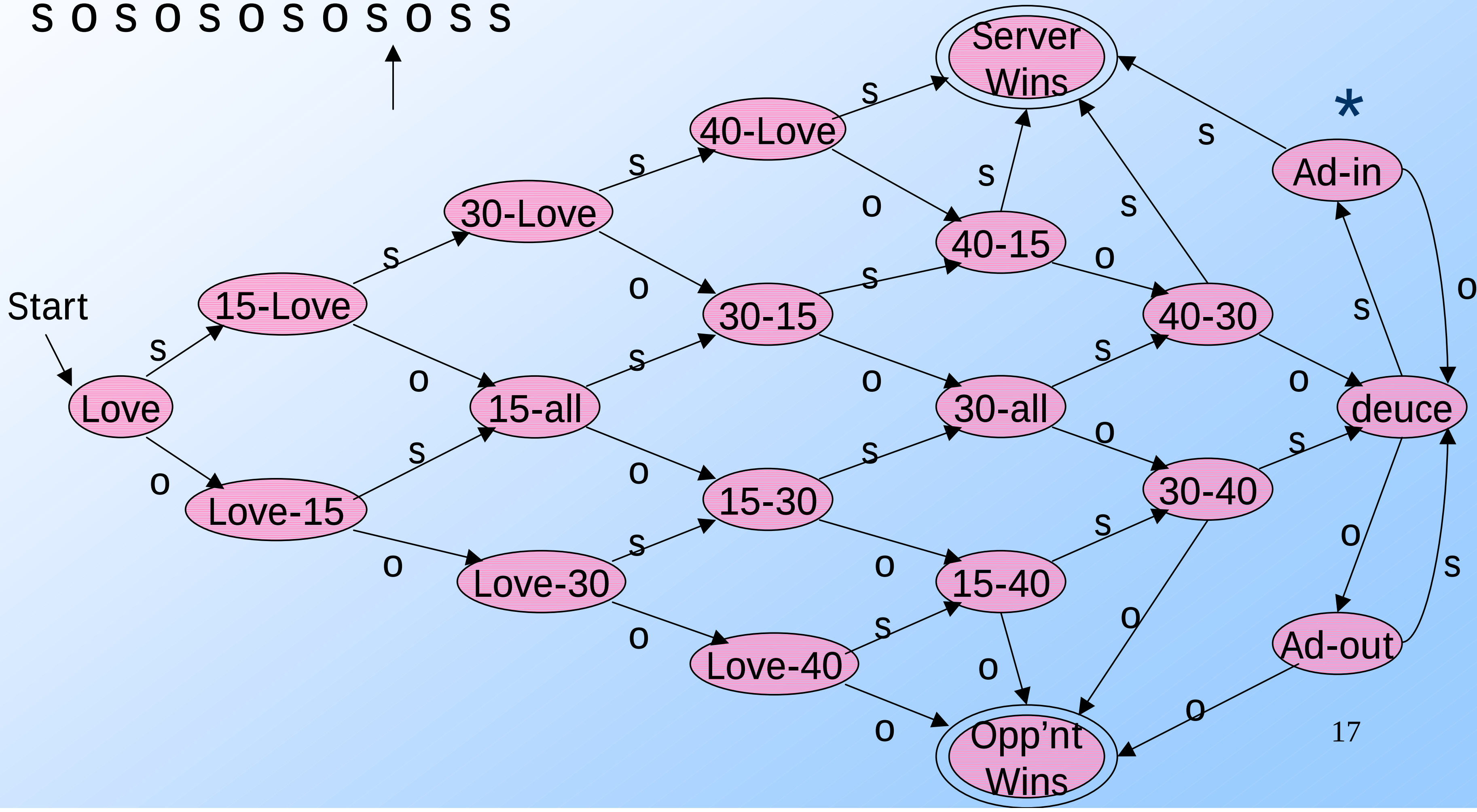
Example: Processing a String

s o s o s o s o s o s o s s



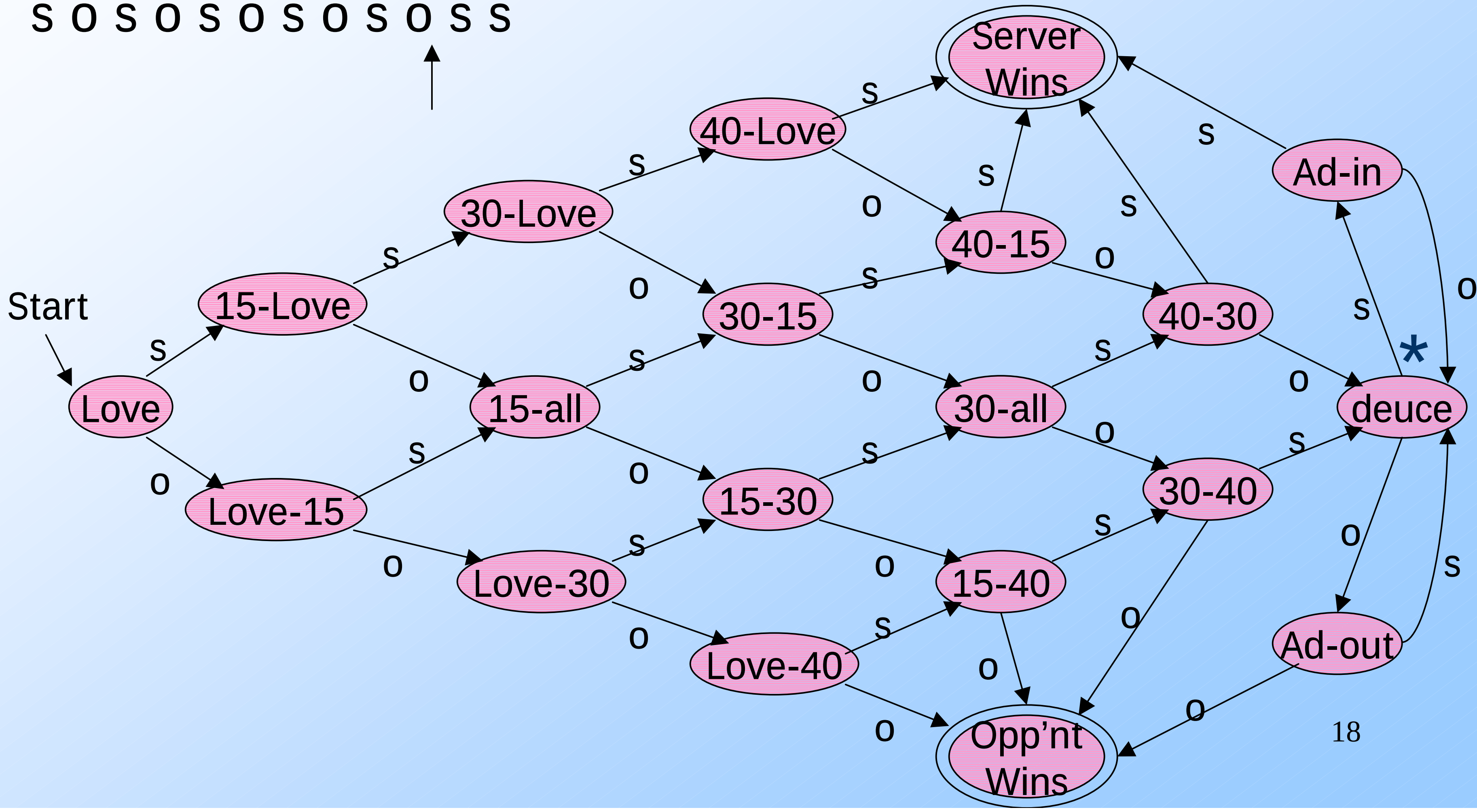
Example: Processing a String

s o s o s o s o s o s o s s



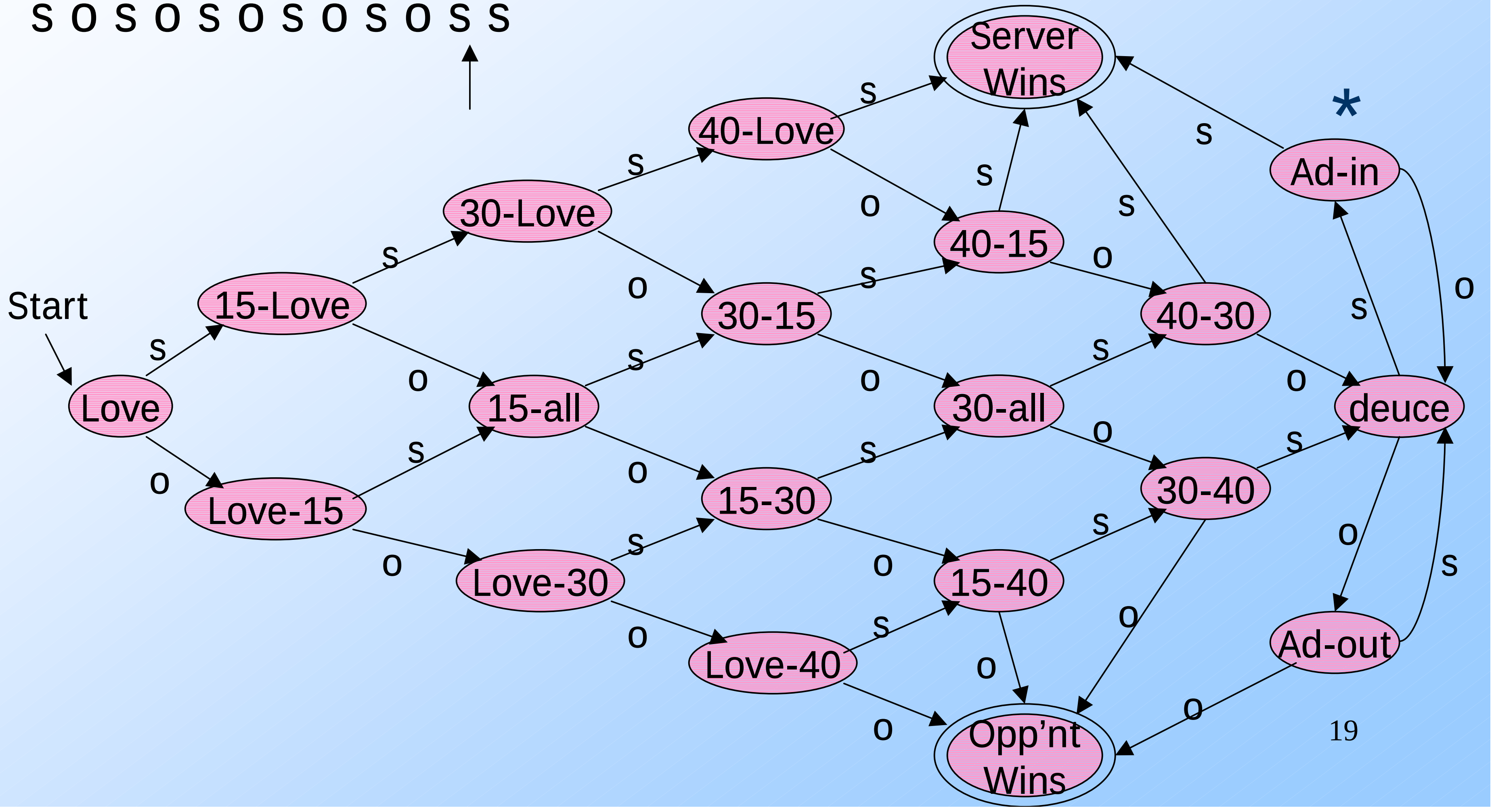
Example: Processing a String

s o s o s o s o s o s o s s

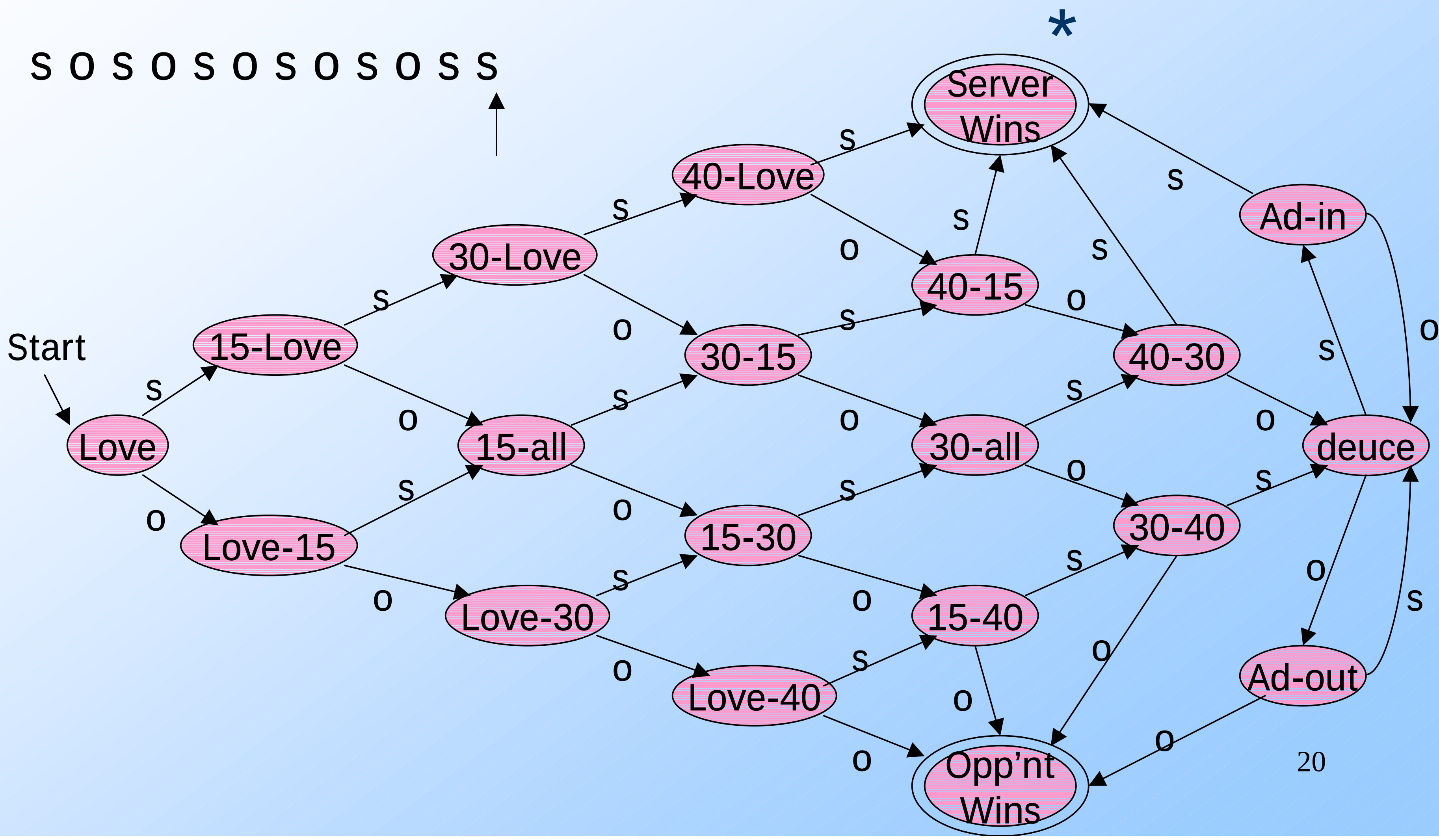


Example: Processing a String

s o s o s o s o s o s s



Example: Processing a String



Language of an Automaton

- The set of strings accepted by an automaton A is the *language* of A.
- Denoted L(A).
- Different sets of final states -> different languages.
- Example: As designed, L(Tennis) = strings that determine the winner.

Deterministic Finite Automata

Alphabets, Strings, and Languages

Transition Graphs and Tables

Some Proof Techniques

Alphabets

- An *alphabet* is any finite set of symbols.
- Examples:

  ASCII, Unicode,

  {0,1} (*binary alphabet*),

  {a,b,c}, {s,o},

  set of signals used by a protocol.

Strings

- A *string* over an alphabet Σ is a list, each element of which is a member of Σ.
  - Strings shown with no commas or quotes, e.g., abc or 01101.
- Σ* = set of all strings over alphabet Σ.
- The *length* of a string is its number of positions.
- ε stands for the *empty string* (string of length 0).

Example: Strings

- $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
- Subtlety: 0 as a string, 0 as a symbol look the same.
  - Context determines the type.

Languages

- A *language* is a subset of Σ* for some alphabet Σ.
- Example: The set of strings of 0's and 1's with no two consecutive 1's.
- L = {ε, 0, 1, 00, 01, 10, 000, 001, 010, 100, 101, 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010, . . . }

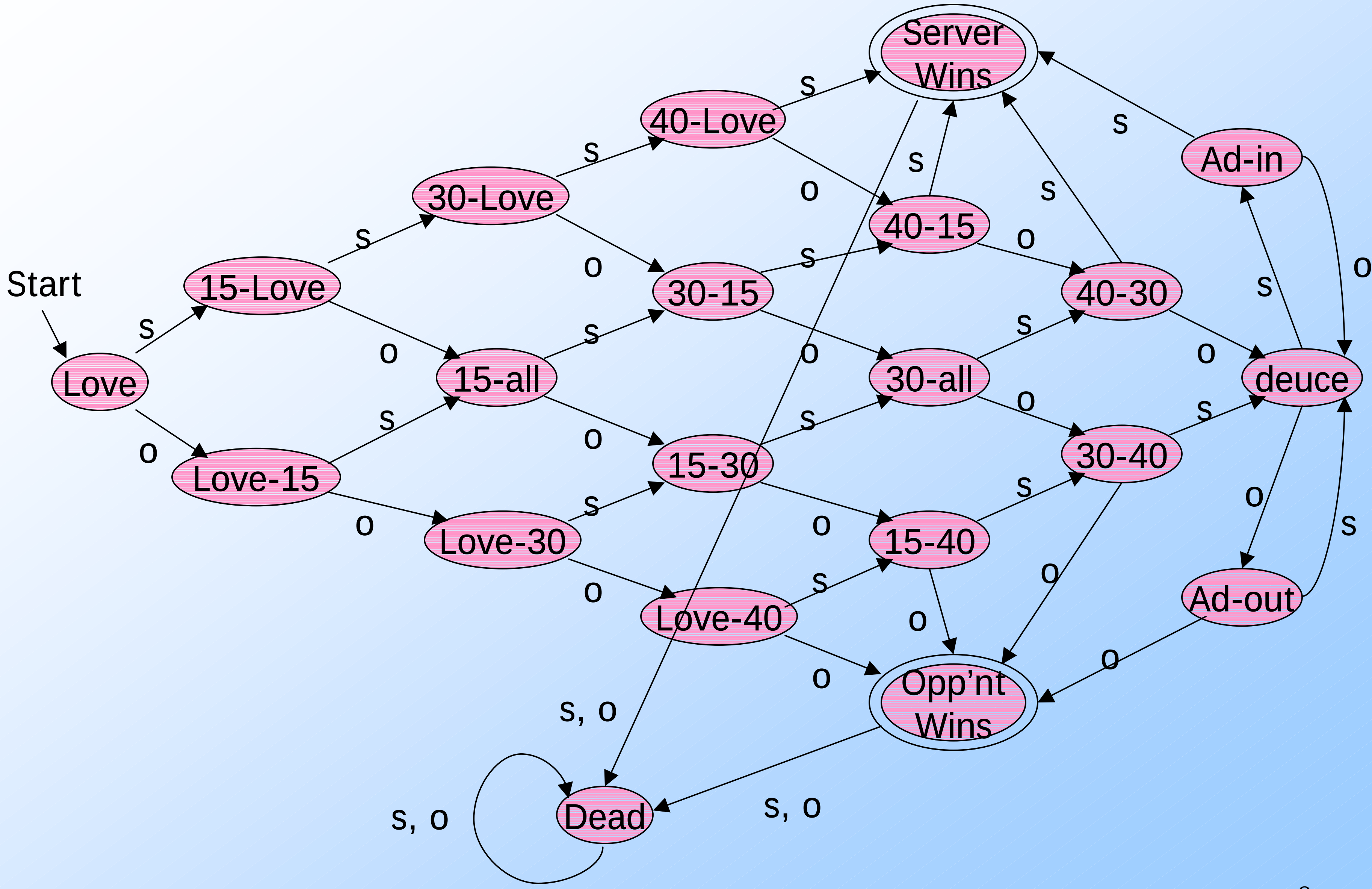
Hmm... 1 of length 0, 2 of length 1, 3, of length 2, 5 of length 3, 8 of length 4. I wonder how many of length 5?

Deterministic Finite Automata

- A formalism for defining languages, consisting of:
  1. A finite set of *states* (Q, typically).
  2. An *input alphabet* (Σ, typically).
  3. A *transition function* (δ, typically).
  4. A *start state* (q_0, in Q, typically).
  5. A set of *final states* ($F \subseteq Q$, typically).
     - "Final" and "accepting" are synonyms.

The Transition Function

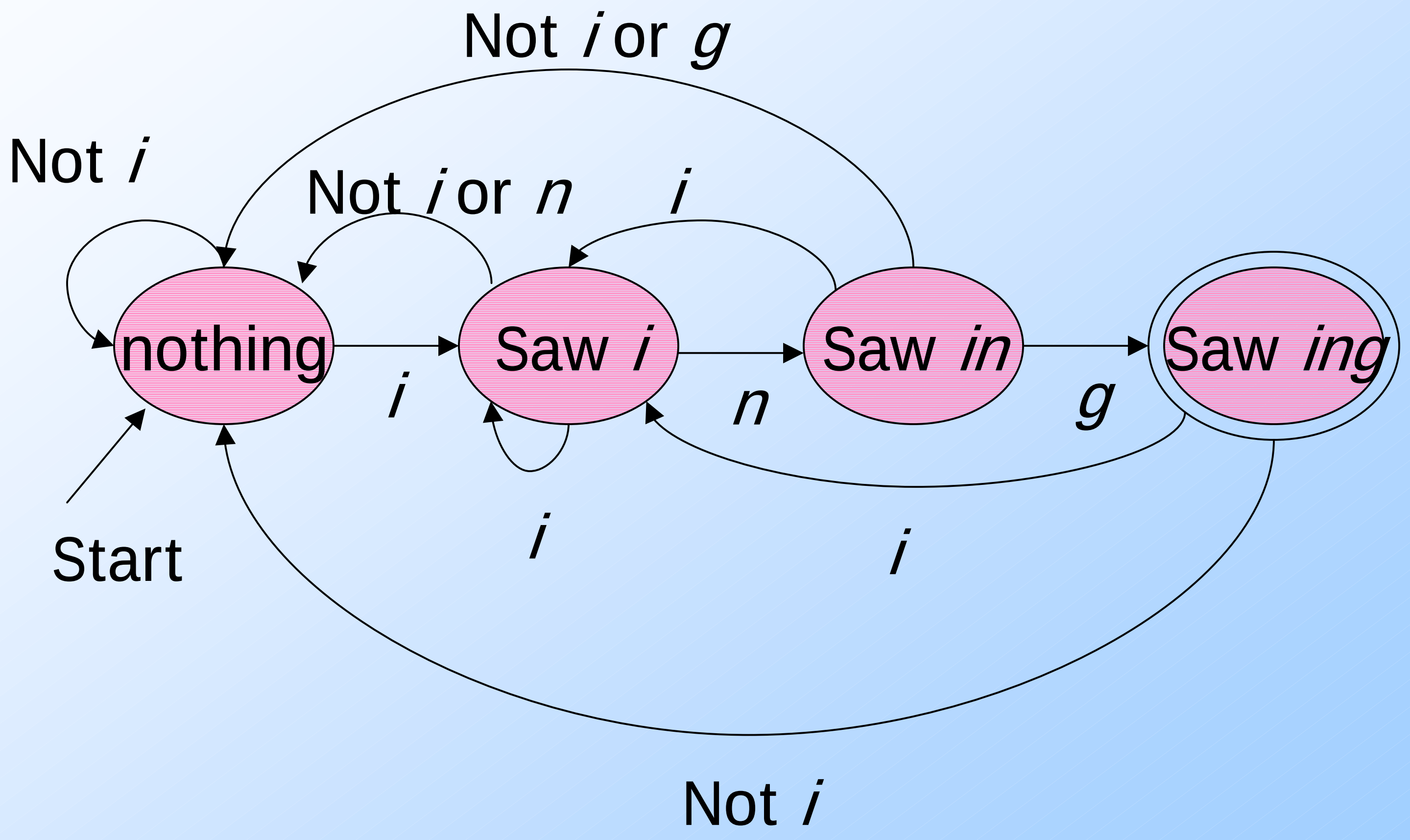
- Takes two arguments: a state and an input symbol.
- δ(q, a) = the state that the DFA goes to when it is in state *q* and input *a* is received.
- Note: always a next state – add a *dead state* if no transition (Example on next slide).


Start
Love
15-Love
Love-15
30-Love
15-all
Love-30
40-Love
30-15
15-30
Love-40
Server Wins
40-15
30-all
15-40
Opp'nt Wins
40-30
30-40
Ad-in
deuce
Ad-out
Dead
s
o
s, o

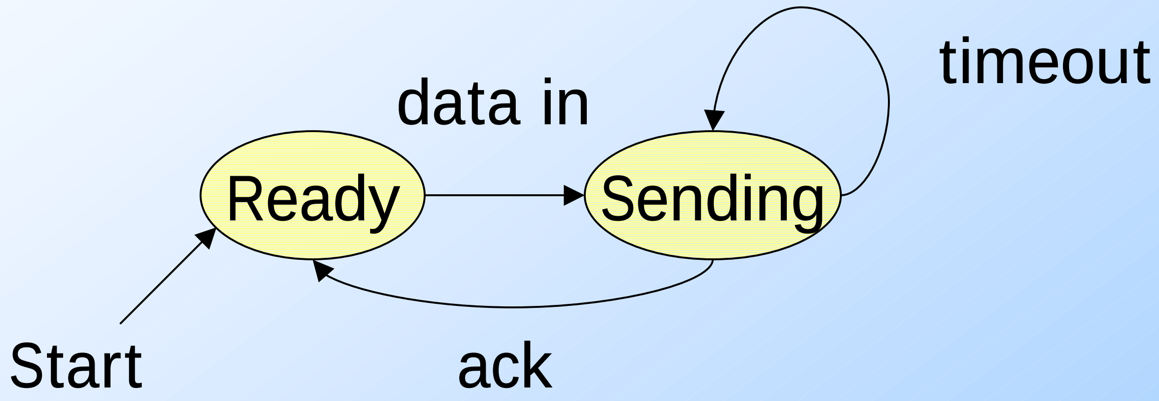
Graph Representation of DFA’s

- Nodes = states.
- Arcs represent transition function.
  - Arc from state p to state q labeled by all those input symbols that have transitions from p to q.
- Arrow labeled “Start” to the start state.
- Final states indicated by double circles.

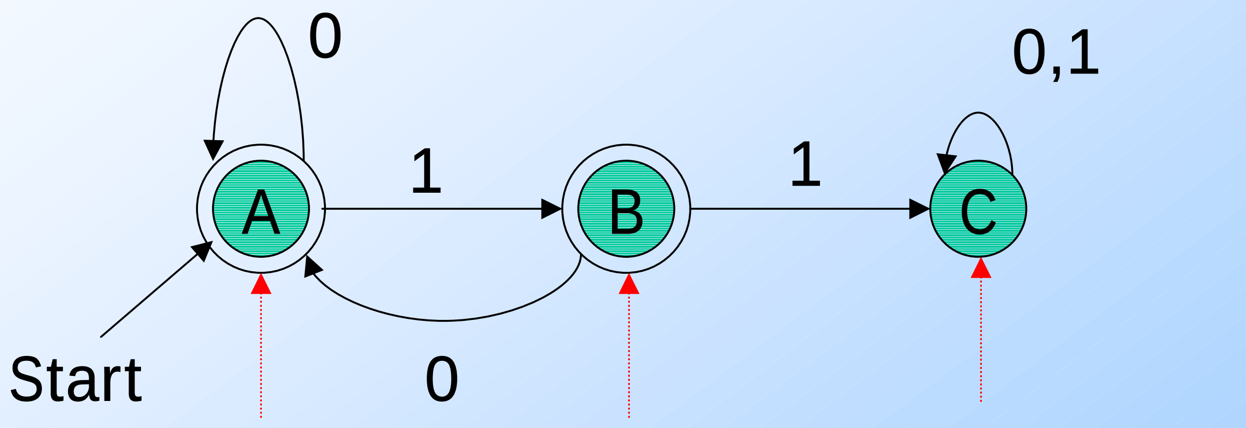
Example: Recognizing Strings Ending in "ing"



Example: Protocol for Sending Data



Example: Strings With No 11



String so far has no 11, does not end in 1.

String so far has no 11, but ends in a single 1.

Consecutive 1's have been seen.

Alternative Representation: Transition Table

Final states starred

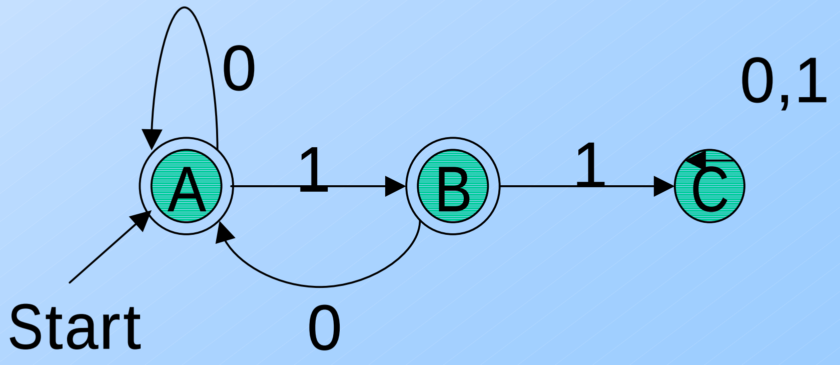
Arrow for start state

	0	1
→ * A	A	B
* B	A	C
C	C	C

Columns = input symbols

Rows = states

Each entry is δ of the row and column.



Convention: Strings and Symbols

- ... w, x, y, z are strings.
- a, b, c,... are single input symbols.

Extended Transition Function

- We describe the effect of a string of inputs on a DFA by extending δ to a state and a string.
- Intuition: Extended δ is computed for state q and inputs $a_1a_2...a_n$ by following a path in the transition graph, starting at q and selecting the arcs with labels a_1, a_2,..., a_n in turn.

Inductive Definition of Extended δ

- Induction on length of string.
- Basis: $\delta(q, \epsilon) = q$
- Induction: $\delta(q,wa) = \delta(\delta(q,w),a)$
  - Remember: w is a string; a is an input symbol, by convention.

Example: Extended Delta

	0	1
A	A	B
B	A	C
C	C	C

$\delta(B,011) = \delta(\delta(B,01),1) = \delta(\delta(\delta(B,0),1),1) =$

$\delta(\delta(A,1),1) = \delta(B,1) = C$

Delta-hat

- We don't distinguish between the given delta and the extended delta or delta-hat.
- The reason:
- $\underbrace{\hat{\delta}}_{\text{Extended deltas}}(q, a) = \delta(\underbrace{\hat{\delta}}_{\text{Extended deltas}}(q, \epsilon), a) = \delta(q, a)$

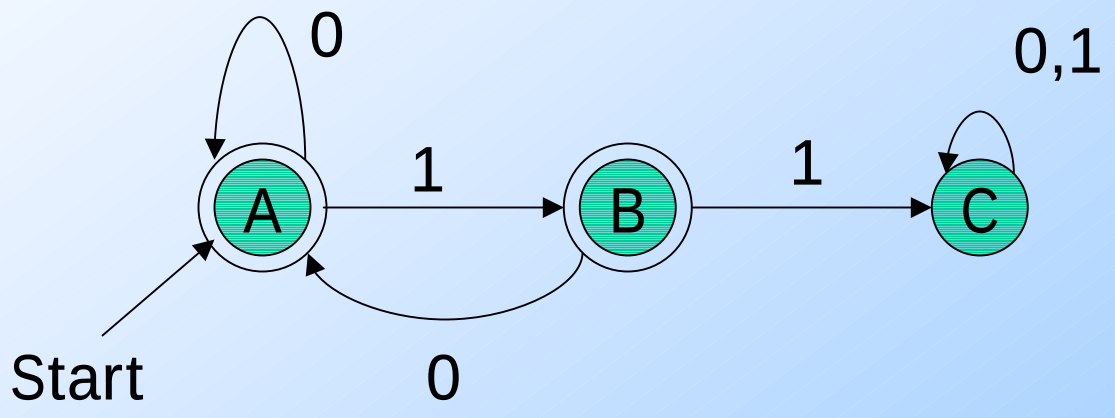
Extended deltas

Language of a DFA

- Automata of all kinds define languages.
- If A is an automaton, L(A) is its language.
- For a DFA A, L(A) is the set of strings labeling paths from the start state to a final state.
- Formally: L(A) = the set of strings w such that $\delta(q_0, w)$ is in F.

Example: String in a Language

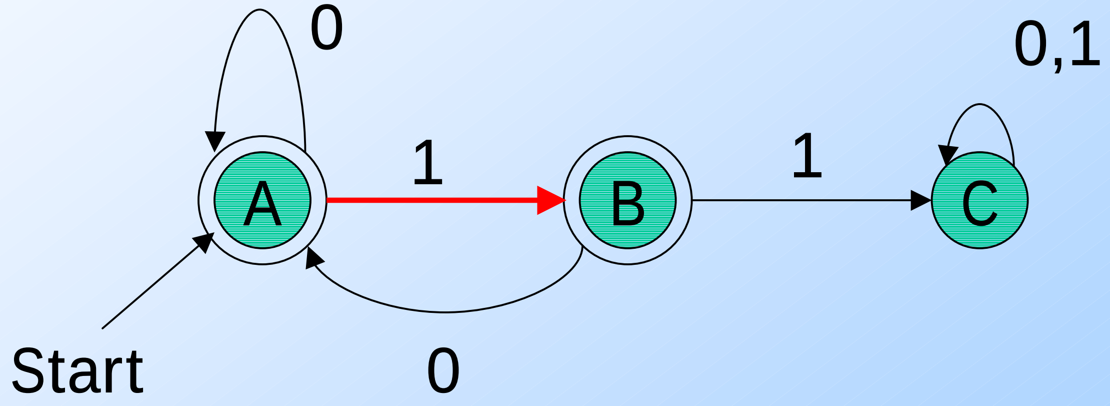
String 101 is in the language of the DFA below. Start at A.



Example: String in a Language

String 101 is in the language of the DFA below.

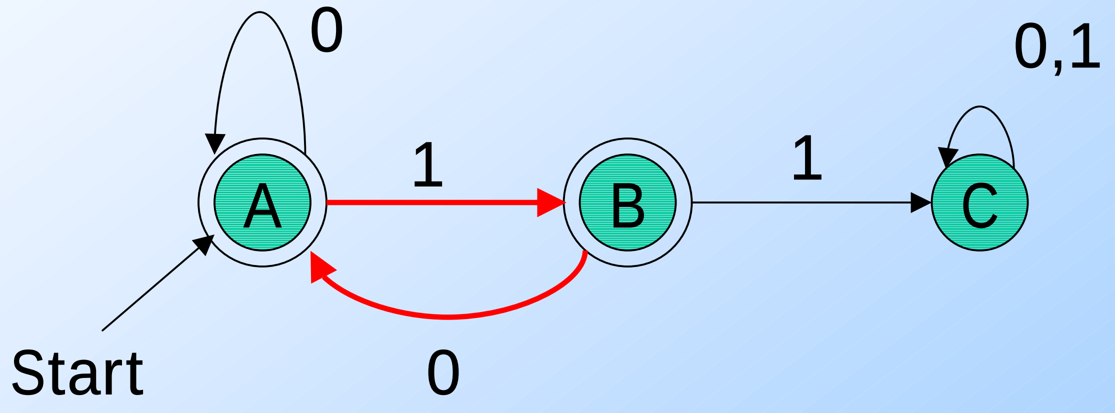
Follow arc labeled 1.



Example: String in a Language

String 101 is in the language of the DFA below.

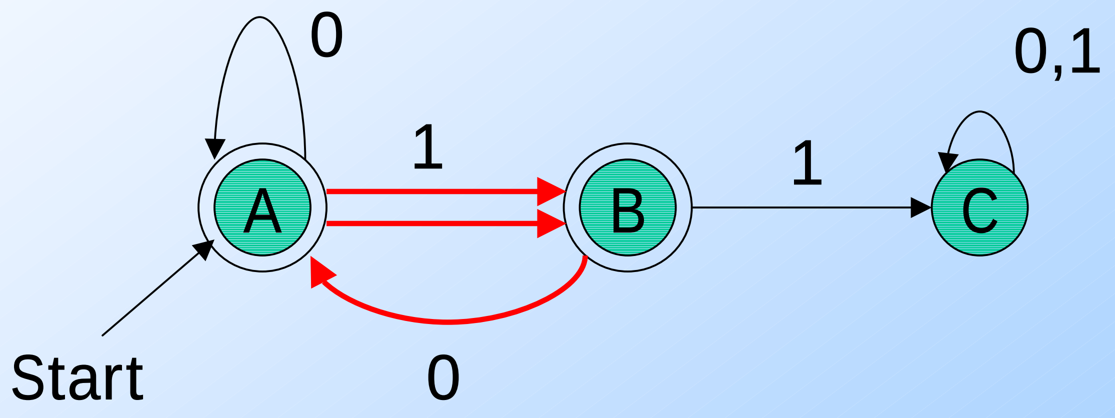
Then arc labeled 0 from current state B.



Example: String in a Language

String 101 is in the language of the DFA below.

Finally arc labeled 1 from current state A. Result is an accepting state, so 101 is in the language.



Example – Concluded

- The language of our example DFA is:

{w | w is in {0,1}* and w does not have two consecutive 1's}

Such that...

These conditions about w are true.

Read a *set former* as "The set of strings w...

Proofs of Set Equivalence

- Often, we need to prove that two descriptions of sets are in fact the same set.
- Here, one set is “the language of this DFA,” and the other is “the set of strings of 0’s and 1’s with no consecutive 1’s.”

Proofs – (2)

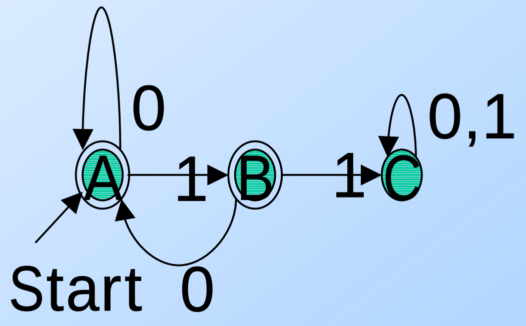
- In general, to prove S = T, we need to prove two parts: S ⊆ T and T ⊆ S.

  That is:

  1. If w is in S, then w is in T.
  2. If w is in T, then w is in S.

- Here, S = the language of our running DFA, and T = “no consecutive 1’s.”

Part 1: S ⊆ T



- To prove: if w is accepted by then w has no consecutive 1's.
- Proof is an induction on length of w.
- Important trick: Expand the inductive hypothesis to be more detailed than the statement you are trying to prove.

The Inductive Hypothesis

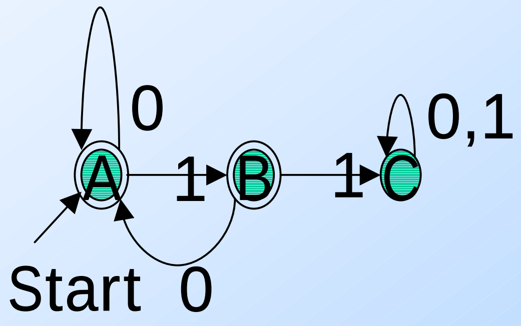
1. If δ(A, w) = A, then w has no consecutive 1's and does not end in 1.
2. If δ(A, w) = B, then w has no consecutive 1's and ends in a single 1.

- Basis: |w| = 0; i.e., w = ϵ.
  - (1) holds since ϵ has no 1's at all.
  - (2) holds *vacuously*, since δ(A, ϵ) is not B.

"length of"

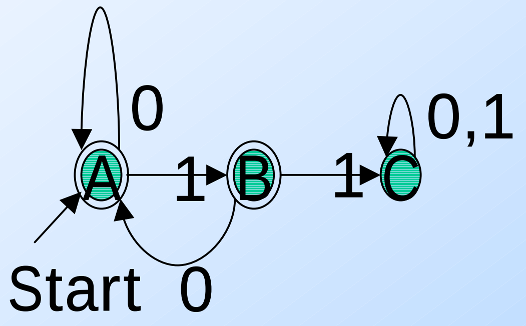
Important concept:
If the "if" part of "if..then" is false, the statement is true.

Inductive Step



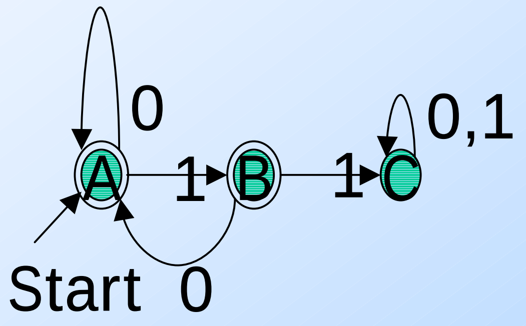
- Assume (1) and (2) are true for strings shorter than w, where |w| is at least 1.
- Because w is not empty, we can write w = xa, where *a* is the last symbol of w, and x is the string that precedes.
- IH is true for x.

Inductive Step – (2)



- Need to prove (1) and (2) for w = xa.
- (1) for w is: If δ(A, w) = A, then w has no consecutive 1's and does not end in 1.
- Since δ(A, w) = A, δ(A, x) must be A or B, and *a* must be 0 (look at the DFA).
- By the IH, x has no 11's.
- Thus, w has no 11's and does not end in 1.

Inductive Step – (3)

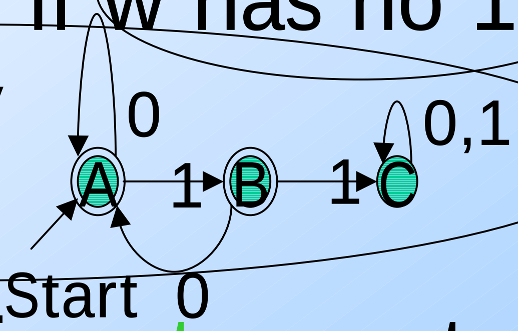


- Now, prove (2) for w = xa: If δ(A, w) = B, then w has no 11's and ends in 1.
- Since δ(A, w) = B, δ(A, x) must be A, and *a* must be 1 (look at the DFA).
- By the IH, x has no 11's and does not end in 1.
- Thus, w has no 11's and ends in 1.

Part 2: T ⊆ S

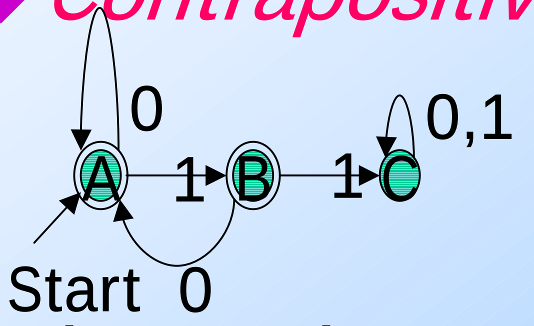
X

- Now, we must prove: if w has no 11's, then w is accepted by



Y

- *Contrapositive* : If w is not accepted by



then w has 11.

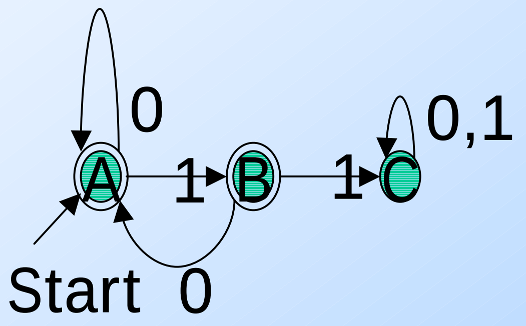
Key idea: contrapositive of "if X then Y" is the equivalent statement "if not Y then not X."

Using the Contrapositive

$$\begin{tikzcd}
\text{Start} \arrow[r] & A \arrow[loop above, "0"] \arrow[r, "1"] & B \arrow[l, bend left, "0"] \arrow[r, "1"] & C \arrow[loop above, "0,1"]
\end{tikzcd}$$

- Because there is a unique transition from every state on every input symbol, each w gets the DFA to exactly one state.
- The only way w is not accepted is if it gets to C.

Using the Contrapositive – (2)



- The only way to get to C [formally: δ(A,w) = C] is if w = x1y, x gets to B, and y is the tail of w that follows what gets to C for the first time.
- If δ(A,x) = B then surely x = z1 for some z.
- Thus, w = z11y and has 11.

Regular Languages

- A language L is *regular* if it is the language accepted by some DFA.
  - Note: the DFA must accept only the strings in L, no others.
- Some languages are not regular.
  - Intuitively, regular languages "cannot count" to arbitrarily high integers.

