COMP-330
Theory of Computation

Fall 2019 -- Prof. Claude Crépeau

Part II : Lec. 10-23

Definition of CFG

- Variables $A, B, C, \langle TERM \rangle, \langle EXPR \rangle$
- Alphabet (of terminals) $0, 1, \#$
- Substitution Rules
  $$A \rightarrow 0A1$$
  $$\langle EXPR \rangle \rightarrow \langle TERM \rangle$$
- Start Variable A

  (left-hand side of the first substitution rule)

Definition of CFG

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Parse Tree

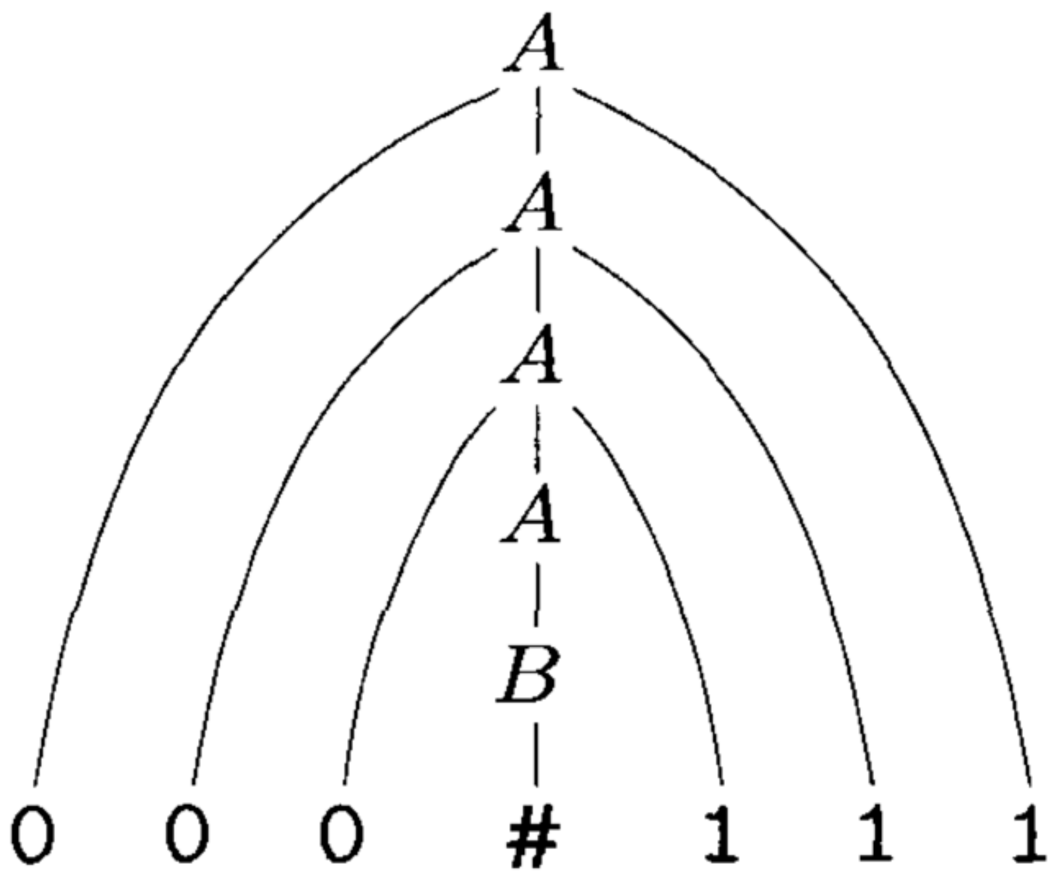


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

Definition of CFL

- If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv <u>yields</u> uwv, written $uAv \Rightarrow uwv$.
- We say that u <u>derives</u> v ($u \overset{*}{\Rightarrow} v$) if u=v or if

  $$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v,\ k \geq 0.$$

- The language of G is $\{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \}$.

Context-Free Grammars

- Formally, grammar G_1 :

$$
\begin{aligned}
V &= \{A,B\} \\
\Sigma &= \{0,1,\#\} \\
R &= \{A \rightarrow 0A1 \mid B, \\
&\qquad B \rightarrow \#\} \\
S &= A
\end{aligned}
$$

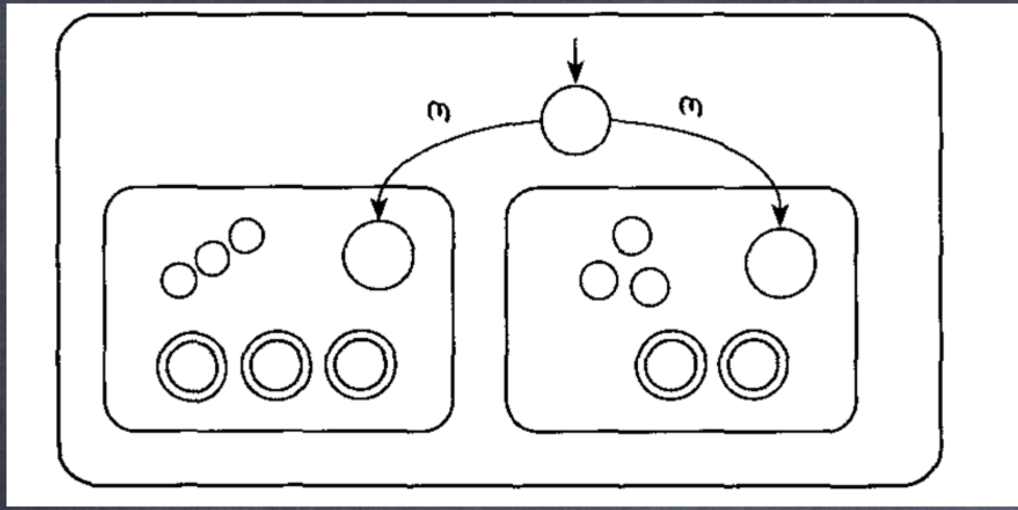
- $L(G_1) = \{ 0^n\#1^n \mid n \geq 0 \}$.

Regular Operations : Kleene's theorem (CFG)

Regular Operations : Kleene's theorem (CFL)

THEOREM

The class of CFLs is closed under the union operation.



Kleene's theorem (CFL)

- Let $G_A=(V_A,\Sigma,R_A,S_A)$ be a CFG generating L_A and $G_B=(V_B,\Sigma,R_B,S_B)$ be a CFG generating L_B ($V_A \cap V_B=\varnothing$).
- Consider

$G_U=(\{S_U\} \cup V_A \cup V_B,$

$\Sigma,$

$\{S_U \rightarrow S_A \mid S_B\} \cup R_A \cup R_B,$

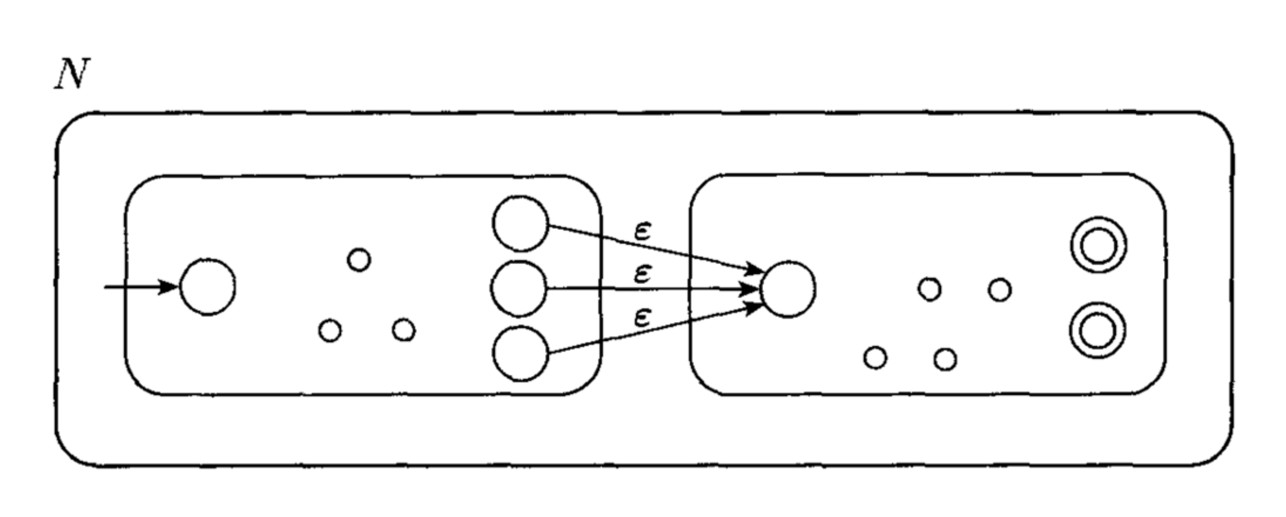
$S_U).$

- $L_U = L_A \cup L_B$.

Regular Operations : Kleene's theorem (CFL)

THEOREM

The class of CFLs is closed under the concatenation operation.



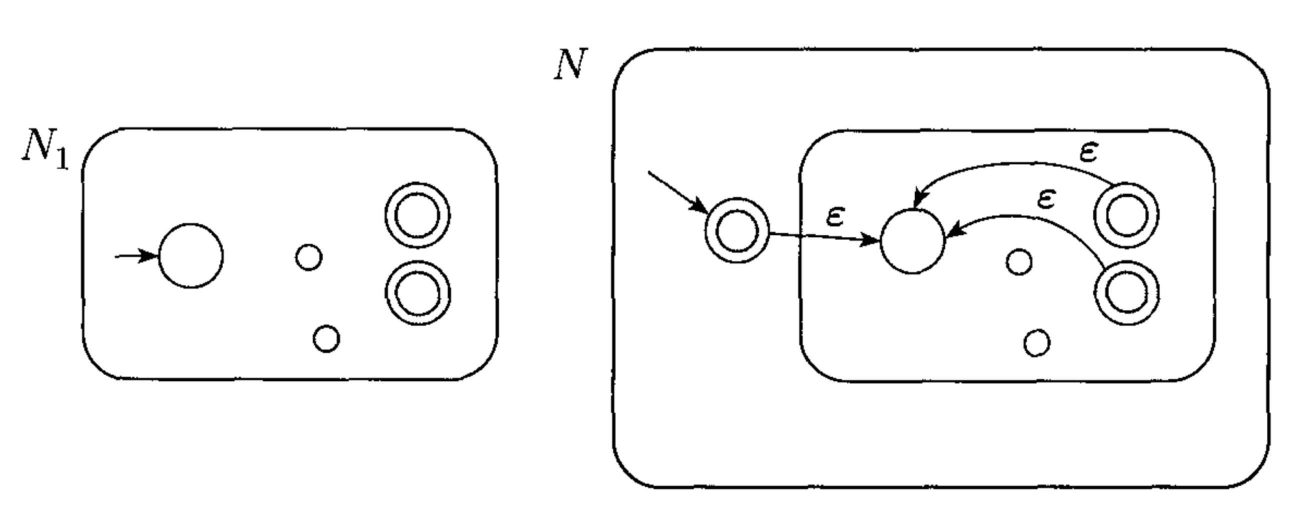
Kleene's theorem (CFL)

- Let $G_A=(V_A,\Sigma,R_A,S_A)$ be a CFG generating L_A and $G_B=(V_B,\Sigma,R_B,S_B)$ be a CFG generating L_B ($V_A \cap V_B=\emptyset$).
- Consider $G_C=($
  - $\{S_C\} \cup V_A \cup V_B$,
  - Σ,
  - $\{S_C \rightarrow S_A S_B\} \cup R_A \cup R_B$,
  - S_C).
- $L_C = L_A \circ L_B$.

Regular Operations : Kleene's theorem (CFL)

THEOREM ....

The class of CFLs is closed under the star operation.



Kleene's theorem (CFL)

- Let $G_A=(V_A,\Sigma,R_A,S_A)$ be a CFG generating L_A.
- Consider $G_S=($
  - $\{S_S\} \cup V_A$,
  - Σ,
  - $\{S_S \rightarrow \boldsymbol{\varepsilon} \mid S_A S_S\} \cup R_A$,
  - S_S).
- $L_S = (L_A)^*$.

Construction tools (and Reductions)

CFLs are closed under union, concatenation and star. If there exists a CFL C s. t. either $A^*=A'$, $A \cup C=A'$, $A \circ C=A'$

(**but neither complement nor intersection**)

or any combinations of these operations then A′ is a CFL as long as A is.

(If A′ is NON-CFL then so is A.)

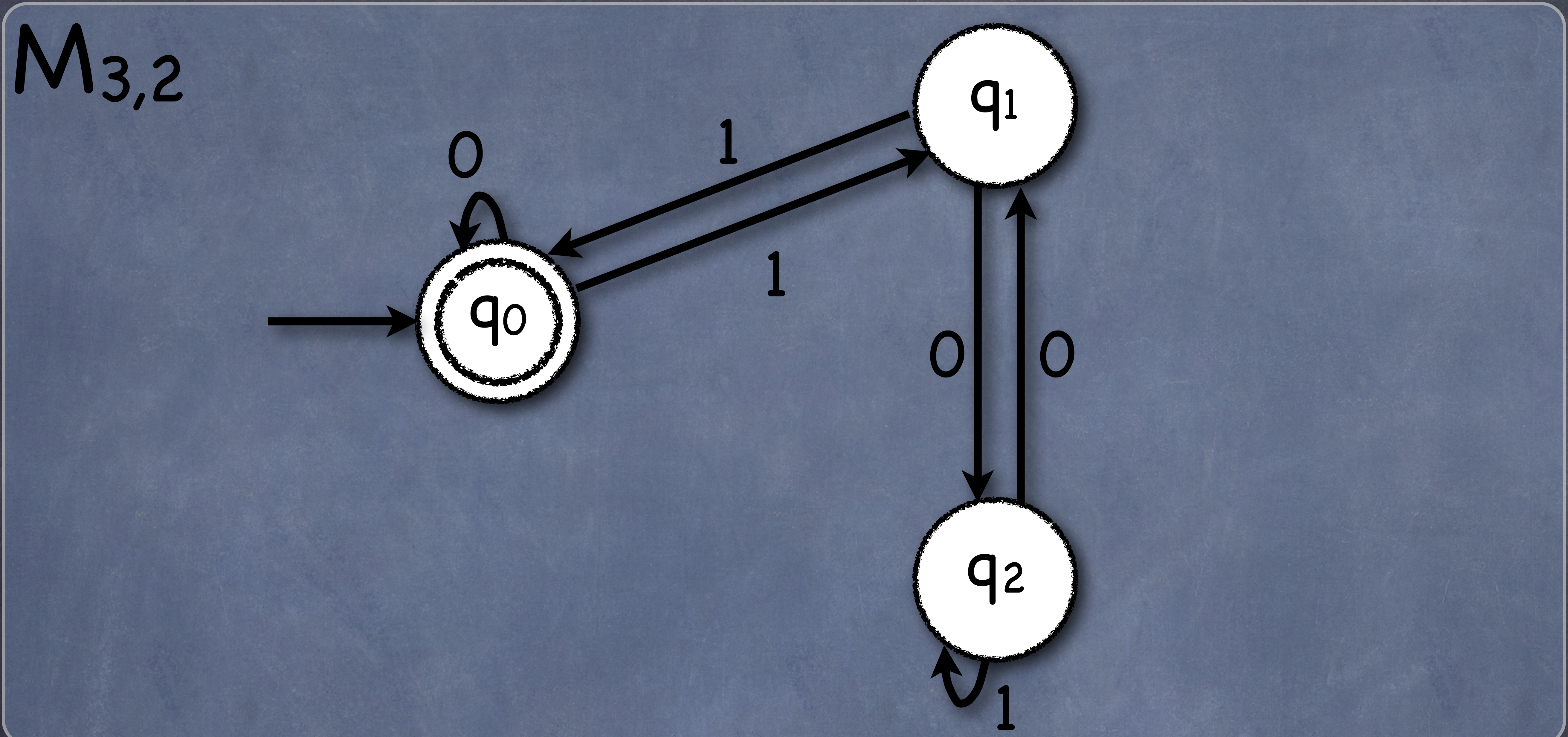
***Construction ***

[A]2.18 **a.** Let C be a context-free language and R be a regular language. Prove that the language $C \cap R$ is context free.

2.18 **(a)** Let C be a context-free language and R be a regular language. Let P be the PDA that recognizes C, and D be the DFA that recognizes R. If Q is the set of states of P and Q' is the set of states of D, we construct a PDA P' that recognizes $C \cap R$ with the set of states $Q \times Q'$. P' will do what P does and also keep track of the states of D. It accepts a string w if and only if it stops at a state $q \in F_P \times F_D$, where F_P is the set of accept states of P and F_D is the set of accept states of D. Since $C \cap R$ is recognized by P', it is context free.

Construction tools

- Constructing a CFG for a regular language L: $M = (Q=\{q_0,q_1,\ldots,q_k\},\Sigma,\delta,q_0,F)$ is converted to $G = (V=\{R_0,R_1,\ldots,R_k\},\Sigma,R,S=R_0)$ where
- R contains rule $R_i \to aR_j$ for each $\delta(q_i,a) = q_j$ in M, and rule $R_i \to \varepsilon$ for each accept-state $q_i \in F$.
- R_0 is the start variable.



- $M_{3,2} = (Q=\{q_0,q_1,q_2\},\{0,1\},\delta,q_0,F)$ is converted to $G_{3,2} = (V=\{R_0,R_1,R_2\},\{0,1\},R,S=R_0)$ where
- R: $R_0 \rightarrow 0R_0 \mid 1R_1 \mid \varepsilon$

  $R_1 \rightarrow 0R_2 \mid 1R_0$

  $R_2 \rightarrow 0R_1 \mid 1R_2$

extra EXAMPLE of CFG

EXAMPLE 2.4

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\texttt{a}, \texttt{+}, \texttt{x}, \texttt{(}, \texttt{)}\}$. The rules are

$$\begin{aligned}
\langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle \texttt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \texttt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle &\rightarrow \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a}
\end{aligned}$$

extra EXAMPLE of CFG

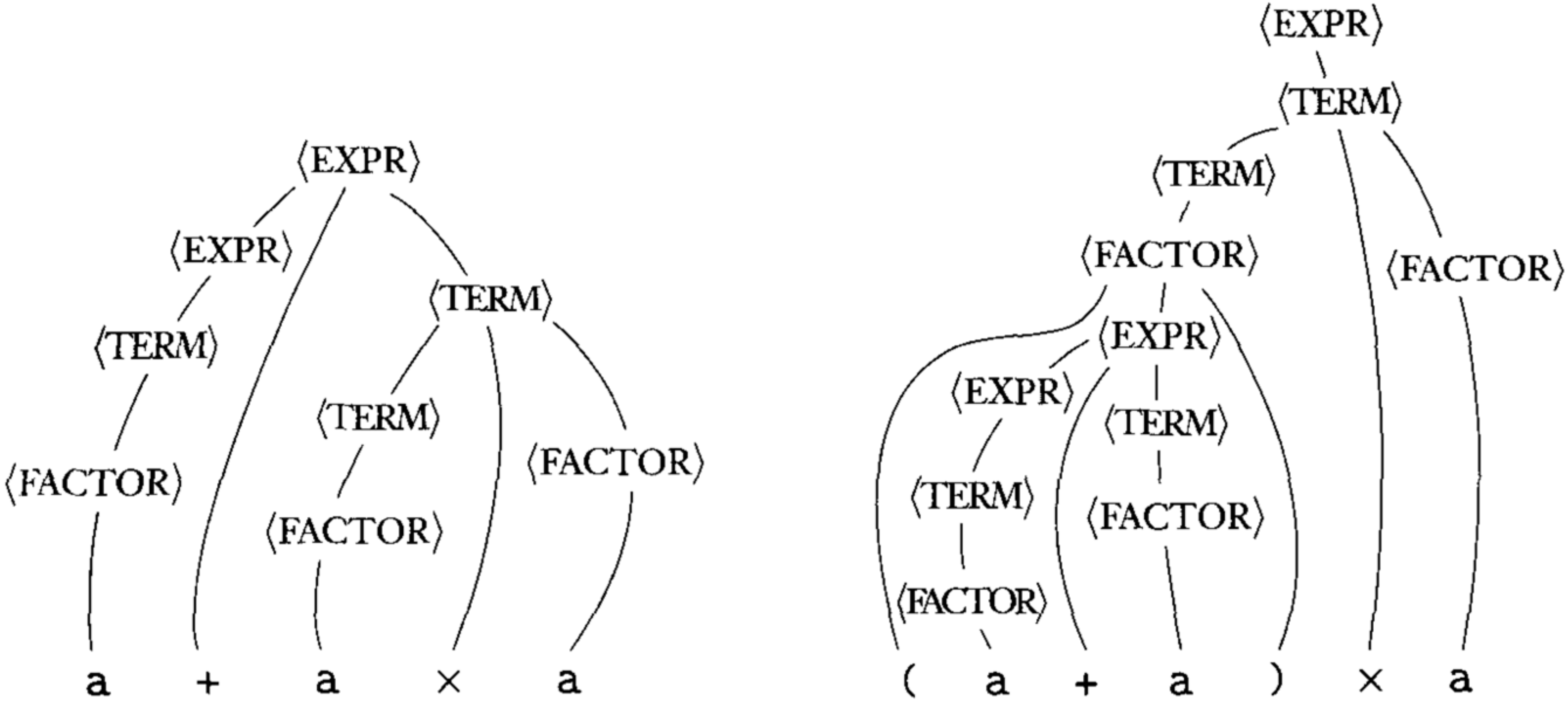


FIGURE 2.5
Parse trees for the strings a+a×a and (a+a)×a

Ambiguity in CFGs

Ambiguity

- A string w is derived <u>ambiguously</u> by a CFG G if it has two or more distinct leftmost derivations. Grammar G is <u>ambigious</u> if it generates some string ambiguously.

