Introduction to Automata Theory

What is Automata Theory?

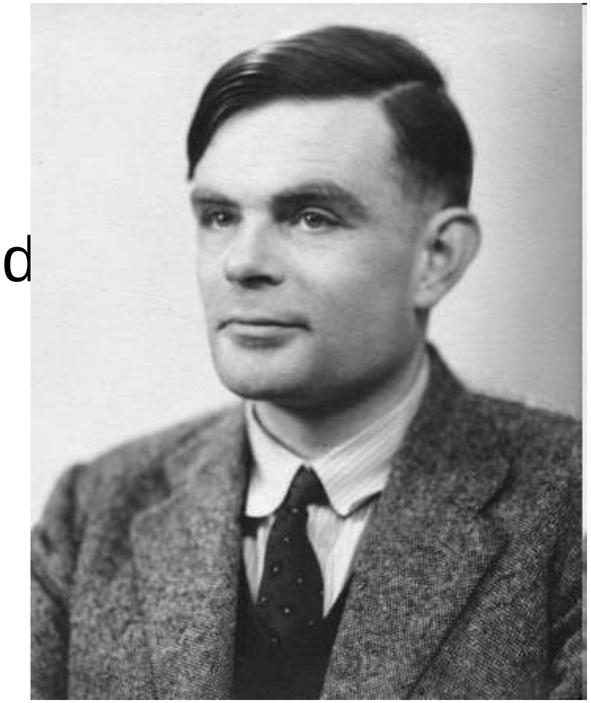
- *Study of abstract computing devices, or "machines"*
- Automaton = an abstract computing device
  - Note: A "device" need not even be a physical hardware!
- A fundamental question in computer science:
  - Find out what different models of machines can do and cannot do
  - The *theory of computation*
- Computability vs. Complexity

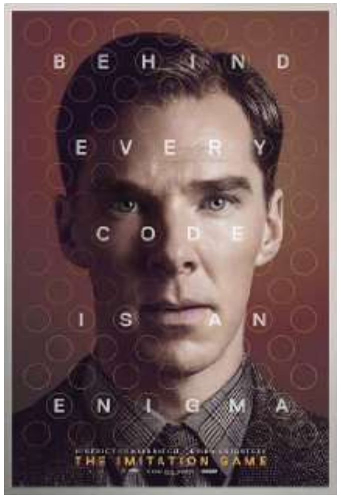
(A pioneer of automata theory)

Alan Turing (1912-1954)

- Father of Modern Computer Science
- English mathematician
- Studied abstract machines called ***Turing machines*** even before computers existed

Heard of the Turing test?

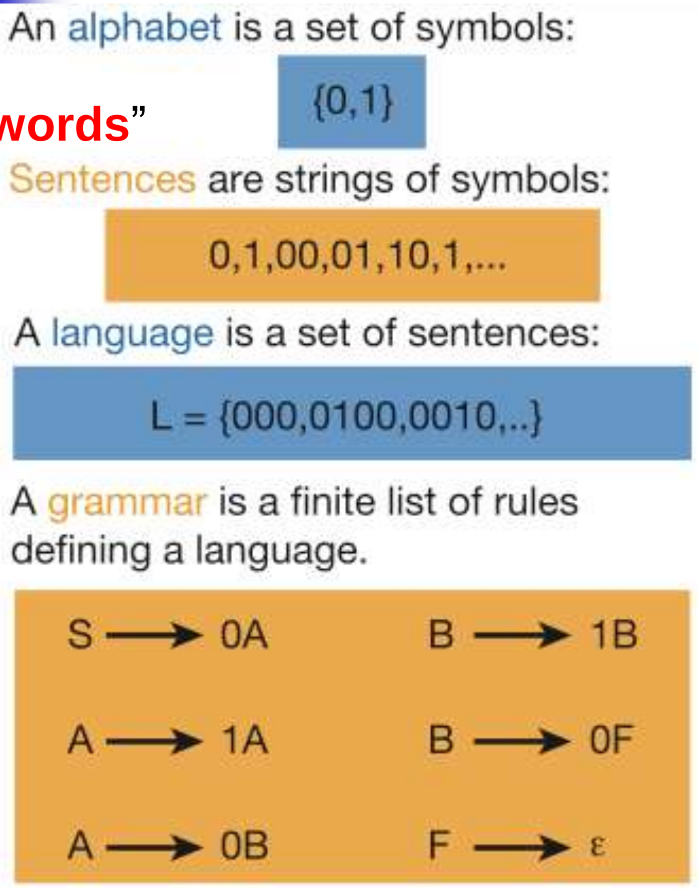




Theory of Computation: A Historical Perspective

1930s	• Alan Turing studies Turing machines • Decidability • Halting problem
1940-1950s	• “Finite automata” machines studied • Noam Chomsky proposes the “Chomsky Hierarchy” for formal languages
1969	Cook introduces “intractable” problems or “NP-Hard” problems
1970-	Modern computer science: compilers, computational & complexity theory evolve

Languages & Grammars



An alphabet is a set of symbols:

{0,1}

Or "**words**"

Sentences are strings of symbols:

0,1,00,01,10,1,...

A language is a set of sentences:

L = {000,0100,0010,..}

A grammar is a finite list of rules defining a language.

$S \longrightarrow 0A$

$A \longrightarrow 1A$

$A \longrightarrow 0B$

$B \longrightarrow 1B$

$B \longrightarrow 0F$

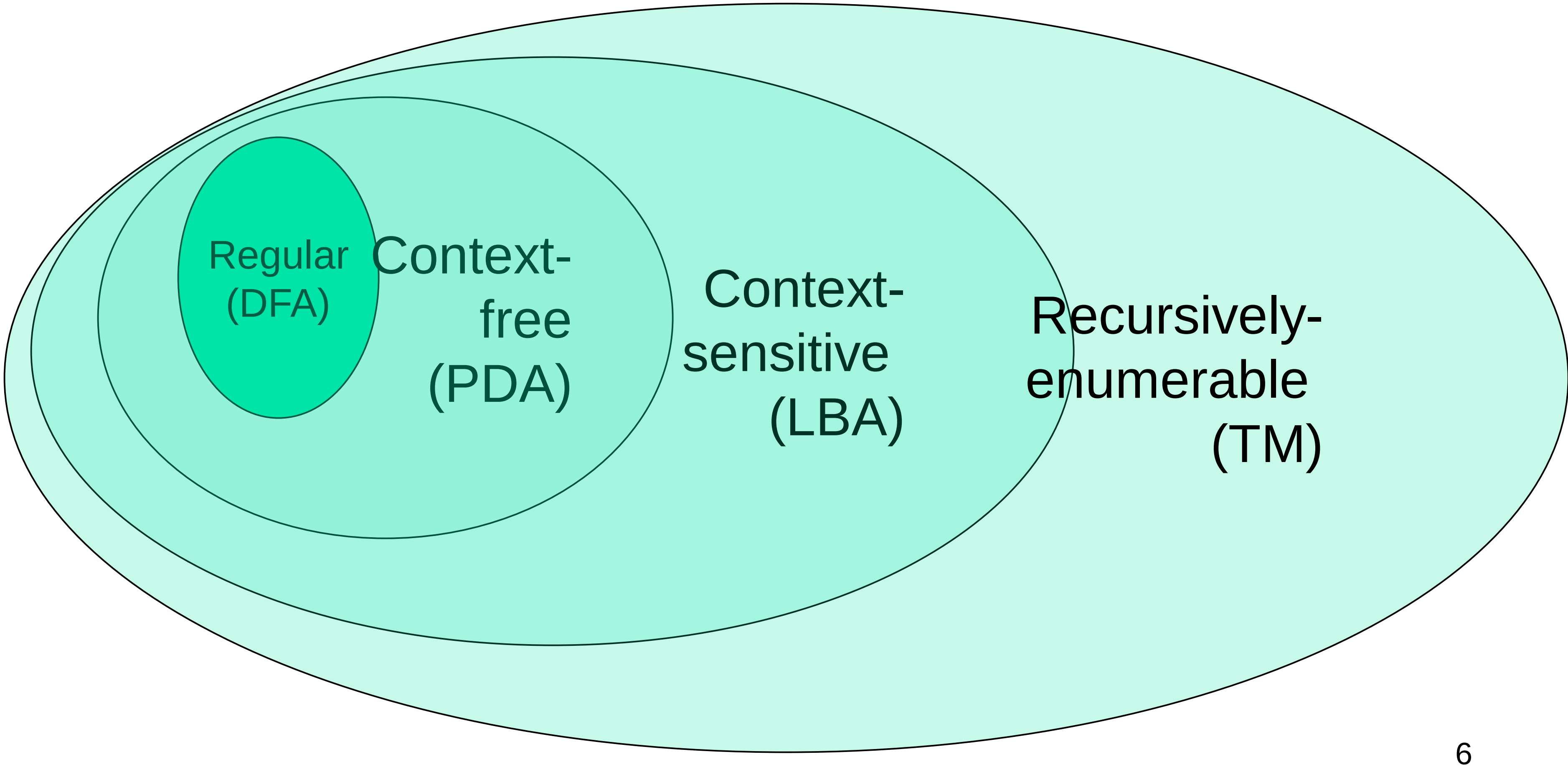
$F \longrightarrow \varepsilon$

Image source: Nowak et al. Nature, vol 417, 2002

- Languages: "*A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols*"
- Grammars: "*A grammar can be regarded as a device that enumerates the sentences of a language*" - nothing more, nothing less
- *N. Chomsky, Information and Control, Vol 2, 1959*

The Chomsky Hierachy

- A containment hierarchy of classes of formal languages



The Central Concepts of Automata Theory

Alphabet

An alphabet is a finite, non-empty set of symbols

- We use the symbol ∑ (sigma) to denote an alphabet
- Examples:
  - Binary: ∑ = {0,1}
  - All lower case letters: ∑ = {a,b,c,..z}
  - Alphanumeric: ∑ = {a-z, A-Z, 0-9}
  - DNA molecule letters: ∑ = {a,c,g,t}
  - …

Strings

A string or word is a finite sequence of symbols chosen from ∑

- ***Empty string is ε (or “epsilon”)***

- Length of a string *w*, denoted by “|*w*|”, is equal to the *number of (non- ε) characters in the string*
  - *E.g., x = 010100* |*x*| = *6*
  - *x = 01 ε 0 ε 1 ε 00 ε* |*x*| = *?*

- *xy* = concatentation of two strings *x* and *y*

Powers of an alphabet

Let Σ be an alphabet.

- Σ^{k} = the set of all strings of length k
- $\Sigma^{*} = \Sigma^{0}$ U Σ^{1} U Σ^{2} U …
- $\Sigma^{+} = \Sigma^{1}$ U Σ^{2} U Σ^{3} U …

Languages

*L is a said to be a language over alphabet ∑, only if L ⊆ ∑**

➔ this is because ∑* is the set of all strings (of all possible length including 0) over the given alphabet ∑

Examples:

1. Let L be *the* language of all strings consisting of *n* 0's followed by *n* 1's:

   L = {ε, 01, 0011, 000111,…}

2. Let L be *the* language of all strings of with equal number of 0's and 1's:

   L = {ε, 01, 10, 0011, 1100, 0101, 1010, 1001,…}

   →

   Canonical ordering of strings in the language

Definition: Ø denotes the Empty language

- Let L = {ε}; Is L=Ø?

NO

The Membership Problem

Given a string w $\in \Sigma^*$and a language L over Σ, decide whether or not w $\in$L.

Example:

Let w = 100011

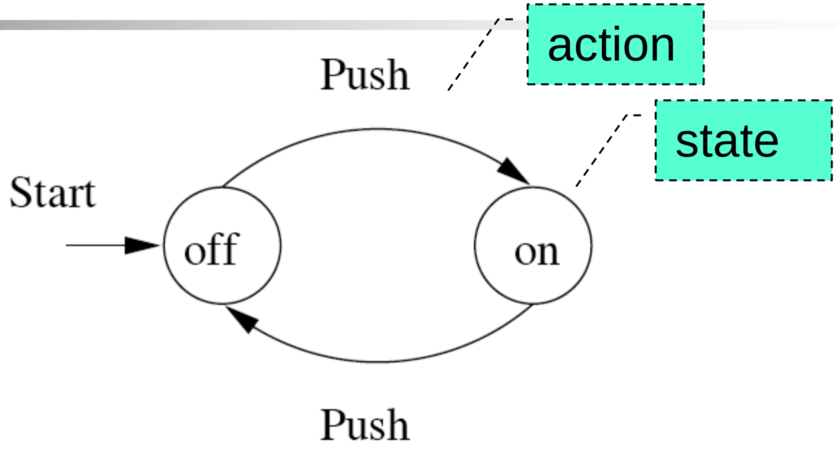
Q) Is w $\in$ the language of strings with equal number of 0s and 1s?

Finite Automata

- Some Applications
  - Software for designing and checking the behavior of digital circuits
  - Lexical analyzer of a typical compiler
  - Software for scanning large bodies of text (e.g., web pages) for pattern finding
  - Software for verifying systems of all types that have a finite number of states (e.g., stock market transaction, communication/network protocol)

Finite Automata : Examples

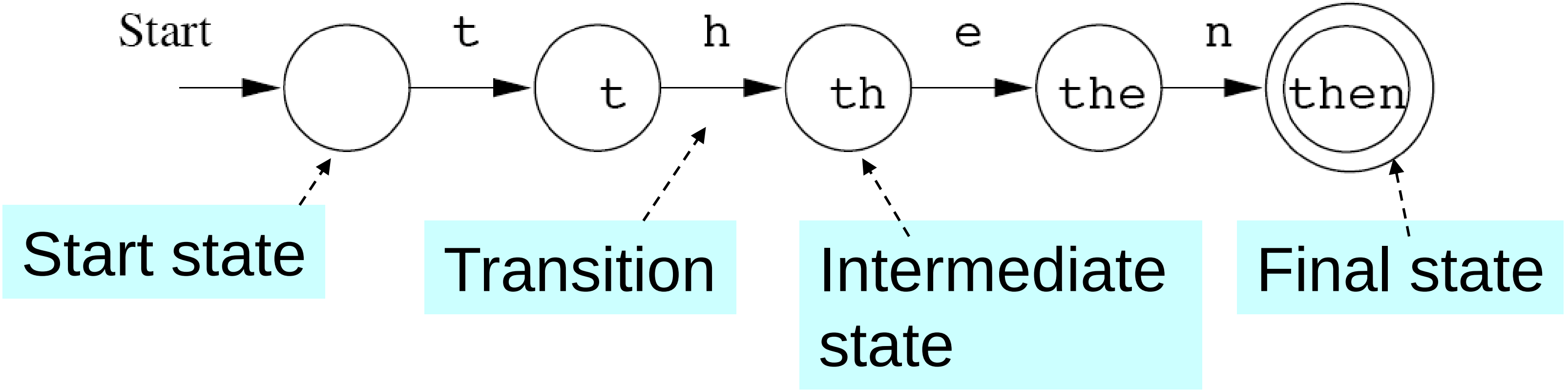
- On/Off switch



action

state

- Modeling recognition of the word “*then*”



Start state

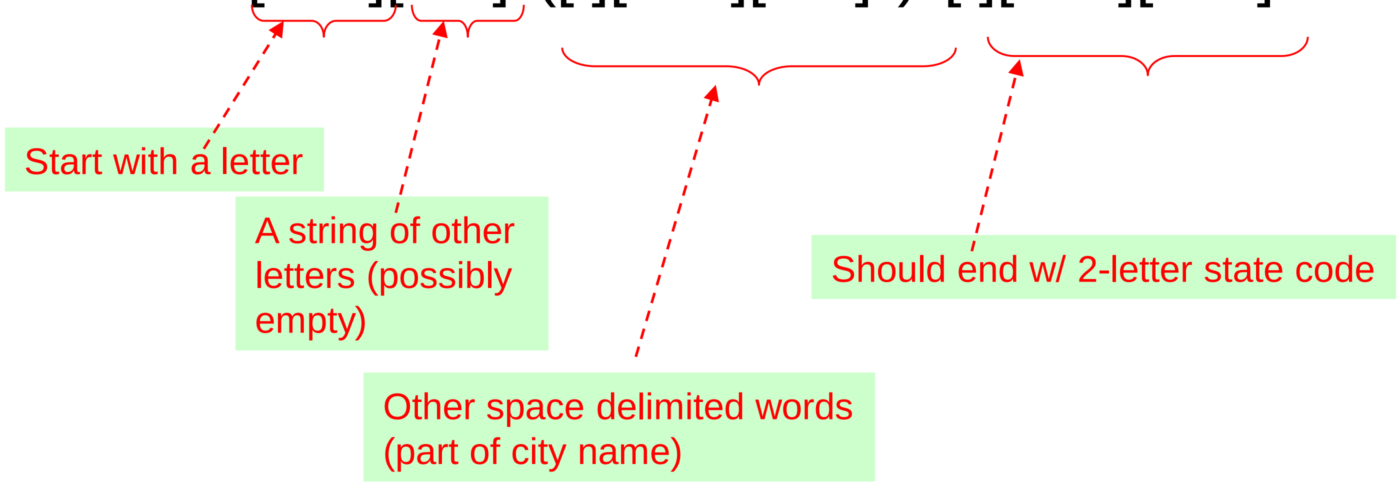
Transition

Intermediate state

Final state

Structural expressions

- Grammars
- Regular expressions
  - E.g., unix style to capture city names such as “Palo Alto CA”:
    - [A-Z][a-z]*([ ][A-Z][a-z]*)*[ ][A-Z][A-Z]



Start with a letter

A string of other letters (possibly empty)

Other space delimited words (part of city name)

Should end w/ 2-letter state code

Formal Proofs

Deductive Proofs

From the given statement(s) to a conclusion statement (what we want to prove)

- Logical progression by direct implications

Example for parsing a statement:

- “If $y \geq 4$, then $2^y \geq y^2$.”

given

conclusion

(there are other ways of writing this).

Example: Deductive proof

***Let Claim 1:** If $y \ge 4$, then $2^y \ge y^2$.*

Let x be any number which is obtained by adding the squares of 4 positive integers.

Claim 2:

Given x and assuming that Claim 1 is true, prove that $2^x \ge x^2$

- Proof:
  1) Given: $x = a^2 + b^2 + c^2 + d^2$
  2) Given: $a \ge 1, b \ge 1, c \ge 1, d \ge 1$
  3) ➔ $a^2 \ge 1, b^2 \ge 1, c^2 \ge 1, d^2 \ge 1$ (by 2)
  4) ➔ $x \ge 4$ (by 1 & 3)
  5) ➔ $2^x \ge x^2$ (by 4 and Claim 1)

"implies" or *"follows"*

On Theorems, Lemmas and Corollaries

We typically refer to:

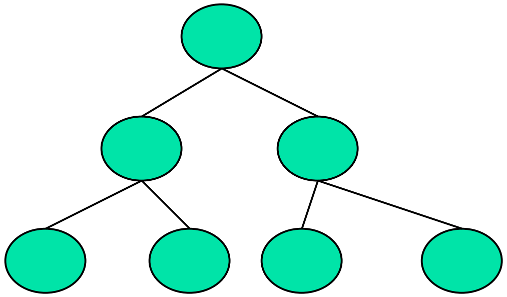
- A major result as a “**theorem**”
- An intermediate result that we show to prove a larger result as a “**lemma**”
- A result that follows from an already proven result as a “**corollary**”

An example:

Theorem: *The height of an n-node binary tree is at least floor(lg n)*

Lemma: *Level i of a perfect binary tree has 2^i nodes.*

Corollary: *A perfect binary tree of height h has 2^{h+1}-1 nodes.*



Quantifiers

"For all" or "*For every"*

- Universal proofs
- Notation= $\forall$

"There exists"

- Used in existential proofs
- Notation= $\exists$

Implication is denoted by =>

- E.g., "IF A THEN B" can also be written as "A=>B"

Proving techniques

- By contradiction
  - Start with the statement contradictory to the given statement
  - E.g., To prove (A => B), we start with:
    - (A and ~B)
    - … and then show that could never happen

    What if you want to prove that "(A and B => C or D)"?

- By induction
  - (3 steps) Basis, inductive hypothesis, inductive step
- By contrapositive statement
  - If *A* then *B* ≡ If ~*B* then ~*A*

Proving techniques…

- By counter-example
  - Show an example that disproves the claim

- Note: There is no such thing called a “proof by example”!
  - So when asked to prove a claim, an example that satisfied that claim is *not* a proof

Different ways of saying the same thing

- “*If* H *then* C”:
  1. H *implies* C
  2. *H => C*
  3. C *if* H
  4. H *only if* C
  5. *Whenever* H *holds*, C *follows*

“*If-and-Only-If*” statements

- “A if and only if B”   (A <==> B)
  - *(if part)* if B then A   (<=)
  - *(only if part)* A only if B   (=>)
    (same as “if A then B”)
- “If and only if” is abbreviated as “iff”
  - i.e., “A iff B”
- Example:
  - Theorem: *Let x be a real number. Then floor of x = ceiling of x if and only if x is an integer.*
- Proofs for iff have two parts
  - One for the “if part” & another for the “only if part”

Summary

- Automata theory & a historical perspective
- Chomsky hierarchy
- Finite automata
- Alphabets, strings/words/sentences, languages
- Membership problem
- Proofs:
    - Deductive, induction, contrapositive, contradiction, counterexample
    - If and only if

Finite Automata

Finite Automaton (FA)

- Informally, a state diagram that comprehensively captures all possible states and transitions that a machine can take while responding to a stream or sequence of input symbols
- Recognizer for “Regular Languages”

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

Deterministic Finite Automata - Definition

- A Deterministic Finite Automaton (DFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of accepting states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> Q
- A DFA is defined by the 5-tuple:
  - {Q, ∑ , q_0,F, δ }

What does a DFA do on reading an input string?

- Input: a word w in ∑*
- Question: Is w acceptable by the DFA?
- Steps:
  - Start at the “start state” q_0
  - For every input symbol in the sequence w do
    - Compute the next state from the current state, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed, the current state is one of the accepting states (F) then *accept w;*
  - Otherwise, *reject w.*

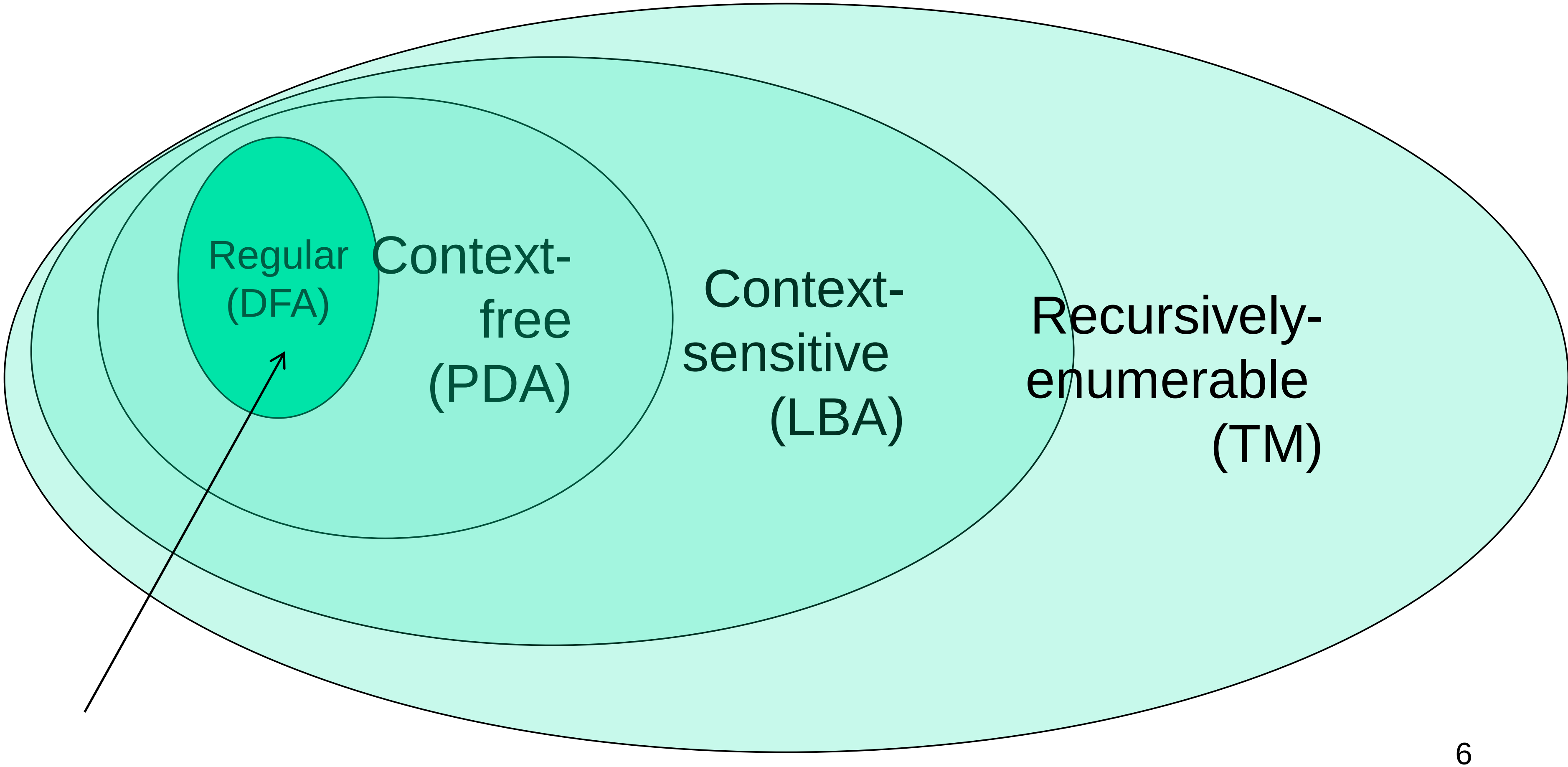
Regular Languages

- Let L(A) be a language *recognized* by a DFA A.
    - Then L(A) is called a "*Regular Language*".

- Locate regular languages in the Chomsky Hierarchy

The Chomsky Hierachy

- A containment hierarchy of classes of formal languages



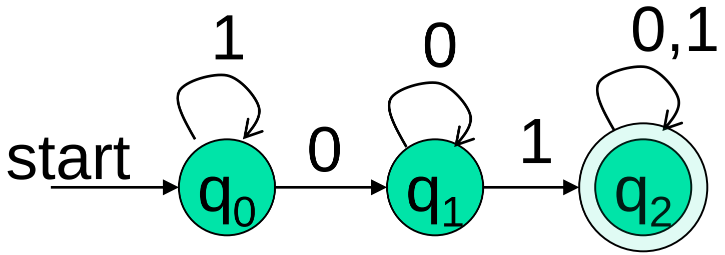
Example #1

- Build a DFA for the following language:
  - L = {w | w is a binary string that contains 01 as a substring}
- Steps for building a DFA to recognize L:
  - ∑ = {0,1}
  - Decide on the states: Q
  - Designate start state and final state(s)
  - δ: Decide on the transitions:
- "Final" states == same as "accepting states"
- Other states == same as "non-accepting states"

Regular expression: (0+1)*01(0+1)*

DFA for strings containing 01

- What makes this DFA deterministic?



Accepting state

- What if the language allows empty strings?

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols

states

δ	**0**	**1**
→**q_0**	q_1	q_0
q_1	q_1	q_2
***q_2**	q_2	q_2

Example #2

Clamping Logic:

- A clamping circuit waits for a ”1” input, and turns on forever. However, to avoid clamping on spurious noise, we’ll design a DFA that waits for *two consecutive 1s* in a row before clamping on.

- Build a DFA for the following language:
  L = { w | w is a bit string which contains the substring 11}
- State Design:
  - q_0 : start state (initially off), also means the most recent input was not a 1
  - q_1: has never seen 11 but the most recent input was a 1
  - q_2: has seen 11 at least once

Example #3

- Build a DFA for the following language:
  L = { w | w is a binary string that has even number of 1s and even number of 0s}
- ?

Extension of transitions (δ) to Paths ($\hat{\delta}$)

- $\hat{\delta}\ (q,w)$ = *destination state* from state q on input string w

- $\hat{\delta}\ (q,wa) = \delta\ (\hat{\delta}(q,w),\ a)$

  - Work out example #3 using the input sequence w=10010, a=1:

    - $\hat{\delta}\ (q_0,wa) = ?$

Language of a DFA

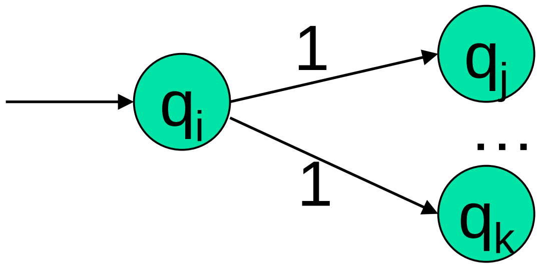
A DFA A accepts string *w* if there is a path from q_0 to an accepting (or final) state that is labeled by *w*

- *i.e., L(A) = { w | $\hat{\delta}(q_0,w) \in F$ }*

- *I.e., L(A) = all strings that lead to an accepting state from q_0*

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA)
  - is of course "non-deterministic"
    - Implying that the machine can exist in more than one state at the same time
    - Transitions could be non-deterministic



• Each transition function therefore maps to a set of states

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of accepting states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> subset of Q
- An NFA is also defined by the 5-tuple:
  - {Q, ∑ , q_0,F, δ }

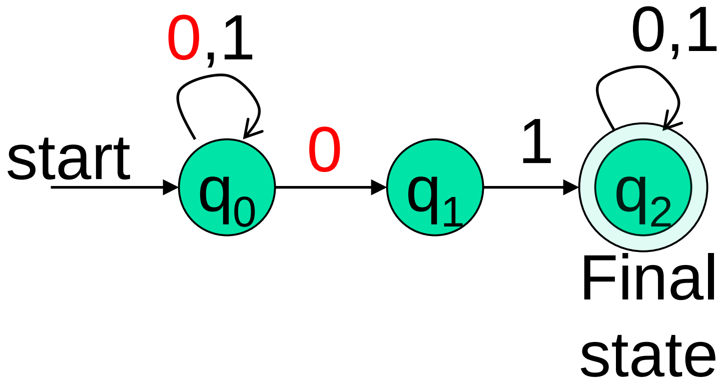
How to use an NFA?

- Input: a word w in ∑*
- Question: Is w acceptable by the NFA?
- Steps:
  - Start at the “start state” q_0
  - For every input symbol in the sequence w do
    - Determine all possible next states from all current states, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed and if at least one of the current states is a final state then *accept w;*
  - Otherwise, *reject w.*

Regular expression: (0+1)*01(0+1)*

NFA for strings containing 01

Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

- $Q = \{q_0,q_1,q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols

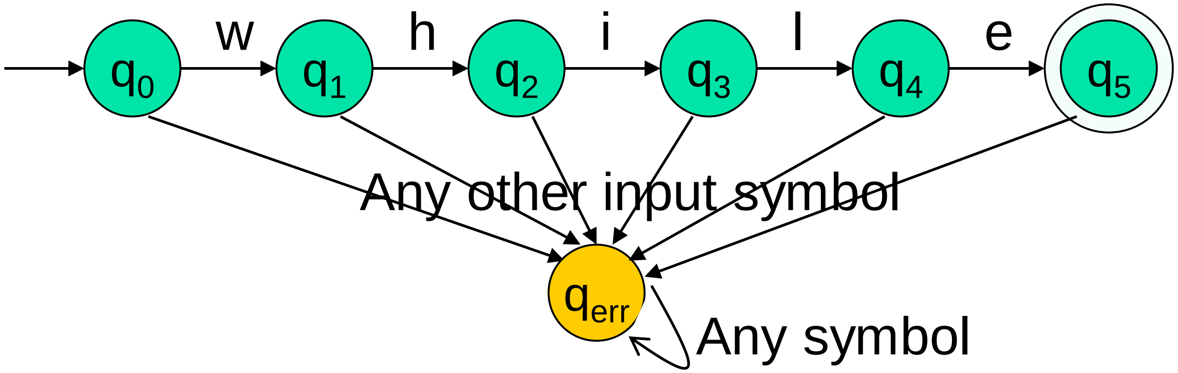
states

δ	**0**	**1**
→**q_0**	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
***q_2**	$\{q_2\}$	$\{q_2\}$

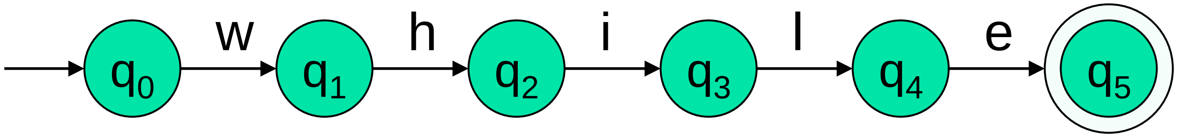
Note: Omitting to explicitly show error states is just a matter of design convenience
(one that is generally followed for NFAs), and
i.e., this feature should not be confused with the notion of non-determinism.

What is an "error state"?

- A DFA for recognizing the key word "*while*"



- An NFA for the same purpose:



Transitions into a dead state are implicit

Example #2

- Build an NFA for the following language:

  L = { w | w ends in 01}

- ?
- Other examples
  - Keyword recognizer (e.g., if, then, else, while, for, include, etc.)
  - Strings where the first symbol is present somewhere later on at least once

Extension of δ to NFA Paths

- Basis: $\hat{\delta}\,(q,\varepsilon) = \{q\}$

- Induction:
  - Let $\hat{\delta}\,(q_0,w) = \{p_1,p_2\ldots,p_k\}$
  - $\delta\,(p_i,a) = S_i$ for $i=1,2\ldots,k$
  - *Then,* $\hat{\delta}\,(q_0,wa) = S_1 \cup S_2 \cup \ldots \cup S_k$

Language of an NFA

- An NFA accepts *w* if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by *w*
- $L(N) = \{ w \mid \hat{\delta}(q_0,w) \cap F \neq \Phi \}$

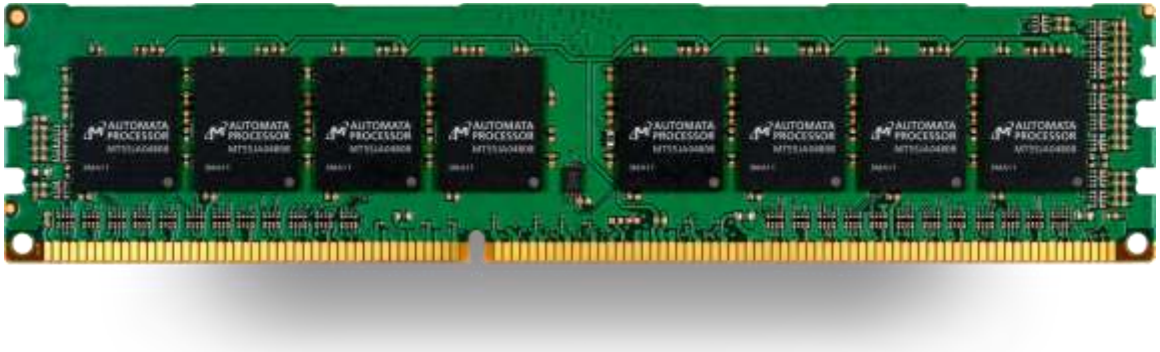
Advantages & Caveats for NFA

- Great for modeling regular expressions
  - String processing - e.g., grep, lexical analyzer

- Could a non-deterministic state machine be implemented in practice?
  - Probabilistic models could be viewed as extensions of non-deterministic state machines
    (e.g., toss of a coin, a roll of dice)
    - They are not the same though
  - A parallel computer could exist in multiple “states” at the same time

Technologies for NFAs

- Micron’s Automata Processor (introduced in 2013)
- 2D array of MISD (multiple instruction single data) fabric w/ thousands to millions of processing elements.
- 1 input symbol = fed to all states (i.e., cores)
- Non-determinism using circuits
- http://www.micronautomata.com/



But, DFAs and NFAs are equivalent in their power to capture langauges !!

Differences: DFA vs. NFA

- **DFA**

1. All transitions are deterministic
   - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state visited is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

- **NFA**

1. Some transitions could be non-deterministic
   - A transition could lead to a subset of states
2. Not all symbol transitions need to be defined explicitly (if undefined will go to an error state – this is just a design convenience, not to be confused with “non-determinism”)
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementations limited but emerging (e.g., Micron automata processor)

Equivalence of DFA & NFA

- Theorem:
  - A language L is accepted by a DFA *if and only if* it is accepted by an NFA.

Should be true for any L

- Proof:

1. If part:
   - Prove by showing every NFA can be converted to an equivalent DFA (in the next few slides…)

2. Only-if part is trivial:
   - Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA. □

Proof for the if-part

- If-part: A language L is accepted by a DFA if it is accepted by an NFA
- rephrasing…
- Given any NFA N, we can construct a DFA D such that L(N)=L(D)

- How to convert an NFA into a DFA?
  - Observation: In an NFA, each transition maps to a *subset* of states
  - Idea: Represent:

    each "subset of NFA_states" ➔ a single "DFA_state"

Subset construction

NFA to DFA by subset construction

- Let N = $\{Q_N, \sum, \delta_N, q_0, F_N\}$
- Goal: Build D=$\{Q_D, \sum, \delta_D, \{q_0\}, F_D\}$ s.t. L(D)=L(N)
- Construction:

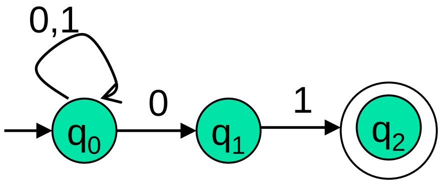
1. Q_D= all subsets of Q_N (i.e., power set)
2. F_D=set of subsets S of Q_N s.t. $S \cap F_N \neq \Phi$
3. δ_D: for each subset S of Q_N and for each input symbol a in $\sum$:
   - $\delta_D(S,a) = \bigcup_{p \text{ in } s} \delta_N(p,a)$

Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

- *L = {w | w ends in 01}*

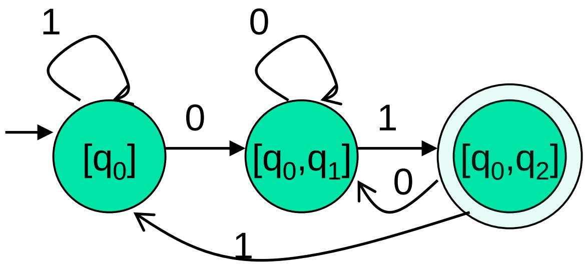
NFA:



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:

δ_D
~~Ø~~
→ $[q_0]$
~~$[q_1]$~~
~~$*[q_2]$~~
$[q_0,q_1]$
$*[q_0,q_2]$
~~$*[q_1,q_2]$~~
~~$*[q_0,q_1,q_2]$~~



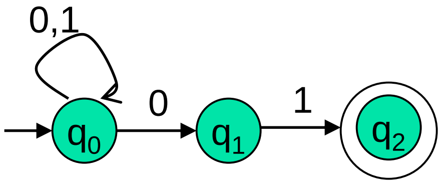
δ_D	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
$*[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

NFA to DFA: Repeating the example using *LAZY CREATION*

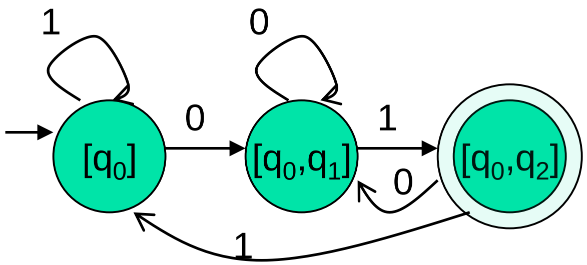
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:



δ_D	0	1
→$[q_0]$	$[q_0,q_1]$	$[q_0]$

Main Idea:

Introduce states as you go (on a need basis)

Correctness of subset construction

Theorem: If D is the DFA constructed from NFA N by subset construction, then L(D)=L(N)

- Proof:
  - Show that $\hat{\delta}_D(\{q_0\},w) \equiv \hat{\delta}_N(q_0,w\}$, for all w
  - Using induction on w's length:
    - Let w = xa
    - $\hat{\delta}_D(\{q_0\},xa) \equiv \delta_D(\hat{\delta}_N(q_0,x\}, a) \equiv \hat{\delta}_N(q_0,w\}$

A bad case where #states(DFA)>>#states(NFA)

- L = {w | w is a binary string s.t., the kth symbol from its end is a 1}

  - NFA has k+1 states

  - But an equivalent DFA needs to have at least 2^k states

<u>(Pigeon hole principle)</u>

  - *m* holes and >*m* pigeons
    - => at least one hole has to contain two or more pigeons

Applications

- Text indexing
  - inverted indexing
  - For each unique word in the database, store all locations that contain it using an NFA or a DFA
- Find pattern P in text T
  - Example: Google querying
- Extensions of this idea:
  - PATRICIA tree, suffix tree

A few subtle properties of DFAs and NFAs

- The machine never really terminates.
  - It is always waiting for the next input symbol or making transitions.
- The machine decides when to consume the next symbol from the input and when to ignore it.
  - (but the machine can never skip a symbol)
- => A transition can happen even *without* really consuming an input symbol (think of consuming ε as a free token) – if this happens, then it becomes an ε-NFA (see next few slides).
- A single transition *cannot* consume more than one (non-ε) symbol.

FA with ε-Transitions

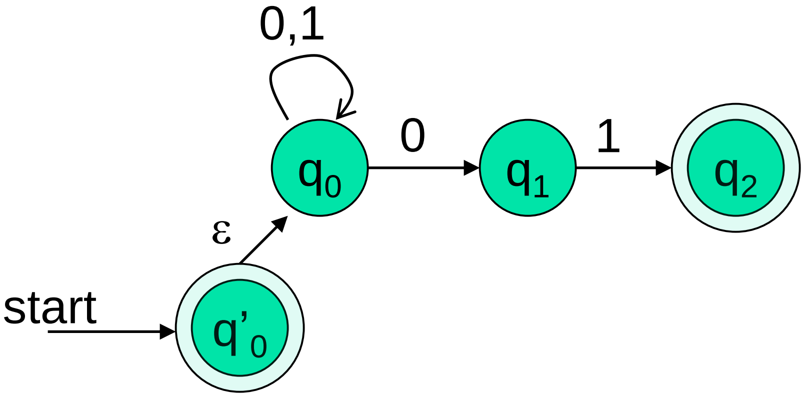
- We can allow explicit ε-transitions in finite automata
  - i.e., a transition from one state to another state without consuming any additional input symbol
  - Explicit ε-transitions between different states introduce non-determinism.
  - Makes it easier sometimes to construct NFAs

***Definition:* ε -NFAs are those NFAs with at least one explicit ε-transition defined.**

- ε -NFAs have one more column in their transition table

Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$

ECLOSE(q'_0)

ECLOSE(q_0)

ECLOSE(q_1)

ECLOSE(q_2)

- ε-closure of a state q, ***ECLOSE(q)***, is the set of all states (including itself) that can be *reached* from q by repeatedly making an arbitrary number of ε-transitions.

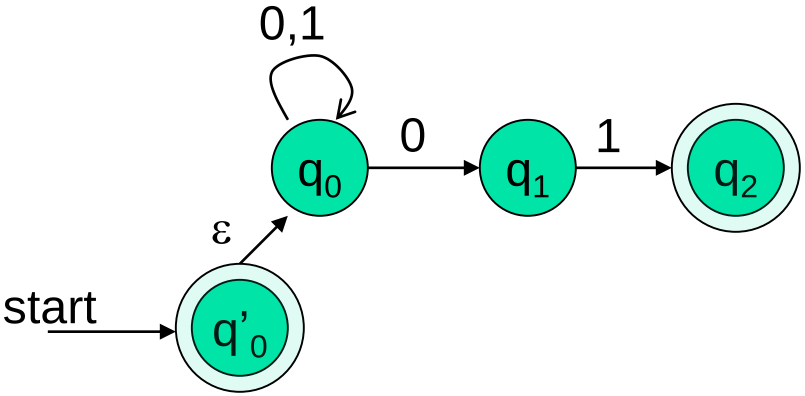
To simulate any transition:

Step 1) Go to all immediate destination states.

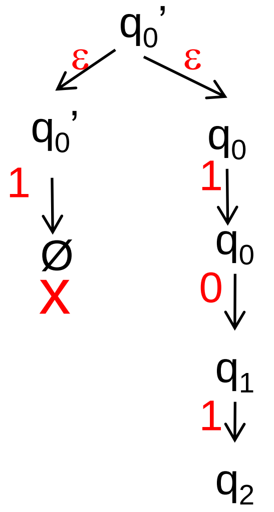
Step 2) From there go to all their ε-closure states as well.

Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



Simulate for w=101:



δ_E	0	1	ε
*q'_0	Ø	Ø	{q'_0,q_0}
q_0	{q_0,q_1}	{q_0}	{q_0}
q_1	Ø	{q_2}	{q_1}
*q_2	Ø	Ø	{q_2}

ECLOSE(q'_0)

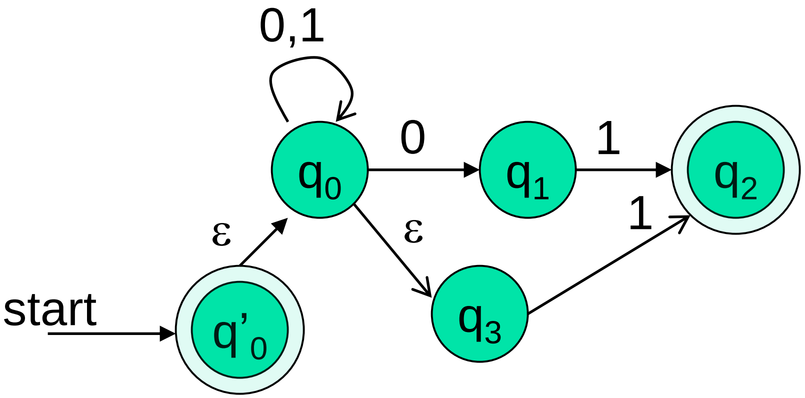
ECLOSE(q_0)

To simulate any transition:

Step 1) Go to all immediate destination states.

Step 2) From there go to all their ε-closure states as well.

Example of another ε-NFA



Simulate for w=101:

?

δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0,q_0,q_3\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0,q_3\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$
q_3	Ø	$\{q_2\}$	$\{q_3\}$

Equivalency of DFA, NFA, ε-NFA

- Theorem: A language L is accepted by some ε-NFA if and only if L is accepted by some DFA

- Implication:
  - DFA ≡ NFA ≡ ε-NFA
  - (all accept Regular Languages)

Eliminating ε-transitions

Let $E = \{Q_E,\Sigma,\delta_E,q_0,F_E\}$ be an ε-NFA

Goal: To build DFA $D=\{Q_D,\Sigma,\delta_D,\{q_D\},F_D\}$ s.t. $L(D)=L(E)$

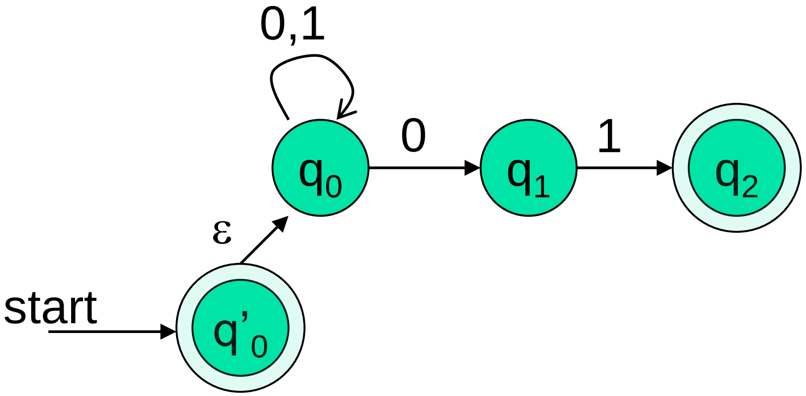
Construction:

1. Q_D= all reachable subsets of Q_E factoring in ε-closures
2. q_D = ECLOSE(q_0)
3. F_D=subsets S in Q_D s.t. $S \cap F_E \neq \Phi$
4. δ_D: for each subset S of Q_E and for each input symbol $a \in \Sigma$:
   - Let $R= \bigcup_{p \text{ in } s} \delta_E(p,a)$ // go to destination states
   - $\delta_D(S,a) = \bigcup_{r \text{ in } R}$ ECLOSE(r) // from there, take a union of all their ε-closures

Reading: Section 2.5.5 in book

Example: ε-NFA → DFA

L = {w | w is empty, or if non-empty will end in 01}

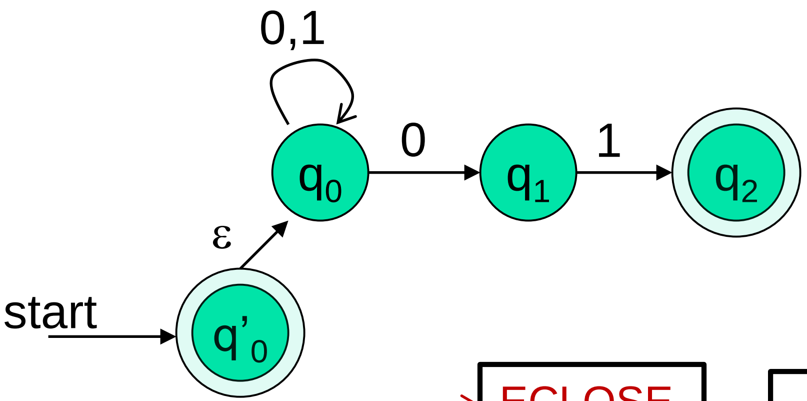


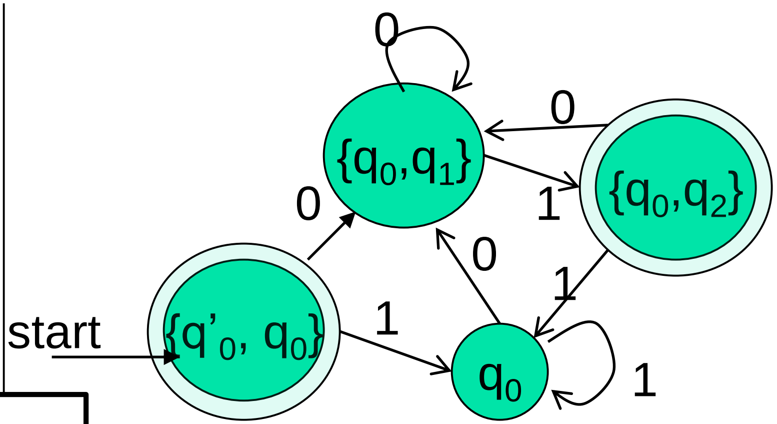
δ_E	0	1	ε
→ *q'_0	Ø	Ø	{q'_0,q_0}
q_0	{q_0,q_1}	{q_0}	{q_0}
q_1	Ø	{q_2}	{q_1}
*q_2	Ø	Ø	{q_2}

δ_D	0	1
→ *{q'_0,q_0}		
…		

Example: ε-NFA ➔ DFA

L = {w | w is empty, or if non-empty will end in 01}





ECLOSE

union

δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$

δ_D	0	1
→ $*\{q'_0,q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Summary

- DFA
  - Definition
  - Transition diagrams & tables
- Regular language
- NFA
  - Definition
  - Transition diagrams & tables
- DFA vs. NFA
- NFA to DFA conversion using subset construction
- Equivalency of DFA & NFA
- Removal of redundant states and including dead states
- ε-transitions in NFA
- Pigeon hole principles
- Text searching applications

Regular Expressions

Regular Expressions vs. Finite Automata

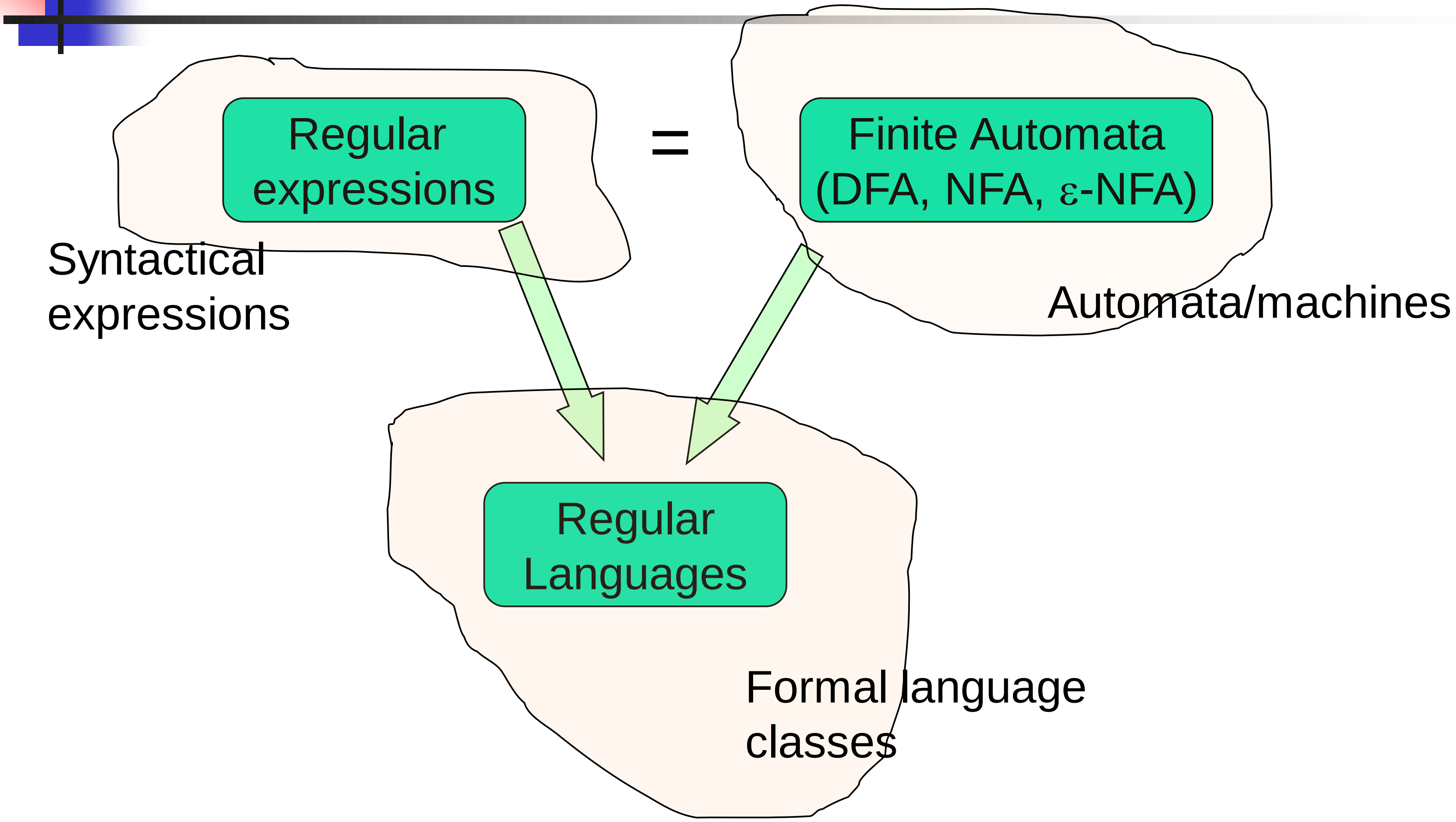
- Offers a declarative way to express the pattern of any string we want to accept
  - E.g., 01*+ 10*

- Automata => more machine-like

  < input: string , output: [accept/reject] >
- Regular expressions => more program syntax-like

- Unix environments heavily use regular expressions
  - E.g., bash shell, grep, vi & other editors, sed
- Perl scripting – good for string processing
- Lexical analyzers such as Lex or Flex

Regular Expressions



Regular expressions

=

Finite Automata (DFA, NFA, ε-NFA)

Syntactical expressions

Automata/machines

Regular Languages

Formal language classes

Language Operators

- Union of two languages:
  - **L U M** = all strings that are either in L or M
  - Note: A union of two languages produces a third language

- Concatenation of two languages:
  - **L . M** = all strings that are of the form *xy* s.t., x $\in$ L and y $\in$ M
  - The *dot* operator is usually omitted
    - i.e., **LM** is same as L.M

“i” here refers to how many strings to concatenate from the parent language L to produce strings in the language L^i

Kleene Closure (the * operator)

- Kleene Closure of a given language L:
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{w \mid \text{for some } w \in L\}$
  - $L^2 = \{ w_1 w_2 \mid w_1 \in L, w_2 \in L$ (duplicates allowed)$\}$
  - $L^i = \{ w_1 w_2 \ldots w_i \mid$ all w’s chosen are $\in L$ (duplicates allowed)$\}$
  - (Note: the choice of each w_i is independent)
  - $\mathbf{L^*} = \bigcup_{i \geq 0} L^i$ (arbitrary number of concatenations)

Example:

- Let L = { 1, 00}
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{1,00\}$
  - $L^2 = \{11,100,001,0000\}$
  - $L^3 = \{111,1100,1001,10000,000000,00001,00100,0011\}$
  - $\mathbf{L^*} = L^0 \cup L^1 \cup L^2 \cup \ldots$

Kleene Closure (special notes)

- L^* is an infinite set iff $|L| \geq 1$ and $L \neq \{\varepsilon\}$ **Why?**
- If $L=\{\varepsilon\}$, then $L^* = \{\varepsilon\}$ **Why?**
- If $L = \Phi$, then $L^* = \{\varepsilon\}$ **Why?**

Σ^* denotes the set of all words over an alphabet Σ

- Therefore, an abbreviated way of saying there is an arbitrary language L over an alphabet Σ is:
  - $L \subseteq \Sigma^*$

Building Regular Expressions

- Let E be a regular expression and the language represented by E is L(E)
- Then:
  - (E) = E
  - L(E + F) = L(E) U L(F)
  - L(E F) = L(E) L(F)
  - L(E*) = (L(E))*

Example: how to use these regular expression properties and language operators?

- *L = { w | w is a binary string which does not contain two consecutive 0s or two consecutive 1s anywhere)*
    - E.g., w = 01010101 is in L, while w = 10010 is not in L
- Goal: Build a regular expression for L
- Four cases for w:
    - Case A: w starts with 0 and |w| is even
    - Case B: w starts with 1 and |w| is even
    - Case C: w starts with 0 and |w| is odd
    - Case D: w starts with 1 and |w| is odd
- Regular expression for the four cases:
    - Case A: (01)*
    - Case B: (10)*
    - Case C: 0(10)*
    - Case D: 1(01)*
- Since L is the union of all 4 cases:
    - Reg Exp for L = (01)* + (10)* + 0(10)* + 1(01)*
- If we introduce ε then the regular expression can be simplified to:
    - Reg Exp for L = (Ɛ +1)(01)*(Ɛ +0)

Precedence of Operators

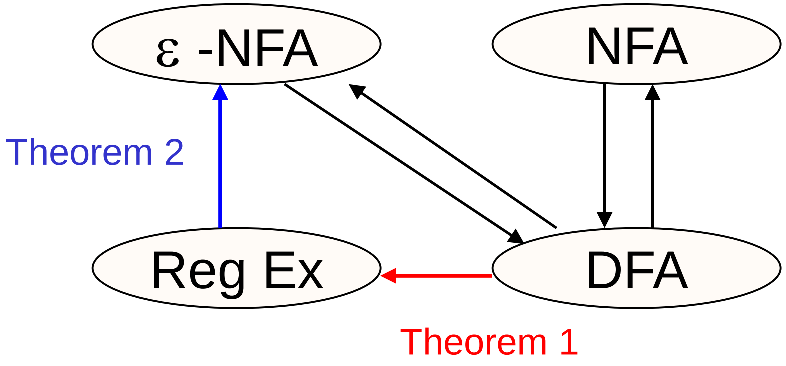
- Highest to lowest
  - * operator (star)
  - .   (concatenation)
  - + operator

- Example:
  - 01* + 1   =   (0 . ((1)*)) + 1

Finite Automata (FA) & Regular Expressions (Reg Ex)

- To show that they are interchangeable, consider the following theorems:
  - *Theorem 1: For every DFA A there exists a regular expression R such that L(R)=L(A)*
  - *Theorem 2: For every regular expression R there exists an ε -NFA E such that L(E)=L(R)*

Proofs in the book

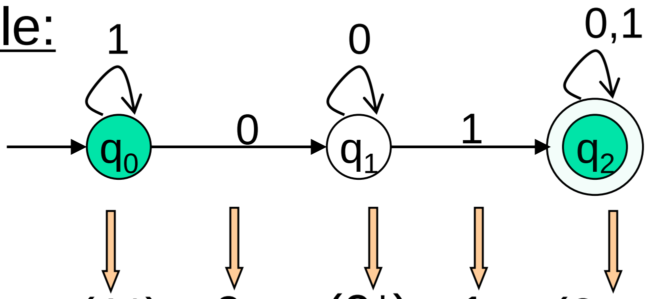


Kleene Theorem

DFA to RE construction

Informally, trace all distinct paths (traversing cycles only once) from the start state to *each of the* final states and enumerate all the expressions along the way

Example:



$$\underbrace{(1^*)}_{1^*}\quad\underbrace{0\quad(0^*)}_{00^*}\quad\underbrace{1}_{1}\quad\underbrace{(0 + 1)^*}_{(0+1)^*}$$

$$1^*00^*1(0+1)^*$$

Q) What is the language?

Reg Ex

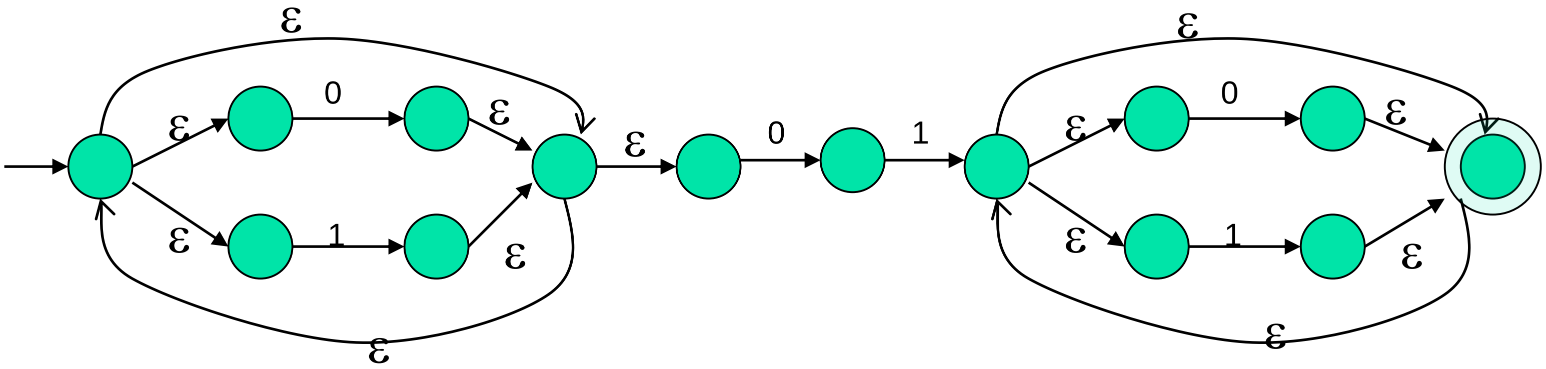
Theorem 2

ε -NFA

RE to ε-NFA construction

Example: (0+1)*01(0+1)*

(0+1)* 01 (0+1)*



Algebraic Laws of Regular Expressions

- Commutative:
  - $E+F = F+E$
- Associative:
  - $(E+F)+G = E+(F+G)$
  - $(EF)G = E(FG)$
- Identity:
  - $E+\Phi = E$
  - $\varepsilon E = E \varepsilon = E$
- Annihilator:
  - $\Phi E = E\Phi = \Phi$

Algebraic Laws…

- Distributive:
  - $E(F+G) = EF + EG$
  - $(F+G)E = FE+GE$
- Idempotent: $E + E = E$
- Involving Kleene closures:
  - $(E^*)^* = E^*$
  - $\Phi^* = \varepsilon$
  - $\varepsilon^* = \varepsilon$
  - $E^+ = EE^*$
  - $E? = \varepsilon + E$

True or False?

Let R and S be two regular expressions. Then:

1. $((R^*)^*)^* = R^*$?
2. $(R+S)^* = R^* + S^*$?
3. $(RS + R)^* RS = (RR^*S)^*$?

Summary

- Regular expressions
- Equivalence to finite automata
- DFA to regular expression conversion
- Regular expression to ε-NFA conversion
- Algebraic laws of regular expressions
- Unix regular expressions and Lexical Analyzer

Properties of Regular Languages

Topics

1) How to prove whether a given language is regular or not?

2) Closure properties of regular languages

3) Minimization of DFAs

Some languages are *not* regular

When is a language is regular?

if we are able to construct one of the following: DFA *or* NFA *or* ε -NFA *or* regular expression

When is it not?

If we can show that no FA can be built for a language

How to prove languages are ***not*** regular?

What if we cannot come up with any FA?

A) Can it be language that is not regular?

B) Or is it that we tried wrong approaches?

How do we *decisively* prove that a language is not regular?

“The hardest thing of all is to find a black cat in a dark room, especially if there is no cat!” -Confucius

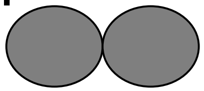
Example of a non-regular language

Let L = {w | w is of the form 0^n1^n , for all n≥0}

- *Hypothesis:* *L is not regular*
- Intuitive rationale: How do you keep track of a running count in an FA?
- A more formal rationale:
  - By contradition, if L is regular then there should exist a DFA for L.
  - Let k = number of states in that DFA.
  - Consider the special word w= 0^k1^k => w $\in$ L
  - DFA is in some state p_i, after consuming the first i symbols in w

Uses Pigeon Hole Principle

Rationale…

- Let $\{p_0, p_1, \ldots\ p_k\}$ be the sequence of states that the DFA should have visited after consuming the first k symbols in w which is 0^k
- But there are only k states in the DFA!
- ==> at least one state should repeat somewhere along the path (by +  Principle)
- ==> Let the repeating state be $p_i = p_J$ for $i < j$
- ==> We can fool the DFA by inputing $0^{(k-(j-i))}1^k$ and still get it to accept (note: k-(j-i) is at most k-1).
- ==> DFA accepts strings w/ unequal number of 0s and 1s, implying that the DFA is wrong! ↯

The Pumping Lemma for Regular Languages

What it is?

The Pumping Lemma is a property of all regular languages.

How is it used?

A technique that is used to show that a given language is not regular

Pumping Lemma for Regular Languages

Let L be a regular language

Then *there exists* some constant ***N*** such that *for every* string $w \in L$ s.t. $|w| \geq N$, *there exists* a way to break w into three parts, $w = xyz$, such that:

1. $y \neq \varepsilon$
2. $|xy| \leq N$
3. For all $k \geq 0$, all strings of the form $xy^kz \in L$

This property should hold for all regular languages.

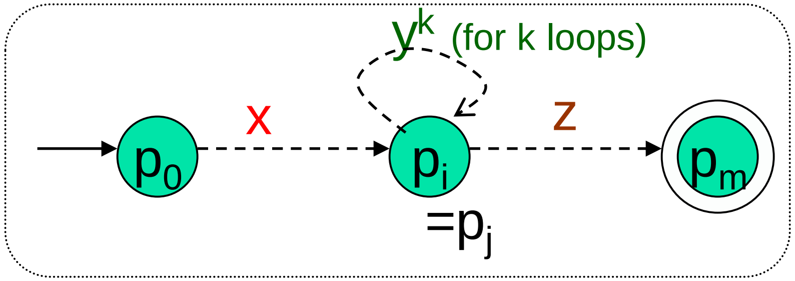
Definition: *N* is called the "Pumping Lemma Constant"

Pumping Lemma: Proof

- L is regular => it should have a DFA.
  - Set *N* := number of states in the DFA
- Any string w∈L, s.t. |w|≥*N*, should have the form: $w=a_1a_2\ldots a_m$, where m≥N
- Let the states traversed after reading the first N symbols be: $\{p_0,p_1,\ldots\ p_N\}$
  - ==> There are N+1 p-states, while there are only N DFA states
  - ==> at least one state has to repeat i.e, $p_i= p_J$ where 0≤i<j≤N (by PHP)

Pumping Lemma: Proof…

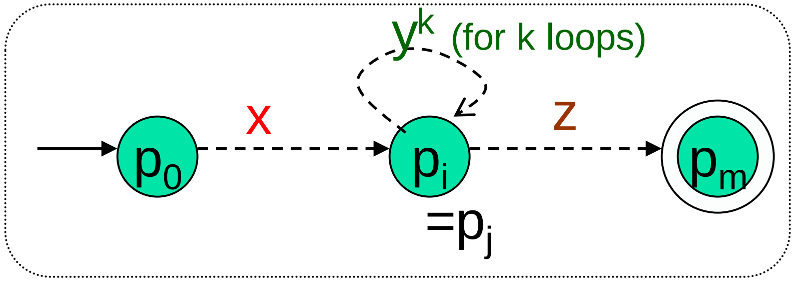
- => We should be able to break w=xyz as follows:
  - $x=a_1a_2..a_i$; $y=a_{i+1}a_{i+2}..a_J$; $z=a_{J+1}a_{J+2}..a_m$
  - x's path will be $p_0..p_i$
  - y's path will be $p_i\, p_{i+1}..p_J$ (but $p_i=p_J$ implying a loop)
  - z's path will be $p_Jp_{J+1}..p_m$
- Now consider another string $w_k=xy^kz$, where k≥0
- Case k=0
  - DFA will reach the accept state p_m
- Case k>0
  - DFA will loop for y^k, and finally reach the accept state p_m for z
- In either case, $w_k \in L$



This proves part (3) of the lemma

Pumping Lemma: Proof…

- For part (1):
  - Since i<j, y ≠ ε



- For part (2):
  - By PHP, the repetition of states has to occur within the first N symbols in w
  - ==> |xy|≤N

□

The Purpose of the Pumping Lemma for RL

- To prove that some languages *cannot be* regular.

How to use the pumping lemma?

Think of playing a 2 person game

- Role 1: ***We*** claim that the language cannot be regular

- Role 2: An ***adversary*** who claims the language is regular

- We show that the adversary’s statement will lead to a contradiction that implyies pumping lemma *cannot* hold for the language.

- We win!!

How to use the pumping lemma? (The Steps)

1. (we) L is not regular.
2. (adv.) Claims that L is regular and gives you a value for N as its P/L constant
3. (we) Using N, choose a string $w \in L$ s.t.,
   1. $|w| \geq N$,
   2. Using w as the template, construct other words w_k of the form xy^kz and show that at least one such $w_k \notin L$

      => this implies we have successfully broken the pumping lemma for the language, and hence that the adversary is wrong.

(Note: In this process, we may have to try many values of k, starting with k=0, and then 2, 3, .. so on, until $w_k \notin L$)

Note: We don’t have any control over N, except that it is positive.
We also don’t have any control over how to split w=xyz,
but xyz should respect the P/L conditions (1) and (2).

Using the Pumping Lemma

- What WE do?

  3. Using *N*, we construct our template string *w*
  4. Demonstrate to the adversary, either through pumping up or down on *w*, that some string $w_k \notin L$ (this should happen regardless of w=xyz)

- What the Adversary does?

  1. Claims L is regular
  2. Provides *N*

Note: This N can be anything (need not necessarily be the #states in the DFA. It's the adversary's choice.)

Example of using the Pumping Lemma to prove that a language is not regular

Let L_{eq} = {w | w is a binary string with equal number of 1s and 0s}

- Your Claim: L_{eq} is not regular
- Proof:
  - By contradiction, let L_{eq} be regular → **adv.**
  - P/L constant should exist → **adv.**
    - Let *N* = that P/L constant
  - Consider input w = 0^N1^N → **you**

    (your choice for the template string)
  - By pumping lemma, we should be able to break w=xyz, such that: → **you**
    1) $y \neq \varepsilon$
    2) $|xy| \leq N$
    3) For all k≥0, the string xy^kz is also in L

Template string $w = 0^N1^N = \underbrace{00\ \ldots.\ 0}_{N}\underbrace{11\ \ldots\ 1}_{N}$

Proof…

➔ you

- Because $|xy| \leq N$, xy should contain only 0s
  - (This and because $y \neq \varepsilon$, implies $y=0^+$)
- Therefore x can contain *at most* N-1 0s
- Also, all the N 1s must be inside z
- By (3), any string of the form $xy^kz \in L_{eq}$ for all $k \geq 0$
- <u>Case k=0:</u> xz has at most N-1 0s but has N 1s
- Therefore, $xy^0z \notin L_{eq}$
- This violates the P/L (a contradiction) ↯

Setting k=0 is referred to as **<u>“pumping down”</u>**

Setting k>1 is referred to as **<u>“pumping up”</u>**

Another way of proving this will be to show that if the #0s is arbitrarily pumped up (e.g., k=2), then the #0s will become exceed the #1s

Exercise 2

Prove $L = \{0^n 1 0^n \mid n \geq 1\}$ is not regular

Note: This n is not to be confused with the pumping lemma constant N. That *can* be different.

In other words, the above question is same as proving:

- $L = \{0^m 1 0^m \mid m \geq 1\}$ is not regular

Example 3: Pumping Lemma

Claim: L = { 0^i | i is a perfect square} is not regular

- Proof:
  - By contradiction, let L be regular.
  - P/L should apply
  - Let *N* = P/L constant
  - Choose $w=0^{N^2}$
  - By pumping lemma, w=xyz satisfying all three rules
  - By rules (1) & (2), y has between 1 and N 0s
  - By rule (3), any string of the form xy^kz is also in L for all k≥0
  - Case k=0:
    - #zeros (xy^0z) = #zeros (xyz) - #zeros (y)
    - $N^2 – N \leq$ #zeros (xy^0z) $\leq N^2 - 1$
    - $\mathbf{(N-1)^2} < N^2 - N \leq$ #zeros (xy^0z) $\leq N^2 - 1 < \mathbf{N^2}$
    - $xy^0z \notin L$
    - But the above will complete the proof ONLY IF N>1.
    - … (proof contd.. Next slide)

Example 3: Pumping Lemma

- (proof contd…)
  - If the adversary pick N=1, then $\mathbf{(N-1)^2} \leq N^2 - N$, and therefore the #zeros(xy^0z) could end up being a perfect square!
  - This means that pumping down (i.e., setting k=0) is not giving us the proof!
  - So lets try pumping up next…
- Case k=2:
  - #zeros (xy^2z) $=$ #zeros (xyz) $+$ #zeros (y)
  - $N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N$
  - $\mathbf{N^2} < N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N < \mathbf{(N+1)^2}$
  - $xy^2z \notin L$ ↯
  - (Notice that the above should hold for all possible N values of N>0. Therefore, this completes the proof.)

Closure properties of Regular Languages

Closure properties for Regular Languages (RL)

This is different from Kleene closure

- *Closure property:*
  - If a set of regular languages are combined using an operator, then the resulting language is also regular
- Regular languages are *closed* under:
  - Union, intersection, complement, difference
  - Reversal
  - Kleene closure
  - Concatenation
  - Homomorphism
  - Inverse homomorphism

Now, lets prove all of this!

RLs are closed under union

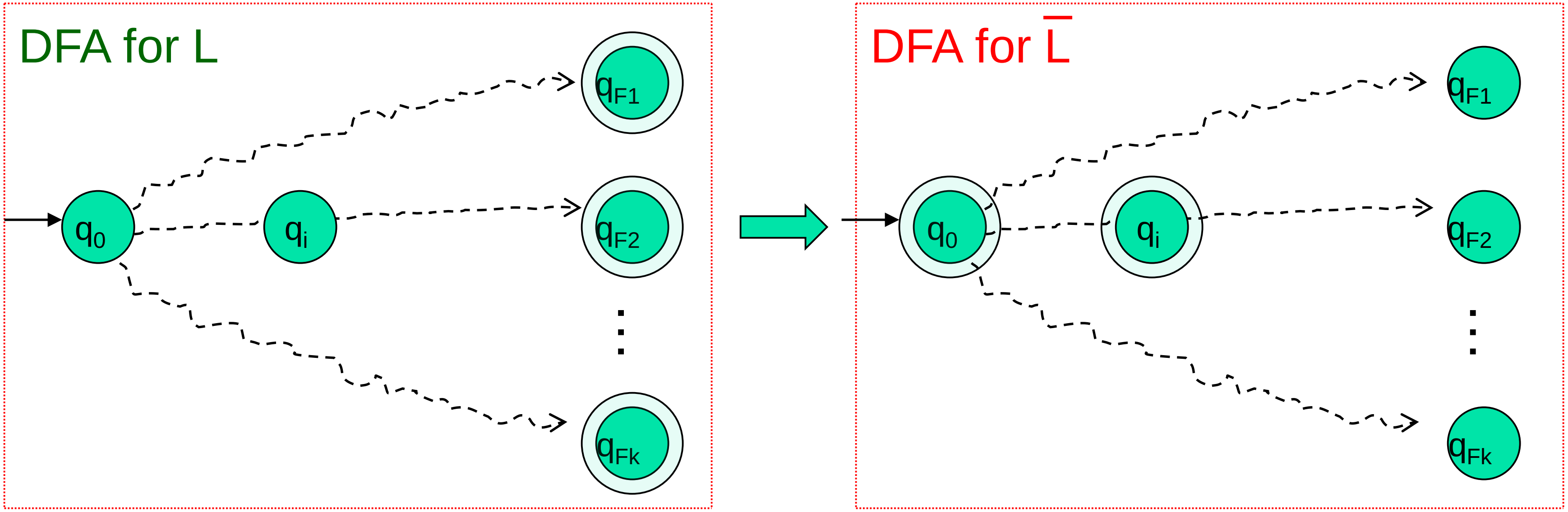
- IF L and M are two RLs THEN:
  - they both have two corresponding regular expressions, R and S respectively
  - (L U M) can be represented using the regular expression R+S
  - Therefore, (L U M) is also regular □

How can this be proved using FAs?

RLs are closed under complementation

- If L is an RL over ∑, then $\overline{L}=\Sigma^*-L$
- To show $\overline{L}$ is also regular, make the following construction

Convert every final state into non-final, and every non-final state into a final state



Assumes q0 is a non-final state. If not, do the opposite.

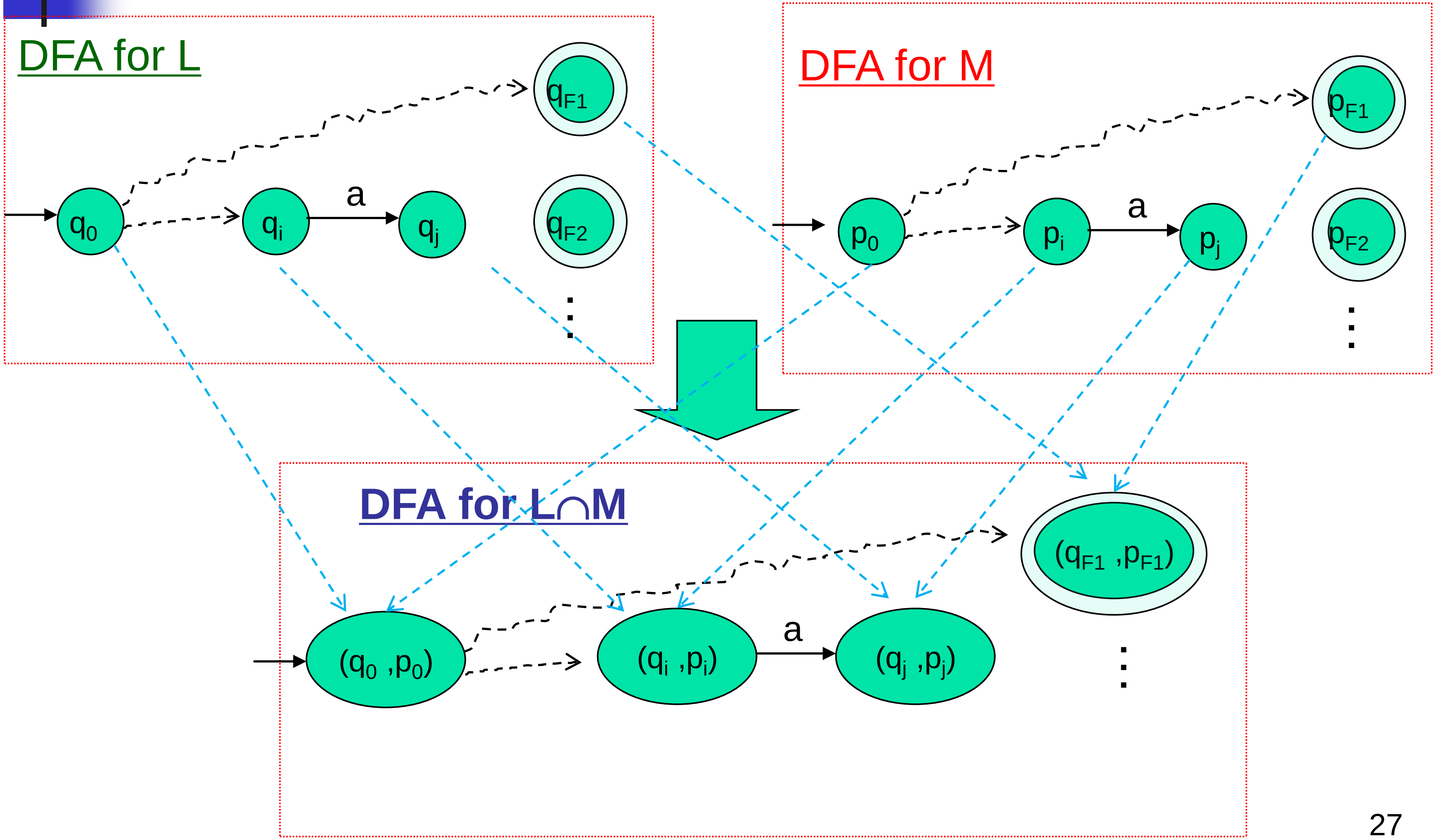
RLs are closed under intersection

- A quick, indirect way to prove:
  - By DeMorgan’s law:
  - $L \cap M = \overline{(\overline{L} \cup \overline{M})}$
  - Since we know RLs are closed under union and complementation, they are also closed under intersection
- A more direct way would be construct a finite automaton for $L \cap M$

DFA construction for $L \cap M$

- A_L = DFA for L = $\{Q_L, \sum, q_L, F_L, \delta_L\}$
- A_M = DFA for M = $\{Q_M, \sum, q_M, F_M, \delta_M\}$
- Build $A_{L \cap M} = \{Q_L x\ Q_M, \sum, (q_L, q_M), F_L x\ F_M, \delta\}$ such that:
  - $\delta((p,q),a) = (\delta_L(p,a), \delta_M(q,a))$, where p in Q_L, and q in Q_M
- This construction ensures that a string w will be accepted if and only if w reaches an accepting state in both input DFAs.

DFA construction for L ∩ M



RLs are closed under set difference

- We observe:
  - $L - M = L \cap \overline{M}$

Closed under intersection

Closed under complementation

- Therefore, L - M is also regular

RLs are closed under reversal

Reversal of a string w is denoted by w^R

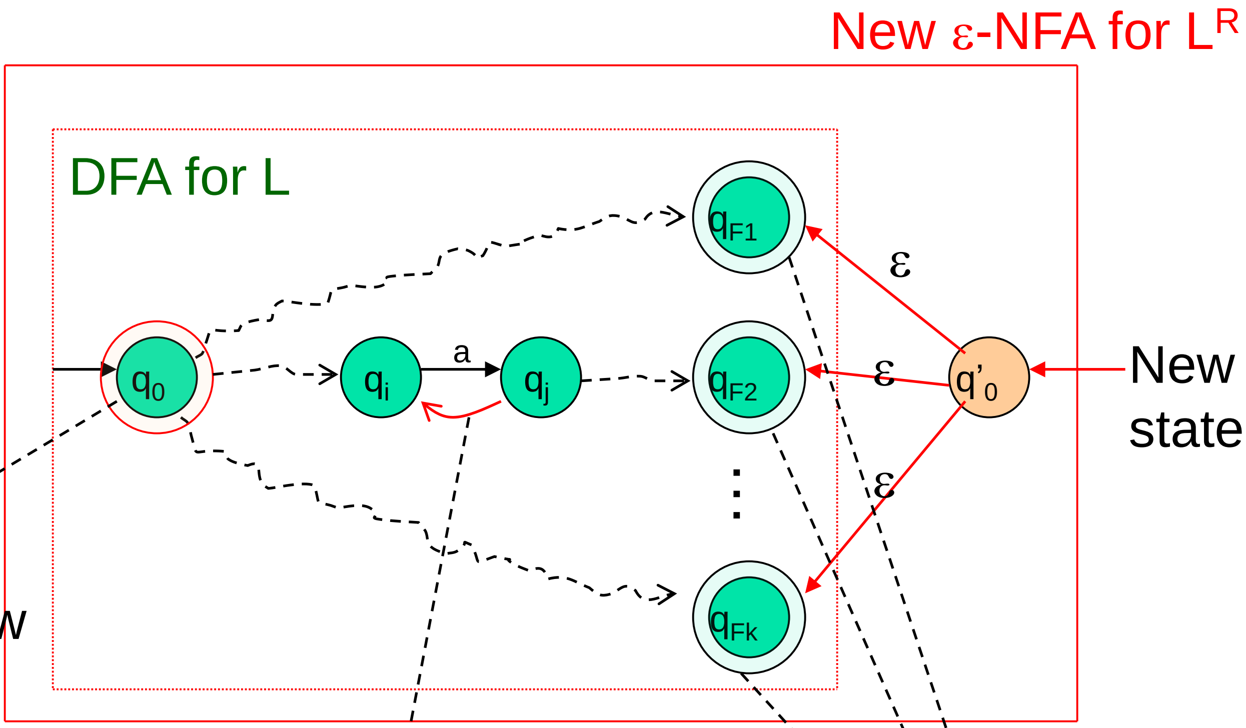
- E.g., w=00111, w^R=11100

Reversal of a language:

- L^R = The language generated by reversing all strings in L

Theorem: If L is regular then L^R is also regular

ε -NFA Construction for L^R



New start state

Make the old start state as the only new final state

Reverse all transitions

Convert the old set of final states into non-final states

What to do if q_0 was one of the final states in the input DFA?

If L is regular, L^R is regular (proof using regular expressions)

- Let E be a regular expression for L
- Given E, how to build E^R?
- <u>Basis:</u> If E= ε, Ø, or a, then E^R=E
- <u>Induction:</u> Every part of E (refer to the part as “F”) can be in only *one* of the three following forms:

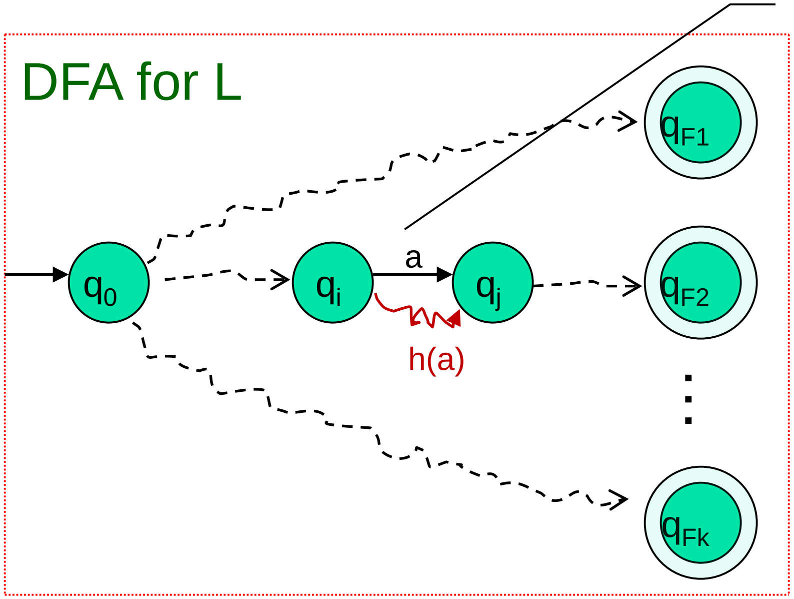
1. $F = F_1+F_2$
   - $F^R = F_1^R+F_2^R$
2. $F = F_1F_2$
   - $F^R = F_2^RF_1^R$
3. $F = (F_1)^*$
   - $(F^R)^* = (F_1^R)^*$

Homomorphisms

- Substitute each symbol in ∑ (main alphabet) by a corresponding string in T (another alphabet)
  - h: ∑--->T*
- Example:
  - Let ∑={0,1} and T={a,b}
  - Let a homomorphic function h on ∑ be:
    - h(0)=ab, h(1)=ε
  - If w=10110, then h(w) = εabεεab = abab
- In general,
  - $h(w) = h(a_1)\ h(a_2) \ldots\ h(a_n)$

Given a DFA for L, how to convert it into an FA for h(L)?

FA Construction for h(L)



Replace every edge "a" by
a path labeled h(a)
in the new DFA

- Build a new FA that simulates h(a) for every symbol a transition in the above DFA
- The resulting FA may or may not be a DFA, but will be a FA for h(L)

Given a DFA for M, how to convert it into an FA for $h^{-1}(M)$?