Example: A Nonregular Language

$L_1 = \{0^n1^n \mid n \geq 1\}$

- Note: a^i is conventional for i a's.
  - Thus, $0^4 = 0000$, e.g.
- Read: "The set of strings consisting of n 0's followed by n 1's, such that n is at least 1.
- Thus, $L_1 = \{01, 0011, 000111, \ldots\}$

Another Example

L_2 = {w | w in {(,)}* and w is *balanced* }

- Balanced parentheses are those sequences of parentheses that can appear in an arithmetic expression.
- E.g.: (), ()(), (()), (()()),...

But Many Languages are Regular

- They appear in many contexts and have many useful properties.
- Example: the strings that represent floating point numbers in your favorite language is a regular language.

Example: A Regular Language

L_3 = { w | w in {0,1}* and w, viewed as a binary integer is divisible by 23}

- The DFA:
  - 23 states, named 0, 1,...,22.
  - Correspond to the 23 remainders of an integer divided by 23.
  - Start and only final state is 0.

Transitions of the DFA for L_3

- If string w represents integer i, then assume $\delta(0, w) = i\%23$.
- Then w0 represents integer 2i, so we want $\delta(i\%23, 0) = (2i)\%23$.
- Similarly: w1 represents 2i+1, so we want $\delta(i\%23, 1) = (2i+1)\%23$.
- Example: $\delta(15,0) = 30\%23 = 7$; $\delta(11,1) = 23\%23 = 0$.

Another Example

L_4 = { w | w in {0,1}* and w, viewed as the reverse of a binary integer is divisible by 23}

- Example: 01110100 is in L_4, because its reverse, 00101110 is 46 in binary.
- Hard to construct the DFA.
- But there is a theorem that says the reverse of a regular language is also regular.

Nondeterministic Finite Automata

Nondeterminism

Subset Construction

ϵ-Transitions

Nondeterminism

- A *nondeterministic finite automaton* has the ability to be in several states at once.
- Transitions from a state on an input symbol can be to any set of states.

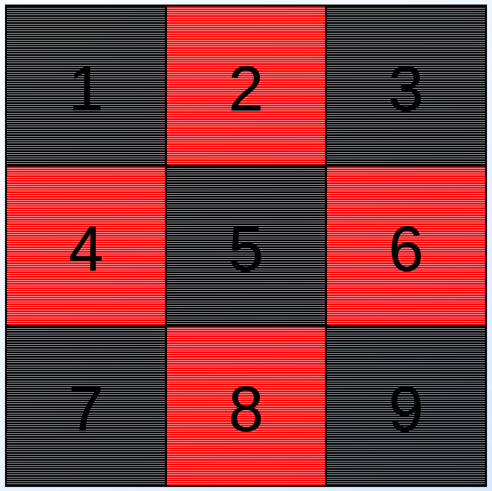
Nondeterminism – (2)

- Start in one start state.
- Accept if any sequence of choices leads to a final state.
- Intuitively: the NFA always "guesses right."

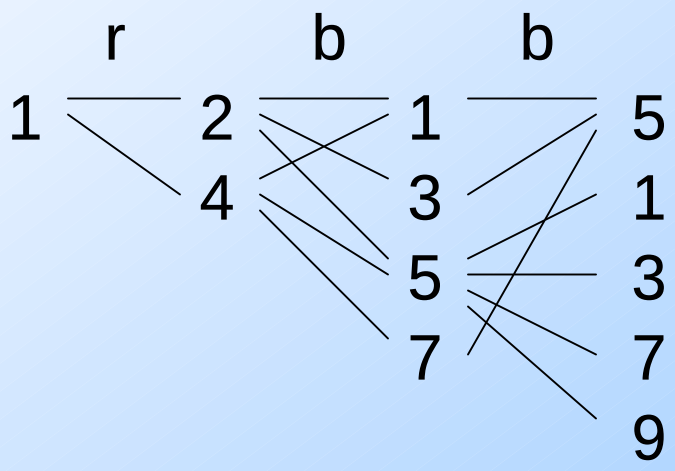
Example: Moves on a Chessboard

- States = squares.
- Inputs = r (move to an adjacent red square) and b (move to an adjacent black square).
- Start state, final state are in opposite corners.

Example: Chessboard – (2)



		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5



← Accept, since final state reached

Formal NFA

- A finite set of states, typically Q.
- An input alphabet, typically Σ.
- A transition function, typically δ.
- A start state in Q, typically q_0.
- A set of final states $F \subseteq Q$.

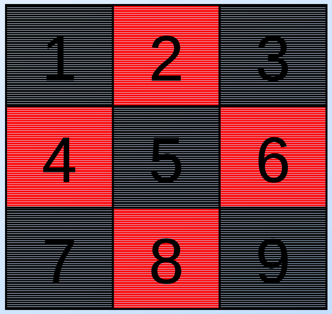
Transition Function of an NFA

- δ(q, a) is a set of states.
- Extend to strings as follows:
- Basis: δ(q, ϵ) = {q}
- Induction: δ(q, wa) = the union over all states p in δ(q, w) of δ(p, a)

Language of an NFA

- A string w is accepted by an NFA if $\delta(q_0, w)$ contains at least one final state.
- The language of the NFA is the set of strings it accepts.

Example: Language of an NFA



- For our chessboard NFA we saw that rbb is accepted.
- If the input consists of only b's, the set of accessible states alternates between {5} and {1,3,7,9}, so only even-length, nonempty strings of b's are accepted.
- What about strings with at least one r?

Equivalence of DFA's, NFA's

- A DFA can be turned into an NFA that accepts the same language.
- If $\delta_D(q, a) = p$, let the NFA have $\delta_N(q, a) = \{p\}$.
- Then the NFA is always in a set containing exactly one state – the state the DFA is in after reading the same input.

Equivalence – (2)

- Surprisingly, for any NFA there is a DFA that accepts the same language.
- Proof is the *subset construction*.
- The number of states of the DFA can be exponential in the number of states of the NFA.
- Thus, NFA's accept exactly the regular languages.

Subset Construction

- Given an NFA with states Q, inputs Σ, transition function δ_N, state state q_0, and final states F, construct equivalent DFA with:
  - States 2^Q (Set of subsets of Q).
  - Inputs Σ.
  - Start state $\{q_0\}$.
  - Final states = all those with a member of F.

Critical Point

- The DFA states have *names* that are sets of NFA states.
- But as a DFA state, an expression like {p,q} must be understood to be a single symbol, not as a set.
- Analogy: a class of objects whose values are sets of objects of another class.

Subset Construction – (2)

- The transition function δ_D is defined by:

$\delta_D(\{q_1,\dots,q_k\}, a)$ is the union over all $i = 1,\dots,k$ of $\delta_N(q_i, a)$.

- Example: We'll construct the DFA equivalent of our "chessboard" NFA.

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}		
{5}		

Alert: What we're doing here is the *lazy* form of DFA construction, where we only construct a state if we are forced to.

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}		
	{2,4,6,8}		
	{1,3,5,7}		

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}		
	{1,3,5,7}		
*	{1,3,7,9}		

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}		
*	{1,3,7,9}		
*	{1,3,5,7,9}		

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}		
*	{1,3,5,7,9}		

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}	{2,4,6,8}	{5}
*	{1,3,5,7,9}		

Example: Subset Construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}	{2,4,6,8}	{5}
*	{1,3,5,7,9}	{2,4,6,8}	{1,3,5,7,9}

Proof of Equivalence: Subset Construction

- The proof is almost a pun.
- Show by induction on $|w|$ that

  $\delta_N(q_0, w) = \delta_D(\{q_0\}, w)$
- Basis: $w = \epsilon$: $\delta_N(q_0, \epsilon) = \delta_D(\{q_0\}, \epsilon) = \{q_0\}$.

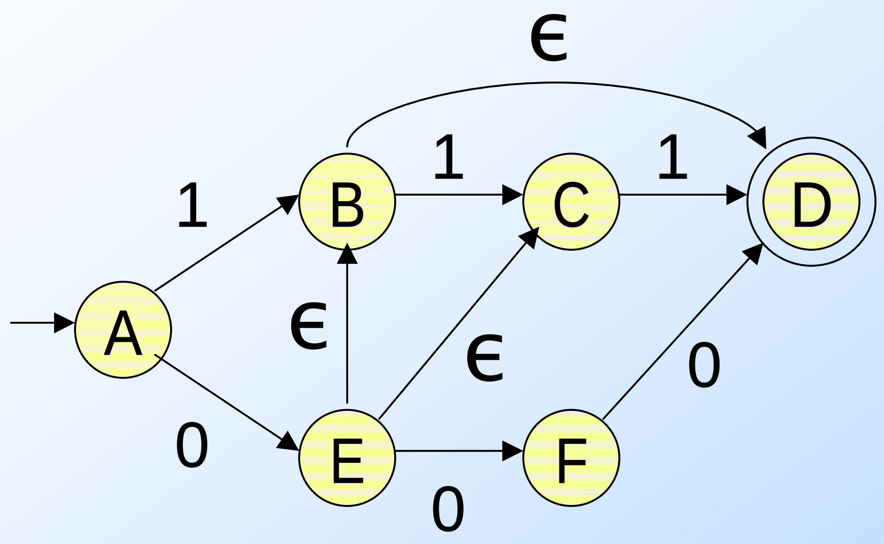
Induction

- Assume IH for strings shorter than w.
- Let w = xa; IH holds for x.
- Let $\delta_N(q_0, x) = \delta_D(\{q_0\}, x) = S$.
- Let T = the union over all states p in S of $\delta_N(p, a)$.
- Then $\delta_N(q_0, w) = \delta_D(\{q_0\}, w) = T$.

NFA's With ϵ-Transitions

- We can allow state-to-state transitions on ϵ input.
- These transitions are done spontaneously, without looking at the input string.
- A convenience at times, but still only regular languages are accepted.

Example: ϵ-NFA



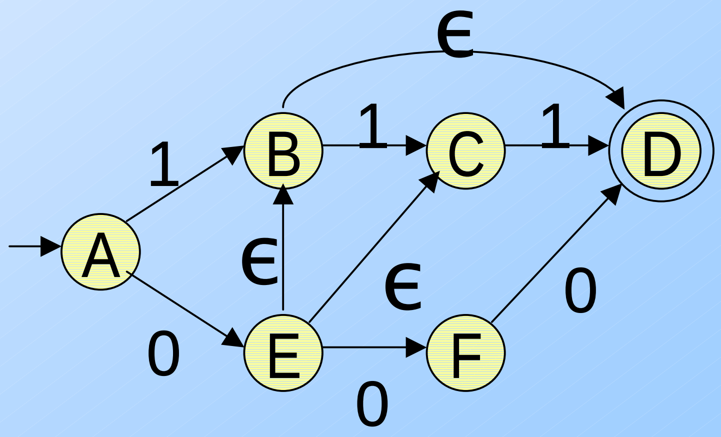
$$\begin{tikzcd}
 & B \arrow[r, "1"] \arrow[rr, bend left, "\epsilon"] & C \arrow[r, "1"] & D \\
\arrow[r] & A \arrow[u, "1"] \arrow[d, "0"'] & & \\
 & E \arrow[uu, "\epsilon"] \arrow[uur, "\epsilon"] \arrow[r, "0"'] & F \arrow[uur, "0"'] &
\end{tikzcd}$$

		0	1	ϵ
→	A	{E}	{B}	∅
	B	∅	{C}	{D}
	C	∅	{D}	∅
*	D	∅	∅	∅
	E	{F}	∅	{B, C}
	F	{D}	∅	∅

Closure of States

- CL(q) = set of states you can reach from state q following only arcs labeled ϵ.
- Example: CL(A) = {A};

  CL(E) = {B, C, D, E}.

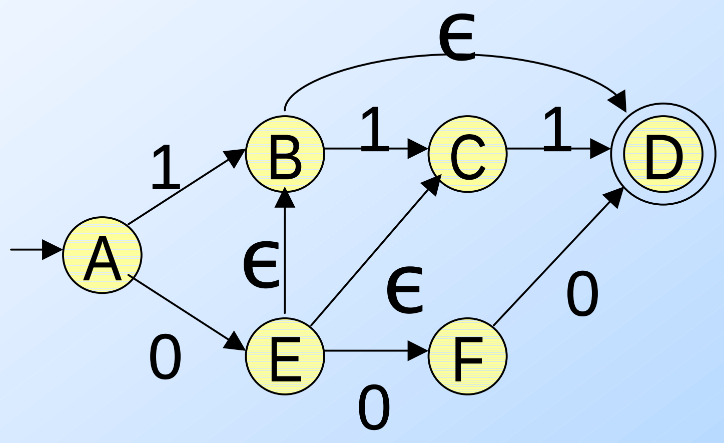


- Closure of a set of states = union of the closure of each state.

Extended Delta

- Intuition: $\hat{\delta}(q, w)$ is the set of states you can reach from q following a path labeled w.
- Basis: $\hat{\delta}(q, \epsilon) = CL(q)$.
- Induction: $\hat{\delta}(q, xa)$ is computed by:
  1. Start with $\hat{\delta}(q, x) = S$.
  2. Take the union of $CL(\delta(p, a))$ for all p in S.

Example: Extended Delta

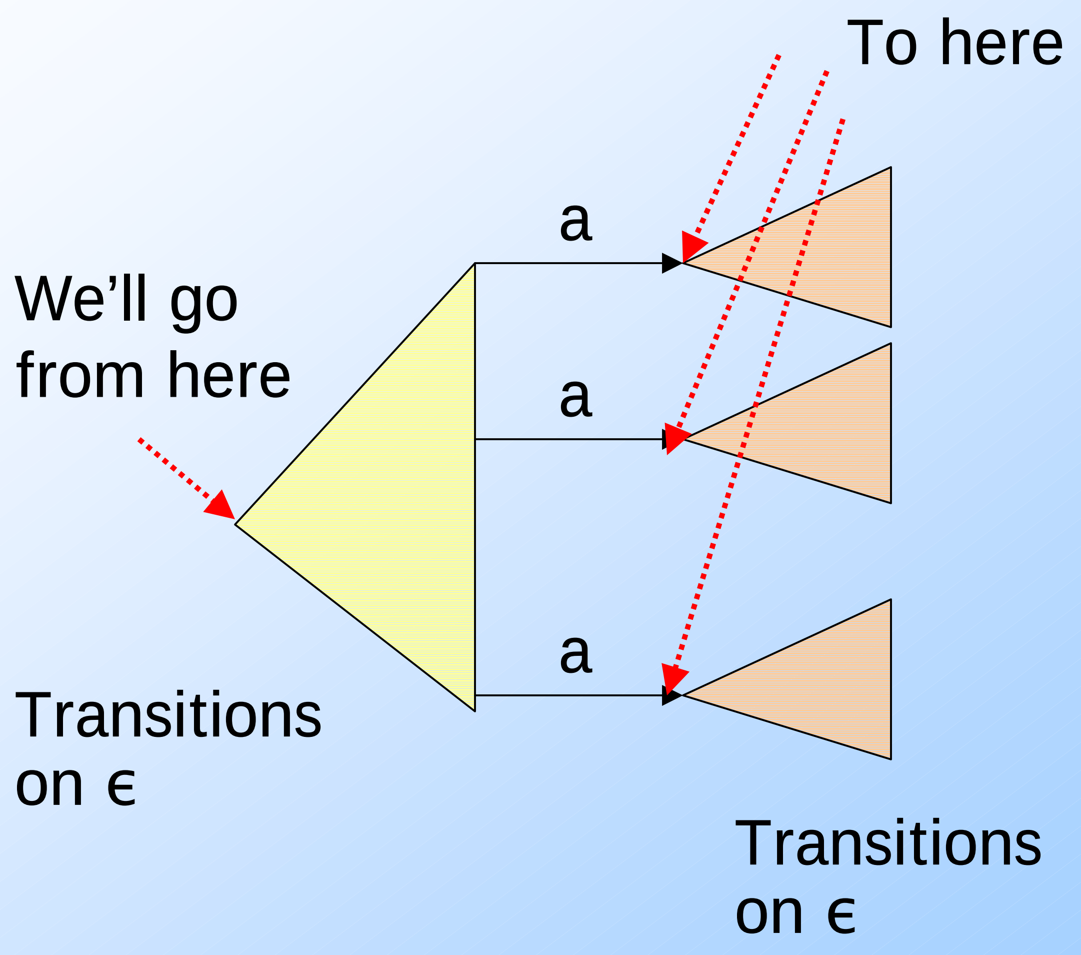


- $\hat{\delta}(A, \epsilon) = CL(A) = \{A\}$.
- $\hat{\delta}(A, 0) = CL(\{E\}) = \{B, C, D, E\}$.
- $\hat{\delta}(A, 01) = CL(\{C, D\}) = \{C, D\}$.
- *Language* of an ϵ-NFA is the set of strings w such that $\hat{\delta}(q_0, w)$ contains a final state.

Equivalence of NFA, ϵ-NFA

- Every NFA is an ϵ-NFA.
  - It just has no transitions on ϵ.
- Converse requires us to take an ϵ-NFA and construct an NFA that accepts the same language.
- We do so by combining ϵ–transitions with the next transition on a real input.

Picture of ϵ-Transition Removal



Equivalence – (2)

- Start with an ϵ-NFA with states Q, inputs Σ, start state q_0, final states F, and transition function δ_E.
- Construct an "ordinary" NFA with states Q, inputs Σ, start state q_0, final states F', and transition function δ_N.

Equivalence – (3)

- Compute $\delta_N(q, a)$ as follows:
  1. Let S = CL(q).
  2. $\delta_N(q, a)$ is the union over all p in S of $\delta_E(p, a)$.
- F' = the set of states q such that CL(q) contains a state of F.

Equivalence – (4)

- Prove by induction on |w| that

$$CL(\delta_N(q_0, w)) = \hat{\delta}_E(q_0, w).$$

- Thus, the ϵ-NFA accepts w if and only if the "ordinary" NFA does.

Example: ϵ-NFA-to-NFA

Interesting closures: CL(B) = {B,D}; CL(E) = {B,C,D,E}

		0	1	ϵ
→	A	{E}	{B}	∅
	B	∅	{C}	{D}
	C	∅	{D}	∅
*	D	∅	∅	∅
	E	{F}	∅	{B, C}
	F	{D}	∅	∅

ϵ-NFA

		0	1
→	A	{E}	{B}
*	B	∅	{C}
	C	∅	{D}
*	D	∅	∅
*	E	{F}	{C, D}
	F	{D}	∅

Doesn't change, since B, C, D have no trans-itions on 0.

Since closures of B and E include final state D.

Since closure of E includes B and C; which have transitions on 1 to C and D.

Summary

- DFA's, NFA's, and ϵ–NFA's all accept exactly the same set of languages: the regular languages.
- The NFA types are easier to design and may have exponentially fewer states than a DFA.
- But only a DFA can be implemented!

Regular Expressions

Definitions

Equivalence to Finite Automata

RE's: Introduction

- *Regular expressions* describe languages by an algebra.
- They describe exactly the regular languages.
- If E is a regular expression, then L(E) is the language it defines.
- We'll describe RE's and their languages recursively.

Operations on Languages

- RE's use three operations: union, concatenation, and Kleene star.
- The union of languages is the usual thing, since languages are sets.
- Example: $\{01,111,10\} \cup \{00, 01\} = \{01,111,10,00\}$.

Concatenation

- The *concatenation* of languages L and M is denoted LM.
- It contains every string wx such that w is in L and x is in M.
- Example: {01,111,10}{00, 01} = {0100, 0101, 11100, 11101, 1000, 1001}.

Kleene Star

- If L is a language, then L*, the *Kleene star* or just "star," is the set of strings formed by concatenating zero or more strings from L, in any order.
- $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \ldots$
- Example: $\{0,10\}^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, \ldots\}$

RE's: Definition

- Basis 1: If *a* is any symbol, then **a** is a RE, and L(**a**) = {a}.
  - Note: {a} is the language containing one string, and that string is of length 1.
- Basis 2: ϵ is a RE, and L(ϵ) = {ϵ}.
- Basis 3: ∅ is a RE, and L(∅) = ∅.

RE's: Definition – (2)

- Induction 1: If E_1 and E_2 are regular expressions, then E_1+E_2 is a regular expression, and $L(E_1+E_2) = L(E_1) \cup L(E_2)$.
- Induction 2: If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$.
- Induction 3: If E is a RE, then E* is a RE, and $L(E^*) = (L(E))^*$.

Precedence of Operators

- Parentheses may be used wherever needed to influence the grouping of operators.
- Order of precedence is * (highest), then concatenation, then + (lowest).

Examples: RE's

- L(**01**) = {01}.
- L(**01**+**0**) = {01, 0}.
- L(**0**(**1**+**0**)) = {01, 00}.
  - Note order of precedence of operators.
- L(**0***) = {ϵ, 0, 00, 000,... }.
- L((**0**+**10**)*(ϵ+**1**)) = all strings of 0's and 1's without two consecutive 1's.

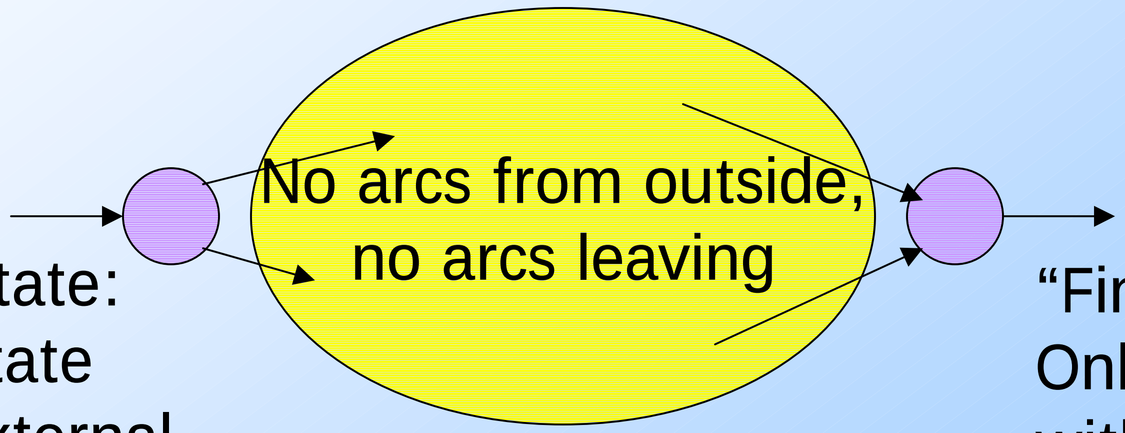
Equivalence of RE's and Finite Automata

- We need to show that for every RE, there is a finite automaton that accepts the same language.
  - Pick the most powerful automaton type: the ϵ-NFA.
- And we need to show that for every finite automaton, there is a RE defining its language.
  - Pick the most restrictive type: the DFA.

Converting a RE to an ϵ-NFA

- Proof is an induction on the number of operators (+, concatenation, *) in the RE.
- We always construct an automaton of a special form (next slide).

Form of ϵ-NFA's Constructed

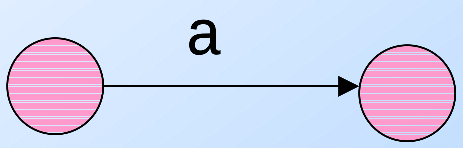


Start state:
Only state
with external
predecessors

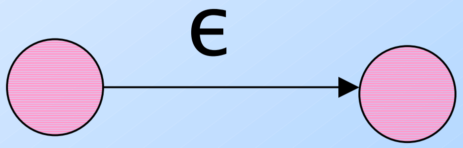
"Final" state:
Only state
with external
successors

RE to ϵ-NFA: Basis

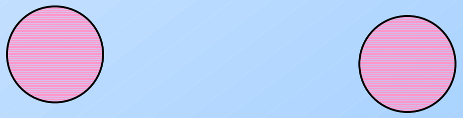
- Symbol **a**:



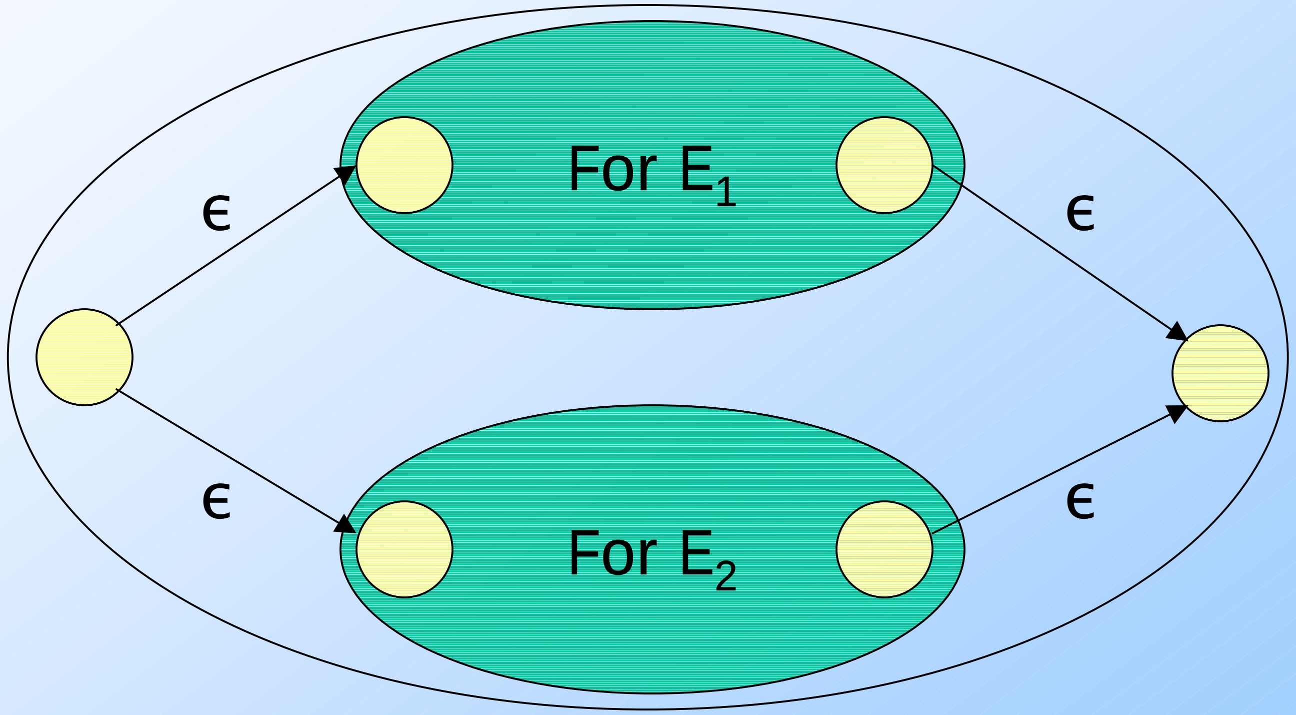
- ϵ:



- ∅:

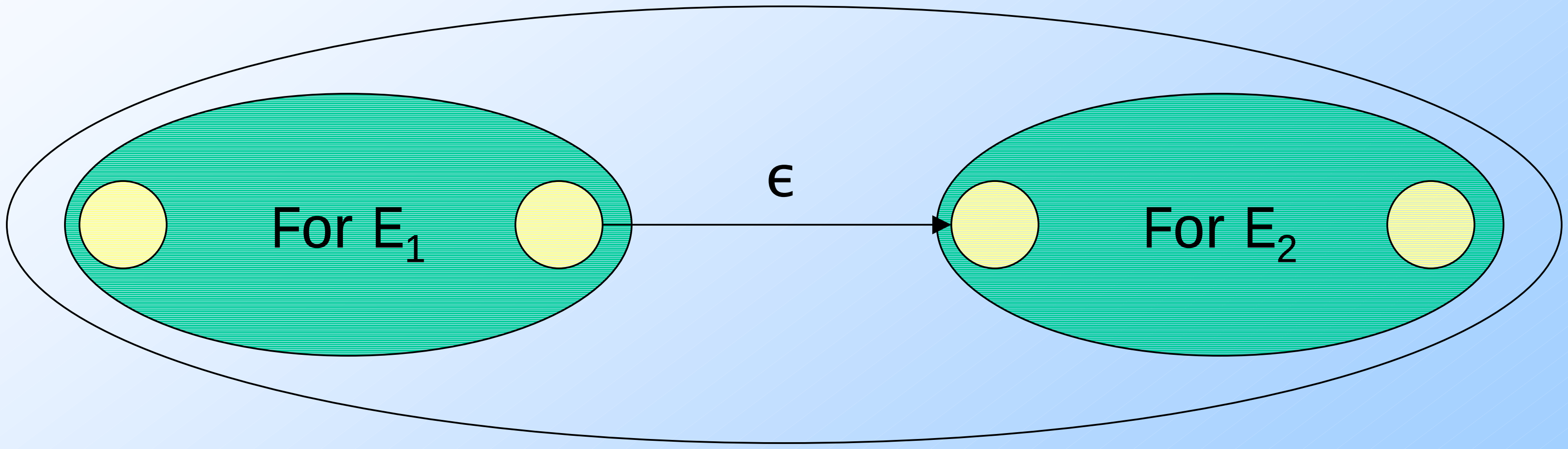


RE to ϵ-NFA: Induction 1 – Union



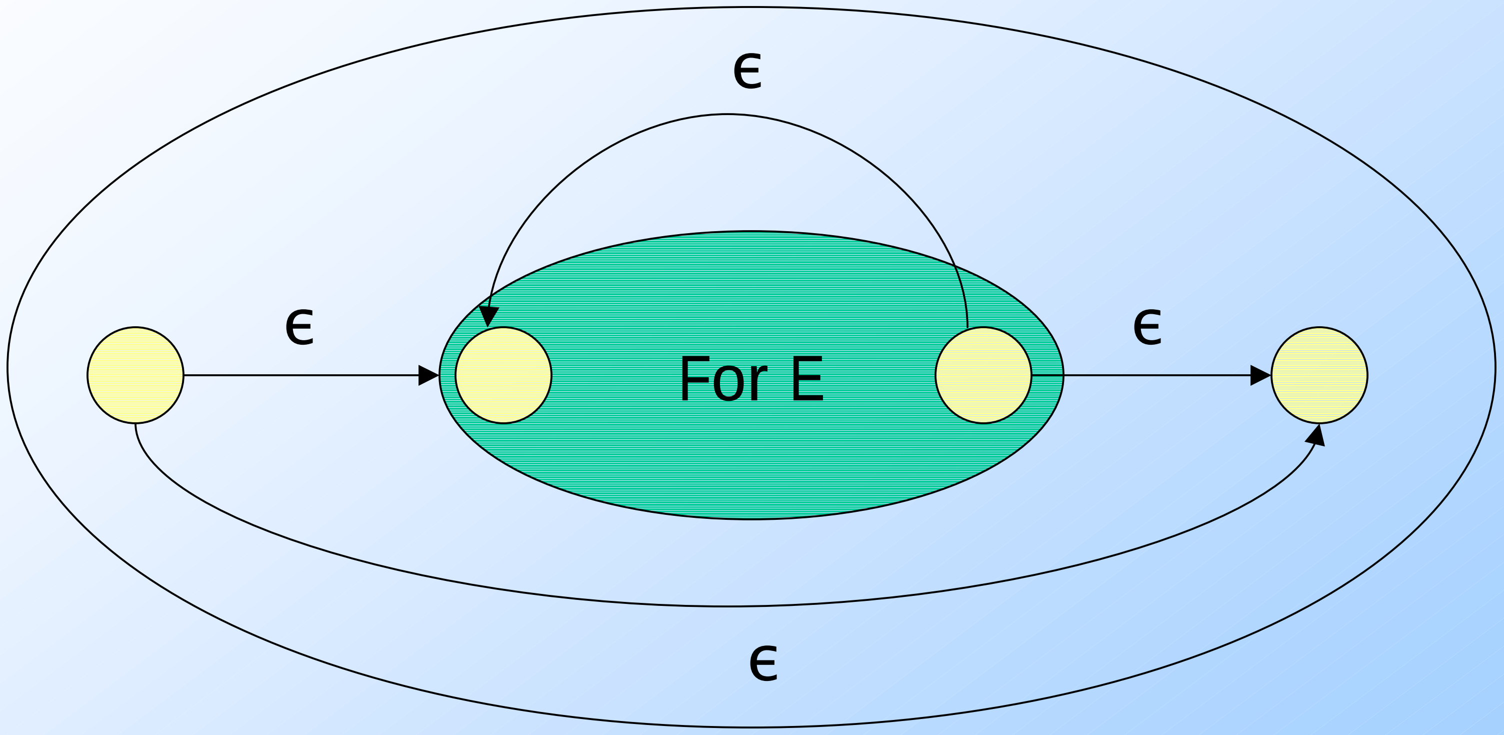
For $E_1 \cup E_2$

RE to ϵ-NFA: Induction 2 – Concatenation



For E_1E_2

RE to ϵ-NFA: Induction 3 – Closure



For E*

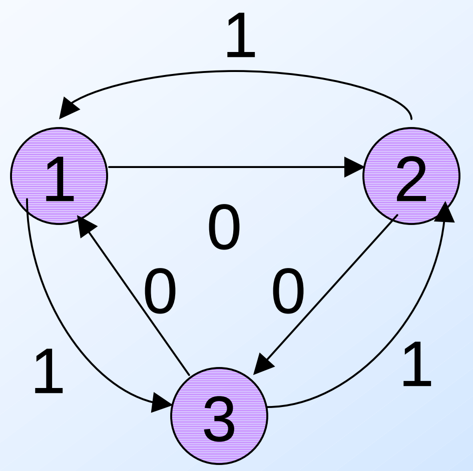
DFA-to-RE

- A strange sort of induction.
- States of the DFA are named 1,2,…,n.
- Induction is on k, the maximum state number we are allowed to traverse along a path.

k-Paths

- A k-path is a path through the graph of the DFA that goes though no state numbered higher than k.
- Endpoints are not restricted; they can be any state.
- n-paths are unrestricted.
- RE is the union of RE's for the n-paths from the start state to each final state.

Example: k-Paths



0-paths from 2 to 3:
RE for labels = **0**.

1-paths from 2 to 3:
RE for labels = **0**+**11**.

2-paths from 2 to 3:
RE for labels =
(**10**)***0**+**1**(**01**)***1**

3-paths from 2 to 3:
RE for labels = ??

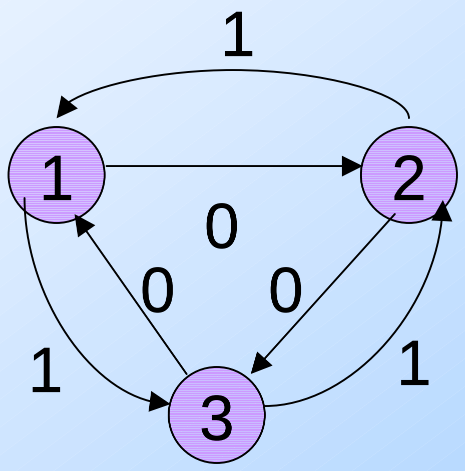
DFA-to-RE

- Basis: k = 0; only arcs or a node by itself.
- Induction: construct RE's for paths allowed to pass through state k from paths allowed only up to k-1.

k-Path Induction

- Let R_{ij}^{k} be the regular expression for the set of labels of k-paths from state i to state j.
- Basis: k=0. R_{ij}^{0} = sum of labels of arc from i to j.
  - $\emptyset$ if no such arc.
  - But add ϵ if i=j.

Example: Basis



- R_{12}^{0} = **0**.
- $R_{11}^{0} = \varnothing + \epsilon = \epsilon$.

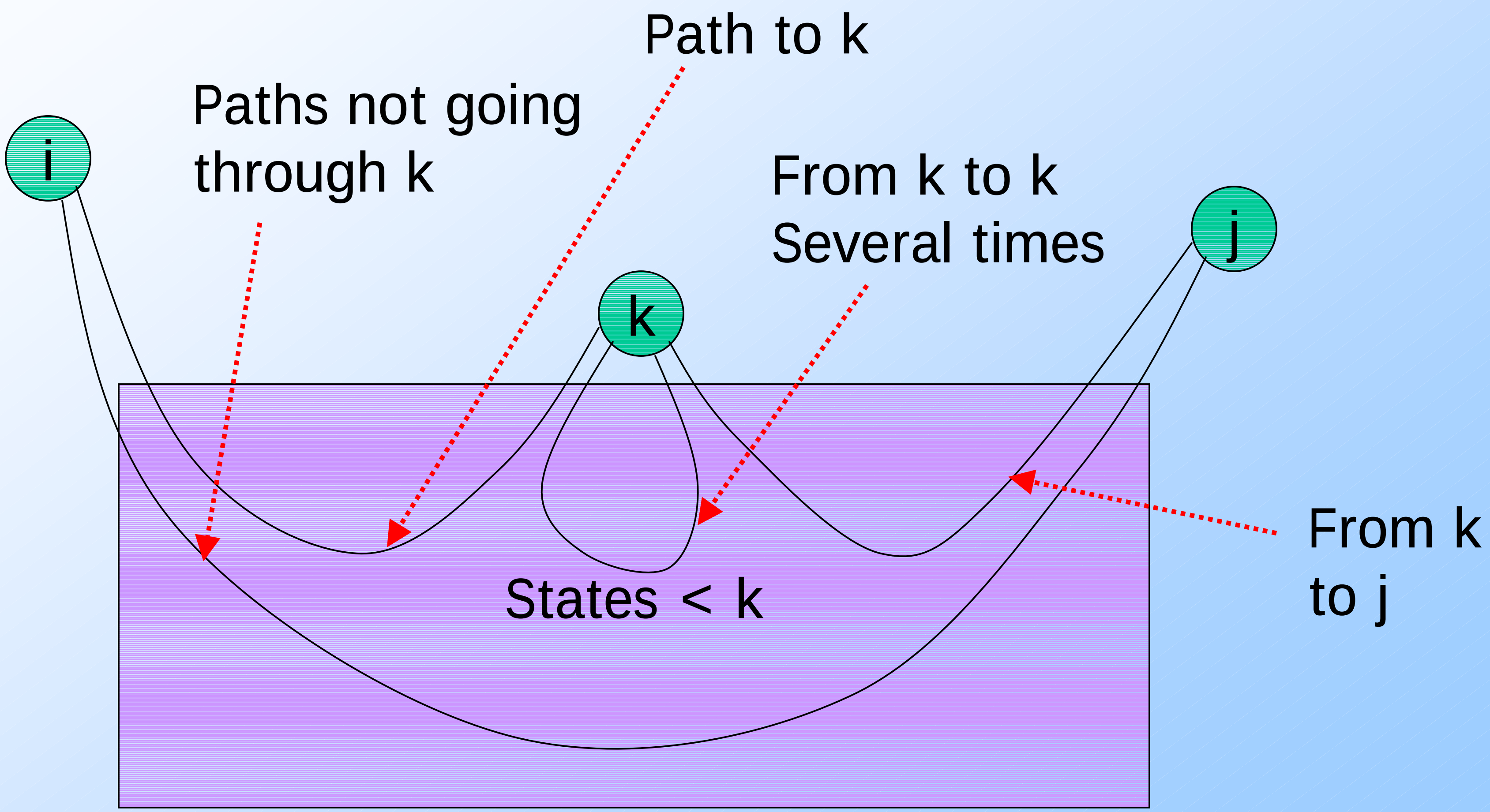
Notice algebraic law:
$\varnothing$ plus anything =
that thing.

k-Path Inductive Case

- A k-path from i to j either:
  1. Never goes through state k, or
  2. Goes through k one or more times.

$$R_{ij}^{k} = \underbrace{R_{ij}^{k-1}}_{\text{Doesn't go through k}} + \underbrace{R_{ik}^{k-1}}_{\text{Goes from i to k the first time}}\underbrace{(R_{kk}^{k-1})^{*}}_{\text{Zero or more times from k to k}}\ \underbrace{R_{kj}^{k-1}}_{\text{Then, from k to j}}.$$

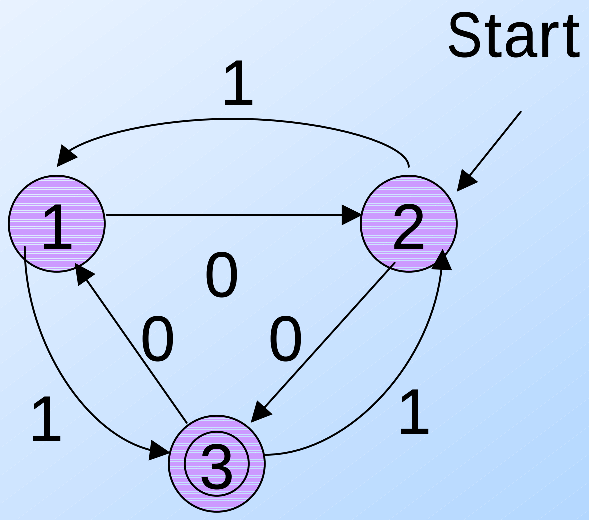
Illustration of Induction



Final Step

- The RE with the same language as the DFA is the sum (union) of R_{ij}^{n}, where:
  1. n is the number of states; i.e., paths are unconstrained.
  2. i is the start state.
  3. j is one of the final states.

Example



- $R_{23}^{3} = R_{23}^{2} + R_{23}^{2}(R_{33}^{2})^{*}R_{33}^{2} = R_{23}^{2}(R_{33}^{2})^{*}$
- R_{23}^{2} = **(10)*0+1(01)*1**
- R_{33}^{2} = ϵ + **0(01)*(1+00)** + **1(10)*(0+11)**
- R_{23}^{3} = **[(10)*0+1(01)*1] [ϵ + (0(01)*(1+00) + 1(10)*(0+11))]***

Summary

- Each of the three types of automata (DFA, NFA, ϵ-NFA) we discussed, and regular expressions as well, define exactly the same set of languages: the regular languages.

Algebraic Laws for RE's

- Union and concatenation behave sort of like addition and multiplication.
  - + is commutative and associative; concatenation is associative.
  - Concatenation distributes over +.
  - Exception: Concatenation is not commutative.

Identities and Annihilators

- $\emptyset$ is the identity for +.
  - $R + \emptyset = R$.
- ϵ is the identity for concatenation.
  - $\epsilon R = R\epsilon = R$.
- $\emptyset$ is the annihilator for concatenation.
  - $\emptyset R = R\emptyset = \emptyset$.

Applications of Regular Expressions

Unix RE's

Text Processing

Lexical Analysis

Some Applications

- RE's appear in many systems, often private software that needs a simple language to describe sequences of events.
- We'll use Junglee as an example, then talk about text processing and lexical analysis.

Junglee

- Started in the mid-90’s by three of my students, Ashish Gupta, Anand Rajaraman, and Venky Harinarayan.
- Goal was to integrate information from Web pages.
- Bought by Amazon when Yahoo! hired them to build a comparison shopper for books.

Integrating Want Ads

- Junglee's first contract was to integrate on-line want ads into a queryable table.
- Each company organized its employment pages differently.
  - Worse: the organization typically changed weekly.

Junglee’s Solution

- They developed a regular-expression language for navigating within a page and among pages.
- Input symbols were:
  - Letters, for forming words like “salary”.
  - HTML tags, for following structure of page.
  - Links, to jump between pages.

Junglee's Solution – (2)

- Engineers could then write RE's to describe how to find key information at a Web site.
  - E.g., position title, salary, requirements,...
- Because they had a little language, they could incorporate new sites quickly, and they could modify their strategy when the site changed.

RE-Based Software Architecture

- Junglee used a common form of architecture:
  - Use RE's plus actions (arbitrary code) as your input language.
  - Compile into a DFA or simulated NFA.
  - Each accepting state is associated with an action, which is executed when that state is entered.

UNIX Regular Expressions

- UNIX, from the beginning, used regular expressions in many places, including the "grep" command.
  - Grep = "Global (search for a) Regular Expression and Print."
- Most UNIX commands use an extended RE notation that still defines only regular languages.

UNIX RE Notation

- $[a_1a_2\ldots a_n]$ is shorthand for $a_1+a_2+\ldots+a_n$.
- *Ranges* indicated by first-dash-last and brackets.
  - Order is ASCII.
  - Examples: [a-z] = "any lower-case letter," [a-zA-Z] = "any letter."
- Dot = "any character."

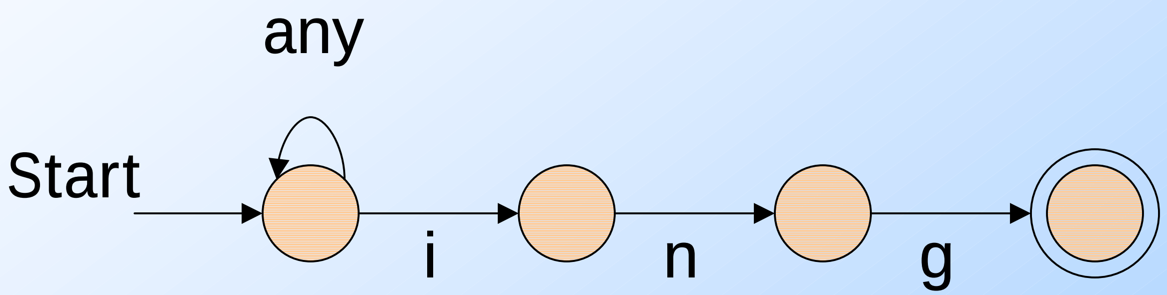
UNIX RE Notation – (2)

- | is used for union instead of +.
- But + has a meaning: "one or more of."
  - E+ = EE*.
  - Example: [a-z]+ = "one or more lower-case letters.
- ? = "zero or one of."
  - E? = E + ϵ.
  - Example: [ab]? = "an optional *a* or *b*."