Ambiguous version of example 2.4

G_5

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

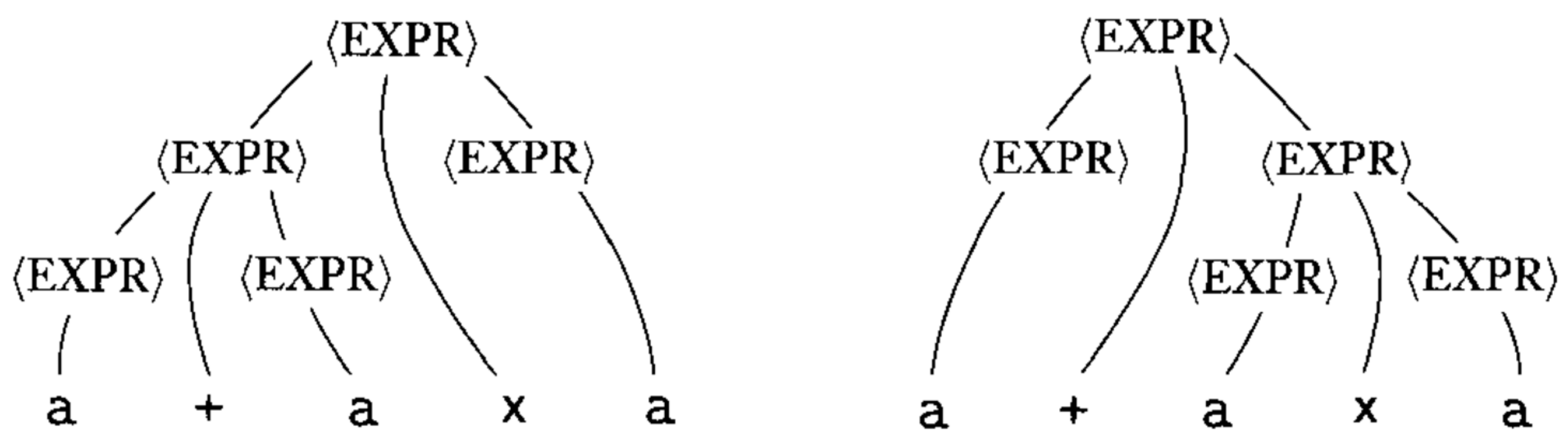


FIGURE 2.6
The two parse trees for the string a+a×a in grammar G_5

Ambiguity

- Ambiguity is not desirable in CFG because it may lead to unexpected interpretations of a string, for instance in the context of arithmetic expressions or programming languages.
- However, some languages are inherently ambiguous, meaning that all grammars generating this language must be ambiguous.
- example : $\{a^i b^j c^k \mid i=j \text{ or } j=k\}$


Noam Chomsky

Chomsky Normal Form

Chomsky Normal Form

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \to \varepsilon$, where S is the start variable.

Chomsky Normal Form

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

2.26 Show that, if G is a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps are required for any derivation of w.

$$A_{\text{CFG}} = \{\langle G, w\rangle |\ G \text{ is a CFG that generates string } w\}.$$

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.

To make this Turing machine into a decider we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 157) we showed that, if G were in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w. In that case checking only derivations with $2n - 1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

- Proof:
- First, we add a new start variable S_0 and the rule $S_0 \rightarrow S$, where S was the original start variable.

Chomsky Normal Form

EXAMPLE 2.10

Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given. The series of grammars presented illustrates the steps in the conversion. Rules shown in bold have just been added. Rules shown in gray have just been removed.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$\begin{aligned} S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} \boldsymbol{S_0} &\boldsymbol{\to} \boldsymbol{S} \\ S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

- Second, we take care of all ε-rules. We remove an ε-rule "$A \rightarrow \varepsilon$", where A is not the start variable.
- Then for each occurrence of A on the right-hand side of a rule we add a new rule with that occurrence deleted.
- Accordingly, each rule "$R \rightarrow A$" is replaced by "$R \rightarrow \varepsilon$" unless it has been already removed.

Chomsky Normal Form

$$\begin{aligned} \boldsymbol{S_0} &\rightarrow \boldsymbol{S} \\ S &\rightarrow ASA \mid \mathtt{a}B \\ A &\rightarrow B \mid S \\ B &\rightarrow \mathtt{b} \mid \varepsilon \end{aligned}$$

2. Remove ε-rules $B \rightarrow \varepsilon$, shown on the left, and $A \rightarrow \varepsilon$, shown on the right.

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid \mathtt{a}B \mid \mathbf{a} \\ A &\rightarrow B \mid S \mid \boldsymbol{\varepsilon} \\ B &\rightarrow \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\ A &\rightarrow B \mid S \mid \varepsilon \\ B &\rightarrow \mathtt{b} \end{aligned}$$

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

- Third, we handle all unit rules by removing each unit rule $A \to B$.
- In consequence whenever $B \to u$ appears, we add the rule $A \to u$ unless this is a unit rule previously removed.

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\
A &\to B \mid S \\
B &\to \mathtt{b}
\end{aligned}
$$

Chomsky Normal Form

3a. Remove unit rules $S \to S$, shown on the left, and $S_0 \to S$, shown on the right.

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S \\
A &\to B \mid S \\
B &\to \mathtt{b}
\end{aligned}
\qquad
\begin{aligned}
S_0 &\to S \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to B \mid S \\
B &\to \mathtt{b}
\end{aligned}
$$

3b. Remove unit rules $A \to B$ and $A \to S$.

$$
\begin{aligned}
S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to B \mid S \mid \boldsymbol{\mathtt{b}} \\
B &\to \mathtt{b}
\end{aligned}
\qquad
\begin{aligned}
S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \boldsymbol{\mathtt{a}} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
B &\to \mathtt{b}
\end{aligned}
$$

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

- Finally, we convert all remaining rules as follows: $A \rightarrow u_1u_2...u_k$ for $k>2$, where each u_i is a variable or terminal with a series of rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$,..., $A_{k-2} \rightarrow u_{k-1}u_k$ where each A_i is a new variable.
- When $k=2$, and $A \rightarrow u_1u_2$, we may replace any terminal u_i by a variable U_i and the rule $U_i \rightarrow u_i$.

Chomsky Normal Form

$$\begin{aligned} S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ A &\to \mathtt{b} \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ B &\to \mathtt{b} \end{aligned}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \to \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \to \mathtt{a}$.)

$$\begin{aligned} S_0 &\to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ S &\to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ A &\to \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\ A_1 &\to SA \\ U &\to \mathtt{a} \\ B &\to \mathtt{b} \end{aligned}$$

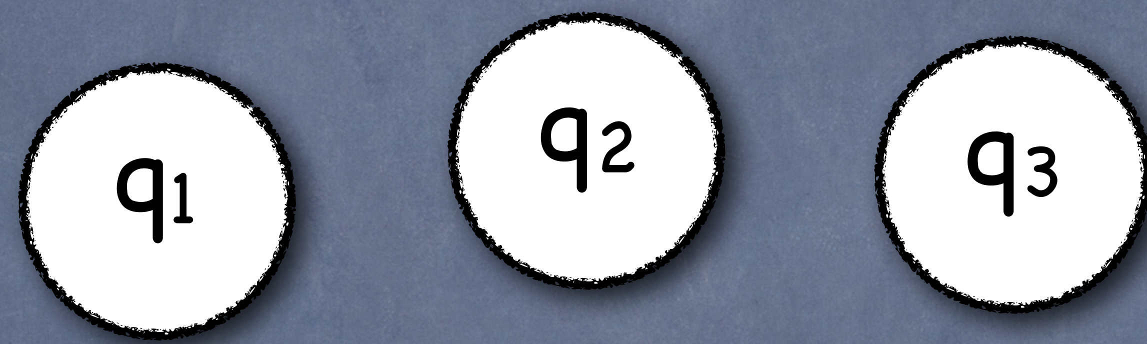
Chomsky Normal Form

$$
\begin{aligned}
S &\rightarrow ASA \mid \mathtt{a}B \\
A &\rightarrow B \mid S \\
B &\rightarrow \mathtt{b} \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}
$$

Definition of PDA

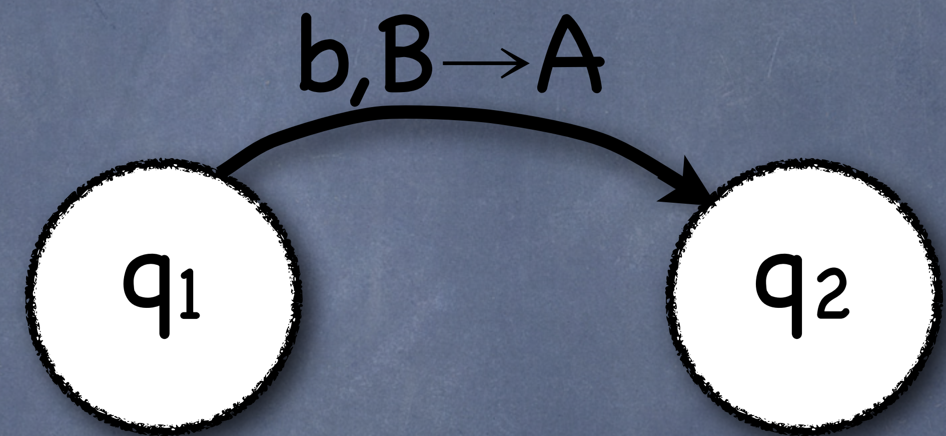
- States



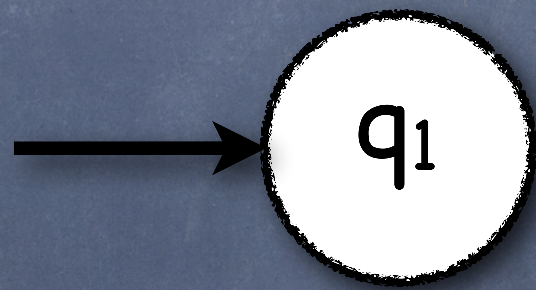
- Alphabets

input: a,b,c,d
STACK: A,B,C,D

- Transition function



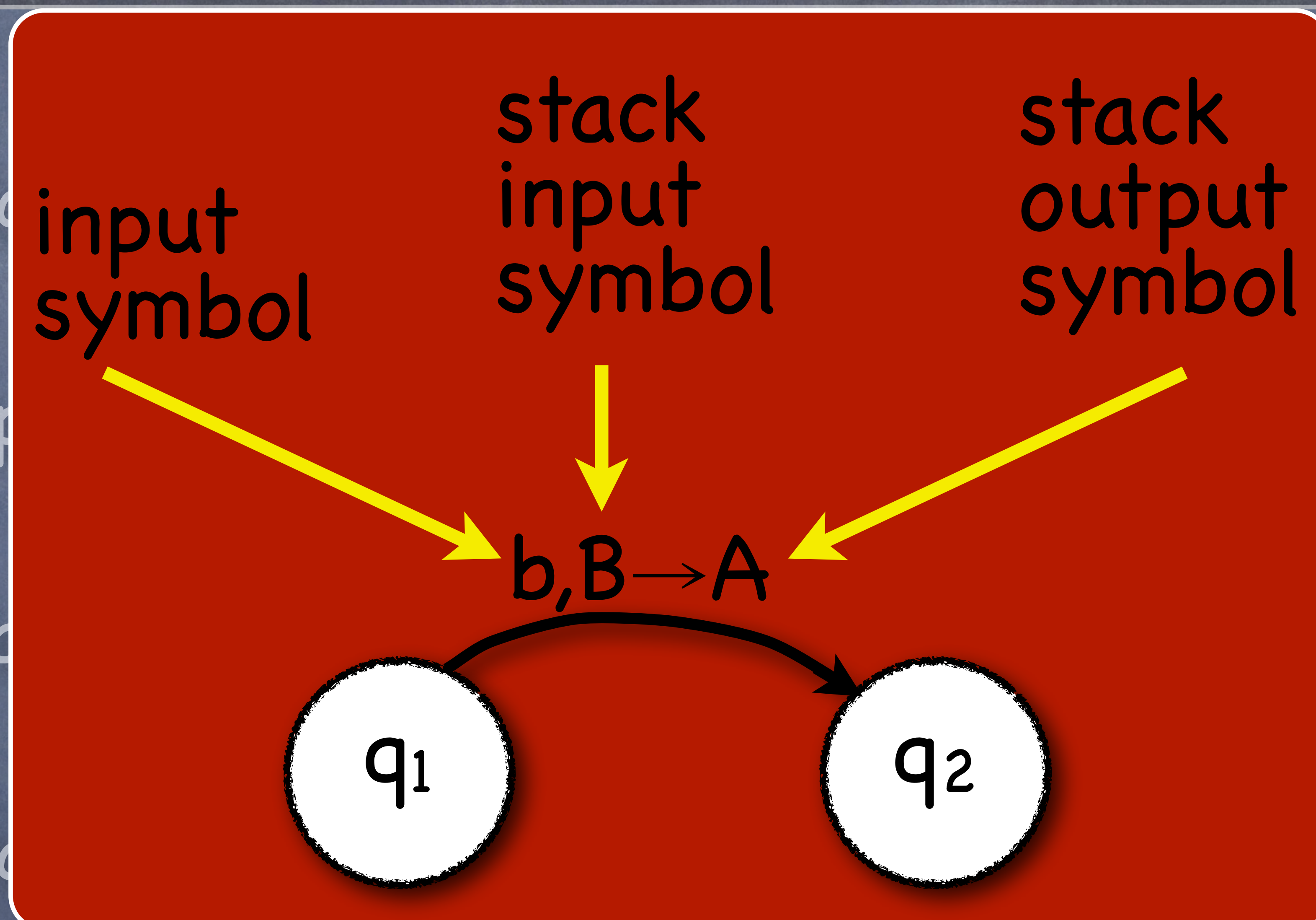
- Start state

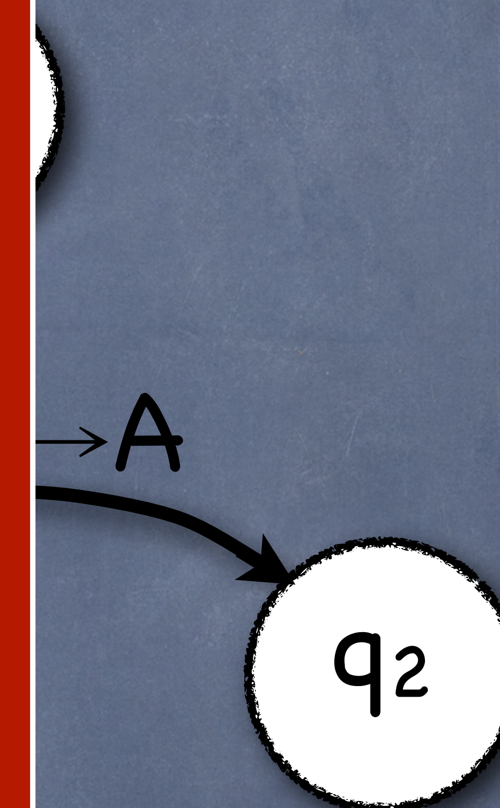


- Accept states

Definition of PDA

- St
- Al
- Tr
- St
- Accept states





Definition of PDA

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Definition of PDA

- Let $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ be a pushdown automaton and let $w=w_1w_2...w_n$ $(n\geq 0)$ be a string where each symbol $w_i \in \Sigma$.

- M **accepts** w if $\exists\, m \geq n$, $\exists\, r_0,r_1,\dots,r_m \in Q$, $\exists\, s_0,s_1,\dots,s_m \in \Gamma^*$ and $\exists\, y_1y_2...y_m = w$, with $y_i \in \Sigma_\varepsilon$ s.t.

  1. $r_0 = q_0$, $s_0 = \varepsilon$
  2. $r_{i+1},b \in \delta(r_i,y_{i+1},a)$ for $i = 0...m-1$, $s_i=at$, $s_{i+1}=bt$
  3. $r_m \in F$ for some $t \in \Gamma^*$, $a,b \in \Gamma_\varepsilon$

EXAMPLE 2.14

The following is the formal description of the PDA (next slide) that recognizes the language $\{0^n1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Examples of PDA

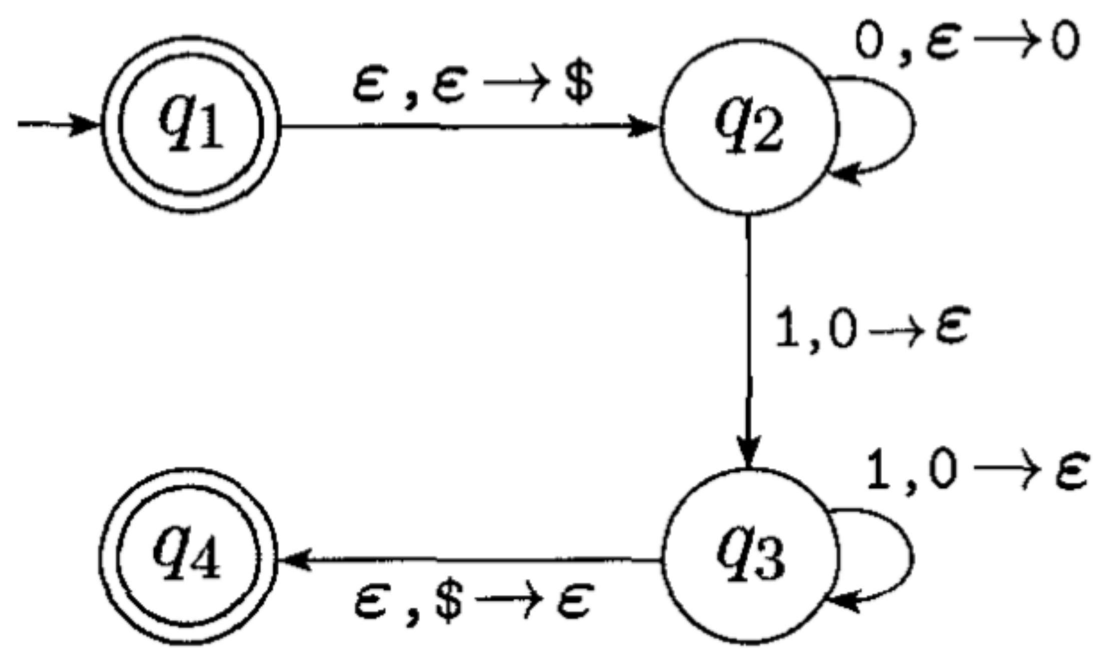


FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

Examples of PDA

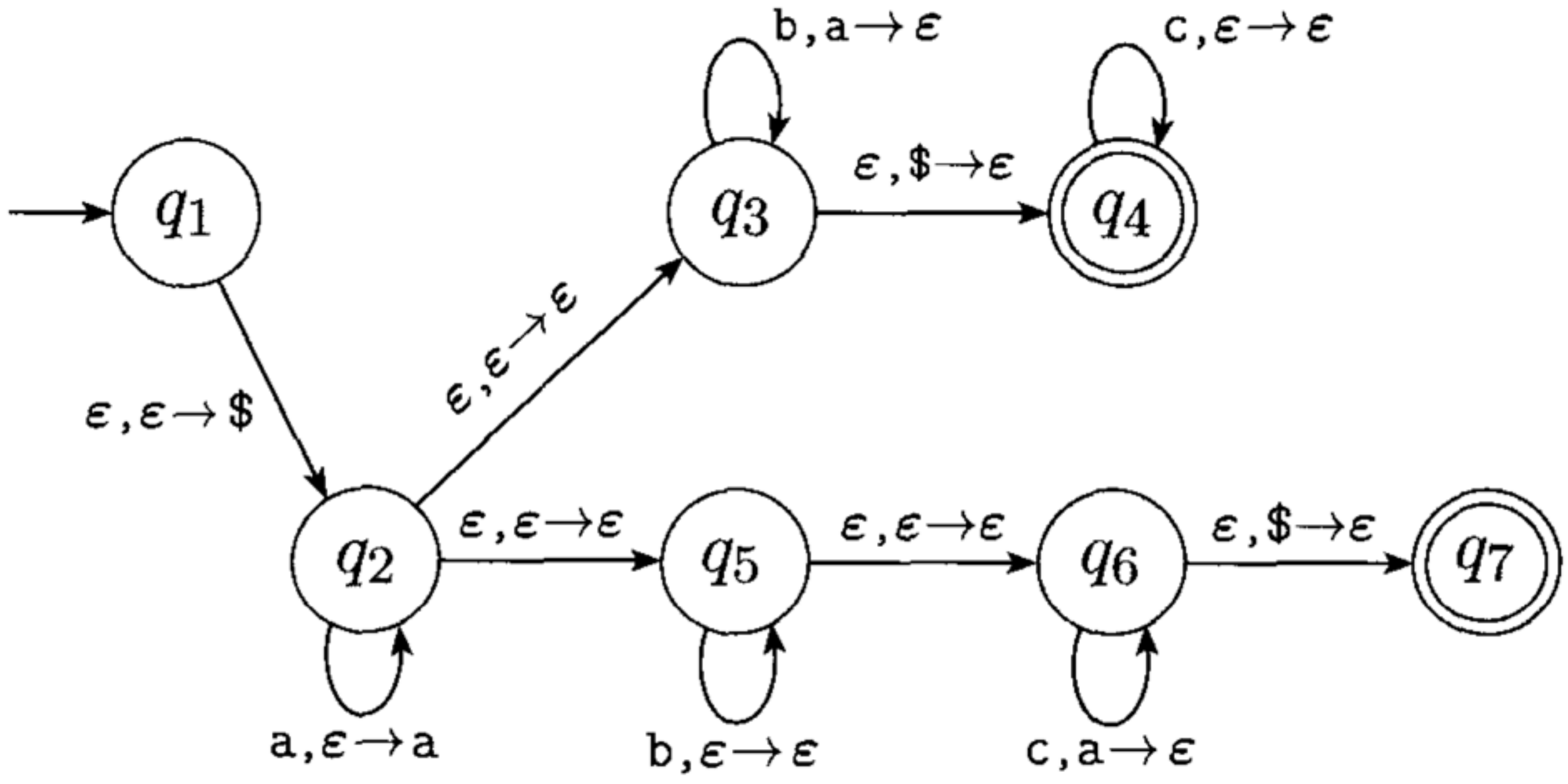


FIGURE 2.17
State diagram for PDA M_2 that recognizes
$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

Examples of PDA

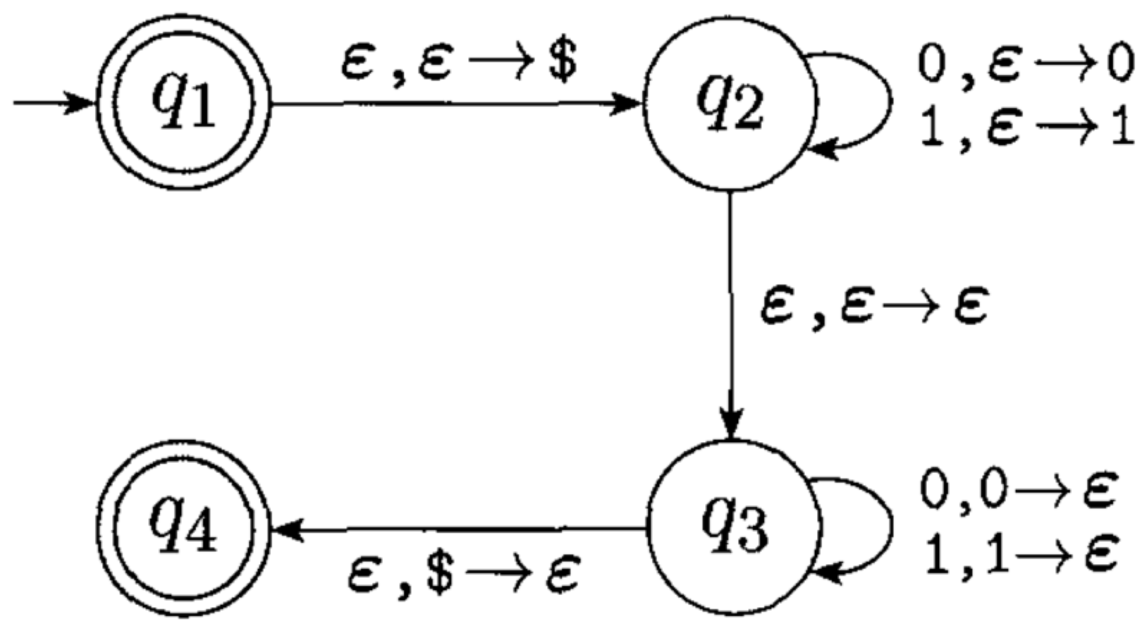


FIGURE 2.19
State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

PDA vs CFG

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

CFG to PDA

LEMMA 2.21 ....