The set of strings in ∑* whose homomorphic translation results in the strings of M

Inverse homomorphism

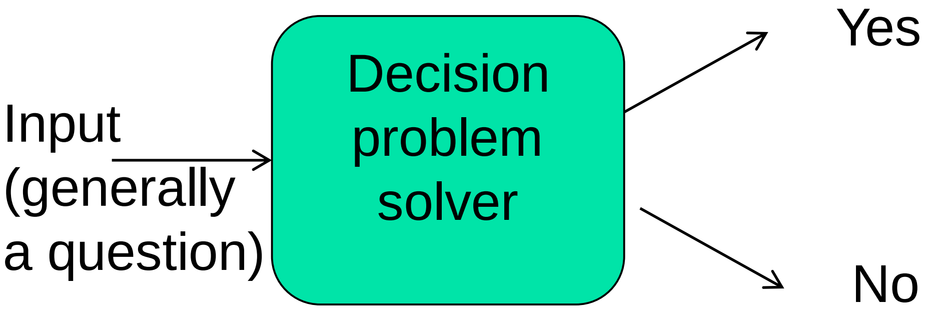
- Let h: ∑--->T*
- Let M be a language over alphabet T
- $h^{-1}(M) = \{w \mid w \in \Sigma^* \text{ s.t., } h(w) \in M \}$

Claim: If M is regular, then so is $h^{-1}(M)$

- Proof:
  - Let A be a DFA for M
  - Construct another DFA A' which encodes $h^{-1}(M)$
  - A' is an exact replica of A, except that its transition functions are s.t. for any input symbol *a* in ∑, A' will simulate *h(a)* in A.
    - $\delta(p,a) = \hat{\delta}(p,h(a))$

Decision properties of regular languages

Any “decision problem” looks like this:



Input (generally a question)

Decision problem solver

Yes

No

Membership question

- Decision Problem: Given L, is w in L?
- Possible answers: Yes or No
- Approach:
  1. Build a DFA for L
  2. Input w to the DFA
  3. If the DFA ends in an accepting state, then yes; otherwise no.

Emptiness test

- Decision Problem: Is L=Ø ?
- Approach:

On a DFA for L:

1. From the start state, run a *reachability* test, which returns:
    1. success: if there is at least one final state that is reachable from the start state
    2. failure: otherwise
2. L=Ø if and only if the reachability test fails

How to implement the reachability test?

Finiteness

- Decision Problem: Is L finite or infinite?
- Approach:

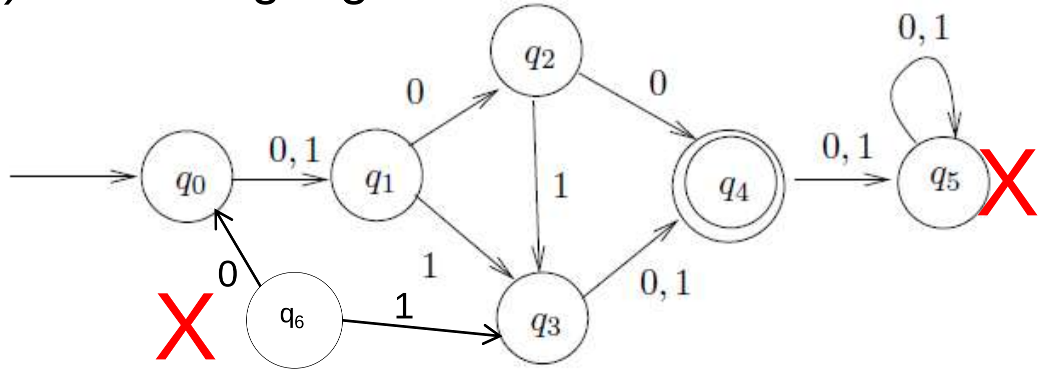
  On a DFA for L:

  1. Remove all states unreachable from the start state
  2. Remove all states that cannot lead to any accepting state.
  3. After removal, check for cycles in the resulting FA
  4. L is finite if there are no cycles; otherwise it is infinite
- Another approach
  - Build a regular expression and look for Kleene closure

How to implement steps 2 and 3?

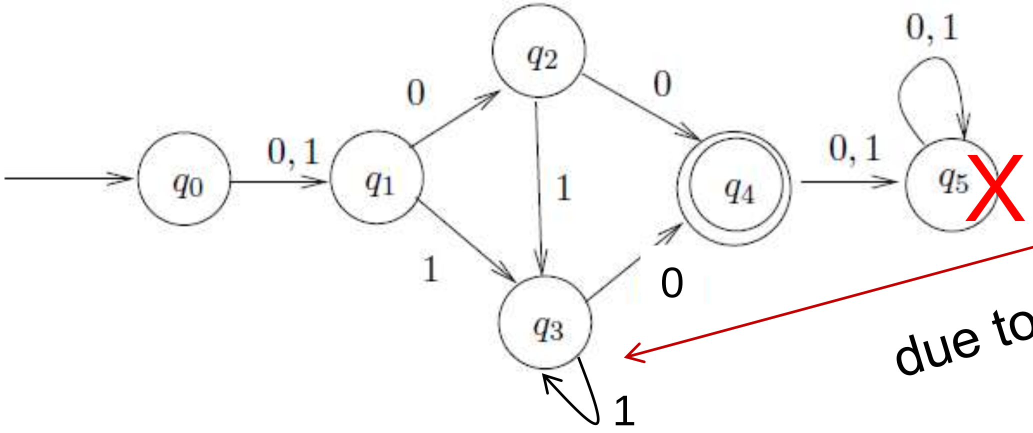
Finiteness test - examples

Ex 1) Is the language of this DFA finite or infinite?



FINITE

Ex 2) Is the language of this DFA finite or infinite?



INFINITE

due to this

Equivalence & Minimization of DFAs

Applications of interest

- Comparing two DFAs:
  - $L(DFA_1) == L(DFA_2)$?

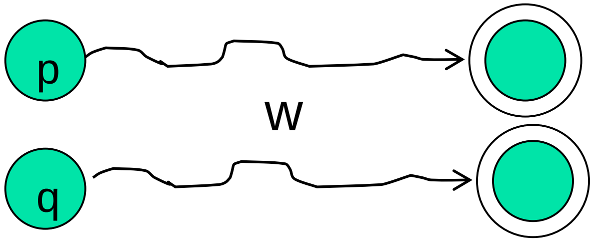
- How to minimize a DFA?
  1. Remove unreachable states
  2. Identify & condense equivalent states into one

When to call two states in a DFA "equivalent"?

Past doesn't matter - only future does!

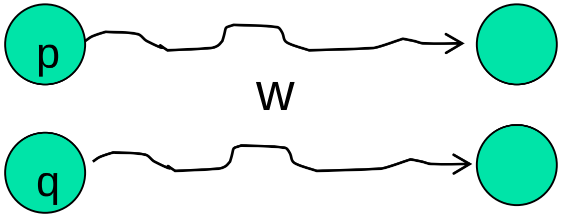
Two states p and q are said to be *equivalent* iff:

i) Any string w accepted by starting at p is also accepted by starting at q;



AND

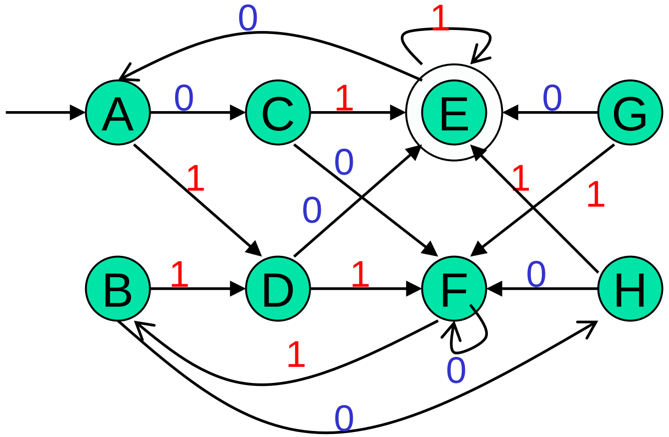
i) Any string w rejected by starting at p is also rejected by starting at q.



➔ p≡q

Computing equivalent states in a DFA

Table Filling Algorithm



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

Pass #0

1. Mark accepting states ≠ non-accepting states

Pass #1

1. Compare every pair of states
2. Distinguish by one symbol transition
3. Mark = or ≠ or blank(tbd)

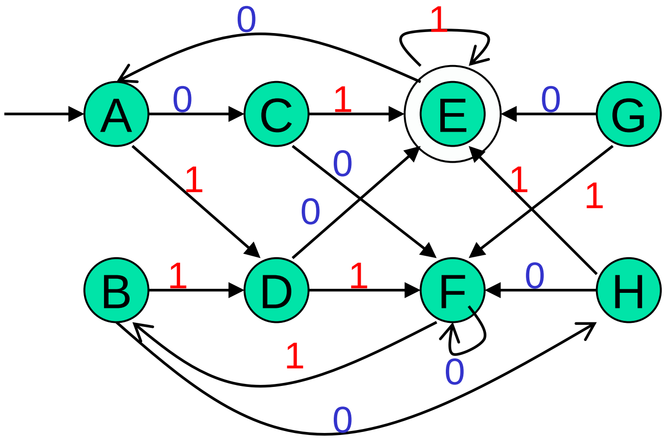
Pass #2

1. Compare every pair of states
2. Distinguish by up to two symbol transitions (until different or same or tbd)

….

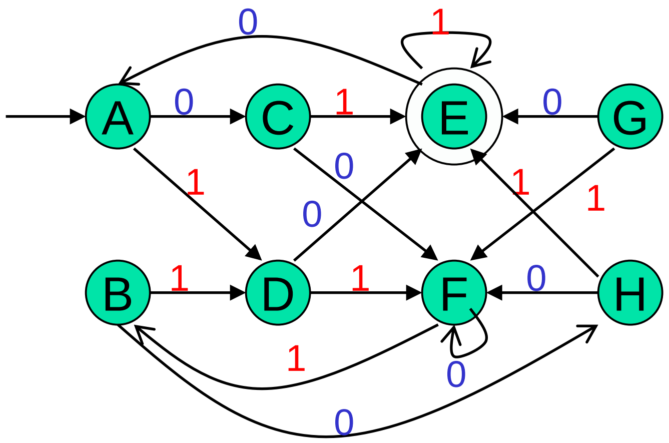
(keep repeating until table complete)

Table Filling Algorithm - step by step



A	=							
B		=						
C			=					
D				=				
E					=			
F						=		
G							=	
H								=
	A	B	C	D	E	F	G	H

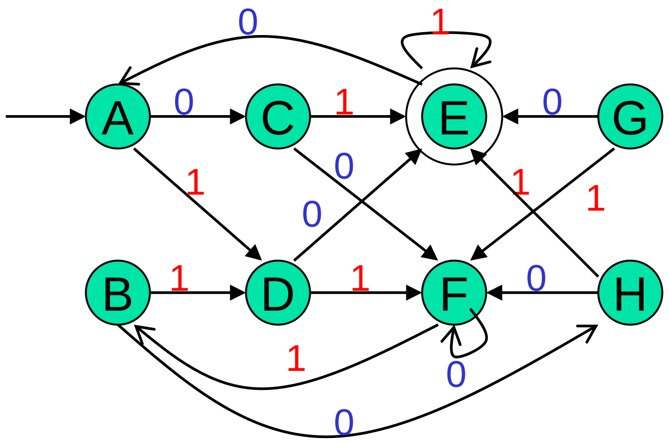
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state

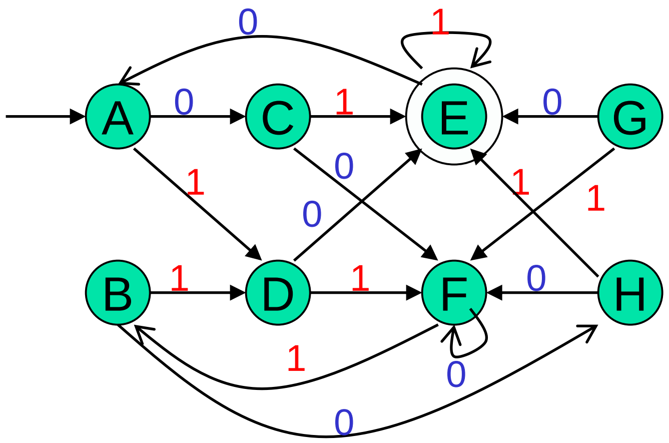
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

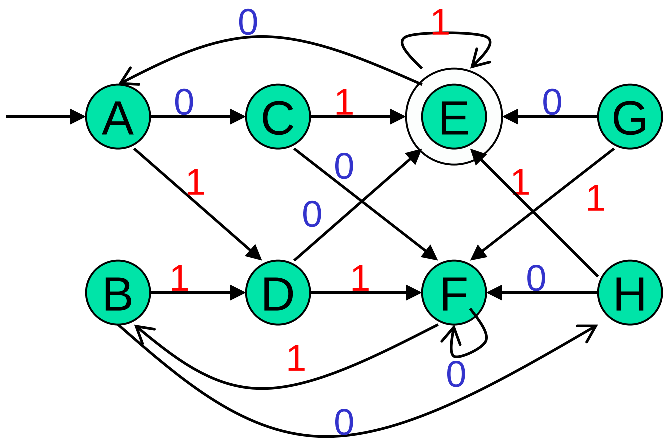
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

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

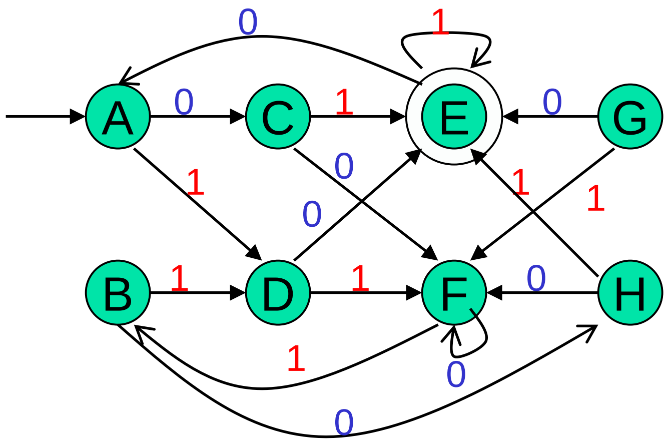
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

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

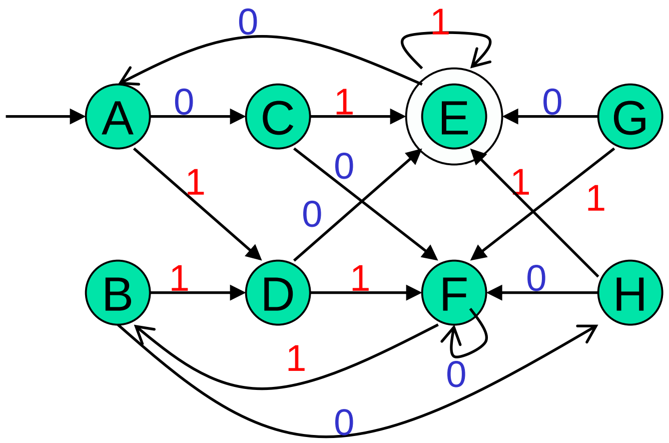
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

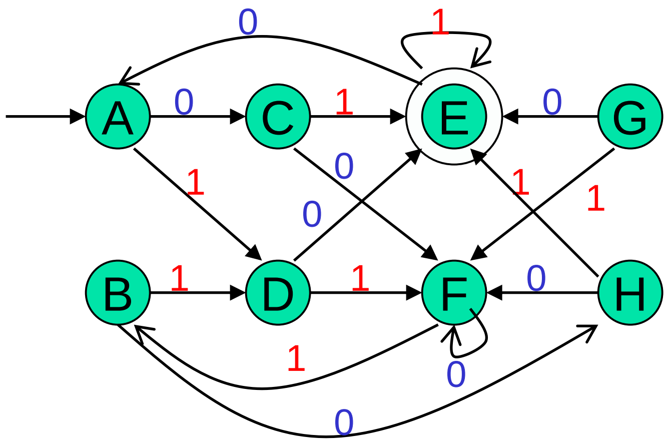
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

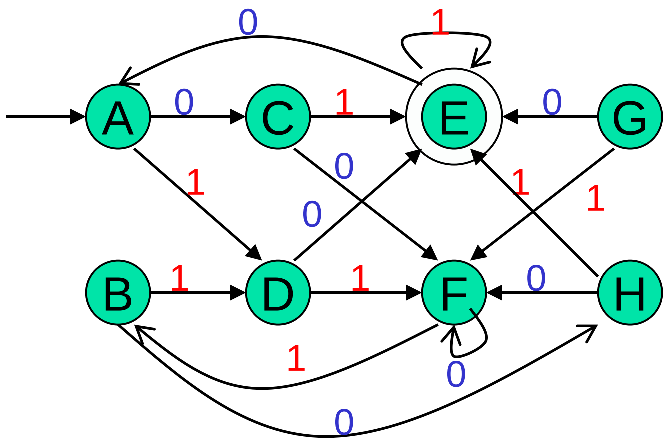
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

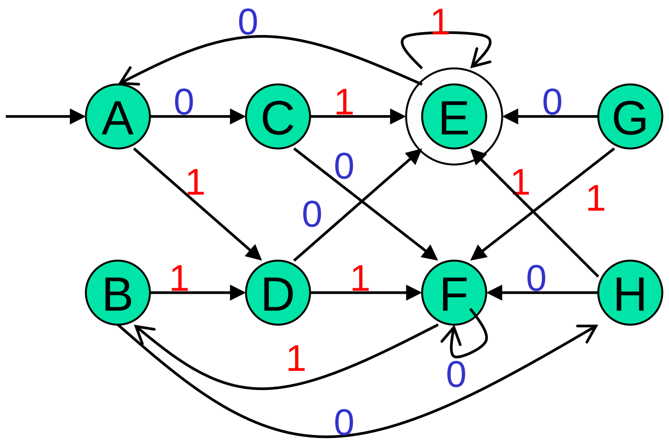
Table Filling Algorithm - step by step



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

1. Mark X between accepting vs. non-accepting state
2. Pass 1:
   Look 1- hop away for distinguishing states or strings
3. Pass 2:
   Look 1-hop away again for distinguishing states or strings
   continue….

Table Filling Algorithm - step by step

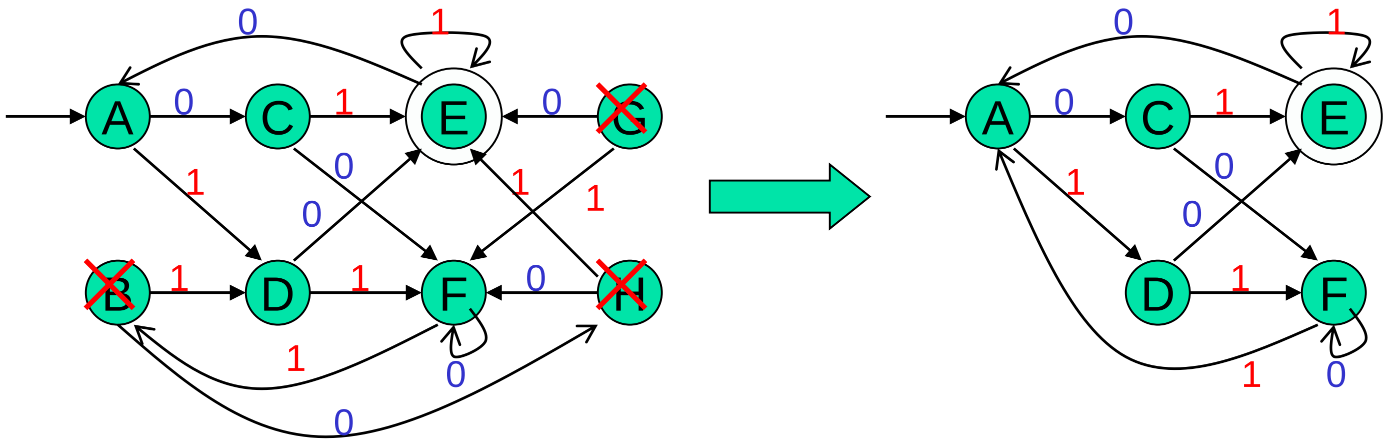


A	=							
B	=	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Pass 1:
   Look 1- hop away for distinguishing states or strings
3. Pass 2:
   Look 1-hop away again for distinguishing states or strings
   continue….

Equivalences:
- A=B
- C=H
- D=G

Table Filling Algorithm - step by step

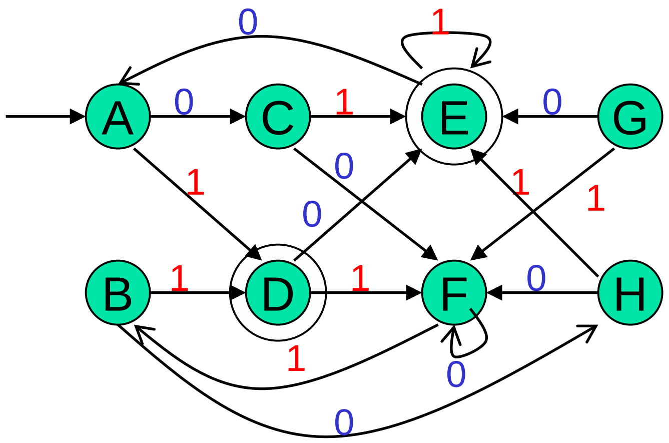


Retrain only one copy for each equivalence set of states

Equivalences:

- A=B
- C=H
- D=G

Table Filling Algorithm – special case



A	=							
B		=						
C			=					
D				=				
E				?	=			
F						=		
G							=	
H								=
	A	B	C	D	E	F	G	H

Q) What happens if the input DFA has more than one final state?

Can all final states initially be treated as equivalent to one another?

Putting it all together …

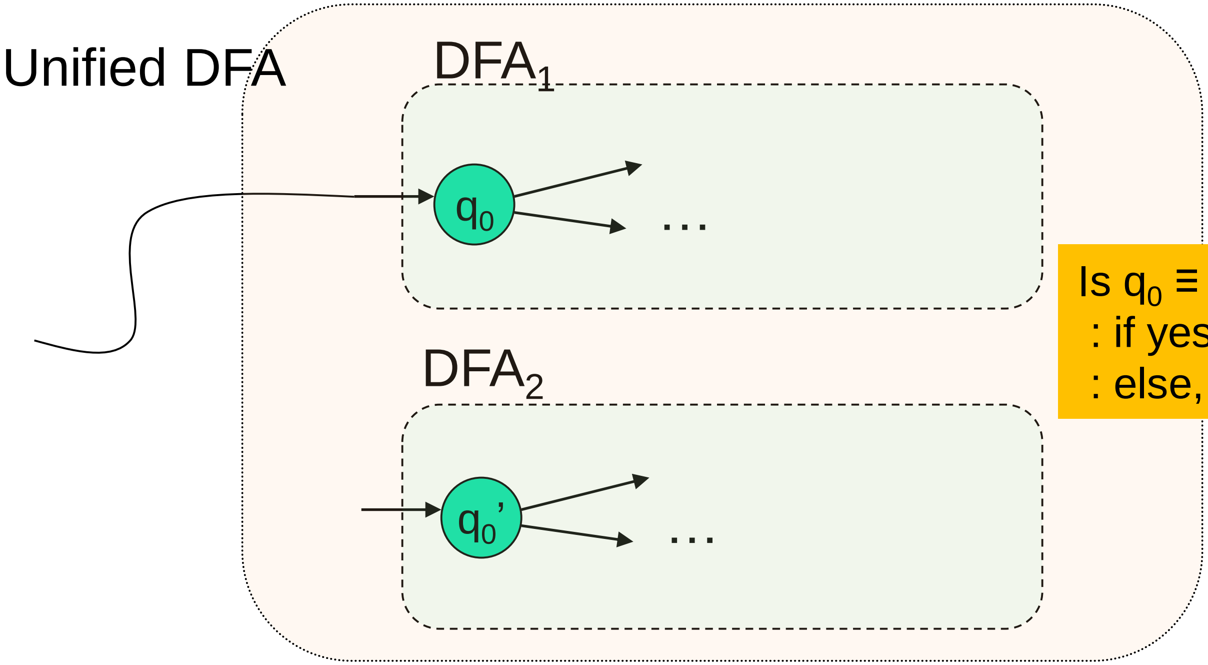
How to minimize a DFA?

- Goal: Minimize the number of states in a DFA
- Algorithm:
  1. Eliminate states unreachable from the start state
  2. Identify and remove equivalent states
  3. Output the resultant DFA

Depth-first traversal from the start state

Table filling algorithm

Are Two DFAs Equivalent?



Is $q_0 \equiv q_0$'?
: if yes, then $DFA_1 \equiv DFA_2$
: else, not equiv.

1. Make a new dummy DFA by just putting together both DFAs
2. Run table-filling algorithm on the unified DFA
3. *IF* the start states of both DFAs are found to be equivalent,

   THEN: $DFA_1 \equiv DFA_2$
   ELSE: different

Summary

- How to prove languages are not regular?
  - Pumping lemma & its applications

- Closure properties of regular languages

- Simplification of DFAs
  - How to remove unreachable states?
  - How to identify and collapse equivalent states?
  - How to minimize a DFA?
  - How to tell whether two DFAs are equivalent?

Topics

1) How to prove whether a given language is regular or not?

Some languages are *not* regular

When is a language is regular?

if we are able to construct one of the following: DFA *or* NFA *or* ε -NFA *or* regular expression

When is it not?

If we can show that no FA can be built for a language

How to prove languages are ***not*** regular?

What if we cannot come up with any FA?

A) Can it be language that is not regular?

B) Or is it that we tried wrong approaches?

How do we *decisively* prove that a language is not regular?

"The hardest thing of all is to find a black cat in a dark room, especially if there is no cat!" -Confucius

The Pumping Lemma for Regular Languages

What it is?

The Pumping Lemma is a property of all regular languages.

How is it used?

A technique that is used to show that a given language is not regular

Pumping Lemma for Regular Languages

Let L be a regular language

Then *there exists* some constant ***N*** such that *for every* string $w \in L$ s.t. $|w| \geq N$, *there exists* a way to break w into three parts, $w=xyz$, such that:

1. $y \neq \varepsilon$
2. $|xy| \leq N$
3. For all $k \geq 0$, all strings of the form $xy^kz \in L$

This property should hold for all regular languages.

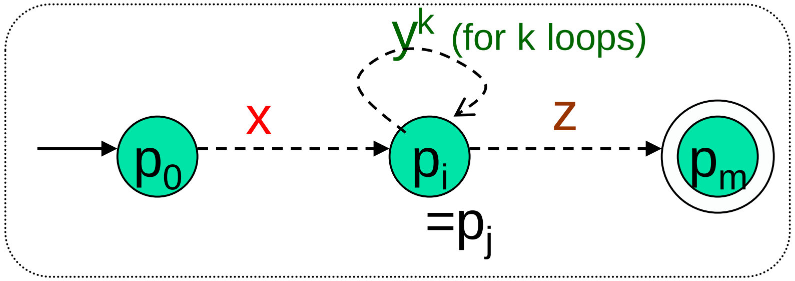
Definition: *N* is called the "Pumping Lemma Constant"

Pumping Lemma: Proof

- L is regular => it should have a DFA.
  - Set *N* := number of states in the DFA
- Any string w∈L, s.t. |w|≥*N*, should have the form: $w=a_1a_2\ldots a_m$, where m≥N
- Let the states traversed after reading the first N symbols be: $\{p_0,p_1,\ldots p_N\}$
  - ==> There are N+1 p-states, while there are only N DFA states
  - ==> at least one state has to repeat i.e, $p_i= p_J$where 0≤i<j≤N

Pumping Lemma: Proof…

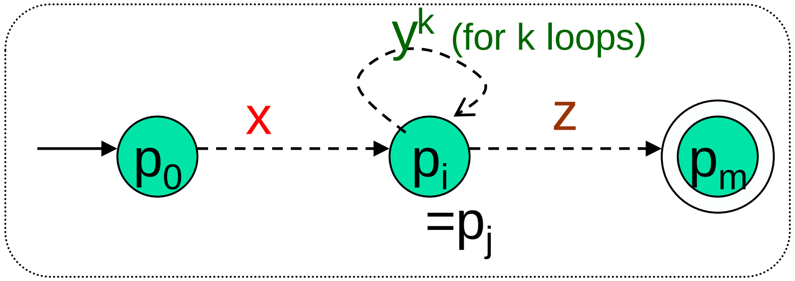
- => We should be able to break w=xyz as follows:
  - $x=a_1a_2..a_i$; $y=a_{i+1}a_{i+2}..a_J$; $z=a_{J+1}a_{J+2}..a_m$
  - x's path will be $p_0..p_i$
  - y's path will be $p_i\, p_{i+1}..p_J$ (but $p_i=p_J$ implying a loop)
  - z's path will be $p_Jp_{J+1}..p_m$
- Now consider another string $w_k=xy^kz$, where k≥0
- Case k=0
  - DFA will reach the accept state p_m
- Case k>0
  - DFA will loop for y^k, and finally reach the accept state p_m for z



- In either case, $w_k \in L$ This proves part (3) of the lemma

Pumping Lemma: Proof…

- For part (1):
  - Since i<j, y ≠ ε



- For part (2):
  - By PHP, the repetition of states has to occur within the first N symbols in w
  - ==> $|xy| \leq N$

□

The Purpose of the Pumping Lemma for RL

- To prove that some languages *cannot be* regular.

How to use the pumping lemma?

Think of playing a 2 person game

- Role 1: ***We*** claim that the language cannot be regular

- Role 2: An ***adversary*** who claims the language is regular

- We show that the adversary's statement will lead to a contradiction that implies pumping lemma *cannot* hold for the language.

- We win!!

How to use the pumping lemma? (The Steps)

1. (we) L is not regular.
2. (adv.) Claims that L is regular and gives you a value for N as its P/L constant
3. (we) Using N, choose a string $w \in L$ s.t.,
    1. $|w| \geq N$,
    2. Using w as the template, construct other words w_k of the form xy^kz and show that at least one such $w_k \notin L$

       => this implies we have successfully broken the pumping lemma for the language, and hence that the adversary is wrong.

(Note: In this process, we may have to try many values of k, starting with k=0, and then 2, 3, .. so on, until $w_k \notin L$)

Note: We don't have any control over N, except that it is positive.
We also don't have any control over how to split w=xyz,
but xyz should respect the P/L conditions (1) and (2).

Using the Pumping Lemma

- What WE do?

3. Using *N*, we construct our template string *w*
4. Demonstrate to the adversary, either through pumping up or down on *w*, that some string $w_k \notin$ L
(this should happen regardless of w=xyz)

- What the Adversary does?

1. Claims L is regular
2. Provides *N*

Note: This N can be anything (need not necessarily be the #states in the DFA. It's the adversary's choice.)

Example of using the Pumping Lemma to prove that a language is not regular

Let L_{eq} = {w | w is a binary string with equal number of 1s and 0s}

- Your Claim: L_{eq} is not regular
- Proof:
  - By contradiction, let L_{eq} be regular → adv.
  - P/L constant should exist → adv.
    - Let *N* = that P/L constant
  - Consider input w = 0^N1^N → you
    (your choice for the template string)
  - By pumping lemma, we should be able to break w=xyz, such that: →you
    1) $y \neq \varepsilon$
    2) $|xy| \leq N$
    3) For all k≥0, the string xy^kz is also in L

Template string $w = 0^N1^N = \underbrace{00\ \ldots\ 0}_{N}\underbrace{11\ \ldots\ 1}_{N}$

Proof…

➔ you

- Because $|xy| \leq N$, xy should contain only 0s
  - (This and because $y \neq \varepsilon$, implies $y=0^+$)
- Therefore x can contain *at most* N-1 0s
- Also, all the N 1s must be inside z
- By (3), any string of the form $xy^kz \in L_{eq}$ for all $k \geq 0$
- <u>Case k=0:</u> xz has at most N-1 0s but has N 1s
- Therefore, $xy^0z \notin L_{eq}$
- This violates the P/L (a contradiction) ↯

Setting k=0 is referred to as **<u>“pumping down”</u>**

Another way of proving this will be to show that if the #0s is arbitrarily pumped up (e.g., k=2), then the #0s will become exceed the #1s

Setting k>1 is referred to as **<u>“pumping up”</u>**

Exercise 2

Prove $L = \{0^n10^n \mid n \geq 1\}$ is not regular

Note: This n is not to be confused with the pumping lemma constant N. That *can* be different.

In other words, the above question is same as proving:

- $L = \{0^m10^m \mid m \geq 1\}$ is not regular

Example 3: Pumping Lemma

Claim: L = { 0^i | i is a perfect square} is not regular

- Proof:
  - By contradiction, let L be regular.
  - P/L should apply
  - Let *N* = P/L constant
  - Choose w=0^{N^2}
  - By pumping lemma, w=xyz satisfying all three rules
  - By rules (1) & (2), y has between 1 and N 0s
  - By rule (3), any string of the form xy^kz is also in L for all k≥0
  - Case k=0:
    - #zeros (xy^0z) = #zeros (xyz) - #zeros (y)
    - $N^2 - N \leq$ #zeros (xy^0z) $\leq N^2 - 1$
    - $\mathbf{(N-1)^2} < N^2 - N \leq$ #zeros (xy^0z) $\leq N^2 - 1 < \mathbf{N^2}$
    - $xy^0z \notin L$
    - But the above will complete the proof ONLY IF N>1.
    - … (proof contd.. Next slide)

Example 3: Pumping Lemma

- (proof contd…)
  - If the adversary pick N=1, then $\mathbf{(N-1)^2} \leq N^2 - N$, and therefore the #zeros(xy^0z) could end up being a perfect square!
  - This means that pumping down (i.e., setting k=0) is not giving us the proof!
  - So lets try pumping up next…
- Case k=2:
  - #zeros (xy^2z) $=$ #zeros (xyz) + #zeros (y)
  - $N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N$
  - $\mathbf{N^2} < N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N < \mathbf{(N+1)^2}$
  - $xy^2z \notin L$ ↯
  - (Notice that the above should hold for all possible N values of N>0. Therefore, this completes the proof.)

Summary

- How to prove languages are not regular?
  - Pumping lemma & its applications

Context-Free Languages & Grammars
(CFLs & CFGs)

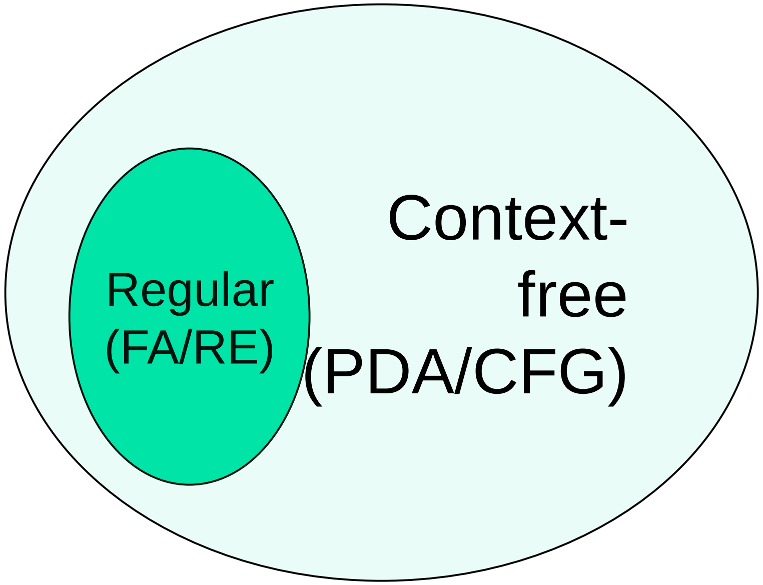
Not all languages are regular

- So what happens to the languages which are not regular?

- Can we still come up with a language recognizer?
  - i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?

Context-Free Languages

- A language class larger than the class of regular languages
- Supports natural, recursive notation called "context-free grammar"
- Applications:
  - Parse trees, compilers
  - XML



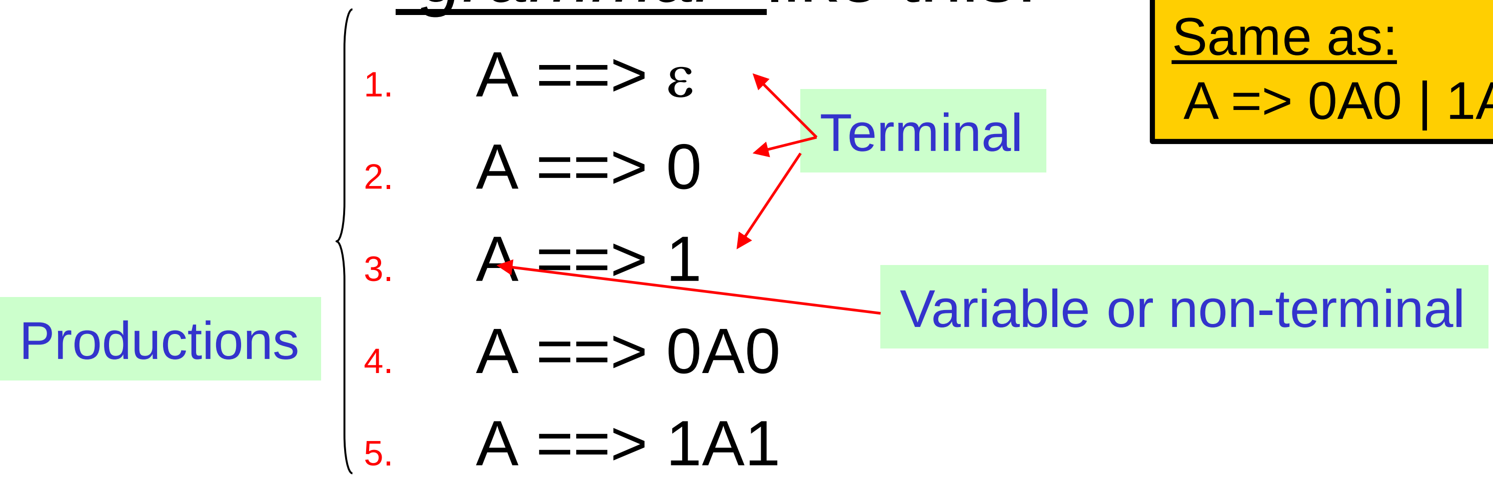
An Example

- A palindrome is a word that reads identical from both ends
  - E.g., $\overset{\longrightarrow\longleftarrow}{\text{madam}}$, $\overset{\longrightarrow\longleftarrow}{\text{redivider}}$, $\overset{\longrightarrow\longleftarrow}{\text{malayalam}}$, $\overset{\longrightarrow\longleftarrow}{010010010}$
- Let L = { w | w is a binary palindrome}
- Is L regular?
  - No.
  - <u>Proof:</u>
    - Let w=0^N10^N (assuming N to be the p/l constant)
    - By Pumping lemma, w can be rewritten as xyz, such that xy^kz is also L (for any k≥0)
    - But |xy|≤N and y≠ε
    - ==> $y=0^+$
    - ==> xy^kz *will NOT* be in L for k=0
    - ==> Contradiction

But the language of palindromes…

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a *“grammar”* like this:



1. A ==> ε
2. A ==> 0
3. A ==> 1
4. A ==> 0A0
5. A ==> 1A1

Terminal

Variable or non-terminal

Productions

Same as:
A => 0A0 | 1A1 | 0 | 1 | ε

How does this grammar work?

How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

G:
A => 0A0 | 1A1 | 0 | 1 | ε

- Example: w=01110
- G can generate w as follows:

1. A => 0A0
2. => 01A10
3. => 01110

Generating a string from a grammar:

1. Pick and choose a sequence of productions that would allow us to generate the string.
2. At every step, substitute one variable with one of its productions.

Context-Free Grammar: Definition

- A context-free grammar G=(V,T,P,S), where:
  - V: set of variables or non-terminals
  - T: set of terminals (= alphabet U {ε})
  - P: set of *productions,* each of which is of the form
    V ==> α_1 | α_2 | …
    - Where each α_i is an arbitrary string of variables and terminals
  - S ==> start variable

CFG for the language of binary palindromes:
G=({A},{0,1},P,A)
P: A ==> 0 A 0 | 1 A 1 | 0 | 1 | ε

More examples

- Parenthesis matching in code
- Syntax checking
- In scenarios where there is a general need for:
  - Matching a symbol with another symbol, or
  - Matching a count of one symbol with that of another symbol, or
  - Recursively substituting one symbol with a string of other symbols

Example #2

- Language of balanced paranthesis

  e.g., ()(((())))((()))….
- CFG?

G:

S => (S) | SS | ε

How would you “interpret” the string “(((()))()())” using this grammar?

Example #3

- A grammar for L = $\{0^m1^n \mid m \geq n\}$

- CFG?

G:
S => 0S1 | A
A => 0A | ε

How would you interpret the string “00000111” using this grammar?

Example #4

A program containing **if-then(-else)** statements

if *Condition* **then** *Statement* **else** *Statement*

(Or)

if *Condition* **then** *Statement*

CFG?

More examples

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3=\{0^i 1^j 2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_4=\{0^i 1^j 2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$

Applications of CFLs & CFGs

- Compilers use parsers for syntactic checking
- Parsers can be expressed as CFGs
  1. Balancing paranthesis:
     - B ==> BB | (B) | *Statement*
     - *Statement* ==> …
  2. If-then-else:
     - S ==> SS | *if Condition then Statement else Statement* | *if Condition then Statement* | *Statement*
     - *Condition* ==> …
     - *Statement* ==> …
  3. C paranthesis matching { … }
  4. Pascal *begin-end* matching
  5. YACC (Yet Another Compiler-Compiler)

More applications

- Markup languages
  - Nested Tag Matching
    - HTML
      - <html> …<p> … <a href=…> … </a> </p> … </html>
    - XML
      - <PC> … <MODEL> … </MODEL> .. <RAM> … </RAM> … </PC>

Tag-Markup Languages

Roll ==> <ROLL> Class Students </ROLL>

Class ==> <CLASS> Text </CLASS>

Text ==> Char Text | Char

Char ==> a | b | … | z | A | B | .. | Z

Students ==> Student Students | ε

Student ==> <STUD> Text </STUD>

Here, the left hand side of each production denotes one non-terminals (e.g., “Roll”, “Class”, etc.)

Those symbols on the right hand side for which no productions (i.e., substitutions) are defined are terminals (e.g., ‘a’, ‘b’, ‘|’, ‘<‘, ‘>’, “ROLL”, etc.)

Structure of a production

head derivation body

$A \quad \text{=======>} \quad \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

The above is same as:

1. $A \Longrightarrow \alpha_1$
2. $A \Longrightarrow \alpha_2$
3. $A \Longrightarrow \alpha_3$

…

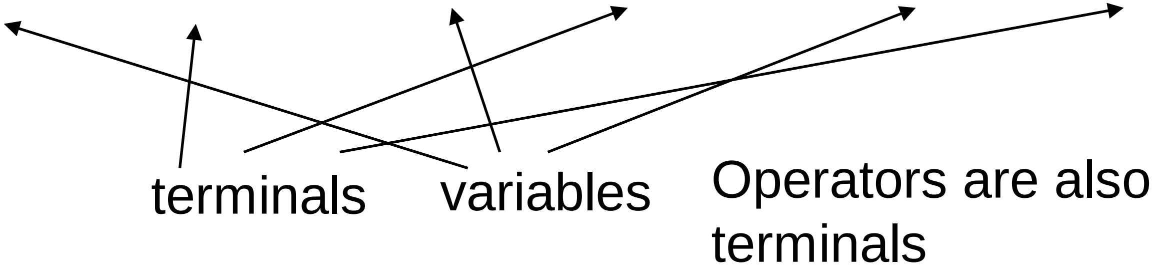
K. $A \Longrightarrow \alpha_k$

CFG conventions

- Terminal symbols <== a, b, c…
- Non-terminal symbols <== A,B,C, …
- Terminal <u>or</u> non-terminal symbols <== X,Y,Z
- Terminal strings <== w, x, y, z
- Arbitrary strings of terminals and non-terminals <== α, β, γ, ..

Syntactic Expressions in Programming Languages

*result = a*b + score + 10 * distance + c*



terminals

variables

Operators are also terminals

Regular languages have only terminals

- Reg expression = [a-z][a-z0-1]*
- If we allow only letters a & b, and 0 & 1 for constants (for simplification)
  - Regular expression = (a+b)(a+b+0+1)*

String membership

How to say if a string belong to the language defined by a CFG?

1. Derivation
   - Head to body
2. Recursive inference
   - Body to head

Both are equivalent forms

<u>Example:</u>

- w = 01110
- Is w a palindrome?