Example: Text Processing

- Remember our DFA for recognizing strings that end in “ing”?
- It was rather tricky.
- But the RE for such strings is easy: **.*ing** where the dot is the UNIX “any”.
- Even an NFA is easy (next slide).

NFA for “Ends in *ing* ”



Lexical Analysis

- The first thing a compiler does is break a program into *tokens* = substrings that together represent a unit.
  - Examples: identifiers, reserved words like “if,” meaningful single characters like “;” or “+”, multicharacter operators like “<=”.

Lexical Analysis – (2)

- Using a tool like Lex or Flex, one can write a regular expression for each different kind of token.
- Example: in UNIX notation, identifiers are something like [A-Za-z][A-Za-z0-9]*.
- Each RE has an associated action.
  - Example: return a code for the token found.

Tricks for Combining Tokens

- There are some ambiguities that need to be resolved as we convert RE's to a DFA.
- Examples:
  1. "if" looks like an identifier, but it is a reserved word.
  2. < might be a comparison operator, but if followed by =, then the token is <=.

Tricks – (2)

- Convert the RE for each token to an ϵ–NFA.
  - Each has its own final state.
- Combine these all by introducing a new start state with ϵ-transitions to the start states of each ϵ–NFA.
- Then convert to a DFA.

Tricks – (3)

- If a DFA state has several final states among its members, give them priority.
- Example: Give all reserved words priority over identifiers, so if the DFA arrives at a state that contains final states for the “if” ϵ–NFA as well as for the identifier ϵ–NFA, if declares “if”, not identifier.

Tricks – (4)

- It’s a bit more complicated, because the DFA has to have an additional power.
- It must be able to read an input symbol and then, when it accepts, put that symbol back on the input to be read later.

Example: Put-Back

- Suppose “<” is the first input symbol.
- Read the next input symbol.
  - If it is “=”, accept and declare the token is <=.
  - If it is anything else, put it back and declare the token is <.

Example: Put-Back – (2)

- Suppose “if” has been read from the input.
- Read the next input symbol.
  - If it is a letter or digit, continue processing.
    - You did not have reserved word “if”; you are working on an identifier.
  - Otherwise, put it back and declare the token is “if”.

Decision Properties of Regular Languages

General Discussion of “Properties”

The Pumping Lemma

Membership, Emptiness, Etc.

Properties of Language Classes

- A *language class* is a set of languages.
  - Example: the regular languages.
- Language classes have two important kinds of properties:
  1. Decision properties.
  2. Closure properties.

Closure Properties

- A *closure property* of a language class says that given languages in the class, an operation (e.g., union) produces another language in the same class.
- Example: the regular languages are obviously closed under union, concatenation, and (Kleene) closure.
  - Use the RE representation of languages.

Representation of Languages

- Representations can be formal or informal.
- Example (formal): represent a language by a RE or FA defining it.
- Example: (informal): a logical or prose statement about its strings:
  - $\{0^n1^n \mid n \text{ is a nonnegative integer}\}$
  - "The set of strings consisting of some number of 0's followed by the same number of 1's."

Decision Properties

- A *decision property* for a class of languages is an algorithm that takes a formal description of a language (e.g., a DFA) and tells whether or not some property holds.
- Example: Is language L empty?

Why Decision Properties?

- Think about DFA's representing protocols.
- Example: "Does the protocol terminate?" = "Is the language finite?"
- Example: "Can the protocol fail?" = "Is the language nonempty?"
  - Make the final state be the "error" state.

Why Decision Properties – (2)

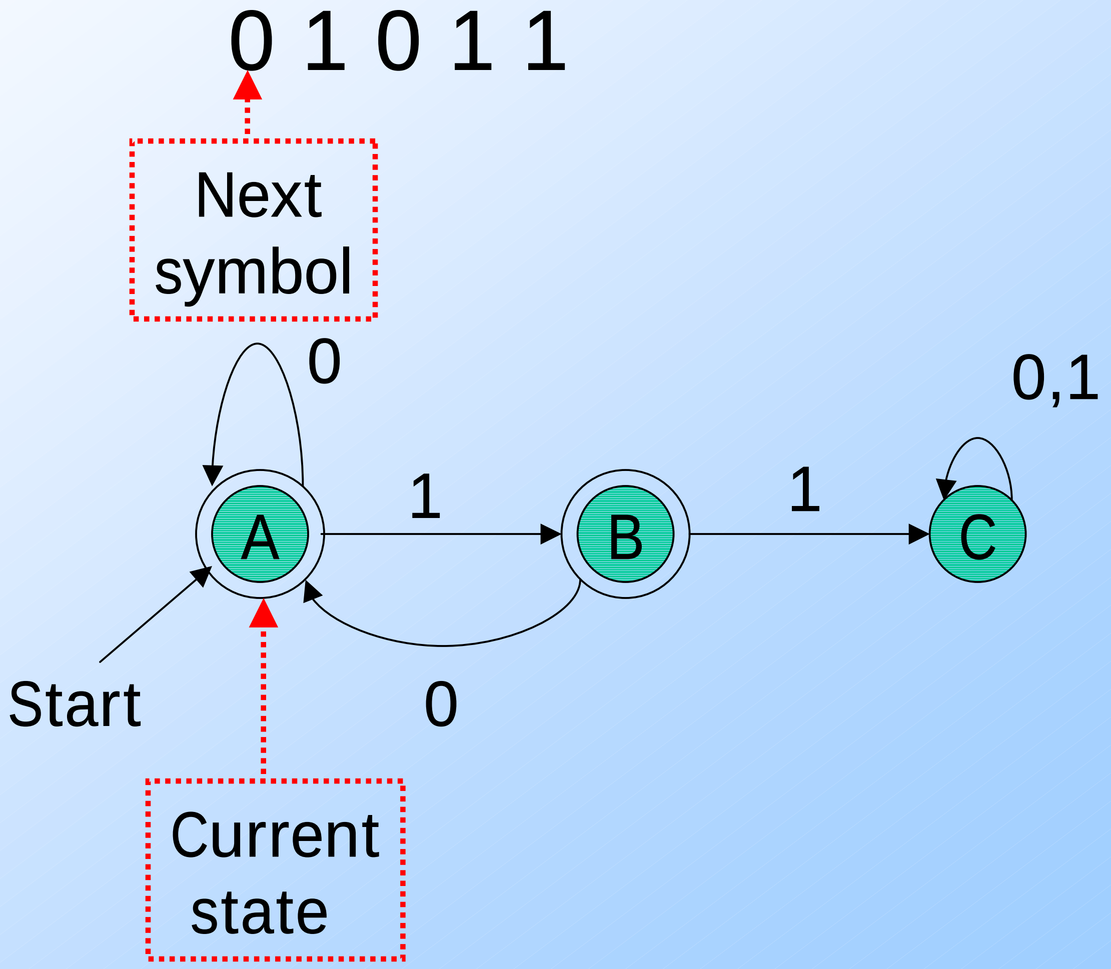
- We might want a "smallest" representation for a language, e.g., a minimum-state DFA or a shortest RE.
- If you can't decide "Are these two languages the same?"
  - I.e., do two DFA's define the same language?

You can't find a "smallest."

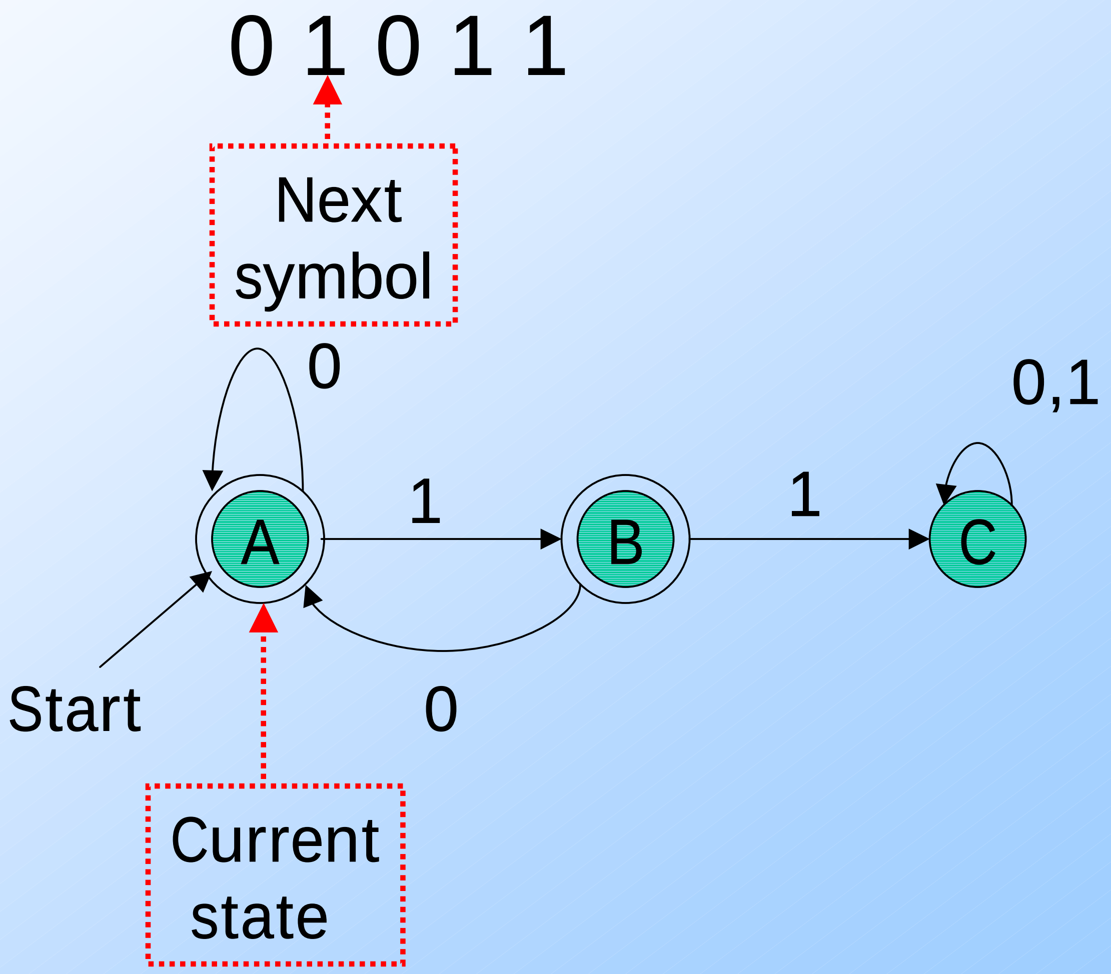
The Membership Problem

- Our first decision property for regular languages is the question: "is string w in regular language L?"
- Assume L is represented by a DFA A.
- Simulate the action of A on the sequence of input symbols forming w.

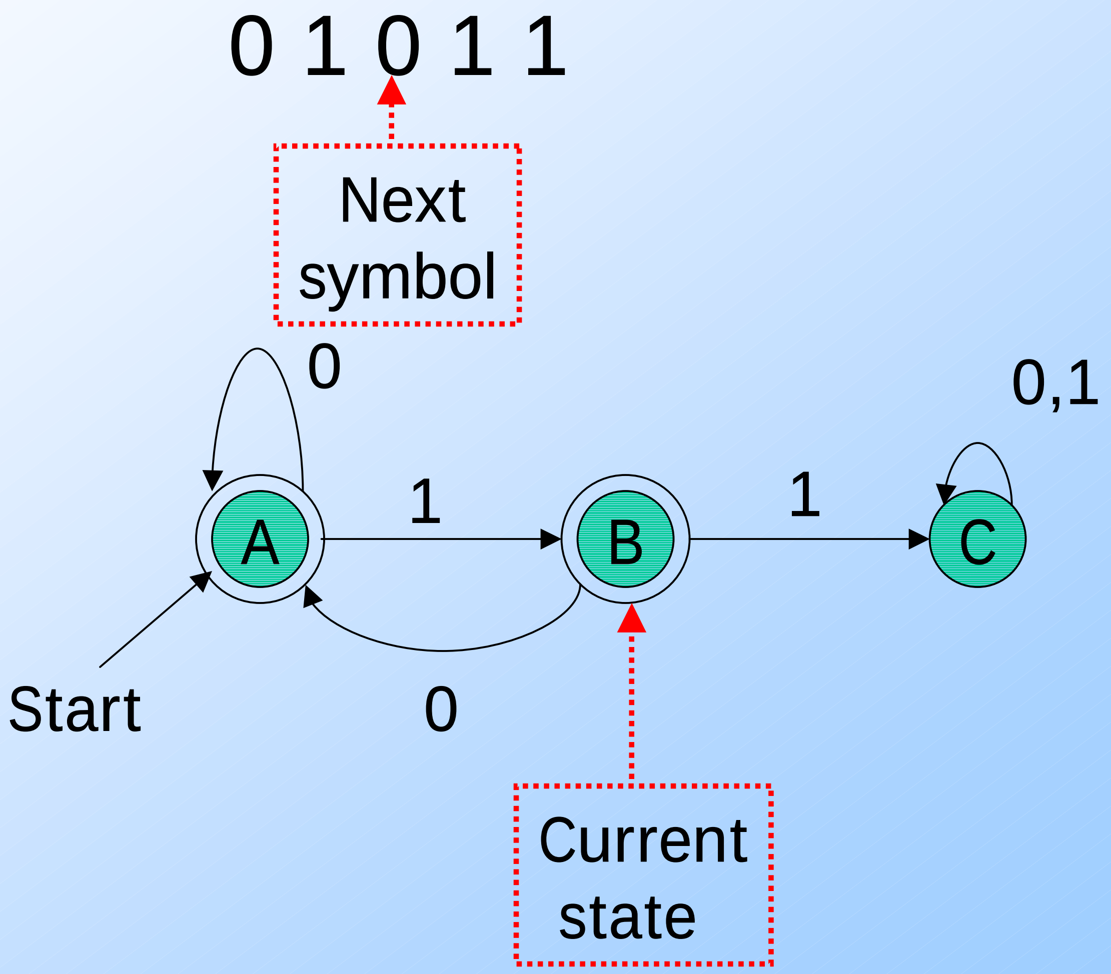
Example: Testing Membership



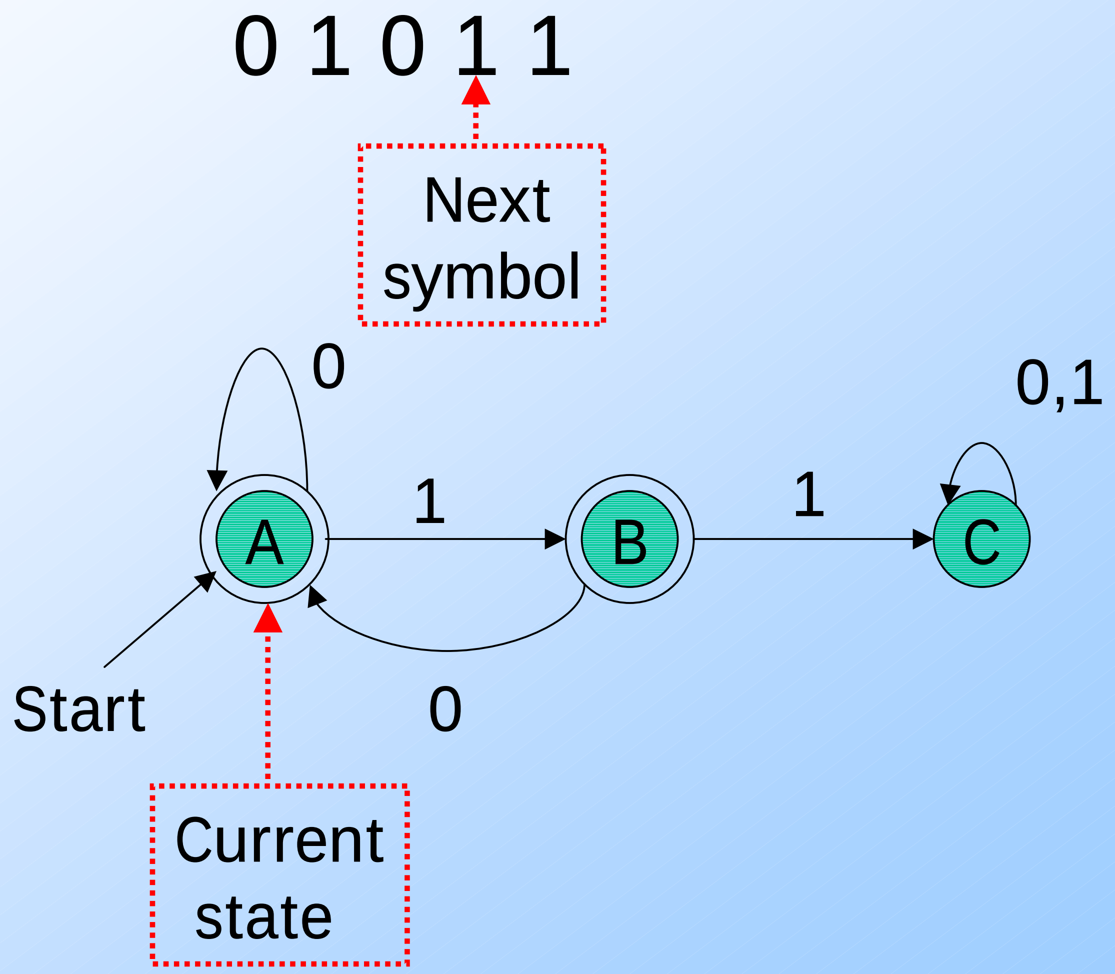
Example: Testing Membership



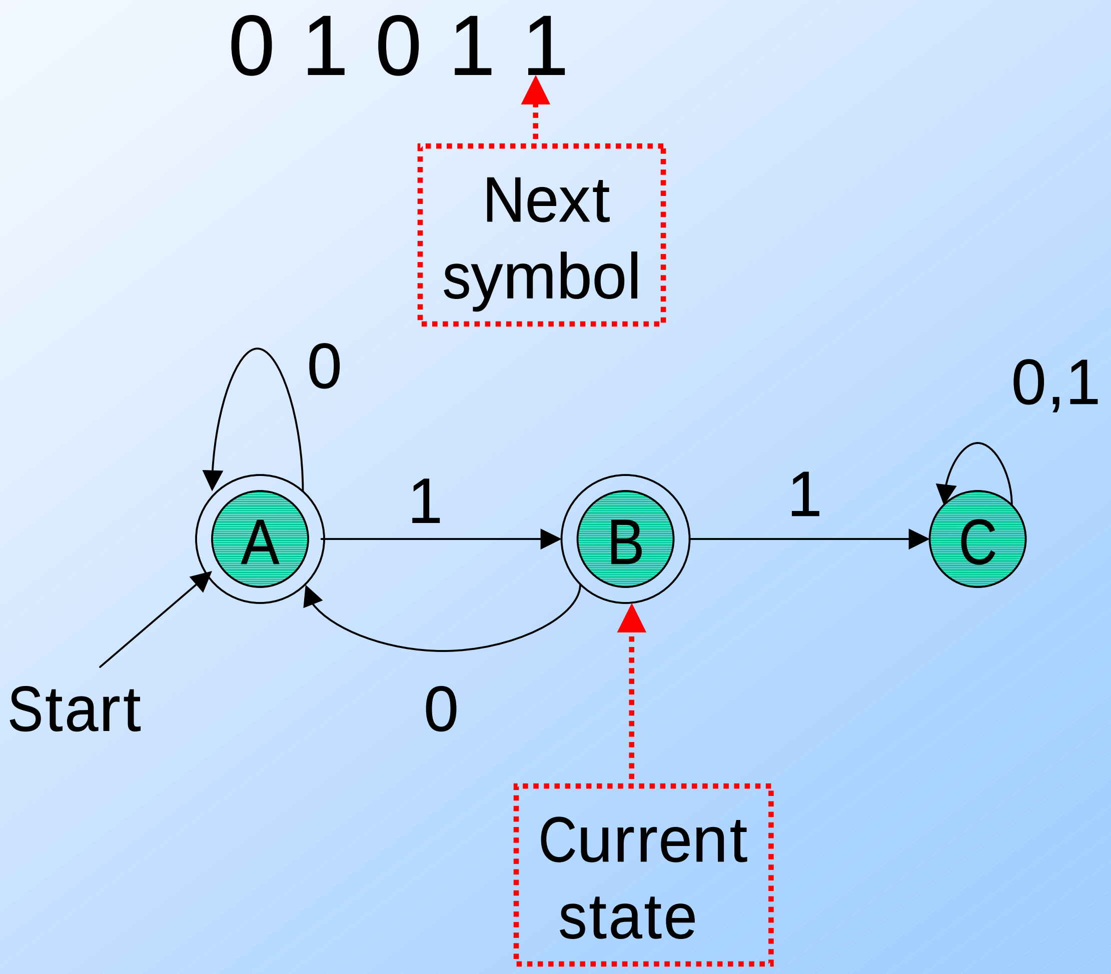
Example: Testing Membership



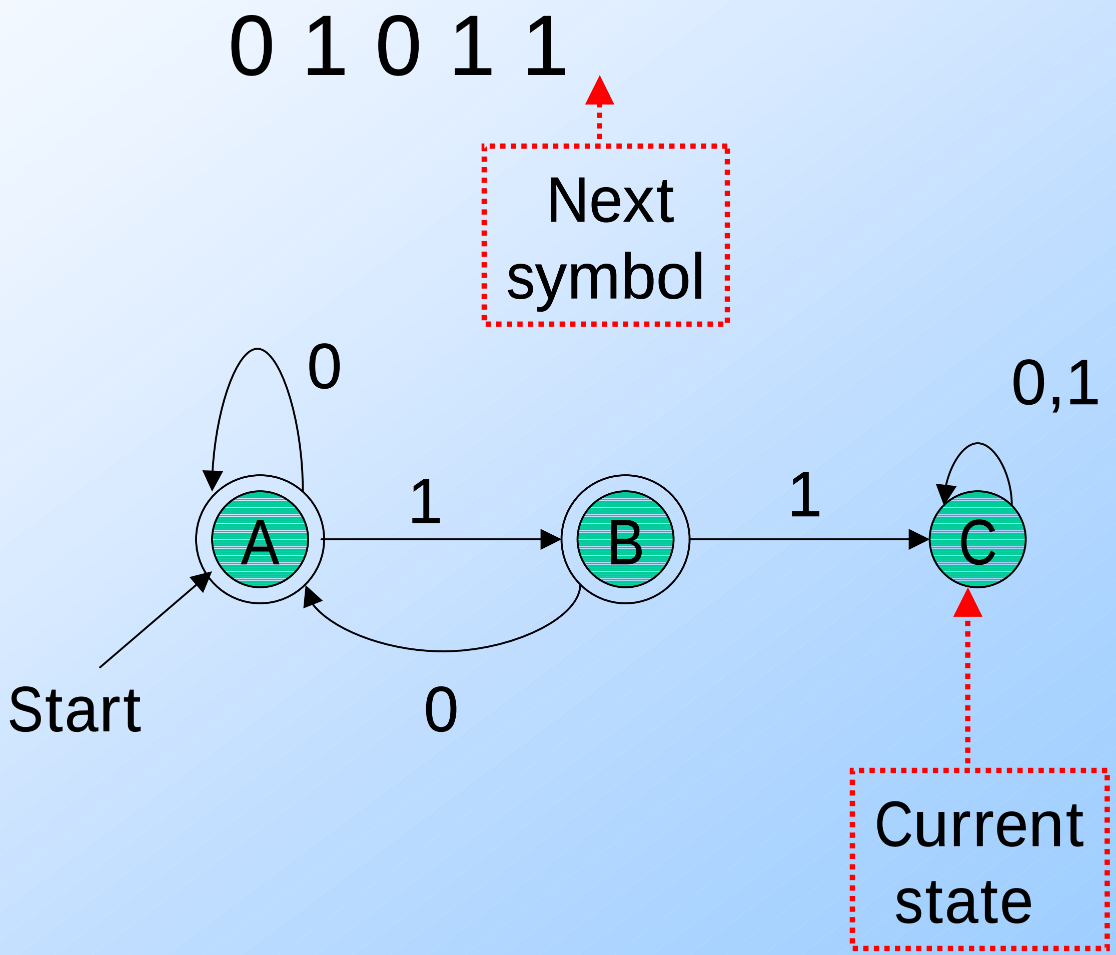
Example: Testing Membership



Example: Testing Membership



Example: Testing Membership



What if We Have the Wrong Representation?

- There is a circle of conversions from one form to another:

$$\begin{tikzcd}
 & \boxed{\text{RE}} \arrow[dl, bend right] & \\
\boxed{\epsilon\text{-NFA}} \arrow[dr, bend right] & & \boxed{\text{DFA}} \arrow[ul, bend right] \\
 & \boxed{\text{NFA}} \arrow[ur, bend right] &
\end{tikzcd}$$

The Emptiness Problem

- Given a regular language, does the language contain any string at all?
- Assume representation is DFA.
- Compute the set of states reachable from the start state.
- If at least one final state is reachable, then yes, else no.

The Infiniteness Problem

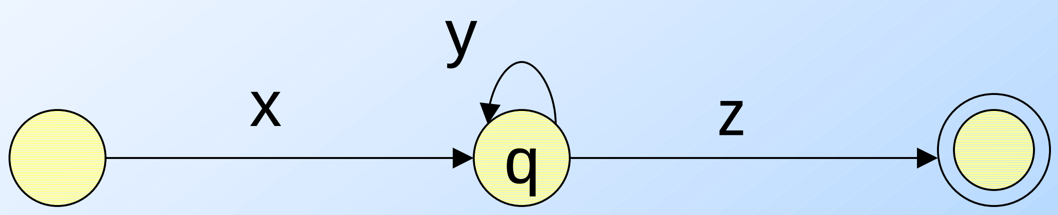
- Is a given regular language infinite?
- Start with a DFA for the language.
- Key idea: if the DFA has *n* states, and the language contains any string of length *n* or more, then the language is infinite.
- Otherwise, the language is surely finite.
  - Limited to strings of length *n* or less.

Proof of Key Idea

- If an n-state DFA accepts a string w of length *n* or more, then there must be a state that appears twice on the path labeled w from the start state to a final state.
- Because there are at least n+1 states along the path.

Proof – (2)

w = xyz



Then xy^iz is in the language for all $i \geq 0$.

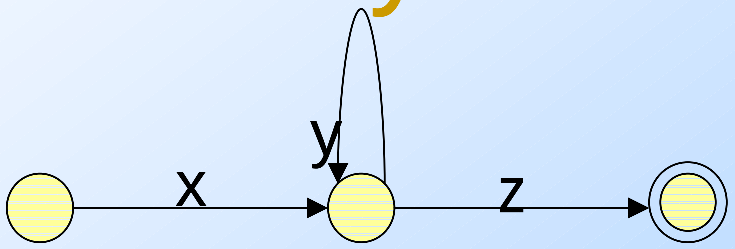
Since y is not ϵ, we see an infinite number of strings in L.

Infiniteness – Continued

- We do not yet have an algorithm.
- There are an infinite number of strings of length > n, and we can't test them all.
- Second key idea: if there is a string of length $\geq$ n (= number of states) in L, then there is a string of length between n and 2n-1.

Proof of 2nd Key Idea

- Remember:



- y is the first cycle on the path.
- So $|xy| \leq n$; in particular, $1 \leq |y| \leq n$.
- Thus, if w is of length 2n or more, there is a shorter string in L that is still of length at least n.
- Keep shortening to reach [n, 2n-1].

Completion of Infiniteness Algorithm

- Test for membership all strings of length between n and 2n-1.
  - If any are accepted, then infinite, else finite.
- A terrible algorithm.
- Better: find cycles between the start state and a final state.

Finding Cycles

1. Eliminate states not reachable from the start state.
2. Eliminate states that do not reach a final state.
3. Test if the remaining transition graph has any cycles.

Finding Cycles – (2)

- But a simple, less efficient way to find cycles is to search forward from a given node N.
- If you can reach N, then there is a cycle.
- Do this starting at each node.

The Pumping Lemma

- We have, almost accidentally, proved a statement that is quite useful for showing certain languages are not regular.
- Called the *pumping lemma for regular languages*.

Statement of the Pumping Lemma

For every regular language L

There is an integer n, such that

For every string w in L of length $\geq$ n

We can write w = xyz such that:

1. $|xy| \leq n$.
2. $|y| > 0$.
3. For all i $\geq$ 0, xy^iz is in L.

Number of states of DFA for L

Labels along first cycle on path labeled w

Example: Use of Pumping Lemma

- We have claimed $\{0^k1^k \mid k \geq 1\}$ is not a regular language.
- Suppose it were. Then there would be an associated n for the pumping lemma.
- Let $w = 0^n1^n$. We can write $w = xyz$, where x and y consist of 0's, and $y \neq \epsilon$.
- But then xyyz would be in L, and this string has more 0's than 1's.

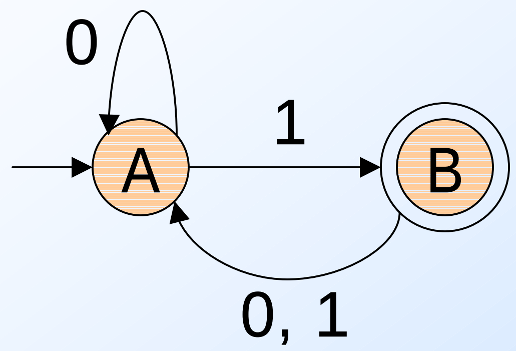
Decision Property: Equivalence

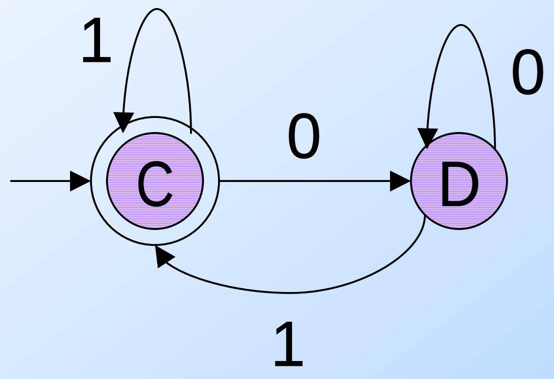
- Given regular languages L and M, is L = M?
- Algorithm involves constructing the *product DFA* from DFA's for L and M.
- Let these DFA's have sets of states Q and R, respectively.
- Product DFA has set of states Q $\times$ R.
  - I.e., pairs [q, r] with q in Q, r in R.

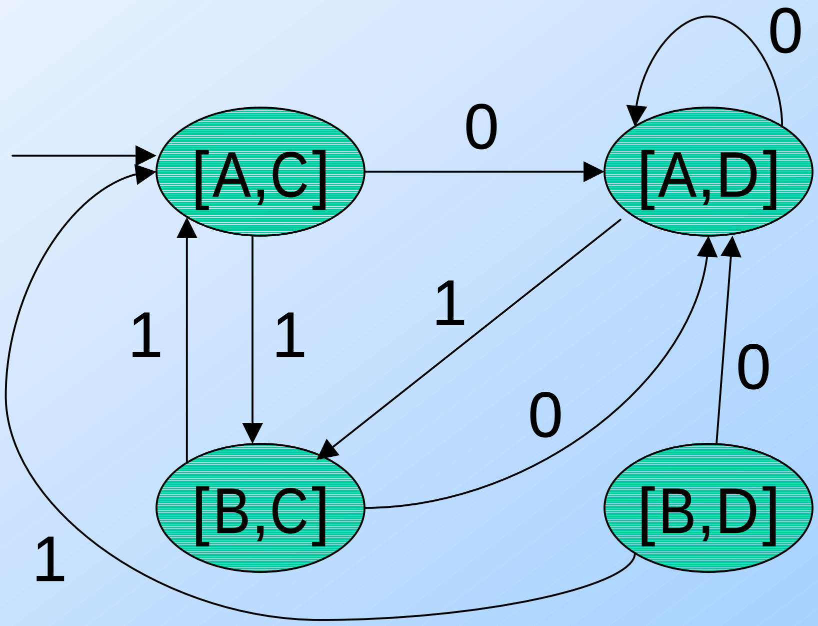
Product DFA – Continued

- Start state = $[q_0, r_0]$ (the start states of the DFA's for L, M).
- Transitions: $\delta([q,r], a) = [\delta_L(q,a), \delta_M(r,a)]$
  - δ_L, δ_M are the transition functions for the DFA's of L, M.
  - That is, we simulate the two DFA's in the two state components of the product DFA.

Example: Product DFA



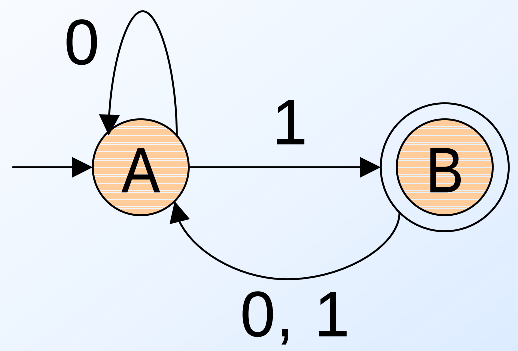


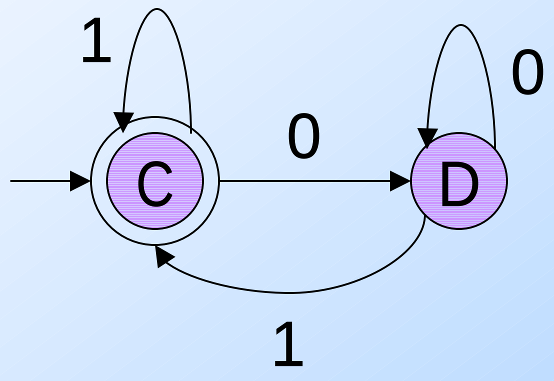


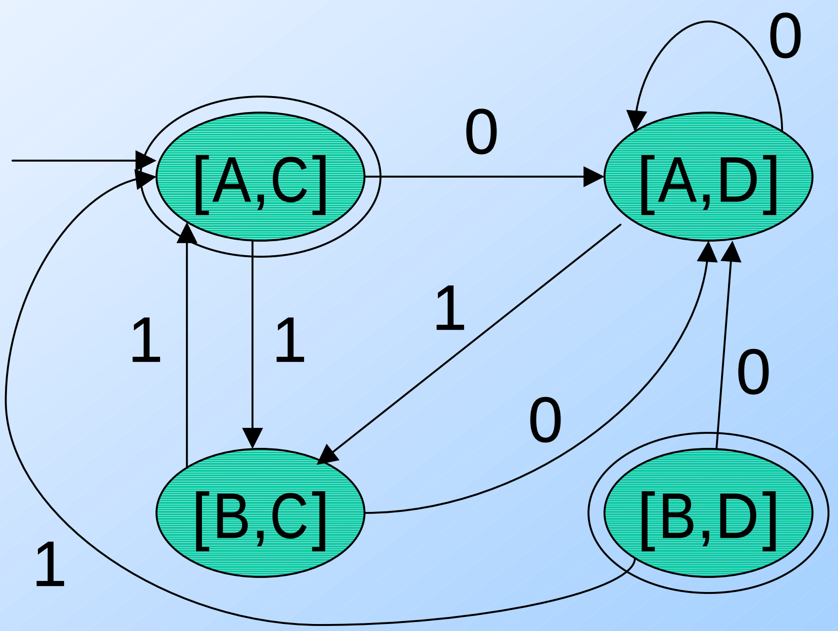
Equivalence Algorithm

- Make the final states of the product DFA be those states [q, r] such that exactly one of q and r is a final state of its own DFA.
- Thus, the product accepts w iff w is in exactly one of L and M.
- L = M if and only if the product automaton's language is empty.

Example: Equivalence





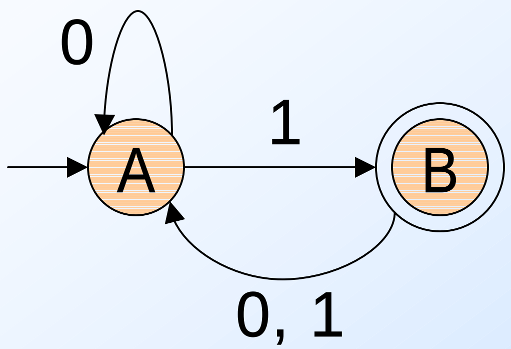


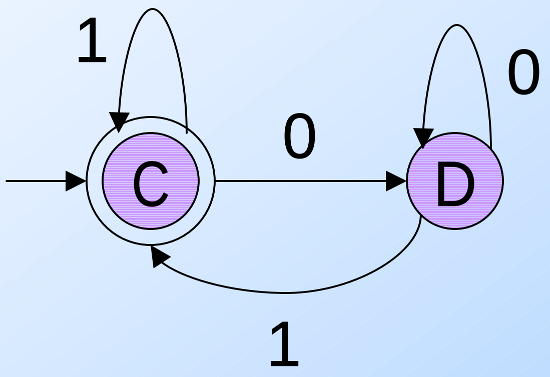
Decision Property: Containment

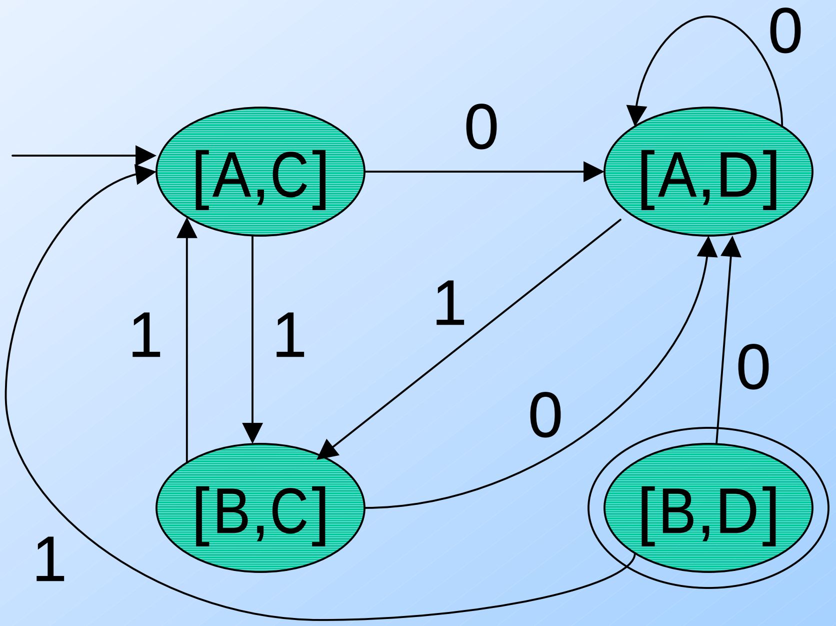
- Given regular languages L and M, is $L \subseteq M$?
- Algorithm also uses the product automaton.
- How do you define the final states [q, r] of the product so its language is empty iff $L \subseteq M$?

Answer: q is final; r is not.

Example: Containment







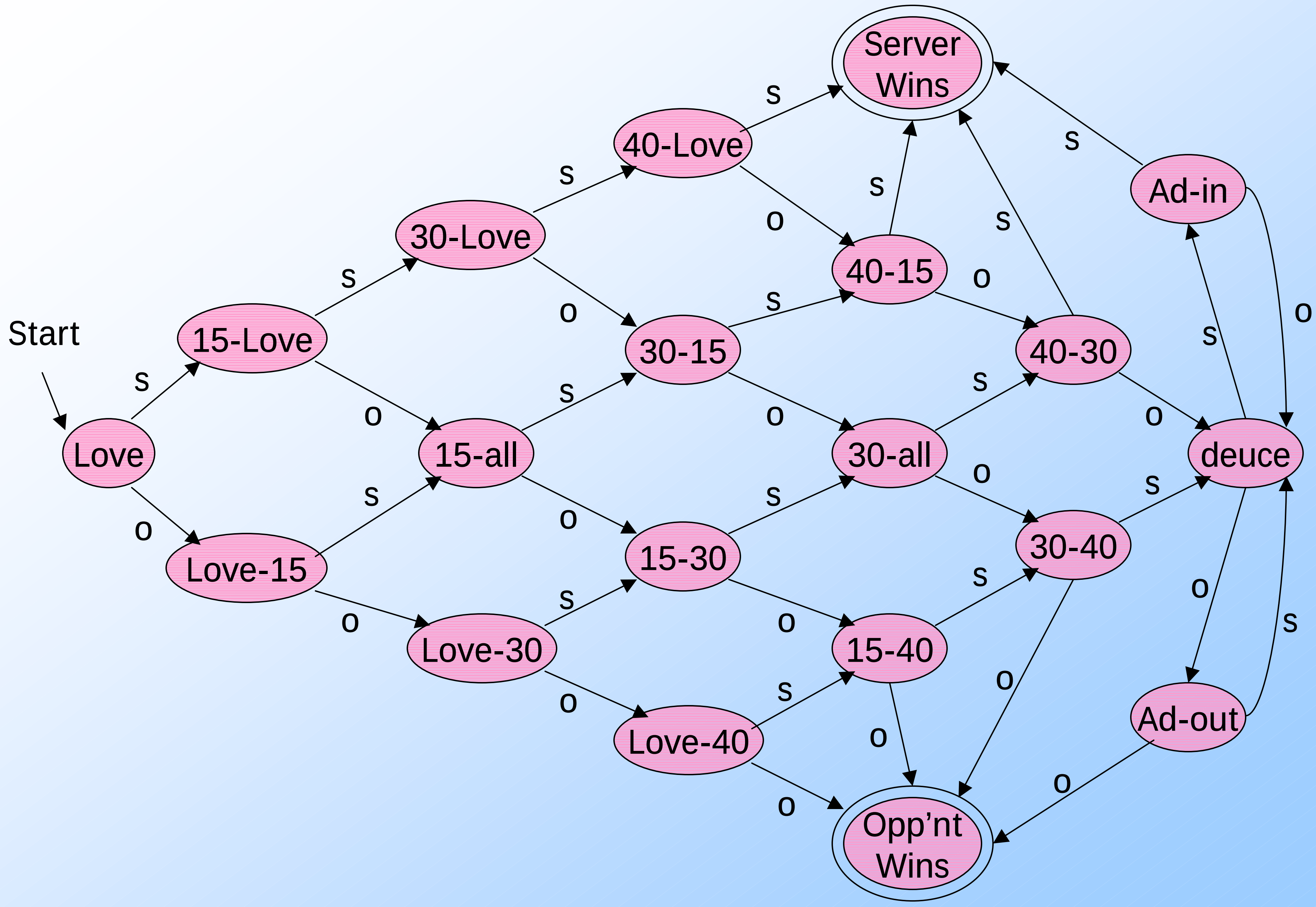
Note: the only final state is unreachable, so containment holds.

The Minimum-State DFA for a Regular Language

- In principle, since we can test for equivalence of DFA's we can, given a DFA *A* find the DFA with the fewest states accepting L(A).
- Test all smaller DFA's for equivalence with *A*.
- But that's a terrible algorithm.

Efficient State Minimization

- Construct a table with all pairs of states.
- If you find a string that *distinguishes* two states (takes exactly one to an accepting state), mark that pair.
- Algorithm is a recursion on the length of the shortest distinguishing string.


Start
Love
15-Love
Love-15
30-Love
15-all
Love-30
40-Love
30-15
15-30
Love-40
40-15
30-all
15-40
40-30
30-40
deuce
Ad-in
Ad-out
Server Wins
Opp'nt Wins
s
o

State Minimization – (2)

- Basis: Mark pairs with exactly one final state.
- Induction: mark [q, r] if for some input symbol *a*, [δ(q,a), δ(r,a)] is marked.
- After no more marks are possible, the unmarked pairs are equivalent and can be merged into one state.

Transitivity of “Indistinguishable”

- If state p is indistinguishable from q, and q is indistinguishable from r, then p is indistinguishable from r.
- Proof: The outcome (accept or don’t) of p and q on input w is the same, and the outcome of q and r on w is the same, then likewise the outcome of p and r.

Constructing the Minimum-State DFA

- Suppose $q_1,\ldots,q_k$ are indistinguishable states.
- Replace them by one *representative* state q.
- Then $\delta(q_1, a),\ldots, \delta(q_k, a)$ are all indistinguishable states.
  - Key point: otherwise, we should have marked at least one more pair.
- Let $\delta(q, a)$ = the representative state for that group.

Example: State Minimization

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}	{2,4,6,8}	{5}
*	{1,3,5,7,9}	{2,4,6,8}	{1,3,5,7,9}

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

Here it is with more convenient state names

Remember this DFA? It was constructed for the chessboard NFA by the subset construction.

Example – Continued

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

	G	F	E	D	C	B
A	x	x				
B	x	x				
C	x	x				
D	x	x				
E	x	x				
F						

Start with marks for the pairs with one of the final states F or G.

Example – Continued

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

$$\begin{array}{c|cccccc} & G & F & E & D & C & B \\ \hline A & x & x & & & & \\ B & x & x & & & & \\ C & x & x & & & & \\ D & x & x & & & & \\ E & x & x & & & & \\ F & & & & & & \end{array}$$

Input r gives no help, because the pair [B, D] is not marked.

Example – Continued

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

	G	F	E	D	C	B
A	x	x	x	x	x	
B	x	x	x	x	x	
C	x	x				
D	x	x				
E	x	x				
F	x					

But input b distinguishes {A,B,F} from {C,D,E,G}. For example, [A, C] gets marked because [C, F] is marked.