If a language is context free, then some pushdown automaton recognizes it.

The following is an informal description of P.

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   c. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

CFG to PDA

- Proof: Given a CFG $G=(V,\Sigma,R,S)$, we now construct a PDA $P=(Q,\Sigma,\Gamma,\delta,q_0,F)$ for it.
- We define a special notation to write an entire string on the stack in one step.
- We can simulate this action by adding extra states to write the string one symbol at a time.

CFG to PDA

- Let q and r be states of the PDA and let $a \in \Sigma_\varepsilon$ $s \in \Gamma_\varepsilon$.
- Starting in state q, say we want to read a from the input and pop s from the stack. Moreover we want to push string $u = u_1 \ldots u_\ell$ back onto the stack at the same time and end in state r.

CFG to PDA

- We implement this action $(a,s \to u_1 \ldots u_\ell)$ by introducing new states $q_1,\ldots,q_{\ell-1}$ and setting the transition function as follows:

$\delta(q,a,s) \ni (q_1,u_\ell)$,

$\delta(q_1,\varepsilon,\varepsilon) = \{(q_2,u_{\ell-1})\}$,

$\delta(q_2,\varepsilon,\varepsilon) = \{(q_3,u_{\ell-2})\}$,

…

$\delta(q_{\ell-1},\varepsilon,\varepsilon) = \{(r,u_1)\}$.

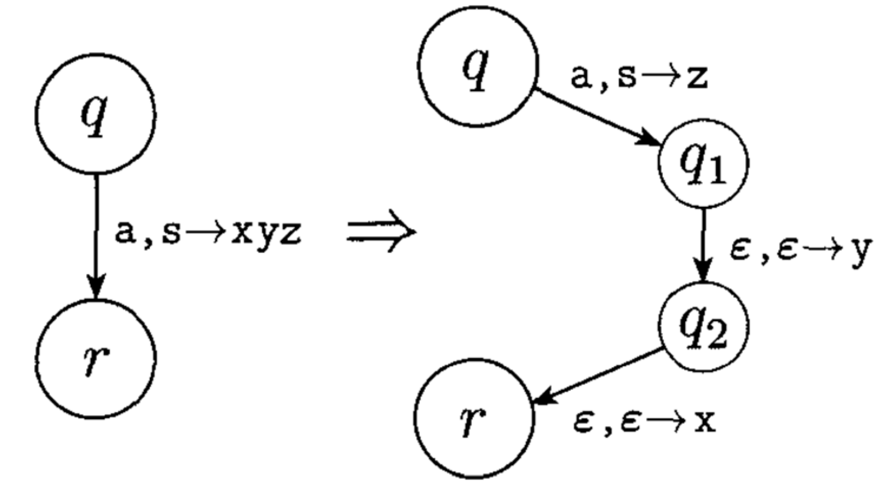


FIGURE 2.23
Implementing the shorthand $(r,xyz) \in \delta(q,a,s)$

CFG to PDA

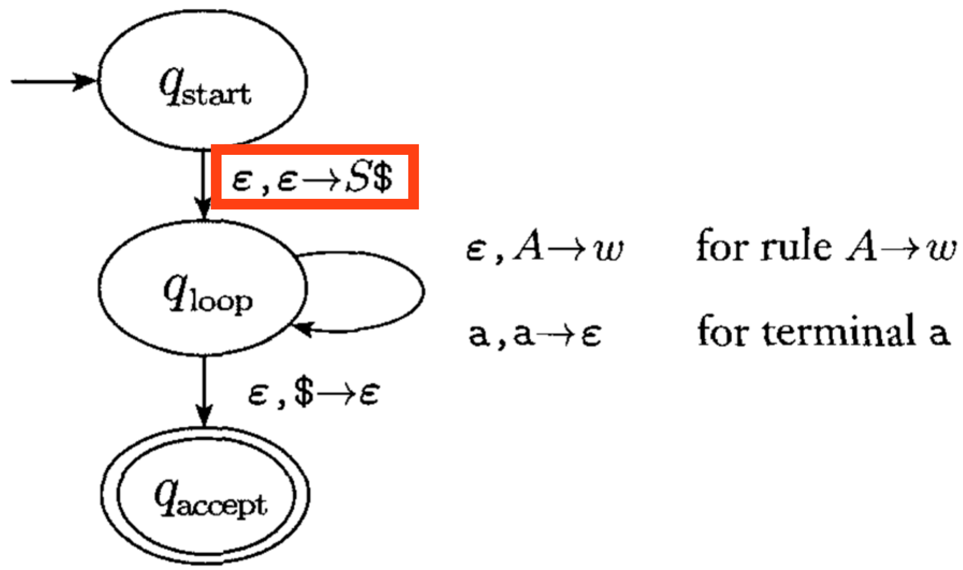


FIGURE 2.24
State diagram of P

The states of P are $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initializing the stack to contain the symbols \$ and S, implementing step 1 in the informal description: $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$. Then we put in transitions for the main loop of step 2.

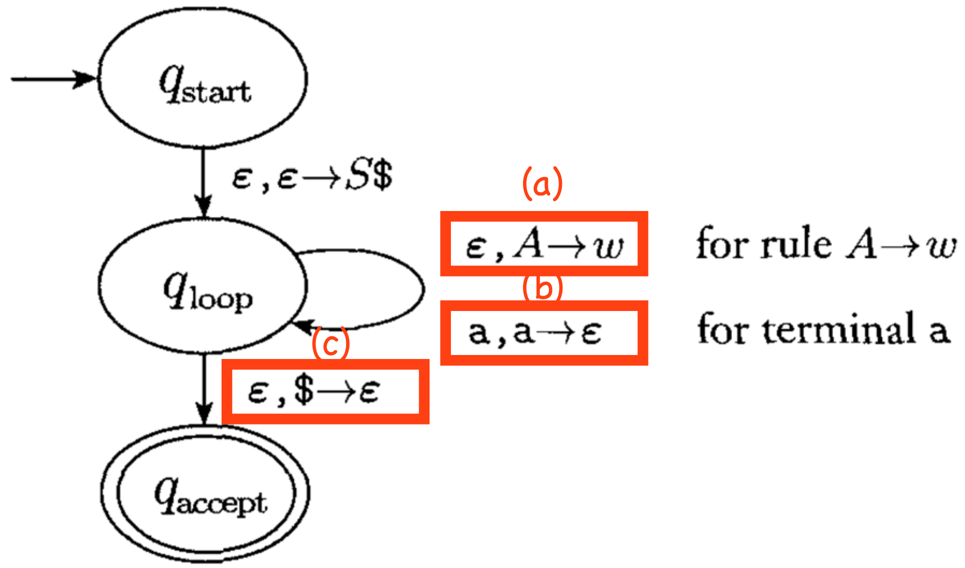


FIGURE 2.24
State diagram of P

CFG to PDA

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w) |$ where $A \to w$ is a rule in $R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker \$ is on the top of the stack. Let $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$.

CFG to PDA

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \rightarrow T\mathtt{a} \mid \varepsilon$$

PDA to CFG

LEMMA 2.27 ....

If a pushdown automaton recognizes some language, then it is context free.

PDA to CFG

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

- Giving *P* features 1 and 2 is easy.
- To give it feature 3, we replace
each transition that simultaneously pops and pushes with a two-transition sequence that goes through a new state,
each transition that neither pops nor pushes with a two-transition sequence that pushes then pops an arbitrary stack symbol.

PDA to CFG

PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G. The variables of G are $\{A_{pq} | \, p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$ in G.
- For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.
- Finally, for each $p \in Q$, put the rule $A_{pp} \to \varepsilon$ in G.

You may gain some insight for this construction from the following figures.

- For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.

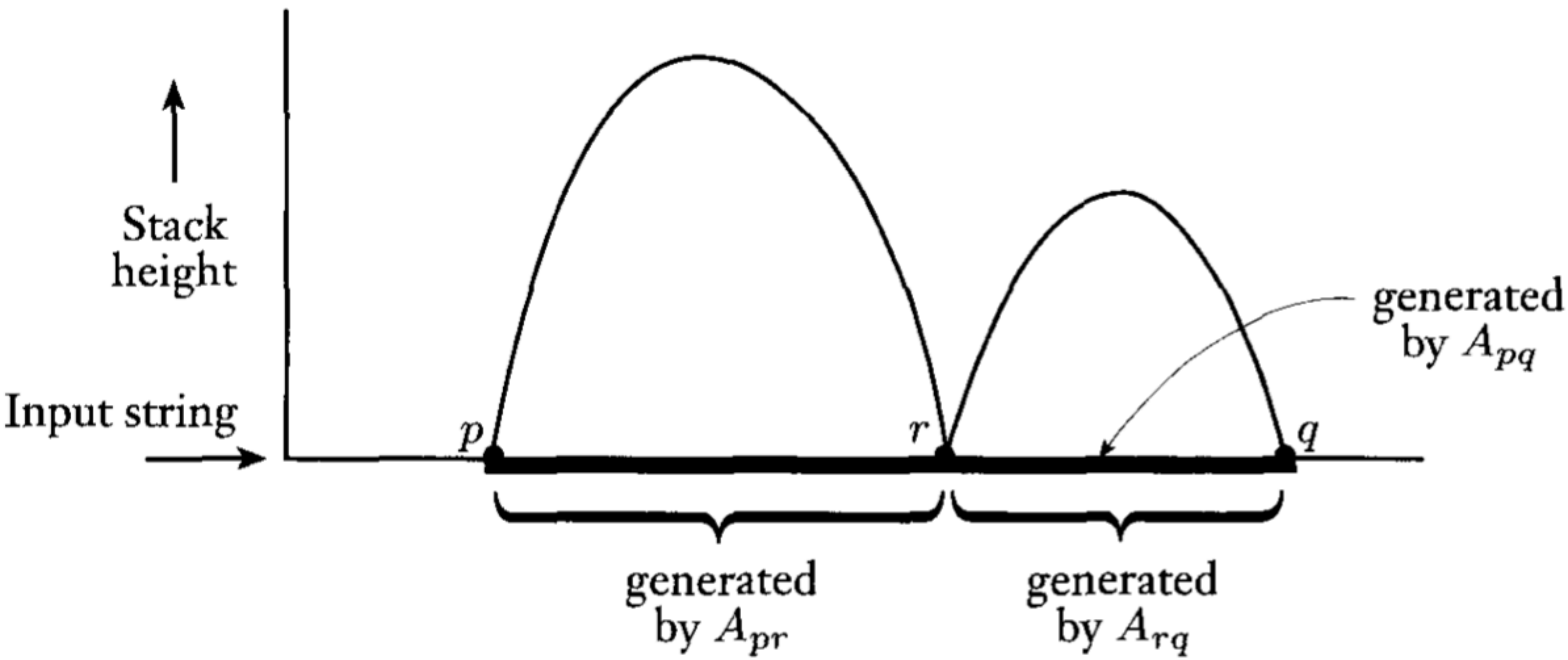


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \to A_{pr}A_{rq}$

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$ in G.

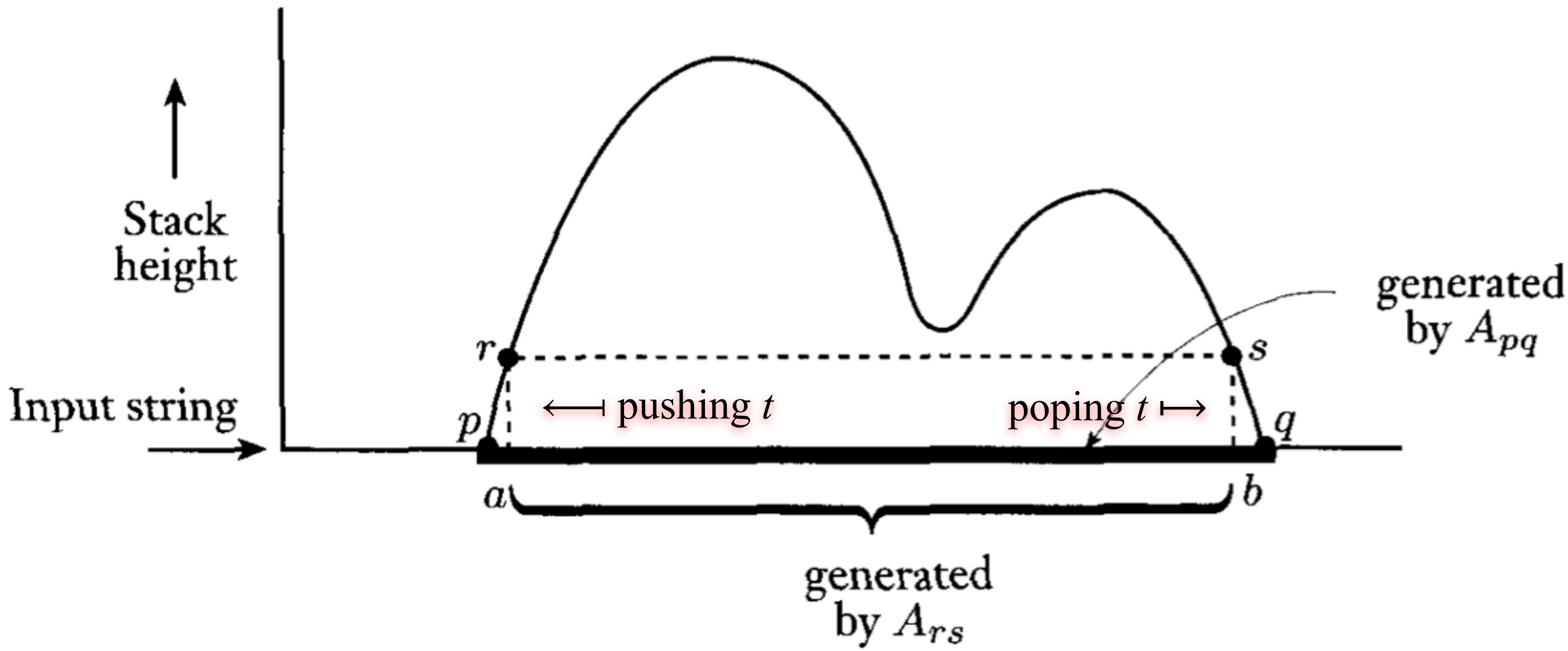


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \to aA_{rs}b$

PDA to CFG

PROOF Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G. The variables of G are $\{A_{pq} | \; p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G's rules.

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$ in G.
- For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.
- Finally, for each $p \in Q$, put the rule $A_{pp} \to \varepsilon$ in G.

PDA to CFG

CLAIM 2.30

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

We prove this claim by induction on the number of steps in the derivation of x from A_{pq}.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

Basis: The derivation has 1 step.
A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \rightarrow \varepsilon$. Clearly, input ε takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k+1$.

Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k+1$ steps. The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. We handle these two cases separately.

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

$$A_{pq} \Rightarrow aA_{rs}b$$

In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \overset{*}{\Rightarrow} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack symbol t. Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

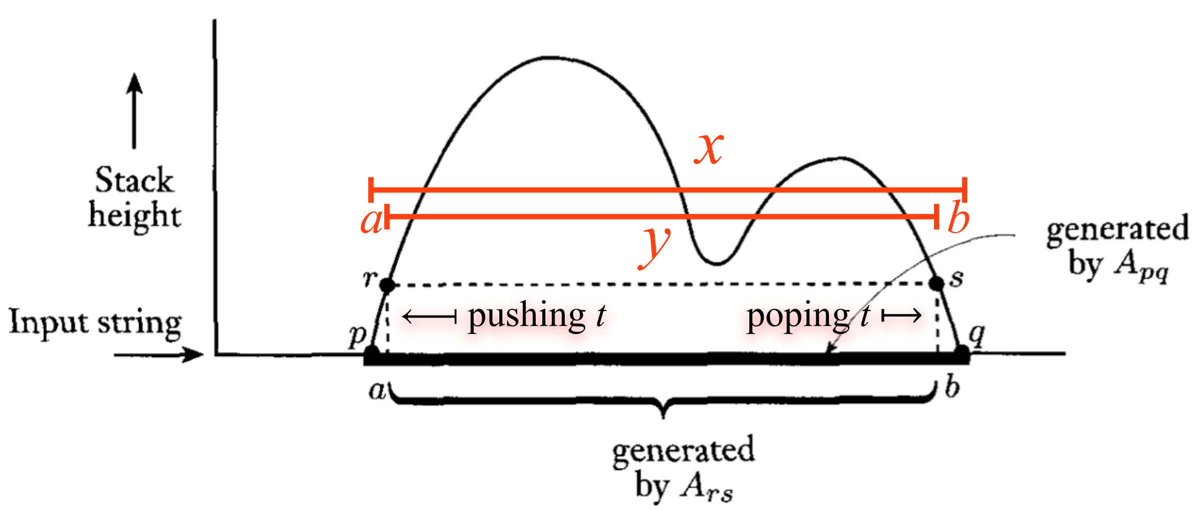


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

If A_{pq} generates x, then x can bring P from a state p (with an empty stack) to a state q (with an empty stack).

$A_{pq} \Rightarrow A_{pr} A_{rq}$

In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \overset{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r, and z can bring P from r to q, with empty stacks at the beginning and end. Hence x can bring it from p with empty stack to q with empty stack. This completes the induction step.

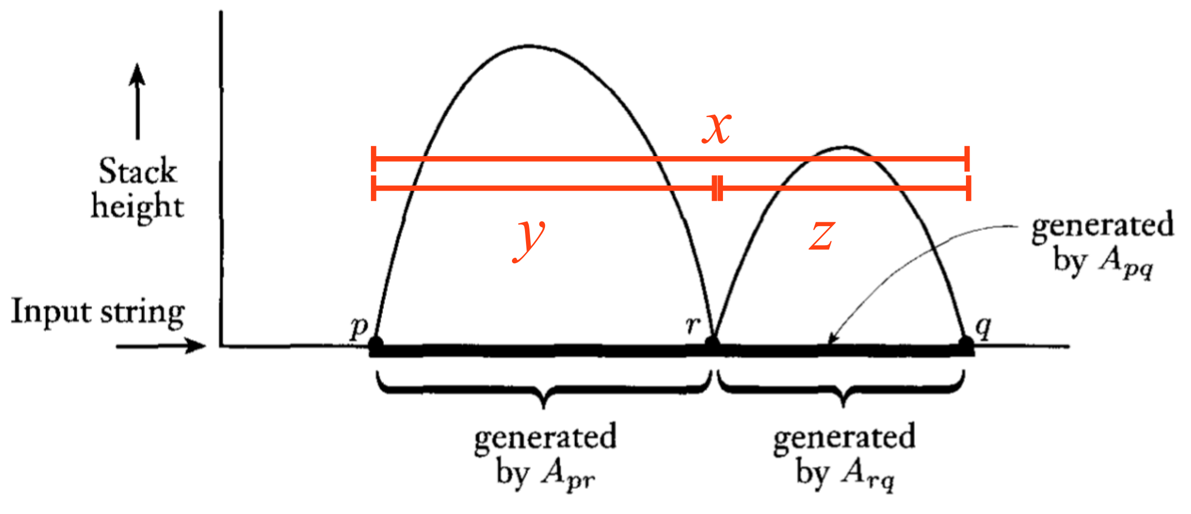


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr} A_{rq}$

PDA to CFG

CLAIM 2.31

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

We prove this claim by induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

Basis: The computation has 0 steps.
If a computation has 0 steps, it starts and ends at the same state—say, p. So we must show that $A_{pp} \overset{*}{\Rightarrow} x$. In 0 steps, P only has time to read the empty string, so $x = \varepsilon$. By construction, G has the rule $A_{pp} \rightarrow \varepsilon$, so the basis is proved.

Induction step: Assume true for computations of length at most k, where $k \geq 0$, and prove true for computations of length $k + 1$.

Suppose that P has a computation wherein x brings p to q with empty stacks in $k + 1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

the stack is empty only at the beginning and end

In the first case, the symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol t. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move, and s be the state before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), and so rule $A_{pq} \to aA_{rs}b$ is in G.

Let y be the portion of x without a and b, so $x = ayb$. Input y can bring

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$ in G.

x so the computation on y has $(k + 1) - 2 = k - 1$ steps. Thus the induction hypothesis tells us that $A_{rs} \overset{*}{\Rightarrow} y$. Hence $A_{pq} \overset{*}{\Rightarrow} x$.

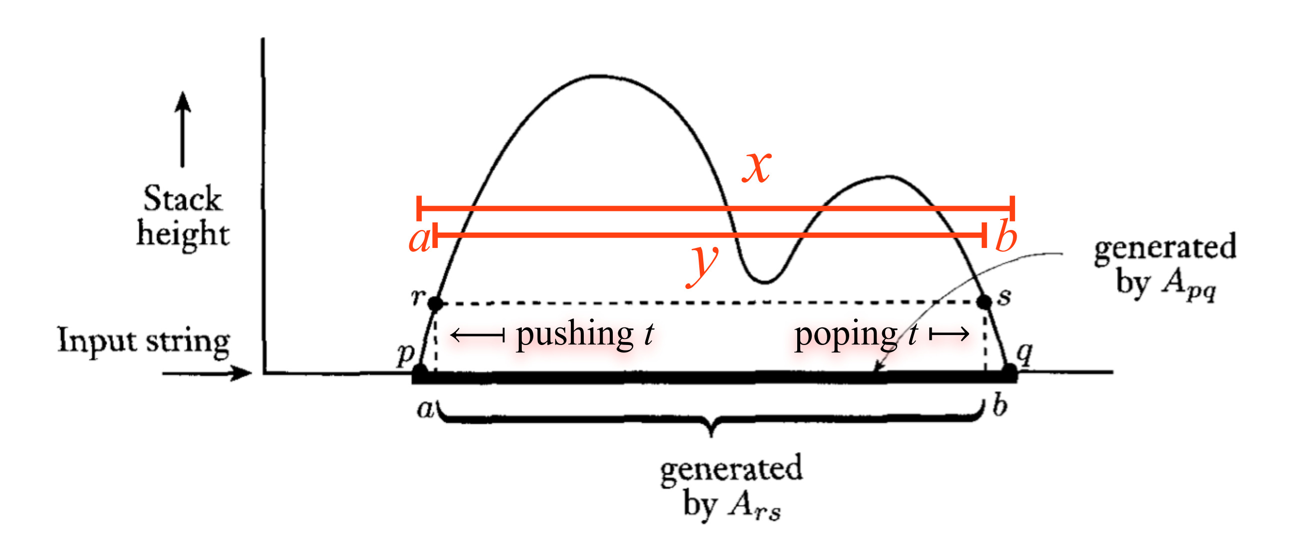


FIGURE 2.29
PDA computation corresponding to the rule $A_{pq} \to aA_{rs}b$

If x can bring P from a state p (with an empty stack) to a state q (with an empty stack), then A_{pq} generates x.

it becomes empty elsewhere, too.