<u>G:</u>

A => 0A0 | 1A1 | 0 | 1 | ε

A => 0A0
=> 01A10
=> 01110

Simple Expressions…

- We can write a CFG for accepting simple expressions
- G = (V,T,P,S)
  - V = {E,F}
  - T = {0,1,a,b,+,*,(,)}
  - S = {E}
  - P:
    - E ==> E+E | E*E | (E) | F
    - F ==> aF | bF | 0F | 1F | a | b | 0 | 1

Generalization of derivation

- Derivation is *head* ==> *body*

- A==>X (A derives X in a single step)
- A $==>^*_G$ X (A derives X in a multiple steps)

- <u>Transitivity:</u>

  IFA $==>^*_G$B, and B $==>^*_G$C, THEN A $==>^*_G$ C

Context-Free Language

- The language of a CFG, G=(V,T,P,S), denoted by L(G), is the set of terminal strings that have a derivation from the start variable S.
  - $L(G) = \{ w \text{ in } T^* \mid S \text{ ==>}^*_G\ w \}$

Left-most & Right-most Derivation Styles

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | ε

Derive the string a*(ab+10) from G:

$E =^{*}=>_{G} a*(ab+10)$

Left-most derivation:

Always substitute leftmost variable

- E
- ==> E * E
- ==> F * E
- ==> aF * E
- ==> a * E
- ==> a * (E)
- ==> a * (E + E)
- ==> a * (F + E)
- ==> a * (aF + E)
- ==> a * (abF + E)
- ==> a * (ab + E)
- ==> a * (ab + F)
- ==> a * (ab + 1F)
- ==> a * (ab + 10F)
- ==> a * (ab + 10)

Right-most derivation:

Always substitute rightmost variable

- E
- ==> E * E
- ==> E * (E)
- ==> E * (E + E)
- ==> E * (E + F)
- ==> E * (E + 1F)
- ==> E * (E + 10F)
- ==> E * (E + 10)
- ==> E * (F + 10)
- ==> E * (aF + 10)
- ==> E * (abF + 0)
- ==> E * (ab + 10)
- ==> F * (ab + 10)
- ==> aF * (ab + 10)
- ==> a * (ab + 10)

Leftmost vs. Rightmost derivations

Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

True - will use parse trees to prove this

Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Yes – easy to prove (reverse direction)

Q3) Could there be words which have more than one leftmost (or rightmost) derivation?

Yes – depending on the grammar

How to prove that your CFGs are correct?

(using induction)

CFG & CFL

G_{pal}:
A => 0A0 | 1A1 | 0 | 1 | ε

- Theorem: A string w in (0+1)* is in L(G_{pal}), if and only if, w is a palindrome.
- Proof:
  - Use induction
    - on string length for the IF part
    - On length of derivation for the ONLY IF part

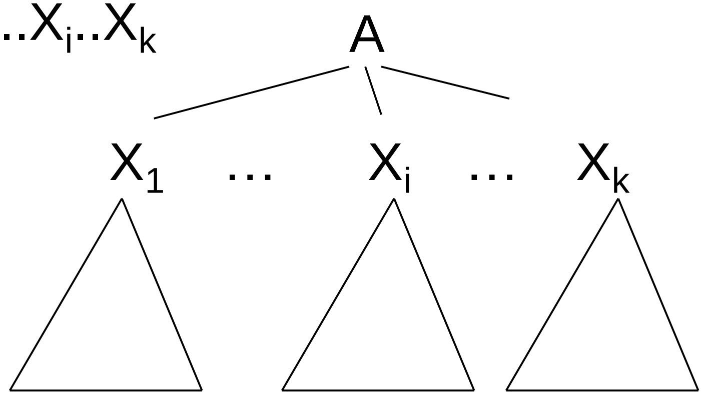
Parse trees

Parse Trees

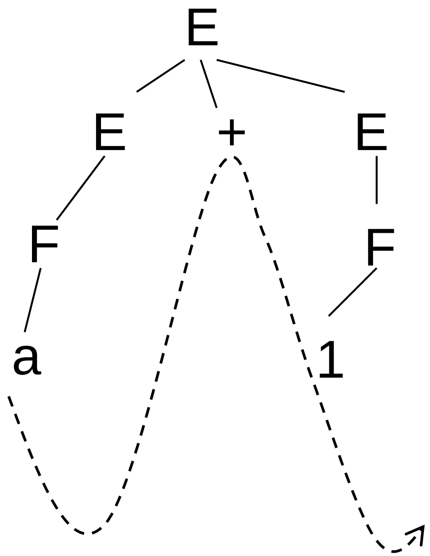
- Each CFG can be represented using a *parse tree:*
  - Each internal node is labeled by a variable in V
  - Each leaf is terminal symbol
  - For a production, $A==>X_1X_2\ldots X_k$, then any internal node labeled A has k children which are labeled from $X_1,X_2,\ldots X_k$ from left to right

Parse tree for production and all other subsequent productions:

$A ==> X_1..X_i..X_k$



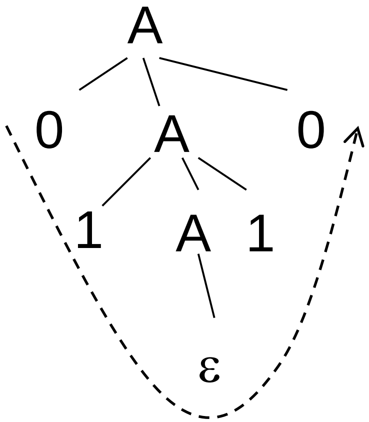
Examples



Parse tree for *a + 1*

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | 0 | 1 | a | b

Recursive inference



Derivation

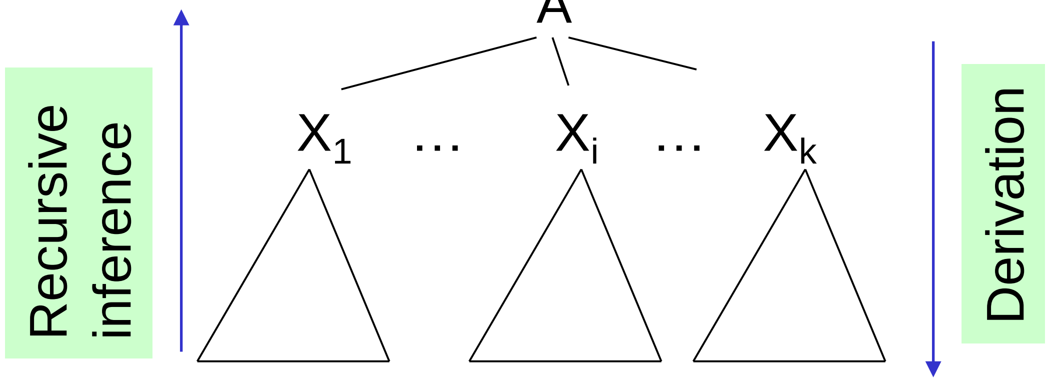
Parse tree for *0110*

G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences

Production:

$A ==> X_1..X_i..X_k$

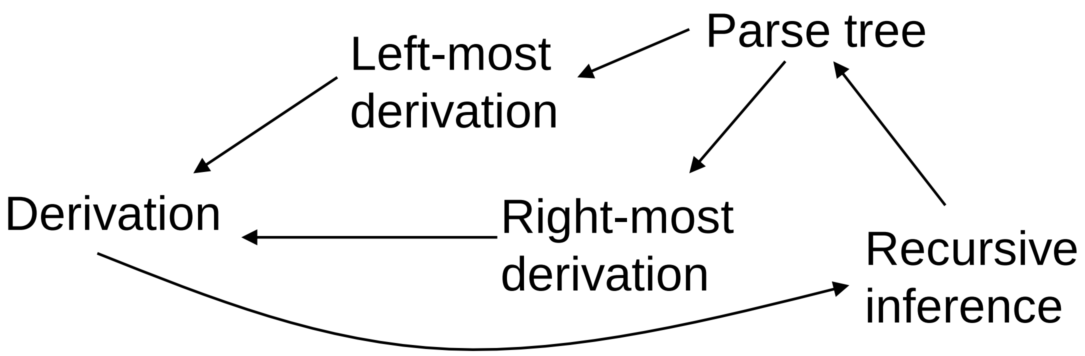


A

X_1 … X_i … X_k

Recursive inference

Derivation



Parse tree

Left-most derivation

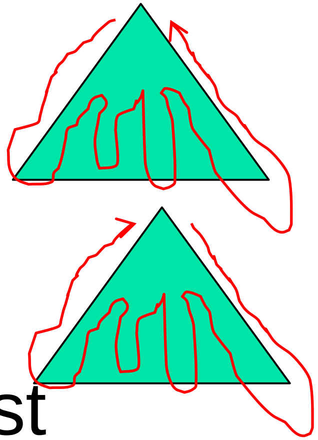
Derivation

Right-most derivation

Recursive inference

Interchangeability of different CFG representations

- Parse tree ==> left-most derivation
  - DFS left to right
- Parse tree ==> right-most derivation
  - DFS right to left
- ==> left-most derivation == right-most derivation
- Derivation ==> Recursive inference
  - Reverse the order of productions
- Recursive inference ==> Parse trees
  - bottom-up traversal of parse tree



Connection between CFLs and RLs

What kind of grammars result for regular languages?

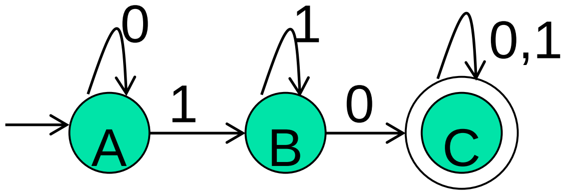
CFLs & Regular Languages

- A CFG is said to be *right-linear* if all the productions are one of the following two forms: *A ==> wB (or) A ==> w*

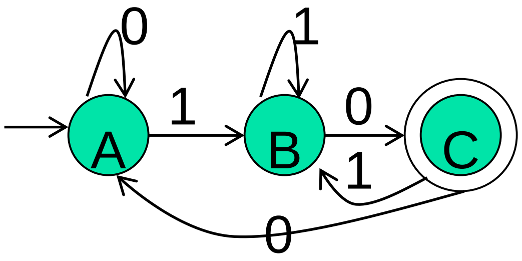
Where:
- A & B are variables,
- w is a string of terminals

- Theorem 1: Every right-linear CFG generates a regular language
- Theorem 2: Every regular language has a right-linear grammar
- Theorem 3: Left-linear CFGs also represent RLs

Some Examples



Right linear CFG?



Right linear CFG?

- A => 01B | C
  B => 11B | 0C | 1A
  C => 1A | 0 | 1

Finite Automaton?

Ambiguity in CFGs and CFLs

Ambiguity in CFGs

- A CFG is said to be *ambiguous* if there exists a string which has more than one left-most derivation

Example:

S ==> AS | ε

A ==> A1 | 0A1 | 01

LM derivation #1:

S => AS

=> 0A1S

=>0A11S

=> 00111S

=> 00111

LM derivation #2:

S => AS

=> A1S

=> 0A11S

=> 00111S

=> 00111

Input string: 00111

Can be derived in two ways

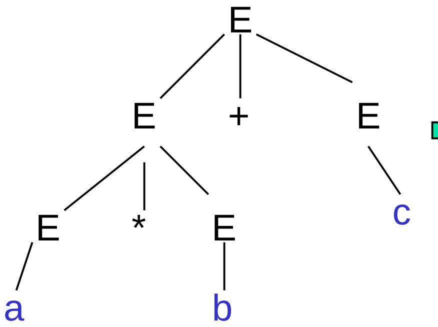
Why does ambiguity matter?

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

Values are different !!!

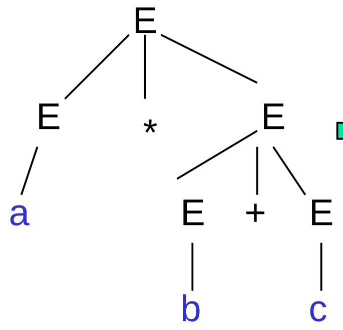
*string = a * b + c*

- <u>LM derivation #1:</u>
  - •E => E + E => E * E + E ==>* a * b + c



(a*b)+c

- <u>LM derivation #2</u>
  - •E => E * E => a * E => a * E + E ==>* a * b + c



a*(b+c)

The calculated value depends on which of the two parse trees is actually used.

Removing Ambiguity in Expression Evaluations

- It MAY be possible to remove ambiguity for some CFLs
  - E.g.,, in a CFG for expression evaluation by imposing rules & restrictions such as precedence
  - This would imply rewrite of the grammar

- Precedence: (), * , +

Modified unambiguous version:

E => E + T | T
T => T * F | F
F => I | (E)
I => a | b | c | 0 | 1

Ambiguous version:

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

How will this avoid ambiguity?

Inherently Ambiguous CFLs

- However, for some languages, it may not be possible to remove ambiguity

- A CFL is said to be *inherently ambiguous* if every CFG that describes it is ambiguous

Example:

- L = { $a^n b^n c^m d^m$ | n,m≥ 1} U {$a^n b^m c^m d^n$ | n,m≥ 1}
- L is inherently ambiguous
- Why?

Input string: $a^n b^n c^n d^n$

Summary

- Context-free grammars
- Context-free languages
- Productions, derivations, recursive inference, parse trees
- Left-most & right-most derivations
- Ambiguous grammars
- Removing ambiguity
- CFL/CFG applications
  - parsers, markup languages

Properties of Context-free Languages

Topics

1) Simplifying CFGs, Normal forms
2) Pumping lemma for CFLs
3) Closure and decision properties of CFLs

How to “simplify” CFGs?

Three ways to simplify/clean a CFG

(clean)

1. Eliminate *useless symbols*

(simplify)

2. Eliminate ε-productions $A \cancel{=>} \varepsilon$

3. Eliminate *unit productions* $A \cancel{=>} B$

Eliminating useless symbols

Grammar cleanup

Eliminating *useless symbols*

A symbol X is *reachable* if there exists:

- $S \Rightarrow^* \alpha X \beta$

A symbol X is *generating* if there exists:

- $X \Rightarrow^* w$,
  - for some $w \in T^*$

For a symbol X to be "useful", it has to be both reachable *and* generating

- $S \Rightarrow^* \alpha X \beta \Rightarrow^* w'$, for some $w' \in T^*$

reachable generating

Algorithm to detect useless symbols

1. First, eliminate all symbols that are *not* generating

2. Next, eliminate all symbols that are *not* reachable

Is the order of these steps important, or can we switch?

Example: Useless symbols

- S➔AB | a
- A➔ b

1. *A, S* are generating
2. *B* is *not generating* (and therefore B is useless)
3. ==> Eliminating B… (i.e., remove all productions that involve B)
   1. S➔ a
   2. A ➔ b
4. Now, A is *not reachable* and therefore is useless
5. Simplified G:
   1. S ➔ a

What would happen if you reverse the order:
i.e., test reachability before generating?

Will fail to remove:
A ➔ b

$X \xrightarrow{*} w$

Algorithm to find all generating symbols

- Given: G=(V,T,P,S)
- Basis:
  - Every symbol in T is obviously generating.
- Induction:
  - Suppose for a production A→ α, where α is generating
  - Then, A is also generating

$S \rightarrow^* \alpha X \beta$

Algorithm to find all reachable symbols

- Given: G=(V,T,P,S)
- Basis:
  - S is obviously reachable (from itself)
- Induction:
  - Suppose for a production A→ $\alpha_1 \alpha_2 \ldots \alpha_k$, where A is reachable
  - Then, all symbols on the right hand side, $\{\alpha_1, \alpha_2, \ldots \alpha_k\}$ are also reachable.

Eliminating ε-productions

A => ε

✗

What’s the point of removing ε-productions?

A → ε

Eliminating ε-productions

Caveat: It is *not* possible to eliminate ε-productions for languages which include ε in their word set

So we will target the grammar for the *rest of the language*

Theorem: If G=(V,T,P,S) is a CFG for a language L, then L\ {ε} has a CFG without ε-productions

Definition: A is “nullable” if A → ε*

- If A is nullable, then any production of the form “B→ CAD” can be simulated by:
  - B → CD | CAD
    - This can allow us to remove ε transitions for A

Algorithm to detect all nullable variables

- Basis:
  - If A→ ε is a production in G, then A is nullable
    (note: A can still have other productions)
- Induction:
  - If there is a production B→ $C_1C_2\ldots C_k$, where *every* C_i is nullable, then B is also nullable

Eliminating ε-productions

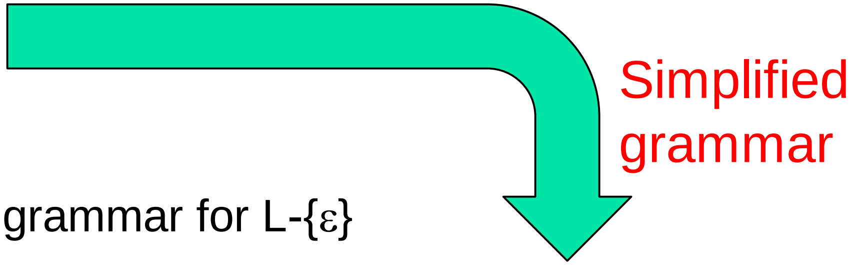
Given: $G=(V,T,P,S)$

Algorithm:

1. Detect all nullable variables in G
2. Then construct $G_1=(V,T,P_1,S)$ as follows:
   1. For each production of the form: $A \rightarrow X_1X_2\ldots X_k$, where $k\geq 1$, suppose ***m*** out of the ***k*** X_i's are nullable symbols
   2. Then G_1 will have $\mathbf{2^m}$ versions for this production
      1. i.e, all combinations where each X_i is either present or absent
   3. Alternatively, if a production is of the form: $A \rightarrow \varepsilon$, then remove it

Example: Eliminating ε-productions

- Let L be the language represented by the following CFG G:
  1. S→AB
  2. A→aAA | ε
  3. B→bBB | ε



Simplified grammar

Goal: To construct G1, which is the grammar for L-{ε}

- Nullable symbols: {A, B}

- G_1 can be constructed from G as follows:
  - B → b | bB | bB | bBB
    - ==> B → b | bB | bBB
  - Similarly, A → a | aA | aAA
  - Similarly, S → A | B | AB

- Note: $L(G) = L(G_1)$ U {ε}

G_1:
- S → A | B | AB
- A → a | aA | aAA
- B → b | bB | bBB

+

- S → ε

Eliminating unit productions

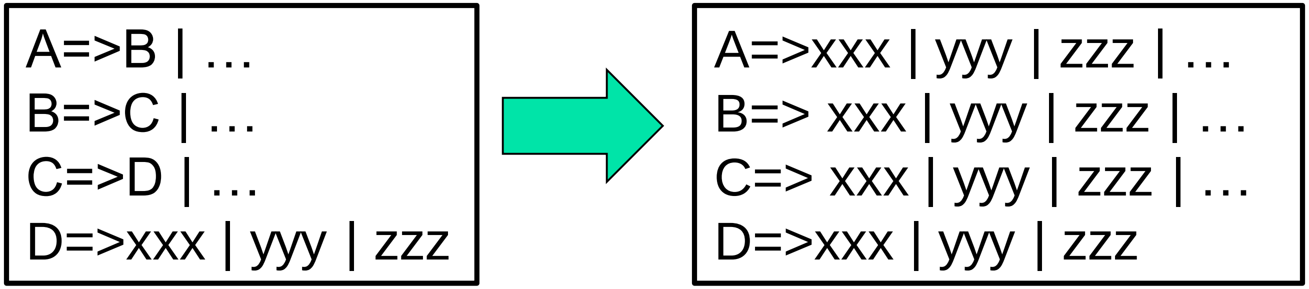
A => B

B has to be a variable

What's the point of removing unit transitions ?

Will save #substitutions

E.g.,



A=>B | …
B=>C | …
C=>D | …
D=>xxx | yyy | zzz

A=>xxx | yyy | zzz | …
B=> xxx | yyy | zzz | …
C=> xxx | yyy | zzz | …
D=>xxx | yyy | zzz

before

after

A ➔ B

Eliminating unit productions

- Unit production is one which is of the form A➔ B, where both A & B are variables
- E.g.,
  1. E ➔ T | E+T
  2. T ➔ F | T*F
  3. F ➔ I | (E)
  4. I ➔ a | b | Ia | Ib | I0 | I1
- How to eliminate unit productions?
  - Replace E➔ T with E ➔ F | T*F
  - Then, upon recursive application wherever there is a unit production:
    - E➔ F | T*F | E+T (substituting for T)
    - E➔ I | (E) | T*F| E+T (substituting for F)
    - E➔ a | b | Ia | Ib | I0 | I1 | (E) | T*F | E+T (substituting for I)
    - Now, E has no unit productions
  - Similarly, eliminate for the remainder of the unit productions

The **Unit Pair Algorithm**: to remove unit productions

- Suppose A➔B$_1$ ➔B$_2$ ➔ … ➔ B$_n$ ➔ α
- Action: Replace all intermediate productions to produce α directly
  - i.e., A➔ α; B$_1$➔ α; … B$_n$ ➔ α;

Definition: (A,B) to be a ***"unit pair"*** if A➔*B

- We can find all unit pairs inductively:
  - Basis: Every pair (A,A) is a unit pair (by definition). Similarly, if A➔B is a production, then (A,B) is a unit pair.
  - Induction: If (A,B) and (B,C) are unit pairs, and A➔C is also a unit pair.

The Unit Pair Algorithm: to remove unit productions

Input: G=(V,T,P,S)

Goal: to build G_1=(V,T,P_1,S) devoid of unit productions

Algorithm:

1. Find all unit pairs in G
2. For each unit pair (A,B) in G:
    1. Add to P_1 a new production A➔α, for every B➔α which is a *non-unit* production
    2. If a resulting production is already there in P, then there is no need to add it.

Example: eliminating unit productions

G:

1. E → T | E+T
2. T → F | T*F
3. F → I | (E)
4. I → a | b | Ia | Ib | I0 | I1

Unit pairs	*Only non-unit productions to be added to P_1*
(E,E)	E → E+T
(E,T)	E → T*F
(E,F)	E → (E)
(E,I)	E → a|b|Ia | Ib | I0 | I1
(T,T)	T → T*F
(T,F)	T → (E)
(T,I)	T → a|b| Ia | Ib | I0 | I1
(F,F)	F → (E)
(F,I)	F → a| b| Ia | Ib | I0 | I1
(I,I)	I → a| b | Ia | Ib | I0 | I1

G_1:

1. E → E+T | T*F | (E) | a| b | Ia | Ib | I0 | I1
2. T → T*F | (E) | a| b | Ia | Ib | I0 | I1
3. F → (E) | a| b | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Putting all this together…

- Theorem: If G is a CFG for a language that contains at least one string other than ε, then there is another CFG G_1, such that $L(G_1)=L(G) - \varepsilon$, *and* G_1 has:
  - no ε -productions
  - no unit productions
  - no useless symbols

- Algorithm:

Step 1) eliminate ε -productions
Step 2) eliminate unit productions
Step 3) eliminate useless symbols

Again,
the order is
important!

Why?

Normal Forms

Why normal forms?

- If all productions of the grammar could be expressed in the same form(s), then:

   a. It becomes easy to design algorithms that use the grammar

   b. It becomes easy to show proofs and properties

Chomsky Normal Form (CNF)

Let G be a CFG for some L-{ε}

Definition:

G is said to be in ***Chomsky Normal Form*** *if all its productions are in one of the following two forms:*

i. ***A → BC*** *where A,B,C are variables, or*

ii. ***A → a*** *where a is a terminal*

- *G has no useless symbols*
- *G has no unit productions*
- *G has no ε-productions*

CNF checklist

Is this grammar in CNF?

G_1:

1. E → E+T | T*F | (E) | Ia | Ib | I0 | I1
2. T → T*F | (E) | Ia | Ib | I0 | I1
3. F → (E) | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Checklist:

- G has no ε-productions ✓
- G has no unit productions ✓
- G has no useless symbols ✓
- But…
  - the normal form for productions is violated

➡ So, the grammar is not in CNF

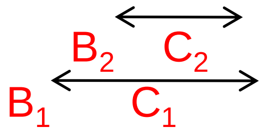
How to convert a G into CNF?

- Assumption: G has no ε-productions, unit productions or useless symbols

1) For every terminal ***a*** that appears in the body of a production:
   i. create a unique variable, say X_a, with a production $X_a \rightarrow a$, and
   ii. replace all other instances of a in G by X_a

2) Now, all productions will be in one of the following two forms:
   - $A \rightarrow B_1B_2\ldots B_k$ (k≥3) or $A \rightarrow a$

3) Replace each production of the form $A \rightarrow B_1B_2B_3\ldots B_k$ by:



B_2 C_2

B_1 C_1

and so on…

- $A \rightarrow B_1C_1$ $C_1 \rightarrow B_2C_2$ … $C_{k-3} \rightarrow B_{k-2}C_{k-2}$ $C_{k-2} \rightarrow B_{k-1}B_k$

Example #1

G in CNF:

G:

S => AS | BABC

A => A1 | 0A1 | 01

B => 0B | 0

C => 1C | 1

⇨

X_0 => 0

X_1 => 1

S => AS | BY_1

Y_1 => AY_2

Y_2 => BC

A => AX_1 | X_0Y_3 | X_0X_1

Y_3 => AX_1

B => X_0B | 0

C => X_1C | 1

All productions are of the form: A=>BC or A=>a

Example #2

G:

1. E → E+T | T*F | (E) | Ia | Ib | I0 | I1
2. T → T*F | (E) | Ia | Ib | I0 | I1
3. F → (E) | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Step (1)

1. $E \rightarrow EX_{+}T \mid TX_{*}F \mid X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
2. $T \rightarrow TX_{*}F \mid X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
3. $F \rightarrow X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
4. $I \rightarrow X_{a} \mid X_{b} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
5. $X_{+} \rightarrow +$
6. $X_{*} \rightarrow *$
7. $X_{+} \rightarrow +$
8. $X_{(} \rightarrow ($
9. …….

Step (2)

1. $E \rightarrow EC_{1} \mid TC_{2} \mid X_{(}C_{3} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
2. $C_{1} \rightarrow X_{+}T$
3. $C_{2} \rightarrow X_{*}F$
4. $C_{3} \rightarrow EX_{)}$
5. T → ………
6. ….

Languages with ε

- For languages that include ε,
  - Write down the rest of grammar in CNF
  - Then add production "S => ε" at the end

E.g., consider:

G:

S => AS | BABC

A => A1 | 0A1 | 01 | ε

B => 0B | 0 | ε

C => 1C | 1 | ε

G in CNF:

X_0 => 0
X_1 => 1

S => AS | BY_1 **| ε**

Y_1 => AY_2
Y_2 => BC

A => AX_1 | X_0Y_3 | X_0X_1

Y_3 => AX_1

B => X_0B | 0

C => X_1C | 1

Other Normal Forms

- Griebach Normal Form (GNF)
  - All productions of the form

$$A \Longrightarrow a\ \alpha$$

Return of the Pumping Lemma !!

Think of languages that cannot be CFL

== think of languages for which a stack will not be enough

e.g., the language of strings of the form *ww*

Why pumping lemma?

- A result that will be useful in proving languages that *are not* CFLs
  - (just like we did for regular languages)

- But before we prove the pumping lemma for CFLs ….
  - Let us first prove an important property about parse trees

Observe that any parse tree generated by a CNF will be a binary tree, where all internal nodes have exactly two children (except those nodes connected to the leaves).

The “parse tree theorem”

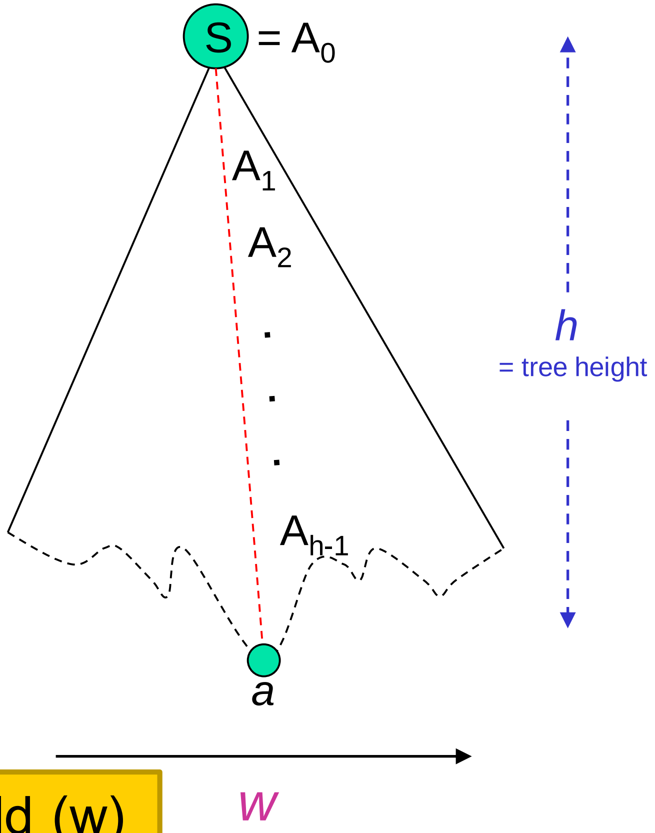
Given:

- Suppose we have a parse tree for a string ***w***, according to a CNF grammar, G=(V,T,P,S)
- Let *h* be the height of the parse tree

Implies:

- $|w| \leq 2^{h-1}$

Parse tree for w



In other words, a CNF parse tree’s string yield (w) can no longer be 2^{h-1}

To show: ***$|w| \leq 2^{h-1}$***

Proof…The size of parse trees

Proof: (using induction on h)

Basis: h = 1

- ➔ Derivation will have to be "S➔a"
- ➔ $|w| = 1 = 2^{1-1}$.

Ind. Hyp: h = k-1

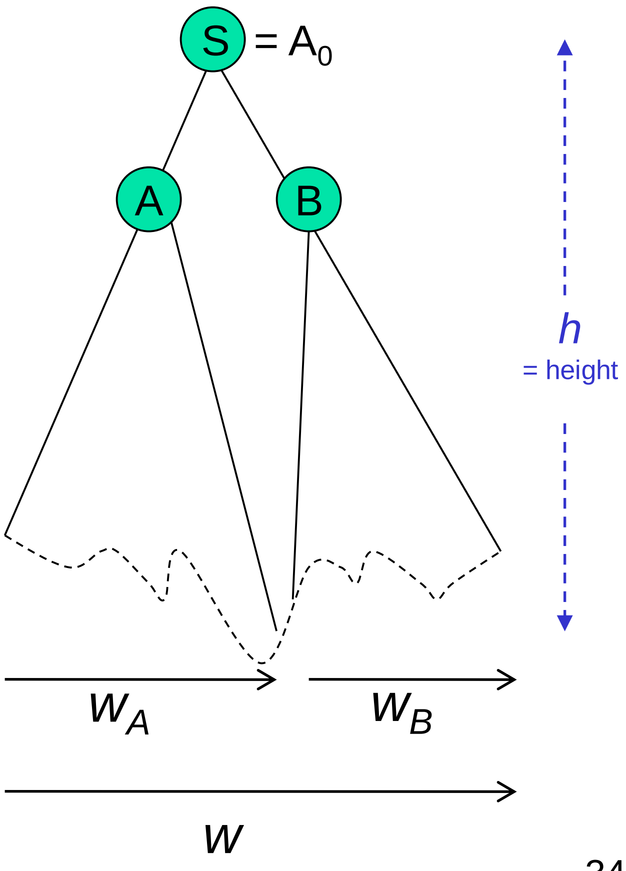
- ➔ $|w| \leq 2^{k-2}$

Ind. Step: h = k

S will have exactly two children: S➔AB

- ➔ Heights of A & B subtrees are at most h-1
- ➔ $w = w_A w_B$, where $|w_A| \leq 2^{k-2}$ and $|w_B| \leq 2^{k-2}$
- ➔ $|w| \leq 2^{k-1}$

Parse tree for w



Implication of the Parse Tree Theorem (assuming CNF)

Fact:

- If the height of a parse tree is h, then
  - ==> $|w| \leq 2^{h-1}$

Implication:

- **If $|w| \geq 2^m$, then**
  - **Its parse tree's height is *at least* m+1**

The Pumping Lemma for CFLs

Let L be a CFL.

Then there exists a constant N, s.t.,

- if $z \in L$ s.t. $|z| \geq N$, then we can write $z=uvwxy$, such that:
  1. $|vwx| \leq N$
  2. $vx \neq \varepsilon$
  3. For all $k \geq 0$: $uv^kwx^ky \in L$

Note: we are pumping in two places (v & x)

Proof: Pumping Lemma for CFL

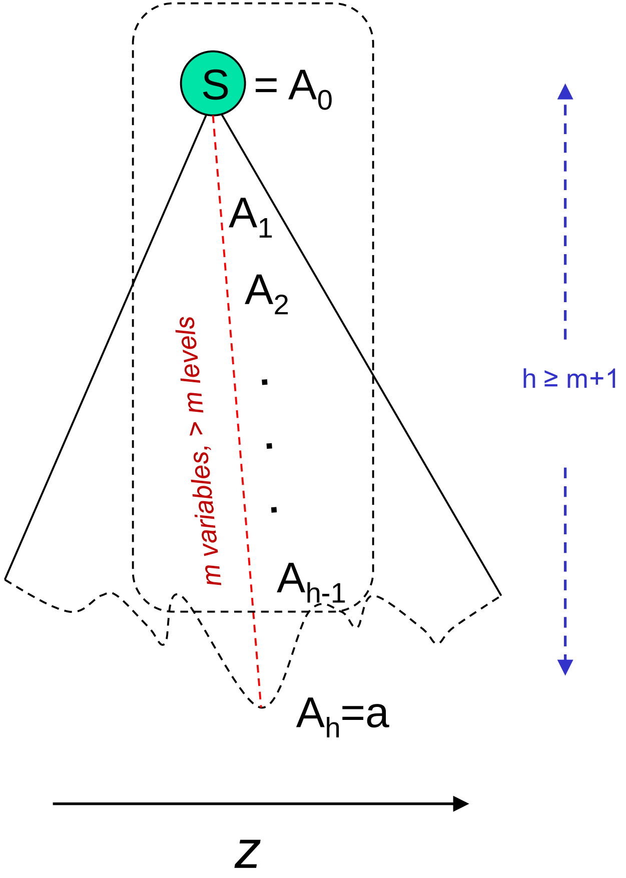
- If L=Φ or contains only ε, then the lemma is trivially satisfied (as it cannot be violated)

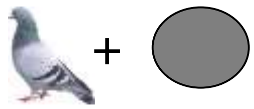
- For any other L which is a CFL:
  - Let G be a CNF grammar for L
  - Let m = number of variables in G
  - Choose $N=2^m$.
  - Pick any $z \in L$ s.t. $|z| \geq N$

    ➔ the parse tree for z should have a height ≥ m+1
    (by the parse tree theorem)

Parse tree for z

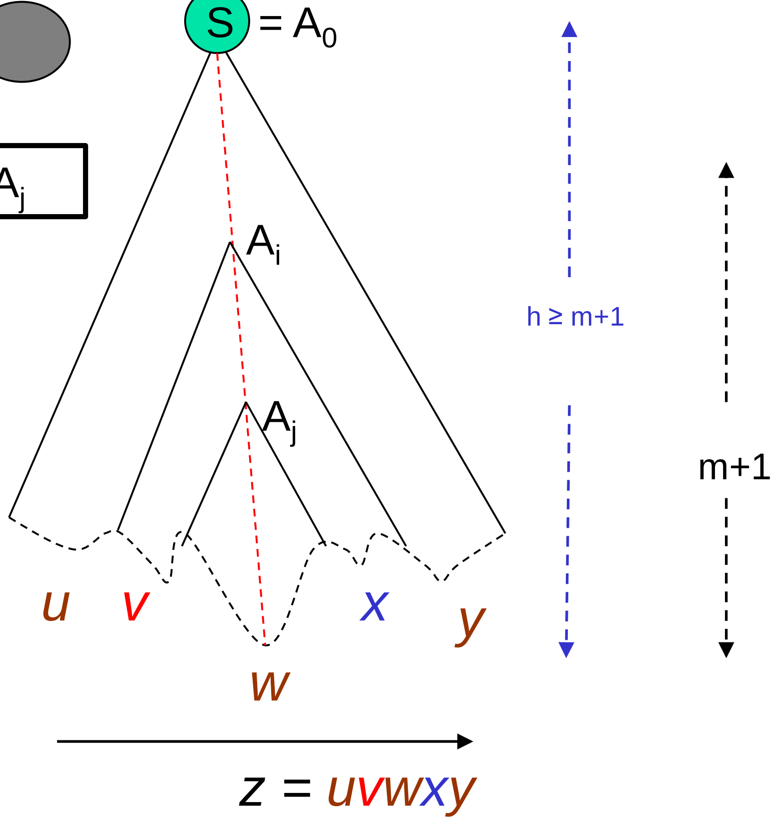
Meaning:
Repetition in the last m+1 variables





$A_i = A_j$

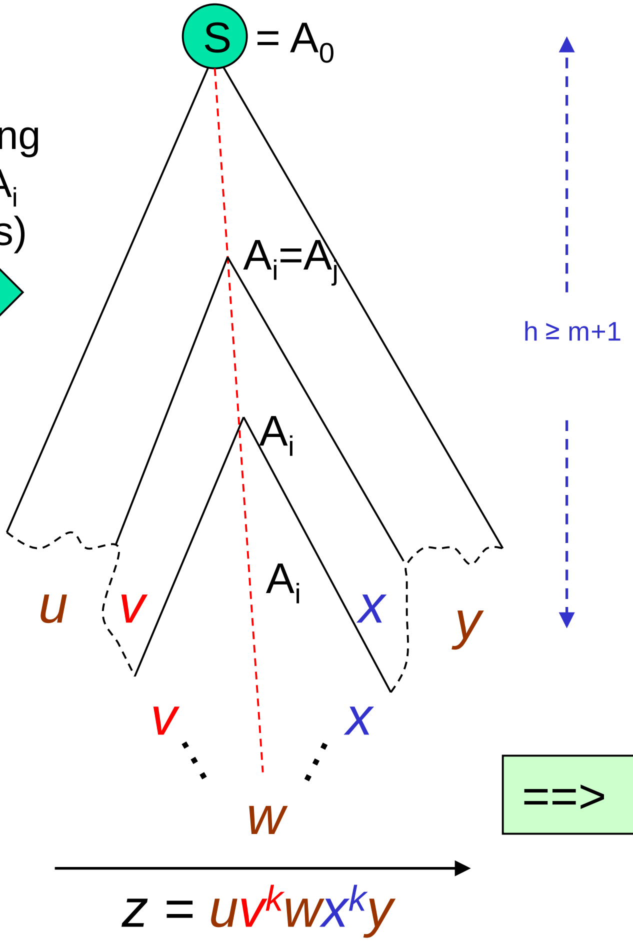
h-m≤ i < j ≤ h



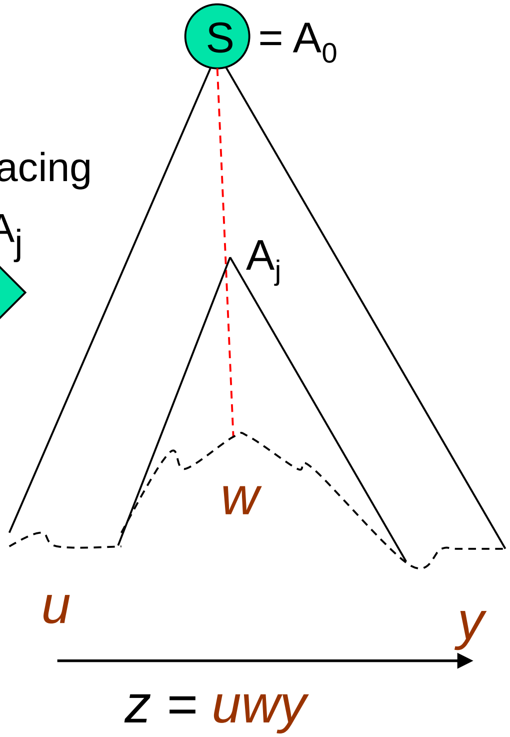
- Therefore, vx≠ε

Extending the parse tree…

Replacing A_j with A_i (k times)



Or, replacing A_i with A_j



==> For all k≥0: $uv^kwx^ky \in L$

Proof contd..

- Also, since A_i's subtree no taller than m+1

  ==> the string generated under A_i's subtree, which is vwx, cannot be longer than 2^m (=N)

  But, 2^m =N

  ==> $|vwx| \leq N$

  This completes the proof for the pumping lemma.

Application of Pumping Lemma for CFLs

Example 1: $L = \{a^m b^m c^m \mid m>0\}$

Claim: L is not a CFL

Proof:

- Let N <== P/L constant
- Pick $z = a^N b^N c^N$
- Apply pumping lemma to z and show that there exists at least one other string constructed from z (obtained by pumping up or down) that is $\notin L$

Proof contd…

- z = uvwxy
- As $z = a^N b^N c^N$ *and* |vwx| ≤ N *and* vx≠ε
  - ==> v, x cannot contain all three symbols (a,b,c)
  - ==> we can pump up or pump down to build another string which is ∉ L

Example #2 for P/L application

- $L = \{ ww \mid w \text{ is in } \{0,1\}^* \}$

- Show that L is not a CFL

  - Try string $z = 0^N 0^N$
    - what happens?
  - Try string $z = 0^N 1^N 0^N 1^N$
    - what happens?

Example 3

- L = { 0^{k^2} | k is any integer)
- Prove L is not a CFL using Pumping Lemma

Example 4

- $L = \{a^i b^j c^k \mid i<j<k \}$

- Prove that L is not a CFL

CFL Closure Properties

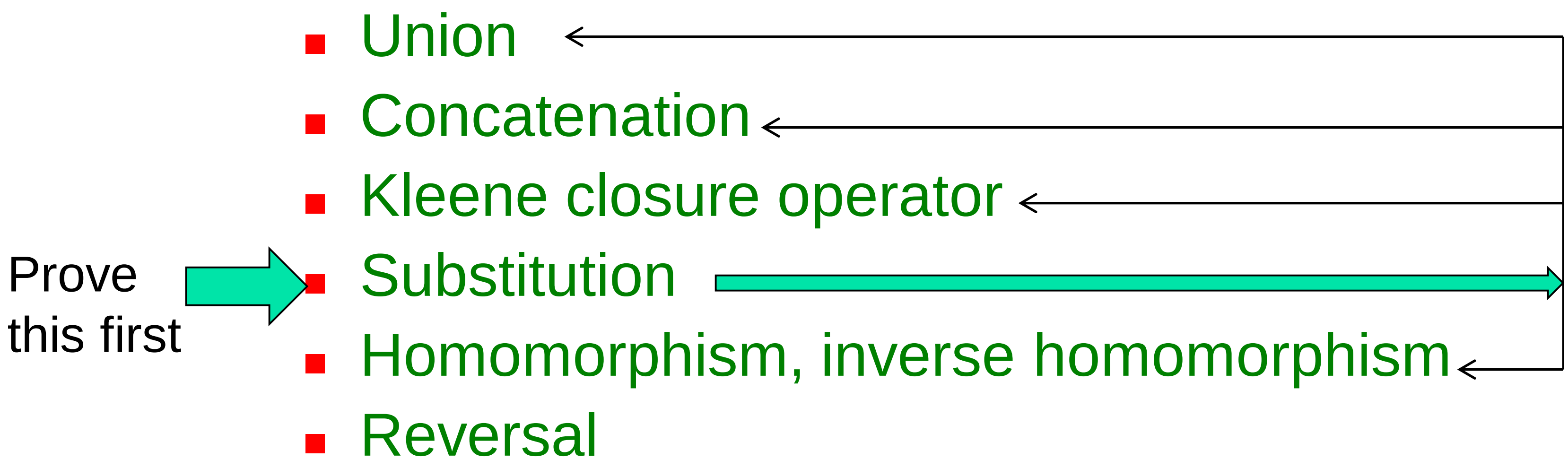
Closure Property Results

- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
  - reversal
- CFLs are *not* closed under:
  - Intersection
  - Difference
  - Complementation

Note: Reg languages are closed under these operators

Strategy for Closure Property Proofs

- First prove "closure under **substitution**"
- Using the above result, prove other closure properties
- CFLs are closed under:



  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
  - Reversal

Prove this first

Note: s(L) can use a different alphabet

The ***Substitution*** operation

For each $a \in \sum$, then let s(a) be a language

If $w=a_1a_2\ldots a_n \in L$, then:

- $s(w) = \{ x_1x_2 \ldots \} \in s(L)$, s.t., $x_i \in s(a_i)$

Example:

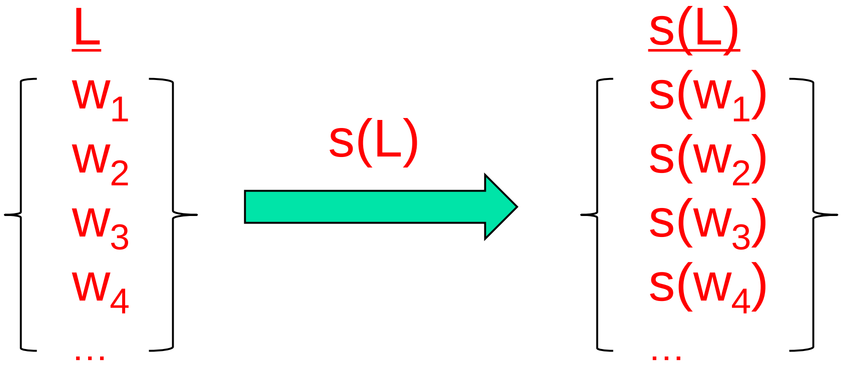
- Let $\sum=\{0,1\}$
- Let: $s(0) = \{a^nb^n \mid n \geq 1\}$, $s(1) = \{aa,bb\}$
- If w=01, s(w)=s(0).s(1)
  - E.g., s(w) contains $a^1 b^1$ aa, $a^1 b^1$bb,
    $a^2 b^2$ aa, $a^2 b^2$bb,
    … and so on.