Example – Continued

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

	G	F	E	D	C	B
A	x	x	x	x	x	
B	x	x	x	x	x	
C	x	x	x	x		
D	x	x				
E	x	x				
F	x					

[C, D] and [C, E] are marked because of transitions on b to marked pair [F, G].

Example – Continued

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

	G	F	E	D	C	B
A	X	X	X	X	X	X
B	X	X	X	X	X	
C	X	X	X	X		
D	X	X				
E	X	X				
F	X					

[A, B] is marked because of transitions on r to marked pair [B, D].

[D, E] can never be marked, because on both inputs they go to the same state.

Example – Concluded

		r	b
→	A	B	C
	B	D	E
	C	D	F
	D	D	G
	E	D	G
*	F	D	C
*	G	D	G

		r	b
→	A	B	C
	B	H	H
	C	H	F
	H	H	G
*	F	H	C
*	G	H	G

	G	F	E	D	C	B
A	x	x	x	x	x	x
B	x	x	x	x	x	
C	x	x	x	x		
D	x	x				
E	x	x				
F	x					

Replace D and E by H.
Result is the minimum-state DFA.

Eliminating Unreachable States

- Unfortunately, combining indistinguishable states could leave us with unreachable states in the “minimum-state” DFA.
- Thus, before or after, remove states that are not reachable from the start state.

Clincher

- We have combined states of the given DFA wherever possible.
- Could there be another, completely unrelated DFA with fewer states?
- No. The proof involves minimizing the DFA we derived with the hypothetical better DFA.

Proof: No Unrelated, Smaller DFA

- Let A be our minimized DFA; let B be a smaller equivalent.
- Consider an automaton with the states of A and B combined.
- Use "distinguishable" in its contrapositive form:
  - If states q and p are indistinguishable, so are δ(q, a) and δ(p, a).

Inferring Indistinguishability

$$\begin{tikzcd}
 & q_0 \arrow[r, "a"] & q \arrow[r, "b"] & r \\
\text{Start states of A and B indistinguishable because L(A) = L(B).} \arrow[ur, dotted] \arrow[dr, dotted] & & \text{Must be indistinguishable} \arrow[u, dotted] \arrow[d, dotted] & \text{Must be indistinguishable} \arrow[u, dotted] \arrow[d, dotted] \\
 & p_0 \arrow[r, "a"] & p \arrow[r, "b"] & s
\end{tikzcd}$$

Inductive Hypothesis

- Every state q of A is indistinguishable from some state of B.
- Induction is on the length of the shortest string taking you from the start state of A to q.

Proof – (2)

- Basis: Start states of A and B are indistinguishable, because L(A) = L(B).
- Induction: Suppose w = xa is a shortest string getting A to state q.
- By the IH, x gets A to some state r that is indistinguishable from some state p of B.
- Then $\delta_A(r, a) = q$ is indistinguishable from $\delta_B(p, a)$.

Proof – (3)

- However, two states of A cannot be indistinguishable from the same state of B, or they would be indistinguishable from each other.
  - Violates transitivity of “indistinguishable.”
- Thus, B has at least as many states as A.

Closure Properties of Regular Languages

Union, Intersection, Difference, Concatenation, Kleene Closure, Reversal, Homomorphism, Inverse Homomorphism

Closure Under Union

- If L and M are regular languages, so is $L \cup M$.
- Proof: Let L and M be the languages of regular expressions R and S, respectively.
- Then R+S is a regular expression whose language is $L \cup M$.

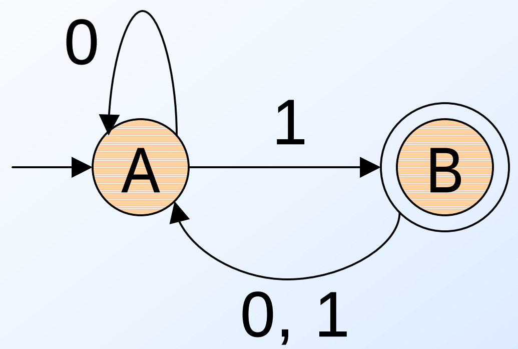
Closure Under Concatenation and Kleene Closure

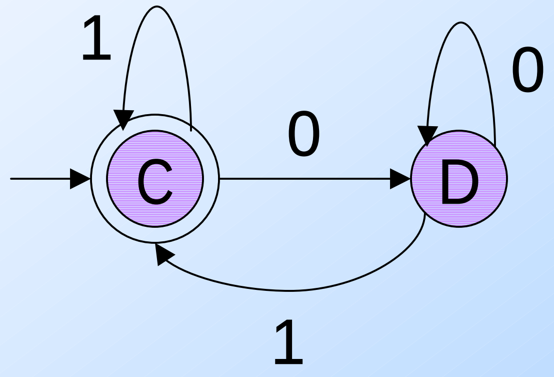
- Same idea:
  - RS is a regular expression whose language is LM.
  - R* is a regular expression whose language is L*.

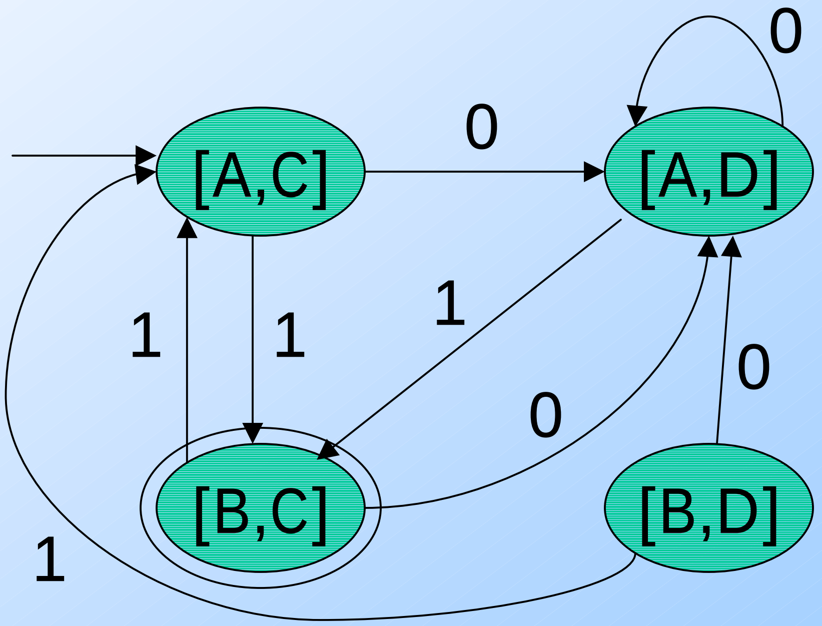
Closure Under Intersection

- If L and M are regular languages, then so is L $\cap$ M.
- Proof: Let A and B be DFA's whose languages are L and M, respectively.
- Construct C, the product automaton of A and B.
- Make the final states of C be the pairs consisting of final states of both A and B.

Example: Product DFA for Intersection







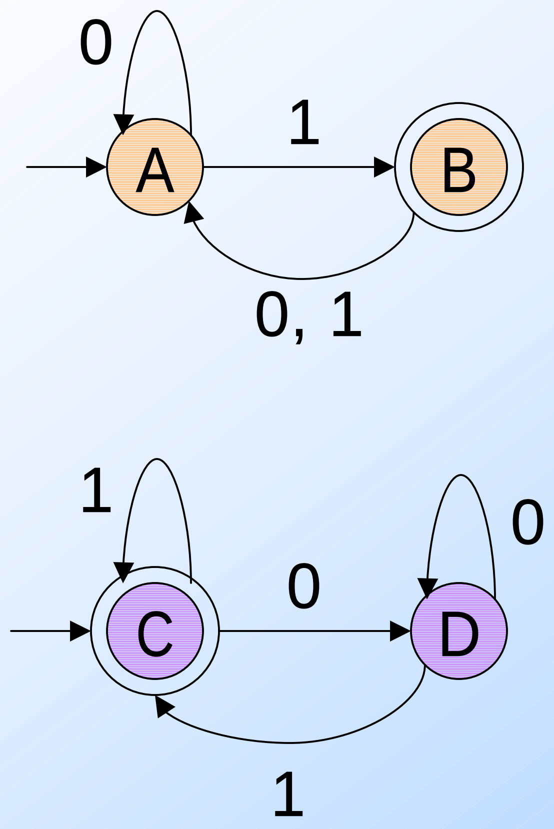
Example: Use of Closure Property

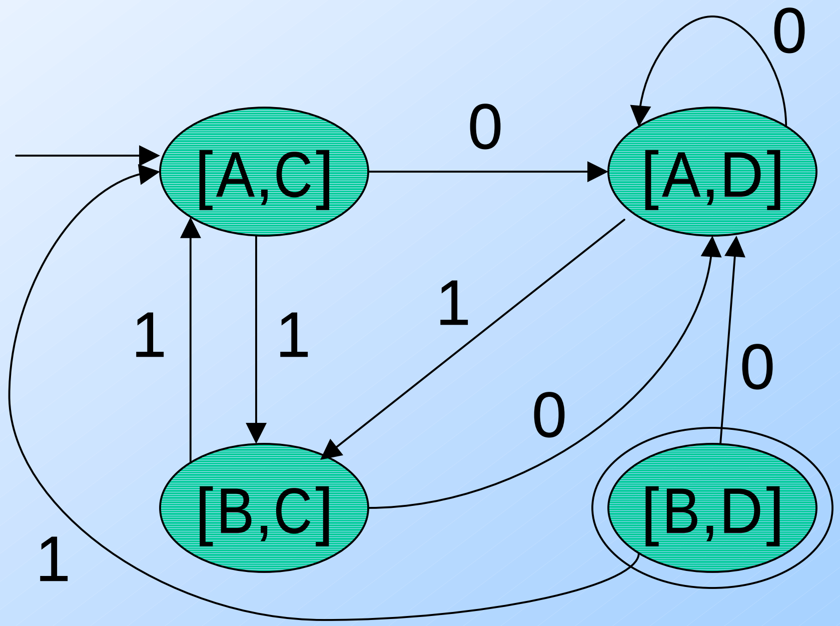
- We proved $L_1 = \{0^n1^n \mid n \geq 0\}$ is not a regular language.
- L_2 = the set of strings with an equal number of 0's and 1's isn't either, but that fact is trickier to prove.
- Regular languages are closed under $\cap$.
- If L_2 were regular, then $L_2 \cap L(\mathbf{0^*1^*}) = L_1$ would be, but it isn't.

Closure Under Difference

- If L and M are regular languages, then so is *L – M* = strings in L but not M.
- Proof: Let A and B be DFA's whose languages are L and M, respectively.
- Construct C, the product automaton of A and B.
- Final states of C are the pairs whose A-state is final but whose B-state is not.

Example: Product DFA for Difference





Closure Under Complementation

- The *complement* of a language L (with respect to an alphabet Σ such that Σ* contains L) is Σ* – L.
- Since Σ* is surely regular, the complement of a regular language is always regular.

Closure Under Reversal

- Recall example of a DFA that accepted the binary strings that, as integers were divisible by 23.
- We said that the language of binary strings whose reversal was divisible by 23 was also regular, but the DFA construction was tricky.
- Here’s the “tricky” construction.

Closure Under Reversal – (2)

- Given language L, L^R is the set of strings whose reversal is in L.
- Example: L = {0, 01, 100};
  L^R = {0, 10, 001}.
- Proof: Let E be a regular expression for L. We show how to reverse E, to provide a regular expression E^R for L^R.

Reversal of a Regular Expression

- Basis: If E is a symbol a, ϵ, or ∅, then $E^R = E$.
- Induction: If E is
  - F+G, then $E^R = F^R + G^R$.
  - FG, then $E^R = G^RF^R$
  - F*, then $E^R = (F^R)^*$.

Example: Reversal of a RE

- Let E = **01* + 10***.
- E^R = $(\mathbf{01^* + 10^*})^R = (\mathbf{01^*})^R + (\mathbf{10^*})^R$
- = $(\mathbf{1^*})^R\mathbf{0}^R + (\mathbf{0^*})^R\mathbf{1}^R$
- = $(\mathbf{1}^R)^*\mathbf{0} + (\mathbf{0}^R)^*\mathbf{1}$
- = **1*0 + 0*1**.

Homomorphisms

- A *homomorphism* on an alphabet is a function that gives a string for each symbol in that alphabet.
- Example: h(0) = ab; h(1) = ϵ.
- Extend to strings by $h(a_1...a_n) = h(a_1)...h(a_n)$.
- Example: h(01010) = ababab.

Closure Under Homomorphism

- If L is a regular language, and h is a homomorphism on its alphabet, then h(L) = {h(w) | w is in L} is also a regular language.
- Proof: Let E be a regular expression for L.
- Apply h to each symbol in E.
- Language of resulting RE is h(L).

Example: Closure under Homomorphism

- Let h(0) = ab; h(1) = ϵ.
- Let L be the language of regular expression **01*** + **10***.
- Then h(L) is the language of regular expression **ab**ϵ* + ϵ(**ab**)*.

Note: use parentheses to enforce the proper grouping.

Example – Continued

- **ab**ϵ* + ϵ(**ab**)* can be simplified.
- ϵ* = ϵ, so **ab**ϵ* = **ab**ϵ.
- ϵ is the identity under concatenation.
  - That is, ϵE = Eϵ = E for any RE *E*.
- Thus, **ab**ϵ + ϵ(**ab**)* = **ab** + (**ab**)*.
- Finally, L(**ab**) is contained in L((**ab**)*), so a RE for h(L) is (**ab**)*.

Inverse Homomorphisms

- Let h be a homomorphism and L a language whose alphabet is the output language of h.
- $h^{-1}(L)$ = {w | h(w) is in L}.

Example: Inverse Homomorphism

- Let h(0) = ab; h(1) = ϵ.
- Let L = {abab, baba}.
- $h^{-1}(L)$ = the language with two 0's and any number of 1's = L(**1*01*01***).

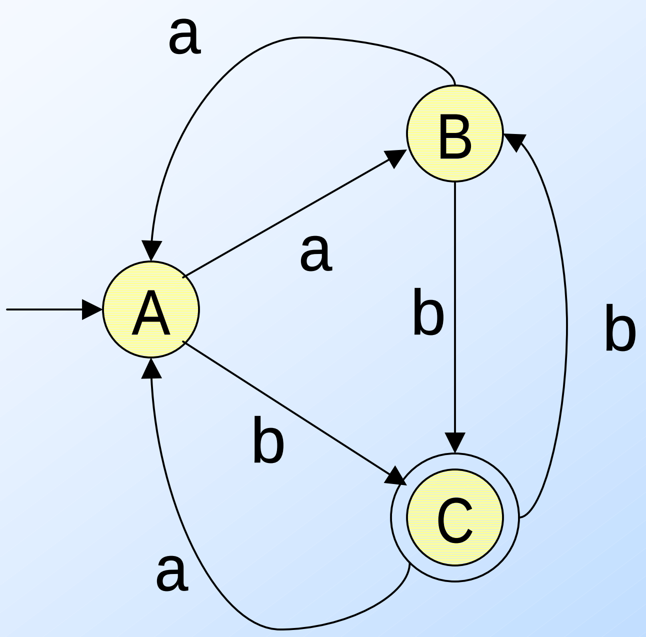
Closure Proof for Inverse Homomorphism

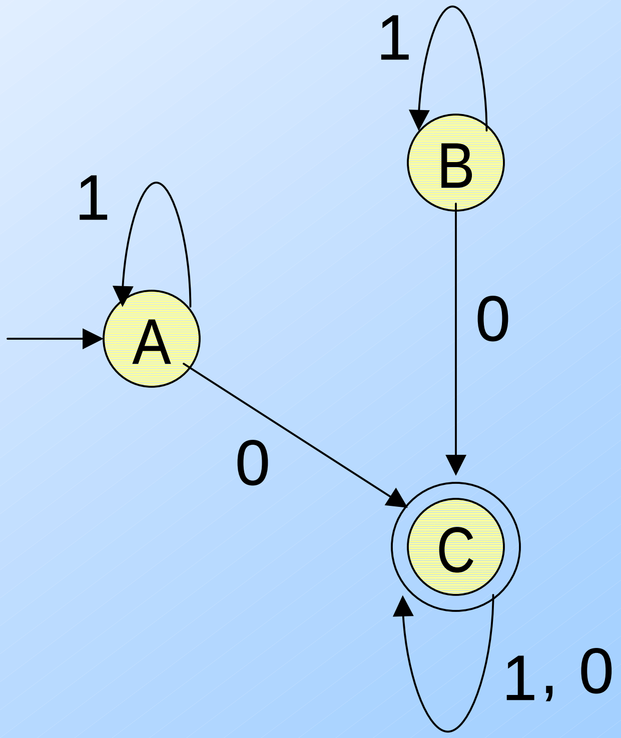
- Start with a DFA A for L.
- Construct a DFA B for $h^{-1}(L)$ with:
  - The same set of states.
  - The same start state.
  - The same final states.
  - Input alphabet = the symbols to which homomorphism h applies.

Proof – (2)

- The transitions for B are computed by applying h to an input symbol *a* and seeing where A would go on sequence of input symbols h(a).
- Formally, $\delta_B(q, a) = \delta_A(q, h(a))$.

Example: Inverse Homomorphism Construction





Since
$h(1) = \epsilon$

Since
$h(0) = ab$

$h(0) = ab$
$h(1) = \epsilon$

Proof – Inverse Homomorphism

- An induction on |w| (omitted) shows that $\delta_B(q_0, w) = \delta_A(q_0, h(w))$.
- Thus, B accepts w if and only if A accepts h(w).

Context-Free Grammars

Formalism

Derivations

Backus-Naur Form

Left- and Rightmost Derivations

Informal Comments

- A *context-free grammar* is a notation for describing languages.
- It is more powerful than finite automata or RE's, but still cannot define all possible languages.
- Useful for nested structures, e.g., parentheses in programming languages.

Informal Comments – (2)

- Basic idea is to use “variables” to stand for sets of strings (i.e., languages).
- These variables are defined recursively, in terms of one another.
- Recursive rules (“productions”) involve only concatenation.
- Alternative rules for a variable allow union.

Example: CFG for $\{ 0^n1^n \mid n \geq 1\}$

- Productions:

  S -> 01

  S -> 0S1
- Basis: 01 is in the language.
- Induction: if w is in the language, then so is 0w1.

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- *Start symbol* = the variable whose language is the one being defined.

Productions

- A *production* has the form variable (*head*) -> string of variables and terminals (*body*).
- Convention:
  - A, B, C,... and also S are variables.
  - a, b, c,... are terminals.
  - ..., X, Y, Z are either terminals or variables.
  - ..., w, x, y, z are strings of terminals only.
  - α, β, γ,... are strings of terminals and/or variables.

Example: Formal CFG

- Here is a formal CFG for { 0^n1^n | n $\geq$ 1}.
- Terminals = {0, 1}.
- Variables = {S}.
- Start symbol = S.
- Productions =

  S -> 01

  S -> 0S1

Derivations – Intuition

- We *derive* strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the body of one of its productions.
  - That is, the "productions for A" are those that have head A.

Derivations – Formalism

- We say $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production.
- Example: S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.

Iterated Derivation

- $=>^*$ means “zero or more derivation steps.”
- Basis: $\alpha =>^* \alpha$ for any string α.
- Induction: if $\alpha =>^* \beta$ and $\beta => \gamma$, then $\alpha =>^* \gamma$.

Example: Iterated Derivation

- S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.
- Thus S =>* S; S =>* 0S1;
  S =>* 00S11; S =>* 000111.

Sentential Forms

- Any string of variables and/or terminals derived from the start symbol is called a *sentential form*.
- Formally, α is a sentential form iff $S \Rightarrow^* \alpha$.

Language of a Grammar

- If G is a CFG, then L(G), the *language of G*, is {w | S =>* w}.
- Example: G has productions S -> ϵ and S -> 0S1.
- L(G) = $\{0^n1^n \mid n \geq 0\}$.

Context-Free Languages

- A language that is defined by some CFG is called a *context-free language*.
- There are CFL's that are not regular languages, such as the example just given.
- But not all languages are CFL's.
- Intuitively: CFL's can count two things, not three.

BNF Notation

- Grammars for programming languages are often written in BNF (*Backus-Naur Form*).
- Variables are words in <...>; Example: <statement>.
- Terminals are often multicharacter strings indicated by boldface or underline; Example: **while** or <u>WHILE</u>.

BNF Notation – (2)

- Symbol ::= is often used for ->.
- Symbol | is used for “or.”
  - A shorthand for a list of productions with the same left side.
- Example: S -> 0S1 | 01 is shorthand for S -> 0S1 and S -> 01.

BNF Notation – Kleene Closure

- Symbol ... is used for “one or more.”
- Example: <digit> ::= 0|1|2|3|4|5|6|7|8|9
  <unsigned integer> ::= <digit>...
- Translation: Replace α... with a new variable A and productions A -> Aα | α.

Example: Kleene Closure

- Grammar for unsigned integers can be replaced by:

  U -> UD | D

  D -> 0|1|2|3|4|5|6|7|8|9

BNF Notation: Optional Elements

- Surround one or more symbols by [...] to make them optional.
- Example: <statement> ::= **if** <condition> **then** <statement> [; **else** <statement>]
- Translation: replace [α] by a new variable A with productions A -> α | ϵ.

Example: Optional Elements

- Grammar for if-then-else can be replaced by:

S -> iCtSA

A -> ;eS | ϵ

BNF Notation – Grouping

- Use {...} to surround a sequence of symbols that need to be treated as a unit.
  - Typically, they are followed by a ... for “one or more.”
- Example: <statement list> ::= <statement> [{;<statement>}...]

Translation: Grouping

- Create a new variable A for {α}.
- One production for A: A -> α.
- Use A in place of {α}.

Example: Grouping

L -> S [{;S}...]

- Replace by L -> S [A...]   A -> ;S
  - A stands for {;S}.
- Then by L -> SB   B -> A... | ϵ   A -> ;S
  - B stands for [A...] (zero or more A's).
- Finally by L -> SB   B -> C | ϵ
  C -> AC | A   A -> ;S
  - C stands for A... .

Leftmost and Rightmost Derivations

- Derivations allow us to replace any of the variables in a string.
  - Leads to many different derivations of the same string.
- By forcing the leftmost variable (or alternatively, the rightmost variable) to be replaced, we avoid these “distinctions without a difference.”

Leftmost Derivations

- Say $wA\alpha \Rightarrow_{lm} w\beta\alpha$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.
- Also, $\alpha \Rightarrow^*_{lm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{lm}$ steps.

Example: Leftmost Derivations

- Balanced-parentheses grammmar:

  S -> SS | (S) | ()

- $S \Rightarrow_{lm} SS \Rightarrow_{lm} (S)S \Rightarrow_{lm} (())S \Rightarrow_{lm} (())()$
- Thus, $S \Rightarrow^*_{lm} (())()$
- $S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (())()$ is a derivation, but not a leftmost derivation.

Rightmost Derivations

- Say $\alpha A w \Rightarrow_{rm} \alpha\beta w$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.
- Also, $\alpha \Rightarrow^*_{rm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{rm}$ steps.

Example: Rightmost Derivations

- Balanced-parentheses grammmar:

  S -> SS | (S) | ()
- S $=>_{rm}$ SS $=>_{rm}$ S() $=>_{rm}$ (S)() $=>_{rm}$ (())()
- Thus, S $=>^*_{rm}$ (())()
- S => SS => SSS => S()S => ()()S => ()()() is neither a rightmost nor a leftmost derivation.

Parse Trees

Definitions

Relationship to Left- and Rightmost Derivations

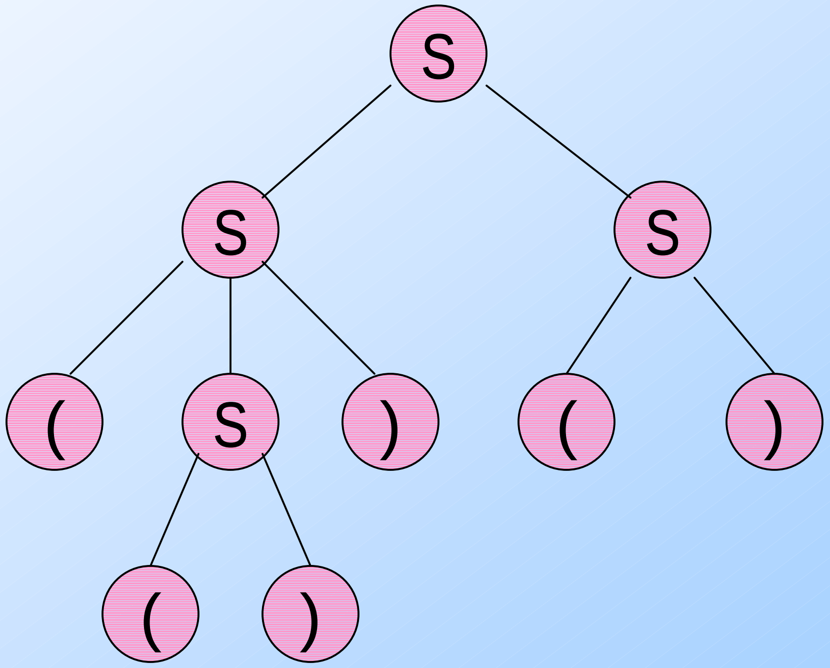
Ambiguity in Grammars

Parse Trees

- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ϵ.
- Interior nodes: labeled by a variable.
  - Children are labeled by the body of a production for the parent.
- Root: must be labeled by the start symbol.

Example: Parse Tree

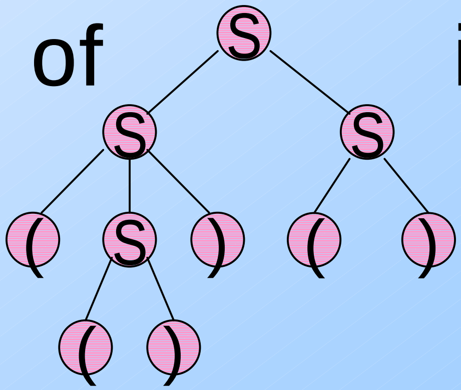
S -> SS | (S) | ()



Yield of a Parse Tree

- The concatenation of the labels of the leaves in left-to-right order
  - That is, in the order of a preorder traversal.

is called the *yield* of the parse tree.

- Example: yield of  is (())()

Generalization of Parse Trees

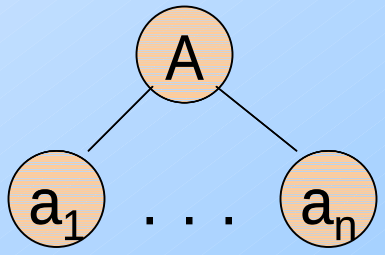
- We sometimes talk about trees that are not exactly parse trees, but only because the root is labeled by some variable A that is not the start symbol.
- Call these *parse trees with root A*.

Parse Trees, Leftmost and Rightmost Derivations

- Trees, leftmost, and rightmost derivations correspond.
- We'll prove:
  1. If there is a parse tree with root labeled A and yield w, then A $=>^*_{lm}$ w.
  2. If A $=>^*_{lm}$ w, then there is a parse tree with root A and yield w.

Proof – Part 1

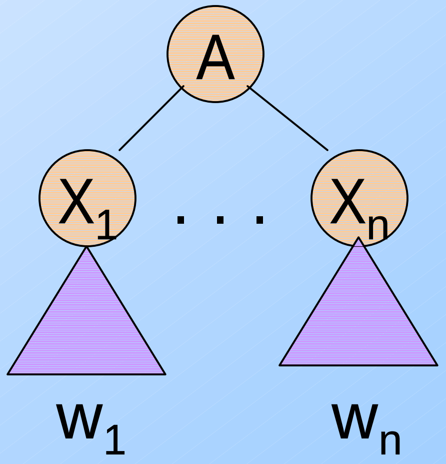
- Induction on the *height* (length of the longest path from the root) of the tree.
- Basis: height 1. Tree looks like



- A -> $a_1 \ldots a_n$ must be a production.
- Thus, A $\Rightarrow^*_{lm}$ $a_1 \ldots a_n$.

Part 1 – Induction

- Assume (1) for trees of height < h, and let this tree have height h:
- By IH, $X_i \Rightarrow^*_{lm} w_i$.
  - Note: if X_i is a terminal, then $X_i = w_i$.
- Thus, $A \Rightarrow_{lm} X_1 \ldots X_n \Rightarrow^*_{lm} w_1 X_2 \ldots X_n \Rightarrow^*_{lm} w_1 w_2 X_3 \ldots X_n \Rightarrow^*_{lm} \ldots \Rightarrow^*_{lm} w_1 \ldots w_n$.

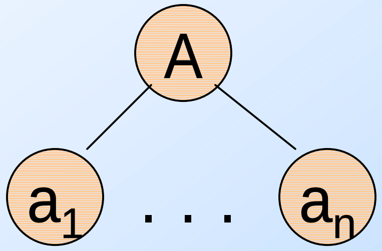


Proof: Part 2

- Given a leftmost derivation of a terminal string, we need to prove the existence of a parse tree.
- The proof is an induction on the length of the derivation.

Part 2 – Basis

- If $A \Rightarrow^*_{lm} a_1 \ldots a_n$ by a one-step derivation, then there must be a parse tree

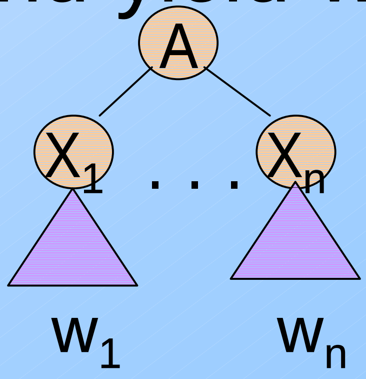


Part 2 – Induction

- Assume (2) for derivations of fewer than $k > 1$ steps, and let $A \Rightarrow^*_{lm} w$ be a k-step derivation.
- First step is $A \Rightarrow_{lm} X_1 \ldots X_n$.
- Key point: w can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.
  - If X_i is a terminal, then $w_i = X_i$.

Induction – (2)

- That is, $X_i =>^*_{lm} w_i$ for all i such that X_i is a variable.
  - And the derivation takes fewer than k steps.
- By the IH, if X_i is a variable, then there is a parse tree with root X_i and yield w_i.
- Thus, there is a parse tree



Parse Trees and Rightmost Derivations

- The ideas are essentially the mirror image of the proof for leftmost derivations.
- Left to the imagination.

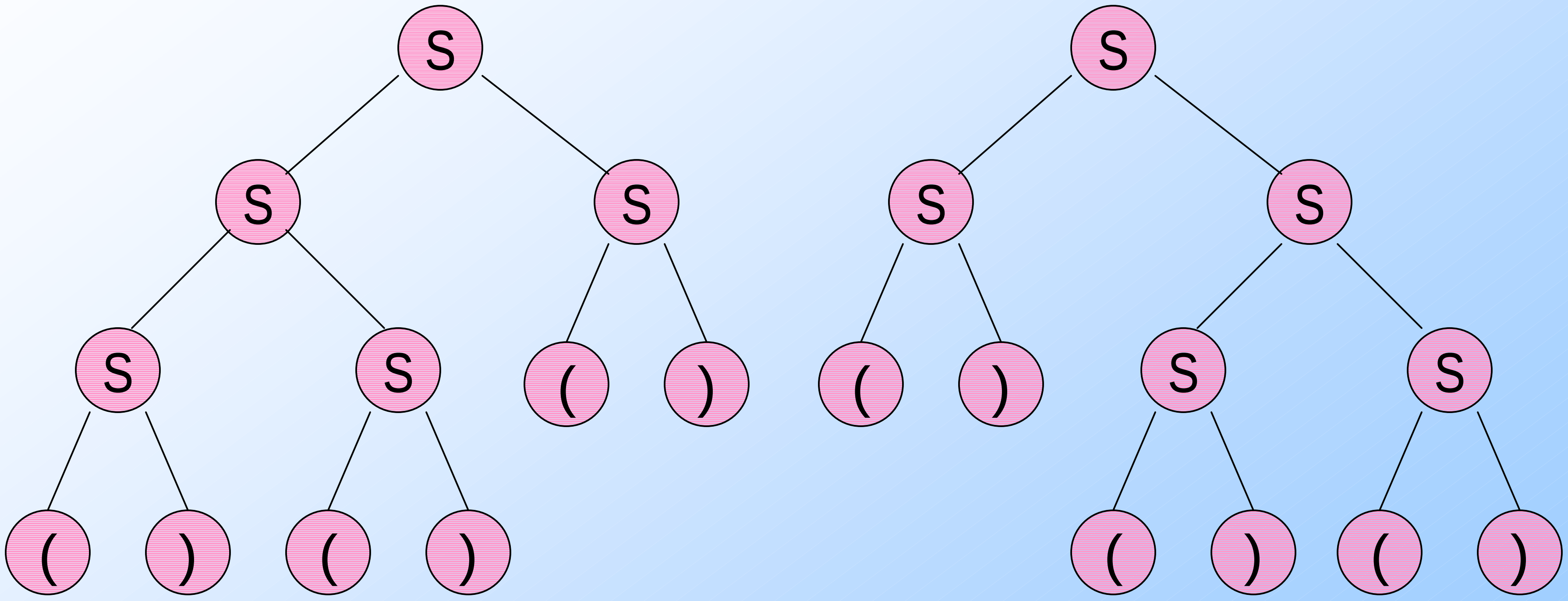
Parse Trees and Any Derivation

- The proof that you can obtain a parse tree from a leftmost derivation doesn't really depend on "leftmost."
- First step still has to be $A \Rightarrow X_1 \ldots X_n$.
- And w still can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.

Ambiguous Grammars

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.
- Example: S -> SS | (S) | ()
- Two parse trees for ()()() on next slide.

Example – Continued



Ambiguity, Left- and Rightmost Derivations

- If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof.
- Conversely, two different leftmost derivations produce different parse trees by the other part of the proof.
- Likewise for rightmost derivations.

Ambiguity, etc. – (2)

- Thus, equivalent definitions of "ambiguous grammar" are:
  1. There is a string in the language that has two different leftmost derivations.
  2. There is a string in the language that has two different rightmost derivations.

Ambiguity is a Property of Grammars, not Languages

- For the balanced-parentheses language, here is another CFG, which is unambiguous.

  B -> (RB | ε

  R ->) | (RR

B, the start symbol, derives balanced strings.

R generates certain strings that have one more right paren than left.

Example: Unambiguous Grammar

B -> (RB | ϵ   R ->) | (RR

- Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right.
    - If we need to expand B, then use B -> (RB if the next symbol is "("; use ϵ if at the end.
    - If we need to expand R, use R ->) if the next symbol is ")" and (RR if it is "(".

The Parsing Process

Remaining Input:

(())()

↑

Next symbol

Steps of leftmost derivation:

B

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

())()

↑

Next symbol

Steps of leftmost derivation:

B

(RB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

))()

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

)()

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

()

↑

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

)

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(()B

(())(RB

(())()B

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

(())()B

(())()

B -> (RB | ϵ   R ->) | (RR

LL(1) Grammars

- As an aside, a grammar such B -> (RB | ϵ R ->) | (RR, where you can always figure out the production to use in a leftmost derivation by scanning the given string left-to-right and looking only at the next one symbol is called LL(1).
  - “Leftmost derivation, left-to-right scan, one symbol of lookahead.”

LL(1) Grammars – (2)

- Most programming languages have LL(1) grammars.
- LL(1) grammars are never ambiguous.

Inherent Ambiguity

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity, as we did for the balanced-parentheses grammar.
- Unfortunately, certain CFL's are *inherently ambiguous*, meaning that every grammar for the language is ambiguous.

Example: Inherent Ambiguity

- The language $\{0^i1^j2^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.
- Intuitively, at least some of the strings of the form $0^n1^n2^n$ must be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

One Possible Ambiguous Grammar

S -> AB | CD	
A -> 0A1 | 01	A generates equal 0's and 1's
B -> 2B | 2	B generates any number of 2's
C -> 0C | 0	C generates any number of 0's
D -> 1D2 | 12	D generates equal 1's and 2's

And there are two derivations of every string with equal numbers of 0's, 1's, and 2's. E.g.:
S => AB => 01B =>012
S => CD => 0D => 012

Normal Forms for CFG’s

Eliminating Useless Variables

Removing Epsilon

Removing Unit Productions

Chomsky Normal Form

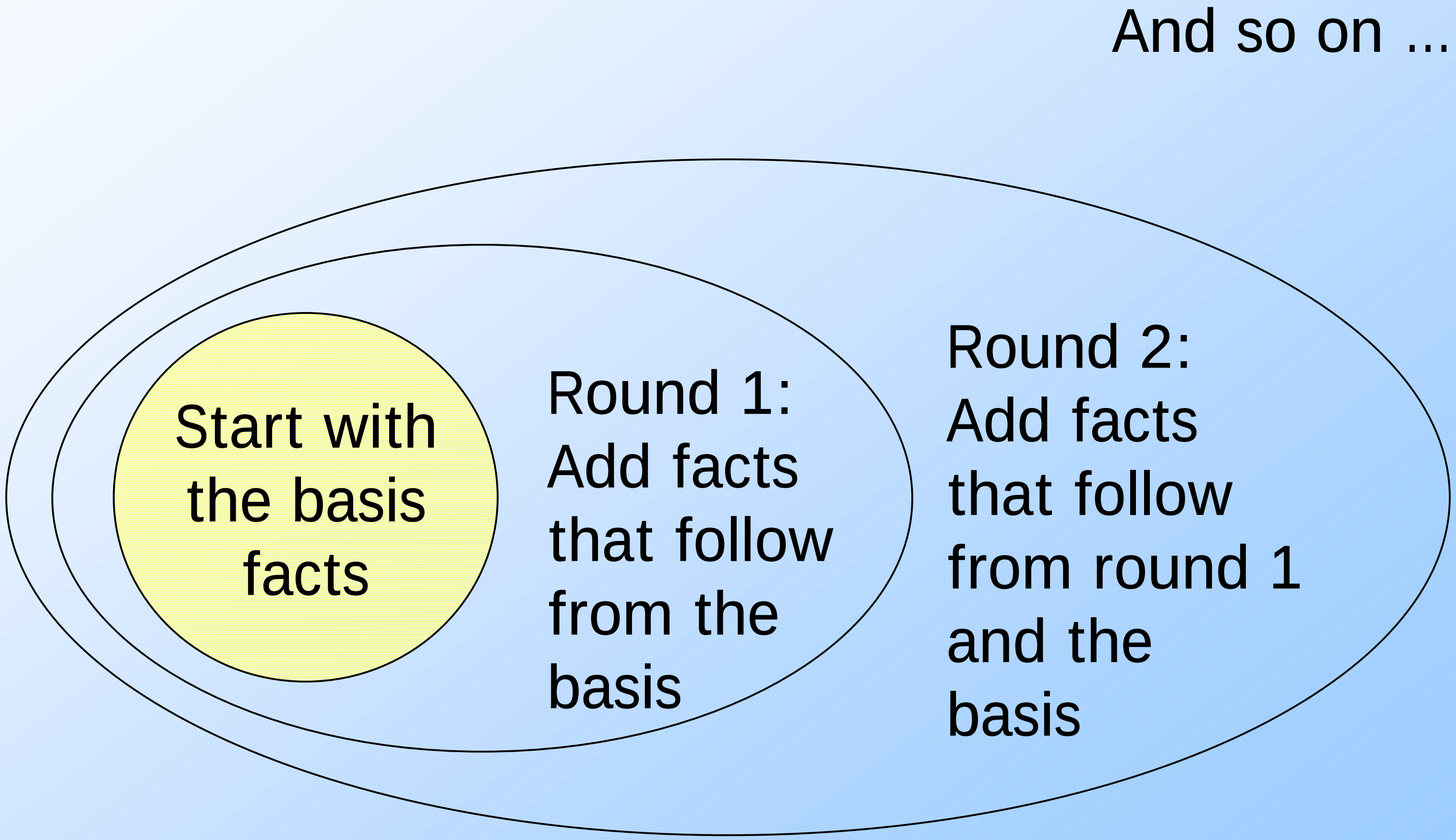
Variables That Derive Nothing

- Consider: S -> AB, A -> aA | a, B -> AB
- Although A derives all strings of a's, B derives no terminal strings.
  - Why? The only production for B leaves a B in the sentential form.
- Thus, S derives nothing, and the language is empty.

Discovery Algorithms

- There is a family of algorithms that work inductively.
- They start discovering some facts that are obvious (the basis).
- They discover more facts from what they already have discovered (induction).
- Eventually, nothing more can be discovered, and we are done.

Picture of Discovery


And so on ...
Start with the basis facts
Round 1: Add facts that follow from the basis
Round 2: Add facts that follow from round 1 and the basis

Testing Whether a Variable Derives Some Terminal String

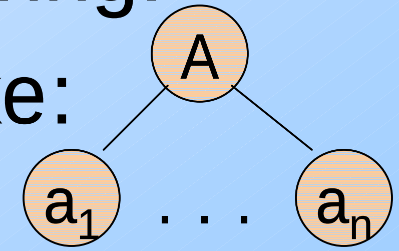
- Basis: If there is a production A -> w, where w has no variables, then A derives a terminal string.
- Induction: If there is a production A -> α, where α consists only of terminals and variables known to derive a terminal string, then A derives a terminal string.

Testing – (2)

- Eventually, we can find no more variables.
- An easy induction on the order in which variables are discovered shows that each one truly derives a terminal string.
- Conversely, any variable that derives a terminal string will be discovered by this algorithm.

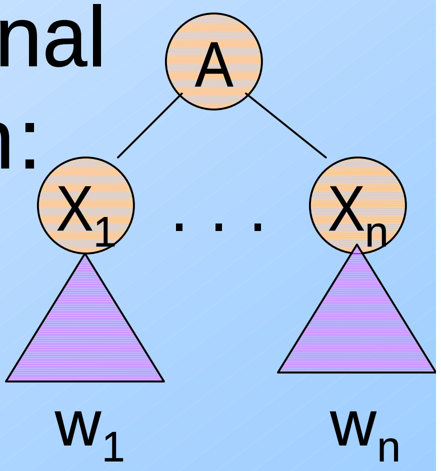
Proof of Converse

- The proof is an induction on the height of the least-height parse tree by which a variable A derives a terminal string.
- Basis: Height = 1. Tree looks like:
- Then the basis of the algorithm tells us that A will be discovered.



Induction for Converse

- Assume IH for parse trees of height < h, and suppose A derives a terminal string via a parse tree of height h:
- By IH, those X_i's that are variables are discovered.
- Thus, A will also be discovered, because it has a right side of terminals and/or discovered variables.



Algorithm to Eliminate Variables That Derive Nothing

1. Discover all variables that derive terminal strings.
2. For all other variables, remove all productions in which they appear in either the head or body.

Example: Eliminate Variables

S -> AB | C, A -> aA | a, B -> bB, C -> c

- Basis: A and C are discovered because of A -> a and C -> c.
- Induction: S is discovered because of S -> C.
- Nothing else can be discovered.
- Result: S -> C, A -> aA | a, C -> c

Unreachable Symbols

- Another way a terminal or variable deserves to be eliminated is if it cannot appear in any derivation from the start symbol.
- Basis: We can reach S (the start symbol).
- Induction: if we can reach A, and there is a production A -> α, then we can reach all symbols of α.

Unreachable Symbols – (2)

- Easy inductions in both directions show that when we can discover no more symbols, then we have all and only the symbols that appear in derivations from S.
- Algorithm: Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

Eliminating Useless Symbols

- A symbol is *useful* if it appears in some derivation of some terminal string from the start symbol.
- Otherwise, it is *useless*.
  Eliminate all useless symbols by:
  1. Eliminate symbols that derive no terminal string.
  2. Eliminate unreachable symbols.

Example: Useless Symbols – (2)

S -> AB, A -> C, C -> c, B -> bB

- If we eliminated unreachable symbols first, we would find everything is reachable.
- A, C, and c would never get eliminated.

Why It Works

- After step (1), every symbol remaining derives some terminal string.
- After step (2) the only symbols remaining are all derivable from S.
- In addition, they still derive a terminal string, because such a derivation can only involve symbols reachable from S.

Epsilon Productions

- We can almost avoid using productions of the form A -> ϵ (called *ϵ-productions*).
  - The problem is that ϵ cannot be in the language of any grammar that has no ϵ–productions.
- Theorem: If L is a CFL, then L-{ϵ} has a CFG with no ϵ-productions.

Nullable Symbols

- To eliminate ϵ-productions, we first need to discover the *nullable symbols* = variables A such that A =>* ϵ.
- Basis: If there is a production A -> ϵ, then A is nullable.
- Induction: If there is a production A -> α, and all symbols of α are nullable, then A is nullable.

Example: Nullable Symbols

S -> AB, A -> aA | ϵ, B -> bB | A

- Basis: A is nullable because of A -> ϵ.
- Induction: B is nullable because of B -> A.
- Then, S is nullable because of S -> AB.

Eliminating ϵ-Productions

- Key idea: turn each production $A \rightarrow X_1 \ldots X_n$ into a family of productions.
- For each subset of nullable X's, there is one production with those eliminated from the right side "in advance."
  - Except, if all X's are nullable (or the body was empty to begin with), do not make a production with ϵ as the right side.