In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x. Then the portions of the computation from p to r and from r to q each contain at most k steps. Say that y is the input read during the first portion and z is the input read during the second portion. The induction hypothesis tells us that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Because rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G, $A_{pq} \overset{*}{\Rightarrow} x$, and the proof is complete.

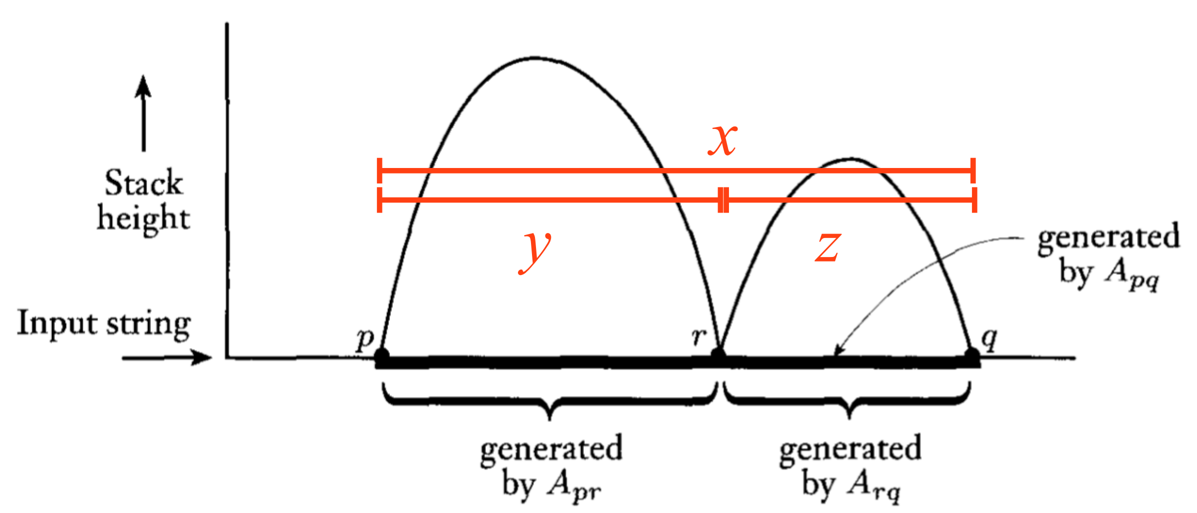


FIGURE 2.28
PDA computation corresponding to the rule $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA vs CFG

LEMMA 2.21

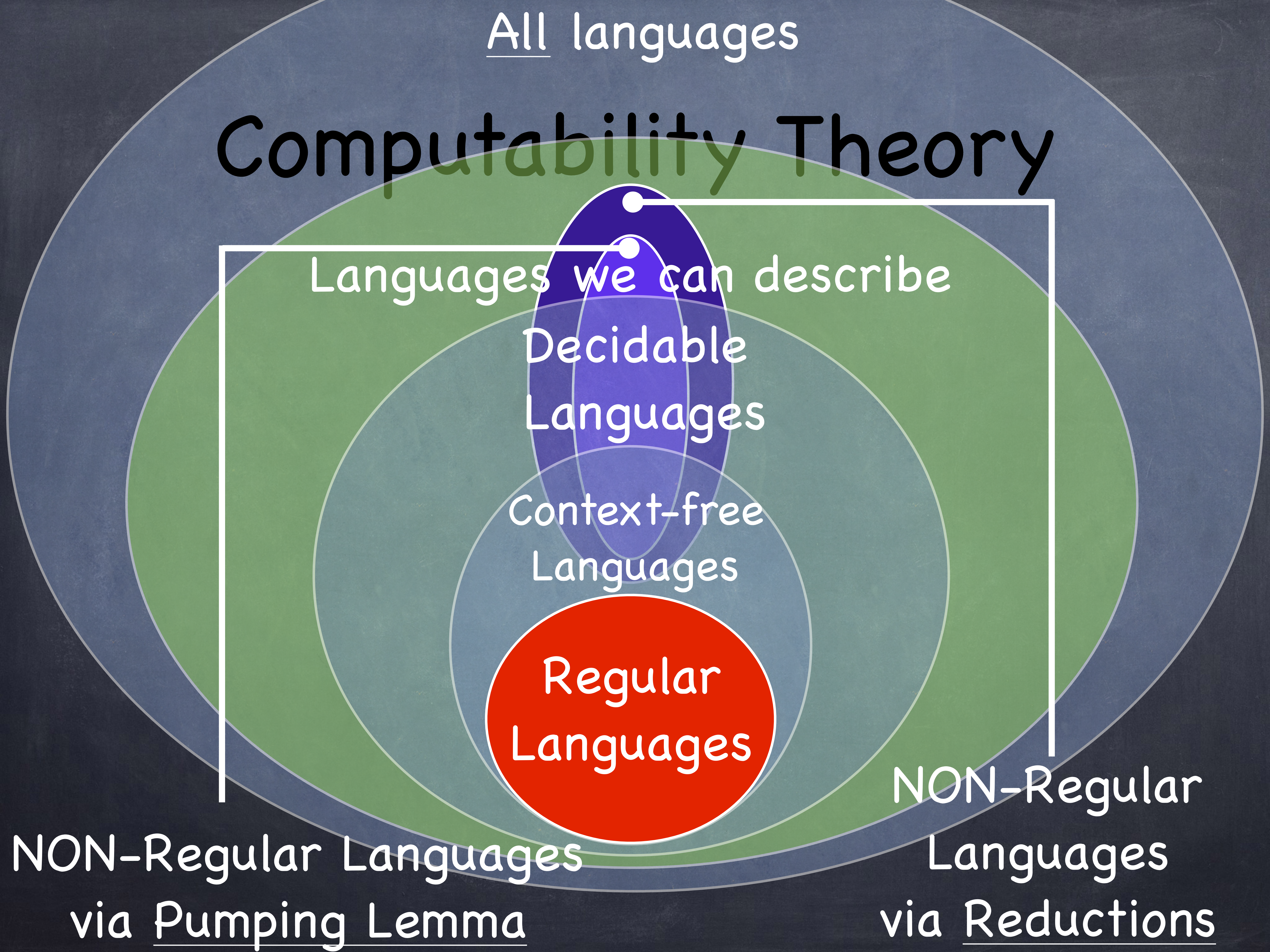
If a language is context free, then some pushdown automaton recognizes it.

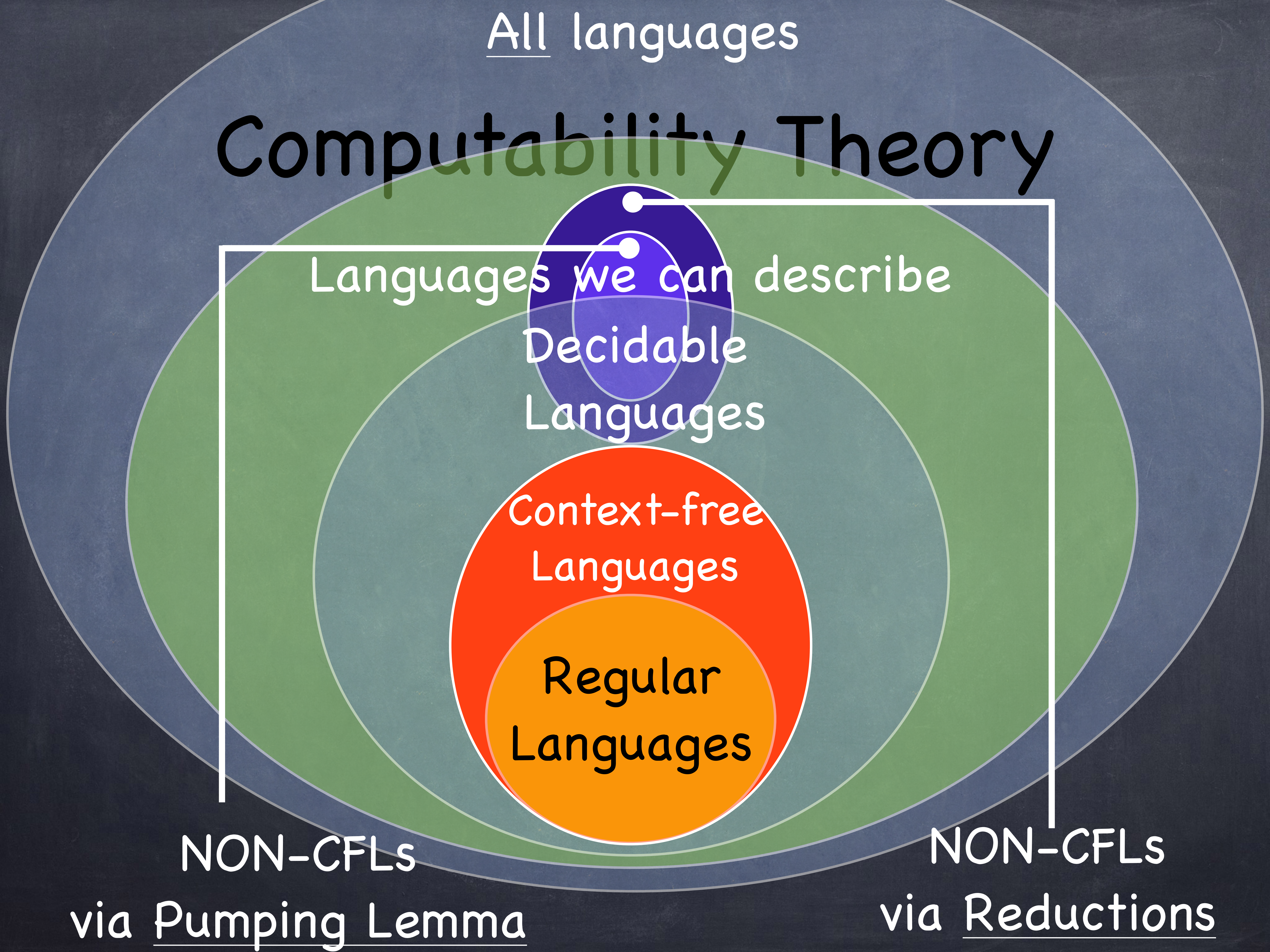
LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.


All languages
Computability Theory
Languages we can describe
Decidable Languages
Context-free Languages
Regular Languages
NON-Regular Languages via Pumping Lemma
NON-Regular Languages via Reductions


All languages
Computability Theory
Languages we can describe
Decidable Languages
Context-free Languages
Regular Languages
NON-CFLs via Pumping Lemma
NON-CFLs via Reductions

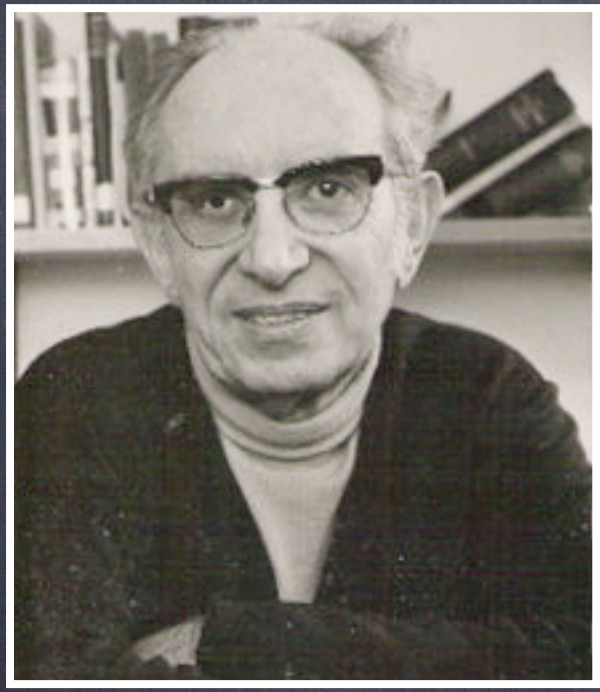
Pumping Lemma for CFLs

THEOREM 2.34

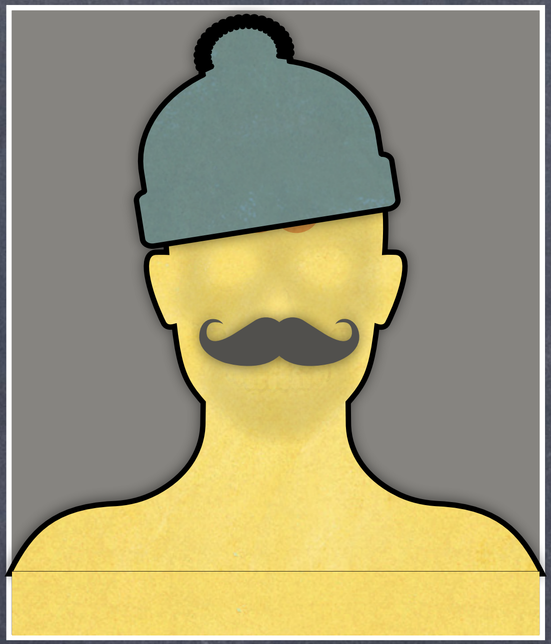
Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Pumping Lemma for CFLs



Yehoshua Bar-Hillel



Micha A. Perles

Eli Shamir

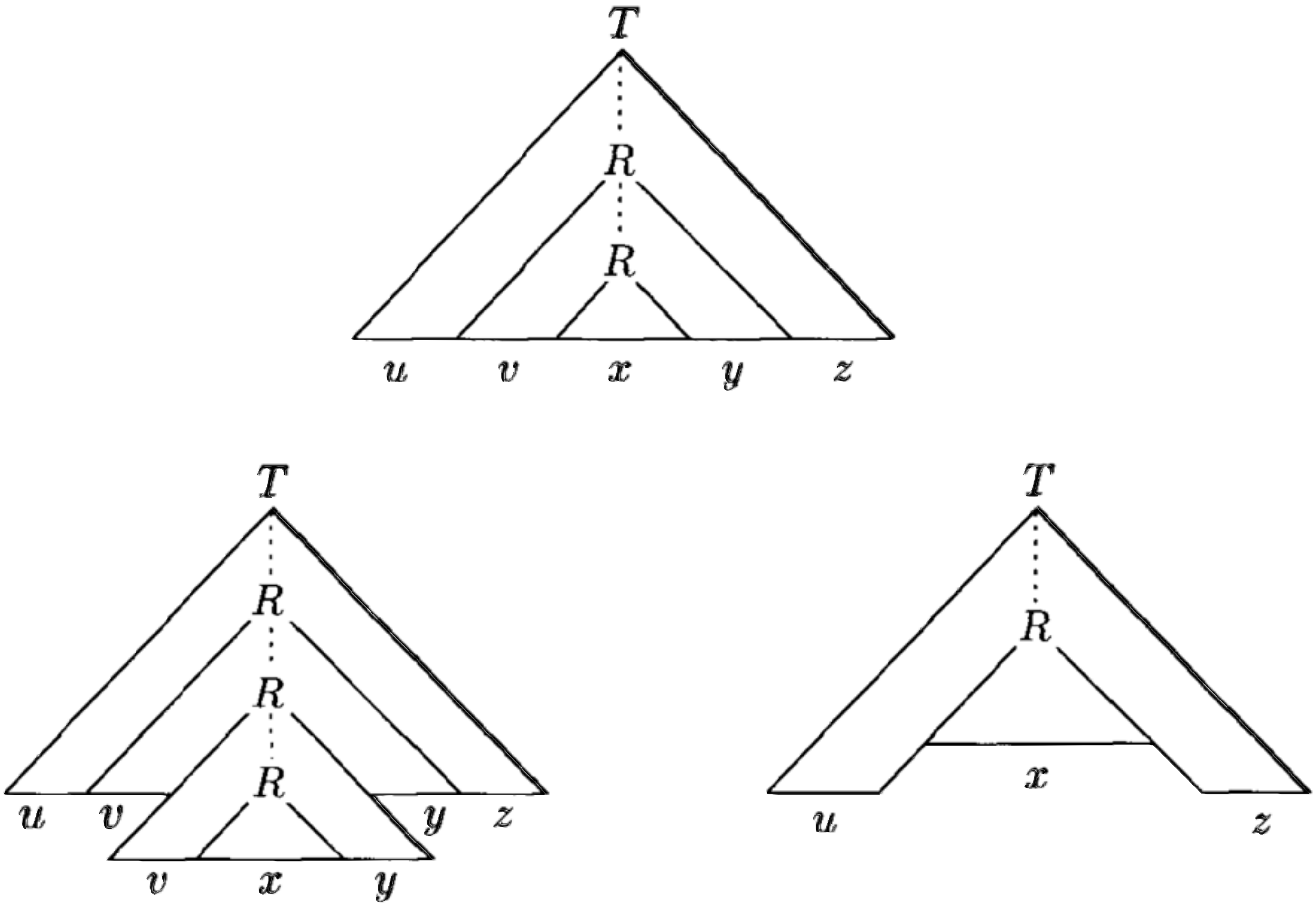


FIGURE 2.35
Surgery on parse trees

THEOREM 2.34

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

$A \in \mathbb{CFL} \implies$

$\exists p \forall s \in A, |s| \geq p, \exists uvxyz = s$ st 1,2,3=true.

$\forall p \exists s \in A, |s| \geq p, \forall uvxyz = s$ st 2,3=true [1=false].

$\implies A \notin \mathbb{CFL}$

$\forall p \exists s \in A, |s| \geq p, \forall uvxyz = s$ s.t. $|vy|>0, |vxy| \leq p$,

then $\exists i \geq 0$ s.t. $s' = uv^ixy^iz \notin A$.

$\implies A \notin \mathbb{CFL}$

Reductions
(& Construction tools)

CFLs are closed under union, concatenation and star. If there exists a CFL C s. t. either $A^*=A'$, $A \cup C=A'$, $A \circ C=A'$

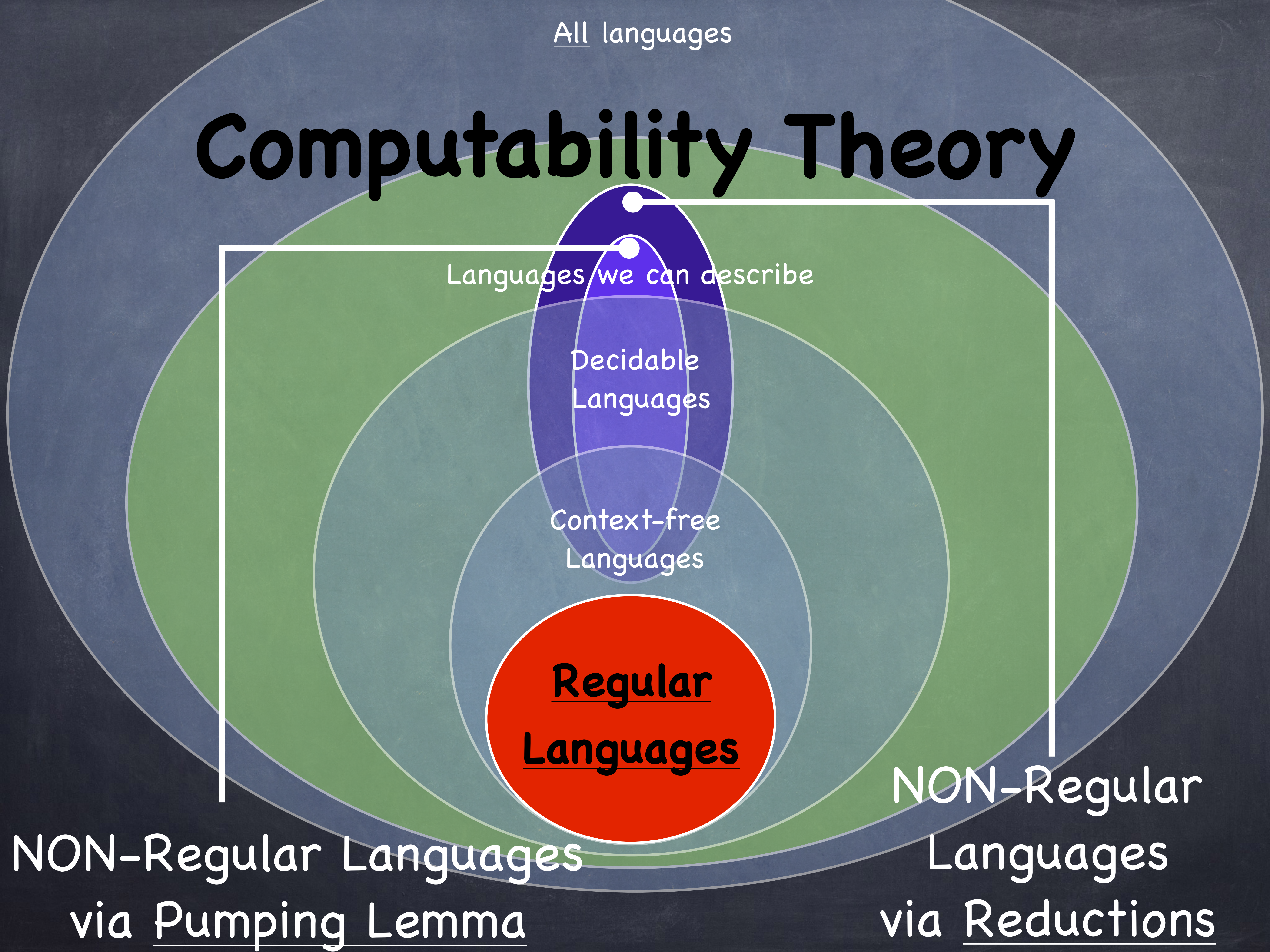
(**<u>but neither complement nor intersection</u>**)
or any combinations of these operations then A′ is a CFL as long as A is.