CFLs are closed under Substitution

IF L is a CFL and a substititution defined on L, s(L), is s.t., s(a) is a CFL for every symbol a, THEN:

- s(L) is also a CFL

What is s(L)?



L

w_1

w_2

w_3

w_4

…

s(L)

s(L)

$s(w_1)$

$s(w_2)$

$s(w_3)$

$s(w_4)$

…

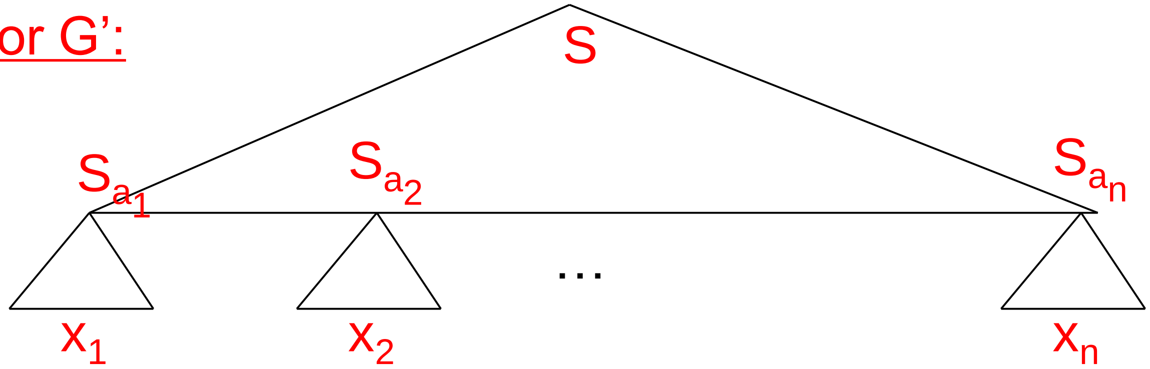
Note: each s(w) is itself a set of strings

CFLs are closed under *Substitution*

- G=(V,T,P,S) : CFG for L
- Because every s(a) is a CFL, there is a CFG for each s(a)
  - Let $G_a = (V_a,T_a,P_a,S_a)$
- Construct G'=(V',T',P',S) for s(L)

- P' consists of:
  - The productions of P, but with every occurrence of terminal "a" in their bodies replaced by S_a.
  - All productions in any P_a, for any $a \in \Sigma$

Parse tree for G':



Substitution of a CFL: example

- Let L = language of binary palindromes s.t., substitutions for 0 and 1 are defined as follows:
  - s(0) = $\{a^n b^n \mid n \geq 1\}$, s(1) = {xx,yy}
- Prove that s(L) is also a CFL.

CFG for L:

$S \Rightarrow 0S0|1S1|\varepsilon$

CFG for s(0):

$S_0 \Rightarrow aS_0b \mid ab$

CFG for s(1):

$S_1 \Rightarrow xx \mid yy$

Therefore, CFG for s(L):

$S \Rightarrow S_0SS_0 \mid S_1\ S\ S_1 \mid \varepsilon$

$S_0 \Rightarrow aS_0b \mid ab$

$S_1 \Rightarrow xx \mid yy$

CFLs are closed under ***union***

Let L_1 and L_2 be CFLs

To show: L_2 U L_2 is also a CFL

Let us show by using the result of *Substitution*

- Make a new language:
  - L_{new} = {a,b} s.t., s(a) = L_1 and s(b) = L_2

  ==> s(L_{new}) == same as == L_1 U L_2

- A more direct, alternative proof
  - Let S_1 and S_2 be the starting variables of the grammars for L_1 and L_2
    - Then, **S_{new} => S_1 | S_2**

CFLs are closed under ***concatenation***

- Let L_1 and L_2 be CFLs

Let us show by using the result of *Substitution*

- Make L_{new}= {ab} s.t.,

  s(a) = L_1 and s(b)= L_2

  ==> L_1 L_2 = s(L_{new})

- A proof without using substitution?

CFLs are closed under *Kleene Closure*

- Let L be a CFL

- Let L_{new} = {a}* and s(a) = L_1
  - Then, L* = s(L_{new})

We won’t use substitution to prove this result

CFLs are closed under *Reversal*

- Let L be a CFL, with grammar G=(V,T,P,S)
- For L^R, construct G^R=(V,T,P^R,S) s.t.,
  - If A==> α is in P, then:
    - A==> α^R is in P^R
    - (that is, reverse every production)

CFLs are *not* closed under Intersection

- Existential proof:
  - $L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$
  - $L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$
- Both L_1 and L_2 are CFLs
  - Grammars?
- But $L_1 \cap L_2$ *cannot* be a CFL
  - Why?
- We have an example, where intersection is not closed.
- Therefore, CFLs are not closed under intersection

CFLs are not closed under complementation

- Follows from the fact that CFLs are not closed under intersection

- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Logic: if CFLs were to be closed under complementation

→ the whole right hand side becomes a CFL (because CFL is closed for union)

→ the left hand side (intersection) is also a CFL

→ but we just showed CFLs are NOT closed under intersection!

→ CFLs *cannot* be closed under complementation.

CFLs are not closed under difference

- Follows from the fact that CFLs are not closed under complementation
- Because, if CFLs are closed under difference, then:
  - $\overline{L} = \Sigma^* - L$
  - So $\overline{L}$ has to be a CFL too
  - Contradiction

Decision Properties

- Emptiness test
  - Generating test
  - Reachability test
- Membership test
  - PDA acceptance

“Undecidable” problems for CFL

- Is a given CFG G ambiguous?
- Is a given CFL inherently ambiguous?
- Is the intersection of two CFLs empty?
- Are two CFLs the same?
- Is a given L(G) equal to $\sum^*$?

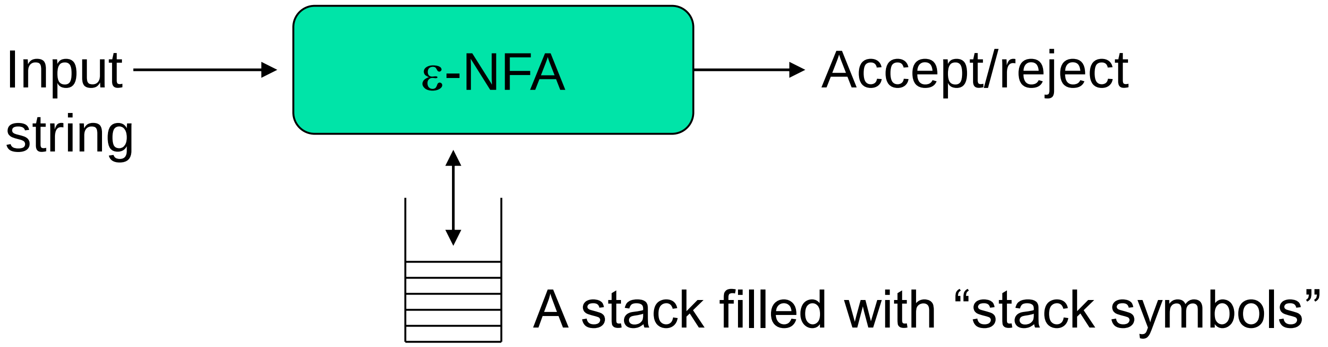
Summary

- Normal Forms
  - Chomsky Normal Form
  - Griebach Normal Form
  - Useful in proroving P/L
- Pumping Lemma for CFLs
  - Main difference: $z=uv^iwx^iy$
- Closure properties
  - Closed under: union, concatentation, reversal, Kleen closure, homomorphism, substitution
  - Not closed under: intersection, complementation, difference

Pushdown Automata (PDA)

PDA - the automata for CFLs

- What is?
  - FA to Reg Lang,   PDA is to CFL
- PDA == [ε -NFA + "a stack"]
- Why a stack?



Pushdown Automata - Definition

- A PDA P := ($Q,\sum,\Gamma, \delta,q_0,Z_0,F$):

Q:	states of the ε-NFA
∑:	input alphabet
Γ :	stack symbols
δ:	transition function
q_0:	start state
Z_0:	Initial stack top symbol
F:	Final/accepting states

old state input symb. Stack top new state(s) new Stack top(s)

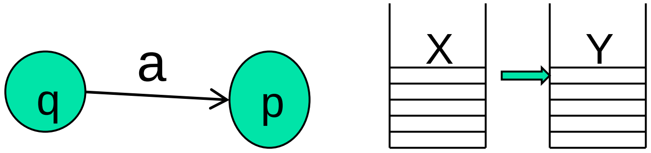
$$\delta : Q \times \Sigma \times \Gamma \Rightarrow Q \times \Gamma$$

δ : The Transition Function

δ(q,a,X) = {(p,Y), …}

Non-determinism

1. state transition from q to p
2. a is the next input symbol
3. X is the current stack *top* symbol
4. Y is the replacement for X; it is in Γ^* (a string of stack symbols)
   - i. Set Y = ε for: Pop(X)
   - ii. If Y=X: stack top is unchanged
   - iii. If $Y=Z_1Z_2 \dots Z_k$: X is popped and is replaced by Y in reverse order (i.e., Z_1 will be the new stack top)



	Y = ?	Action
i)	Y=ε	Pop(X)

ii)	Y=X	Pop(X)
		Push(X)

iii)	$Y=Z_1Z_2..Z_k$	Pop(X)
		Push(Z_k)
		Push(Z_{k-1})
		…
		Push(Z_2)
		Push(Z_1)

Example

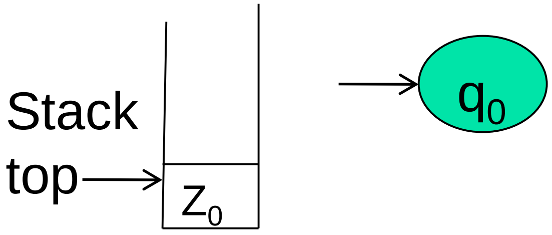
Let $L_{wwr} = \{ww^R \mid w \text{ is in } (0+1)^*\}$

- CFG for L_{wwr} : S==> 0S0 | 1S1 | ε
- PDA for L_{wwr} :
- $P := (Q,\sum, \Gamma, \delta,q_0,Z_0,F)$

$= (\{q_0, q_1, q_2\},\{0,1\},\{0,1,Z_0\},\delta,q_0,Z_0,\{q_2\})$

PDA for L_{wwr}

Initial state of the PDA:



1. $\delta(q_0,0, Z_0)=\{(q_0,0Z_0)\}$
2. $\delta(q_0,1, Z_0)=\{(q_0,1Z_0)\}$

First symbol push on stack

3. $\delta(q_0,0, 0)=\{(q_0,00)\}$
4. $\delta(q_0,0, 1)=\{(q_0,01)\}$
5. $\delta(q_0,1, 0)=\{(q_0,10)\}$
6. $\delta(q_0,1, 1)=\{(q_0,11)\}$

Grow the stack by pushing new symbols on top of old (w-part)

7. $\delta(q_0, \varepsilon, 0)=\{(q_1, 0)\}$
8. $\delta(q_0, \varepsilon, 1)=\{(q_1, 1)\}$
9. $\delta(q_0, \varepsilon, Z_0)=\{(q_1, Z_0)\}$

Switch to popping mode, nondeterministically (boundary between w and w^R)

10. $\delta(q_1,0, 0)=\{(q_1, \varepsilon)\}$
11. $\delta(q_1,1, 1)=\{(q_1, \varepsilon)\}$

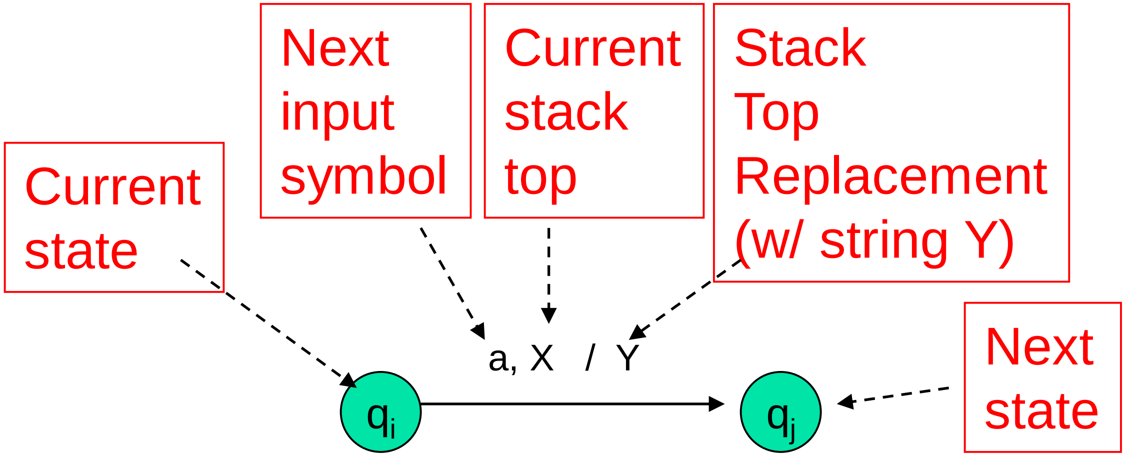
Shrink the stack by popping matching symbols (w^R-part)

12. $\delta(q_1, \varepsilon, Z_0)=\{(\mathbf{q_2}, Z_0)\}$

Enter acceptance state

PDA as a state diagram

$\delta(q_i, a, X) = \{(q_j, Y)\}$



Current state

Next input symbol

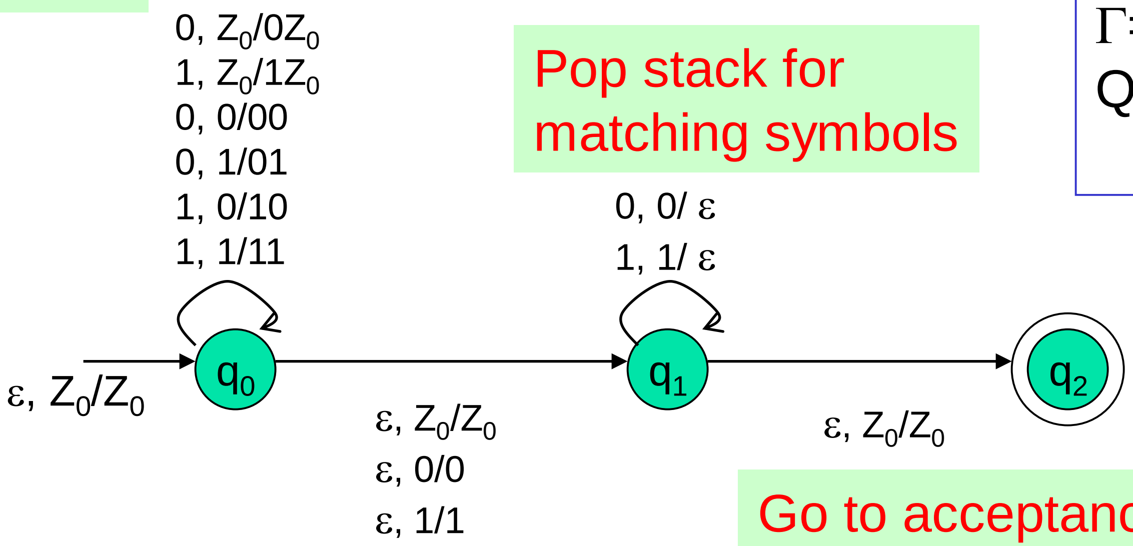
Current stack top

Stack Top Replacement (w/ string Y)

Next state

PDA for L_{wwr}: Transition Diagram

$\sum = \{0, 1\}$
$\Gamma = \{Z_0, 0, 1\}$
$Q = \{q_0, q_1, q_2\}$



Grow stack

Pop stack for matching symbols

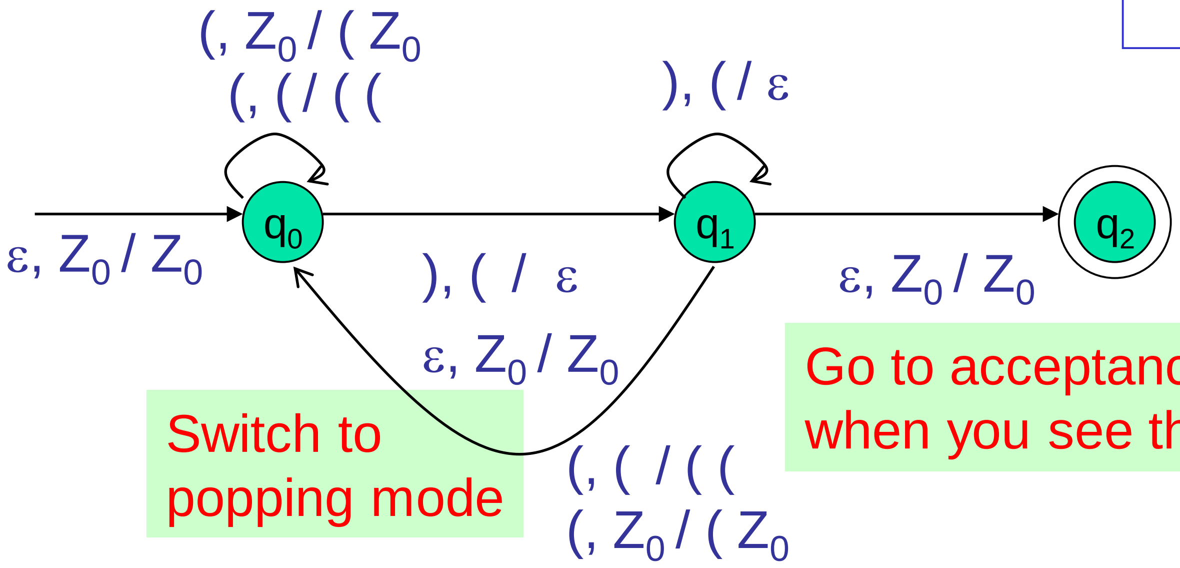
Switch to popping mode

Go to acceptance

This would be a non-deterministic PDA

Example 2: language of balanced paranthesis

$\Sigma = \{ (,) \}$
$\Gamma = \{Z_0, (\}$
$Q = \{q_0, q_1, q_2\}$



Grow stack

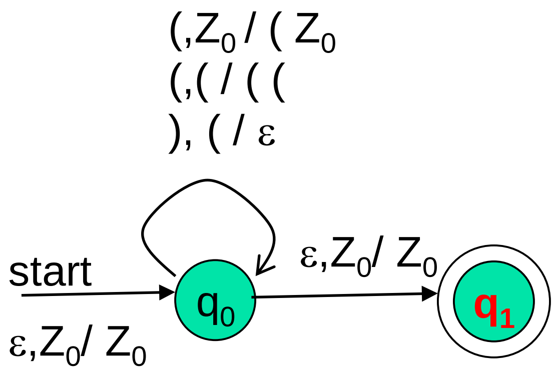
Pop stack for matching symbols

Switch to popping mode

Go to acceptance (by final state) when you see the stack bottom symbo

To allow adjacent blocks of nested paranthesis

Example 2: language of balanced paranthesis (another design)



$\sum$ = { (,) }
Γ= {Z_0, (}
Q = {q_0,q_1}

PDA’s Instantaneous Description (ID)

A PDA has a configuration at any given instance:

(q,w,y)

- q - current state
- w - remainder of the input (i.e., unconsumed part)
- y - current stack contents as a string from top to bottom of stack

If δ(q,a, X)={(p, A)} is a transition, then the following are also true:

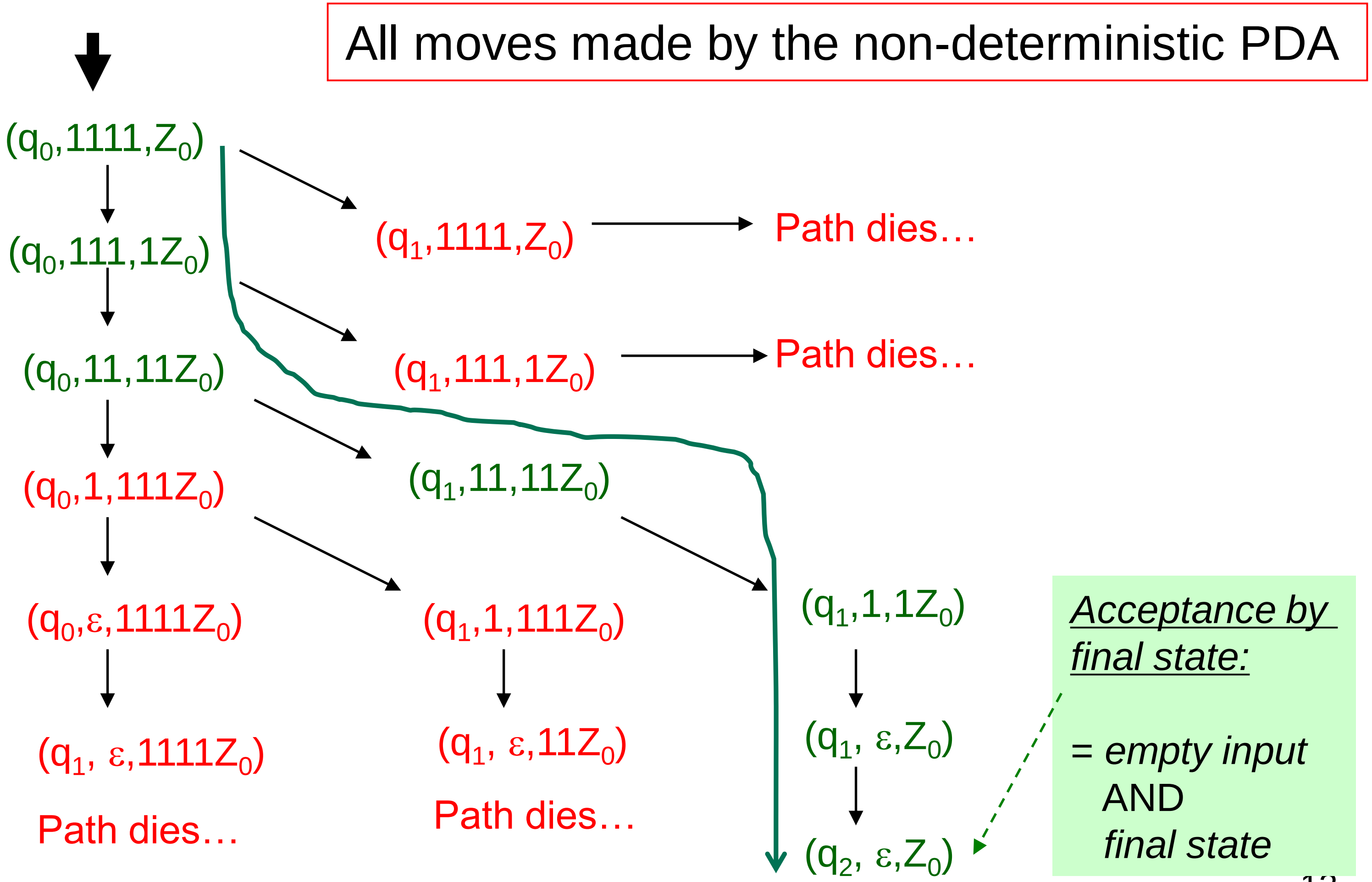
- (q, a, X) |--- (p,ε,A)
- (q, aw, XB) |--- (p,w,AB)

|--- sign is called a “turnstile notation” and represents one move

|---* sign represents a sequence of moves

How does the PDA for L_{wwr} work on input “1111”?

All moves made by the non-deterministic PDA



$(q_0,1111,Z_0)$

$(q_0,111,1Z_0)$

$(q_0,11,11Z_0)$

$(q_0,1,111Z_0)$

$(q_0,\varepsilon,1111Z_0)$

$(q_1, \varepsilon,1111Z_0)$

Path dies…

$(q_1,1111,Z_0)$ Path dies…

$(q_1,111,1Z_0)$ Path dies…

$(q_1,11,11Z_0)$

$(q_1,1,111Z_0)$

$(q_1, \varepsilon,11Z_0)$

Path dies…

$(q_1,1,1Z_0)$

(q_1, ε,Z_0)

(q_2, ε,Z_0)

Acceptance by final state:

= *empty input*
AND
final state

There are two types of PDAs that one can design:
those that accept by final state or by empty stack

Acceptance by…

- ***PDAs that accept by final state:***
  - For a PDA P, the language accepted by P, denoted by L(P) by *final state*, is:
    - $\{w \mid (q_0,w,Z_0) \vdash^{*} (q,\varepsilon, A)\}$, s.t., $q \in F$

Checklist:
- input exhausted?
- in a final state?

- ***PDAs that accept by empty stack:***
  - For a PDA P, the language accepted by P, denoted by N(P) by *empty stack*, is:
    - $\{w \mid (q_0,w,Z_0) \vdash^{*} (q, \varepsilon, \varepsilon)\}$, for any $q \in Q$.

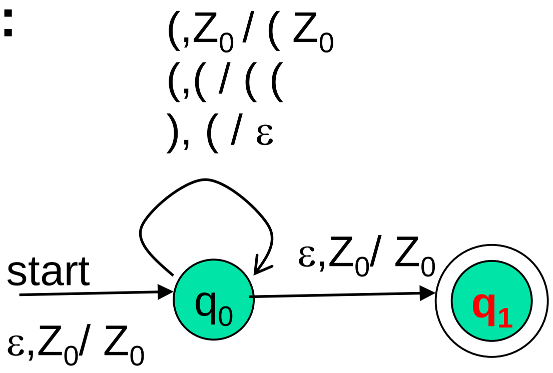
Q) Does a PDA that accepts by empty stack
need any final state specified in the design?

Checklist:
- input exhausted?
- is the stack empty?

Example: L of balanced parenthesis

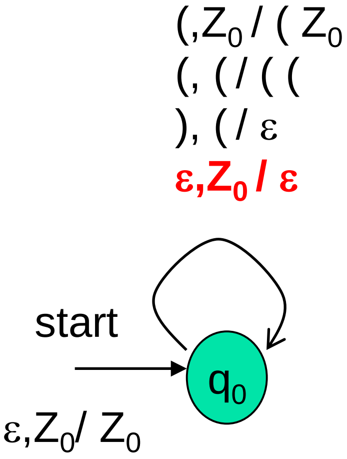
PDA that accepts by final state

P_F:



An equivalent PDA that accepts by empty stack

P_N:



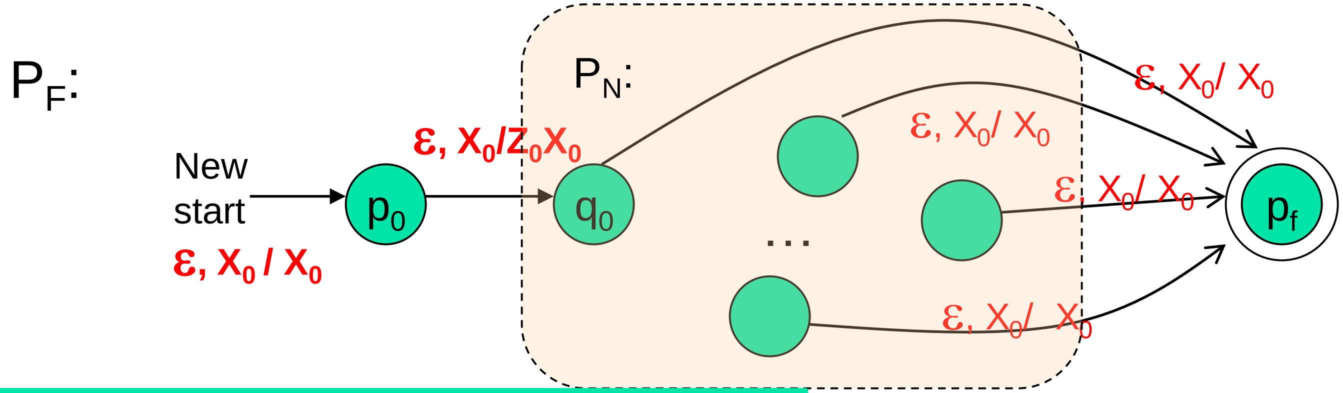
How will these two PDAs work on the input: ((( )) ( )) ( )

PDAs accepting by final state and empty stack are equivalent

- P_F <= PDA accepting by final state
  - $P_F = (Q_F, \sum, \Gamma, \delta_F, q_0, Z_0, F)$
- P_N <= PDA accepting by empty stack
  - $P_N = (Q_N, \sum, \Gamma, \delta_N, q_0, Z_0)$
- Theorem:
  - *(P_N==> P_F)* For every P_N, there exists a P_F s.t. $L(P_F)=L(P_N)$
  - *(P_F==> P_N)* For every P_F, there exists a P_N s.t. $L(P_F)=L(P_N)$

P_N==> P_F construction

- Whenever P_N's stack becomes empty, make P_F go to a final state without consuming any addition symbol
- To detect empty stack in P_N: P_F pushes a new stack symbol X_0 (not in Γ of P_N) initially before simultating P_N


P_F:
P_N:
New start
ε, X_0 / X_0
ε, X_0/Z_0X_0
p_0
q_0
ε, X_0/ X_0
ε, X_0/ X_0
ε, X_0/ X_0
…
ε, X_0/ X_0
p_f

$P_F = (Q_N \cup \{p_0,p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$

Example: Matching parenthesis “(” “)”

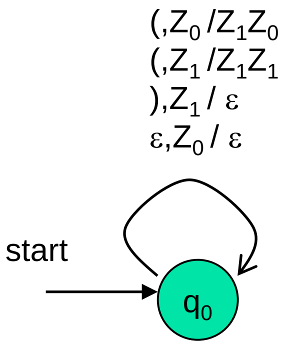
P_N: $(\ \{q_0\}, \{(,)\}, \{Z_0,Z_1\}, \delta_N, q_0, Z_0\ )$

δ_N:

$\delta_N(q_0,(,Z_0) = \{\ (q_0,Z_1Z_0)\ \}$
$\delta_N(q_0,(,Z_1) = \{\ (q_0, Z_1Z_1)\ \}$

$\delta_N(q_0,),Z_1) = \{\ (q_0, \varepsilon)\ \}$

$\delta_N(q_0, \varepsilon,Z_0) = \{\ (q_0, \varepsilon)\ \}$

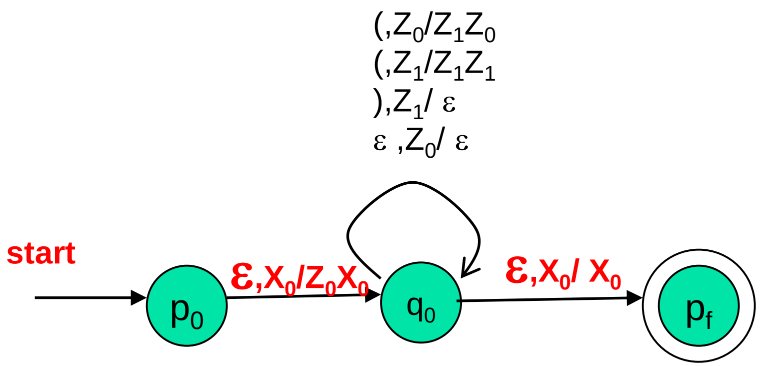


Accept by empty stack

P_f: $(\ \{p_0,q_0\ ,p_f\}, \{(,)\}, \{X_0,Z_0,Z_1\}, \delta_f, p_0, X_0\ , p_f)$

δ_f:

$\delta_f(p_0, \varepsilon,X_0) = \{\ (q_0,Z_0)\ \}$
$\delta_f(q_0,(,Z_0) = \{\ (q_0,Z_1\ Z_0)\ \}$
$\delta_f(q_0,(,Z_1) = \{\ (q_0, Z_1Z_1)\ \}$
$\delta_f(q_0,),Z_1) = \{\ (q_0, \varepsilon)\ \}$
$\delta_f(q_0, \varepsilon,Z_0) = \{\ (q_0, \varepsilon)\ \}$
$\delta_f(p_0, \varepsilon,X_0) = \{\ (p_f, X_0\ )\ \}$



Accept by final state

How to convert an final state PDA into an empty stack PDA?

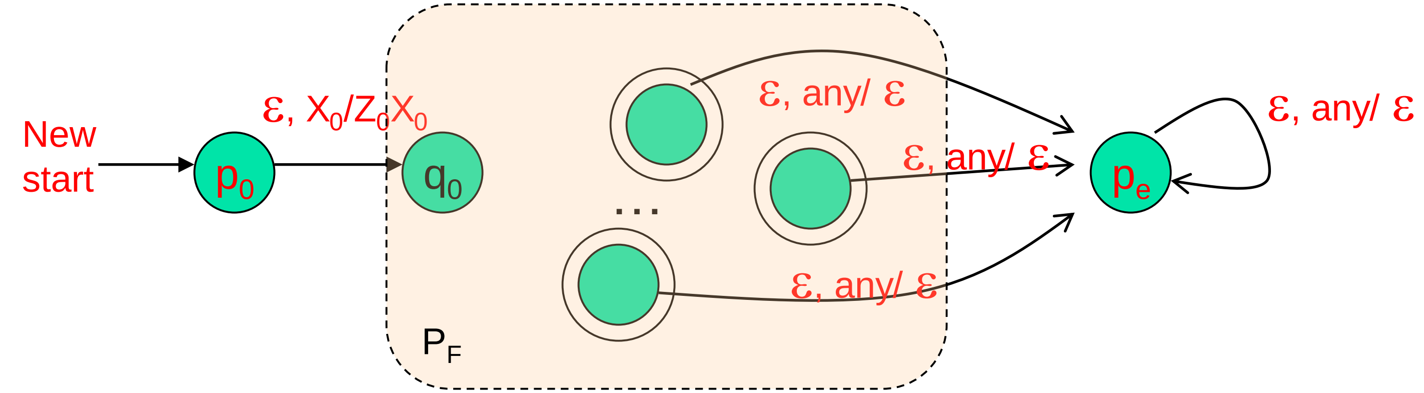
P_F==> P_N construction

- Main idea:
  - Whenever P_F reaches a final state, just make an ε -transition into a new end state, clear out the stack and accept
  - Danger: What if P_F design is such that it clears the stack midway *without* entering a final state?

    ➔ to address this, add a new start symbol X_0 (not in Γ of P_F)

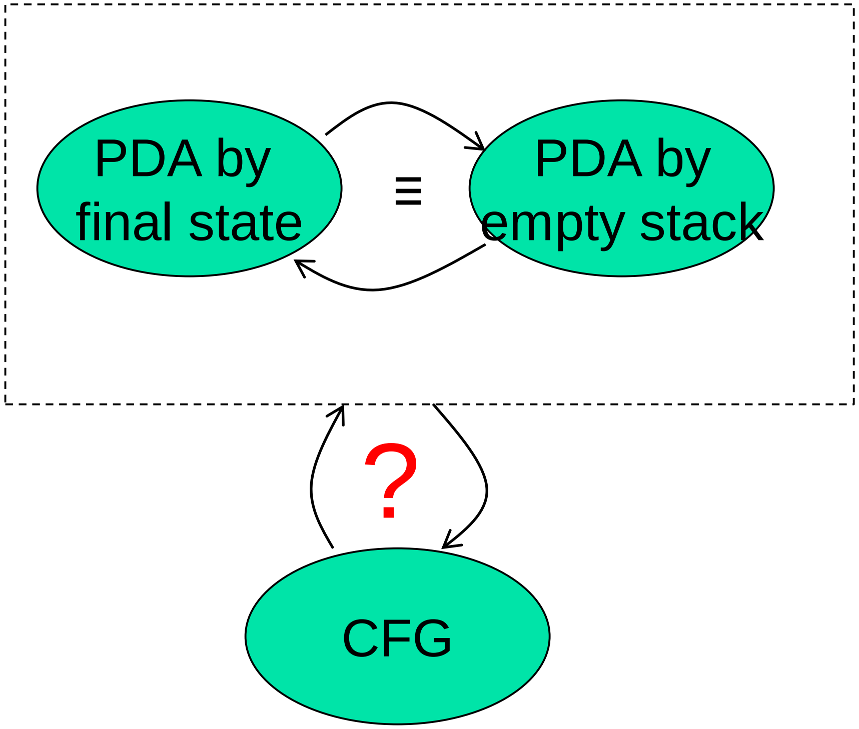
$P_N = (Q \cup \{p_0,p_e\}, \sum, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$

P_N:



Equivalence of PDAs and CFGs

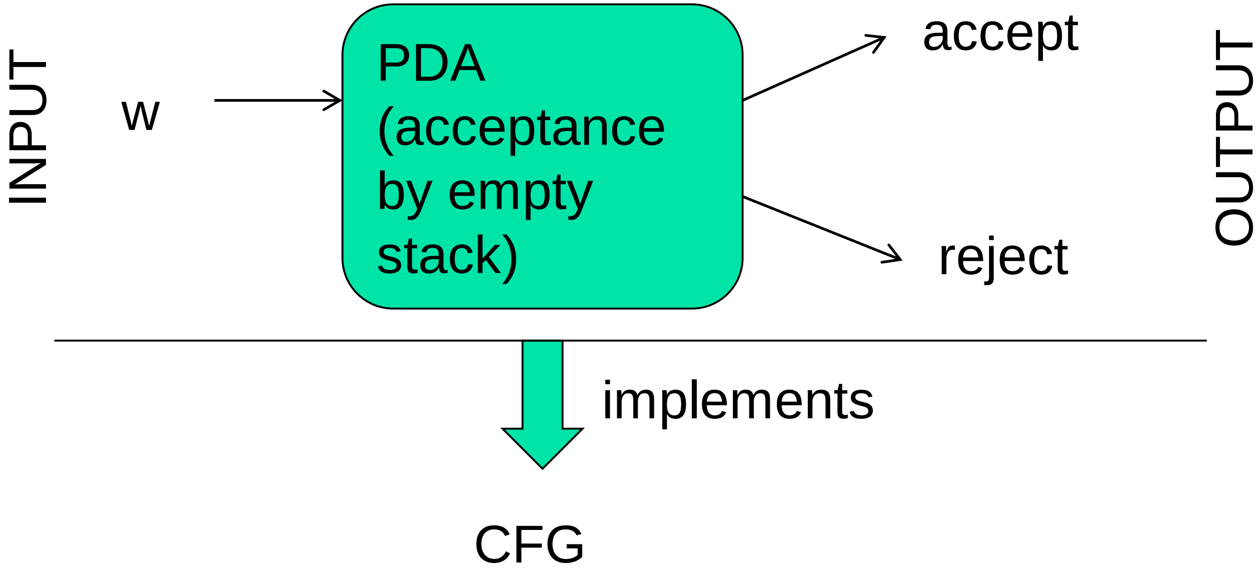
CFGs == PDAs ==> CFLs



This is same as: “implementing a CFG using a PDA”

Converting CFG to PDA

Main idea: The PDA simulates the leftmost derivation on a given w, and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.



This is same as: “implementing a CFG using a PDA”

Converting a CFG into a PDA

Main idea: The PDA simulates the leftmost derivation on a given w, and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.

Steps:

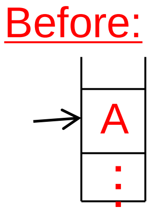
1. Push the right hand side of the production onto the stack, with leftmost symbol at the stack top
2. If stack top is the leftmost variable, then replace it by all its productions (each possible substitution will represent a *distinct* path taken by the non-deterministic PDA)
3. If stack top has a terminal symbol, and if it matches with the next symbol in the input string, then pop it

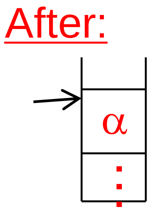
State is inconsequential (only one state is needed)

Formal construction of PDA from CFG

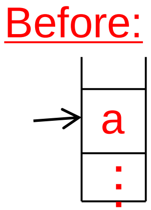
Note: Initial stack symbol (S) same as the start variable in the grammar

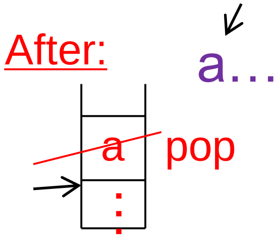
- Given: G= (V,T,P,S)
- Output: P_N = ({q}, T, V U T, δ, q, S)
- δ:





  - For all A ∈ V , add the following transition(s) in the PDA:
    - $\delta(q, \varepsilon, A) = \{ (q, \alpha) \mid \text{“}A \Longrightarrow \alpha\text{”} \in P\}$

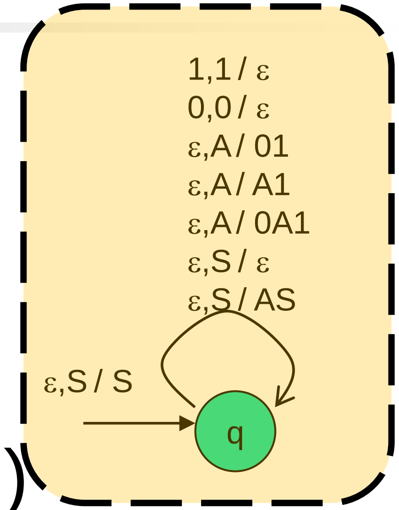




  - For all *a* ∈ T, add the following transition(s) in the PDA:
    - $\delta(q,a,a)= \{ (q, \varepsilon) \}$

Example: CFG to PDA

- G = ({S,A}, {0,1}, P, S)
- P:
  - S ==> AS | ε
  - A ==> 0A1 | A1 | 01
- PDA = ({q}, {0,1}, {0,1,A,S}, δ, q, S)
- δ:
  - δ(q, ε , S) = { (q, AS), (q, ε)}
  - δ(q, ε , A) = { (q,0A1), (q,A1), (q,01) }
  - δ(q, 0, 0) = { (q, ε) }
  - δ(q, 1, 1) = { (q, ε) }



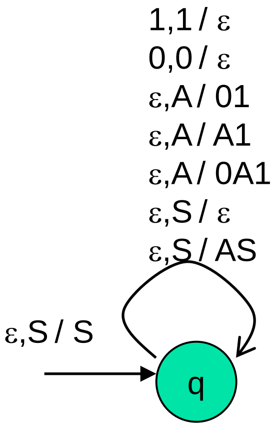
How will this new PDA work?

Lets simulate string <u>0011</u>

Simulating string 0011 on the new PDA …

PDA (δ):

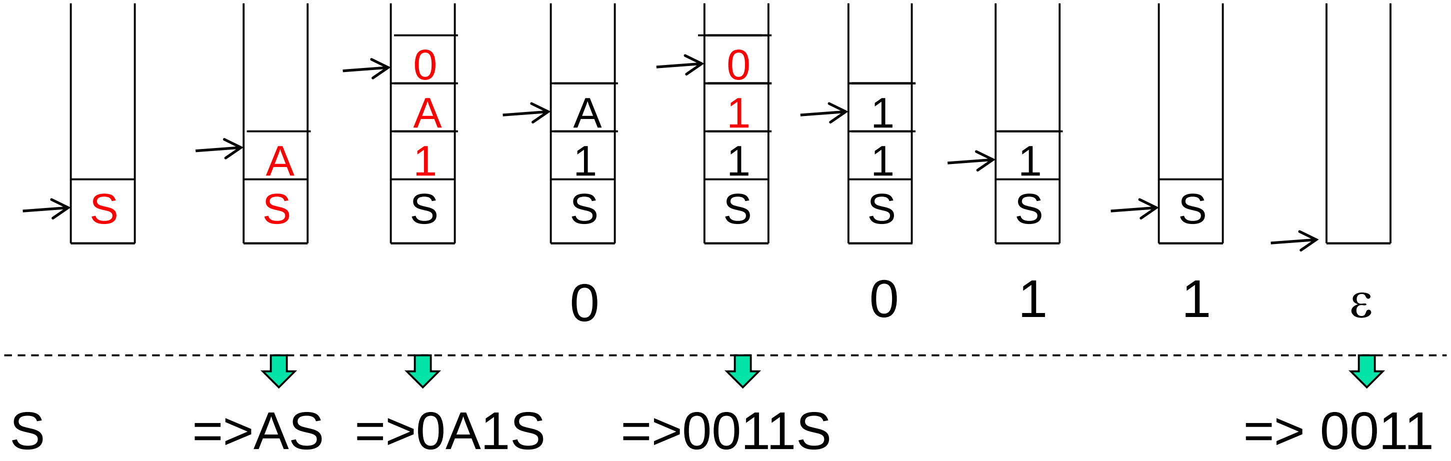
δ(q, ε , S) = { (q, AS), (q, ε)}
δ(q, ε , A) = { (q,0A1), (q,A1), (q,01) }
δ(q, 0, 0) = { (q, ε) }
δ(q, 1, 1) = { (q, ε) }



Leftmost deriv.:

S => AS
=> 0A1S
=> 0011S
=> 0011

Stack moves (shows only the successful path):



Accept by empty stack

Converting a PDA into a CFG

- Main idea: Reverse engineer the productions from transitions

If $\delta(q,a,Z) \Rightarrow (p, Y_1Y_2Y_3\ldots Y_k)$:

1. State is changed from q to p;
2. Terminal *a* is consumed;
3. Stack top symbol Z is popped and replaced with a sequence of k variables.