Example: Eliminating ϵ-Productions

S -> ABC, A -> aA | ϵ, B -> bB | ϵ, C -> ϵ

- A, B, C, and S are all nullable.
- New grammar:

  S -> ~~ABC~~ | AB | ~~AC~~ | ~~BC~~ | A | B | ~~C~~

  A -> aA | a

  B -> bB | b

Note: C is now useless.
Eliminate its productions.

Why it Works

- Prove that for all variables A:
  1. If $w \neq \epsilon$ and $A \Rightarrow^*_{old} w$, then $A \Rightarrow^*_{new} w$.
  2. If $A \Rightarrow^*_{new} w$ then $w \neq \epsilon$ and $A \Rightarrow^*_{old} w$.
- Then, letting A be the start symbol proves that L(new) = L(old) – $\{\epsilon\}$.
- (1) is an induction on the number of steps by which A derives w in the old grammar.

Proof of 1 – Basis

- If the old derivation is one step, then A -> w must be a production.
- Since w ≠ ϵ, this production also appears in the new grammar.
- Thus, $A \Rightarrow_{new} w$.

Proof of 1 – Induction

- Let A $=>^*_{old}$ w be a k-step derivation, and assume the IH for derivations of fewer than k steps.
- Let the first step be A $=>_{old}$ $X_1 \ldots X_n$.
- Then w can be broken into w = $w_1 \ldots w_n$, where $X_i =>^*_{old} w_i$, for all i, in fewer than k steps.

Induction – Continued

- By the IH, if $w_i \neq \epsilon$, then $X_i \Rightarrow^*_{new} w_i$.
- Also, the new grammar has a production with A on the left, and just those X_i's on the right such that $w_i \neq \epsilon$.
  - Note: they all can't be ϵ, because $w \neq \epsilon$.
- Follow a use of this production by the derivations $X_i \Rightarrow^*_{new} w_i$ to show that A derives w in the new grammar.

Unit Productions

- A *unit production* is one whose body consists of exactly one variable.
- These productions can be eliminated.
- Key idea: If A =>* B by a series of unit productions, and B -> α is a non-unit-production, then add production A -> α.
- Then, drop all unit productions.

Unit Productions – (2)

- Find all pairs (A, B) such that A =>* B by a sequence of unit productions only.
- Basis: Surely (A, A).
- Induction: If we have found (A, B), and B -> C is a unit production, then add (A, C).

Proof That We Find Exactly the Right Pairs

- By induction on the order in which pairs (A, B) are found, we can show A =>* B by unit productions.
- Conversely, by induction on the number of steps in the derivation by unit productions of A =>* B, we can show that the pair (A, B) is discovered.

Proof The the Unit-Production-Elimination Algorithm Works

- Basic idea: there is a leftmost derivation A $=>^*_{lm}$ w in the new grammar if and only if there is such a derivation in the old.
- A sequence of unit productions and a non-unit production is collapsed into a single production of the new grammar.

Cleaning Up a Grammar

- Theorem: if L is a CFL, then there is a CFG for L – {ϵ} that has:
  1. No useless symbols.
  2. No ϵ-productions.
  3. No unit productions.
- I.e., every body is either a single terminal or has length $\geq$ 2.

Cleaning Up – (2)

- Proof: Start with a CFG for L.
- Perform the following steps in order:

1. Eliminate ϵ-productions.
2. Eliminate unit productions.
3. Eliminate variables that derive no terminal string.
4. Eliminate variables not reached from the start symbol.

Must be first. Can create unit productions or useless variables.

Chomsky Normal Form

- A CFG is said to be in *Chomsky Normal Form* if every production is of one of these two forms:
  1. A -> BC (body is two variables).
  2. A -> a (body is a single terminal).
- Theorem: If L is a CFL, then L – {ϵ} has a CFG in CNF.

Proof of CNF Theorem

- Step 1: “Clean” the grammar, so every body is either a single terminal or of length at least 2.
- Step 2: For each body $\neq$ a single terminal, make the right side all variables.
  - For each terminal *a* create new variable A_a and production A_a -> a.
  - Replace *a* by A_a in bodies of length $\geq$ 2.

Example: Step 2

- Consider production A -> BcDe.
- We need variables A_c and A_e. with productions A_c -> c and A_e -> e.
  - Note: you create at most one variable for each terminal, and use it everywhere it is needed.
- Replace A -> BcDe by A -> BA_cDA_e.

CNF Proof – Continued

- Step 3: Break right sides longer than 2 into a chain of productions with right sides of two variables.
- Example: A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.
  - F and G must be used nowhere else.

Example of Step 3 – Continued

- Recall A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.
- In the new grammar, A => BF => BCG => BCDE.
- More importantly: Once we choose to replace A by BF, we must continue to BCG and BCDE.
  - Because F and G have only one production.

Pushdown Automata

Definition

Moves of the PDA

Languages of the PDA

Deterministic PDA’s

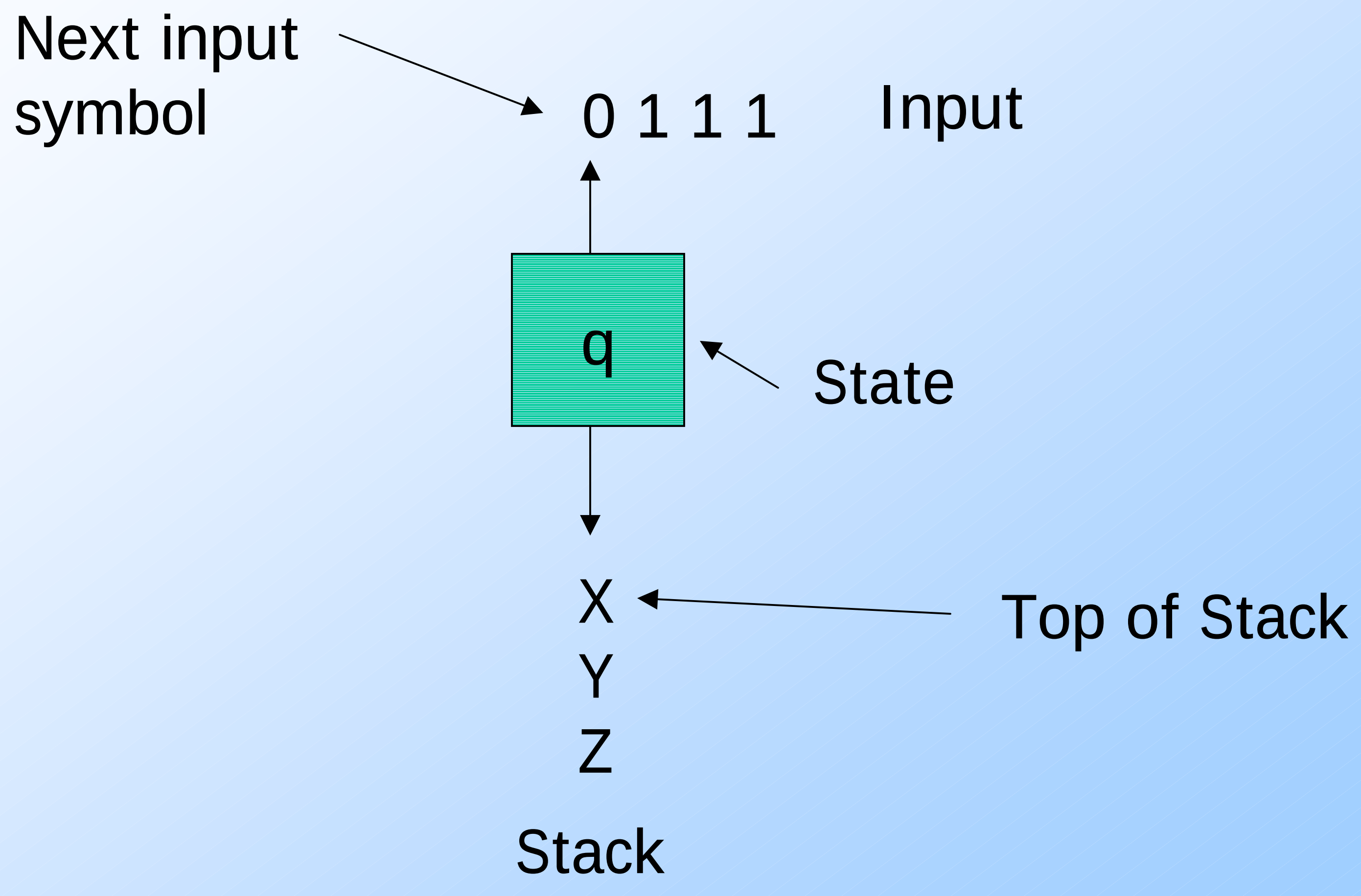
Pushdown Automata

- The PDA is an automaton equivalent to the CFG in language-defining power.
- Only the nondeterministic PDA defines all the CFL's.
- But the deterministic version models parsers.
  - Most programming languages have deterministic PDA's.

Intuition: PDA

- Think of an ϵ-NFA with the additional power that it can manipulate a stack.
- Its moves are determined by:
  1. The current state (of its "NFA"),
  2. The current input symbol (or ϵ), and
  3. The current symbol on top of its stack.

Picture of a PDA



Intuition: PDA – (2)

- Being nondeterministic, the PDA can have a choice of next moves.
- In each choice, the PDA can:
  1. Change state, and also
  2. Replace the top symbol on the stack by a sequence of zero or more symbols.
     - Zero symbols = “pop.”
     - Many symbols = sequence of “pushes.”

PDA Formalism

- A PDA is described by:
  1. A finite set of *states* (Q, typically).
  2. An *input alphabet* (Σ, typically).
  3. A *stack alphabet* (Γ, typically).
  4. A *transition function* (δ, typically).
  5. A *start state* (q_0, in Q, typically).
  6. A *start symbol* (Z_0, in Γ, typically).
  7. A set of *final states* ($F \subseteq Q$, typically).

Conventions

- a, b, ... are input symbols.
  - But sometimes we allow ϵ as a possible value.
- ..., X, Y, Z are stack symbols.
- ..., w, x, y, z are strings of input symbols.
- α, β,... are strings of stack symbols.

The Transition Function

- Takes three arguments:
  1. A state, in Q.
  2. An input, which is either a symbol in Σ or ϵ.
  3. A stack symbol in Γ.
- δ(q, a, Z) is a set of zero or more actions of the form (p, α).
  - p is a state; α is a string of stack symbols.

Actions of the PDA

- If $\delta(q, a, Z)$ contains (p, α) among its actions, then one thing the PDA can do in state q, with *a* at the front of the input, and Z on top of the stack is:
  1. Change the state to p.
  2. Remove *a* from the front of the input (but *a* may be ϵ).
  3. Replace Z on the top of the stack by α.

Example: PDA

- Design a PDA to accept $\{0^n1^n \mid n \geq 1\}$.
- The states:
  - q = start state. We are in state q if we have seen only 0's so far.
  - p = we've seen at least one 1 and may now proceed only if the inputs are 1's.
  - f = final state; accept.

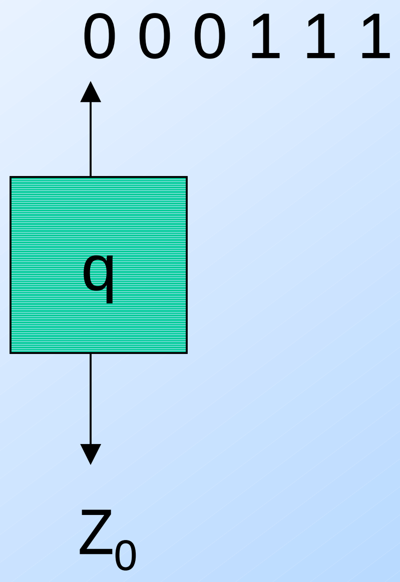
Example: PDA – (2)

- The stack symbols:
  - Z_0 = start symbol. Also marks the bottom of the stack, so we know when we have counted the same number of 1's as 0's.
  - X = marker, used to count the number of 0's seen on the input.

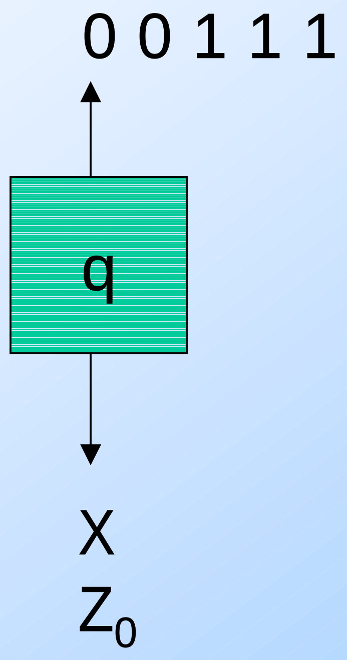
Example: PDA – (3)

- The transitions:
  - $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.
  - $\delta(q, 0, X) = \{(q, XX)\}$. These two rules cause one X to be pushed onto the stack for each 0 read from the input.
  - $\delta(q, 1, X) = \{(p, \epsilon)\}$. When we see a 1, go to state p and pop one X.
  - $\delta(p, 1, X) = \{(p, \epsilon)\}$. Pop one X per 1.
  - $\delta(p, \epsilon, Z_0) = \{(f, Z_0)\}$. Accept at bottom.

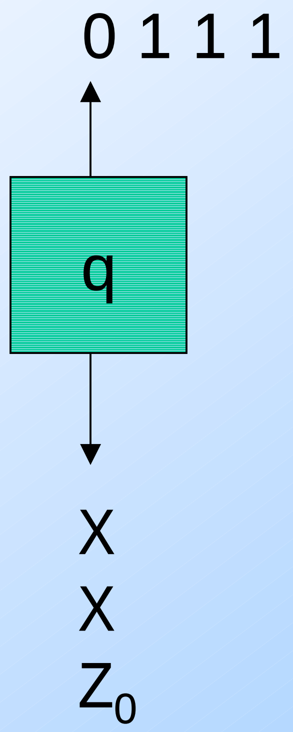
Actions of the Example PDA



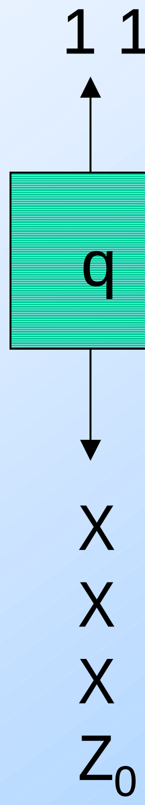
Actions of the Example PDA



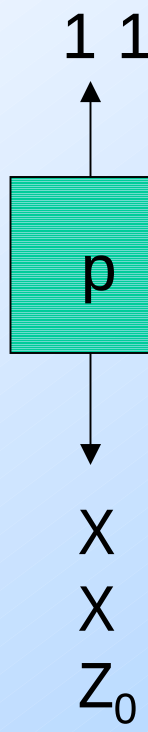
Actions of the Example PDA



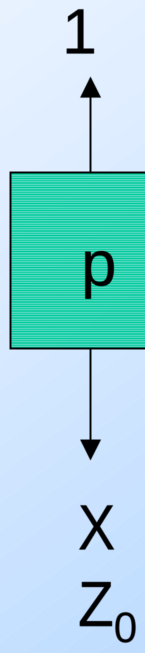
Actions of the Example PDA



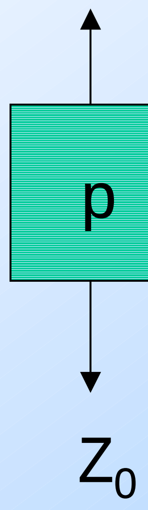
Actions of the Example PDA



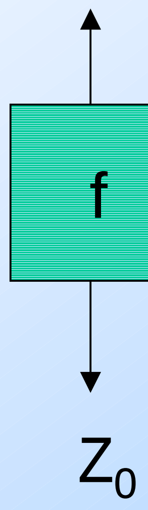
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Instantaneous Descriptions

- We can formalize the pictures just seen with an *instantaneous description* (ID).
- A ID is a triple (q, w, α), where:
  1. q is the current state.
  2. w is the remaining input.
  3. α is the stack contents, top at the left.

The “Goes-To” Relation

- To say that ID I can become ID J in one move of the PDA, we write $I \vdash J$.
- Formally, $(q, aw, X\alpha) \vdash (p, w, \beta\alpha)$ for any w and α, if $\delta(q, a, X)$ contains (p, β).
- Extend $\vdash$ to $\vdash^*$, meaning “zero or more moves,” by:
  - Basis: $I \vdash^* I$.
  - Induction: If $I \vdash^* J$ and $J \vdash K$, then $I \vdash^* K$.

Example: Goes-To

- Using the previous example PDA, we can describe the sequence of moves by: $(q, 000111, Z_0) \vdash (q, 00111, XZ_0) \vdash (q, 0111, XXZ_0) \vdash (q, 111, XXXZ_0) \vdash (p, 11, XXZ_0) \vdash (p, 1, XZ_0) \vdash (p, \epsilon, Z_0) \vdash (f, \epsilon, Z_0)$
- Thus, $(q, 000111, Z_0) \vdash^* (f, \epsilon, Z_0)$.
- What would happen on input 0001111?

Answer

- $(q, 0001111, Z_0) \vdash (q, 001111, XZ_0) \vdash (q, 01111, XXZ_0) \vdash (q, 1111, XXXZ_0) \vdash (p, 111, XXZ_0) \vdash (p, 11, XZ_0) \vdash (p, 1, Z_0) \vdash (f, 1, Z_0)$
- Note the last ID has no move.
- 0001111 is not accepted, because the input is not completely consumed.

Language of a PDA

- The common way to define the language of a PDA is by *final state*.
- If P is a PDA, then L(P) is the set of strings w such that $(q_0, w, Z_0) \vdash^* (f, \epsilon, \alpha)$ for final state f and any α.

Language of a PDA – (2)

- Another language defined by the same PDA is by *empty stack*.
- If P is a PDA, then N(P) is the set of strings w such that $(q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)$ for any state q.

Equivalence of Language Definitions

1. If L = L(P), then there is another PDA P' such that L = N(P').
2. If L = N(P), then there is another PDA P'' such that L = L(P'').

Proof: L(P) -> N(P') Intuition

- P' will simulate P.
- If P accepts, P' will empty its stack.
- P' has to avoid accidentally emptying its stack, so it uses a special bottom-marker to catch the case where P empties its stack without accepting.

Proof: L(P) -> N(P')

- P' has all the states, symbols, and moves of P, plus:
  1. Stack symbol X_0 (the start symbol of P'), used to guard the stack bottom.
  2. New start state s and "erase" state e.
  3. $\delta(s, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. Get P started.
  4. Add $\{(e, \epsilon)\}$ to $\delta(f, \epsilon, X)$ for any final state f of P and any stack symbol X, including X_0.
  5. $\delta(e, \epsilon, X) = \{(e, \epsilon)\}$ for any X.

Proof: N(P) -> L(P”) Intuition

- P” simulates P.
- P” has a special bottom-marker to catch the situation where P empties its stack.
- If so, P” accepts.

Proof: N(P) -> L(P'')

- P'' has all the states, symbols, and moves of P, plus:

1. Stack symbol X_0 (the start symbol), used to guard the stack bottom.
2. New start state s and final state f.
3. $\delta(s, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. Get P started.
4. $\delta(q, \epsilon, X_0) = \{(f, \epsilon)\}$ for any state q of P.

Deterministic PDA's

- To be deterministic, there must be at most one choice of move for any state q, input symbol *a*, and stack symbol X.
- In addition, there must not be a choice between using input ϵ or real input.
  - Formally, δ(q, a, X) and δ(q, ϵ, X) cannot both be nonempty.

Equivalence of PDA, CFG

Conversion of CFG to PDA

Conversion of PDA to CFG

Overview

- When we talked about closure properties of regular languages, it was useful to be able to jump between RE and DFA representations.
- Similarly, CFG's and PDA's are both useful to deal with properties of the CFL's.

Overview – (2)

- Also, PDA's, being "algorithmic," are often easier to use when arguing that a language is a CFL.
- Example: It is easy to see how a PDA can recognize balanced parentheses; not so easy as a grammar.

Converting a CFG to a PDA

- Let L = L(G).
- Construct PDA P such that N(P) = L.
- P has:
  - One state q.
  - Input symbols = terminals of G.
  - Stack symbols = all symbols of G.
  - Start symbol = start symbol of G.

Intuition About P

- At each step, P represents some *left-sentential form* (step of a leftmost derivation).
- If the stack of P is α, and P has so far consumed x from its input, then P represents left-sentential form xα.
- At empty stack, the input consumed is a string in L(G).

Transition Function of P

1. $\delta(q, a, a) = (q, \epsilon)$. (*Type 1* rules)
   - This step does not change the LSF represented, but “moves” responsibility for *a* from the stack to the consumed input.
2. If $A \rightarrow \alpha$ is a production of G, then $\delta(q, \epsilon, A)$ contains (q, α). (*Type 2* rules)
   - Guess a production for A, and represent the next LSF in the derivation.

Proof That L(P) = L(G)

- We need to show that $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{lm} w\alpha$.
- Part 1: “only if” is an induction on the number of steps made by P.
- Basis: 0 steps.
  - Then $\alpha = S$, $w = \epsilon$, and $S \Rightarrow^*_{lm} S$ is surely true.

Induction for Part 1

- Consider n moves of P: (q, wx, S) $\vdash$* (q, x, α) and assume the IH for sequences of n-1 moves.
- There are two cases, depending on whether the last move uses a Type 1 or Type 2 rule.

Use of a Type 1 Rule

- The move sequence must be of the form $(q, yax, S) \vdash^* (q, ax, a\alpha) \vdash (q, x, \alpha)$, where $ya = w$.
- By the IH applied to the first n-1 steps, $S \Rightarrow^*_{lm} ya\alpha$.
- But $ya = w$, so $S \Rightarrow^*_{lm} w\alpha$.

Use of a Type 2 Rule

- The move sequence must be of the form $(q, wx, S) \vdash^* (q, x, A\beta) \vdash (q, x, \gamma\beta)$, where $A \to \gamma$ is a production and $\alpha = \gamma\beta$.
- By the IH applied to the first n-1 steps, $S \Rightarrow^*_{lm} wA\beta$.
- Thus, $S \Rightarrow^*_{lm} w\gamma\beta = w\alpha$.

Proof of Part 2 (“if”)

- We also must prove that if $S \Rightarrow^*_{lm} w\alpha$, then $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x.
- Induction on number of steps in the leftmost derivation.
- Ideas are similar; omitted.

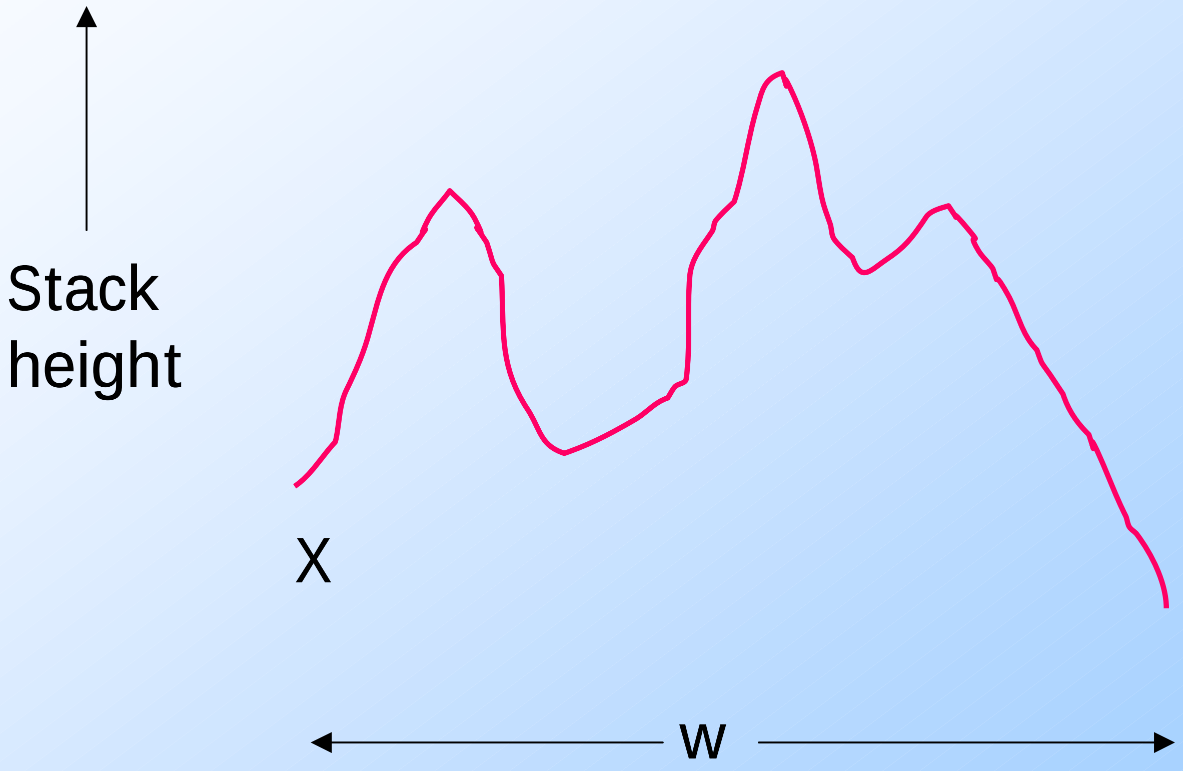
Proof – Completion

- We now have $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{lm} w\alpha$.
- In particular, let $x = \alpha = \epsilon$.
- Then $(q, w, S) \vdash^* (q, \epsilon, \epsilon)$ if and only if $S \Rightarrow^*_{lm} w$.
- That is, w is in N(P) if and only if w is in L(G).

From a PDA to a CFG

- Now, assume L = N(P).
- We'll construct a CFG G such that L = L(G).
- Intuition: G will have variables [pXq] generating exactly the inputs that cause P to have the net effect of popping stack symbol X while going from state p to state q.
  - P never gets below this X while doing so.

Picture: Popping X



Variables of G

- G's variables are of the form [pXq].
- This variable generates all and only the strings w such that
  $(p, w, X) \vdash^* (q, \epsilon, \epsilon)$.
- Also a start symbol S we'll talk about later.

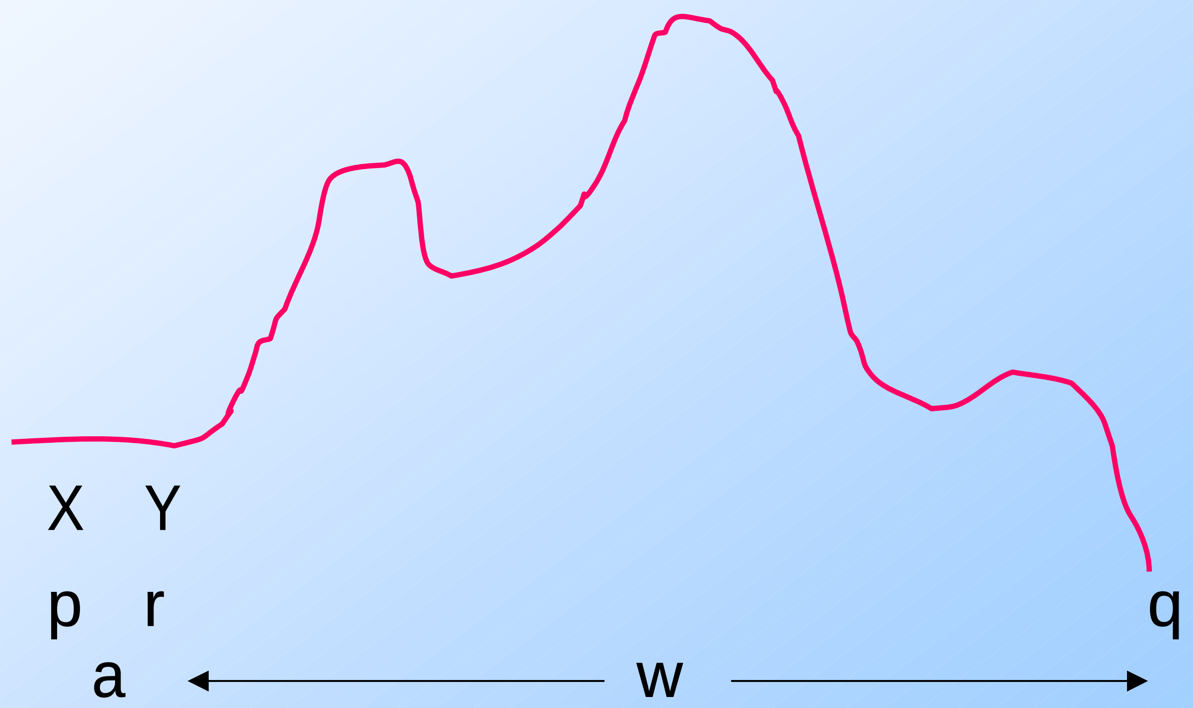
Productions of G

- Each production for [pXq] comes from a move of P in state p with stack symbol X.
- Simplest case: δ(p, a, X) contains (q, ϵ).
  - Note *a* can be an input symbol or ϵ.
- Then the production is [pXq] -> a.
- Here, [pXq] generates *a*, because reading *a* is one way to pop X and go from p to q.

Productions of G – (2)

- Next simplest case: δ(p, a, X) contains (r, Y) for some state r and symbol Y.
- G has production [pXq] -> a[rYq].
  - We can erase X and go from p to q by reading *a* (entering state r and replacing the X by Y) and then reading some w that gets P from r to q while erasing the Y.

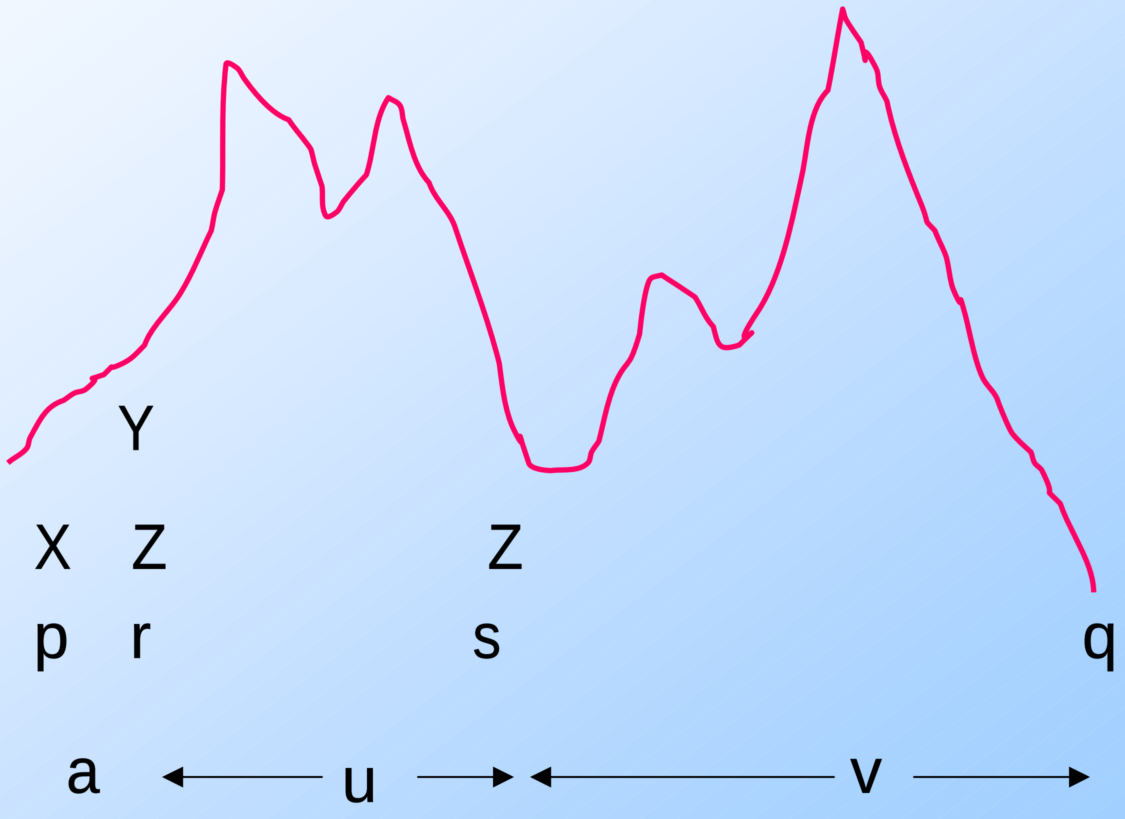
Picture of the Action



Productions of G – (3)

- Third simplest case: δ(p, a, X) contains (r, YZ) for some state r and symbols Y and Z.
- Now, P has replaced X by YZ.
- To have the net effect of erasing X, P must erase Y, going from state r to some state s, and then erase Z, going from s to q.

Picture of the Action



Third-Simplest Case – Concluded

- Since we do not know state s, we must generate a family of productions:

  [pXq] -> a[rYs][sZq]

  for all states s.
- [pXq] =>* auv whenever [rYs] =>* u and [sZq] =>* v.

Productions of G: General Case

- Suppose δ(p, a, X) contains (r, $Y_1,\ldots Y_k$) for some state r and k $\geq$ 3.
- Generate family of productions

[pXq] ->

$$a[rY_1s_1][s_1Y_2s_2]\ldots[s_{k-2}Y_{k-1}s_{k-1}][s_{k-1}Y_kq]$$

Completion of the Construction

- We can prove that $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$ if and only if $[q_0Z_0p] \Rightarrow^* w$.
  - Proof is two easy inductions.
- But state p can be anything.
- Thus, add to G another variable S, the start symbol, and add productions S -> $[q_0Z_0p]$ for each state p.

The Pumping Lemma for CFL’s

Statement

Applications

Intuition

- Recall the pumping lemma for regular languages.
- It told us that if there was a string long enough to cause a cycle in the DFA for the language, then we could “pump” the cycle and discover an infinite sequence of strings that had to be in the language.

Intuition – (2)

- For CFL's the situation is a little more complicated.
- We can always find two pieces of any sufficiently long string to "pump" in tandem.
  - That is: if we repeat each of the two pieces the same number of times, we get another string in the language.

Statement of the CFL Pumping Lemma

For every context-free language L

There is an integer n, such that

For every string z in L of length $\geq$ n

There exists z = uvwxy such that:

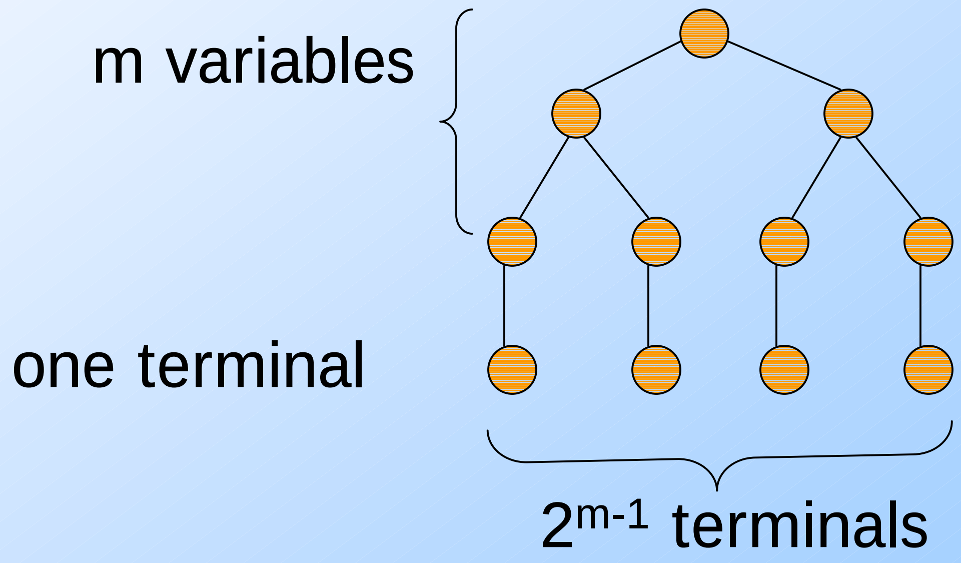
1. $|vwx| \leq n$.
2. $|vx| > 0$.
3. For all $i \geq 0$, uv^iwx^iy is in L.

Proof of the Pumping Lemma

- Start with a CNF grammar for L – {ϵ}.
- Let the grammar have m variables.
- Pick n = 2^m.
- Let z, of length $\geq$ n, be in L.
- We claim ("*Lemma 1* ") that a parse tree with yield z must have a path of length m+2 or more.

Proof of Lemma 1

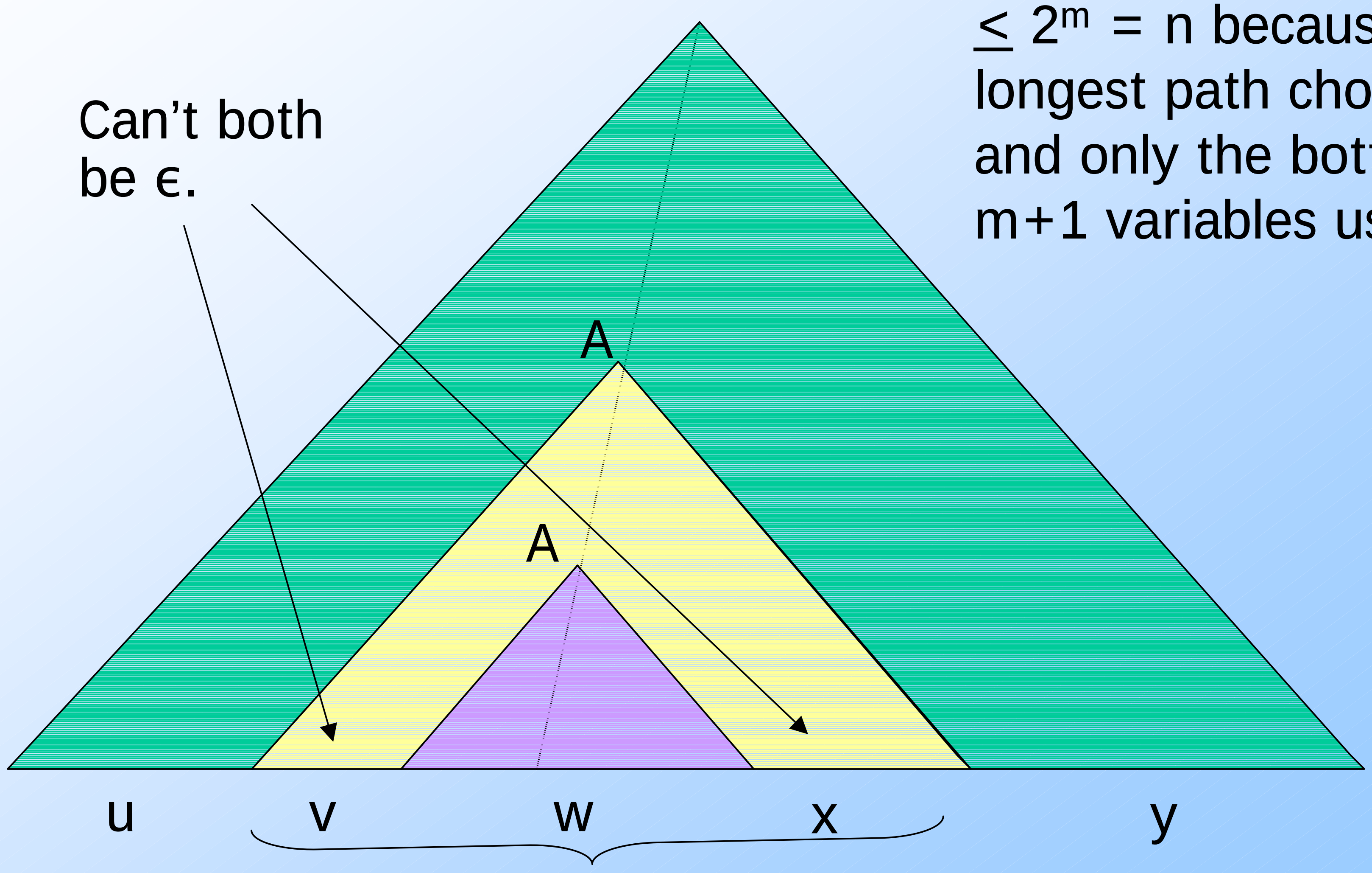
- If all paths in the parse tree of a CNF grammar are of length $\leq$ m+1, then the longest yield has length 2^{m-1}, as in:



Back to the Proof of the Pumping Lemma

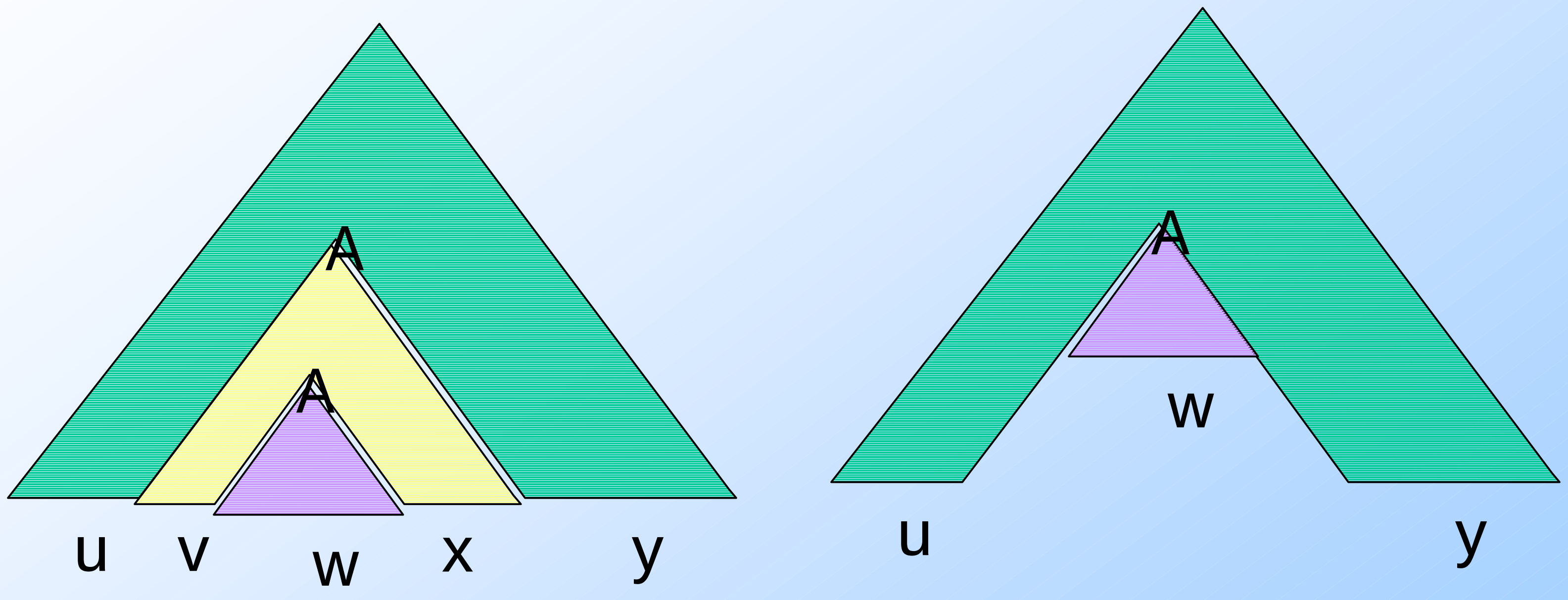
- Now we know that the parse tree for z has a path with at least m+1 variables.
- Consider some longest path.
- There are only m different variables, so among the lowest m+1 we can find two nodes with the same label, say A.
- The parse tree thus looks like:

Parse Tree in the Pumping-Lemma Proof

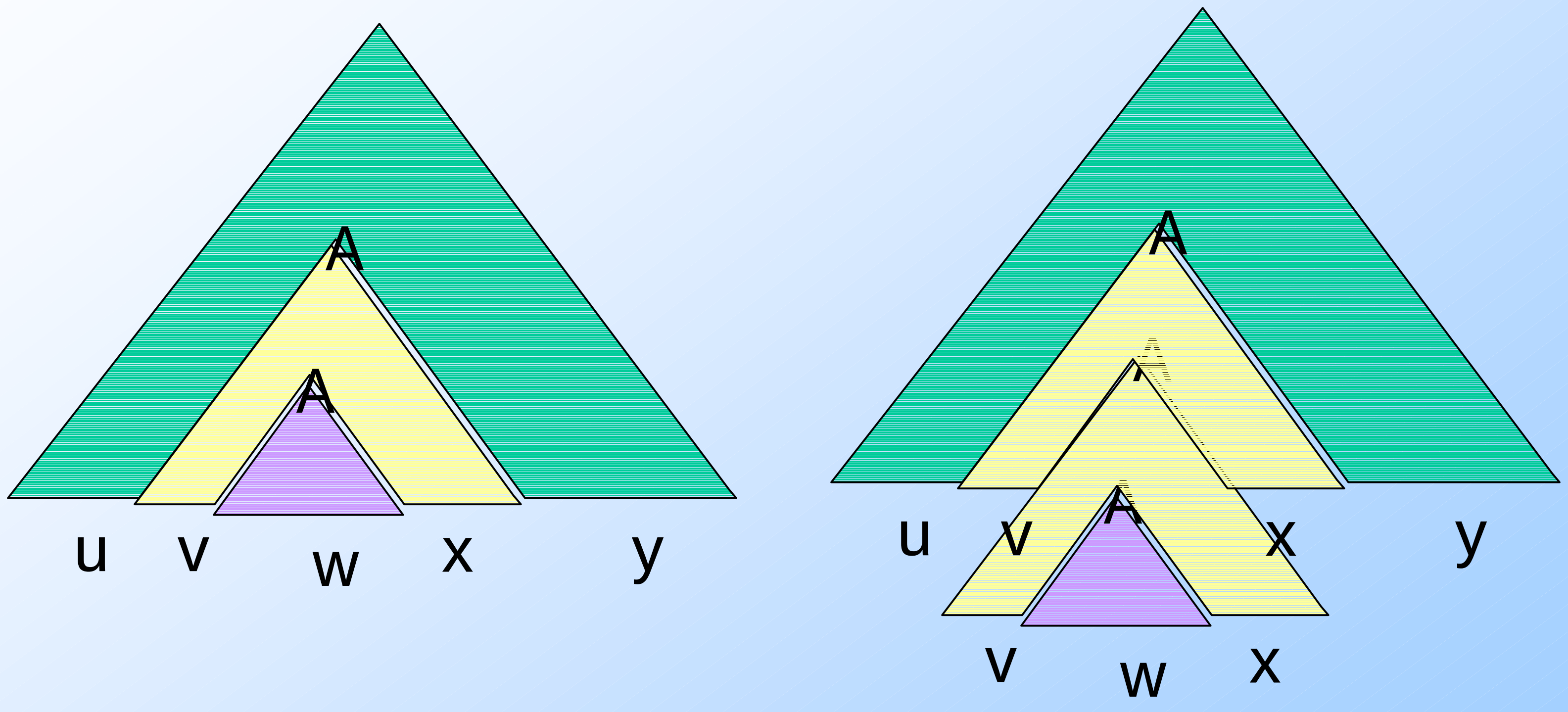


$\leq 2^m = n$ because a longest path chosen and only the bottom m+1 variables used.