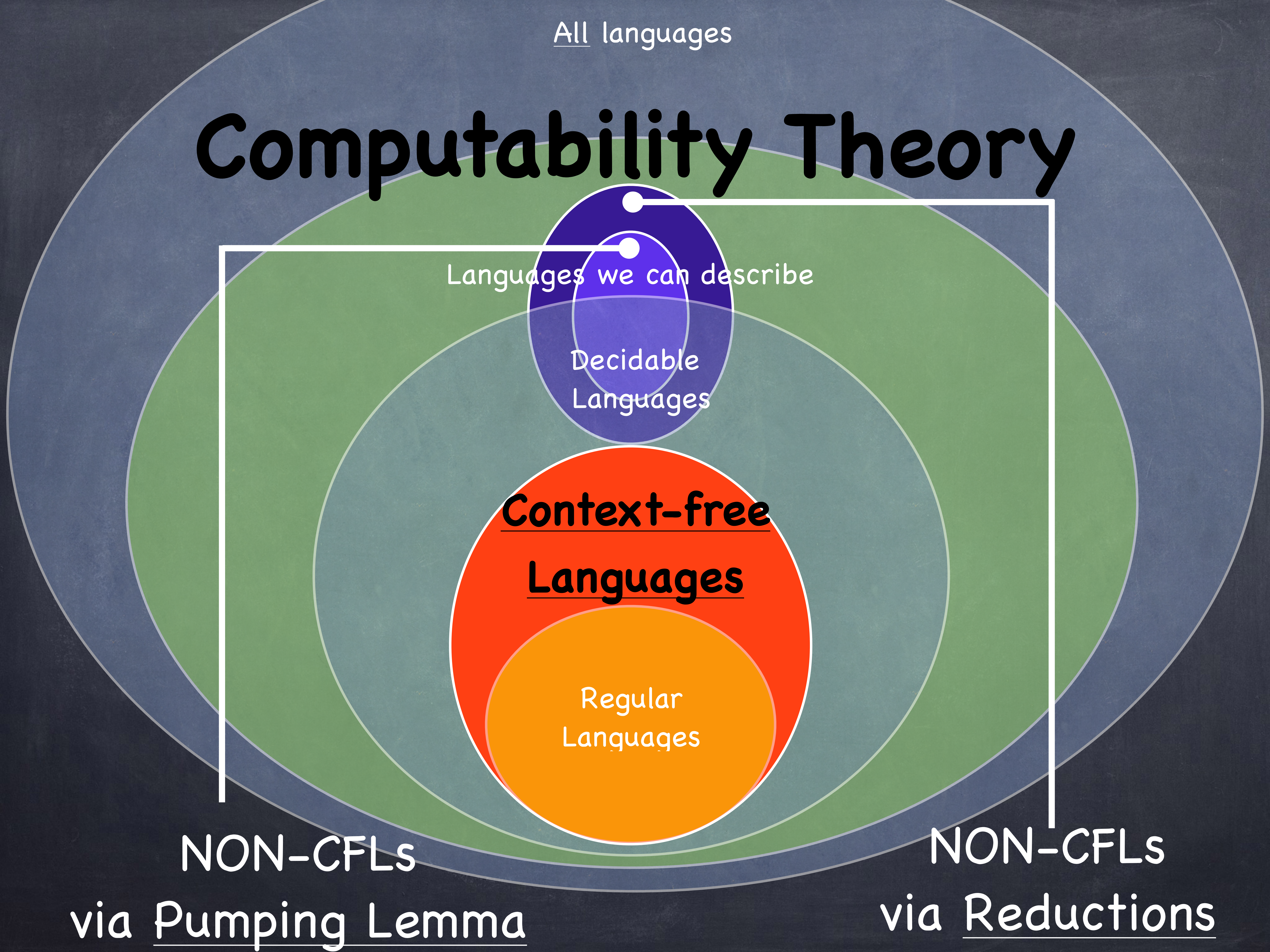
(If A′ is NON-CFL then so is A.)

*** Reduction example ***

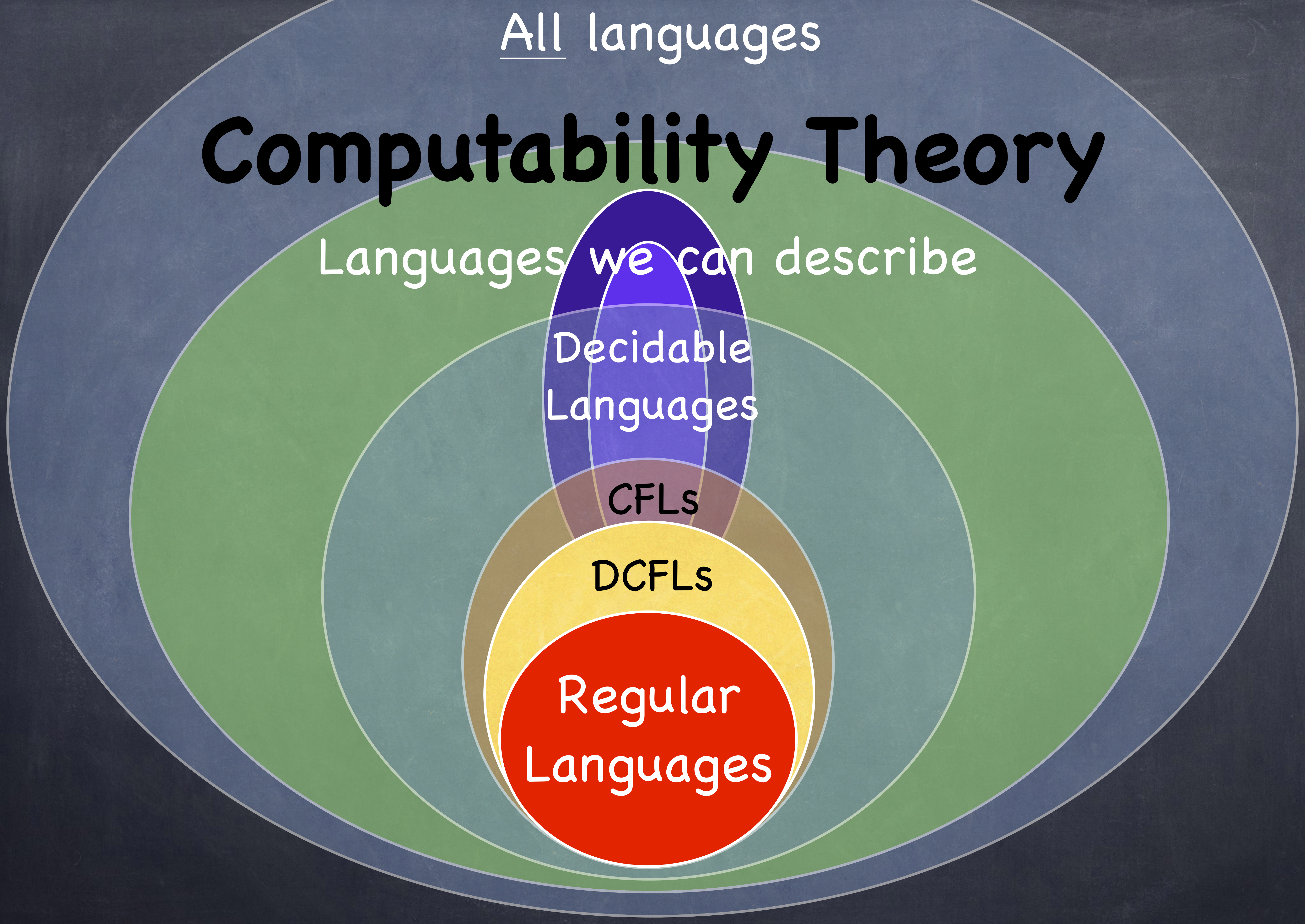
[A]**2.18** **a.** Let C be a context-free language and R be a regular language. Prove that the language $C \cap R$ is context free.

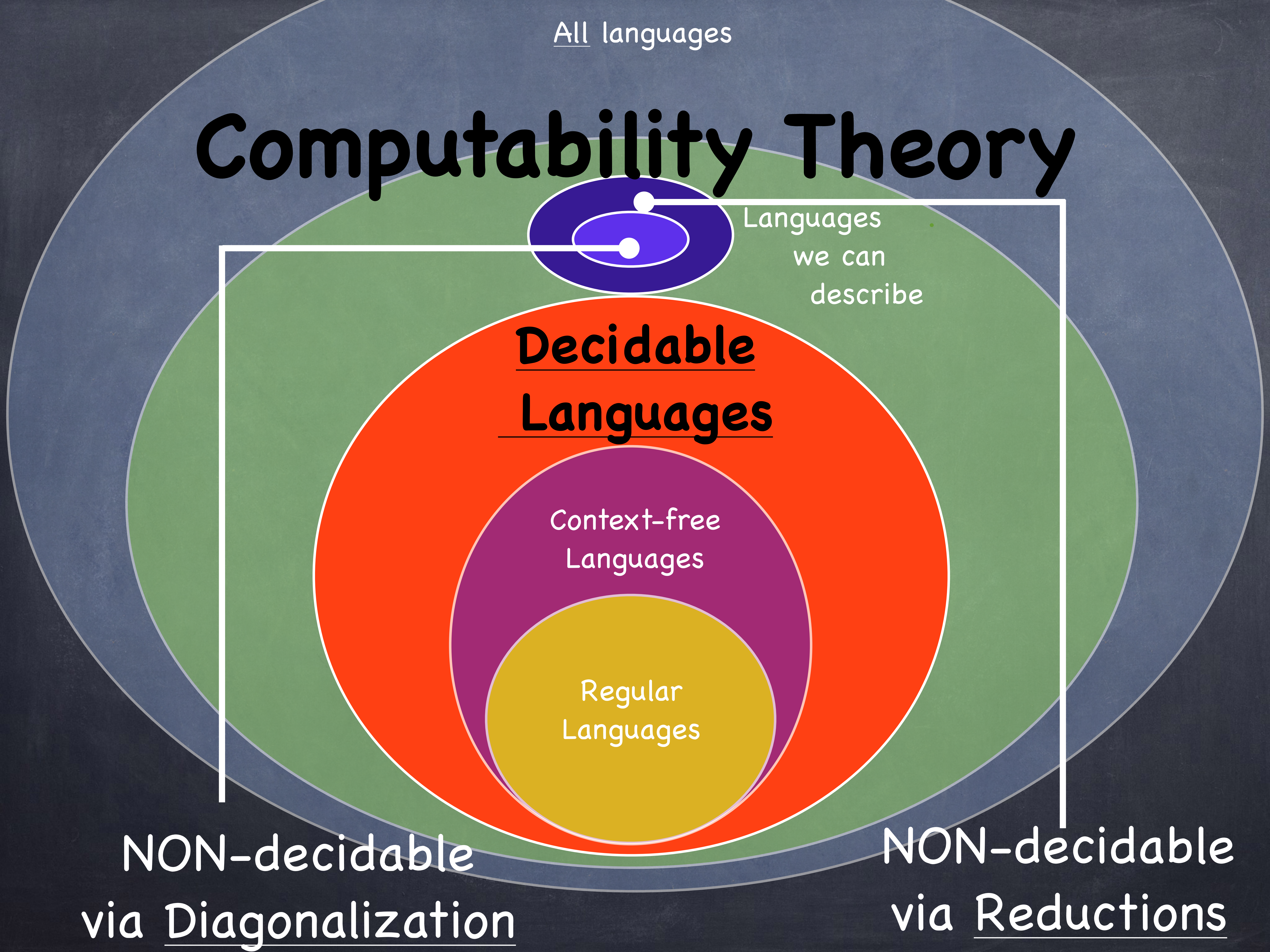
2.18 **(a)** Let C be a context-free language and R be a regular language. Let P be the PDA that recognizes C, and D be the DFA that recognizes R. If Q is the set of states of P and Q' is the set of states of D, we construct a PDA P' that recognizes $C \cap R$ with the set of states $Q \times Q'$. P' will do what P does and also keep track of the states of D. It accepts a string w if and only if it stops at a state $q \in F_P \times F_D$, where F_P is the set of accept states of P and F_D is the set of accept states of D. Since $C \cap R$ is recognized by P', it is context free.


All languages
Computability Theory
Languages we can describe
Decidable Languages
Context-free Languages
Regular Languages
NON-Regular Languages via Pumping Lemma
NON-Regular Languages via Reductions


All languages
Computability Theory
Languages we can describe
Decidable Languages
Context-free Languages
Regular Languages
NON-CFLs via Pumping Lemma
NON-CFLs via Reductions

Computability Theory




All languages
Computability Theory
Languages
we can
describe
Decidable
Languages
Context-free
Languages
Regular
Languages
NON-decidable
via Diagonalization
NON-decidable
via Reductions

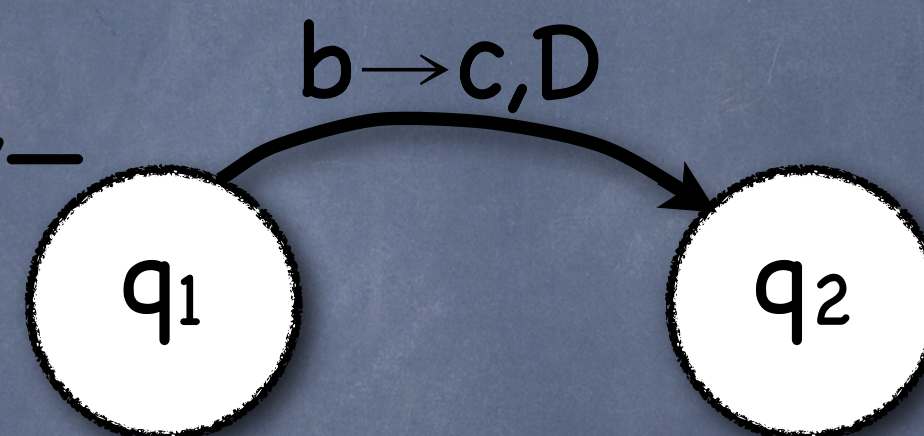
Turing MACHINES

Alan Turing

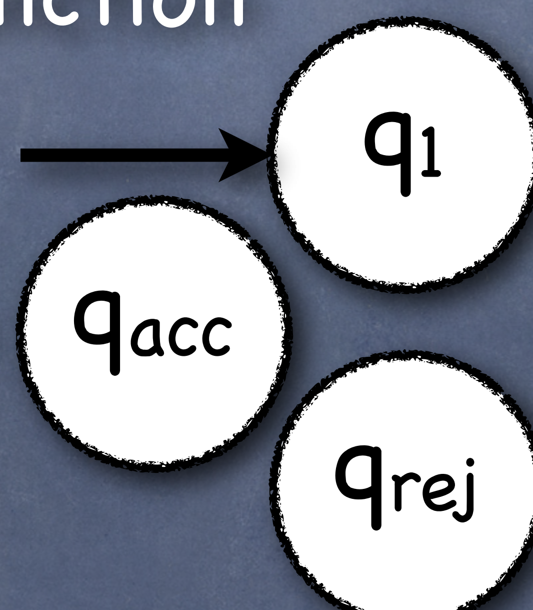
Definition of TM

- States

- Input Alphabet a,b,c
- Tape Alphabet a,b,c,A,B,C,_
- Transition function

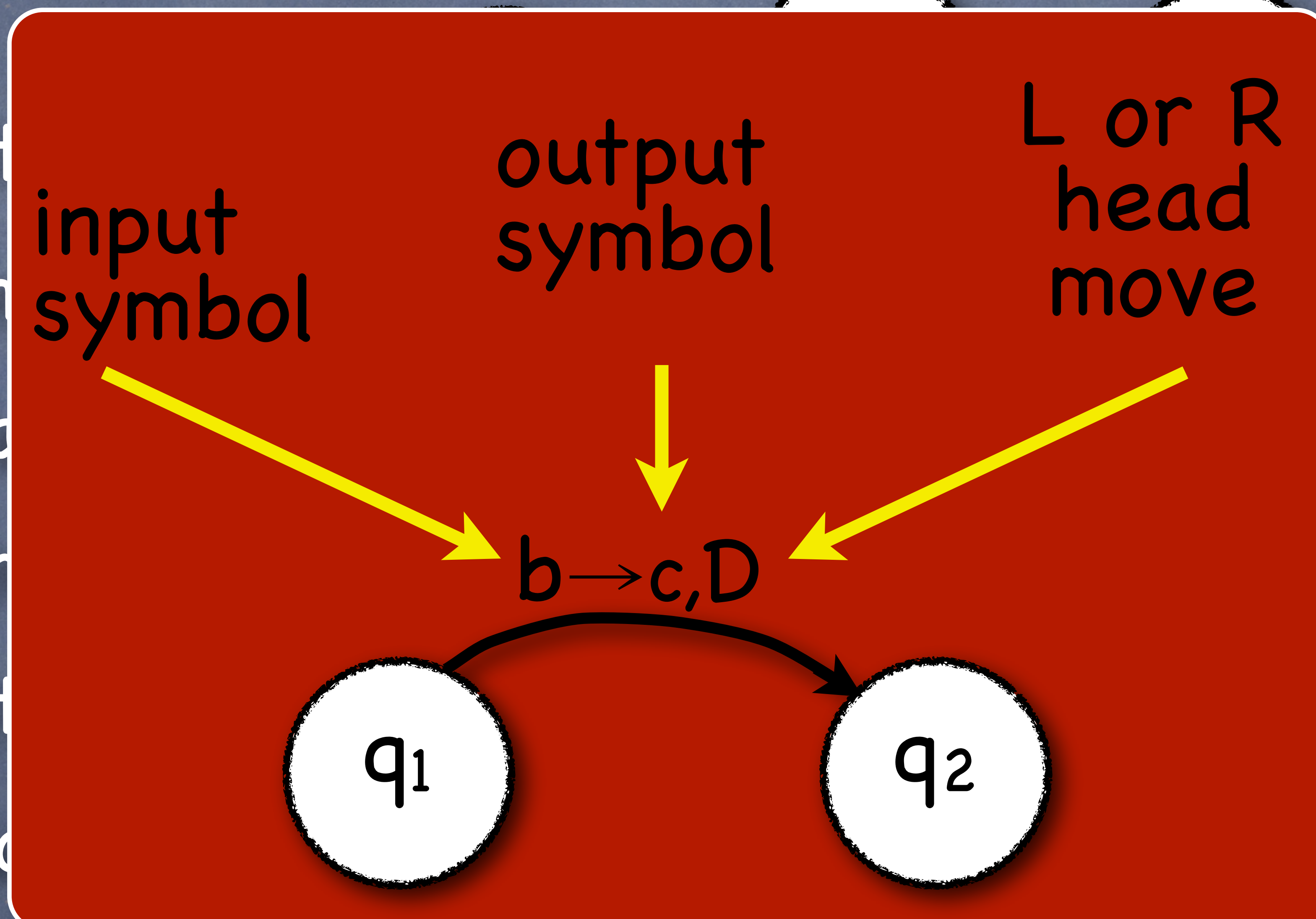


- Start state
- Accept state
- Reject state



Definition of TM

- St
- In
- To
- Tr
- St
- Ac
- Reject state



→c,D

q2

qrej

TM definition

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

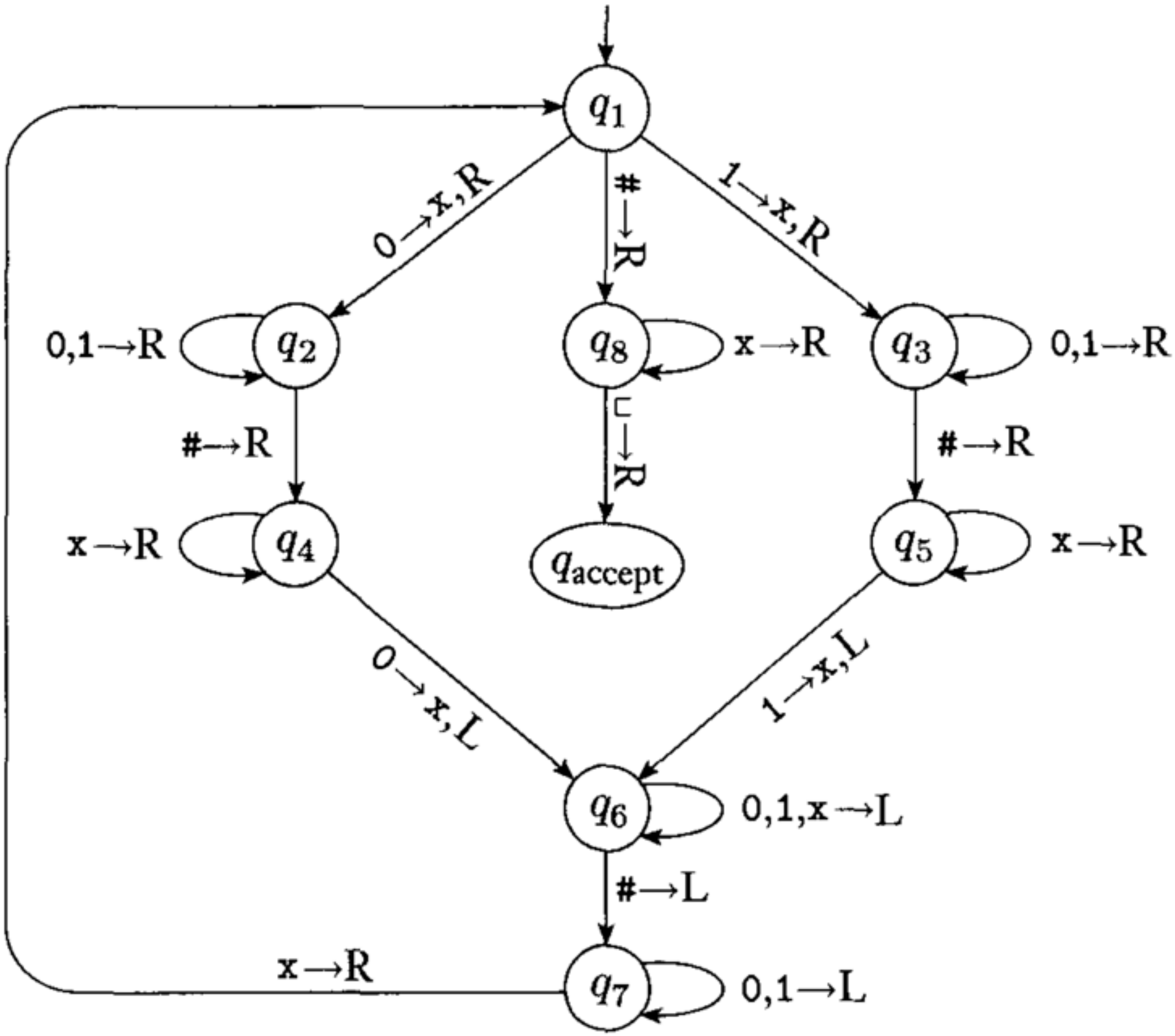


FIGURE 3.10
State diagram for Turing machine M_1

TM Configuration

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a ***configuration*** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ we write $u\,q\,v$ for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v. For example, $1011q_7 01111$ represents the configuration when the tape is 101101111, the current state is q_7, and the head is currently on the second 0. The following figure depicts a Turing machine with that configuration.