- Action: Create a grammar variable called "[qZp]" which includes the following production:
  - $[qZp] \Rightarrow a[pY_1q_1]\ [q_1Y_2q_2]\ [q_2Y_3q_3]\ldots\ [q_{k-1}Y_kq_k]$

- Proof discussion (in the book)

Example: Bracket matching

- To avoid confusion, we will use *b*="(" and *e*=")"

P_N: $(\{q_0\}, \{b,e\}, \{Z_0,Z_1\}, \delta, q_0, Z_0)$

1. $\delta(q_0,b,Z_0) = \{ (q_0,Z_1Z_0) \}$
2. $\delta(q_0,b,Z_1) = \{ (q_0,Z_1Z_1) \}$
3. $\delta(q_0,e,Z_1) = \{ (q_0, \varepsilon) \}$
4. $\delta(q_0, \varepsilon, Z_0) = \{ (q_0, \varepsilon) \}$

0. $S \Rightarrow [q_0Z_0q_0]$
1. $[q_0Z_0q_0] \Rightarrow b\ [q_0Z_1q_0]\ [q_0Z_0q_0]$
2. $[q_0Z_1q_0] \Rightarrow b\ [q_0Z_1q_0]\ [q_0Z_1q_0]$
3. $[q_0Z_1q_0] \Rightarrow e$
4. $[q_0Z_0q_0] \Rightarrow \varepsilon$

Let $A=[q_0Z_0q_0]$
Let $B=[q_0Z_1q_0]$

0. $S \Rightarrow A$
1. $A \Rightarrow b\ B\ A$
2. $B \Rightarrow b\ B\ B$
3. $B \Rightarrow e$
4. $A \Rightarrow \varepsilon$

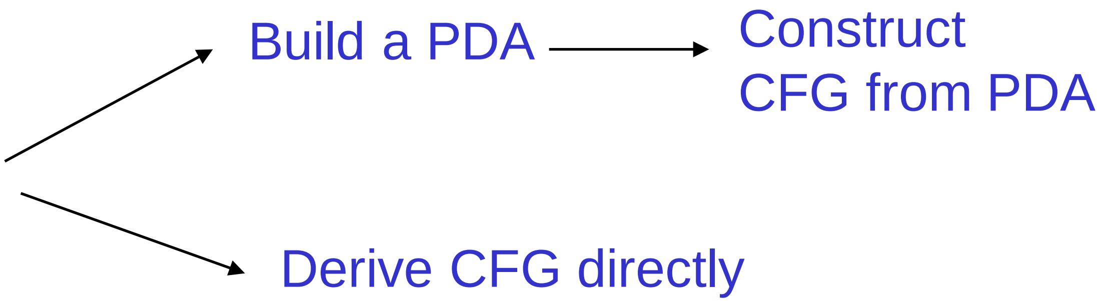
Simplifying,

0. $S \Rightarrow b\ B\ S \mid \varepsilon$
1. $B \Rightarrow b\ B\ B \mid e$

If you were to directly write a CFG:

$S \Rightarrow b\ S\ e\ S \mid \varepsilon$

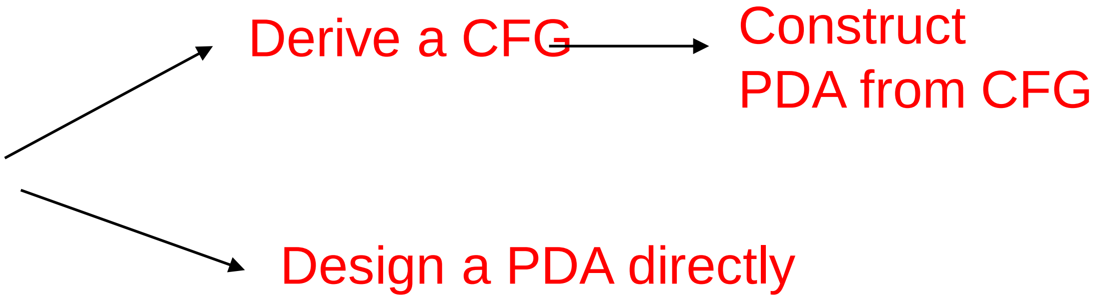
Two ways to build a CFG



Build a PDA → Construct CFG from PDA (indirect)

Derive CFG directly (direct)

Similarly… Two ways to build a PDA

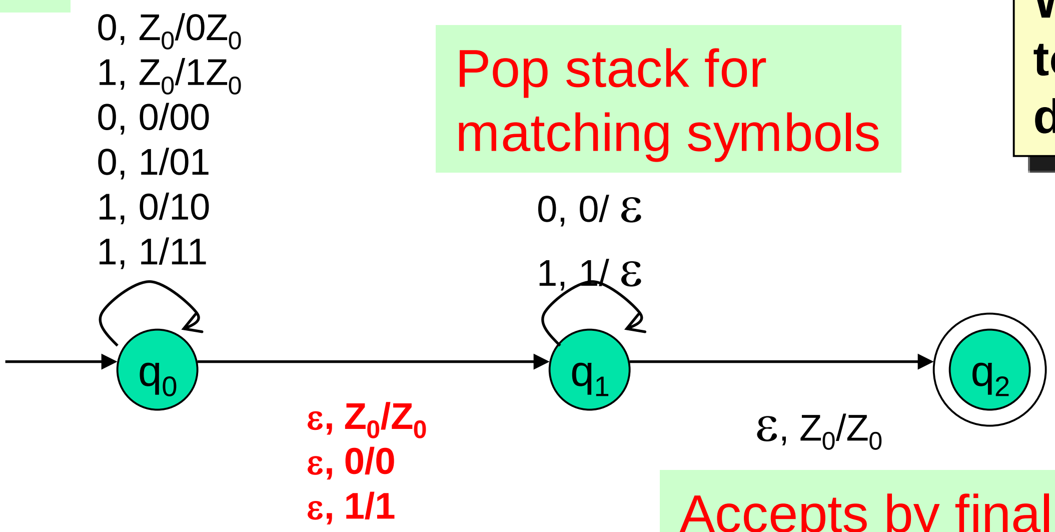


Derive a CFG → Construct PDA from CFG (indirect)

Design a PDA directly (direct)

Deterministic PDAs

This PDA for L_{wwr} is non-deterministic



Grow stack

Pop stack for matching symbols

Switch to popping mode

Accepts by final state

Why does it have to be non-deterministic?

To remove guessing, impose the user to insert c in the middle

Example shows that: Nondeterministic PDAs ≠ D-PDAs

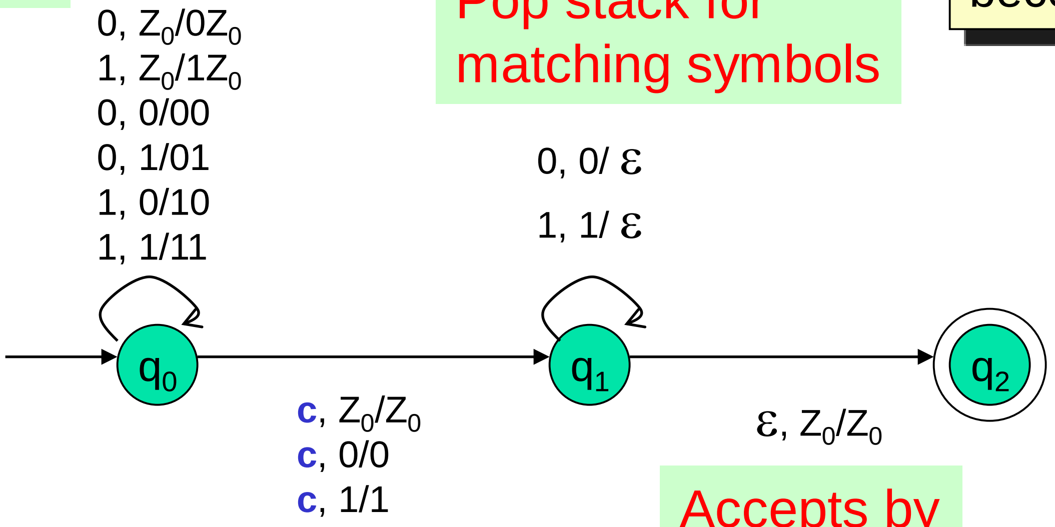
D-PDA for $L_{wcwr} = \{wcw^R \mid c$ is some special symbol not in $w\}$

Note:

- all transitions have become deterministic

Grow stack

Pop stack for matching symbols



Switch to popping mode

Accepts by final state

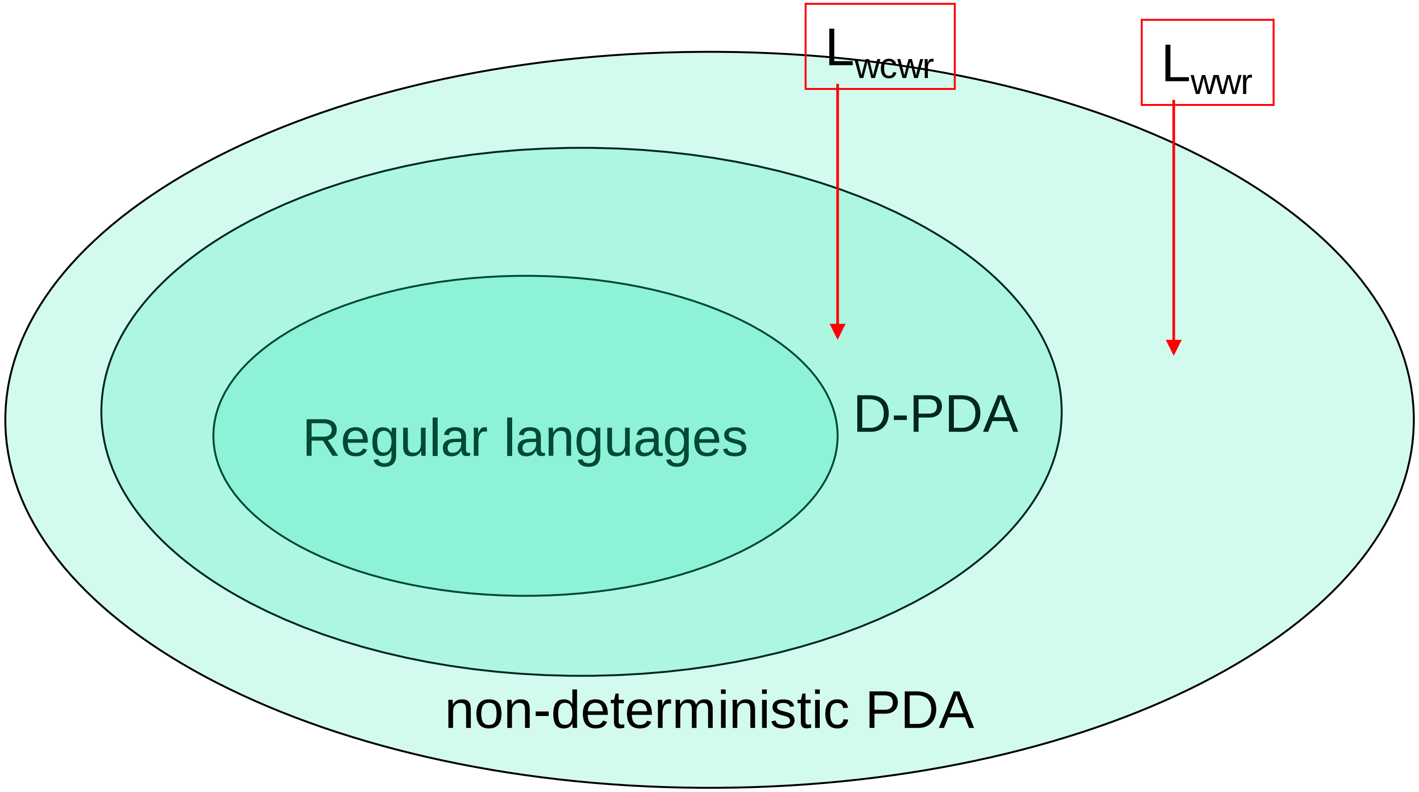
Deterministic PDA: Definition

- A PDA is *deterministic* if and only if:

1. δ(q,a,X) has *at most one* member for any $a \in \sum \cup \{\varepsilon\}$

→ If δ(q,a,X) is non-empty for some $a \in \sum$, then $\delta(q, \varepsilon, X)$ must be empty.

PDA vs DPDA vs Regular languages



Summary

- PDAs for CFLs and CFGs
  - Non-deterministic
  - Deterministic
- PDA acceptance types
  1. By final state
  2. By empty stack
- PDA
  - IDs, Transition diagram
- Equivalence of CFG and PDA
  - CFG => PDA construction
  - PDA => CFG construction

Turing Machines

Turing Machines are…

- Very powerful (abstract) machines that could simulate any modern day computer (although very, very slowly!)

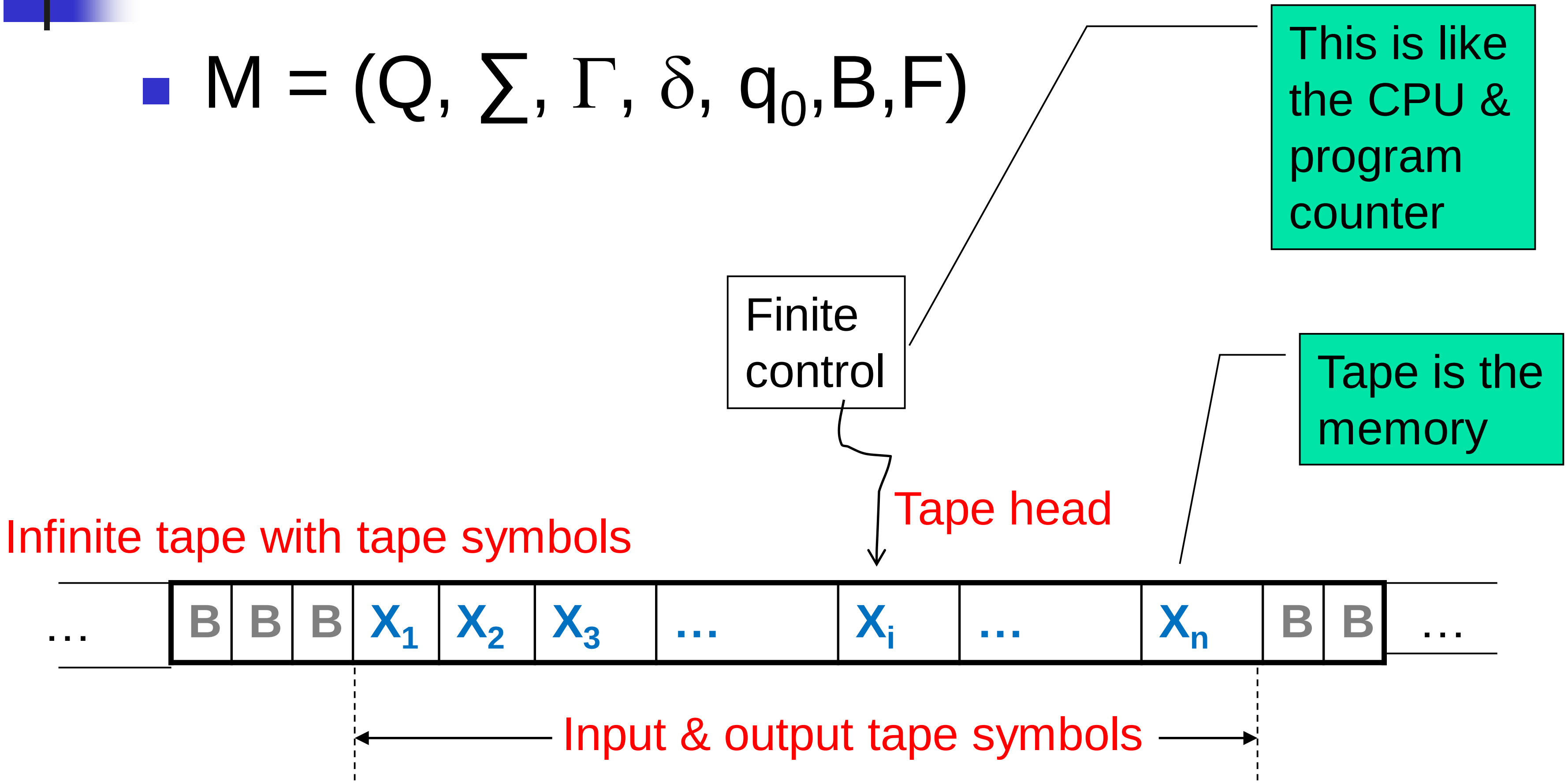
- Why design such a machine?
  - If a problem cannot be "solved" even using a TM, then it implies that the problem is ***undecidable***

For every input, answer YES or NO

- Computability vs. Decidability

A Turing Machine (TM)

- M = (Q, ∑, Γ, δ, q_0,B,F)



This is like the CPU & program counter

Finite control

Tape is the memory

Tape head

Infinite tape with tape symbols

… B B B X_1 X_2 X_3 … X_i … X_n B B …

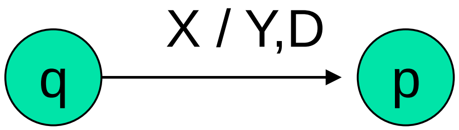
Input & output tape symbols

B: blank symbol (special symbol reserved to indicate data boundary)

You can also use:
➔ for R
🡐 for L

Transition function

- One move (denoted by |---) in a TM does the following:
  - ***δ(q,X) = (p,Y,D)***
    - q is the current state
    - X is the current tape symbol pointed by tape head
    - State changes from q to p
    - After the move:
      - X is replaced with symbol Y
      - If D="L", the tape head moves "left" by one position.
        Alternatively, if D="R" the tape head moves "right" by one position.



ID of a TM

- Instantaneous Description or ID :
  - $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$

    means:
    - q is the current state
    - Tape head is pointing to X_i
    - $X_1X_2\ldots X_{i-1}X_iX_{i+1}\ldots X_n$ are the current tape symbols

- $\delta(q,X_i) = (p,Y,R)$ is same as:

  $X_1\ldots X_{i-1}qX_i\ldots X_n$ |---- $X_1\ldots X_{i-1}YpX_{i+1}\ldots X_n$
- $\delta(q,X_i) = (p,Y,L)$ is same as:

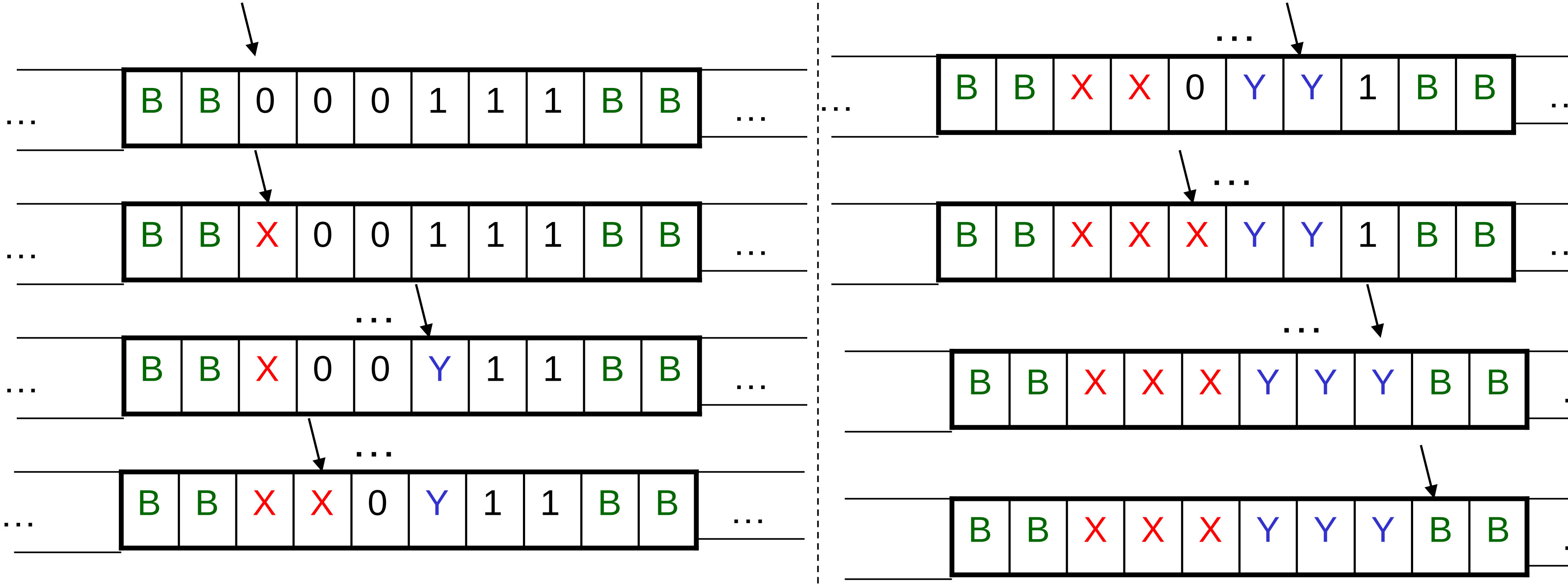
  $X_1\ldots X_{i-1}qX_i\ldots X_n$ |---- $X_1\ldots pX_{i-1}YX_{i+1}\ldots X_n$

Way to check for Membership

- Is a string *w* accepted by a TM?
- Initial condition:
  - The (whole) input string *w* is present in TM, preceded and followed by infinite blank symbols
- Final acceptance:
  - Accept *w* if TM enters final state and halts
  - If TM halts and not final state, then reject

Example: L = $\{0^n1^n \mid n \geq 1\}$

- Strategy: w = 000111



… B B 0 0 0 1 1 1 B B …

… B B X 0 0 1 1 1 B B …

… B B X 0 0 Y 1 1 B B …

… B B X X 0 Y 1 1 B B …

… B B X X 0 Y Y 1 B B …

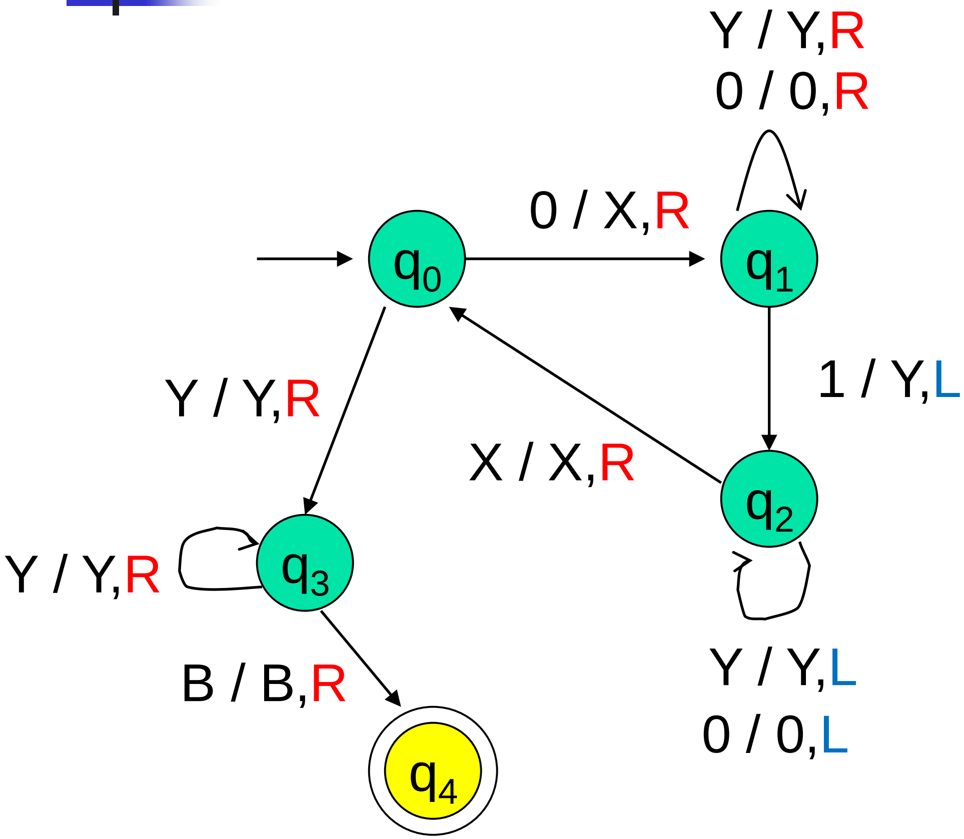
B B X X X Y Y 1 B B

B B X X X Y Y Y B B

B B X X X Y Y Y B B

Accept

TM for $\{0^n1^n \mid n \geq 1\}$



1. Mark next unread 0 with X and move right
2. Move to the right all the way to the first unread 1, and mark it with Y
3. Move back (to the left) all the way to the last marked X, and then move one position to the right
4. If the next position is 0, then goto step 1.
   Else move all the way to the right to ensure there are no excess 1s. If not move right to the next blank symbol and stop & accept.

*state diagram representation preferred

TM for $\{0^n1^n \mid n \geq 1\}$

	Next Tape Symbol				
Curr. State	0	1	X	Y	B
→ q_0	(q_1,X,R)	-	-	(q_3,Y,R)	-
q_1	$(q_1,0,R)$	(q_2,Y,L)	-	(q_1,Y,R)	-
q_2	$(q_2,0,L)$	-	(q_0,X,R)	(q_2,Y,L)	-
q_3	-	-	-	(q_3,Y,R)	(q_4,B,R)
$*q_4$	-	--	-	-	-

Table representation of the state diagram

TMs for calculations

- TMs can also be used for calculating values
  - Like arithmetic computations
  - Eg., addition, subtraction, multiplication, etc.

Example 2: monus subtraction

"m -- n" = max{m-n,0}

0^m10^n ➔ ...B 0^{m-n} B.. (*if m>n*)

...BB…B.. (*otherwise*)

1. For every 0 on the left (mark X), mark off a 0 on the right (mark Y)
2. Repeat process, until one of the following happens:
   1. // No more 0s remaining on the left of 1
      Answer is 0, so flip all excess 0s on the right of 1 to Bs (and the 1 itself) and halt
   2. //No more 0s remaining on the right of 1
      Answer is m-n, so simply halt after making 1 to B

Give state diagram

Example 3: Multiplication

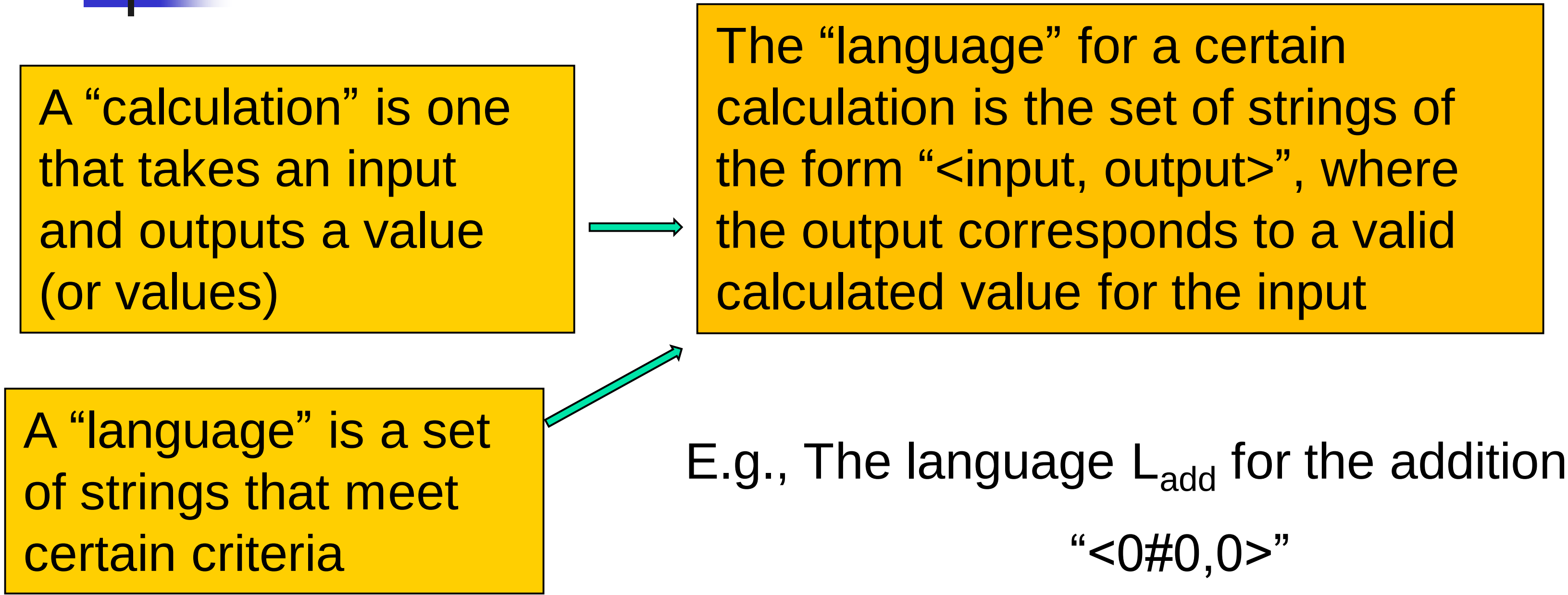
- 0^m10^n1 (input), $0^{mn}1$ (output)

- Pseudocode:

1. Move tape head back & forth such that for every 0 seen in 0^m, write n 0s to the right of the last delimiting 1
2. Once written, that zero is changed to B to get marked as finished
3. After completing on all m 0s, make the remaining n 0s and 1s also as Bs

Give state diagram

Calculations vs. Languages



A "calculation" is one that takes an input and outputs a value (or values)

A "language" is a set of strings that meet certain criteria

The "language" for a certain calculation is the set of strings of the form "<input, output>", where the output corresponds to a valid calculated value for the input

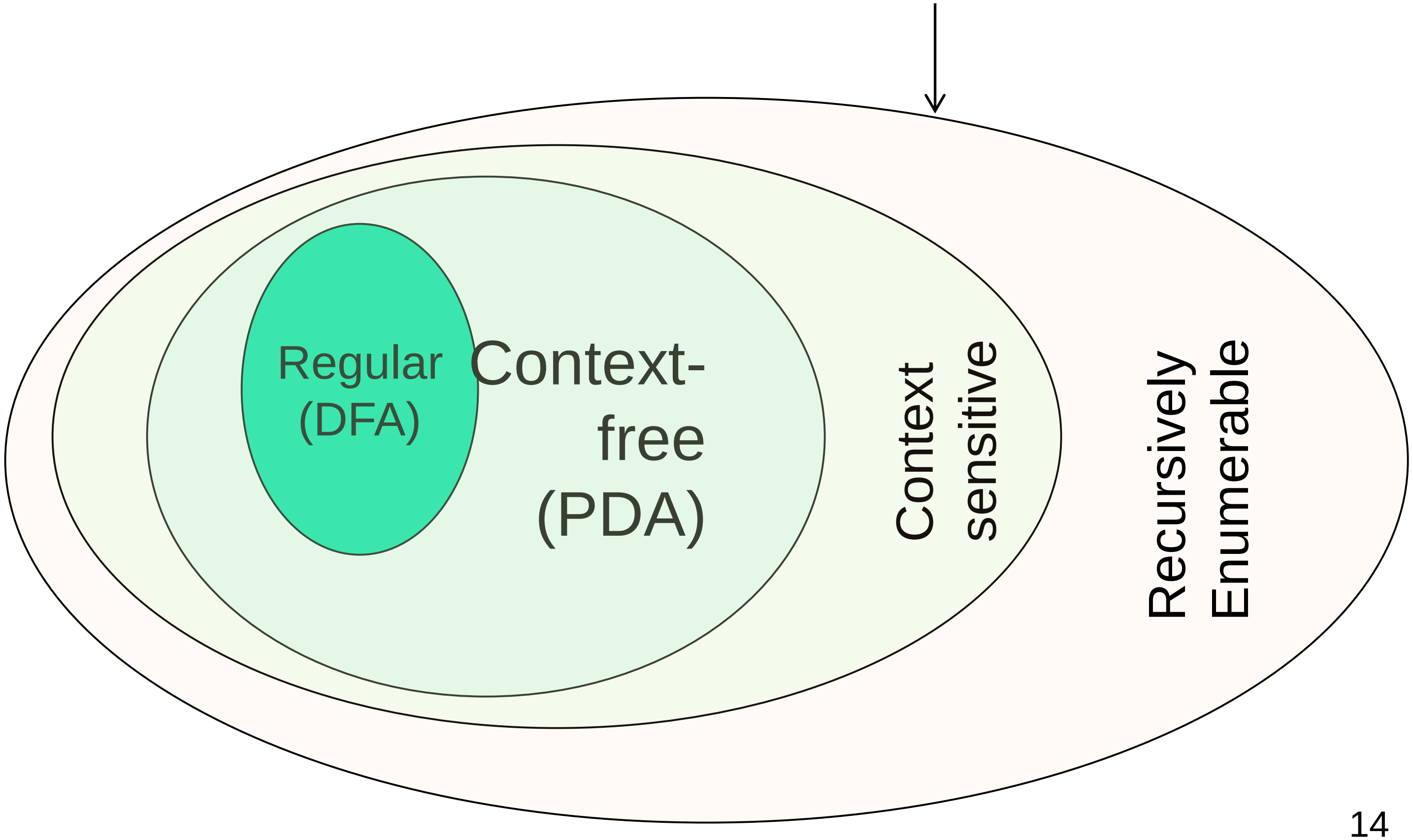
E.g., The language L_{add} for the addition operation

"<0#0,0>"
"<0#1,1>"
…
"<2#4,6>"
…

Membership question == verifying a solution
e.g., is "<15#12,27>" a member of L_{add} ?

Language of the Turing Machines

- *Recursive Enumerable (RE) language*

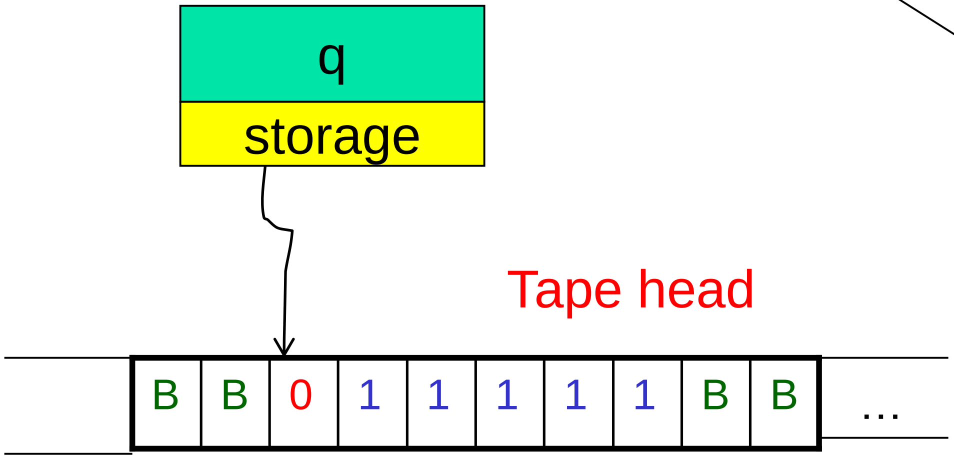


Variations of Turing Machines

Generic description
Will work for both a=0 and a=1

TMs with *storage*

- E.g., TM for 01* + 10*



Current state

Current Storage symbol

Tape symbol

Next state

New Storage symbol

Transition function δ:

- $\delta([q_0,B],a) = ([q_1,a], a, R)$
- $\delta([q_1,a],\overline{a}) = ([q_1,a], \overline{a}, R)$
- $\delta([q_1,a],B) = ([q_2,B], B, R)$

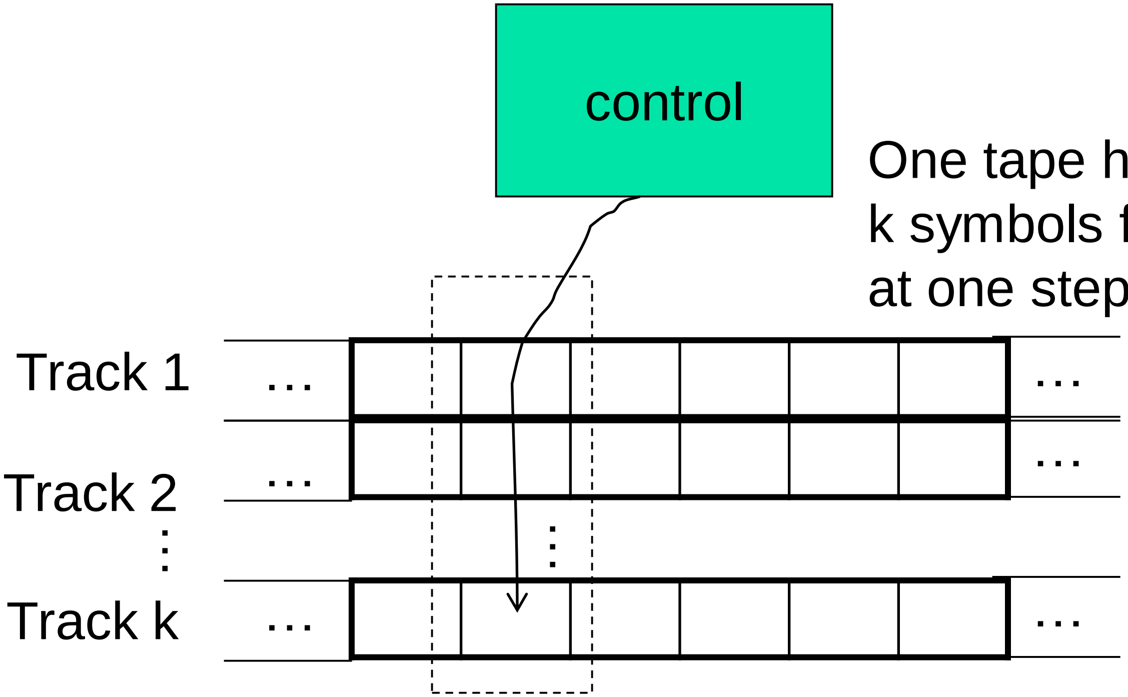
[q,a]: where q is current state,
a is the symbol in storage

Are the standard TMs equivalent to TMs with storage?

Yes

Multi-track Turing Machines

- TM with multiple tracks,
  but just one unified tape head

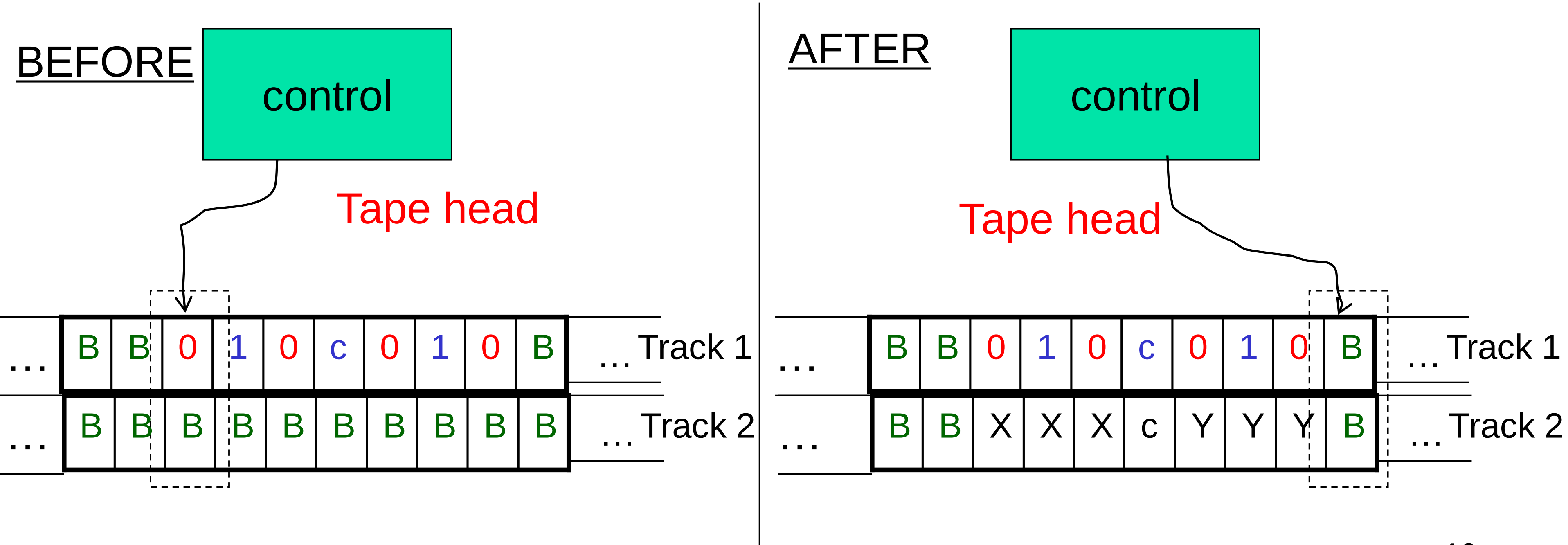


One tape head to read
k symbols from the k tracks
at one step.