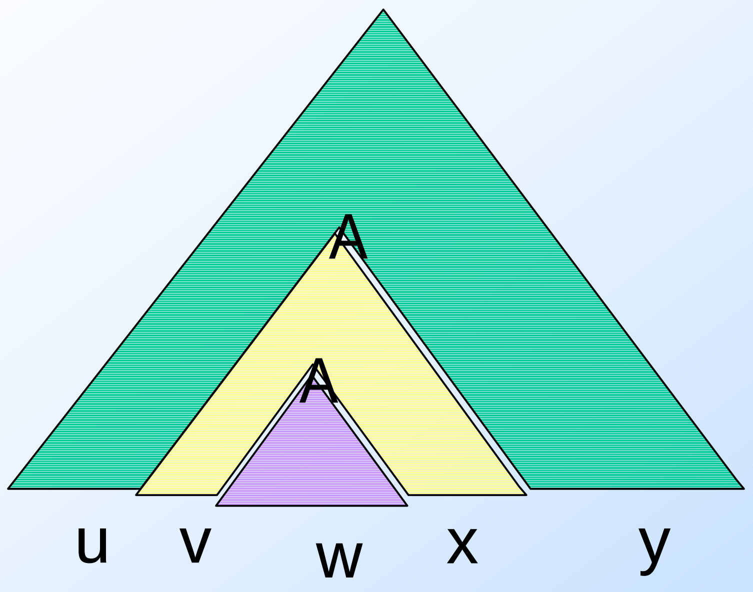
Pump Zero Times

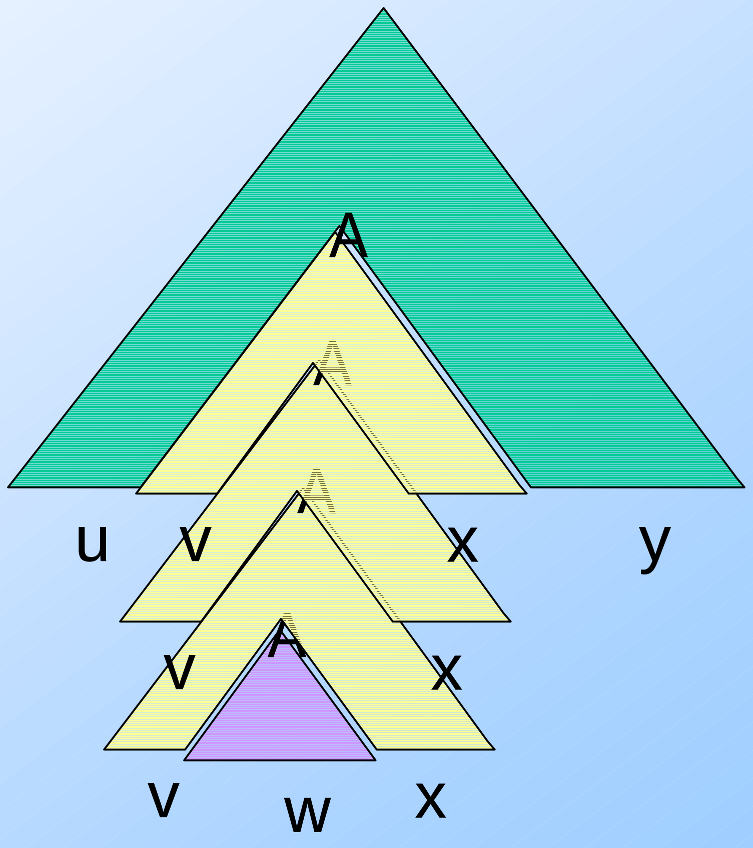


Pump Twice



Pump Thrice Etc., Etc.





Using the Pumping Lemma

- $\{0^i10^i \mid i \geq 1\}$ is a CFL.
  - We can match one pair of counts.
- But $L = \{0^i10^i10^i \mid i \geq 1\}$ is not.
  - We can't match two pairs, or three counts as a group.
- Proof using the pumping lemma.
- Suppose L were a CFL.
- Let n be L's pumping-lemma constant.

Using the Pumping Lemma – (2)

- Consider $z = 0^n10^n10^n$.
- We can write $z = uvwxy$, where $|vwx| \leq n$, and $|vx| \geq 1$.
- Case 1: vx has no 0's.
  - Then at least one of them is a 1, and uwy has at most one 1, which no string in L does.

Using the Pumping Lemma – (3)

- Still considering $z = 0^n10^n10^n$.
- Case 2: vx has at least one 0.
  - vwx is too short (length $\leq$ n) to extend to all three blocks of 0's in $0^n10^n10^n$.
  - Thus, uwy has at least one block of n 0's, and at least one block with fewer than n 0's.
  - Thus, uwy is not in L.

Properties of Context-Free Languages

Decision Properties

Closure Properties

Summary of Decision Properties

- As usual, when we talk about “a CFL” we really mean “a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.
- There are algorithms to decide if:

1. String w is in CFL L.
2. CFL L is empty.
3. CFL L is infinite.

Non-Decision Properties

- Many questions that can be decided for regular sets cannot be decided for CFL’s.
- Example: Are two CFL’s the same?
- Example: Are two CFL’s disjoint?
  - How would you do that for regular languages?
- Need theory of Turing machines and decidability to prove no algorithm exists.

Testing Emptiness

- We already did this.
- We learned to eliminate useless variables.
- If the start symbol is one of these, then the CFL is empty; otherwise not.

Testing Membership

- Want to know if string w is in L(G).
- Assume G is in CNF.
  - Or convert the given grammar to CNF.
  - w = ϵ is a special case, solved by testing if the start symbol is nullable.
- Algorithm (*CYK*) is a good example of dynamic programming and runs in time O(n^3), where n = |w|.

CYK Algorithm

- Let $w = a_1 \ldots a_n$.
- We construct an n-by-n triangular array of sets of variables.
- X_{ij} = {variables A | A =>* $a_i \ldots a_j$}.
- Induction on $j–i+1$.
  - The length of the derived string.
- Finally, ask if S is in X_{1n}.

CYK Algorithm – (2)

- Basis: X_{ii} = {A | A -> a_i is a production}.
- Induction: X_{ij} = {A | there is a production A -> BC and an integer k, with $i \leq k < j$, such that B is in X_{ik} and C is in $X_{k+1,j}$.

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

String w = ababa

$$\begin{tikzcd}
X_{12}=\{B,S\} & X_{23}=\{A\} & X_{34}=\{B,S\} & X_{45}=\{A\} & \\
X_{11}=\{A,C\} \arrow[u] \arrow[u, bend right] & X_{22}=\{B,C\} \arrow[ul] \arrow[ul, bend left] & X_{33}=\{A,C\} & X_{44}=\{B,C\} & X_{55}=\{A,C\}
\end{tikzcd}$$

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

String w = ababa

$$\begin{tikzcd}
X_{13}=\{\} & & \text{Yields nothing} & & \\
X_{12}=\{B,S\} & X_{23}=\{A\} \arrow[ul] & X_{34}=\{B,S\} & X_{45}=\{A\} & \\
X_{11}=\{A,C\} \arrow[uu] & X_{22}=\{B,C\} & X_{33}=\{A,C\} & X_{44}=\{B,C\} & X_{55}=\{A,C\}
\end{tikzcd}$$

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

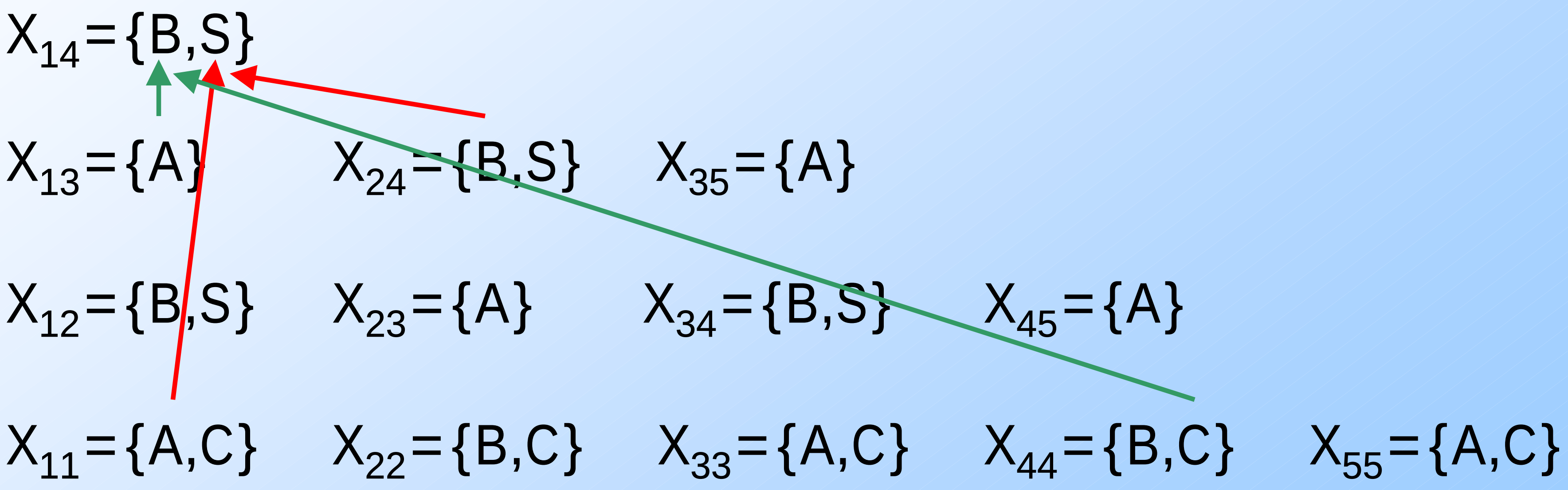
String w = ababa

$$\begin{tikzcd}
X_{13}=\{A\} & X_{24}=\{B,S\} & X_{35}=\{A\} & & \\
X_{12}=\{B,S\} \arrow[u] & X_{23}=\{A\} & X_{34}=\{B,S\} & X_{45}=\{A\} & \\
X_{11}=\{A,C\} & X_{22}=\{B,C\} & X_{33}=\{A,C\} \arrow[uull] & X_{44}=\{B,C\} & X_{55}=\{A,C\}
\end{tikzcd}$$

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

String w = ababa



X_{14}={B,S}

X_{13}={A} X_{24}={B,S} X_{35}={A}

X_{12}={B,S} X_{23}={A} X_{34}={B,S} X_{45}={A}

X_{11}={A,C} X_{22}={B,C} X_{33}={A,C} X_{44}={B,C} X_{55}={A,C}

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b

String w = ababa

$$\begin{tikzcd}
X_{15}=\{A\} \\
X_{14}=\{B,S\} \arrow[u] & X_{25}=\{A\} \\
X_{13}=\{A\} & X_{24}=\{B,S\} & X_{35}=\{A\} \\
X_{12}=\{B,S\} & X_{23}=\{A\} & X_{34}=\{B,S\} & X_{45}=\{A\} \\
X_{11}=\{A,C\} & X_{22}=\{B,C\} & X_{33}=\{A,C\} & X_{44}=\{B,C\} & X_{55}=\{A,C\} \arrow[uuuullll]
\end{tikzcd}$$

Testing Infiniteness

- The idea is essentially the same as for regular languages.
- Use the pumping lemma constant n.
- If there is a string in the language of length between n and 2n-1, then the language is infinite; otherwise not.

Closure Properties of CFL's

- CFL's are closed under union, concatenation, and Kleene closure.
- Also, under reversal, homomorphisms and inverse homomorphisms.
- But not under intersection or difference.

Closure of CFL's Under Union

- Let L and M be CFL's with grammars G and H, respectively.
- Assume G and H have no variables in common.
  - Names of variables do not affect the language.
- Let S_1 and S_2 be the start symbols of G and H.

Closure Under Union – (2)

- Form a new grammar for L $\cup$ M by combining all the symbols and productions of G and H.
- Then, add a new start symbol S.
- Add productions S -> S_1 | S_2.

Closure Under Union – (3)

- In the new grammar, all derivations start with S.
- The first step replaces S by either S_1 or S_2.
- In the first case, the result must be a string in L(G) = L, and in the second case a string in L(H) = M.

Closure of CFL's Under Concatenation

- Let L and M be CFL's with grammars G and H, respectively.
- Assume G and H have no variables in common.
- Let S_1 and S_2 be the start symbols of G and H.

Closure Under Concatenation – (2)

- Form a new grammar for LM by starting with all symbols and productions of G and H.
- Add a new start symbol S.
- Add production S -> S_1S_2.
- Every derivation from S results in a string in L followed by one in M.

Closure Under Star

- Let L have grammar G, with start symbol S_1.
- Form a new grammar for L* by introducing to G a new start symbol S and the productions S -> S_1S | ϵ.
- A rightmost derivation from S generates a sequence of zero or more S_1's, each of which generates some string in L.

Closure of CFL's Under Reversal

- If L is a CFL with grammar G, form a grammar for L^R by reversing the body of every production.
- Example: Let G have S -> 0S1 | 01.
- The reversal of L(G) has grammar S -> 1S0 | 10.

Closure of CFL’s Under Homomorphism

- Let L be a CFL with grammar G.
- Let h be a homomorphism on the terminal symbols of G.
- Construct a grammar for h(L) by replacing each terminal symbol *a* by h(a).

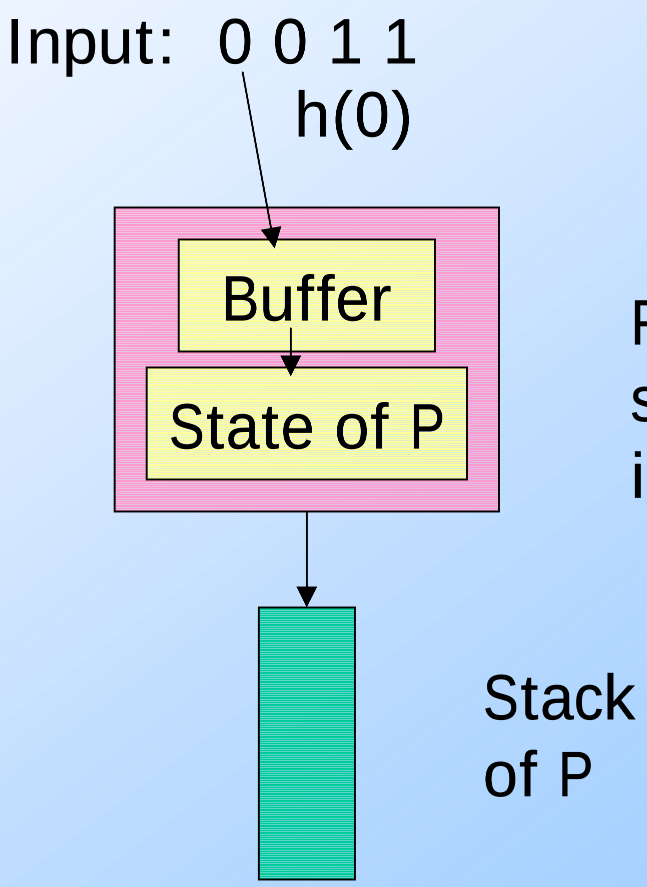
Example: Closure Under Homomorphism

- G has productions S -> 0S1 | 01.
- h is defined by h(0) = ab, h(1) = ϵ.
- h(L(G)) has the grammar with productions S -> abS | ab.

Closure of CFL's Under Inverse Homomorphism

- Here, grammars don't help us, but a PDA construction serves nicely.
- Let L = L(P) for some PDA P.
- Construct PDA P' to accept $h^{-1}(L)$.
- P' simulates P, but keeps, as one component of a two-component state a buffer that holds the result of applying h to one input symbol.

Architecture of P’



Read first remaining symbol in buffer as if it were input to P.

Formal Construction of P'

- States are pairs [q, w], where:
  1. q is a state of P.
  2. w is a suffix of h(a) for some symbol *a*.
     - Thus, only a finite number of possible values for w.
- Stack symbols of P' are those of P.
- Start state of P' is $[q_0, \epsilon]$.

Construction of P' – (2)

- Input symbols of P' are the symbols to which h applies.
- Final states of P' are the states [q, ϵ] such that q is a final state of P.

Transitions of P'

1. $\delta'([q, \epsilon], a, X) = \{([q, h(a)], X)\}$ for any input symbol *a* of P' and any stack symbol X.
   - When the buffer is empty, P' can reload it.
2. $\delta'([q, bw], \epsilon, X)$ contains $([p, w], \alpha)$ if $\delta(q, b, X)$ contains (p, α), where b is either an input symbol of P or ϵ.
   - Simulate P from the buffer.

Proving Correctness of P'

- We need to show that L(P') = h^{-1}(L(P)).
- Key argument: P' makes the transition $([q_0, \epsilon], w, Z_0) \vdash^* ([q, x], \epsilon, \alpha)$ if and only if P makes transition $(q_0, y, Z_0) \vdash^* (q, \epsilon, \alpha)$, h(w) = yx, and x is a suffix of the last symbol of w.
- Proof in both directions is an induction on the number of moves made.

Nonclosure Under Intersection

- Unlike the regular languages, the class of CFL's is not closed under $\cap$.
- We know that $L_1 = \{0^n1^n2^n \mid n \geq 1\}$ is not a CFL (use the pumping lemma).
- However, $L_2 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$ is.
  - CFG: S -> AB, A -> 0A1 | 01, B -> 2B | 2.
- So is $L_3 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$.
- But $L_1 = L_2 \cap L_3$.

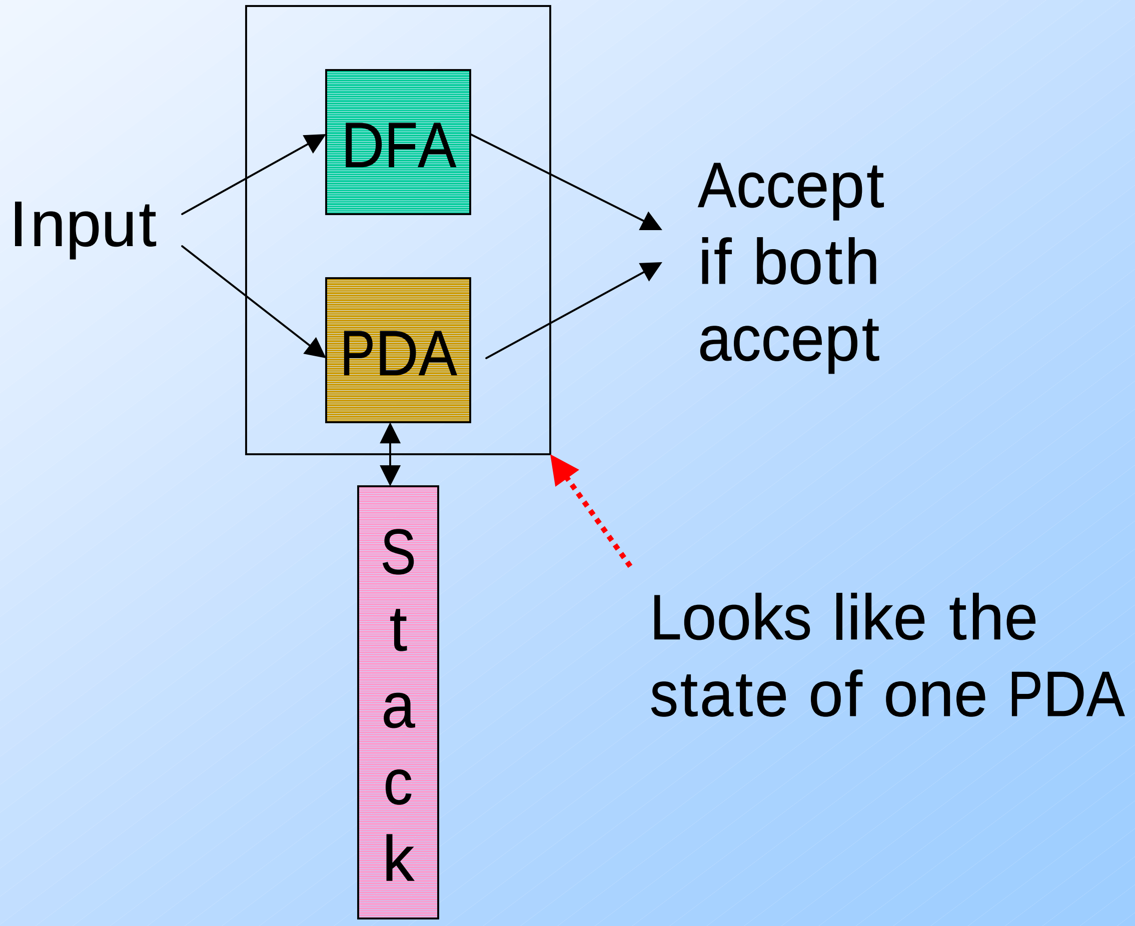
Nonclosure Under Difference

- We can prove something more general:
  - Any class of languages that is closed under difference is closed under intersection.
- Proof: $L \cap M = L - (L - M)$.
- Thus, if CFL's were closed under difference, they would be closed under intersection, but they are not.

Intersection with a Regular Language

- Intersection of two CFL's need not be context free.
- But the intersection of a CFL with a regular language is always a CFL.
- Proof involves running a DFA in parallel with a PDA, and noting that the combination is a PDA.
  - PDA's accept by final state.

DFA and PDA in Parallel



Formal Construction

- Let the DFA A have transition function δ_A.
- Let the PDA P have transition function δ_P.
- States of combined PDA are [q,p], where q is a state of A and p a state of P.
- $\delta([q,p], a, X)$ contains $([\delta_A(q,a),r], \alpha)$ if $\delta_P(p, a, X)$ contains (r, α).
  - Note a could be ε, in which case $\delta_A(q,a) = q$.

Formal Construction – (2)

- Final states of combined PDA are those [q,p] such that q is a final state of A and p is an accepting state of P.
- Initial state is the pair ($[q_0,p_0]$ consisting of the initial states of each.
- Easy induction: $([q_0,p_0], w, Z_0) \vdash^* ([q,p], \varepsilon, \alpha)$ if and only if $\delta_A(q_0,w) = q$ and in P: $(p_0, w, Z_0) \vdash^* (p, \varepsilon, \alpha)$.

Undecidability

Everything is an Integer

Countable and Uncountable Sets

Turing Machines

Recursive and Recursively Enumerable Languages

Integers, Strings, and Other Things

- Data types have become very important as a programming tool.
- But at another level, there is only one type, which you may think of as integers or strings.
- Key point: Strings that are programs are just another way to think about the same one data type.

Example: Text

- Strings of ASCII or Unicode characters can be thought of as binary strings, with 8 or 16 bits/character.
- Binary strings can be thought of as integers.
- It makes sense to talk about “the i-th string.”

Binary Strings to Integers

- There's a small glitch:
  - If you think simply of binary integers, then strings like 101, 0101, 00101,... all appear to be "the fifth string."
- Fix by prepending a "1" to the string before converting to an integer.
  - Thus, 101, 0101, and 00101 are the 13th, 21st, and 37th strings, respectively.

Example: Images

- Represent an image in (say) GIF.
- The GIF file is an ASCII string.
- Convert string to binary.
- Convert binary string to integer.
- Now we have a notion of “the i-th image.”

Example: Proofs

- A formal proof is a sequence of logical expressions, each of which follows from the ones before it.
- Encode mathematical expressions of any kind in Unicode.
- Convert expression to a binary string and then an integer.

Proofs – (2)

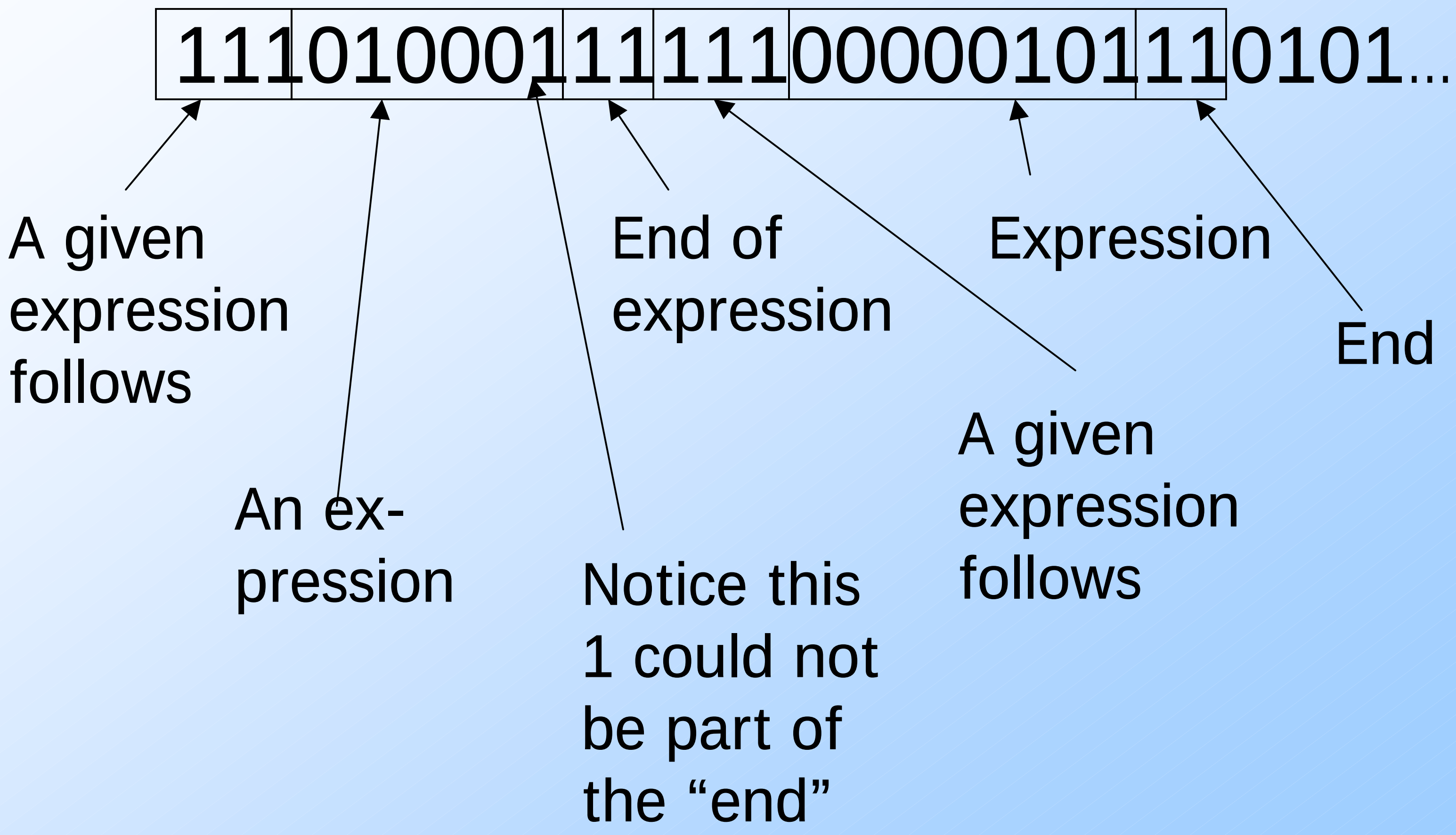
- But a proof is a sequence of expressions, so we need a way to separate them.
- Also, we need to indicate which expressions are given and which follow from previous expressions.

Proofs – (3)

- Quick-and-dirty way to introduce new symbols into binary strings:

1. Given a binary string, precede each bit by 0.
   - Example: 101 becomes 010001.
2. Use strings of two or more 1's as the special symbols.
   - Example: 111 = "the following expression is given"; 11 = "end of expression."

Example: Encoding Proofs



111010001111111000000101110101…
A given expression follows
An ex-pression
Notice this 1 could not be part of the "end"
End of expression
A given expression follows
Expression
End

Example: Programs

- Programs are just another kind of data.
- Represent a program in ASCII.
- Convert to a binary string, then to an integer.
- Thus, it makes sense to talk about "the i-th program."
- Hmm...There aren't all that many programs.

Finite Sets

- A *finite set* has a particular integer that is the count of the number of members.
- Example: {a, b, c} is a finite set; its *cardinality* is 3.
- It is impossible to find a 1-1 mapping between a finite set and a proper subset of itself.

Infinite Sets

- Formally, an *infinite set* is a set for which there is a 1-1 correspondence between itself and a proper subset of itself.
- Example: the positive integers {1, 2, 3,...} is an infinite set.
  - There is a 1-1 correspondence 1<->2, 2<->4, 3<->6,... between this set and a proper subset (the set of even integers).

Countable Sets

- A *countable set* is a set with a 1-1 correspondence with the positive integers.
  - Hence, all countable sets are infinite.
- Example: All integers.
  - 0<->1; -i <-> 2i; +i <-> 2i+1.
  - Thus, order is 0, -1, 1, -2, 2, -3, 3,...
- Examples: set of binary strings, set of Java programs.

Example: Pairs of Integers

- Order the pairs of positive integers first by sum, then by first component:
- [1,1], [2,1], [1,2], [3,1], [2,2], [1,3], [4,1], [3,2],..., [1,4], [5,1],...
- Interesting exercise: figure out the function f(i,j) such that the pair [i,j] corresponds to the integer f(i,j) in this order.

Enumerations

- An *enumeration* of a set is a 1-1 correspondence between the set and the positive integers.
- Thus, we have seen enumerations for strings, programs, proofs, and pairs of integers.

How Many Languages?

- Are the languages over {0,1} countable?
- No; here’s a proof.
- Suppose we could enumerate all languages over {0,1} and talk about “the i-th language.”
- Consider the language L = { w | w is the i-th binary string and w is not in the i-th language}.

Proof – Continued

L_j

- Clearly, L is a language over {0,1}.
- Thus, it is the j-th language for some particular j.
- Let x be the j-th string.
- Is x in L?
  - If so, x is not in L by definition of L.
  - If not, then x is in L by definition of L.

x

Recall: L = { w | w is the i-th binary string and w is not in the i-th language}.

j-th

Proof – Concluded

- We have a contradiction: x is neither in L nor not in L, so our sole assumption (that there was an enumeration of the languages) is wrong.
- Comment: This is really bad; there are more languages than programs.
- E.g., there are languages with no membership algorithm.

Diagonalization Picture

Strings

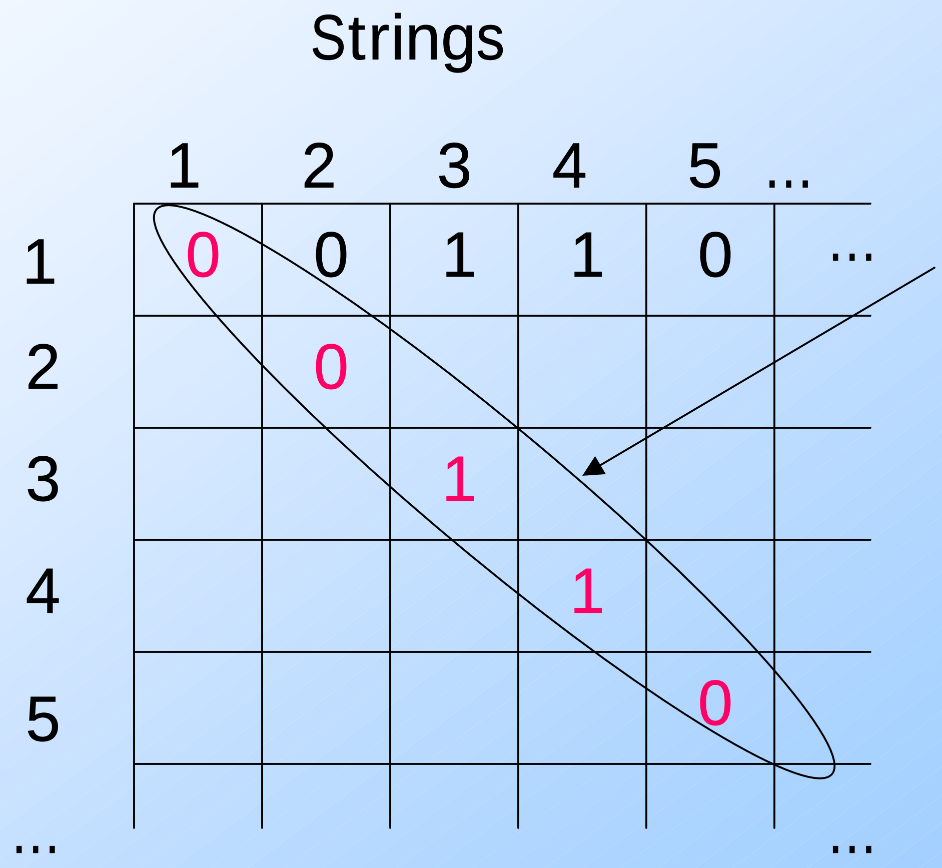
Languages

	1	2	3	4	5	...
1	1	0	1	1	0	...
2		1				
3			0			
4				0		
5					1	
...						...

Diagonalization Picture

Flip each diagonal entry

Languages

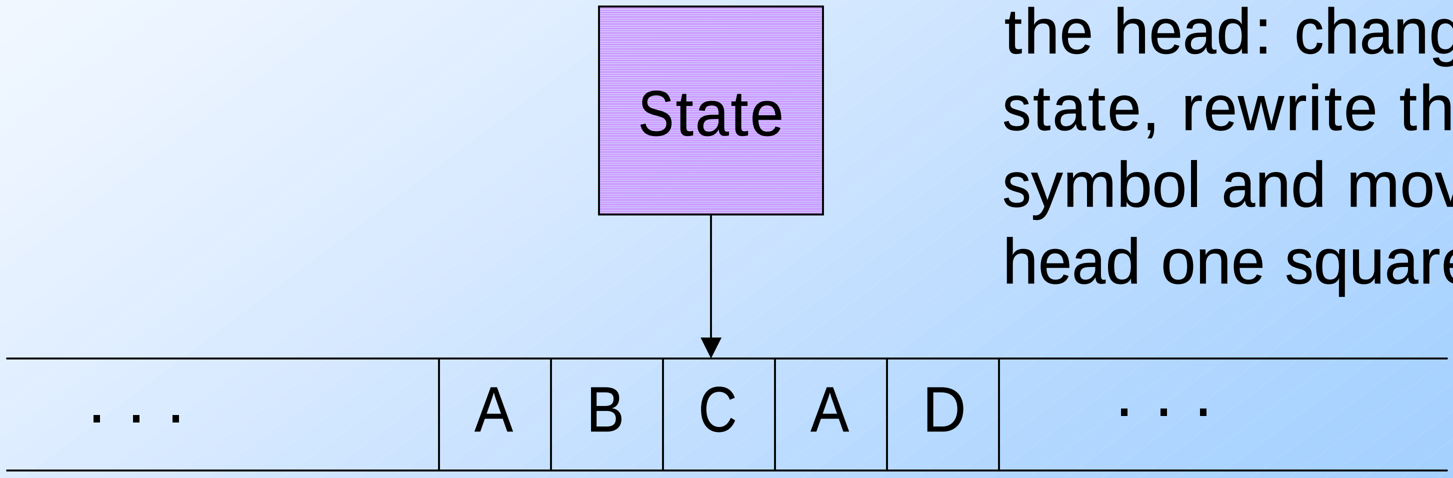


Can’t be a row – it disagrees in an entry of each row.

Turing-Machine Theory

- The purpose of the theory of Turing machines is to prove that certain specific languages have no algorithm.
- Start with a language about Turing machines themselves.
- Reductions are used to prove more common questions undecidable.

Picture of a Turing Machine



Action: based on the state and the tape symbol under the head: change state, rewrite the symbol and move the head one square.

Infinite tape with squares containing tape symbols chosen from a finite alphabet

Why Turing Machines?

- Why not deal with C programs or something like that?
- Answer: You can, but it is easier to prove things about TM's, because they are so simple.
  - And yet they are as powerful as any computer.
    - More so, in fact, since they have infinite memory.

Turing-Machine Formalism

- A TM is described by:
  1. A finite set of *states* (Q, typically).
  2. An *input alphabet* (Σ, typically).
  3. A *tape alphabet* (Γ, typically; contains Σ).
  4. A *transition function* (δ, typically).
  5. A *start state* (q_0, in Q, typically).
  6. A *blank symbol* (B, in Γ– Σ, typically).
     - All tape except for the input is blank initially.
  7. A set of *final states* ($F \subseteq Q$, typically).

Conventions

- a, b, ... are input symbols.
- ..., X, Y, Z are tape symbols.
- ..., w, x, y, z are strings of input symbols.
- α, β,... are strings of tape symbols.

The Transition Function

- Takes two arguments:
  1. A state, in Q.
  2. A tape symbol in Γ.
- δ(q, Z) is either undefined or a triple of the form (p, Y, D).
  - p is a state.
  - Y is the new tape symbol.
  - D is a *direction*, L or R.

Example: Turing Machine

- This TM scans its input right, looking for a 1.
- If it finds one, it changes it to a 0, goes to final state f, and halts.
- If it reaches a blank, it changes it to a 1 and moves left.

Example: Turing Machine – (2)

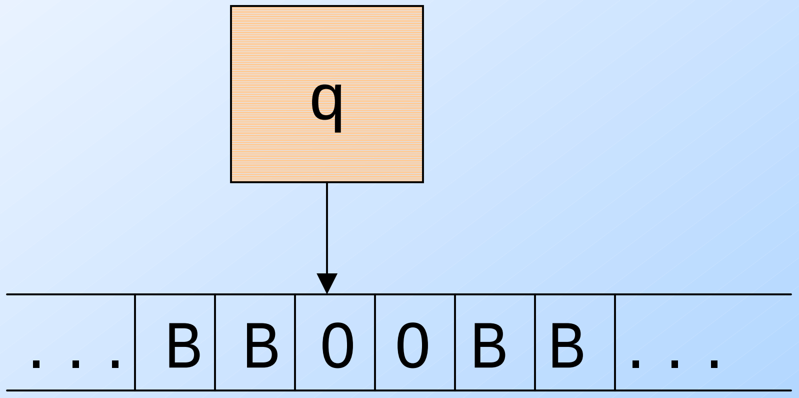
- States = {q (start), f (final)}.
- Input symbols = {0, 1}.
- Tape symbols = {0, 1, B}.
- $\delta(q, 0) = (q, 0, R)$.
- $\delta(q, 1) = (f, 0, R)$.
- $\delta(q, B) = (q, 1, L)$.

Simulation of TM

δ(q, 0) = (q, 0, R)

δ(q, 1) = (f, 0, R)

δ(q, B) = (q, 1, L)

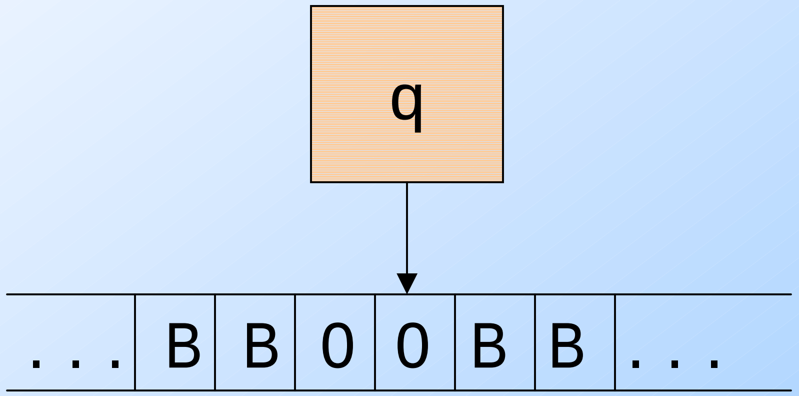


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$

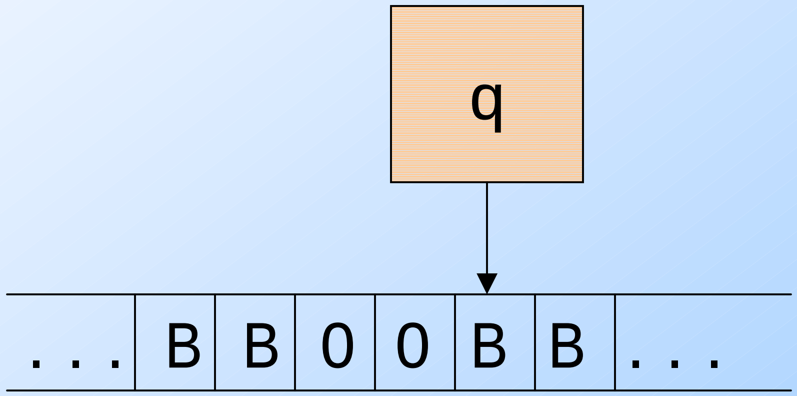


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$

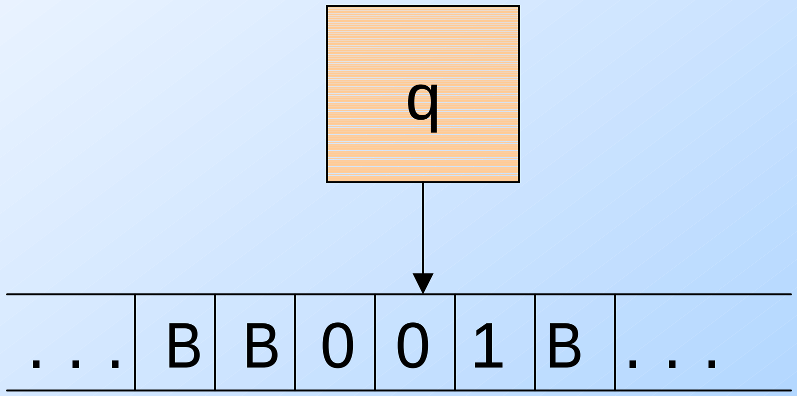


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$

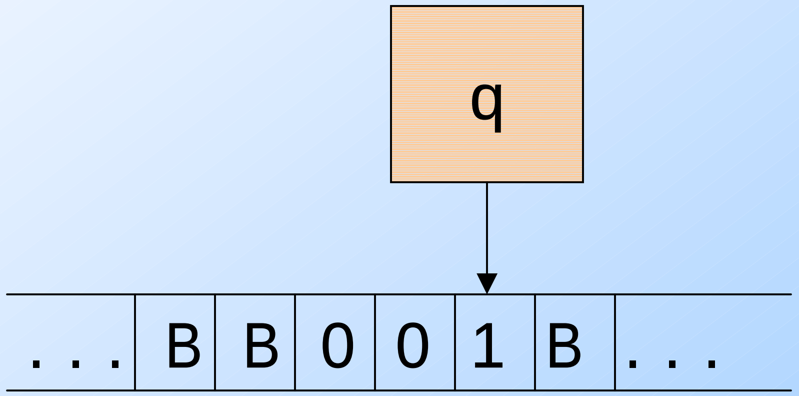


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$

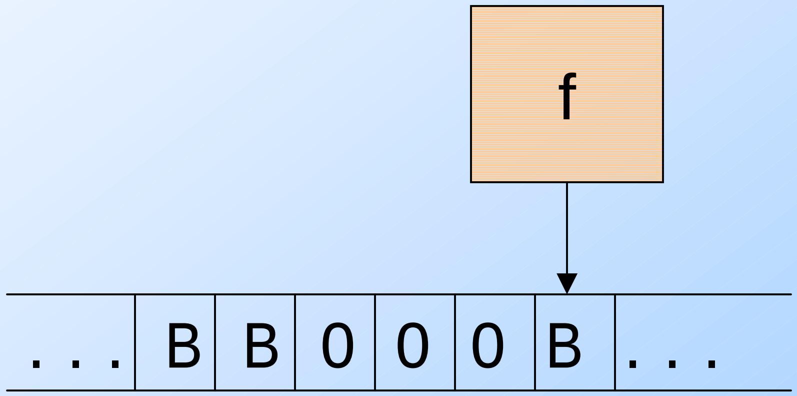


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$



No move is possible. The TM halts and accepts.