TM Computation

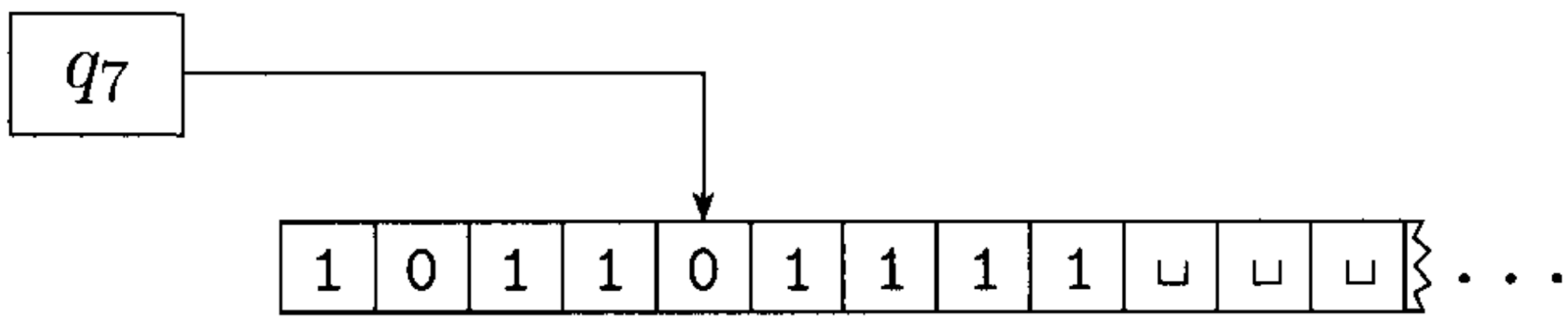


FIGURE 3.4
A Turing machine with configuration $1011q_701111$

TM definition

- For all $a,b,c \in \Gamma$, $u,v \in \Gamma^*$, $q_i,q_j \in Q$
- Config. $ua\mathbf{q_i}bv$ yields
  config. $u\mathbf{q_j}acv$ if $\delta(q_i,b) = q_j,c,L$
- Config. $ua\mathbf{q_i}bv$ yields
  config. $uac\mathbf{q_j}v$ if $\delta(q_i,b) = q_j,c,R$
- Special cases:
  Config. $\mathbf{q_i}bv$ yields $\mathbf{q_j}cv$ if $\delta(q_i,b) = q_j,c,L$
  Config. $\mathbf{q_i}bv$ yields $c\mathbf{q_j}v$ if $\delta(q_i,b) = q_j,c,R$

TM definition

ua $\mathbf{q_i}$ bv

yields (L)

u $\mathbf{q_j}$ acv

- Fo
- Co
  co
- Co
  co
- Sp
  Co ... $_j$,c,L
  Config: q_i bv yields c q_j v if $\delta(q_i,b) = q_j$,c,R

TM definition

- Fo
- Co
  co
- Co
  co
- Sp
  Co ... $_j$,c,L
  Config: $\mathbf{q_i}$ bv yields c $\mathbf{q_j}$ v if $\delta(q_i,b) = q_j$,c,R

ua $\mathbf{q_i}$ bv

yields (R)

uac $\mathbf{q_j}$ v

TM definition

q_i bv

yields (L)

q_j cv

- Fo
- Co
  co
- Co
  co
- Sp
  Co ... q_j,c,L
  Config. q_i bv yields c q_j v if $\delta(q_i,b) = q_j$,c,R

TM definition

q_i bv

yields (R)

c q_j v

- Fo
- Co
  co
- Co
  co
- Sp
  Co ... q_j,c,L
  Config: q_i bv yields c q_j v if $\delta(q_i,b) = q_j$,c,R

TM Computation

- Start configuration: $q_0 w$ (w = input string)
- Accepting configuration: state = q_{accept}
- Rejecting configuration: state = q_{reject}

TM Computation

- Turing Machine M accepts input w if there exists configurations $C_0, C_1, \ldots, C_m$ such that
  - C_0 is a start configuration
  - C_i yields C_{i+1} for $0 \leq i < m$
  - C_m is an accepting configuration.
- The collection of strings that M accepts is the language of M or the language recognized by M, denoted $L(M)$.

TM Computation

DEFINITION 3.5

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.[1]

- A TM decides a language if it recognizes it and halts (reaches an accepting or rejecting states) on all input strings.

DEFINITION 3.6

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.[2]

[1]Often named **Recursively-Enumerable** in the literature.

[2]Often named **Recursive** in the literature.

TM Examples

EXAMPLE 3.7

Here we describe a Turing machine (TM) M_2 that decides $A = \{0^{2^n} \mid n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

M_2 = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

TM Examples

Now we give the formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, x, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.

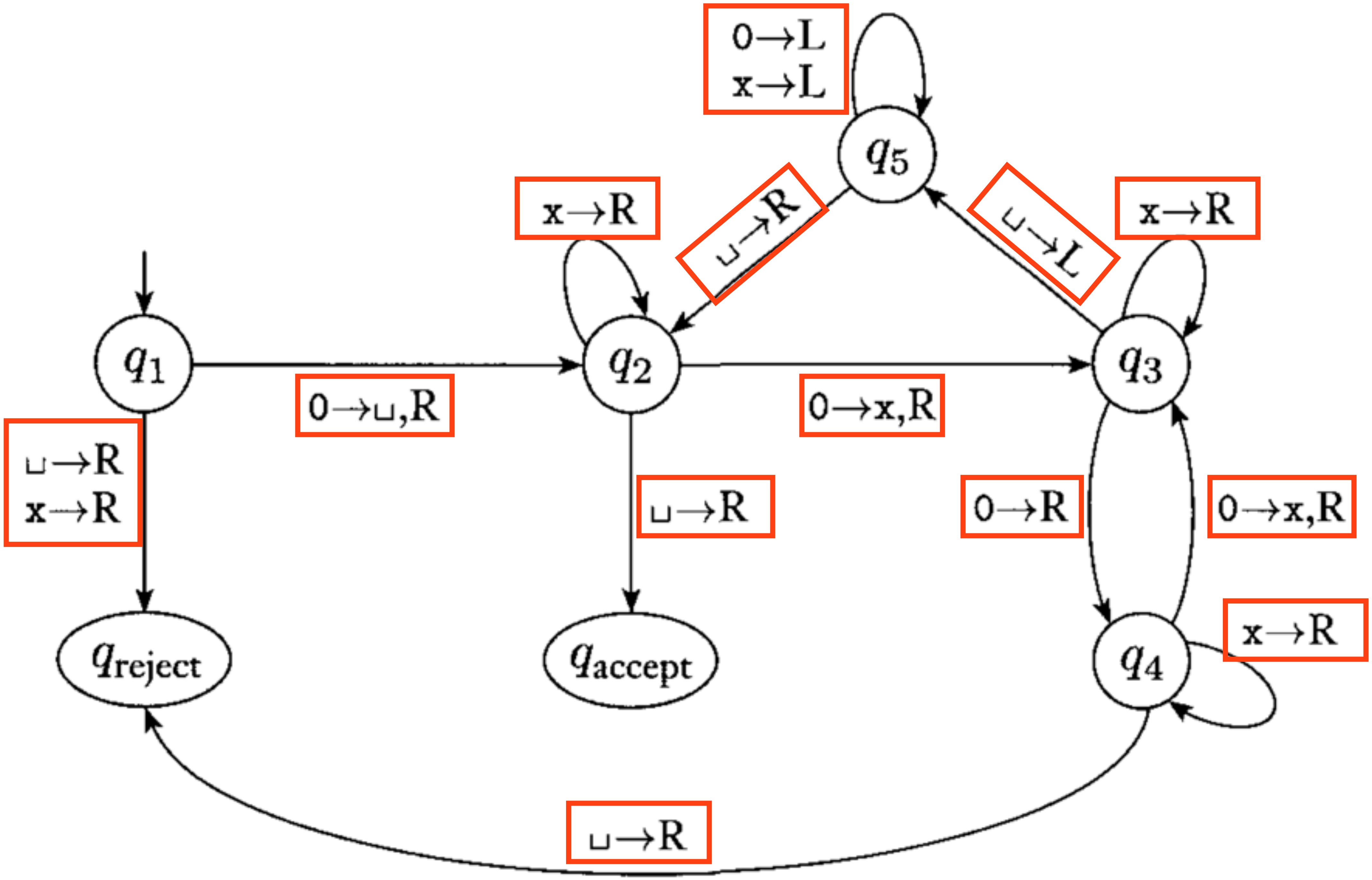


FIGURE 3.8
State diagram for Turing machine M_2

TM Examples

EXAMPLE 3.11

Here, a TM M_3 is doing some elementary arithmetic. It decides the language $C = \{a^i b^j c^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$.

M_3 = "On input string w:

1. Scan the input from left to right to determine whether it is a member of $a^+b^+c^+$ and *reject* if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, *reject*.
4. Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, *accept*; otherwise, *reject*."

TM Examples

M_4 = "On input w:

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, *accept*. If that symbol was a #, continue with the next stage. Otherwise, *reject*.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so *accept*.
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, *reject*.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so *accept*.
5. Go to Stage 3."

More Turing MACHINES

- Multitape Turing Machines
- Non-Deterministic Turing Machines
- Enumerator Turing Machines
- Everything else...

Multitape TM

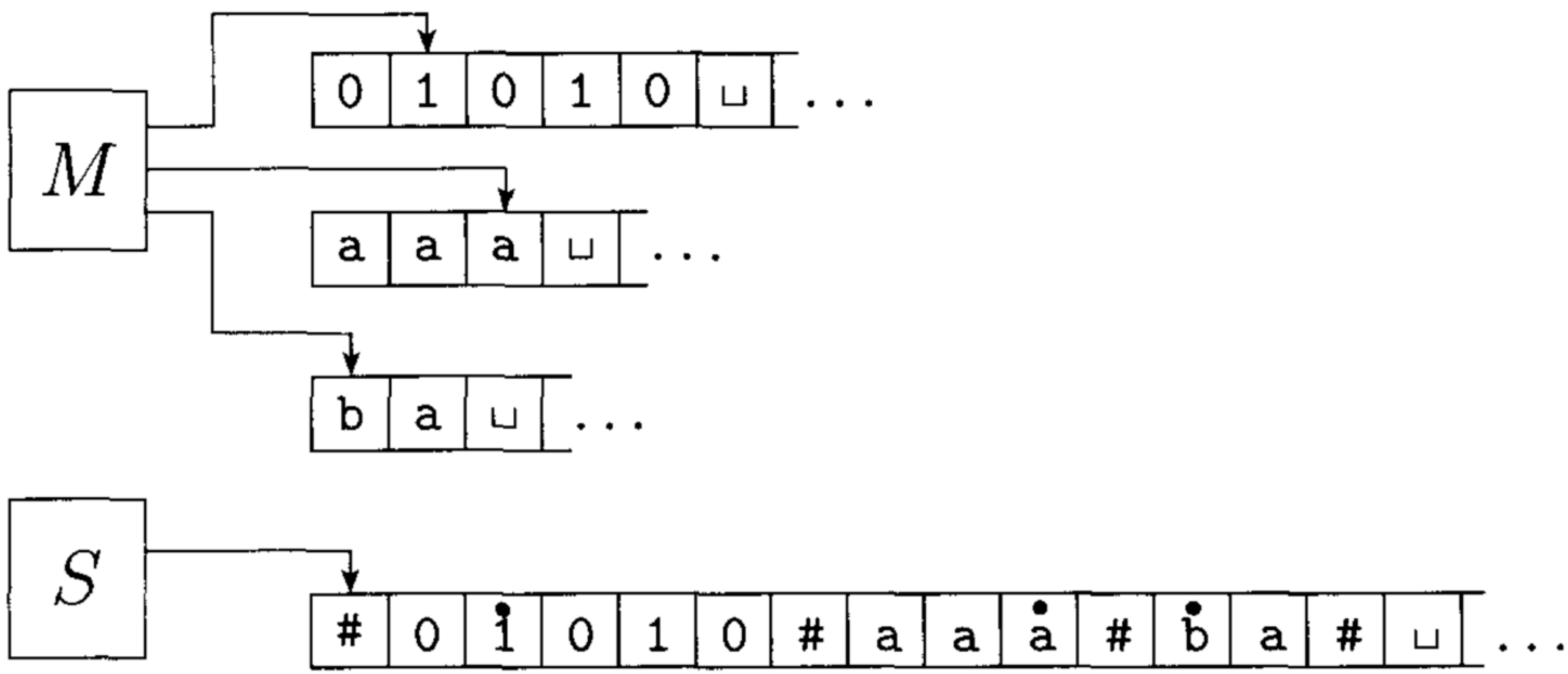


FIGURE 3.14
Representing three tapes with one

Multitape TM

$$\delta\colon Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k,$$

where k is the number of tapes. The expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$$

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

Multitape TM

$S =$ "On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains

$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#$$

2. To simulate a single move, S scans its tape from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

Multitape TM

COROLLARY 3.15

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

PROOF A Turing-recognizable language is recognized by an ordinary (single-tape) Turing machine, which is a special case of a multitape Turing machine. That proves one direction of this corollary. The other direction follows from Theorem 3.13.

Non-deterministic TM

The transition function for a nondeterministic Turing machine has the form

$$\delta \colon Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Non-deterministic TM

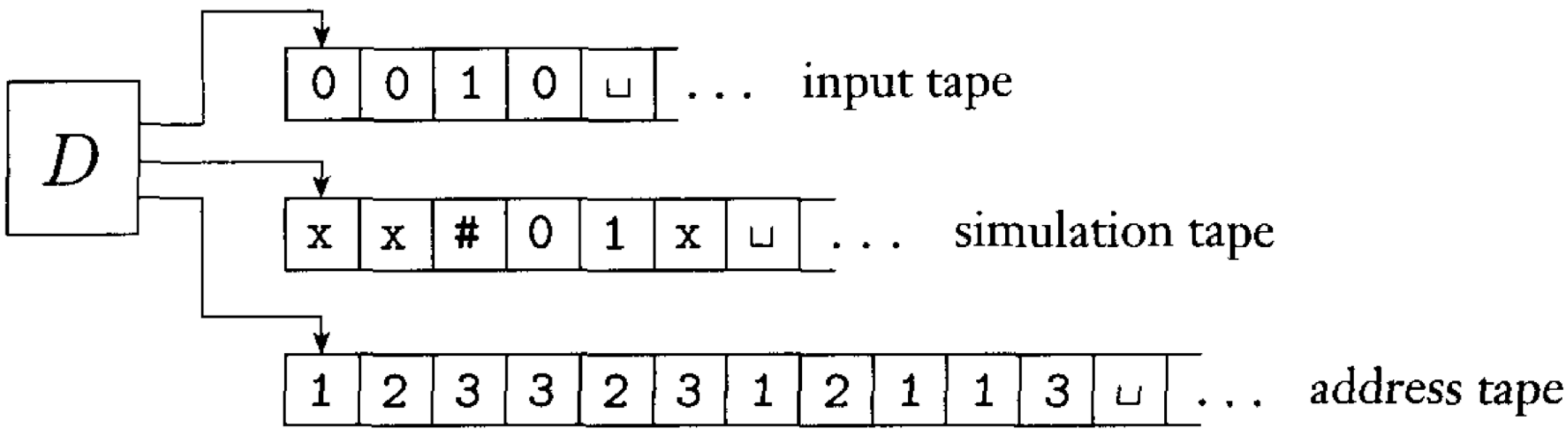


FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

Non-deterministic TM

1. Initially tape 1 contains the input w, and tapes 2 and 3 are empty.

2. Copy tape 1 to tape 2.

3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.

4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to stage 2.

Enumerator TM

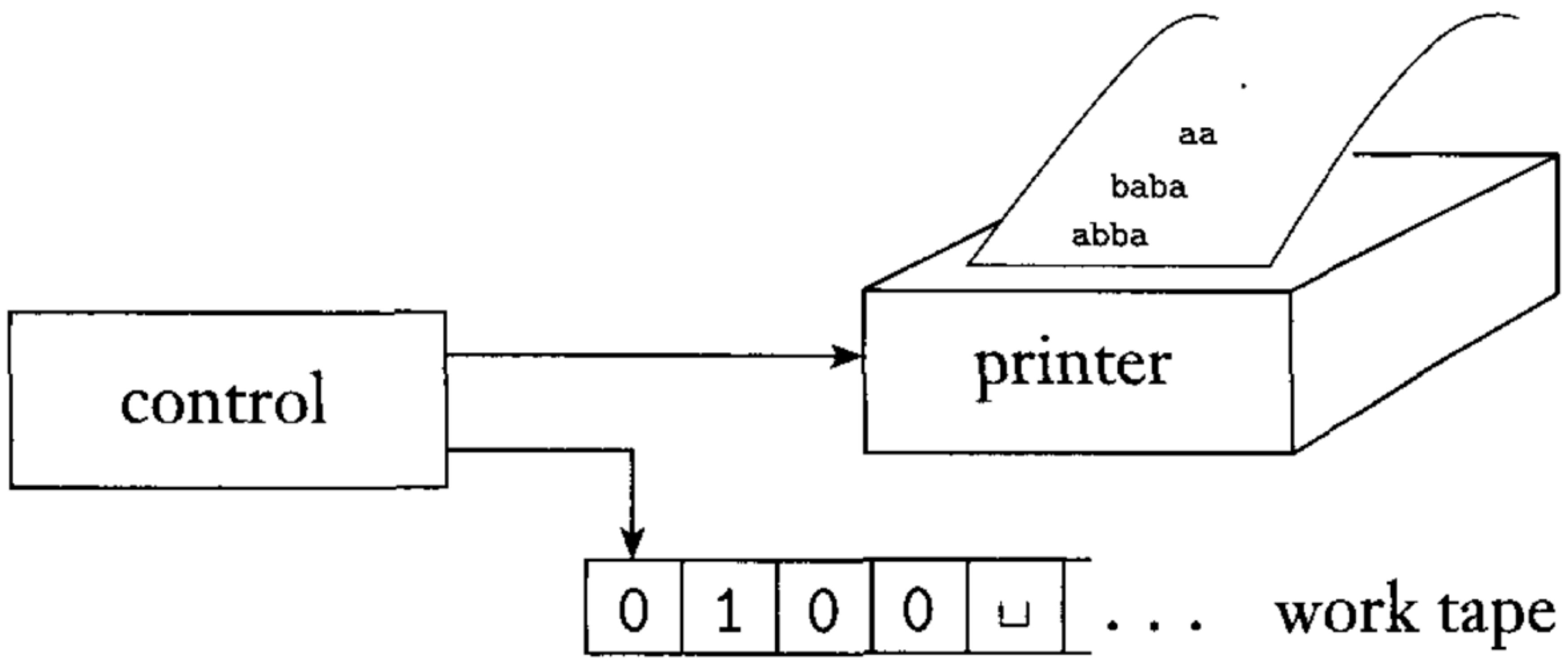


FIGURE 3.20
Schematic of an enumerator

Enumerator TM

THEOREM 3.21 ....

A language is Turing-recognizable if and only if some enumerator enumerates it.

Enumerator TM

PROOF First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. The TM M works in the following way.

M = "On input w:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, *accept*."

Clearly, M accepts those strings that appear on E's list.

Enumerator TM

Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.

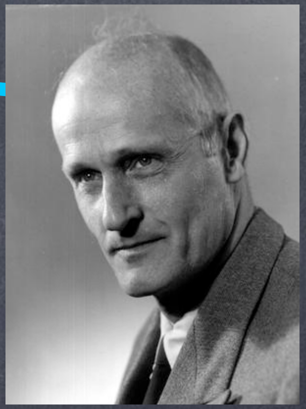
$E =$ "Ignore the input.

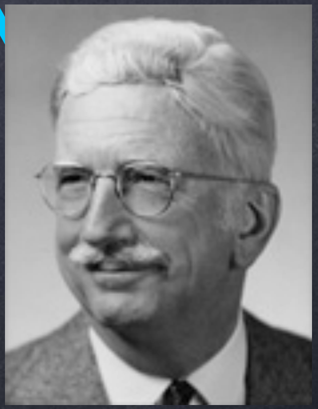
1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding s_j."

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

Everything Else

- Lambda-calculus
- Recursive Functions
- Programming languages:
  - FORTRAN, PASCAL, C, JAVA,...
  - LISP, SCHEME,...





Church-Turing Thesis

Alonzo Church

Alan Turing

Church-Turing Thesis

Intuitive notion of algorithms	equals	*Turing machine algorithms*

FIGURE 3.22
The Church–Turing Thesis

Hilbert's 10^{th} problem

- Let P be an integer-coefficient polynomial in **several variables**:
$$P(x,y,z)=24x^2y^3+17xz+5y+25$$
- Is there a set of integers for x,y,z such that
$$P(x,y,z)=0 \text{ ?}$$
- This problem is undecidable… but is Turing-Recognizable…
- Needed a formal model of computing to prove impossibility.

Yuri Matiyasevich

Single variable Poly

Let

$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

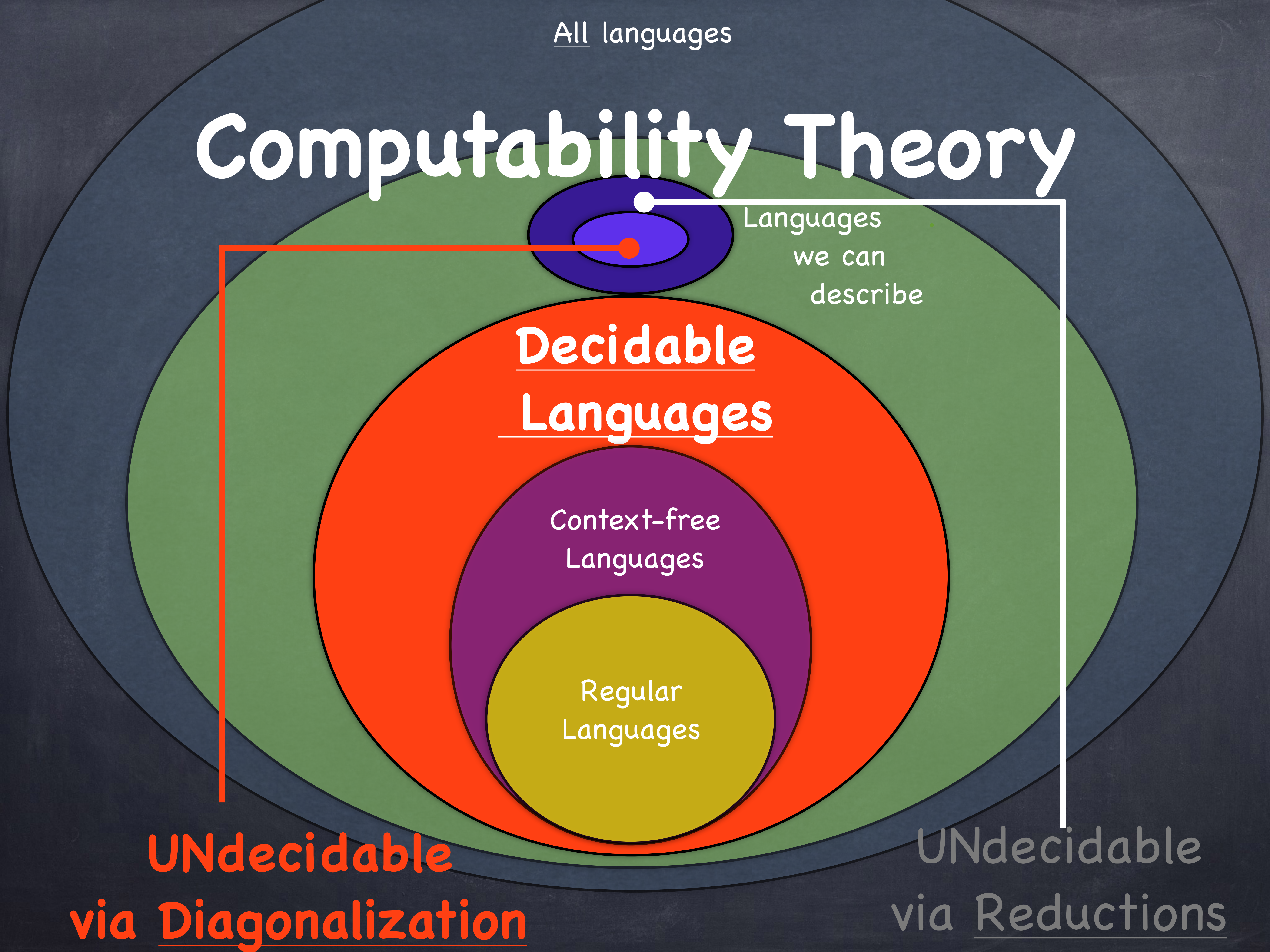
Here is a TM M_1 that recognizes D_1:

$M_1 =$ "The input is a polynomial p over the variable x.