Multi-Track TMs

- TM with multiple “tracks” but just one head

E.g., TM for {wcw | w∈ {0,1}* }
but w/o modifying original input string



BEFORE

… B B 0 1 0 c 0 1 0 B … Track 1

… B B B B B B B B B B … Track 2

AFTER

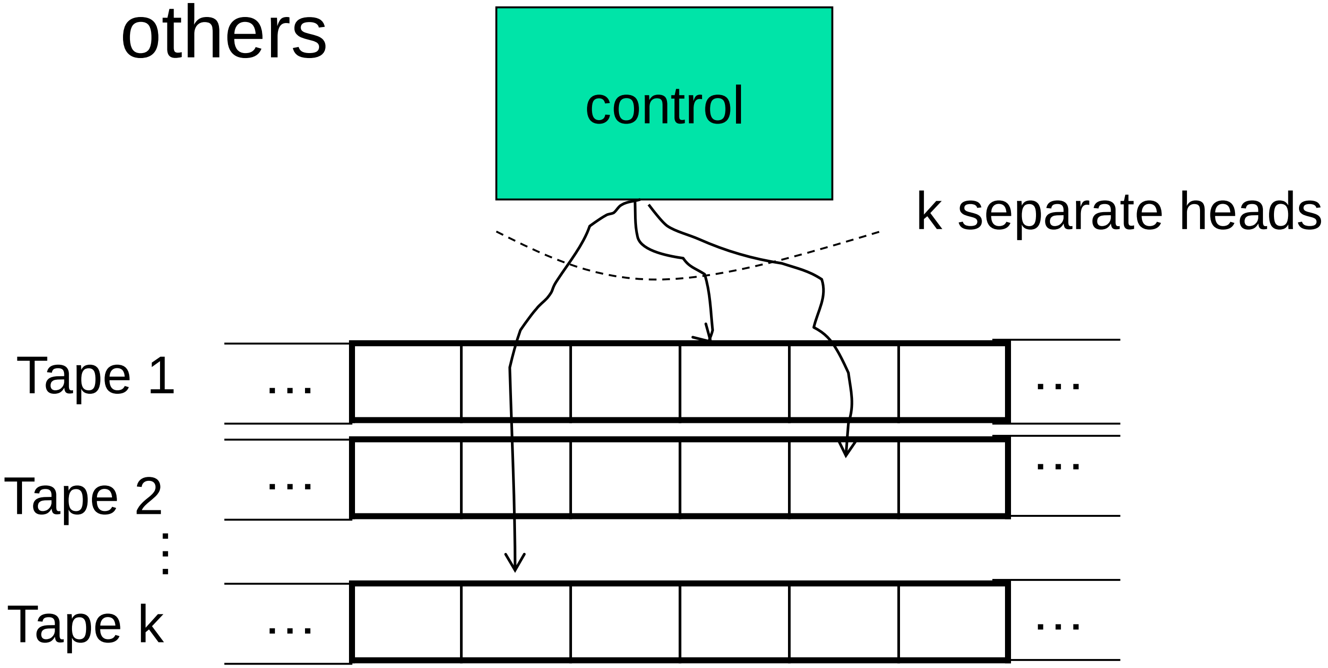
… B B 0 1 0 c 0 1 0 B … Track 1

… B B X X X c Y Y Y B … Track 2

Second track mainly used as a scratch space for marking

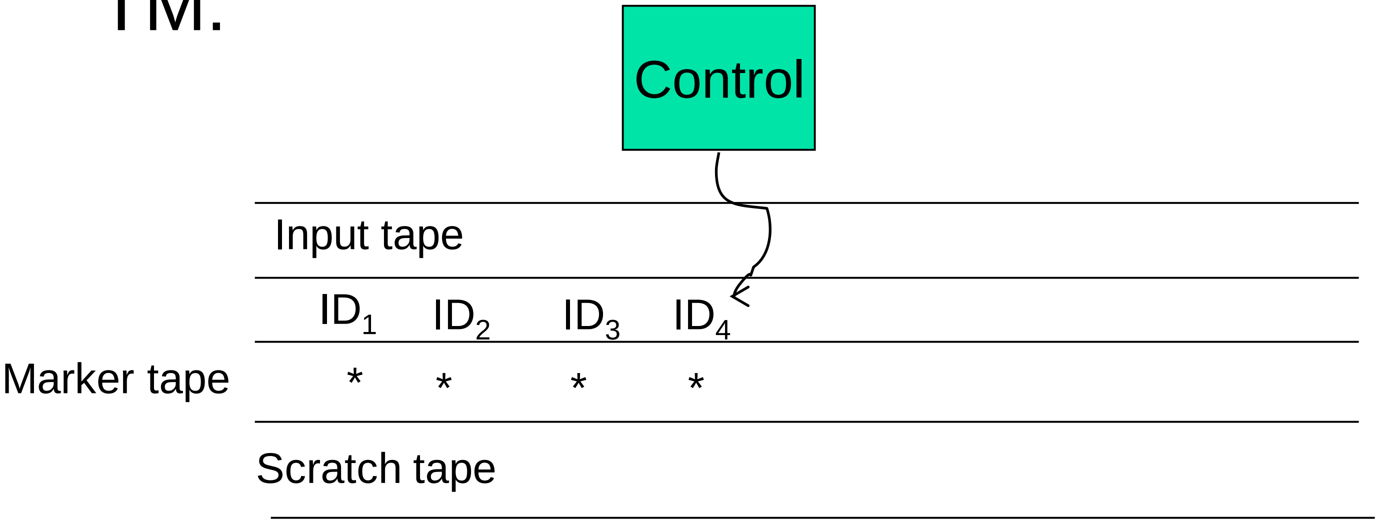
Multi-tape Turing Machines

- TM with multiple tapes, *each tape with a separate head*
  - Each head can move independently of the others



Non-deterministic TMs

- A TM can have non-deterministic moves:
  - $\delta(q,X) = \{ (q_1,Y_1,D_1), (q_2,Y_2,D_2), \dots \}$
- Simulation using a multitape deterministic TM:



Summary

- TMs == Recursively Enumerable languages
- TMs can be used as both:
  - Language recognizers
  - Calculators/computers
- ***Basic TM is <u>equivalent</u> to all the below:***
  1. *TM + storage*
  2. *Multi-track TM*
  3. *Multi-tape TM*
  4. *Non-deterministic TM*
- TMs are like universal computing machines with unbounded storage

Undecidability

Decidability vs. Undecidability

- There are two types of TMs (based on halting):

  (*Recursive*)

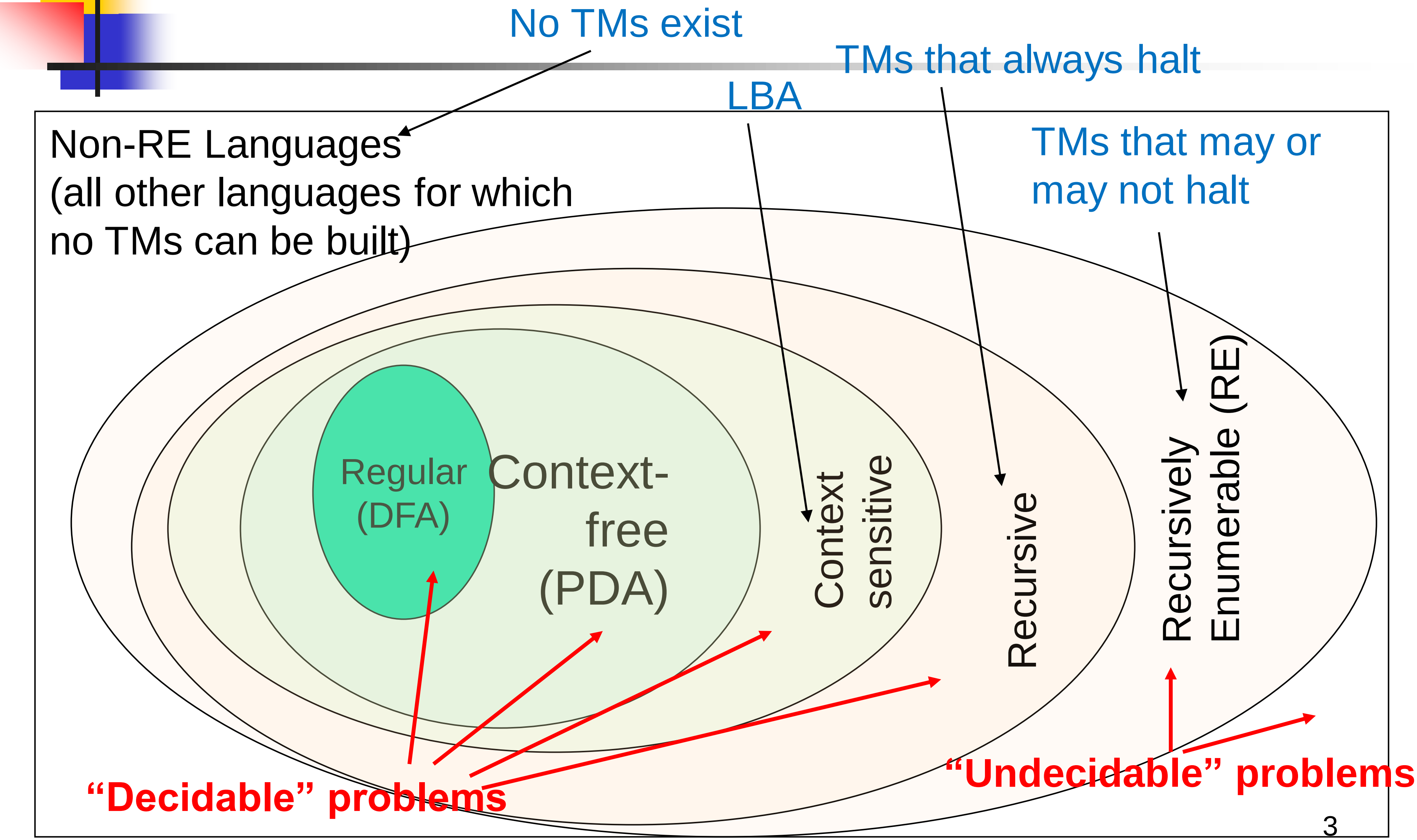
  TMs that *always* halt, no matter accepting or non-accepting ≡ **DECIDABLE** PROBLEMS

  (*Recursively enumerable*)

  TMs that *are guaranteed to halt* only on acceptance. If non-accepting, it may or may not halt (i.e., could loop forever).

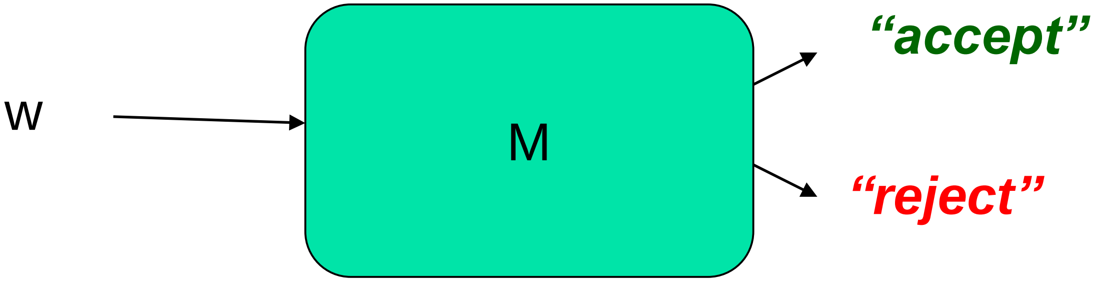
- **<u>Undecidability:</u>**
  - Undecidable problems are those that are <u>not</u> recursive

Recursive, RE, Undecidable languages

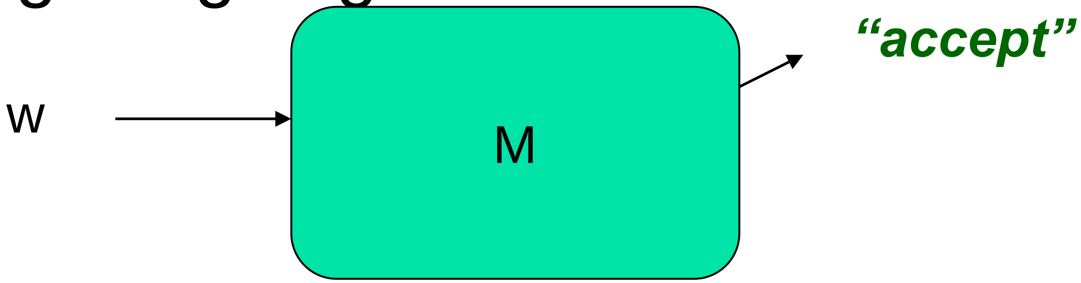


Recursive Languages & Recursively Enumerable (RE) languages

- Any TM for a Recursive language is going to look like this:



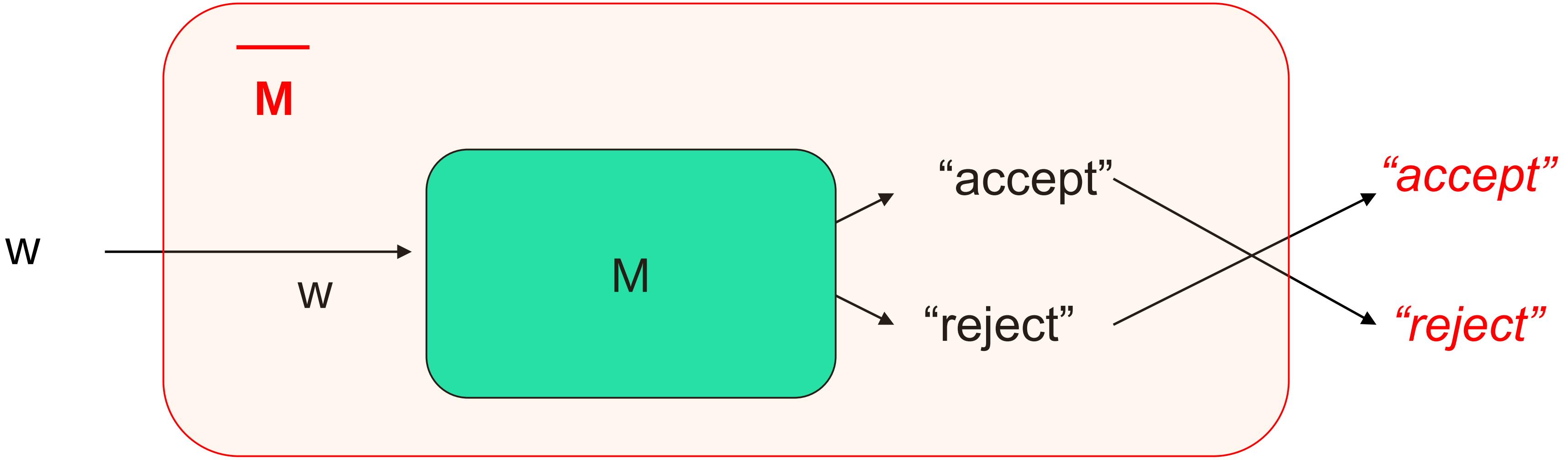
- Any TM for a Recursively Enumerable (RE) language is going to look like this:



Closure Properties of:
- the Recursive language class, and
- the Recursively Enumerable language class

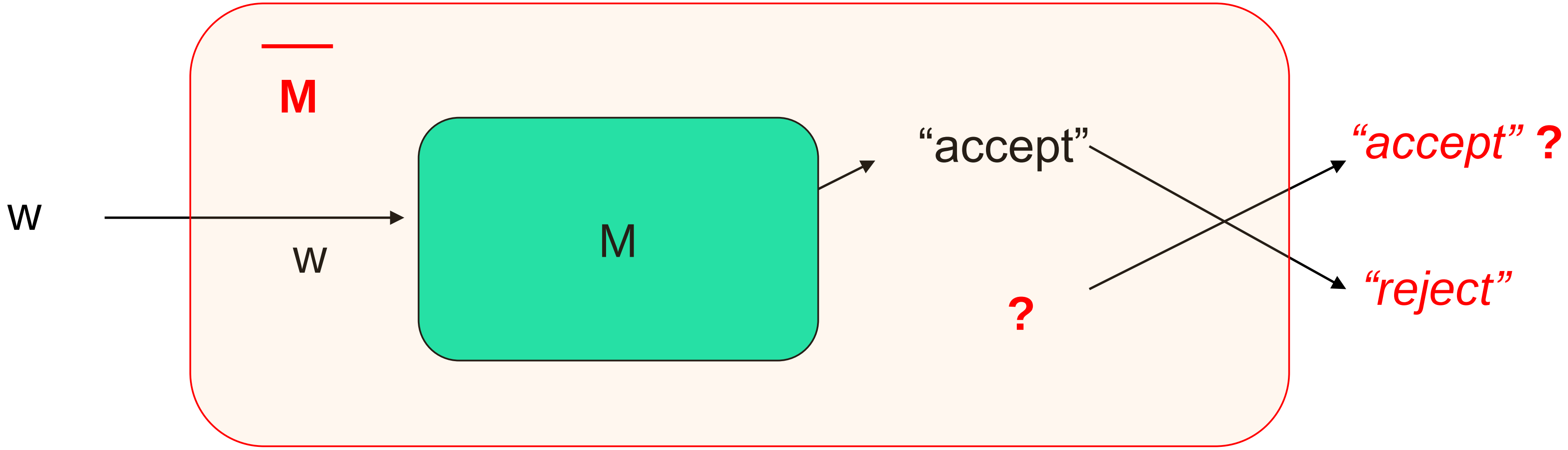
Recursive Languages are closed under complementation

- If L is Recursive, $\overline{L}$ is also Recursive



Are Recursively Enumerable Languages closed under complementation? (NO)

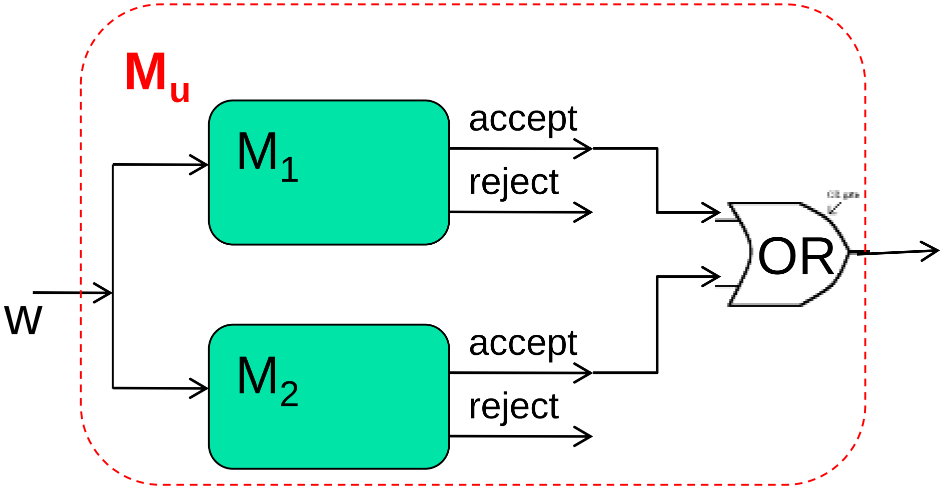
- If L is RE, $\overline{L}$ need not be RE



Recursive Langs are closed under Union

- Let M_u = TM for L_1 U L_2
- M_u construction:

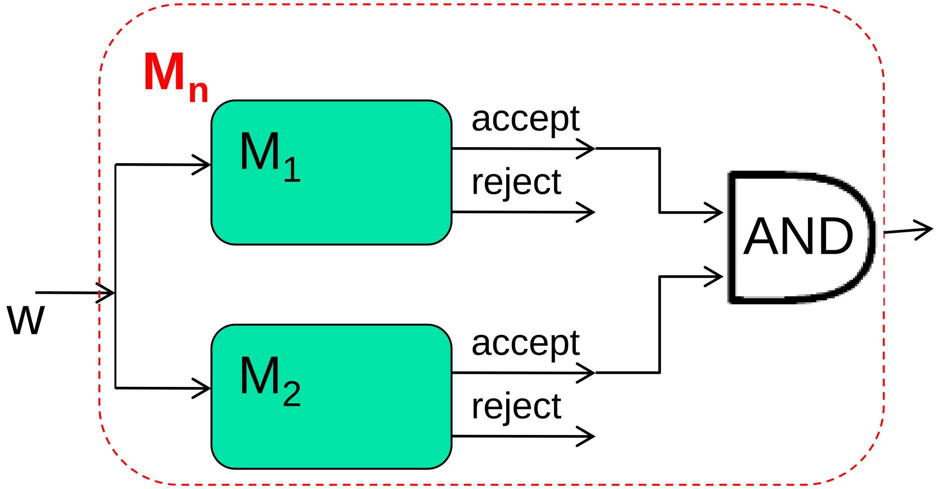
1. Make 2-tapes and copy input w on both tapes
2. Simulate M_1 on tape 1
3. Simulate M_2 on tape 2
4. If either M_1 or M_2 accepts, then M_u accepts
5. Otherwise, M_u rejects.



Recursive Langs are closed under Intersection

- Let M_n = TM for $L_1 \cap L_2$
- M_n construction:

1. Make 2-tapes and copy input w on both tapes
2. Simulate M_1 on tape 1
3. Simulate M_2 on tape 2
4. If M_1 AND M_2 accepts, then M_n accepts
5. Otherwise, M_n rejects.



Other Closure Property Results

- Recursive languages are also closed under:
  - Concatenation
  - Kleene closure (star operator)
  - Homomorphism, and inverse homomorphism
- RE languages are closed under:
  - Union, intersection, concatenation, Kleene closure

- RE languages are *not* closed under:
  - complementation

“Languages” vs. “Problems”

A “language” is a set of strings

Any “problem” can be expressed as a set of all strings that are of the form:

- “<input, output>”

e.g., Problem (a+b) ≡ Language of strings of the form { “a#b, a+b” }

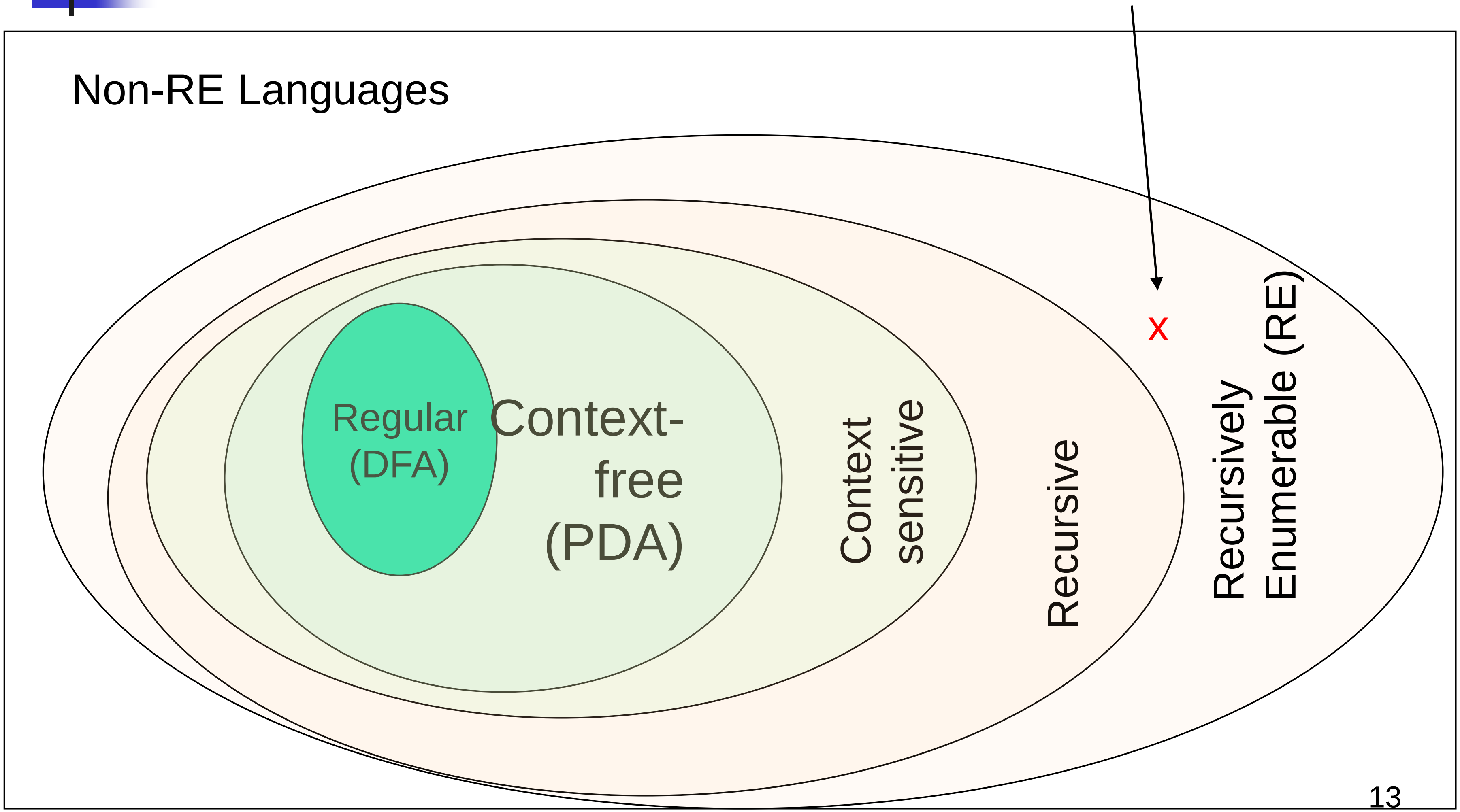
==> Every problem also corresponds to a language!!

Think of the language for a “problem” == a *verifier* for the problem

The Halting Problem

An example of a <u>recursive enumerable</u> problem that is also <u>undecidable</u>

The Halting Problem



Non-RE Languages

Regular (DFA)

Context-free (PDA)

Context sensitive

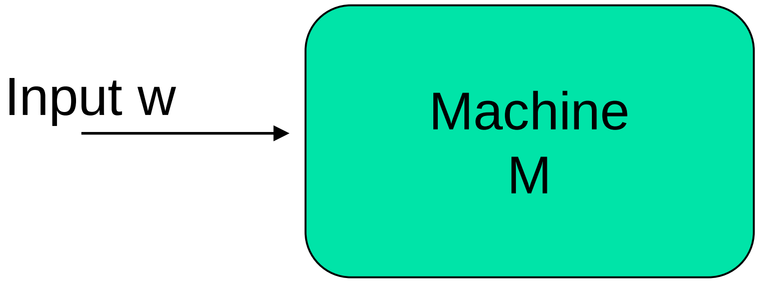
Recursive

Recursively Enumerable (RE)

What is the Halting Problem?

Definition of the “halting problem”:

- *Does a givenTuring Machine M halt on a given input w?*



A Turing Machine simulator

The Universal Turing Machine

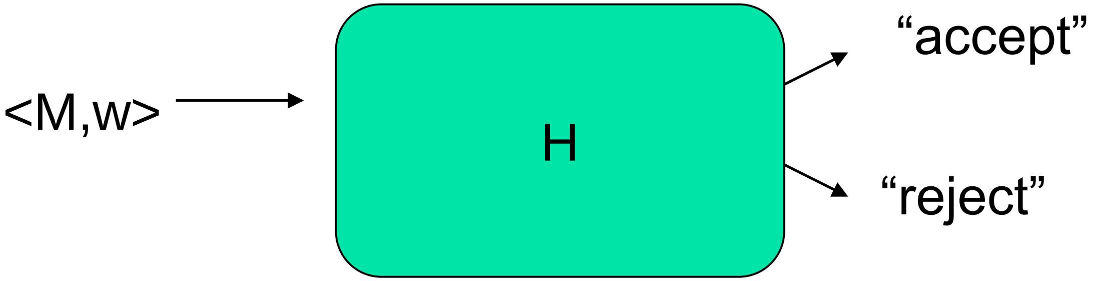
- Given: TM **M** & its input **w**
- Aim: Build another TM called “H”, that will output:
  - “*accept*” if M accepts w, and
  - “reject” otherwise
- An algorithm for H:
  - Simulate M on w
  - $$H(<M,w>) = \begin{cases} \textit{accept}, & \text{if } M \text{ accepts w} \\ \textit{reject}, & \text{if M does does not accept w} \end{cases}$$

Implies: H is in RE

Question: If M does *not* halt on w, what will happen to H?

A Claim

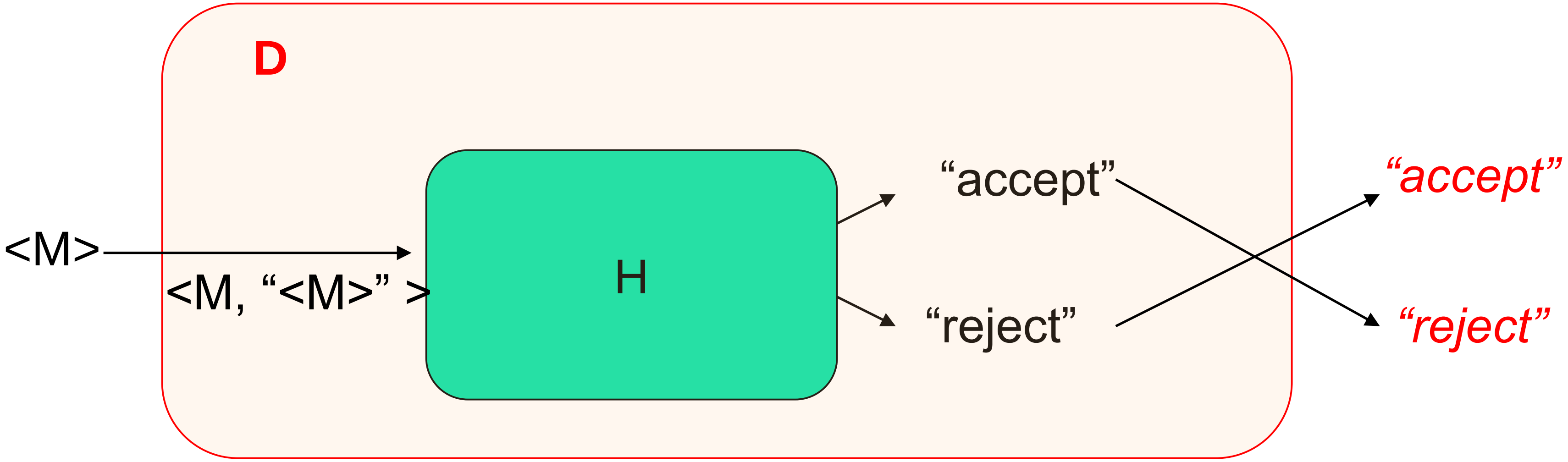
- Claim: No H that is always guaranteed to halt, can exist!
- Proof: (Alan Turing, 1936)
  - By contradiction, let us assume H exists



Therefore, if H exists → D also should exist.
But can such a D exist? (if not, then H also cannot exist)

HP Proof (step 1)

- Let us construct a new TM **D** using H as a subroutine:
  - On input <M>:
    1. Run H on input <M, <M> >; //(i.e., run M on M itself)
    2. Output the *opposite* of what H outputs;



HP Proof (step 2)

- The notion of inputing "<M>" to M itself
  - A program can be input to itself (e.g., a compiler is a program that takes any program as input)

$$D(\langle M \rangle) = \begin{cases} \textit{accept}, & \text{if M does } \textit{not} \text{ accept } \langle M \rangle \\ \textit{reject}, & \text{if M accepts } \langle M \rangle \end{cases}$$

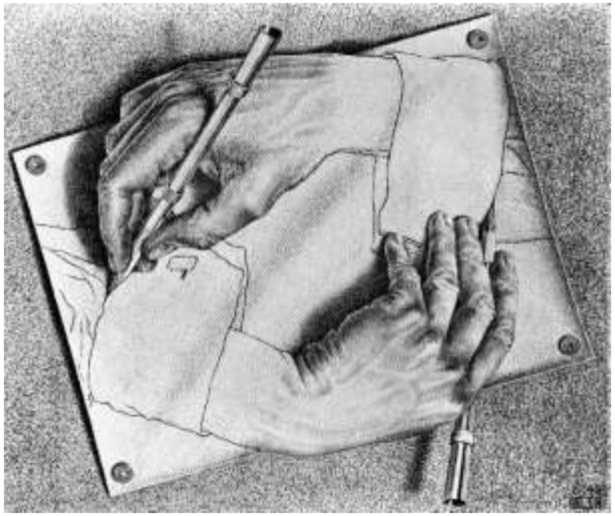
Now, what happens if D is input to itself?

$$D(\langle D \rangle) = \begin{cases} \textit{accept}, & \text{if D does not accept } \langle D \rangle \\ \textit{reject}, & \text{if D accepts } \langle D \rangle \end{cases}$$

A contradiction!!! ==> Neither D nor H can exist.

Of Paradoxes & Strange Loops

E.g., Barber's paradox, Achilles & the Tortoise (Zeno's paradox)
MC Escher's paintings



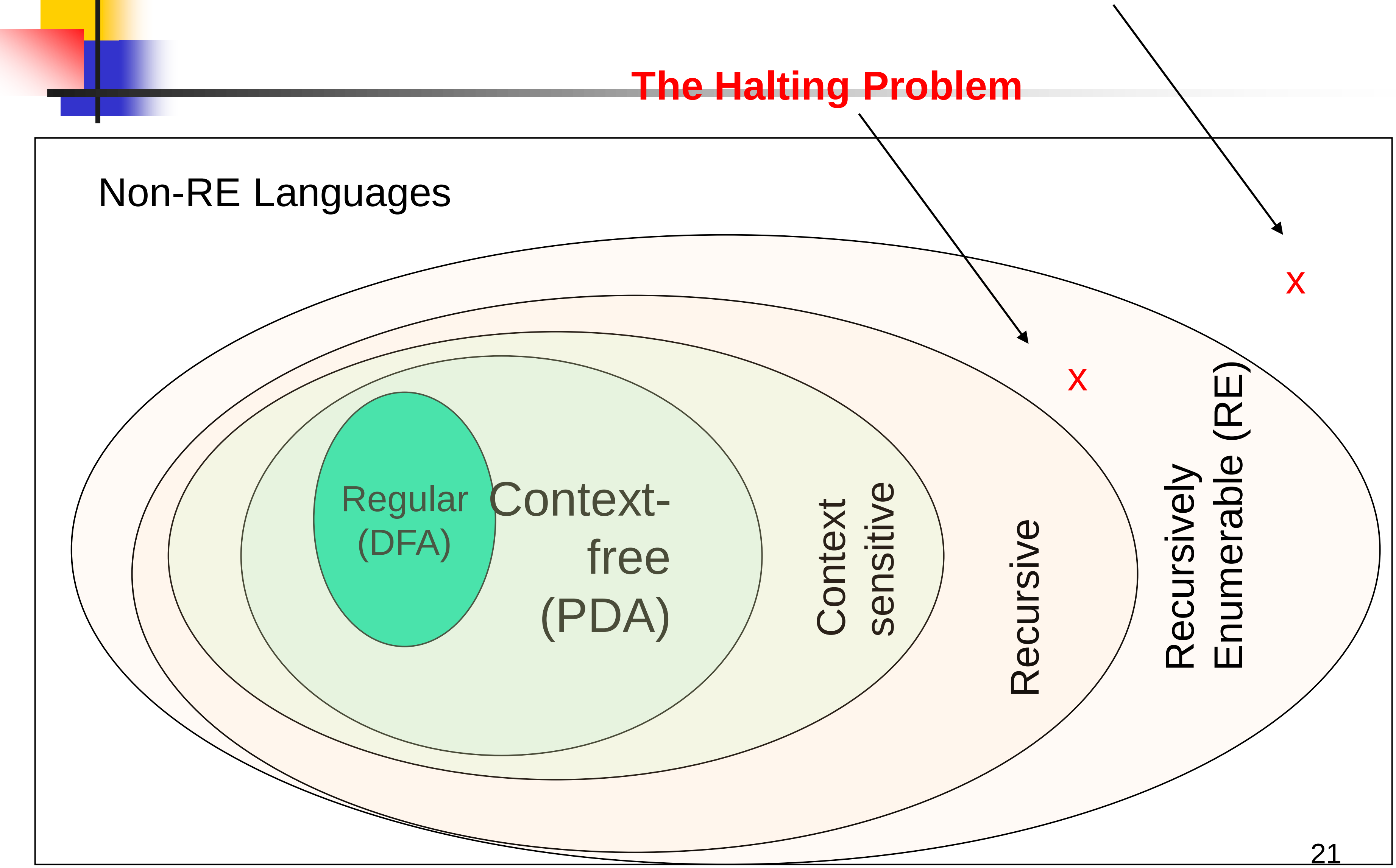
A fun book for further reading:

"Godel, Escher, Bach: An Eternal Golden Braid"
by Douglas Hofstadter (Pulitzer winner, 1980)

The Diagonalization Language

Example of a language that is
<u>not recursive enumerable</u>

(i.e, no TMs exist)



The Diagonalization language

The Halting Problem

A Language about TMs & acceptance

- Let L be the language of all strings <M,w> s.t.:
  1. M is a TM (coded in binary) with input alphabet also binary
  2. w is a binary string
  3. M accepts input w.

Enumerating all binary strings

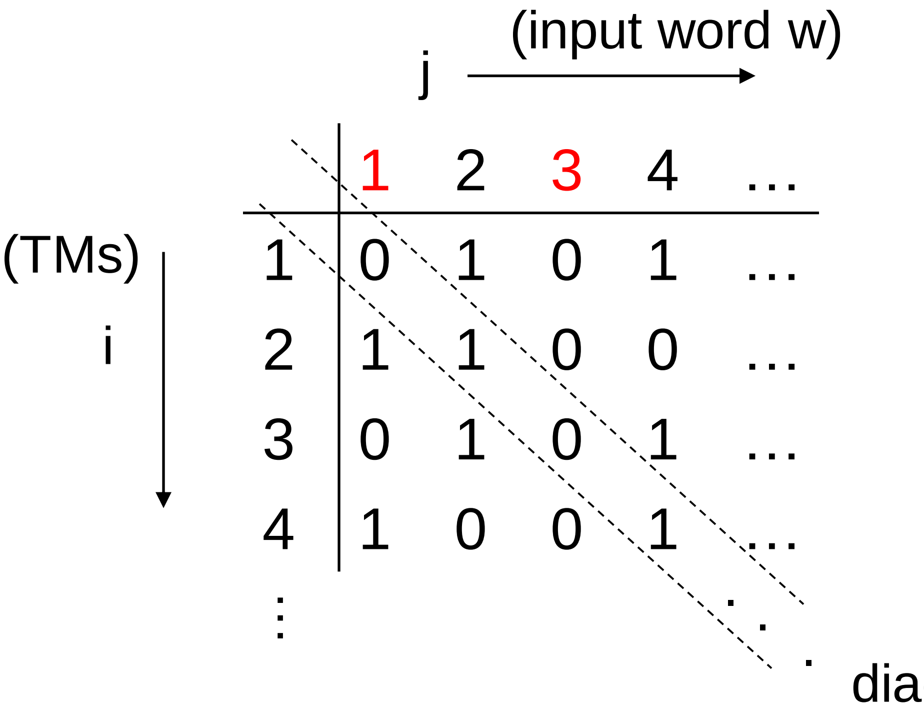
- Let w be a binary string
- Then 1w ≡ i, where i is some integer
  - E.g., If w=ε, then i=1;
  - If w=0, then i=2;
  - If w=1, then i=3; so on…
- If 1w≡ i, then call w as the ith word or ith binary string, denoted by w_i.
- ***==> A canonical ordering of all binary strings:***
  - ***{ε, 0, 1, 00, 01, 10, 11, 000, 100, 101, 110, …..}***
  - $\boldsymbol{\{w_1, w_2, w_3, w_4, \ldots. w_i, \ldots\}}$

Any TM M can also be binary-coded

- $M = \{ Q, \{0,1\}, \Gamma, \delta, q_0, B, F \}$
  - Map all states, tape symbols and transitions to integers (==>binary strings)
  - $\delta(q_i,X_j) = (q_k,X_l,D_m)$ will be represented as:
    - ==> $0^i 1\ 0^j 1\ 0^k 1\ 0^l 1\ 0^m$
- <u>Result: </u>Each TM can be written down as a long binary string
- ==> Canonical ordering of TMs:
  - $\{M_1, M_2, M_3, M_4, \ldots.\ M_i, \ldots\}$

The Diagonalization Language

- $L_d = \{ w_i \mid w_i \notin L(M_i) \}$
  - The language of all strings whose corresponding machine does *not* accept itself (i.e., its own code)



	1	2	3	4	…
1	0	1	0	1	…
2	1	1	0	0	…
3	0	1	0	1	…
4	1	0	0	1	…
⋮					⋱

- Table: T[i,j] = 1, if M_i accepts w_j
  = 0, otherwise.

- Make a new language called **$L_d = \{w_i \mid T[i,i] = 0\}$**

L_d is not RE (i.e., has no TM)

- Proof (by contradiction):
- Let M be the TM for L_d
- ==> M has to be equal to some M_k s.t.

  $L(M_k) = L_d$
- ==> Will w_k belong to $L(M_k)$ or not?

1. If $w_k \in L(M_k)$ ==> T[k,k]=1 ==> $w_k \notin L_d$
2. If $w_k \notin L(M_k)$ ==> T[k,k]=0 ==> $w_k \in L_d$

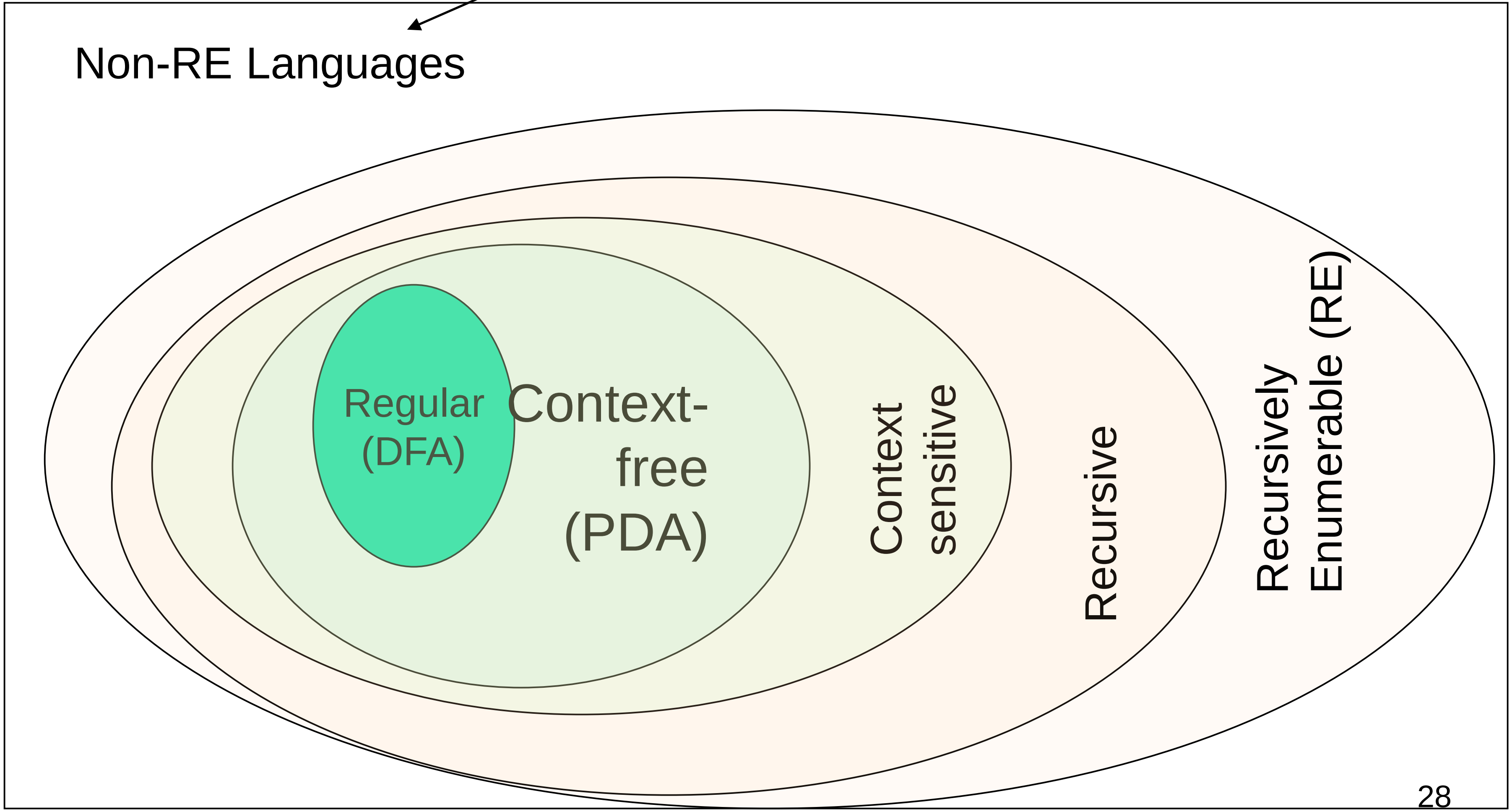
- A contradiction either way!!

Why should there be languages that do not have TMs?

We thought TMs can solve everything!!

Non-RE languages

How come there are languages here?
(e.g., diagonalization language)



One Explanation

There are more languages than TMs

  - By pigeon hole principle:
  - ==> some languages cannot have TMs

- But how do we show this?

- Need a way to “*count & compare*” two infinite sets (languages and TMs)

How to count elements in a set?

Let A be a set:

- If A is finite ==> counting is trivial
- If A is infinite ==> how do we count?
- And, how do we compare two infinite sets by their size?

Cantor's definition of set "size" for infinite sets (1873 A.D.)

Let N = {1,2,3,…} (all natural numbers)

Let E = {2,4,6,…} (all even numbers)

Q) Which is bigger?