Instantaneous Descriptions of a Turing Machine

- Initially, a TM has a tape consisting of a string of input symbols surrounded by an infinity of blanks in both directions.
- The TM is in the start state, and the head is at the leftmost input symbol.

TM ID's – (2)

- An ID is a string $\alpha q\beta$, where $\alpha\beta$ includes the tape between the leftmost and rightmost nonblanks.
- The state q is immediately to the left of the tape symbol scanned.
- If q is at the right end, it is scanning B.
  - If q is scanning a B at the left end, then consecutive B's at and to the right of q are part of α.

TM ID's – (3)

- As for PDA's we may use symbols $\vdash$ and $\vdash^*$ to represent "becomes in one move" and "becomes in zero or more moves," respectively, on ID's.
- Example: The moves of the previous TM are $q00 \vdash 0q0 \vdash 00q \vdash 0q01 \vdash 00q1 \vdash 000f$

Formal Definition of Moves

1. If $\delta(q, Z) = (p, Y, R)$, then
   - $\alpha qZ\beta \vdash \alpha Yp\beta$
   - If Z is the blank B, then also $\alpha q \vdash \alpha Yp$
2. If $\delta(q, Z) = (p, Y, L)$, then
   - For any X, $\alpha XqZ\beta \vdash \alpha pXY\beta$
   - In addition, $qZ\beta \vdash pBY\beta$

Languages of a TM

- A TM defines a language by final state, as usual.
- $L(M) = \{w \mid q_0w \vdash^* I$, where I is an ID with a final state$\}$.
- Or, a TM can accept a language by halting.
- $H(M) = \{w \mid q_0w \vdash^* I$, and there is no move possible from ID $I\}$.

Equivalence of Accepting and Halting

1. If L = L(M), then there is a TM M' such that L = H(M').
2. If L = H(M), then there is a TM M" such that L = L(M").

Proof of 1: Final State -> Halting

- Modify M to become M' as follows:
  1. For each final state of M, remove any moves, so M' halts in that state.
  2. Avoid having M' accidentally halt.
     - Introduce a new state s, which runs to the right forever; that is $\delta(s, X) = (s, X, R)$ for all symbols X.
     - If q is not a final state, and $\delta(q, X)$ is undefined, let $\delta(q, X) = (s, X, R)$.

Proof of 2: Halting -> Final State

- Modify M to become M” as follows:
  1. Introduce a new state f, the only final state of M”.
  2. f has no moves.
  3. If $\delta(q, X)$ is undefined for any state q and symbol X, define it by $\delta(q, X) = (f, X, R)$.

Recursively Enumerable Languages

- We now see that the classes of languages defined by TM's using final state and halting are the same.
- This class of languages is called the *recursively enumerable languages*.
  - Why? The term actually predates the Turing machine and refers to another notion of computation of functions.

Recursive Languages

- An *algorithm* is a TM, accepting by final state, that is guaranteed to halt whether or not it accepts.
- If L = L(M) for some TM M that is an algorithm, we say L is a *recursive language*.
  - Why? Again, don't ask; it is a term with a history.

Example: Recursive Languages

- Every CFL is a recursive language.
  - Use the CYK algorithm.
- Almost anything you can think of is recursive.

Intractable Problems

Time-Bounded Turing Machines

Classes **P** and **NP**

Polynomial-Time Reductions

Time-Bounded TM's

- A Turing machine that, given an input of length n, always halts within T(n) moves is said to be *T(n)-time bounded*.
  - The TM can be multitape.
  - Sometimes, it can be nondeterministic.

The class **P**

- If a DTM M is T(n)-time bounded for some polynomial T(n), then we say M is *polynomial-time* ("*polytime*") bounded.
- And L(M) is said to be in the class **P**.
- Important point: when we talk of **P**, it doesn't matter whether we mean "by a computer" or "by a TM" (next slide).

Polynomial Equivalence of Computers and TM's

- A multitape TM can simulate a computer that runs for time O(T(n)) in at most O(T^2(n)) of its own steps.
- If T(n) is a polynomial, so is T^2(n).

Examples of Problems in **P**

- Is w in L(G), for a given CFG G?
  - Input = w.
  - Use CYK algorithm, which is O(n^3).
- Is there a path from node x to node y in graph G?
  - Input = x, y, and G.
  - Use depth-first search, which is O(n) on a graph of n nodes and arcs.

Running Times Between Polynomials

- You might worry that something like O(n log n) is not a polynomial.
- However, to be in **P**, a problem only needs an algorithm that runs in time less than some polynomial.
- Surely O(n log n) is less than the polynomial O(n^2).

A Tricky Case: Knapsack

- The *Knapsack Problem* is: given positive integers i_1, i_2 ,..., i_n, can we divide them into two sets with equal sums?
- Perhaps we can solve this problem in polytime by a dynamic-programming algorithm:
  - Maintain a table of all the differences we can achieve by partitioning the first j integers.

Knapsack – (2)

- Basis: j = 0. Initially, the table has "true" for 0 and "false" for all other differences.
- Induction: To consider i_j, start with a new table, initially all false.
- Then, if the entry for m is "true" in the old table set the entries for $m+i_j$ and $m-i_j$ to "true" in the new table.

Knapsack – (3)

- Suppose we measure running time in terms of the sum of the integers, say s.
- Each table needs only space O(s) to represent all the positive and negative differences we could achieve.
- Each table can be constructed in time O(s).

Knapsack – (4)

- Since $n \leq s$, we can build the final table in $O(s^2)$ time.
- From that table, we can see if 0 is achievable and solve the problem.

Subtlety: Measuring Input Size

- "Input size" has a specific meaning: the length of the representation of the problem instance as it is input to a TM.
- For the Knapsack Problem, you cannot always write the input in a number of characters that is polynomial in the sum of the integers.

Knapsack – Bad Case

- Suppose we have n integers, each of which is around 2^n.
- We can write integers in binary, so the input takes $O(n^2)$ space to write down.
- But the tables require space $O(n2^n)$.
- All n tables in time $O(n^2 2^n)$.
  - Or, since we like to use n as the input size, input of length n requires $O(n2^{\mathrm{sqrt}(n)})$ time.

Redefining Knapsack

- We are free to describe another problem, call it *Pseudo-Knapsack*, where integers are represented in unary.
- Pseudo-Knapsack is in **P**.

The Class **NP**

- The running time of a nondeterministic TM is the maximum number of steps taken along any branch.
- If that time bound is polynomial, the NTM is said to be *polynomial-time bounded*.
- And its language/problem is said to be in the class **NP**.

Example: **NP**

- The Knapsack Problem is definitely in **NP**, even using the conventional binary representation of integers.
- Use nondeterminism to guess a partition of the input into two subsets.
- Sum the two subsets and compare.

P Versus **NP**

- Originally a curiosity of Computer Science, mathematicians now recognize as one of the most important open problems the question **P** = **NP**?
- There are thousands of problems that are in **NP** but appear not to be in **P**.
- But no proof that they aren’t really in **P**.

Complete Problems

- One way to address the **P** = **NP** question is to identify *complete problems* for NP.
- An *NP-complete problem* has the property that it is in **NP**, and if it is in **P**, then every problem in **NP** is also in **P**.
- Defined formally via "polytime reductions."

Complete Problems – Intuition

- A complete problem for a class embodies every problem in the class, even if it does not appear so.
- Compare: PCP embodies every TM computation, even though it does not appear to do so.
- Strange but true: Knapsack embodies every polytime NTM computation.

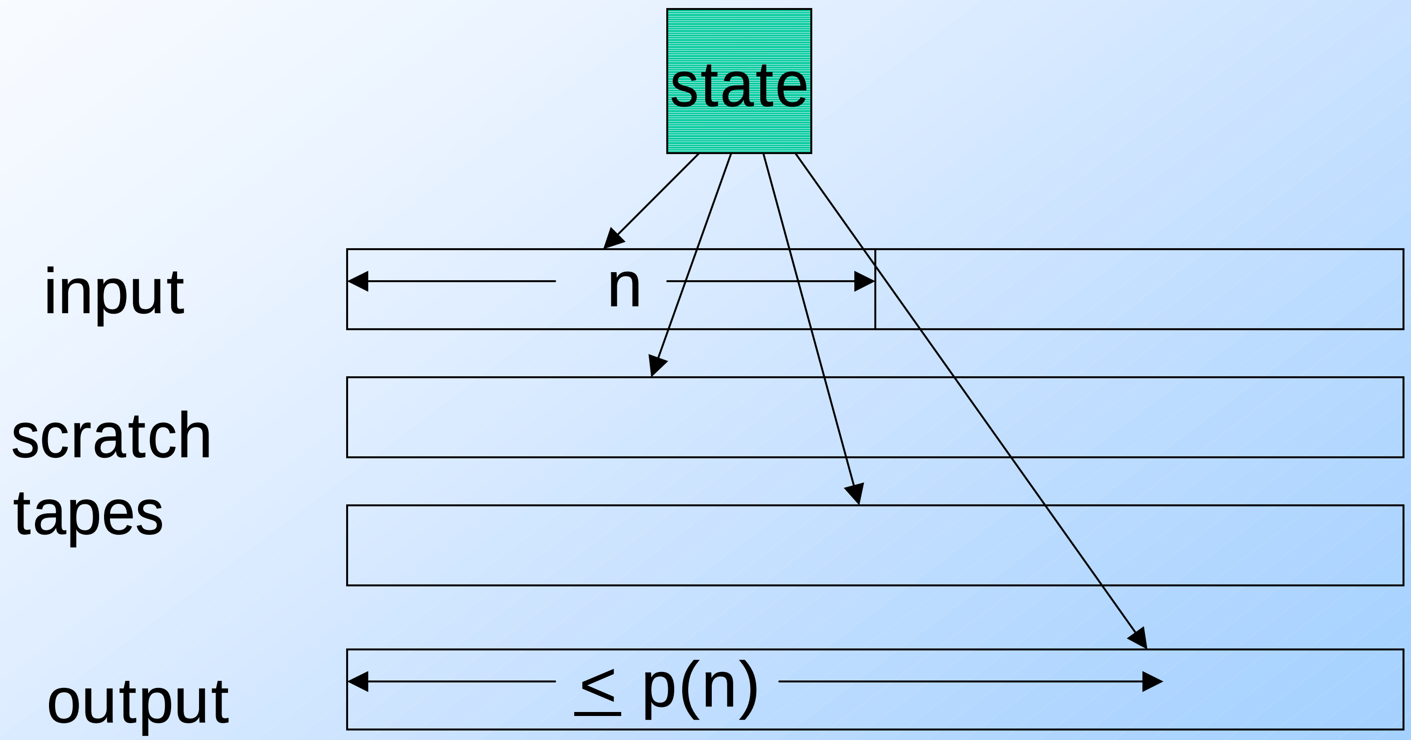
Polytime Reductions

- Goal: find a way to show problem *L* to be NP-complete by reducing every language/problem in **NP** to *L* in such a way that if we had a deterministic polytime algorithm for *L*, then we could construct a deterministic polytime algorithm for any problem in **NP**.

Polytime Reductions – (2)

- We need the notion of a *polytime transducer* – a TM that:
  1. Takes an input of length n.
  2. Operates deterministically for some polynomial time p(n).
  3. Produces an output on a separate *output tape*.
- Note: output length is at most p(n).

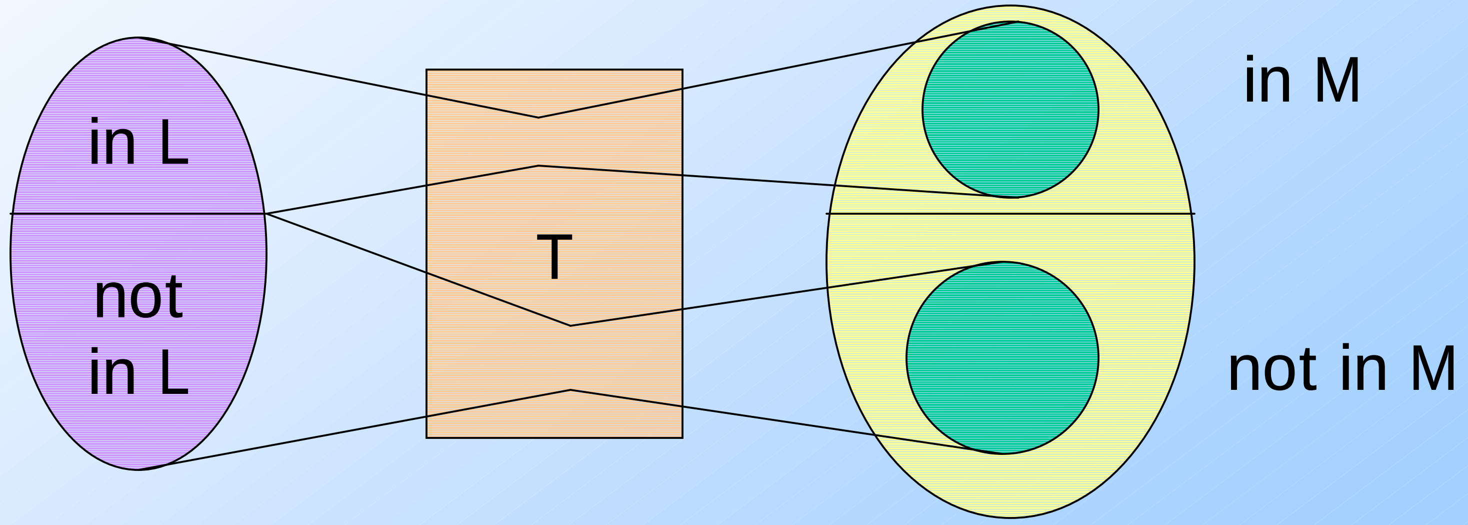
Polytime Transducer



Polytime Reductions – (3)

- Let L and M be langauges.
- Say L is *polytime reducible* to M if there is a polytime transducer T such that for every input w to T, the output x = T(w) is in M if and only if w is in L.

Picture of Polytime Reduction



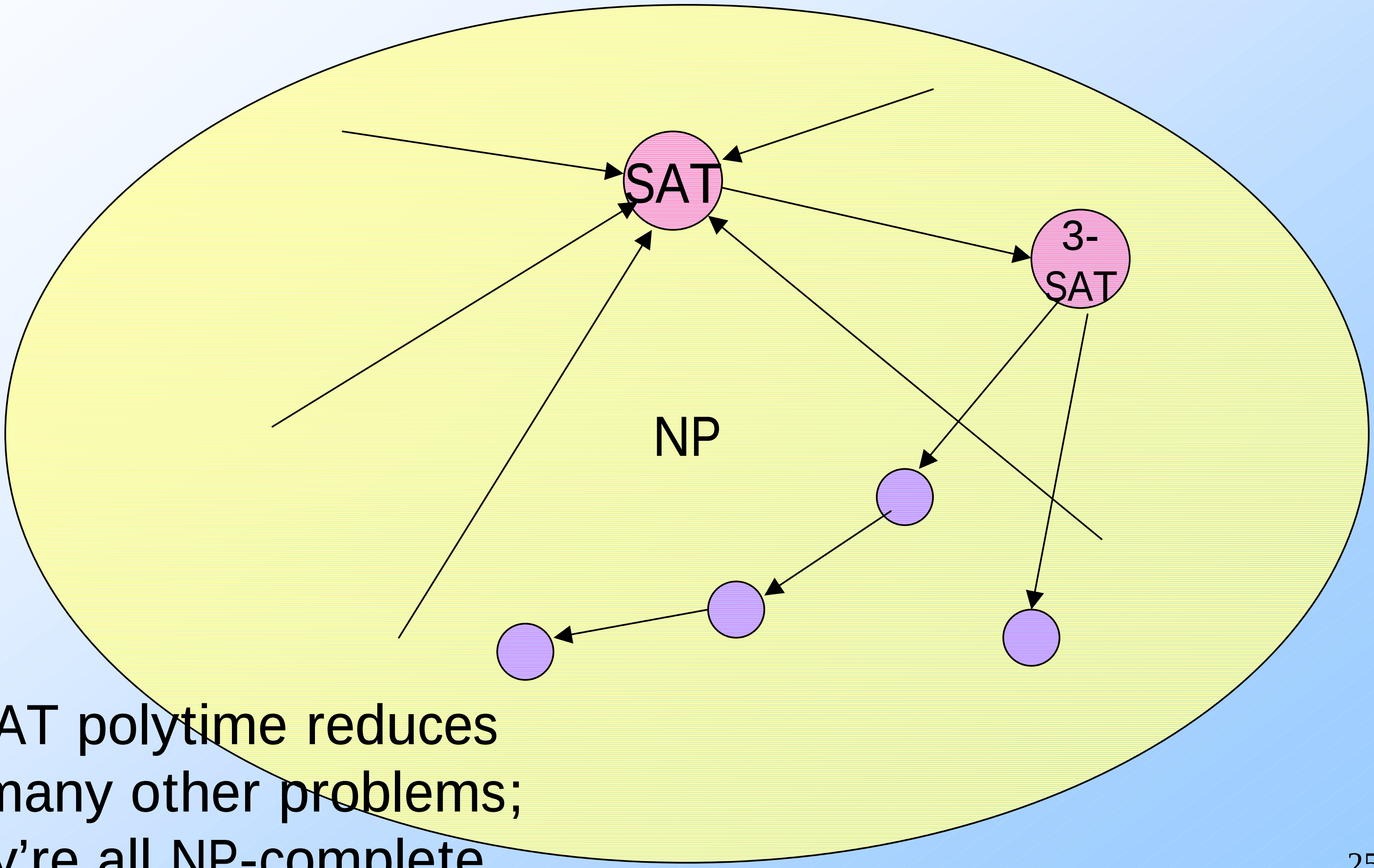
NP-Complete Problems

- A problem/language M is said to be *NP-complete* if it is in **NP**, and for every language L in **NP**, there is a polytime reduction from L to M.
- Fundamental property: if M has a polytime algorithm, then so does L.
  - I.e., if M is in **P**, then every L in **NP** is also in **P**, or "**P** = **NP**."

The Plan

All of **NP** polytime reduces to SAT, which is therefore NP-complete

SAT polytime reduces to 3-SAT



3-SAT polytime reduces to many other problems; they're all NP-complete

Proof That Polytime Reductions “Work”

- Suppose M has an algorithm of polynomial time q(n).
- Let L have a polytime transducer T to M, taking polynomial time p(n).
- The output of T, given an input of length n, is at most of length p(n).
- The algorithm for M on the output of T takes time at most q(p(n)).

Proof – (2)

- We now have a polytime algorithm for L:
  1. Given w of length n, use T to produce x of length $\leq$ p(n), taking time $\leq$ p(n).
  2. Use the algorithm for M to tell if x is in M in time $\leq$ q(p(n)).
  3. Answer for w is whatever the answer for x is.
- Total time $\leq$ p(n) + q(p(n)) = a polynomial.

The Satisfiability Problem

Cook's Theorem: An NP-Complete Problem

Restricted SAT: CSAT, 3SAT

Boolean Expressions

- Boolean, or propositional-logic expressions are built from variables and constants using the operators AND, OR, and NOT.
  - Constants, and the values of variables, are true and false, represented by 1 and 0, respectively.
  - We'll use concatenation for AND, + for OR, - for NOT.

Example: Boolean expression

- (x+y)(-x + -y) is true only when variables x and y have opposite truth values.
- Note: parentheses can be used at will, and are needed to modify the precedence order NOT (highest), AND, OR.

The Satisfiability Problem (*SAT*)

- Study of boolean expressions generally is concerned with the set of *truth assignments* (assignments of 0 or 1 to each of the variables) that make the function true.
- NP-completeness needs only a simpler question (SAT): does there exist a truth assignment making the expression true?

Example: SAT

- (x+y)(-x + -y) is satisfiable.
- There are, in fact, two satisfying truth assignments:
  1. x=0; y=1.
  2. x=1; y=0.
- x(-x) is not satisfiable.

SAT as a Language/Problem

- An instance of SAT is a boolean expression.
- Must be coded in a finite alphabet.
- Use special symbols (,), +, - as themselves.
- Represent the i-th variable by symbol x followed by integer i in binary.

Example: Encoding for SAT

- (x+y)(-x + -y) would be encoded by the string `(x1+x10)(-x1+-x10)`

SAT is in **NP**

- There is a multitape NTM that can decide if a Boolean expression of length n is satisfiable.
- The NTM takes $O(n^2)$ time along any path.
- Use nondeterminism to guess a truth assignment on a second tape.
- Replace all variables by guessed truth values.
- Evaluate the expression for this assignment.
- Accept if true.

Cook’s Theorem

- SAT is NP-complete.
- To prove, we must show how to construct a polytime reduction from each language L in **NP** to SAT.
- Start by assuming the most restricted possible form of NTM for L (next slide).

Assumptions About NTM for L

1. One tape only.
2. Head never moves left of the initial position.
3. States and tape symbols are disjoint.

◆ Key Points: States can be named arbitrarily, and the constructions many-tapes-to-one and two-way-infinite-tape-to-one at most square the time.

More About the NTM M for L

- Let p(n) be a polynomial time bound for M.
- Let w be an input of length n to M.
- If M accepts w, it does so through a sequence $I_0 \vdash I_1 \vdash \ldots \vdash I_{p(n)}$ of p(n)+1 ID's.
  - Assume trivial move from a final state.
- Each ID is of length at most p(n)+1, counting the state.

From ID Sequences to Boolean Expressions

- The Boolean expression that the transducer for L will construct from w will have $(p(n)+1)^2$ "*variables*."
- Let variable X_{ij} represent the j-th position of the i-th ID.
  - i and j each range from 0 to p(n).

Picture of Computation as an Array

$$
\begin{array}{r|llll|}
\hline
\text{Initial ID} & X_{00} & X_{01} & \dots & X_{0p(n)} \\
\hline
I_1 & X_{10} & X_{11} & \dots & X_{1p(n)} \\
\hline
. & & & . & \\
. & & & . & \\
. & & & . & \\
\hline
I_{p(n)} & X_{p(n)0} & X_{p(n)1} & \dots & X_{p(n)p(n)} \\
\hline
\end{array}
$$

Intuition

- From M and w we construct a boolean expression that forces the X's to represent one of the possible ID sequences of NTM M with input w, if it is to be satisfiable.
- And the expression *is* satisfiable if some sequence leads to acceptance.

From ID's to Boolean Variables

- The X_{ij}'s are not boolean variables; they are states and tape symbols of M.
- However, we can represent the value of each X_{ij} by a family of Boolean variables y_{ijA}, for each possible state or tape symbol A.
- y_{ijA} is true if and only if $X_{ij} = A$.

Points to Remember

1. The Boolean expression has components that depend on n.
   - These must be of size polynomial in n.
2. Other pieces depend only on M.
   - No matter how many states/symbols M has, these are of constant size.
3. Any logical expression about a set of variables whose size is independent of n can be written in constant time.

Designing the Expression

- We want the Boolean expression that describes the X_{ij}'s to be satisfiable if and only if the NTM M accepts w.
- Four conditions:
  1. Unique: only one symbol per position.
  2. Starts right: initial ID is q_0w.
  3. Moves right: each ID follows from the previous by a move of M.
  4. Finishes right: M accepts.

Unique

- Take the AND over all i, j, Y, and Z of $(-y_{ijY} + -y_{ijZ})$.
- That is, it is not possible for X_{ij} to be both symbols Y and Z.

Starts Right

- The Boolean Function needs to assert that the first ID is the correct one with $w = a_1...a_n$ as input.
  1. $X_{00} = q_0$.
  2. $X_{0i} = a_i$ for $i = 1,..., n$.
  3. $X_{0i} = B$ (blank) for $i = n+1,..., p(n)$.
- Formula is the AND of y_{0iZ} for all i, where Z is the symbol in position i.

Finishes Right

- The last ID must have an accepting state.
- Form the OR of Boolean variables $y_{p(n),j,q}$ where j is arbitrary and q is an accepting state.

Running Time So Far

- Unique requires $O(p^2(n))$ symbols be written.
  - Parentheses, signs, propositional variables.
- Algorithm is easy, so it takes no more time than $O(p^2(n))$.
- Starts Right takes $O(p(n))$ time.
- Finishes Right takes $O(p(n))$ time.

Running Time – (2)

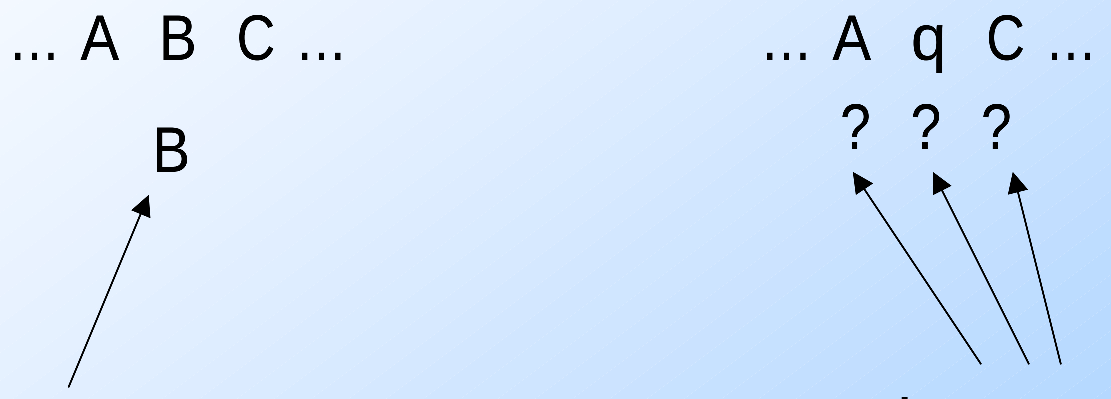
- Caveat: Technically, the propositions that are output of the transducer must be coded in a fixed alphabet, e.g., x10011 rather than y_{ijA}.
- Thus, the time and output length have an additional factor O(log n) because there are $O(p^2(n))$ variables.
- But log factors do not affect polynomials

Moves Right

Works because Unique assures only one y_{ijX} true.

- $X_{ij} = X_{i-1,j}$ whenever the state is none of $X_{i-1,j-1}$, $X_{i-1,j}$, or $X_{i-1,j+1}$.
- For each i and j, construct an expression that says (in propositional variables) the OR of "$X_{ij} = X_{i-1,j}$" and all $y_{i-1,k,A}$ where A is a state symbol (k = i-1, i, or i+1).
  - Note: $X_{ij} = X_{i-1,j}$ is the OR of $y_{ijA}.y_{i-1,jA}$ for all symbols A.

Constraining the Next Symbol



Easy case; must be B

Hard case; all three may depend on the move of M

Moves Right – (2)

- In the case where the state is nearby, we need to write an expression that:
  1. Picks one of the possible moves of the NTM M.
  2. Enforces the condition that when $X_{i-1,j}$ is the state, the values of $X_{i,j-1}$, $X_{i,j}$, and $X_{i,j+1}$. are related to $X_{i-1,j-1}$, $X_{i-1,j}$, and $X_{i-1,j+1}$ in a way that reflects the move.

Example: Moves Right

Suppose δ(q, A) contains (p, B, L).

Then one option for any i, j, and C is:

C q A

p C B

If δ(q, A) contains (p, B, R), then an option for any i, j, and C is:

C q A

C B p

Moves Right – (3)

- For each possible move, and for each i and j, express the constraints on the six X's by a Boolean expression.
- For each i and j, take the OR over all possible moves.
- Take the AND over all i and j.
- Small point: for edges (e.g., state at 0), assume invisible symbols are blank.

Running Time

- We have to generate $O(p^2(n))$ Boolean expressions, but each is constructed from the moves of the NTM M, which is fixed in size, independent of the input w.
- Takes time $O(p^2(n))$ and generates an output of that length.
  - Times log n, because variables must be coded in a fixed alphabet.

Cook's Theorem – Finale

- In time $O(p^2(n) \log n)$ the transducer produces a Boolean expression, the AND of the four components: Unique, Starts, Finishes, and Moves Right.
- If M accepts w, the ID sequence gives us a satisfying truth assignment.
- If satisfiable, the truth values tell us an accepting computation of M.

Conjunctive Normal Form

- A Boolean expression is in *Conjunctive Normal Form* (CNF) if it is the AND of *clauses*.
- Each clause is the OR of *literals*.
- A literal is either a variable or the negation of a variable.
- Problem *CSAT* : is a Boolean expression in CNF satisfiable?

Example: CNF

(x + -y + z)(-x)(-w + -x + y + z) (…

NP-Completeness of CSAT

- The proof of Cook's theorem can be modified to produce a formula in CNF.
- Unique is already the AND of clauses.
- Starts Right is the AND of clauses, each with one variable.
- Finishes Right is the OR of variables, i.e., a single clause.

NP-Completeness of CSAT – (2)

- Only Moves Right is a problem, and not much of a problem.
- It is the AND of expressions for each i and j.
- Those expressions are fixed, independent of n.

NP-Completeness of CSAT – (3)

- You can convert any expression to CNF.
- It may exponentiate the size of the expression and therefore take time to write down that is exponential in the size of the original expression, but these numbers are all fixed for a given NTM M and independent of n.

Conversion to CNF

- For each truth assignment to the variables of the expression that make the expression false:
- Use a clause that is the OR of each variable negated iff it is assigned "true."
- Thus, the clause is false for this truth assignment and only this.
- Take the AND of all these clauses.

Example: Conversion to CNF

- Consider expression –x + yz.
- There are three falsifying assignments: (x=1, y=1, z=0), (x=1, y=0, z=0), and (x=1, y=0, z=1).
- The resulting CNF expression is therefore (-x+-y+z)(-x+y+z)(-x+y+-z).

k-SAT

- If a Boolean expression is in CNF and every clause has exactly k literals, we say the expression is in *k-CNF*.
- The problem *k-SAT* is SAT restricted to expressions in k-CNF.
- Example: We just saw a 3-SAT formula

(-x+-y+z)(-x+y+z)(-x+y+-z).

3-SAT

- This problem is NP-complete.
- Clearly it is in **NP**, since SAT is.
- It is not true that every Boolean expression can be converted to an equivalent 3-CNF expression.

3SAT – (2)

- But we don't need equivalence.
- We need to reduce every CNF expression E to some 3-CNF expression that is satisfiable if and only if E is.
- Reduction involves introducing new variables into long clauses, so we can split them apart.

Reduction of CSAT to 3SAT

- Let $(x_1+\ldots+x_k)$ be a clause in some CSAT instance, with $k \geq 4$.
  - Note: the x's are literals, not variables; any of them could be negated variables.
- Introduce new variables $y_1,\ldots,y_{k-3}$ that appear in no other clause.

CSAT to 3SAT – (2)

- Replace $(x_1+\ldots+x_k)$ by $(x_1+x_2+y_1)(x_3+y_2+ -y_1) \ldots (x_i+y_{i-1}+ -y_{i-2}) \ldots (x_{k-2}+y_{k-3}+ -y_{k-4})(x_{n-1}+x_k+ -y_{k-3})$
- If there is a satisfying assignment of the x's for the CSAT instance, then one of the literals x_i must be made true.
- Assign y_j = true if $j < i-1$ and y_j = false for larger j.

CSAT to 3SAT – (3)

- Suppose $(x_1+x_2+y_1)(x_3+y_2+ -y_1)$ ... $(x_{k-2}+y_{k-3}+ -y_{k-4})(x_{k-1}+x_k+ -y_{k-3})$ is satisfiable, but none of the x's is true.
- The first clause forces y_1 = true.
- Then the second clause forces y_2 = true.
- And so on ... all the y's must be true.
- But then the last clause is false.

CSAT to 3SAT – (4)

- There is a little more to the reduction, for handling clauses of 1 or 2 literals.
- Replace (x) by $(x+y_1+y_2)$ $(x+y_1+ -y_2)$ $(x+ -y_1+y_2)$ $(x+ -y_1+ -y_2)$.
  - Remember: the y's are different variables for each CNF clause.
- Replace (w+x) by (w+x+y)(w+x+ -y).

CSAT to 3SAT Running Time

- This reduction is surely polynomial.
- In fact it is linear in the length of the CSAT instance.
- Thus, we have polytime-reduced CSAT to 3-SAT.
- Since CSAT is NP-complete, so is 3-SAT.

More NP-Complete Problems

NP-Hard Problems

Tautology Problem

Node Cover

Knapsack

Next Steps

- We can now reduce 3-SAT to a large number of problems, either directly or indirectly.
- Each reduction must be polytime.
- Usually we focus on length of the output from the transducer, because the construction is easy.

Next Steps – (2)

- Another essential part of an NP-completeness proof is showing the problem is in **NP**.
- Sometimes, we can only show a problem *NP-hard* = "if the problem is in **P**, then **P** = **NP**," but the problem may not be in **NP**.

Example: NP-Hard Problem

- The *Tautology Problem* is: given a Boolean expression, is it satisfied by all truth assignments?
  - Example: x + -x + yz
- Not obviously in **NP**, but it's complement is.
  - Guess a truth assignment; accept if that assignment doesn't satisfy the expression.

Co-**NP**

- A problem/language whose complement is in **NP** is said to be in ***Co-NP***.
- Note: **P** is closed under complementation.
- Thus, **P** $\subseteq$ Co-**NP**.
- Also, if **P** = **NP**, then **P** = **NP** = Co-**NP**.

Tautology is NP-Hard

- While we can't prove Tautology is in **NP**, we can prove it is NP-hard.
- Suppose we had a polytime algorithm for Tautology.
- Take any Boolean expression E and convert it to NOT(E).
  - Obviously linear time.

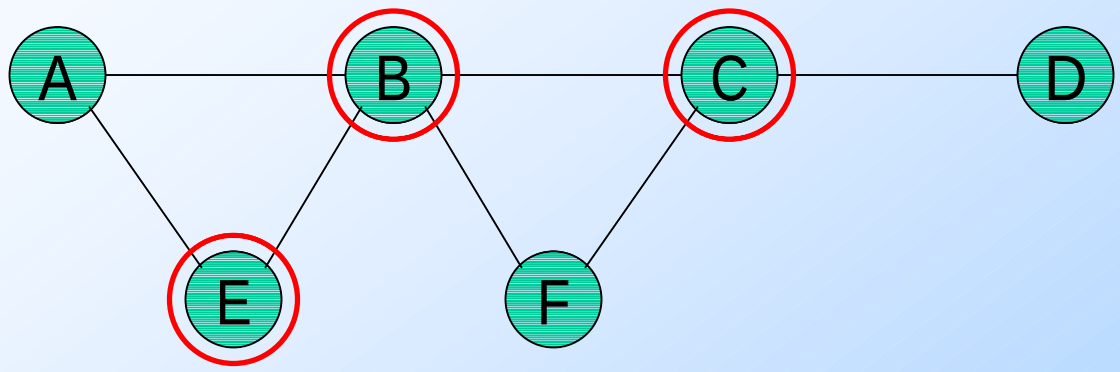
Tautology is NP-Hard – (2)

- E is satisfiable if and only NOT(E) is not a tautology.
- Use the hypothetical polytime algorithm for Tautology to test if NOT(E) is a tautology.
- Say "yes, E is in SAT" if NOT(E) is not a tautology and say "no" otherwise.
- Then SAT would be in **P**, and **P** = **NP**.

The Node Cover Problem

- Given a graph G, we say N is a *node cover* for G if every edge of G has at least one end in N.
- The problem Node Cover is: given a graph G and a "budget" k, does G have a node cover of k or fewer nodes?

Example: Node Cover



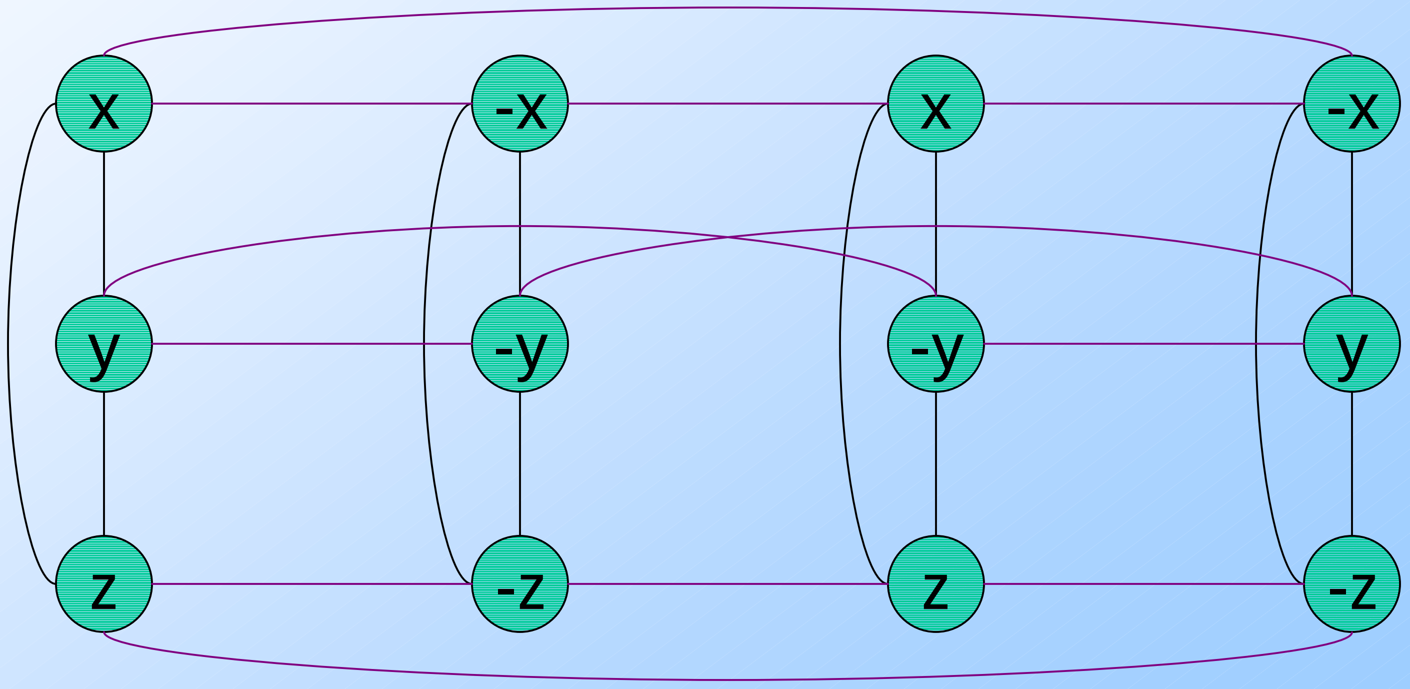
One possible node cover
of size 3: {B, C, E}

NP-Completeness of Node Cover

- Reduction from 3-SAT.
- For each clause (X+Y+Z) construct a “column” of three nodes, all connected by *vertical* edges.
- Add a *horizontal* edge between nodes that represent any variable and its negation.
- Budget = twice the number of clauses.

Example: The Reduction to Node Cover

(x + y + z)(-x + -y + -z)(x + -y +z)(-x + y + -z)



Budget = 8

Example: Reduction – (2)

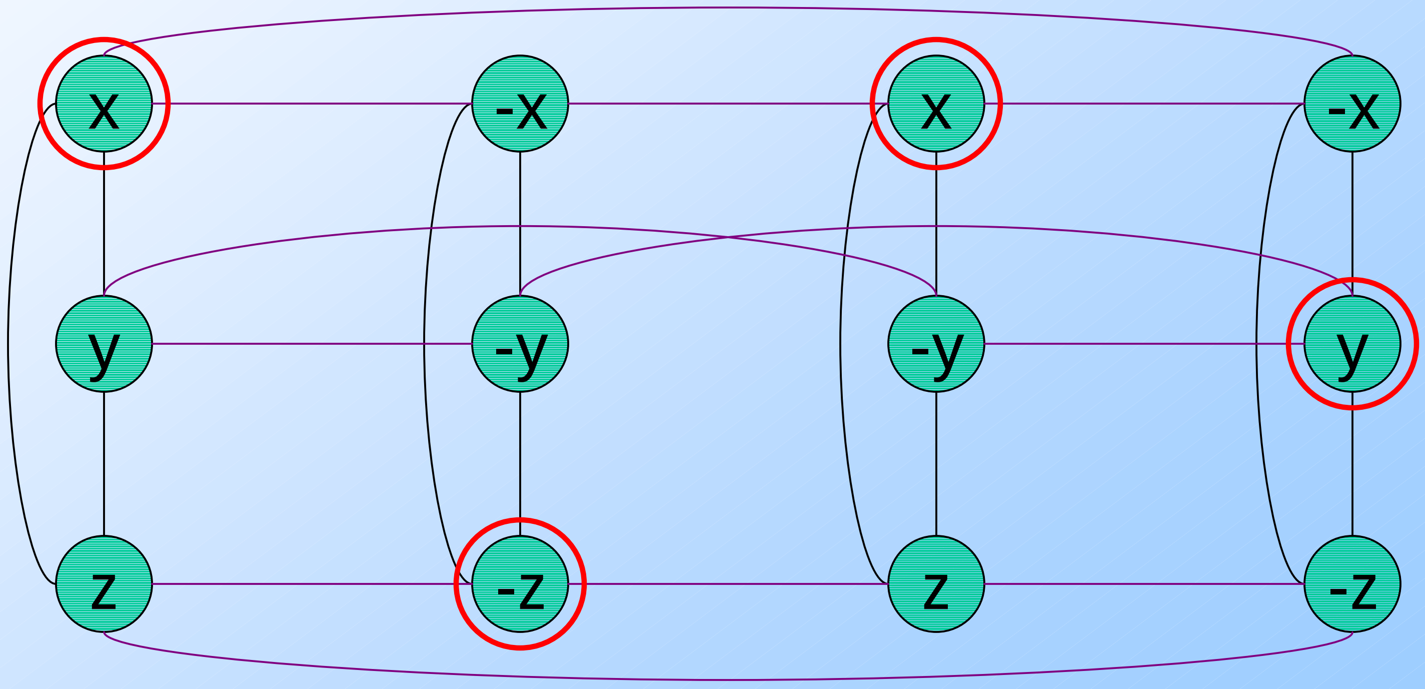
- A node cover must have at least two nodes from every column, or some vertical edge is not covered.
- Since the budget is twice the number of columns, there must be exactly two nodes in the cover from each column.
- Satisfying assignment corresponds to the node in each column not selected.

Example: Reduction – (3)

(x + y + z)(-x + -y + -z)(x + -y +z)(-x + y + -z)

Truth assignment: x = y = T; z = F

Pick a true node in each column

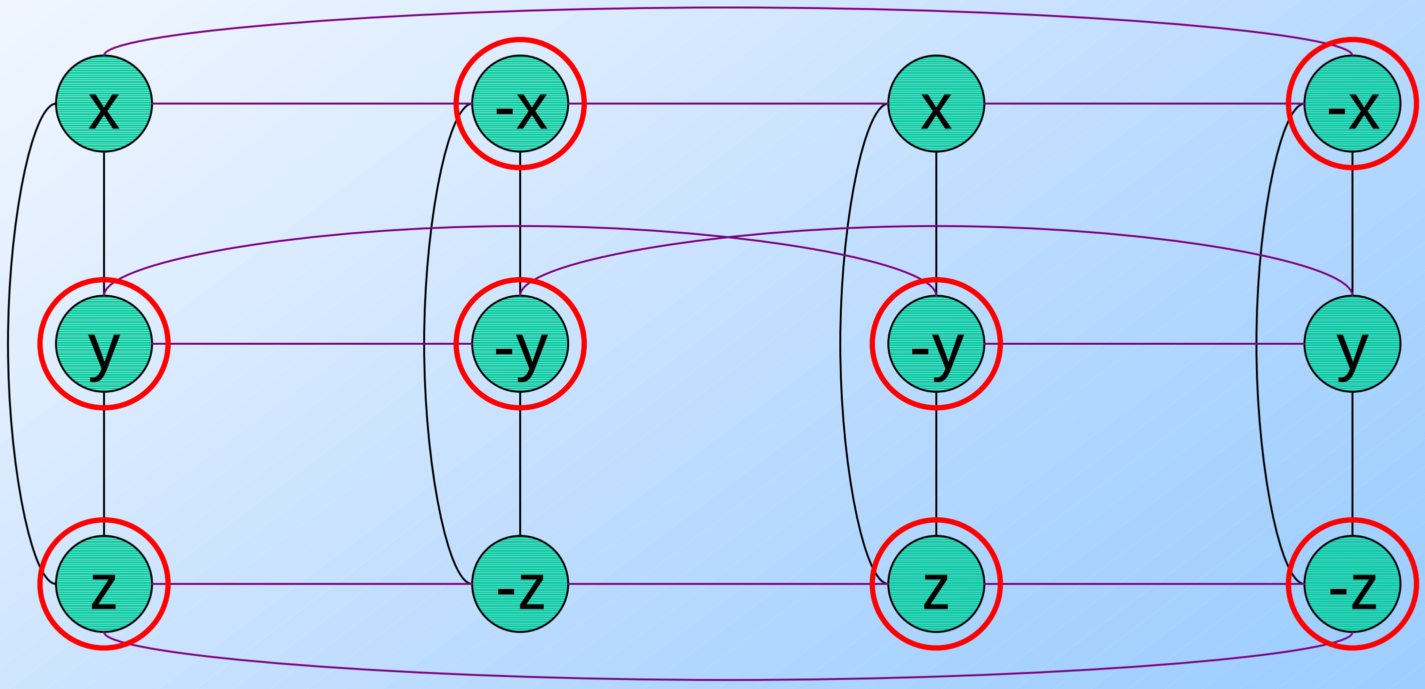


Example: Reduction – (4)

(x + y + z)(-x + -y + -z)(x + -y +z)(-x + y + -z)

Truth assignment: x = y = T; z = F

The other nodes form a node cover



Proof That the Reduction Works

- The reduction is clearly polytime.
- Need to show: If we construct from 3-SAT instance E a graph G and a budget k, then G has a node cover of size $\leq$ k if and only if E is satisfiable.