1. Evaluate p with x set successively to the values 0, 1, −1, 2, −2, 3, −3, … If at any point the polynomial evaluates to 0, *accept*."

3.21 Let $c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$ be a polynomial with a root at $x = x_0$. Let $c_{\max}$ be the largest absolute value of a c_i. Show that

$$|x_0| < (n+1)\frac{c_{\max}}{|c_1|}.$$


All languages
Computability Theory
Languages
we can
describe
Decidable
Languages
Context-free
Languages
Regular
Languages
UNdecidable
via Diagonalization
UNdecidable
via Reductions

Turing Decidability

Alan Turing

Format & Notations

- Represent objects as strings
- $\langle O_1, O_2, \ldots, O_k \rangle$ is the string representing objects $O_1, O_2, \ldots, O_k$
- Many encodings are possible.
- Implicitly, at beginning of an algorithm, check that input is in the correct format, otherwise reject.

Format & Notations

EXAMPLE 3.23

Let A be the language consisting of all strings representing undirected graphs that are connected. Recall that a graph is ***connected*** if every node can be reached from every other node by traveling along the edges of the graph. We write

$$A = \{\langle G\rangle \mid G \text{ is a connected undirected graph}\}.$$

The following is a high-level description of a TM M that decides A.

Format & Notations

$M =$ "On input $\langle G \rangle$, the encoding of a graph G:

1. Select the first node of G and mark it.

2. Repeat the following stage until no new nodes are marked:

3. For each node in G, mark it if it is attached by an edge to a node that is already marked.

4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

Decidable Languages

Decidable	Undecidable
A_{DFA}	EQ_{CFG}
A_{NFA}	A_{TM}
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

Decidable Languages about DFA

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

THEOREM 4.1

A_{DFA} is a decidable language.

PROOF IDEA We simply need to present a TM M that decides A_{DFA}.

$M =$ "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Decidable Languages about DFA

We can prove a similar theorem for nondeterministic finite automata. Let

$$A_{\mathsf{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}.$$

THEOREM 4.2

A_{NFA} is a decidable language.

$N =$ "On input $\langle B, w\rangle$ where B is an NFA, and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Decidable Languages about DFA

Similarly, we can determine whether a regular expression generates a given string. Let $A_{\mathsf{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$.

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

P = "On input $\langle R, w\rangle$ where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Decidable Languages about DFA

$$E_{\mathsf{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}.$$

THEOREM 4.4 ....

E_{DFA} is a decidable language.

PROOF A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition we can design a TM T that uses a marking algorithm similar to that used in Example 3.23.

T = "On input $\langle A \rangle$ where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."

Decidable Languages about DFA

$$EQ_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}.$$

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem we use Theorem 4.4. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

Once we have constructed C we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

$F =$ "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

Decidable Languages about CFG

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}.$$

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF The TM S for A_{CFG} follows.

$S =$ "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w, except if $n = 0$, then instead list all derivations with 1 step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

Decidable Languages about CFG

$$E_{\mathsf{CFG}} = \{\langle G \rangle |\ G \text{ is a CFG and } L(G) = \emptyset\}.$$

THEOREM 4.8

E_{CFG} is a decidable language.

PROOF

$R =$ "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \rightarrow U_1U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Decidable Languages about CFG

THEOREM 4.9

Every context-free language is decidable.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

1. Run TM S on input $\langle G, w \rangle$
2. If this machine accepts, *accept*; if it rejects, *reject*."

Decidable Languages

Decidable	Undecidable
A_{DFA}	EQ_{CFG}
A_{NFA}	A_{TM}
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

Undecidable Languages about CFG

Next we consider the problem of determining whether two context-free grammars generate the same language. Let

$$EQ_{\mathsf{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}.$$

Undecidable Languages about TM

A_{DFA} and A_{CFG} were decidable, A_{TM} is not. Let

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

THEOREM 4.11

A_{TM} is undecidable.

A_{TM} is Turing-recognizable.

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

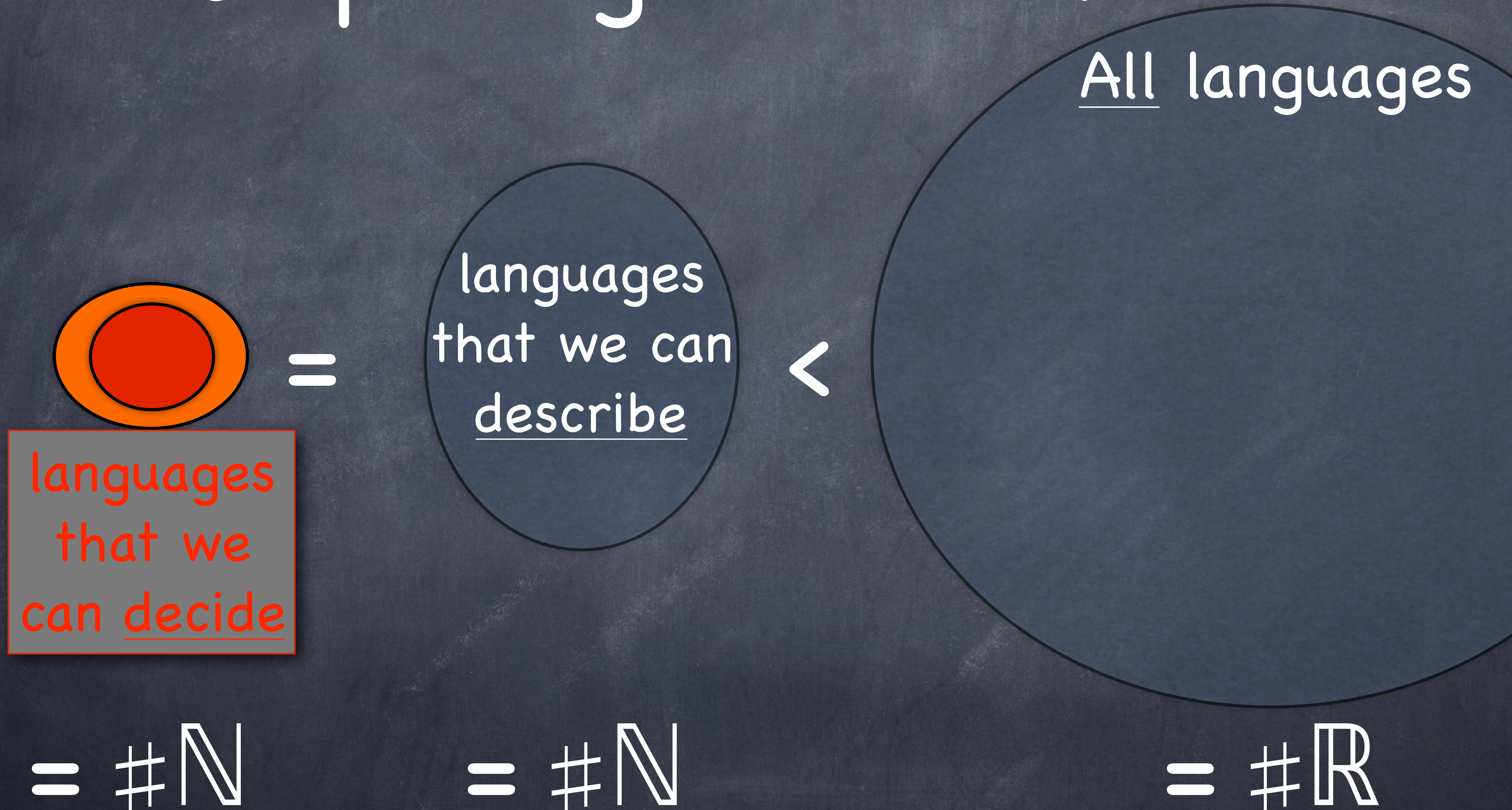
Comparing Cardinalities

All languages

languages that we can recognize = languages that we can describe < All languages

$= \#\mathbb{N}$ $\quad = \#\mathbb{N}$ $\quad = \#\mathbb{R}$

Comparing Cardinalities



Undecidable Language about TM

THE ACCEPTANCE PROBLEM IS UNDECIDABLE

Now we are ready to prove Theorem 4.11, the undecidability of the language

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

Undecidable Language about TM

Assumption: H exists

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

Undecidable Language about TM

H exists ⇒ D exists

$D =$ "On input $\langle M \rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs; that is, if H accepts, *reject* and if H rejects, *accept*."

Undecidable Language about TM

Properties of D

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

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

CONTRADICTION

Undecidable Language about TM

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

CONTRADICTION
CONTRADICTION
CONTRADICTION

- H accepts $\langle M, w\rangle$ exactly when M accepts w
- D rejects $\langle M\rangle$ exactly when M accepts $\langle M\rangle$
- D rejects $\langle D\rangle$ exactly when D accepts $\langle D\rangle$

$D =$ "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs; that is, if H accepts, *reject* and if H rejects, *accept*."

Undecidable Language about TM

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

FIGURE 4.19
Entry i, j is *accept* if M_i accepts $\langle M_j \rangle$

Undecidable Language about TM

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$		$\vdots$			

FIGURE 4.20
Entry i, j is the value of H on input $\langle M_i, \langle M_j \rangle \rangle$

Undecidable Language about TM

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

FIGURE 4.21
If D is in the **table**, a contradiction occurs at "?"

Diagonalization

Decidable	Undecidable
A_{DFA}	EQ_{CFG}
A_{NFA}	**A_{TM}**
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

Unrecognizable Language about TM

THEOREM 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

Let M_1 and M_2 be TMs respectively recognizing L and its complement $\bar{L}$.

M = "On input w:

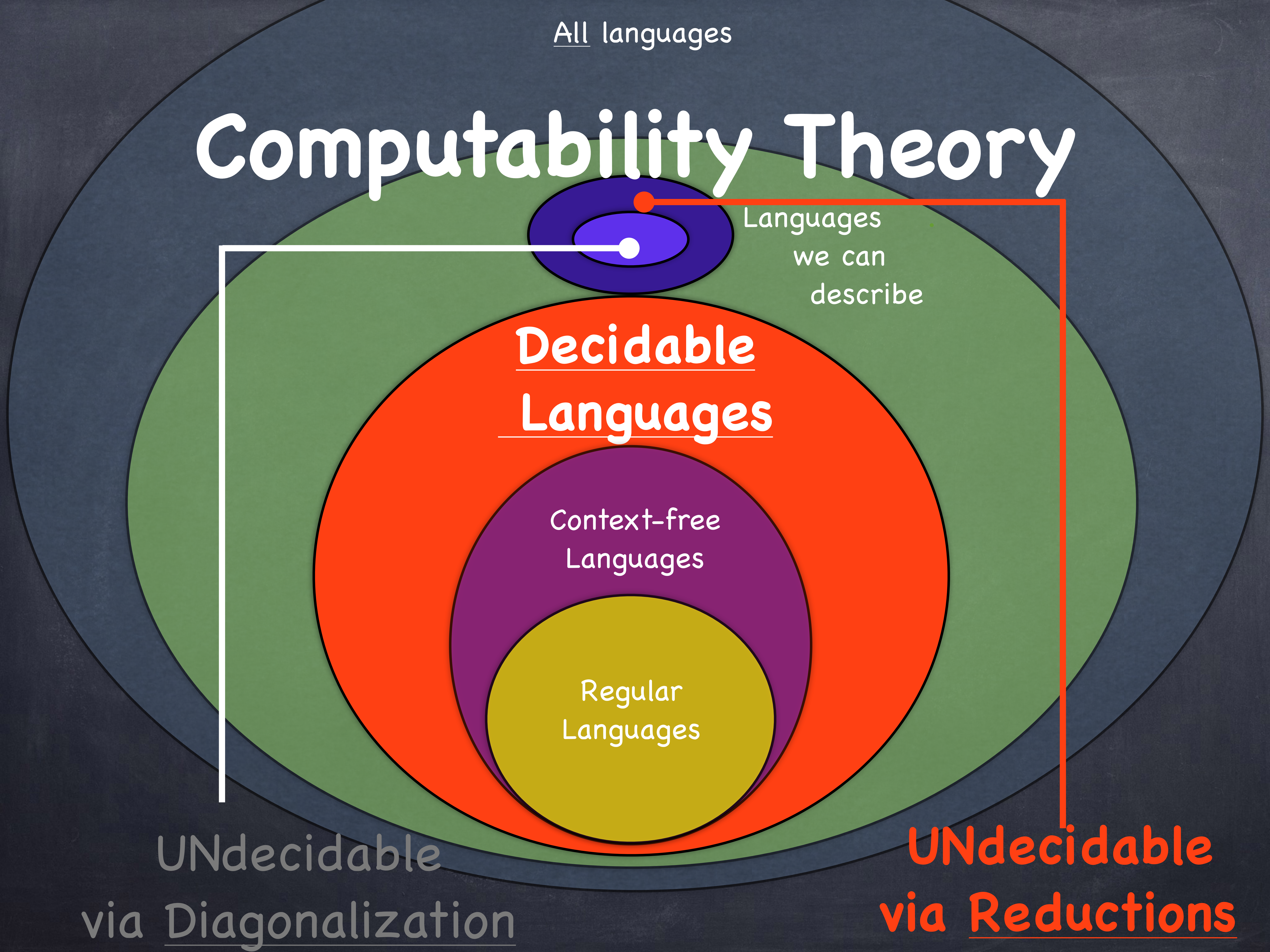
1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, *accept*; if M_2 accepts, *reject*."

Unrecognizable Language about TM

COROLLARY 4.23

$\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{\mathsf{TM}}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{\mathsf{TM}}}$ must not be Turing-recognizable.


All languages
Computability Theory
Languages
we can
describe
Decidable
Languages
Context-free
Languages
Regular
Languages
UNdecidable
via Diagonalization
UNdecidable
via Reductions

Reducibility

Decidable	Undecidable
A_{DFA}	EQ_{CFG}
A_{NFA}	A_{TM}
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

Reducibility

Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A. So in our example, A is the problem of finding your way around the city and B is the problem of obtaining a map. Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}.$$

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

Reducibility

$$E_{\mathsf{TM}} = \{\langle M\rangle |\ M \text{ is a TM and } L(M) = \emptyset\}.$$

THEOREM 5.2

E_{TM} is undecidable.

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

$M_1 =$ "On input x:
1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Reducibility

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1\rangle$.
3. If R accepts, *reject*; if R rejects, *accept*."

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it needs only add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

Reducibility

$$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}.$$

THEOREM 5.3

$REGULAR_{\mathsf{TM}}$ is undecidable.

Reducibility

PROOF We let R be a TM that decides $REGULAR_{\mathsf{TM}}$ and construct TM S to decide A_{TM}. Then S works in the following manner.

$S =$ "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Construct the following TM M_2.
   $M_2 =$ "On input x:
   1. If x has the form 0^n1^n, *accept*.
   2. If x does not have this form, run M on input w and *accept* if M accepts w."
2. Run R on input $\langle M_2\rangle$.
3. If R accepts, *accept*; if R rejects, *reject*."

$$L(M_2) = \begin{cases} \{0^n1^n \mid n \geq 0\} & \text{if } \mathbf{M} \text{ rejects } \mathbf{w} \\ \Sigma^* & \text{if } \mathbf{M} \text{ accepts } \mathbf{w} \end{cases}$$

Reducibility

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

THEOREM 5.4

EQ_{TM} is undecidable.

Reducibility

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

$S =$ "On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

Reducibility

ALL_{CFG}

Decidable	Undecidable
A_{DFA}	EQ_{CFG}
A_{NFA}	A_{TM}
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

MPCP

$AMBIG_{CFG}$

Post Correspondence Problem

Post Correspondence Problem

aaa	a	bbb	aa	__	b
bb	bb	a	a	bb	

- An instance of **PCP** with 6 dominos.

- A solution to **PCP**

aa	bbb	b	__	__
a	a		bb	bb

Post Correspondence Problem

u_1	u_2	u_3	. . .	u_n
v_1	v_2	v_3		v_n

- Given n dominos, $[u_1/v_1]$ ... $[u_n/v_n]$ where each u_i or v_i is a string of symbols.
- Is there an integer k and a sequence $\langle i_1, i_2, i_3, \ldots, i_k \rangle$ (with each $1 \leq i_j \leq n$) s.t.

$$u_{i_1} \circ u_{i_2} \circ u_{i_3} \circ \ \ldots \ \circ u_{i_k} = v_{i_1} \circ v_{i_2} \circ v_{i_3} \circ \ \ldots \ \circ v_{i_k} \ ?$$

A Solution to PCP

$$\frac{u_1}{v_1} \quad \frac{u_2}{v_2} \quad \frac{u_3}{v_3} \quad \cdots \quad \frac{u_n}{v_n}$$

- A solution is of this form:

$$\frac{u_{i_1}}{v_{i_1}} \quad \frac{u_{i_2}}{v_{i_2}} \quad \frac{u_{i_3}}{v_{i_3}} \quad \frac{u_{i_4}}{v_{i_4}} \quad \frac{u_{i_5}}{v_{i_5}} \quad \cdots \quad \frac{u_{i_k}}{v_{i_k}}$$

s.t.

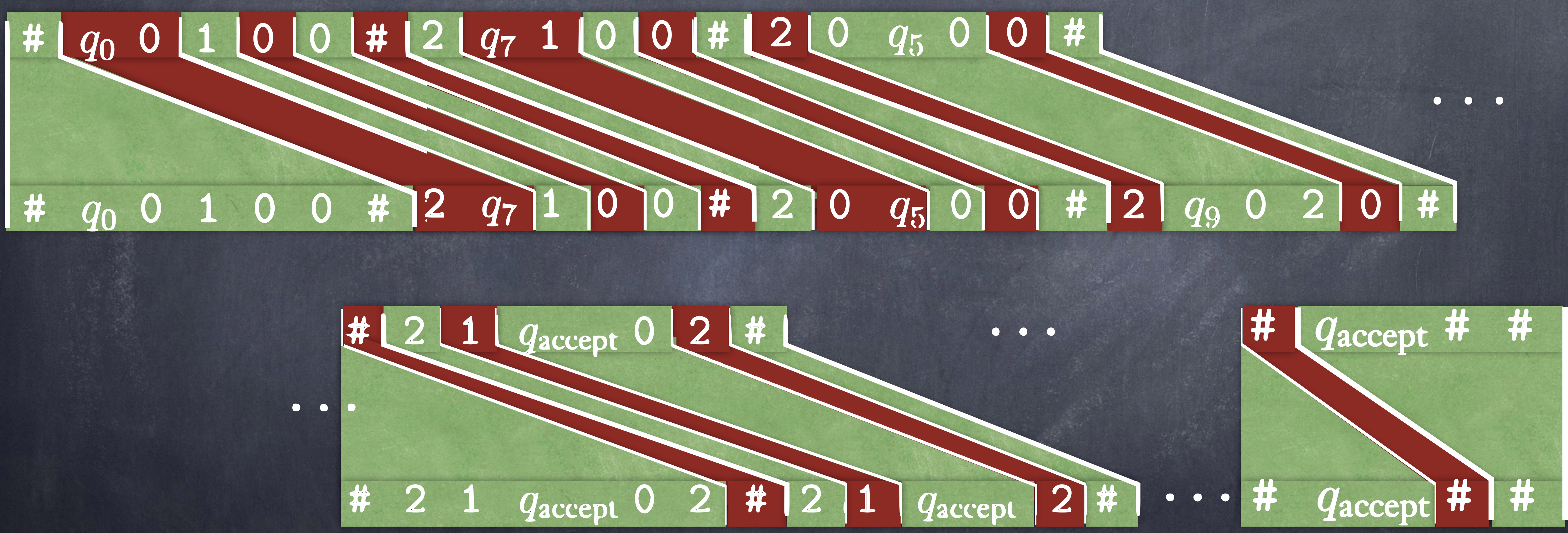
$$u_{i_1} \circ u_{i_2} \circ u_{i_3} \circ \ldots \circ u_{i_k} = v_{i_1} \circ v_{i_2} \circ v_{i_3} \circ \ldots \circ v_{i_k} ?$$

Post Correspondence Problem

- **Theorem:**

  The Post Correspondence Problem cannot be **decided** by any algorithm (or computer program). In particular, no algorithm can identify in a finite amount of time some instances that have a **No** outcome. However, if a solution exists, we can find it. **PCP** is Turing-recognizable.

Reducing A_{TM} to MPCP
a (mostly) complete example



Post Correspondence Problem

- <u>**Proof Idea**</u>:

  Reduction - if **PCP** was **decidable** then the **ACCEPTANCE** problem would be **decidable** as well.

Computation History

DEFINITION 5.5

Let M be a Turing machine and w an input string. An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A ***rejecting computation history*** for M on w is defined similarly, except that C_l is a rejecting configuration.

ATM : a Reduction to MPCP

A story in seven parts

start (part 1.)

$\left[\frac{\#}{\#q_0 w_1 w_2 \cdots w_n \#}\right]$

R-move (part 2.)

$\left[\frac{q_0 0}{2 q_7}\right]$

L-move (part 3.)

$\left[\frac{0 q_5 0}{q_9 0 2}\right]$ … $\left[\frac{2 q_5 0}{q_9 2 2}\right]$

copy (part 4.)

$\left[\frac{0}{0}\right]$ … $\left[\frac{\sqcup}{\sqcup}\right]$

blank (part 5.)

$\left[\frac{\#}{\#}\right]$ $\left[\frac{\#}{\sqcup \#}\right]$

clean (part 6.)