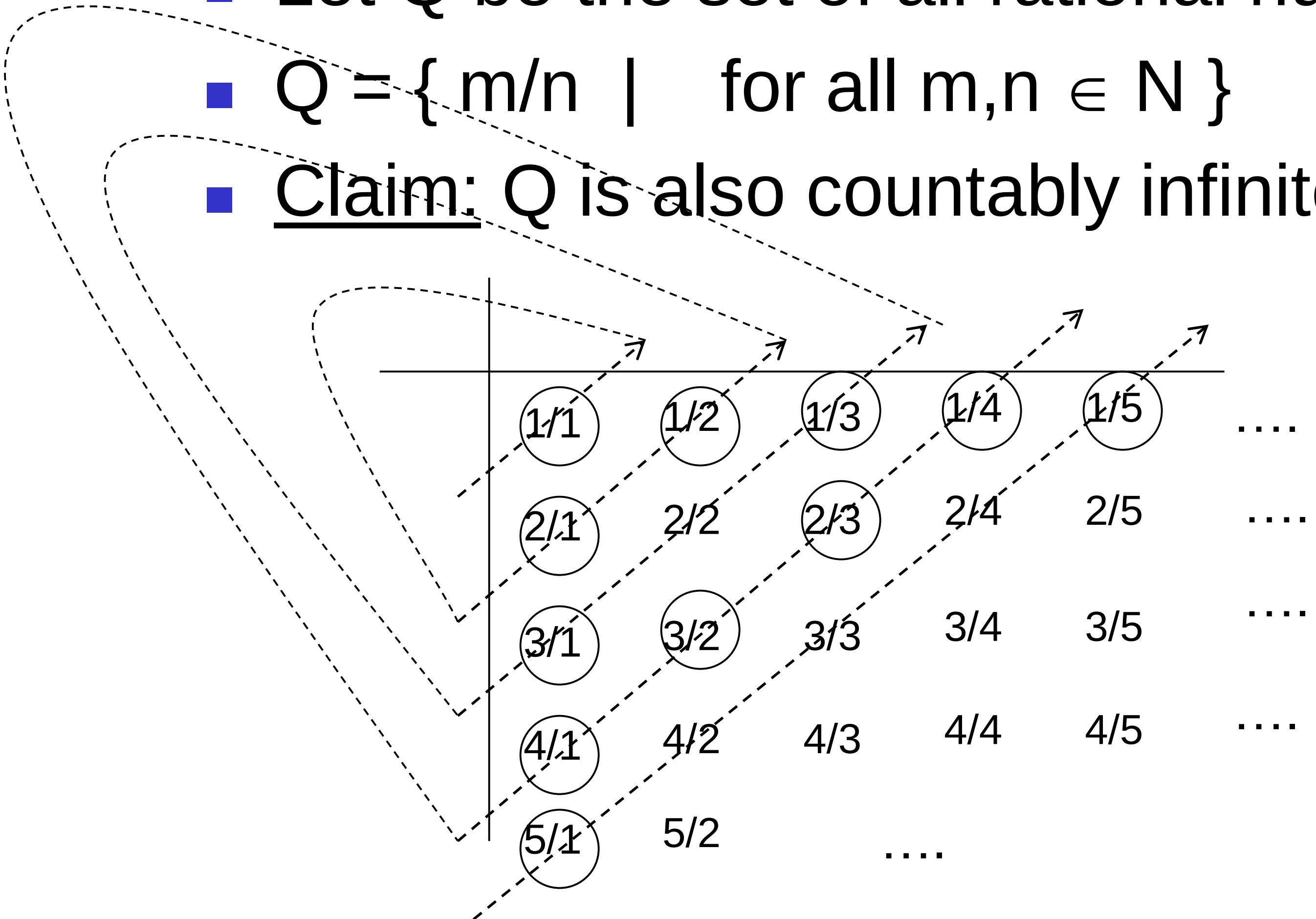
- A) Both sets are of the same size
  - "***Countably infinite***"
  - Proof: Show by one-to-one, onto set correspondence from N ==> E

n	*f(n)*
1	2
2	4
3	6
.	.
.	.

i.e, for every element in N,
there is a unique element in E,
and vice versa.

Example #2

- Let Q be the set of all rational numbers
- Q = { m/n | for all m,n ∈ N }
- Claim: Q is also countably infinite; => |Q|=|N|



1/1	1/2	1/3	1/4	1/5	….
2/1	2/2	2/3	2/4	2/5	….
3/1	3/2	3/3	3/4	3/5	….
4/1	4/2	4/3	4/4	4/5	….
5/1	5/2	….			

Really, really big sets!
(even bigger than countably infinite sets)

Uncountable sets

Example:

- Let R be the set of all real numbers
- Claim: R is uncountable

n	*f(n)*
1	3 . $\underline{1}$ 4 1 5 9 …
2	5 . 5 $\underline{5}$ 5 5 5 …
3	0 . 1 2 $\underline{3}$ 4 5 …
4	0 . 5 1 4 $\underline{3}$ 0 …
.	
.	
.	

Build x s.t. x cannot possibly occur in the table

E.g. x = 0 . 2 6 4 4 …

Therefore, some languages cannot have TMs…

- The set of all TMs is countably infinite
- The set of all Languages is uncountable
- ==> There should be some languages without TMs (by PHP)

Summary

- Problems vs. languages
- Decidability
  - Recursive
- Undecidability
  - Recursively Enumerable
  - Not RE
  - Examples of languages
- The diagonalization technique
- Reducability

Final Review

Objectives

- Introduce concepts in automata theory and theory of computation
- Identify different formal language classes and their relationships
- Design grammars and recognizers for different formal languages
- Prove or disprove theorems in automata theory using its properties
- Determine the decidability and intractability of computational problems

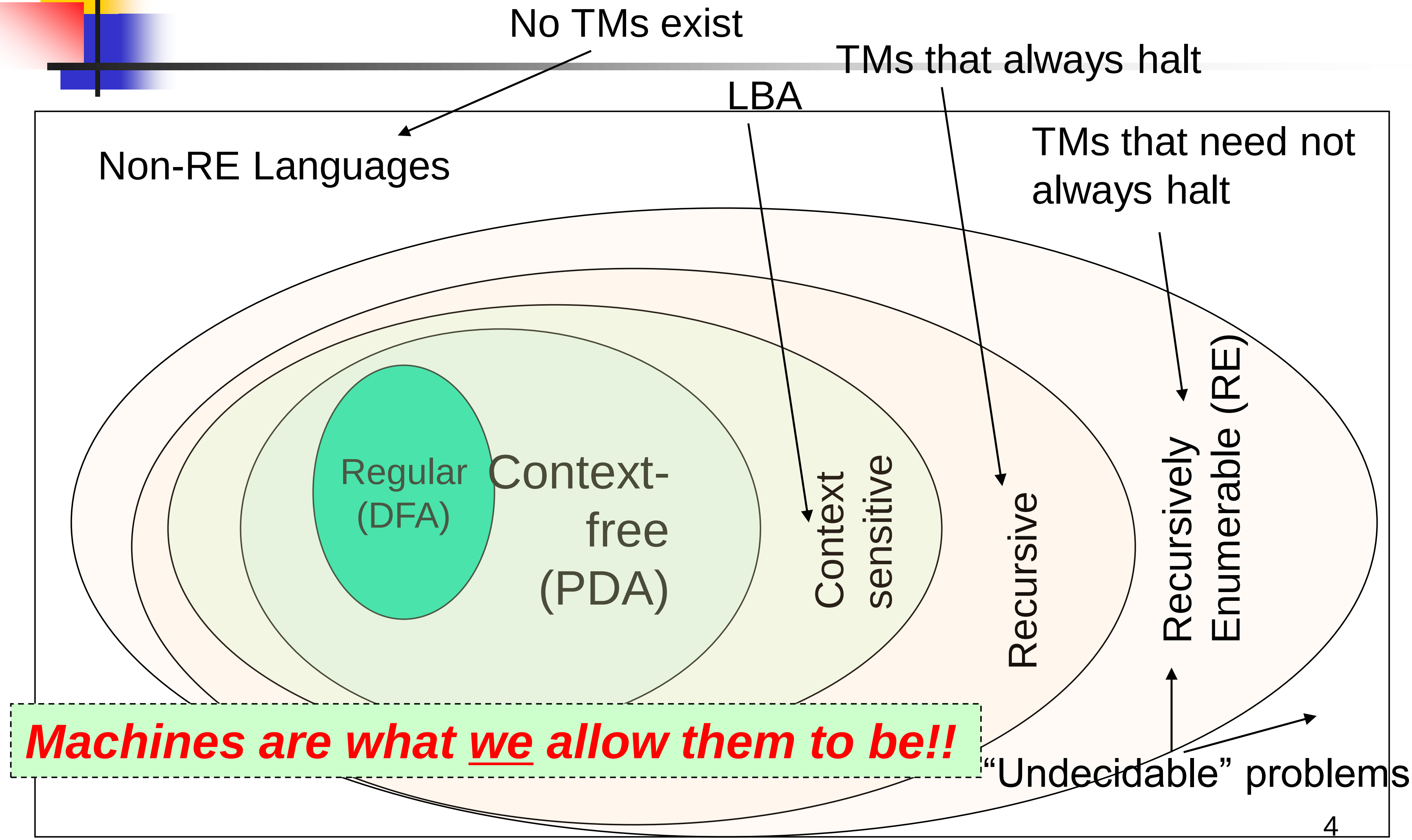
Main Topics

Part 1) Regular Languages

Part 2) Context-Free Languages

Part 3) Turing Machines & Computability

The Chomsky hierarchy for formal languages



No TMs exist

TMs that always halt

LBA

Non-RE Languages

TMs that need not always halt

Regular (DFA)

Context-free (PDA)

Context sensitive

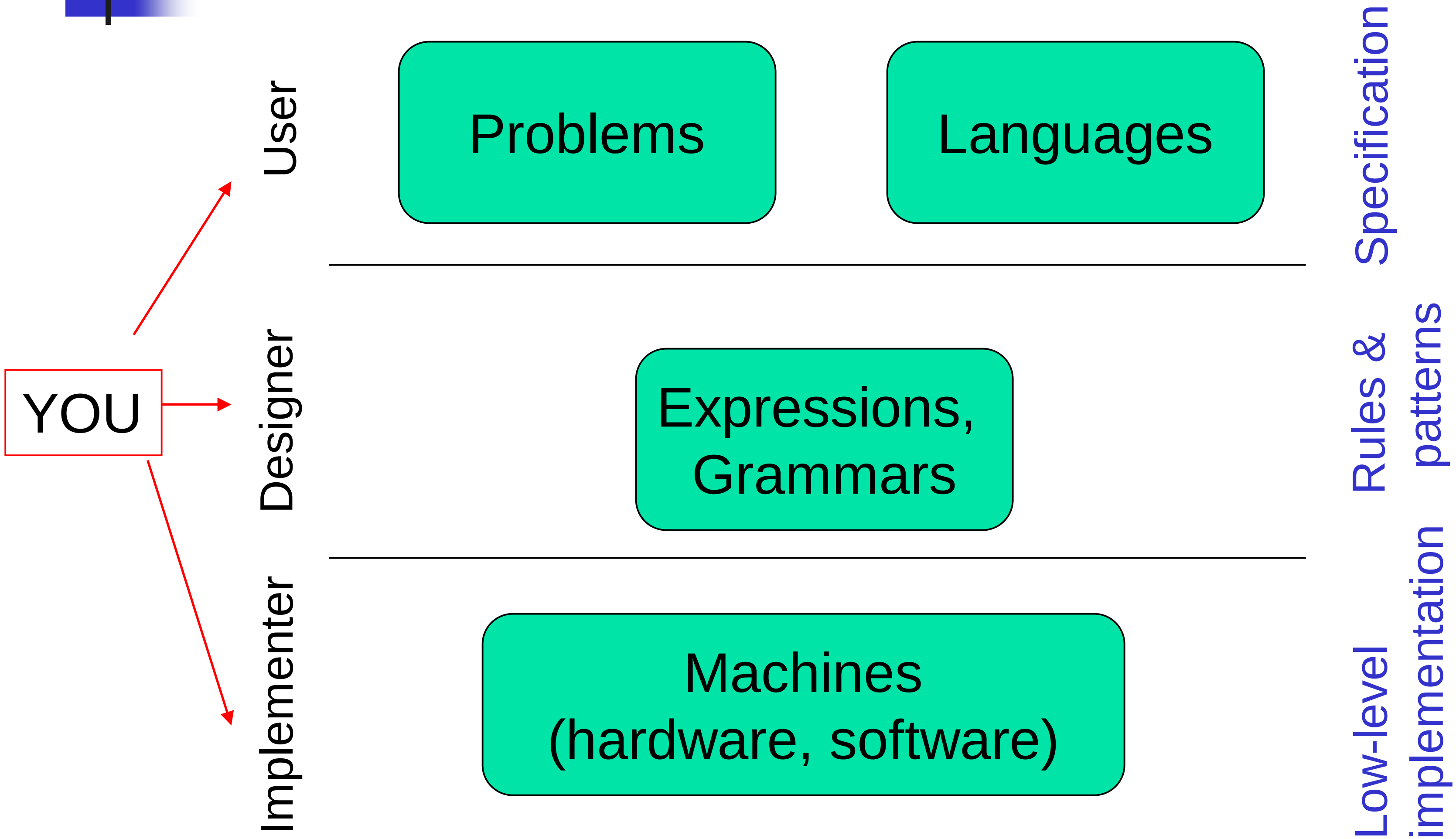
Recursive

Recursively Enumerable (RE)

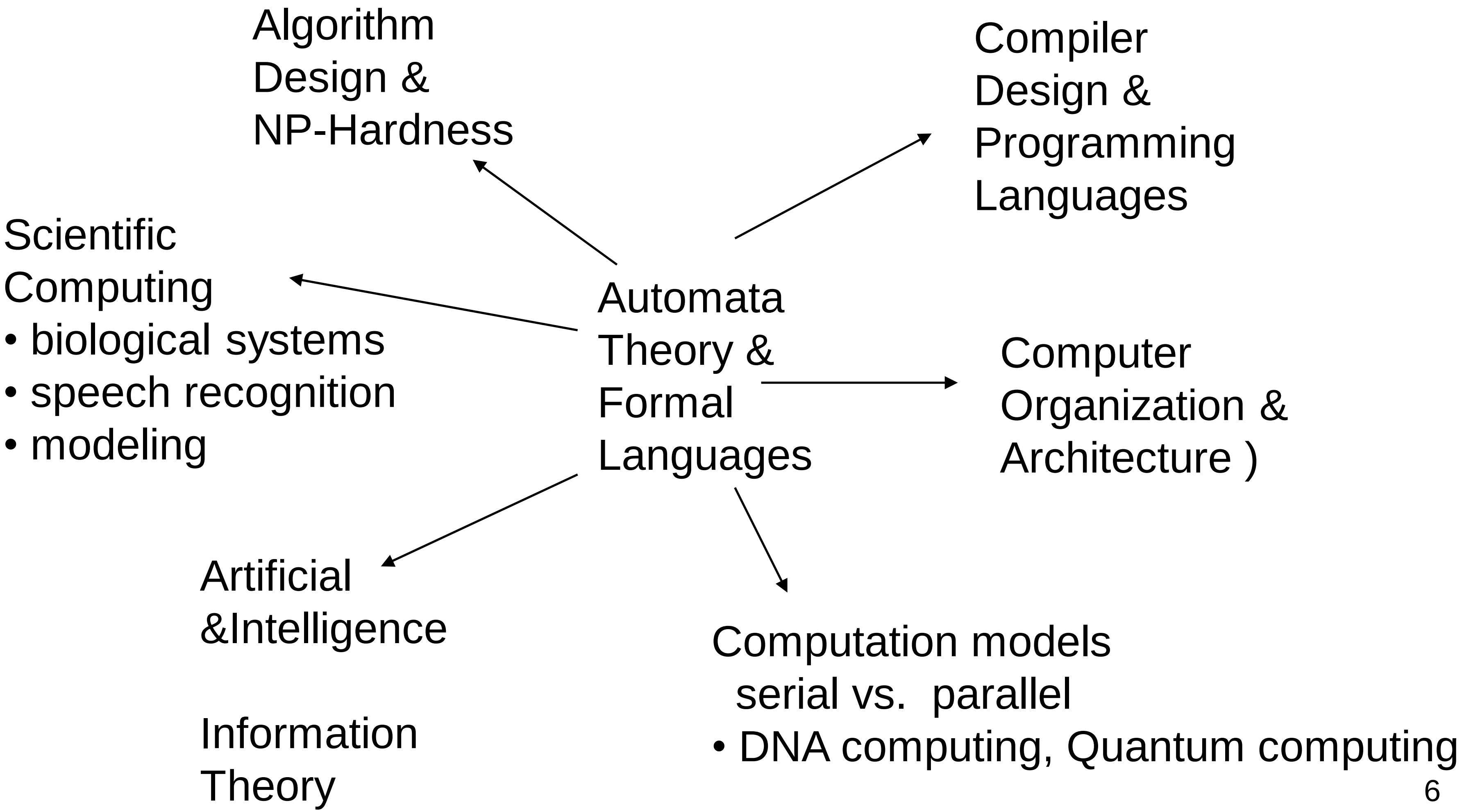
"Undecidable" problems

Machines are what we allow them to be!!

Interplay between different computing components



Automata Theory & Modern-day Applications



Automata Theory & Formal Languages

Algorithm Design & NP-Hardness

Compiler Design & Programming Languages

Scientific Computing
- biological systems
- speech recognition
- modeling

Computer Organization & Architecture)

Artificial &Intelligence

Information Theory

Computation models
serial vs. parallel
- DNA computing, Quantum computing

Regular Languages Topics

- Simplest of all language classes

- Finite Automata
  - NFA, DFA, ε-NFA
- Regular expressions
- Regular languages & properties
  - Closure
  - Minimization

Finite Automata

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

- ε-NFA is an NFA that allows ε-transitions

- What are their differences?
- Conversion methods

Deterministic Finite Automata

- A DFA is defined by the 5-tuple:
  - $\{Q, \sum, q_0, F, \delta\}$
- Two ways to represent:
  - State-diagram
  - State-transition table
- DFA construction checklist:
  - States & their meanings
  - Capture all possible combinations/input scenarios
    - break into cases & subcases wherever possible)
  - Are outgoing transitions defined for every symbol from every state?
  - Are final/accepting states marked?
  - Possibly, dead-states will have to be included

Non-deterministic Finite Automata

- A NFA is defined by the 5-tuple:
  - $\{Q, \sum, q_0, F, \delta\}$
- Two ways to represent:
  - State-diagram
  - State-transition table

- NFA construction checklist:
  - Introduce states only as needed
  - Capture only valid combinations
    - Ignore invalid input symbol transitions (allow that path to die)
  - Outgoing transitions defined only for valid symbols from every state
  - Are final/accepting states marked?

NFA to DFA conversion

- Checklist for NFA to DFA conversion
  - Two approaches:
    - Enumerate all possible subsets, or
    - Use *lazy construction* strategy (to save time)
      - Introduce subset states only as needed
  - Any subset containing an accepting state is also accepting in the DFA
  - Have you made a special entry for Φ, the empty subset?
    - This will correspond to dead state

ε-NFA to DFA conversion

- Checklist for €-NFA to DFA conversion
  - First take ECLOSE(start state)
  - New start state = ECLOSE(start state)
  - Remember: ECLOSE(q) include q

  - Same two approaches as NFA to DFA:
    - Enumerate all possible subsets, or
    - Use *lazy construction* strategy (to save time)
      - Introduce subset states only as needed

  - Only difference: take ECLOSE *both before & after* transitions

  - The subset Φ corresponds to a "dead state"

Regular Expressions

- A way to express accepting patterns
- Operators for Reg. Exp.
  - (E), L(E+F), L(EF), L(E*)..
- Reg. Language → Reg. Exp. (checklist):
  - Capture all cases of valid input strings
  - Express each case by a reg. exp.
  - Combine all of them using the + operator
  - Pay attention to operator precedence

Regular Expressions…

- DFA to Regular expression
  - Enumerate all paths from start to every final state
  - Generate regular expression for each segment, and concatenate
  - Combine the reg. exp. for all each path using the + operator
- Reg. Expression to ε-NFA conversion
  - Inside-to-outside construction
  - Start making states for every atomic unit of RE
  - Combine using: concatenation, + and * operators as appropriate
  - For connecting adjacent parts, use ε-jumps
  - Remember to note down final states

Regular Expressions…

- Algebraic laws
  - Commutative
  - Associative
  - Distributive
  - Identity
  - Annihiliator
  - Idempotent
  - Involving Kleene closures (* operator)

English description of lang.

- For finite automata
- For Regular expressions
- When asked for “English language descriptions”:
  - Always give the description of *the underlying language that is accepted by that machine or expression*

    (and not of the machine or expression)

Pumping Lemma

- Purpose: Regular or not? Verification technique
- Steps/Checklist for Pumping Lemma:
  - Let n ← pumping lemma constant
  - Then construct input w which has n or more characters
  - Now w=xyz should satisfy P/L
  - Check all three conditions
  - Then use one of these 2 strategies to arrive at contradiction for some other string constructed from w:
    - Pump up (k >= 2)
    - Pump down (k=0)

Reg. Lang. Properties

- Closed under:
  - Union
  - Intersection
  - Complementation
  - Set difference
  - Reversal
  - Homomorphism & inverse homomorphism
- Look at all DFA/NFA constructions for the above

Other Reg. Lang. Properties

- Membership question
- Emptiness test
  - Reachability test
- Finiteness test
  - Remove states that are:
    - Unreachable, or cannot lead to accepting
  - Check for cycle in left-over graph
  - Or the reg. expression approach

DFA minimization

- Steps:
  - Remove unreachable states first
  - Detect equivalent states
- Table-filing algorithm (checklist):
  - First, mark X for accept vs. non-accepting
  - Pass 1:
    - Then mark X where you can distinguish by just using one symbol transition
    - Also mark = whenever states are equivalent.
  - Pass 2:
    - Distinguish using already distinguished states (one symbol)
  - Pass 3:
    - Repeat for 2 symbols (on the state pairs left undistinguished)
    - …
  - Terminate when all entries have been filled
  - Finally modify the state diagram by keeping one representative state for every equivalent class

Other properties

- Are 2 DFAs equivalent?
  - Application of table filling algo

CFL Topics

- CFGs
- PDAs
- CFLs & pumping lemma
- CFG simplification & normal forms
- CFL properties

CFGs

- G=(V,T,P,S)
- Derivation, recursive inference, parse trees
  - Their equivalence
- Leftmost & rightmost derivation
  - Their equivalence
  - Generate from parse tree
- Regular languages vs. CFLs
  - Right-linear grammars

CFGs

- Designing CFGs
  - Techniques that can help:
    - Making your own start symbol for combining grammars
      - Eg., $S \Rightarrow S_1 \mid S_2$ (or) $S \Rightarrow S_1 S_2$
    - Matching symbols: (e.g., S => a S a | …)
    - Replicating structures side by side: (e.g., S => a S b S)
    - Use variables for specific purposes (e.g., specific sub-cases)
    - To go to an acceptance from a variable
      - ==> end the recursive substitution by making it generate terminals directly
      - A => w
    - Conversely, to *not* go to acceptance from a variable, have productions that lead to other variables
- Proof of correctness
  - Use induction on the string length

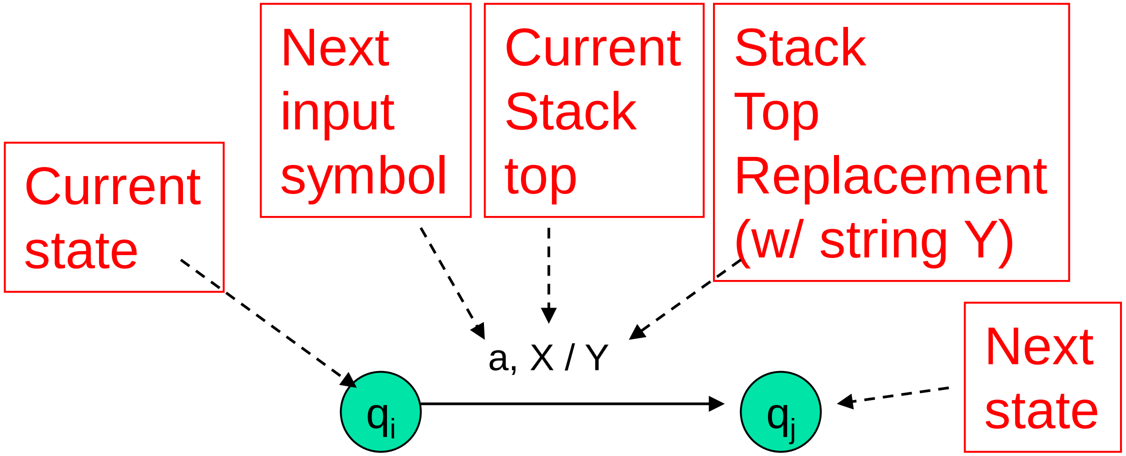
CFGs…

- Ambiguity of CFGs
  - One string <==> more than one parse tree
  - Finding one example is sufficient
- Converting ambiguous CFGs to non-ambiguous CFGs
  - Not always possible
  - If possible, uses ambiguity resolving techniques (e.g., precedence)
- Ambiguity of CFL
  - It is not possible to build even a single unambiguous CFG

There can be only 1 stack top symbol
There can be many symbols for the replacement

PDAs

- PDA ==> ε-NFA + “a stack”
- $P = (Q,\sum,\Gamma, \delta,q_0,Z_0,F)$
- $\delta(q,a,X) = \{(p,Y), \ldots\}$
- ID : (q, aw, XB) |--- (p,w,AB)
- State diagram way to show the design of PDAs


Current state

Next input symbol

Current Stack top

Stack Top Replacement (w/ string Y)

Next state

Designing PDAs

- Techniques that can help:
  - Two types of PDAs
    - Acceptance by empty stack
      - If no more input and stack becomes empty
    - Acceptance by final state
      - If no more input and end in final state
  - Convert one form to another
  - Assign state for specific purposes
  - Pushing & popping stack symbols for matching
  - Convert CFG to PDA
  - Introducing new stack symbols may help
  - Take advantage of non-determinism

CFG Simplification

1. Eliminate ε-productions: A => ε
   - ==> substitute for A (with & without)
   - Find nullable symbols first and substitute next
2. Eliminate unit productions: A=> B
   - ==> substitute for B directly in A
   - Find unit pairs and then go production by production
3. Eliminate useless symbols
   - Retain only reachable and generating symbols

- Order is important : steps (1) => (2) => (3)

Chomsky Normal Form

- All productions of the form:
    - A => BC   or   A=> a
- Grammar does not contain:
    - Useless symbols, unit and €-productions
- Converting CFG (without S=>* ε) into CNF
    - Introduce new variables that collectively represent a sequence of other variables & terminals
    - New variables for each terminal
- CNF ==> Parse tree size
    - If the length of the longest path in the parse tree is n, then $|w| \leq 2^{n-1}$.

Pumping Lemma for CFLs

- Then there exists a constant N, s.t.,
  - if z is any string in L s.t. |z|≥N, then we can write z=uvwxy, subject to the following conditions:
    1. |vwx| ≤ N
    2. vx≠ ε
    3. For all k≥0, uv^kwx^ky is in L

Using Pumping Lemmas for CFLs

- Steps:
  1. Let N be the P/L constant
  2. Pick a word z in the language s.t. |z|≥N
     - (choice critical - an arbitrary choice may not work)
  3. z=uvwxy
  4. First, argue that because of conditions (1) & (2), the portions covered by vwx on the main string z will have to satisfy some properties
  5. Next, argue that by pumping up or down you will get a new string from z that is not in L

Closure Properties for CFL

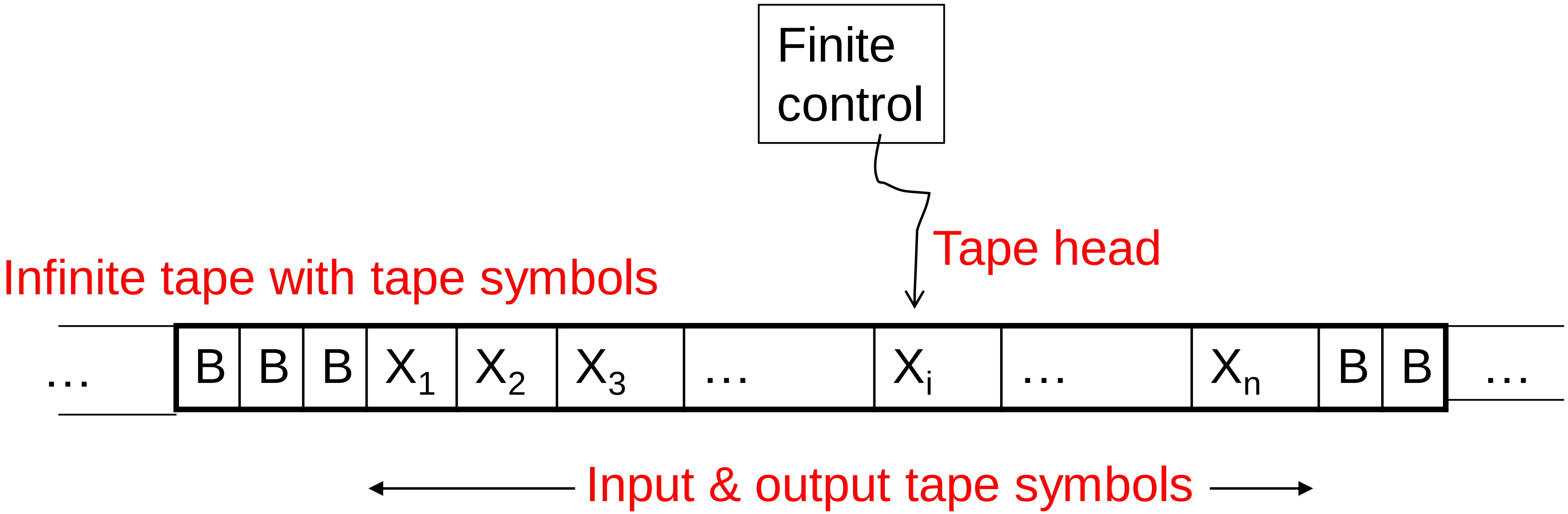
- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
- CFLs are *not* closed under:
  - Intersection
  - Difference
  - Complementation

Closure Properties

- Watch out for
  - custom-defined operators
    - Eg.. Prefix(L), or “L x M”
  - Custom-defined symbols
    - Other than the standard 0,1,a,b,c..
    - E.g, #, c, ..

The Basic Turing Machine (TM)

- $M = (Q, \sum, \Gamma, \delta, q_0, B, F)$



Finite control

Tape head

Infinite tape with tape symbols

…	B	B	B	X_1	X_2	X_3	…	X_i	…	X_n	B	B	…

Input & output tape symbols

B: end tape symbol (special)

Turing Machines & Variations

- Basic TM
- TM w/ storage
- Multi-track TM
- Multi-tape TM
- Non-deterministic TM

Unless otherwise stated, it is OK to give TM design in the pseudocode format

TM design

- Use any variant feature that may simplify your design
  - Storage - to remember last important symbol seen
  - A new track - to mark (without disturbing the input)
  - A new tape - to have flexibility in independent head motion in different directions
- Acceptance only by final state
- No need to show dead states
- Use ε-transitions if needed
- Invent your own tape symbols as needed

Recursive, RE, non-RE

- Recursive Language
  - TMs that always halt
- Recursively Enumerable
  - TMs that always halt only on acceptance
- Non-RE
  - No TMs exist that are guaranteed to halt even on accept
- Need to know the conceptual differences among the above language classes
  - Expect objective and/or true/false questions

Recursive Closure Properties

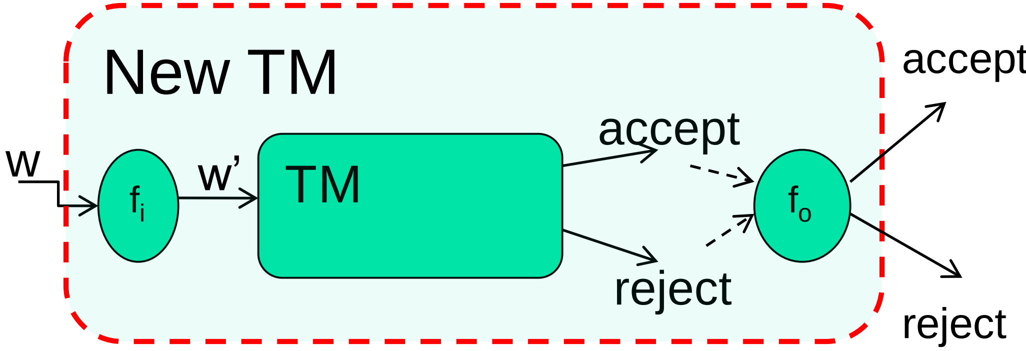
- Closed under:
  - Complementation, union, intersection, concatenation (discussed in class)
  - Kleene Closure, Homomorphism (not discussed in class but think of extending)

Tips to show closure properties on Recursive & RE languages

Build a new machine that wraps around the TM for the input language(s)

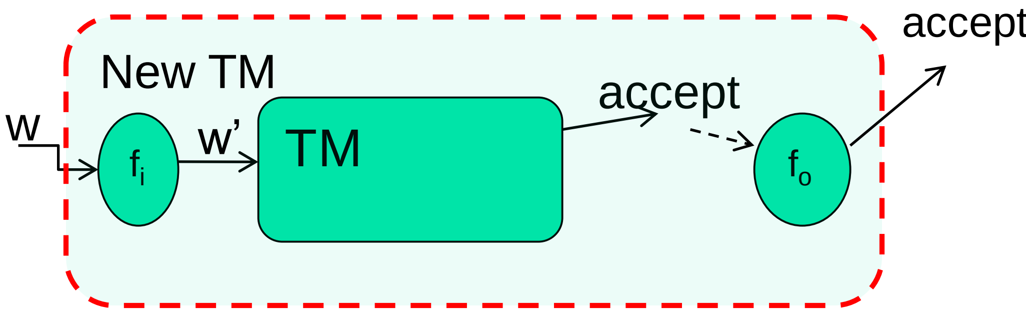
You need to define the input and output transformations (f_i and f_o)

- <u>For Recursive languages:</u>
  - The old TM is always going to halt (w/ accept or reject)
    => So will the new TM



- <u>For Recursively Enumerable languages:</u>
  - The old TM is guaranteed to halt only on acceptance

    => So will the new TM



MCQ s

Theory of Computation - I

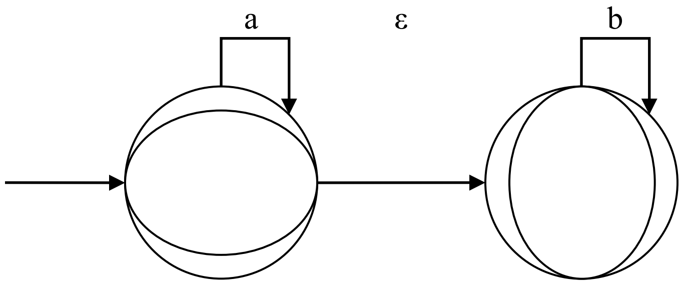
Q1. The regular expression (00+01+10+11)* corresponds to the Language [ ]

a) All Strings starts with 00
b) ALL strings ends with 11
c) Even length strings
d) None of the above

Answer---- c

It is not necessary to strings to start with 00 and end with 11, it generates even length strings

Q2. The regular expression for the Following NFA with ε is [ ]



a) a* b) a*b c) ab* d) a*b*

Answer ---d
The set accepted by the machine is { ε, a, b, ab, bb, aab, aabb, abbb, aaaab,…}
Hence a*b* is the regular expression

Q3. Every regular language is a cfl True/False [ ]

Answer----True
Since regular grammar is a subset of Context free Grammar

Q4. Which of the following grammar is not ambiguous [
]

a) S→S+S / S-S /a
b) S→iEtS / iEtSeS /a , E→b
c) S→SbS/a
d) None of these

Answer---d
All the grammars in options (a),(b) &c generates more than one parse tree for some string ,hence all the grammars given are ambiguous

Q5. Which one of the following problem for L cfl and R regular is not decidable
[ ]

a) L=R
b)R$\subseteq$ L
c) L=Ǿ
d) none of the above

Answer---d
All the given problems are decidable

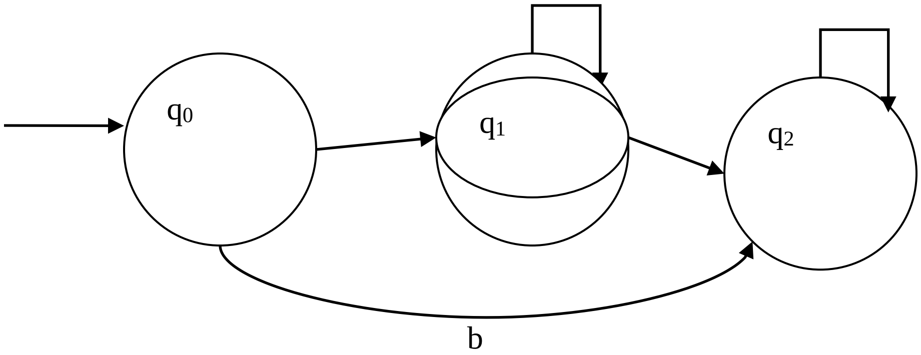
Q6. The following Language L={o^n1^m\n>=0 and m>=0} can be designed by [ ]

a) Finite Automata
b) Pushdown Automata
c) Turing Machine
d) All the above

Answer---d
Since the Language generated by the grammar is 0*1* , It can designed by finite automata, Hence it can be designed by PDA and TM

Q7. In the Following figue q2 is

a b a, b [ ]



a)Final state
b)Start state
c)Trap state
d)None of the above

Answer---c
Trap state is a state from which we will be not coming back to final state for any input. In the above diagram q2 is trap state

Q8. NFA and DFA differs in [ ]
a) Start state b) Mapping function
c) Input alphabet d) All the above

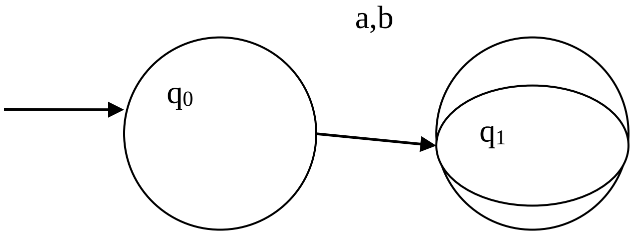
Answer--b

Start state and input alphabet are same when nfa is converted to dfa, only differs in mapping function

Q9. In NFA we have Q states, then we have __________ states in DFA [ ]
a)2*Q b)Q/2 c)2^Q d)Q^2*2

Answer ---c
Converting nfa to dfa

Q10. The regular expression for the following machine is [ ]
a)ab b)a+b c)a d)b



Answer---b

Q11. In the following diagram ε-closure(q_0) is [ ]

a){q_1,q_2,q_3} b) {q_1,q_3}

c){q_0,q_1,q_3] d) {q_0,q_1,q_2,q_3}

Answer ---d

Q12.The following moore m/c gives how many time the substring _____ occurs in the long input string with input alphabet x,y

[ ]

a)xy b)xxy c)xyxy d)xyy

Answer ---b

Q13. Consider the following languages

L1={o^n1^m / n>=1 and m>=1} is a regular language

L2={O^{2n} / n>=1 } is a regular language

Which of the following statement is correct [ ]

a) L1 is correct
b) L2 is correct
c) Both L1 and L2 are correct
d) None of L1 and L2 are correct

Answer ---c

L1 is equivalent to 0*1* , which is regular language

L2 can be designed by Finite automata

Hence L1 and L2 are regular languages

Q14. The following language is regular True/False

[ ]

L={a^p / p is prime]

Answer --- False

By pumping lemma for regular set, we cannot select v for the uv ^{i}w , such that the string belongs to prime number of a's

Q15.The Following grammar generates the language [ ]

S→AXC / XBC / AYC /ABY
X→aXb / ab
Y→bYC / bc
A→aA /a
B->bB / b
C->cC /c

a) { $a^i b^j c^k$ / i≠j or j=k}

b) { $a^i b^j c^k$ / i=j or j=k}
c) { $a^i b^j c^k$ / i=j or j≠k}
d) { $a^i b^j c^k$ / i≠j or j≠k}

Answer ---d

X represents equal number of a’s &b’s
Y represents equal number of b’s &c’s
A represents one or more number of a’s
B represents one or more number of b’s
C represents one or more number of c’s

Q16. The useless symbols in the following grammar are [ ]
S→AB / a
A→a
a) A b) B c) A&B d) None of these

Answer --- c
B is useless symbol as B is not generating any terminal
Since S -> AB, A is also useless symbol

Q17. Which of the following machine requires stack [ ]
a) Finite automata b) PushDownAutomata
c) Turing machine d) None of these

Answer ---b
Finite automata and Turing machine does not have stack
PushDownAutomata requires stack

Q18. In abstract Syntax Tree, the interior nodes corresponds to
[ ]
a) variables b)Terminals c)Any grammar symbol d) all of the above

Answer---b
In Abstract Syntax Tree all nodes corresponds to terminals.

Q19. One of the following problem is not decidable for cfl [ ]

a) membership b) ambiguity
c) When L is empty d) Whether L is finite

Answer --- b
Ambiguity is undecidable for context free langauage

Q20.Considet the following grammar [ ]

E→E+T / T
T→T*F /F
F→ (E) / id
The above grammar has
a) Ambiguity b) left recursion c) both a &b d) None

Answer --- c
Since the grammar is generating more than one parse for any string eg id+id*id
Since the productions are of the form A->Aα/β eg E->E+T/T
Hence grammar is ambiguous and left recursive

Q21. Which of the following regular expressions are equivalent
I) 1*(1+ ε) II) 1^+ III) 1* IV) ε [ ]
a) I &II b) I & III c) I & IV d) None

Answer --- a
Both represent the same set { ε, 1, 11, 111,…}

Q22. The following regular grammar [ ]

A→0A / 1B
B→ 0A /1B /0
Represents the languages ending with
a) 11 b) 00 c) 10 d) 01

Answer ---c
A->0A
->01B
->010A
->0101B
->01010
Consider any derivation, string will end in 10

Or
Construct Finite Automata and check the inputs accepted by the machine

Q23. The regular expression (P+Q)* is equivalent to
[ ]
a) (P* Q*)*
b) (P* + Q*)*
c) P* + 2*P*Q+Q*
d) Both a & b

Answer – d
By mathematical induction

Q24. The regular expression equivalent to the following FA is [ ]

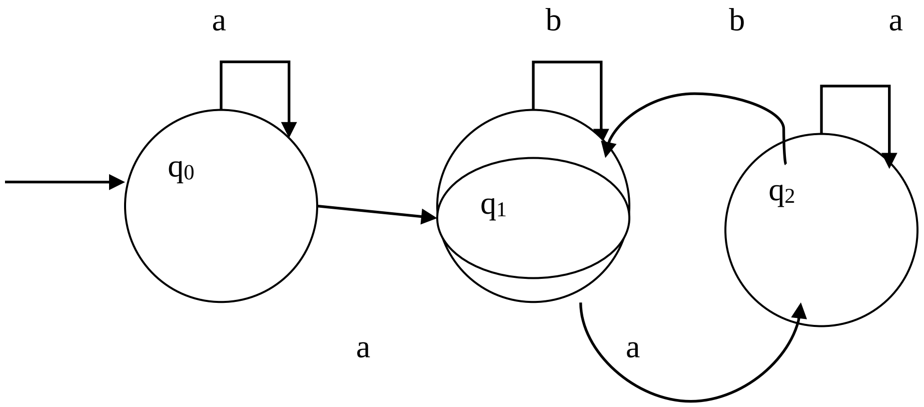
a) 0* b) 1+ c) 0*1* d) 0*1*

Answer ---c

Q25. If G is a context-sensitive grammar, there is an algorithm to determine L (G) is infinite True/False [ ]

Answer --- False
Language generated by context sensitive grammar is finite or infinite is undecidable

Q26. English description of the language accepted by the automation depicted in the following diagram [ ]



a) all strings exactly one ‘a’
b) if it has got more than one ‘a’ it should end with b
c) Both a & b

d)None of these

Answer --- c

The regular expression is a+a*b , hence both a&b are true

Q27. The regular expression for the strings with an odd number of 1's [ ]

a) 0*10* (10*10*)*
b) 0*1(10*10)*1
c) 0*11(101)*
d) (1+01)* (1+01)

Answer ---a

In this regular expression 0*10* (10*10*)* we have (10*10*)* which is having even number of 1's ,and 0*10* (10*10*)* makes odd number of 1's

Common Data Question

Q28. Consider The following PDA M= ({q_0, q_1, q_2}, {a, b}, {X}, S, z_o, q_o, q_2} [ ]

δ (q_0, a, z_0) = (q_0, Xz_0)
δ (q_0, a, X) = (q_0, XX)
δ (q_0, b, X) = (q_1, X)
δ (q_1, b, X) = (q_1, X)
δ (q_1, a, X) = (q_2, ε)
δ (q_2, a, X) = (q_2, ε)
δ (q_2, ε, z_0) = (q_2, ε)

Q28. i) give the language accepting by empty store {$a^mb^mc^n$ / m, n, >, 1}

a) {$a^mb^nc^n$ / m, n>, 1}
b) {$a^mb^mc^m$ / m, n>, 1}
c) {$a^mb^nc^m$ / m, n, >, 1}

Answer ---d
Consider the input aaabbccc it will make the stack empty
OR
Design a PDA and check the input aaabbccc

Q28.ii)give the instantenous description for the input aaabbccc

Linked Question

Q29.Given grammar G({S},{a,b},P,S)is S->aSa

Q29 i) Add some productions to the grammar so that it generates even palindrome

Answer ---- S->bSb / ε

S->aSa->abSba->abab

Q29 ii)Add some productins to the grammar so that it generates odd palindrome
Answer ---- S->bSb / a /b

S->aSa->abSba->ababa

Q30. Which one of the following regular expressions is NOT equivalent to the regular expression (a + b + c)* ?
A) (a* + b* + c*)*
B) (a*b*c*)*
C) ((ab)* + c*)*
D) (a*b* + c*)*

Answer ---c