Proof: If

- Suppose we have a satisfying assignment A for E.
- For each clause of E, pick one of its three literals that A makes true.
- Put in the node cover the two nodes for that clause that do not correspond to the selected literal.
- Total = k nodes – meets budget.

Proof: If – (2)

- The selected nodes cover all vertical edges.
  - Why? Any two nodes for a clause cover the triangle of vertical edges for that clause.
- Horizontal edges are also covered.
  - A horizontal edge connects nodes for some x and -x.
  - One is false in A and therefore its node must be selected for the node cover.

Proof: Only If

- Suppose G has a node cover with at most k nodes.
- One node cannot cover the vertical edges of any column, so each column has exactly 2 nodes in the cover.
- Construct a satisfying assignment for E by making true the literal for any node not in the node cover.

Proof: Only If – (2)

- Worry: What if there are unselected nodes corresponding to both x and -x?
  - Then we would not have a truth assignment.
- But there is a horizontal edge between these nodes.
- Thus, at least one is in the node cover.

The Knapsack Problem

- We shall prove NP-complete a version of Knapsack with a target:
  - Given a list L of integers and a target k, is there a subset of L whose sum is exactly k?
- Later, we'll reduce this version of Knapsack to our earlier one: given an integer list L, can we divide it into two equal parts?

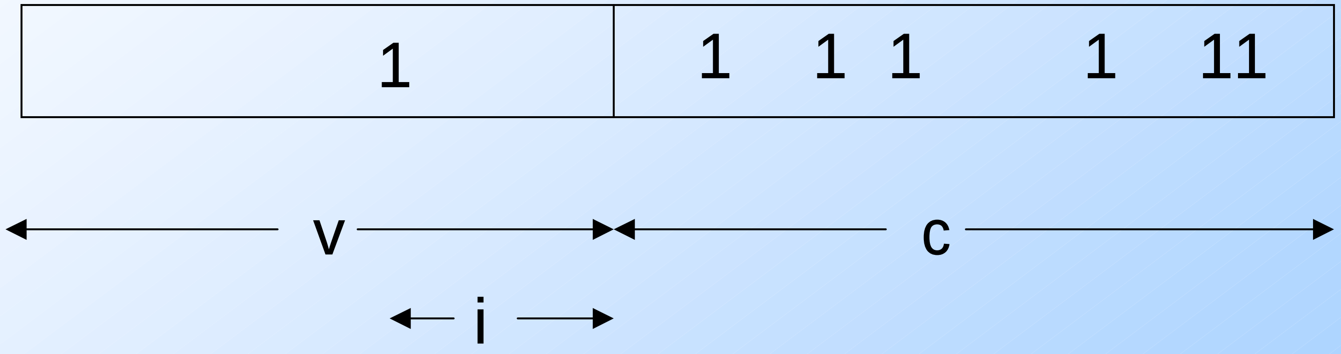
Knapsack-With-Target is in **NP**

- Guess a subset of the list L.
- Add ‘em up.
- Accept if the sum is k.

Polytime Reduction of 3-SAT to Knapsack-With-Target

- Given 3-SAT instance E, we need to construct a list L and a target k.
- Suppose E has c clauses and v variables.
- L will have base-32 integers of length c+v, and there will be 3c+2v of them.

Picture of Integers for Literals

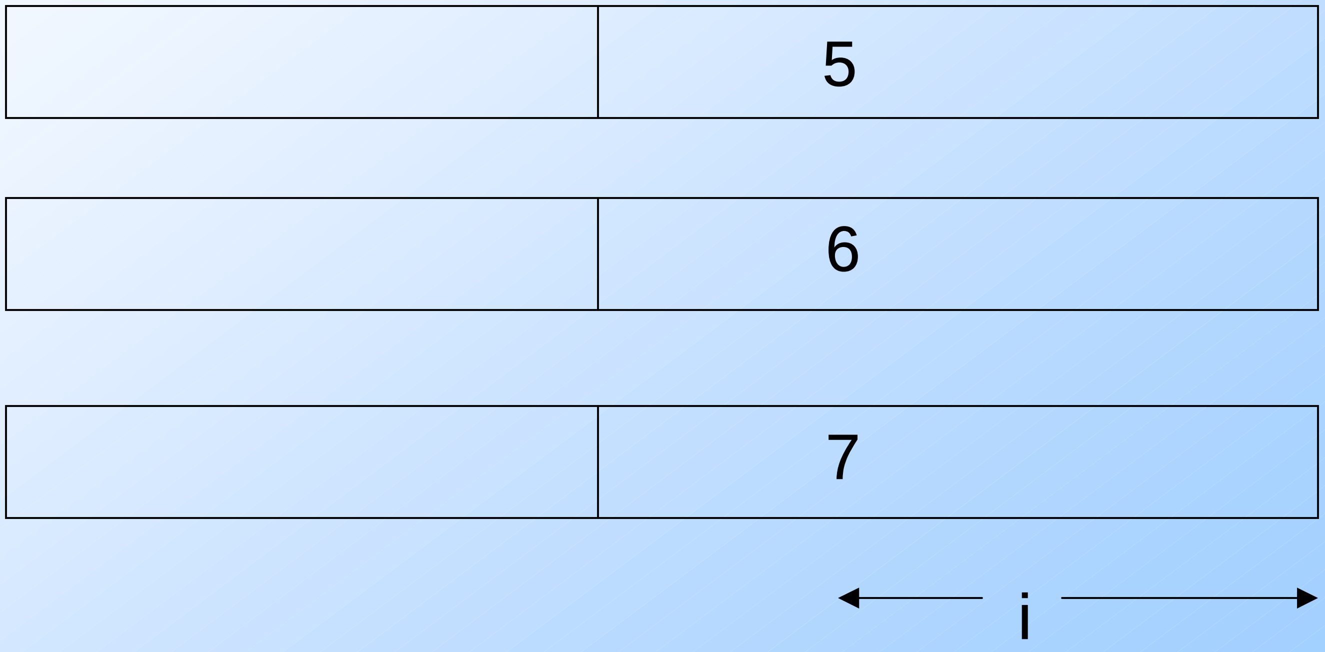


1 in i-th position if this integer is for x_i or $-x_i$.

1's in all positions such that this literal makes the clause true.

All other positions are 0.

Pictures of Integers for Clauses



For the i-th clause

Example: Base-32 Integers

$(x + y + z)(x + -y + -z)$

- c = 2; v = 3.
- Assume x, y, z are variables 1, 2, 3, respectively.
- Clauses are 1, 2 in order given.

Example: (x + y + z)(x + -y + -z)

- For x: 00111
- For -x: 00100
- For y: 01001
- For -y: 01010
- For z: 10001
- For -z: 10010

- For first clause: 00005, 00006, 00007
- For second clause: 00050, 00060, 00070

The Target

- $k = 8(1+32+32^2+\ldots+32^{c-1}) + 32^c(1+32+32^2+\ldots+32^{v-1})$
- That is, 8 for the position of each clause and 1 for the position of each variable.
  - $k = (11\ldots188\ldots8)_{32}$.
- Key Point: Base-32 is high enough that there can be no carries between positions.

Key Point: Details

- Among all the integers, the sum of digits in the position for a variable is 2.
- And for a clause, it is 1+1+1+5+6+7 = 21.
  - 1's for the three literals in the clause; 5, 6, and 7 for the integers for that clause.
- Thus, the target must be met on a position-by-position basis.

Key Point: Concluded

- Thus, if a set of integers matches the target, it must include exactly one of the integers for x and -x.
- For each clause, at least one of the integers for literals must have a 1 there, so we can choose either 5, 6, or 7 to make 8 in that position.

Proof the Reduction Works

- Each integer can be constructed from the 3-SAT instance E in time proportional to its length.
  - Thus, reduction is $O(n^2)$.
- If E is satisfiable, take a satisfying assignment A.
- Pick integers for those literals that A makes true.

Proof the Reduction Works – (2)

- The selected integers sum to between 1 and 3 in the digit for each clause.
- For each clause, choose the integer with 5, 6, or 7 in that digit to make a sum of 8.
- These selected integers sum to exactly the target.

Proof: Converse

- We must also show that a sum of integers equal to the target k implies E is satisfiable.
- In each digit for a variable x, either the integer for x or the digit for -x, but not both is selected.
- Let truth assignment A make this literal true.

Proof: Converse – (2)

- In the digits for the clauses, a sum of 8 can only be achieved if among the integers for the variables, there is at least one 1 in that digit.
- Thus, truth assignment A makes each clause true, so it satisfies E.

The *Partition-Knapsack Problem*

- This problem is what we originally referred to as “knapsack.”
- Given a list of integers L, can we partition it into two disjoint sets whose sums are equal?
- Partition-Knapsack is NP-complete; reduction from Knapsack-With-Target.

Reduction of Knapsack-With-Target to Partition-Knapsack

- Given instance (L, k) of Knapsack-With-Target, compute the sum s of all the integers in L.
  - Linear in input size.
- Output is L followed by two integers: 2k and s.
- Example: L = 3, 4, 5, 6; k = 7.
  - Partition-Knapsack instance = 3, 4, 5, 6, 14, 18.

Solution Solution

Proof That Reduction Works

- The sum of all integers in the output instance is 2(s+k).
  - Thus, the two partitions must each sum to s+k.
- If the input instance has a subset of L that sums to k, then pick it plus the integer s to solve the output instance.

Proof: Converse

- Suppose the output instance of Partition-Knapsack has a solution.
- The integers s and 2k cannot be in the same partition.
  - Because their sum is more than half 2(s+k).
- Thus, the subset of L that is in the partition with s sums to k.
  - Thus, it solves the input instance.

Equivalence of PDA, CFG

Conversion of CFG to PDA

Conversion of PDA to CFG

Overview

- When we talked about closure properties of regular languages, it was useful to be able to jump between RE and DFA representations.
- Similarly, CFG's and PDA's are both useful to deal with properties of the CFL's.

Overview – (2)

- Also, PDA's, being "algorithmic," are often easier to use when arguing that a language is a CFL.
- Example: It is easy to see how a PDA can recognize balanced parentheses; not so easy as a grammar.

Converting a CFG to a PDA

- Let L = L(G).
- Construct PDA P such that N(P) = L.
- P has:
  - One state q.
  - Input symbols = terminals of G.
  - Stack symbols = all symbols of G.
  - Start symbol = start symbol of G.

Intuition About P

- At each step, P represents some *left-sentential form* (step of a leftmost derivation).
- If the stack of P is α, and P has so far consumed x from its input, then P represents left-sentential form xα.
- At empty stack, the input consumed is a string in L(G).

Transition Function of P

1. $\delta(q, a, a) = (q, \epsilon)$. (*Type 1* rules)
   - This step does not change the LSF represented, but "moves" responsibility for *a* from the stack to the consumed input.
2. If $A \rightarrow \alpha$ is a production of G, then $\delta(q, \epsilon, A)$ contains (q, α). (*Type 2* rules)
   - Guess a production for A, and represent the next LSF in the derivation.

Proof That L(P) = L(G)

- We need to show that $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{lm} w\alpha$.
- Part 1: "only if" is an induction on the number of steps made by P.
- Basis: 0 steps.
  - Then $\alpha = S$, $w = \epsilon$, and $S \Rightarrow^*_{lm} S$ is surely true.

Induction for Part 1

- Consider n moves of P: $(q, wx, S) \vdash^* (q, x, \alpha)$ and assume the IH for sequences of n-1 moves.
- There are two cases, depending on whether the last move uses a Type 1 or Type 2 rule.

Use of a Type 1 Rule

- The move sequence must be of the form $(q, yax, S) \vdash^* (q, ax, a\alpha) \vdash (q, x, \alpha)$, where $ya = w$.
- By the IH applied to the first n-1 steps, $S \Rightarrow^*_{lm} ya\alpha$.
- But $ya = w$, so $S \Rightarrow^*_{lm} w\alpha$.

Use of a Type 2 Rule

- The move sequence must be of the form $(q, wx, S) \vdash^* (q, x, A\beta) \vdash (q, x, \gamma\beta)$, where $A \rightarrow \gamma$ is a production and $\alpha = \gamma\beta$.
- By the IH applied to the first n-1 steps, $S \Rightarrow^*_{lm} wA\beta$.
- Thus, $S \Rightarrow^*_{lm} w\gamma\beta = w\alpha$.

Proof of Part 2 (“if”)

- We also must prove that if $S \Rightarrow^*_{lm} w\alpha$, then $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x.
- Induction on number of steps in the leftmost derivation.
- Ideas are similar; omitted.

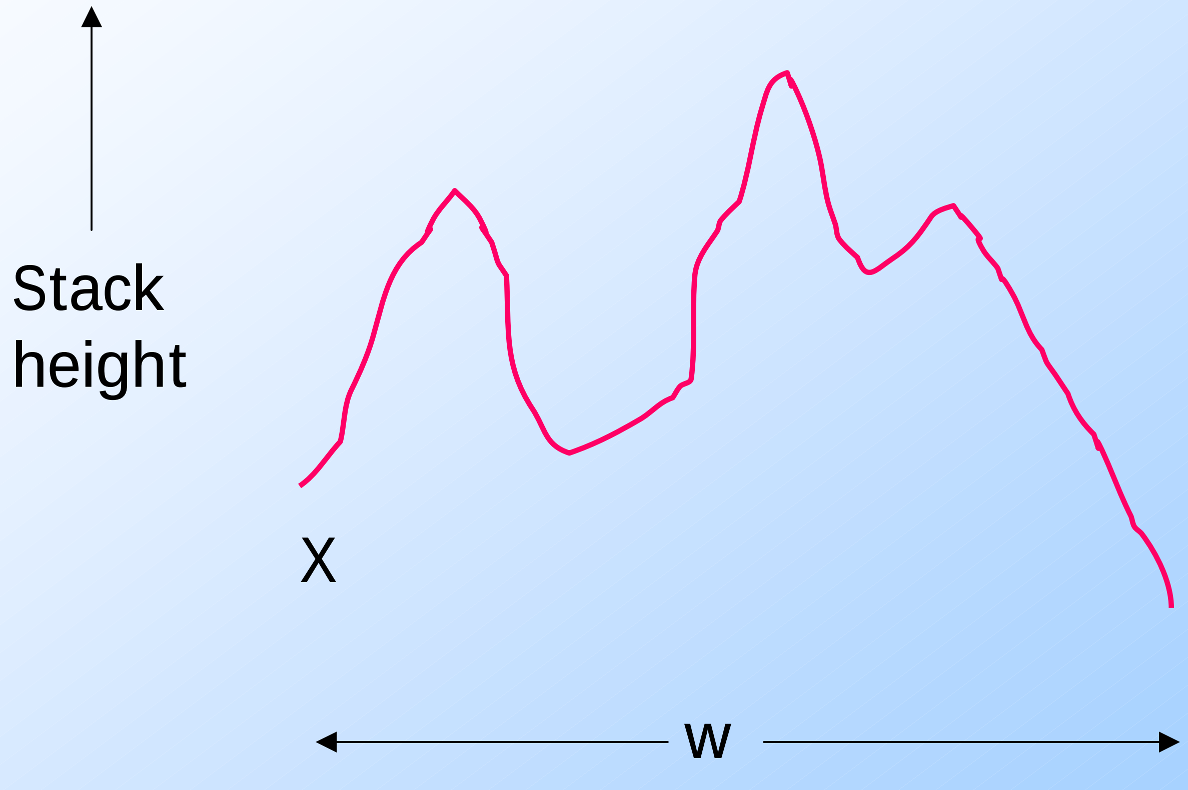
Proof – Completion

- We now have $(q, wx, S) \vdash^* (q, x, \alpha)$ for any x if and only if $S \Rightarrow^*_{lm} w\alpha$.
- In particular, let $x = \alpha = \epsilon$.
- Then $(q, w, S) \vdash^* (q, \epsilon, \epsilon)$ if and only if $S \Rightarrow^*_{lm} w$.
- That is, w is in N(P) if and only if w is in L(G).

From a PDA to a CFG

- Now, assume L = N(P).
- We'll construct a CFG G such that L = L(G).
- Intuition: G will have variables [pXq] generating exactly the inputs that cause P to have the net effect of popping stack symbol X while going from state p to state q.
  - P never gets below this X while doing so.

Picture: Popping X



Variables of G

- G’s variables are of the form [pXq].
- This variable generates all and only the strings w such that

  $(p, w, X) \vdash^* (q, \epsilon, \epsilon)$.
- Also a start symbol S we’ll talk about later.

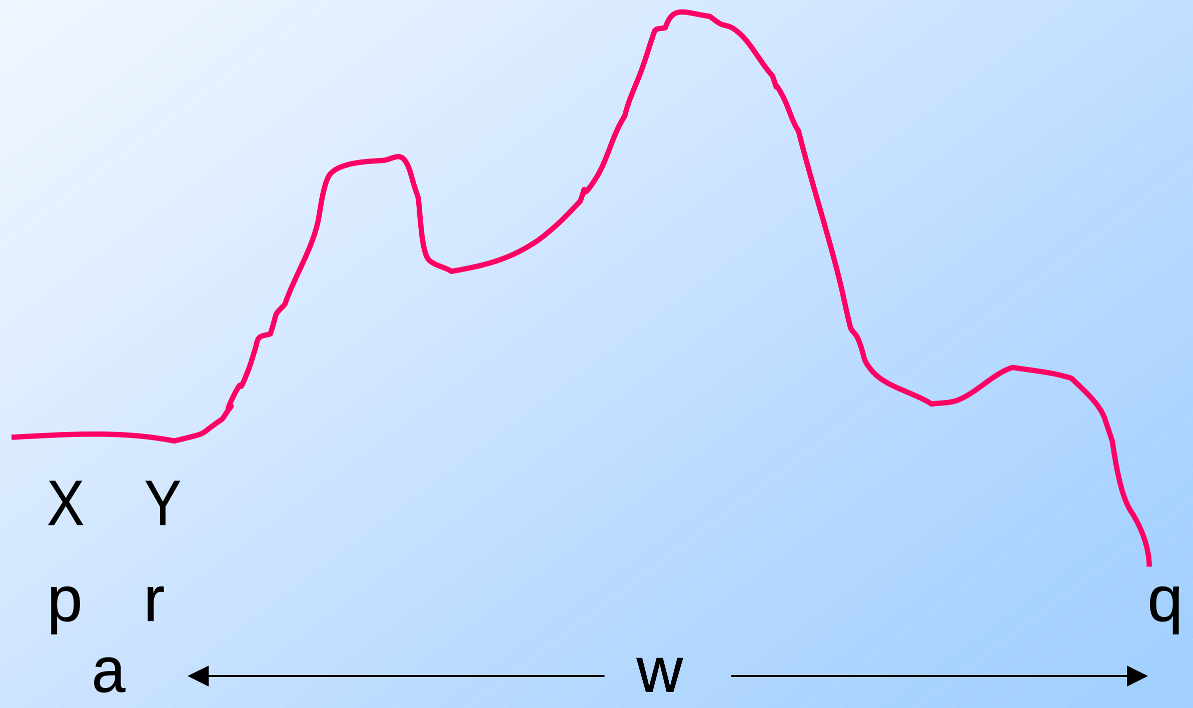
Productions of G

- Each production for [pXq] comes from a move of P in state p with stack symbol X.
- Simplest case: δ(p, a, X) contains (q, ϵ).
  - Note *a* can be an input symbol or ϵ.
- Then the production is [pXq] -> a.
- Here, [pXq] generates *a*, because reading *a* is one way to pop X and go from p to q.

Productions of G – (2)

- Next simplest case: δ(p, a, X) contains (r, Y) for some state r and symbol Y.
- G has production [pXq] -> a[rYq].
  - We can erase X and go from p to q by reading *a* (entering state r and replacing the X by Y) and then reading some w that gets P from r to q while erasing the Y.

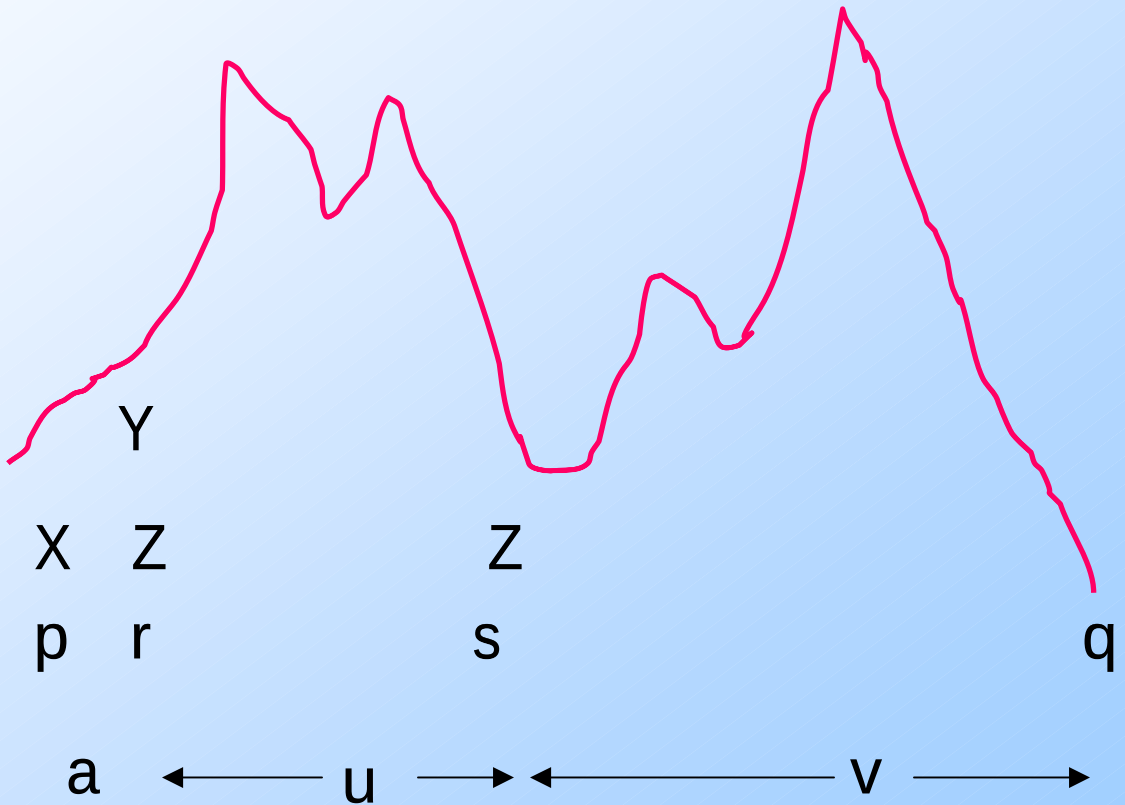
Picture of the Action



Productions of G – (3)

- Third simplest case: $\delta(p, a, X)$ contains (r, YZ) for some state r and symbols Y and Z.
- Now, P has replaced X by YZ.
- To have the net effect of erasing X, P must erase Y, going from state r to some state s, and then erase Z, going from s to q.

Picture of the Action



Third-Simplest Case – Concluded

- Since we do not know state s, we must generate a family of productions:

  [pXq] -> a[rYs][sZq]

  for all states s.
- [pXq] =>* auv whenever [rYs] =>* u and [sZq] =>* v.

Productions of G: General Case

- Suppose δ(p, a, X) contains (r, $Y_1,\dots Y_k$) for some state r and k $\geq$ 3.
- Generate family of productions

[pXq] ->

$$a[rY_1s_1][s_1Y_2s_2]\dots[s_{k-2}Y_{k-1}s_{k-1}][s_{k-1}Y_kq]$$

Completion of the Construction

- We can prove that $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$ if and only if $[q_0Z_0p] \Rightarrow^* w$.
  - Proof is two easy inductions.
- But state p can be anything.
- Thus, add to G another variable S, the start symbol, and add productions $S \to [q_0Z_0p]$ for each state p.

Discussion of Common Problems – 2

Kleene Star

Paths in DFA

Challenge Problem 1

Kleene Star

- Example
  - **1***
  - Does NOT mean an infinite long string of 1's
  - L(**1***)={ε,1,11,111,1111,…}
  - Each element in L(**1***) has finite length

Infiniteness

- Infinite objects are important in mathematics (set of integers, a line containing infinite number of points)
- In a computational model, you can never read such object into a computer
  - Unless you can represent it in finite form
    - Regular expression for a regular language
    - Triple (a,b,c) for a line ax+by+c=0

What is "Last"

- A DFA which accepts all strings of 0's and 1's except those whose last character is 1
- What about the string **1**?
- The last character of any string $a_1a_2 \ldots a_n$ is just a_n ($n \geq 1$).
- So **1** is not accepted by this DFA
- ε has no "last" character, so it IS accepted by this DFA

Convert DFA into RE

- Review
  - k-Path Induction
  - Let R_{ij}^{k} be the regular expression for the set of labels of k-paths from state i to j

k-Path Inductive Case

- A k-path from i to j either:
  1. Never goes through state k, or
  2. Goes through k one or more times.

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^{*} R_{kj}^{k-1}.$$

Doesn't go through k

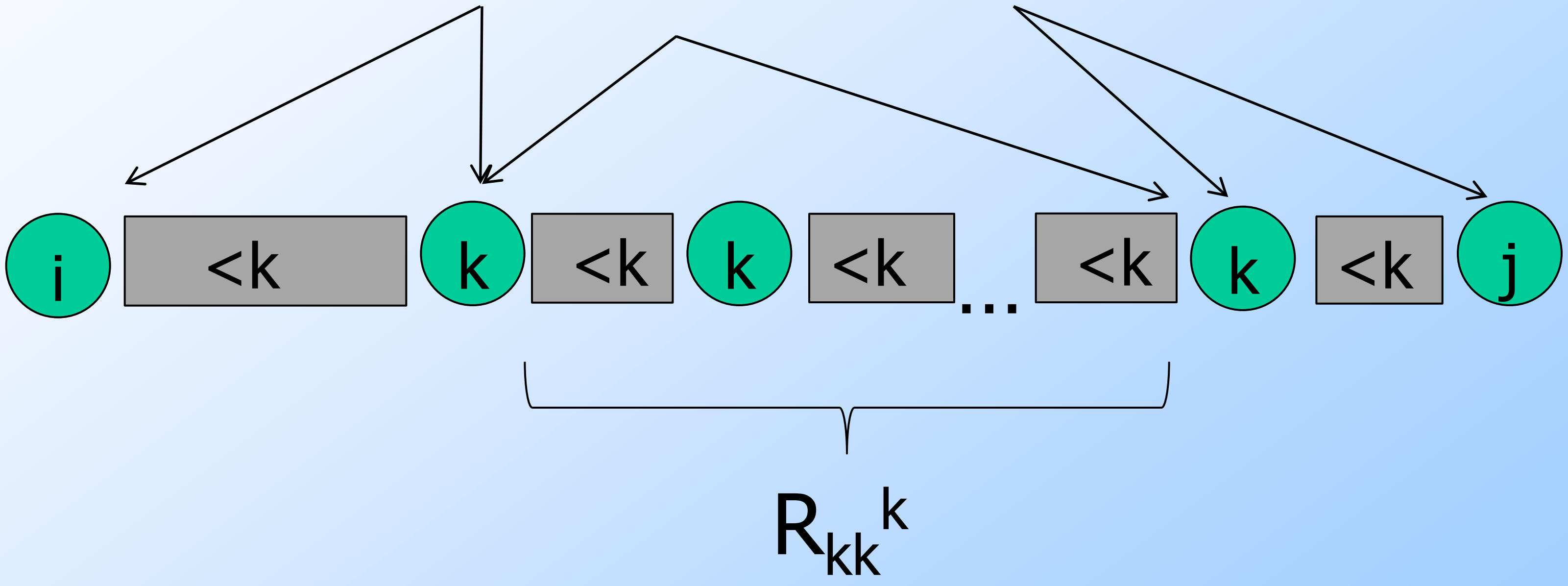
Goes from i to k the first time

Zero or more times from k to k

Then, from k to j

Zoom in

- What is $R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$?



$(R_{kk}^{k-1})^* = R_{kk}^{k}$

Challenge Problem 1

- L is a language with alphabet {0, 1, 2}
- L contains no strings that have any three consecutive 0's, any three consecutive 1's, or any three consecutive 2's.
  - e.g. 11000220 is not in L
- Prove that L is regular and give a DFA for L.

Challenge Problem 1

- The complement of L has a regular expression: (0+1+2)*(000+111+222)(0+1+2)*
  - All strings that DO contain three consecutive 0's or 1's or 2's.

- Regular languages are closed under complement

Challenge Problem 1

- DFA for L
  - The state represents the run of the same symbol that appears at the end of the string
    - Start state S
    - State a_0, a_{00}, a_1, a_{11}, a_2, a_{22}
    - Dead state D
- Example:
  - 012011 should get to a_{11}

Challenge Problem 1

	0	1	2
$\rightarrow$*S	a_0	a_1	a_2
*a_0	a_{00}	a_1	a_2
*a_1	a_0	a_{11}	a_2
*a_2	a_0	a_1	a_{22}
*a_{00}	D	a_1	a_2
*a_{11}	a_0	D	a_2
*a_{22}	a_0	a_1	D
D	D	D	D

Problem Session 3

Half(L)

Things More Powerful Than a Turing Machine

Some Concerns About Proofs

Half(L)

- If L is any language, Half(L) is the set of strings w such that for some string x, where |x| = |w|, wx is in L.
- If L is regular, so is Half(L).
- Construction: given a DFA A for L, we construct an ϵ-NFA B for Half(L).

Construction of NFA B

- States = pairs of states [p,q] of A, plus additional start state s_0.
- Intuition: If B reads input w, then $p = \delta_A(q_0, w)$.
  - q_0 = start state of A.
- q is any state such that there is some string x, with $|x| = |w|$ such that $\delta_A(q, x)$ is an accepting state.

Accepting States of B

- Those pairs of the form [q, q].
- Notice: If B is in a state [q, q], then it has read some input w, such that $\delta_A(q_0, w) = q$ and there is some input x with $|x| = |w|$, such that $\delta_A(q, x)$ is an accepting state.
- That means wx is in L(A), and w is the first half of wx.

Transitions of B

- $\delta_B(s_0, \epsilon) = \{[q_0, f] \mid f$ is an accepting state of A$\}$.
- B never returns to s_0.
- First move guarantees that B is in the correct state after having read no input.
  - Notice: $[q_0, q_0]$ is an accepting state of B if and only if ϵ is in L(A).

Transitions of B – (2)

- $\delta_B([p,q], a) = \{[r, s]$ | such that:

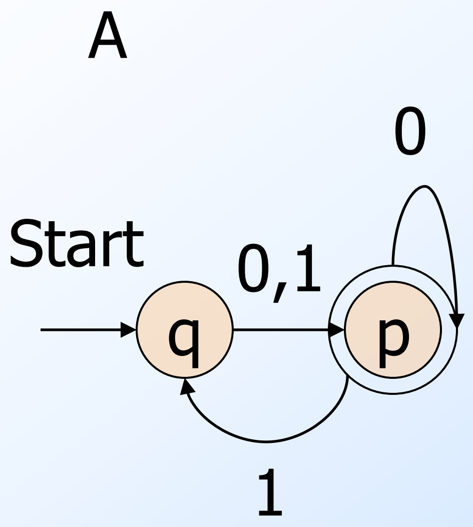
1. $\delta_A(p, a) = r$.
2. There is some input symbol b such that $\delta_A(s, b) = q\}$.

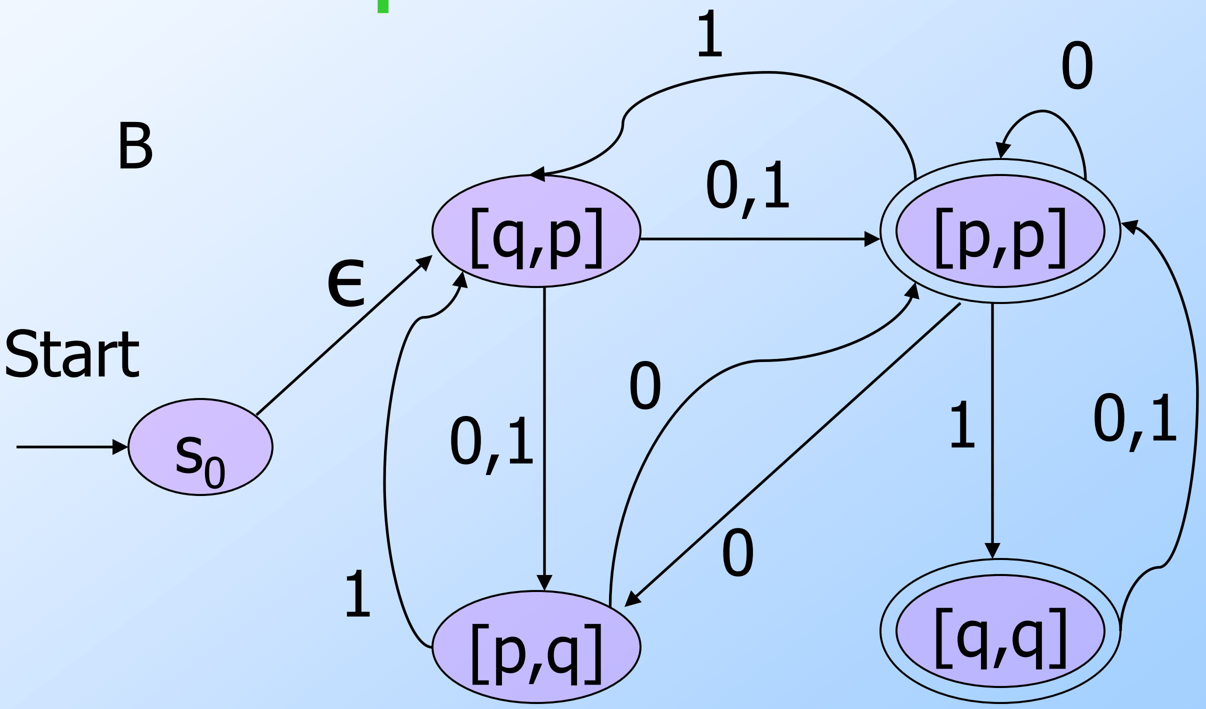
- (1) guarantees the first component continues to track the state of A.
- (2) guarantees the second component is any state that leads to acceptance via some string of length equal to input so far.

Inductive Proof That This Works

- By induction on $|w|$: $\delta_B([q_0, f], w) = \{[p, q] \mid$ such that:
  1. $\delta_A(q_0, w) = p$, and
  2. For some x, with $|x| = |w|$, $\delta_A(q, x) = f\}$.
- Complete the proof by observing the initial transitions out of s_0 to $[q_0, f]$, for accepting states f, and the definition of the accepting states of B.

Example





Is … More Powerful Than a Turing Machine?

- From an early post: "Can aspect systems do anything a Turing machine can't?"
- I don't know what an "aspect system" is.
- But if it is something that runs on a computer, then no.
- Why? because a Turing machine can simulate a real computer, and hence anything that runs on one.

What About Quantum Computers?

- People have imagined that there will be quantum computers that behave something like nondeterministic computers.
- There has been some progress by physicists on communication via quantum effects.

Quantum Computers – (2)

- The physics of quantum computers is suspect.
  - These would have to be enormous to isolate different bits of storage.
- But even if you had a quantum computer, it could still be simulated by a nondeterministic TM, and thus by a deterministic TM.

Can One PDA Stack Simulate Two?

- I claimed one could not, but I never proved it.
- If you try, you can't but that's no proof.
- Precise definition needed: A construction whereby one PDA P is constructed from two others, P_1 and P_2, so P accepts the intersection of the languages of P_1 and P_2.

Proof

- Assume such a construction exists.
- Let P_1 be a PDA that accepts $\{0^i1^j2^k \mid i=j\geq 1, k\geq 1\}$ and let P_2 be a PDA that accepts $\{0^i1^j2^k \mid j=k\geq 1, i\geq 1\}$.
- Then P would accept $L = \{0^i1^i2^i \mid i \geq 1\}$.
- But we know L is not a CFL, therefore has no PDA accepting it.

Proof – Continued

- We assumed only one thing: that we could construct P from P_1 and P_2.
- Since the conclusion, that L is a CFL, is known to be false, the assumption must be false.
- That is, no such construction exists.

Behind the Curtains of the Proof

- First, we assumed that if a statement S implies something false, then S is false.
- That seems to make sense, but it has to be an axiom of logic.
- Why? "proof" would be "by contradiction," thus using itself in its proof.
  - Aside: similarly, a "proof" that induction works requires an inductive proof.

Behind the Curtains – (2)

- We also made another assertion: the assumption "you can simulate two stacks with one" was the only unproved part of the proof, and therefore at fault.
  - Argument used many times in the course.
- But there were many other steps, some glossed over or left for your imagination.

Proofs as a Social Process

- If there were another unproved point, then my proof of "one stack can't simulate two" would not be valid.
- But proofs are subject to discussion and argument.
- If someone has a point they doubt, they can bring it up and it will be resolved one way or the other.

Aside: Social Processes – (2)

- Many years ago, Alan Perlis, Rich DeMillo and Dick Lipton published a paper arguing:
  - Proofs can only be believed because smart mathematicians will examine them and find flaws if they exist.
  - Proofs of program correctness are boring, and no one will bother to examine them.

Problem Session 4

Solving Instances Vs. Solving Problems

How to Market Your P=NP Solution

Telling What a Program Does

- Suppose I write a program to sort integers.
- Can't I feed it a list of integers and see whether they come out sorted?
  - Or a million lists and check them all?
- Yes, but maybe your program will fail to sort the 1,000,001th list you try.

Solving Instances Vs. Solving Problems

- Instances of a problem are not problems.
- Suppose TM M accepts language L, and w is a string.
  - One TM M_{yes} ignores its input and accepts.
  - Another TM M_{no} ignores its input and rejects.
- One answers the question "is w in L?"
- But I can't tell which.

Polytime Algorithms for Part of an NP-complete problem

- Suppose I have a polytime algorithm that works correctly on all tested instances of an NP-complete problem.
- Could I sell that solution?
  - Could I keep it secret so only I could solve NP-complete problems in polytime?

Zero-Knowledge Proofs

- Developed by Shafi Goldwasser, Silvio Micali, and Charles Rackoff in 1985.
  - An early idea in cryptographic protocols.
- Let’s you prove that you know something without revealing how you know it.
  - Hypothetical polytime algorithm for an NP-complete problem was key motivation.

Polytime Algorithm for Instances of a Problem

- It is possible to find polytime algorithms that solve instances of NP-complete problems.
  - Or even a million instances.
- But as for the analogous matter of decidability, it doesn't help us.

Testing Polytime Solutions

- What about a polytime “solution” to SAT that we test on 1,000,000 inputs and it gives the correct answer each time.
- How do you know it is correct?
- If a satisfying assignment exists, we could expect the tester to show us one.
  - Which we could then check.

Testing Polytime Solutions – (2)

- But what if the algorithm says “no”?
- We can’t even check that it is correct in less than exponential time.
- But if the expression has many satisfying assignments, we could try a few at random.
- Alas, expressions with one satisfying assignment exist.
  - Consider Cook’s construction applied to a DTM.

Thanks

- Thanks for being part of this course.
- Good luck on the final exam.

Discussion of Common Problems – 1

Types

Language an Automaton Accepts

Sound of No Hands Clapping

Types

- Types or classes are not just important in programming; they are vital in mathematics and automata theory.
- We’ve seen distinctions among types:
  - Characters vs. strings.
  - Sets vs. elements.

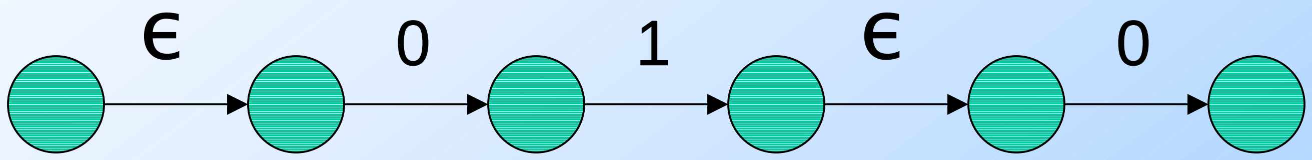
Strings Vs. Characters

- A common and important distinction in programming languages.
  - ”a” is of type String; ’a’ is of type char.
- Especially important: ϵ is of type String.
- Oddity: in the ϵ-NFA, we see arcs labeled by strings (ϵ in particular) and by characters (ordinary inputs like 0).

Strings Vs. Characters – (2)

- It’s not a problem; characters can be coerced to strings.
  - Example: in Java ”” + ’0’ = ”0”.
- Similarly, in an ϵ-NFA, you can mentally coerce the character labels to strings of length 1.
- And for any kind of finite automaton, labels of paths are strings.

Example: Label of a Path



Concatenation of the labels, each treated as a string is 010.

Sets Vs. Elements

- These are always different types.
- Especially, strings are elements, while sets of strings (e.g., languages) are sets.
- ϵ is a string.
- The empty set ∅ is a set.
- Sets can have “members”; elements never do.

Sets Vs. Elements – (2)

- The empty set is the only set in the world that does not have any members.
- Notice that strings like ϵ or 001 do not have members, but for a different reason:
  - Elements cannot have members.

States of an NFA

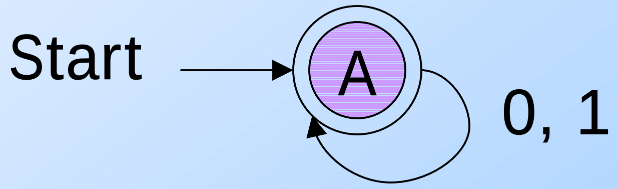
- States are always elements.
- The subset construction seems to construct DFA states that are sets of NFA states.
- Really, the DFA states *correspond* to sets of NFA states, but are elements with names like “Sally” or “q”.
  - Convenient to use something like {p, q} for the name of a DFA state.

Language of an Automaton

- Automata accept strings.
  - Labels of paths from the state state to an accepting state.
- They also accept languages.
  - EXACTLY the set of strings that the automaton accepts.
- Thus, many strings, but ONE language.

Fallacy

- We had a number of forum discussions where people took "automaton A accepts language L" to mean that all the strings of L are accepted by A.
- If that were the case, all languages would be "accepted" by:



Sound of No Hands Clapping

- People sometimes have trouble with the edge cases of general statements.
- Example: We know what the sum of integers is; but what if there are 0 integers?
- Example: we know what it means for a string to have an even number of 0's; but what if that string is empty?

Sum of Zero Integers

- We know what the sum of several integers is, e.g., 4 + 7 + 3.
- What is the sum of no integers?
- The only sensible choice is the identity for the operation +; i.e., 0.

Programming View

- If we wanted to sum integers a[i] for i = 0, 1,..., n-1, we would write, e.g.:

```
sum = 0;
for (i=0; i<n; i++) sum += a[i];
```

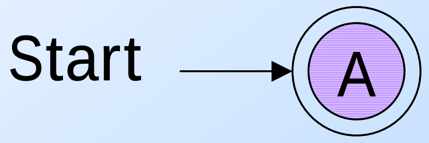
- What is the result if n=0? (Ans.: sum = 0).

Other Operators

- p OR q OR r? OR of zero propositions = false (the identity for OR).
- p AND q AND r? AND of zero propositions = true (the identity for AND).
- p*q*r? product of zero numbers = 1 (the identity for multiplication).
- ”w” ”x” ”y”? concatenation of zero strings = ϵ (the identity for concatenation).

Example: Concatenation of Zero Strings

- Suppose the start state is also accepting.



- The path from state A to A has, as label, the concatenation of zero strings.
- Implies that ϵ is accepted by this DFA.

Is 0 Odd or Even?

- Even, because the remainder of 0 divided by 2 is 0.
  - I.e., 0 = 2*0 + 0.
- The empty string has zero of every symbol.
- So ϵ has an even number of 0's an even number of 1's, and so on.

Automata – Not a Lady Automaton

- And let me add one more point – not of mathematics but of diction.
- "Automaton" is singular, and its plural is irregular: "automata."
- Oddly: the theory is called "automata theory," but other theories tend to be singular.
  - Examples: String theory, quantum theory.