$\left[\frac{a\, q_{accept}}{q_{accept}}\right]$ $\left[\frac{q_{accept}\, a}{q_{accept}}\right]$

the end (part 7.)

$\left[\frac{q_{accept}\#\#}{\#}\right]$

Reducing MPCP to PCP

We now show how to convert P' to P, an instance of the PCP that still simulates M on w. We do so with a somewhat technical trick. The idea is to build the requirement of starting with the first domino directly into the problem so that stating the explicit requirement becomes unnecessary. We need to introduce some notation for this purpose.

Let $u = u_1 u_2 \cdots u_n$ be any string of length n. Define $\star u$, $u\star$, and $\star u \star$ to be the three strings

$$\begin{aligned}
\star u &= * u_1 * u_2 * u_3 * \cdots * u_n \\
u\star &= u_1 * u_2 * u_3 * \cdots * u_n * \\
\star u \star &= * u_1 * u_2 * u_3 * \cdots * u_n * .
\end{aligned}$$

Here, $\star u$ adds the symbol $*$ before every character in u, $u\star$ adds one after each character in u, and $\star u \star$ adds one both before and after each character in u.

Reducing MPCP to PCP

To convert P' to P, an instance of the PCP, we do the following. If P' were the collection

$$\left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

we let P be the collection

$$\left\{ \left[\frac{\star t_1}{\star b_1 \star}\right], \left[\frac{\star t_1}{b_1 \star}\right], \left[\frac{\star t_2}{b_2 \star}\right], \left[\frac{\star t_3}{b_3 \star}\right], \ldots, \left[\frac{\star t_k}{b_k \star}\right], \left[\frac{* \diamond}{\diamond}\right] \right\}.$$

Reducing MPCP to PCP

Considering P as an instance of the PCP, we see that the only domino that could possibly start a match is the first one,

$$\left[\frac{\star t_1}{\star b_1 \star}\right],$$

because it is the only one where both the top and the bottom start with the same symbol—namely, $*$. Besides forcing the match to start with the first domino, the presence of the $*$s doesn't affect possible matches because they simply interleave with the original symbols. The original symbols now occur in the even positions of the match. The domino

$$\left[\frac{*\diamond}{\diamond}\right]$$

is there to allow the top to add the extra $*$ at the end of the match.

Reducibility

ALL_{CFG}

Decidable	Undecidable
A_{DFA}	**EQ_{CFG}**
A_{NFA}	A_{TM}
A_{REX}	$HALT_{TM}$
E_{DFA}	E_{TM}
EQ_{DFA}	$REGULAR_{TM}$
A_{CFG}	EQ_{TM}
E_{CFG}	PCP

MPCP

$AMBIG_{CFG}$

Reducibility

$ALL_{\text{CFG}} = \{\langle G \rangle | \; G \text{ is a CFG and } L(G) = \Sigma^*\}.$

THEOREM 5.13

ALL_{CFG} is undecidable.

EQ_{CFG} decidable ⇒ ALL_{CFG} decidable

$EQ_{CFG} = \{ \langle G_1,G_2 \rangle \mid G_1,G_2 \text{ are CFGs and } L(G_1)=L(G_2)\}$

Let $\langle G_2 \rangle$ be such that $L(G_2)=\Sigma^*$. (G_2: $R \rightarrow \varepsilon \mid 0R \mid 1R$)

$\langle G \rangle \in ALL_{CFG} \Leftrightarrow \langle G,G_2 \rangle \in EQ_{CFG}$

ALL_{CFG} decidable ⇒ A_{TM} decidable

We now describe how to use a decision procedure for ALL_{CFG} to decide A_{TM}. For a TM M and an input w, we construct a CFG G that generates all strings if and only if M does not accept w. So if M does accept w, G does *not* generate some particular string. This string is—guess what—the accepting computation history for M on w. That is, G is designed to generate all strings that are *not* accepting computation histories for M on w.

To make the CFG G generate all strings that fail to be an accepting computation history for M on w, we utilize the following strategy. A string may fail to be an accepting computation history for several reasons. An accepting computation history for M on w appears as $\#C_1\#C_2\#\cdots\#C_l\#$, where C_i is the configuration of M on the ith step of the computation on w. Then, G generates all strings

1. that *do not* start with C_1,
2. that *do not* end with an accepting configuration, or
3. in which some C_i *does not* properly yield C_{i+1} under the rules of M.

If M does not accept w, no accepting computation history exists, so *all* strings fail in one way or another. Therefore, G would generate all strings, as desired.

PDA D(↔G) for M does not accept w

$$\#\underbrace{\overset{\longrightarrow}{}}_{C_1}\#\underbrace{\overset{\longleftarrow}{}}_{C_2^{\mathcal{R}}}\#\underbrace{\overset{\longrightarrow}{}}_{C_3}\#\underbrace{\overset{\longleftarrow}{}}_{C_4^{\mathcal{R}}}\#\quad\cdots\quad\#\underbrace{}_{C_l}\#$$

FIGURE 5.14
Every other configuration written in reverse order

$$\#\underbrace{\longrightarrow}_{C_1}\#\underbrace{\longleftarrow}_{C_2^{\mathcal{R}}}\#\underbrace{\longrightarrow}_{C_3}\#\underbrace{\longleftarrow}_{C_4^{\mathcal{R}}}\#\;\cdots\;\#\underbrace{\qquad}_{C_l}\#$$

- One branch checks on whether the beginning of the input string is C_1 and accepts if it isn't.
- Another branch checks on whether the input string ends with a configuration containing the accept state, q_{accept}, and accepts if it isn't.
- The third branch is supposed to accept if some C_i does not properly yield C_{i+1}:
  - It works by scanning the input until it nondeterministically decides that it has come to C_i.
  - Next, it pushes C_i onto the stack until it comes to the end as marked by the # symbol.
  - Then D pops the stack to compare with C_{i+1}.
  - They are supposed to match except around the head position, where the difference is dictated by the transition function of M.
  - Finally, D accepts if it discovers a mismatch or an improper update.

PDA D(↔G) for ⟨M⟩ does not accept w

$$L(D) = \begin{cases} \Sigma^* \setminus \{\text{accepting computation history}\} & \text{if } \mathbf{M} \text{ accepts } \mathbf{w} \\ \Sigma^* & \text{if } \mathbf{M} \text{ rejects } \mathbf{w} \end{cases}$$

- On input ⟨M,w⟩ generate ⟨G⟩ s.t.
  $L(G)=\Sigma^* \leftrightarrow$ M rejects w
- If All_{CFG} is decidable, then so is A_{TM}.

Mapping Reducibility

Computable Functions

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

EXAMPLE 5.18

All usual arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n. We don't give any details here, leaving them as exercises.

Mapping Reducibility

FORMAL DEFINITION OF MAPPING REDUCIBILITY

Now we define mapping reducibility. As usual we represent computational problems by languages.

DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

The function f is called the ***reduction*** of A to B.

Mapping Reducibility

The following figure illustrates mapping reducibility.

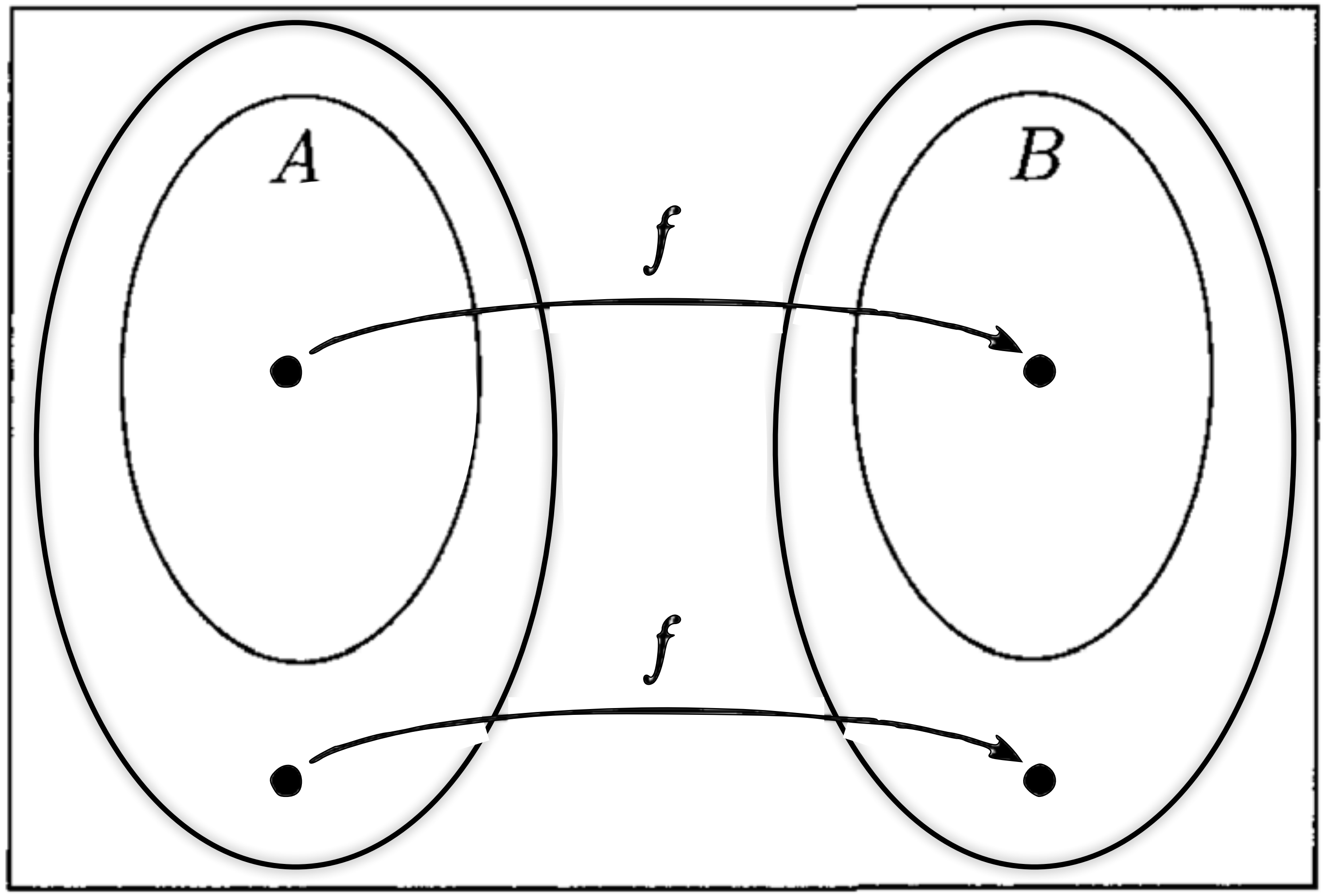


FIGURE 5.21

THEOREM 5.22

If $A \leq_m B$ and B is decidable, then A is decidable.

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Therefore N works as desired.

COROLLARY 5.23

If $A \leq_m B$ and A is undecidable, then B is undecidable.

EXAMPLE 5.24

In Theorem 5.1 we used a reduction from A_{TM} to prove that $HALT_{\mathsf{TM}}$ is undecidable. This reduction showed how a decider for $HALT_{\mathsf{TM}}$ could be used to give a decider for A_{TM}. We can demonstrate a mapping reducibility from A_{TM} to $HALT_{\mathsf{TM}}$ as follows. To do so we must present a computable function f that takes input of the form $\langle M, w\rangle$ and returns output of the form $\langle M', w'\rangle$, where

$$\langle M, w\rangle \in A_{\mathsf{TM}} \text{ if and only if } \langle M', w'\rangle \in HALT_{\mathsf{TM}}.$$

The following machine F computes a reduction f.

F = "On input $\langle M, w\rangle$:

1. Construct the following machine M'.
   M' = "On input x:
   1. Run M on x.
   2. If M accepts, *accept*.
   3. If M rejects, enter a loop."
2. Output $\langle M', w\rangle$."

Mapping Reducibility

EXAMPLE 5.25

The proof of the undecidability of the Post correspondence problem in Theorem 5.15 contains two mapping reductions. First, it shows that $A_{\mathsf{TM}} \leq_{\mathrm{m}} MPCP$ and then it shows that $MPCP \leq_{\mathrm{m}} PCP$. In both cases we can easily obtain the actual reduction function and show that it is a mapping reduction. As Exercise 5.6 shows, mapping reducibility is transitive, so these two reductions together imply that $A_{\mathsf{TM}} \leq_{\mathrm{m}} PCP$.

Mapping Reducibility

THEOREM 5.28

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

The proof is the same as that of Theorem 5.22, except that M and N are recognizers instead of deciders.

COROLLARY 5.29

If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

Mapping Reducibility

In a typical application of this corollary, we let A be $\overline{A_{\mathsf{TM}}}$, the complement of A_{TM}. We know that $\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable from Corollary 4.23. The definition of mapping reducibility implies that $A \leq_{\mathrm{m}} B$ means the same as $\overline{A} \leq_{\mathrm{m}} \overline{B}$. To prove that B isn't recognizable we may show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{B}$. We can also use mapping reducibility to show that certain problems are neither Turing-recognizable nor co-Turing-recognizable, as in the following theorem.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

PROOF First we show that EQ_{TM} is not Turing-recognizable. We do so by showing that A_{TM} is reducible to $\overline{EQ_{\mathsf{TM}}}$. The reducing function f works as follows.

$F =$ "On input $\langle M, w \rangle$ where M is a TM and w a string:

1. Construct the following two machines M_1 and M_2.
   $M_1 =$ "On any input:
   1. *Reject*."

   $M_2 =$ "On any input:
   1. Run M on w. If it accepts, *accept*."
2. Output $\langle M_1, M_2 \rangle$."

Here, M_1 accepts nothing. If M accepts w, M_2 accepts everything, and so the two machines are not equivalent. Conversely, if M doesn't accept w, M_2 accepts nothing, and they are equivalent. Thus f reduces A_{TM} to $\overline{EQ_{\mathsf{TM}}}$, as desired.

THEOREM 5.30

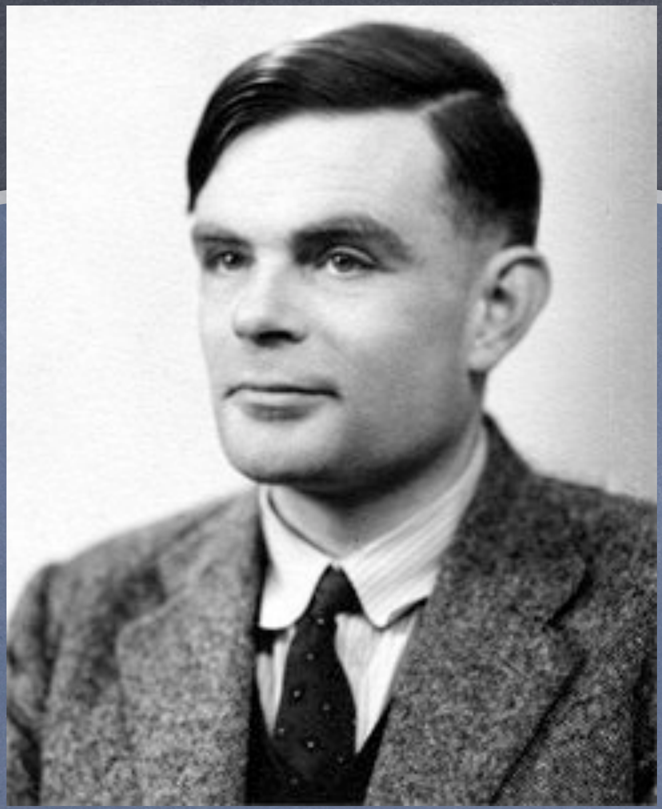
EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

To show that $\overline{EQ_{\mathsf{TM}}}$ is not Turing-recognizable we give a reduction from A_{TM} to the complement of $\overline{EQ_{\mathsf{TM}}}$—namely, EQ_{TM}. Hence we show that $A_{\mathsf{TM}} \leq_m EQ_{\mathsf{TM}}$. The following TM G computes the reducing function g.

$G =$ "The input is $\langle M, w\rangle$ where M is a TM and w a string:

1. Construct the following two machines M_1 and M_2.
   $M_1 =$ "On any input:
      1. *Accept*."
   $M_2 =$ "On any input:
      1. Run M on w.
      2. If it accepts, *accept*."
2. Output $\langle M_1, M_2\rangle$."

The only difference between f and g is in machine M_1. In f, machine M_1 always rejects, whereas in g it always accepts. In both f and g, M accepts w iff M_2 always accepts. In g, M accepts w iff M_1 and M_2 are equivalent. That is why g is a reduction from A_{TM} to EQ_{TM}.



Turing Reducibility

Turing Reducibility

DEFINITION 6.18

An ***oracle*** for a language B is an external device that is capable of reporting whether any string w is a member of B. An ***oracle Turing machine*** is a modified Turing machine that has the additional capability of querying an oracle. We write M^B to describe an oracle Turing machine that has an oracle for language B.

Turing Reducibility

EXAMPLE 6.19

Consider an oracle for A_{TM}. An oracle Turing machine with an oracle for A_{TM} can decide more languages than an ordinary Turing machine . Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input. It can also decide E_{TM}, the emptiness testing problem for TMs with the following procedure called $T^{A_{\mathsf{TM}}}$.

$T^{A_{\mathsf{TM}}}$ = "On input $\langle M \rangle$, where M is a TM:

1. Construct the following TM N.
   N = "On any input:
   1. Run M in parallel on all strings in Σ^*.
   2. If M accepts any of these strings, *accept*."
2. Query the oracle to determine whether $\langle N, 0 \rangle \in A_{\mathsf{TM}}$.
3. If the oracle answers NO, *accept*; if YES, *reject*."

Turing Reducibility

DEFINITION 6.20

Language A is ***Turing reducible*** to language B, written $A \leq_{\mathrm{T}} B$, if A is decidable relative to B.

Turing Reducibility

THEOREM 6.21

If $A \leq_T B$ and B is decidable, then A is decidable.

PROOF If B is decidable, then we may replace the oracle for B by an actual procedure that decides B. Thus we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A.

Computability Theory

Tractable Problems

Tractable Problems (P)

- 2-colorability of maps.
- Primality testing.
  (but probably not factoring)
- Solving NxNxN Rubik's cube.
- Finding a word in a dictionary.
- Sorting elements...

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.
- More formally, there exists a TM to compute solutions to the problem and there exists a polynomial Q such that the number of steps on each input x before halting is no more than Q(|x|).

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.
- Computer Science studies mostly techniques to approach and find efficient solutions to tractable problems.
- Some problems may be efficiently solvable but we might not be able to prove that…

Tractable Problems (P)

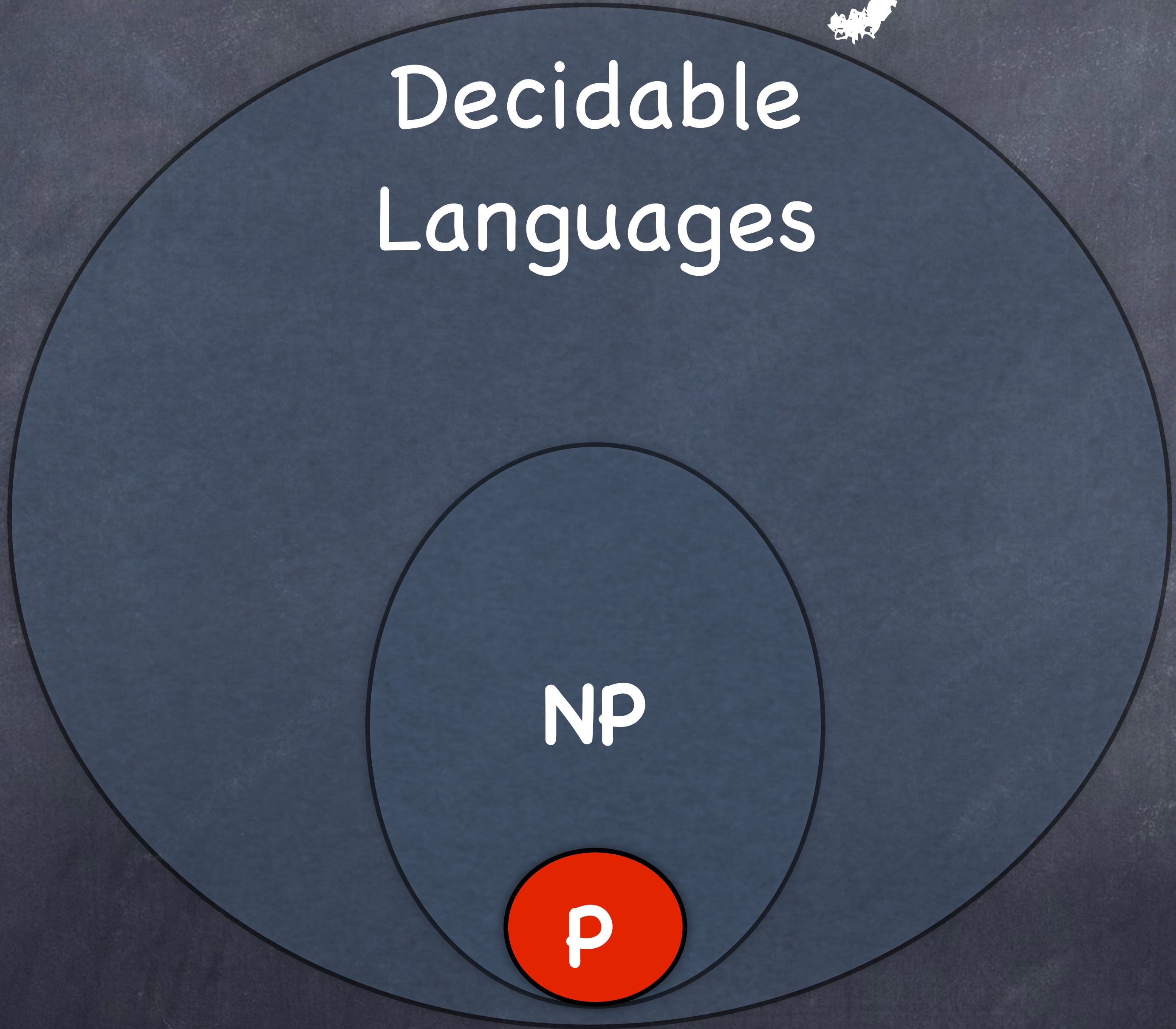
- The name P stands for Polynomial-Time computable.
- Q: Why choose this level of granularity ? Why not choose linear-time for instance ?
- A: because P is the same for all types of Turing machines and any reasonable model. This is not true of linear-time for instance...

Tractable Problems (P)

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Complexity Theory



P = NP ?

K-colouring of Maps (planar graphs)

- K=1 only the maps with zero or one region are 1-colourable.
- K=2 easy to decide. Impossible as soon as 3 regions touch each other.
- K=3 No known efficient algorithm to decide. It is easy to verify a solution.
- K≥4 all maps are 4-colourable. (long proof) Does not imply easy to find a 4-colouring.

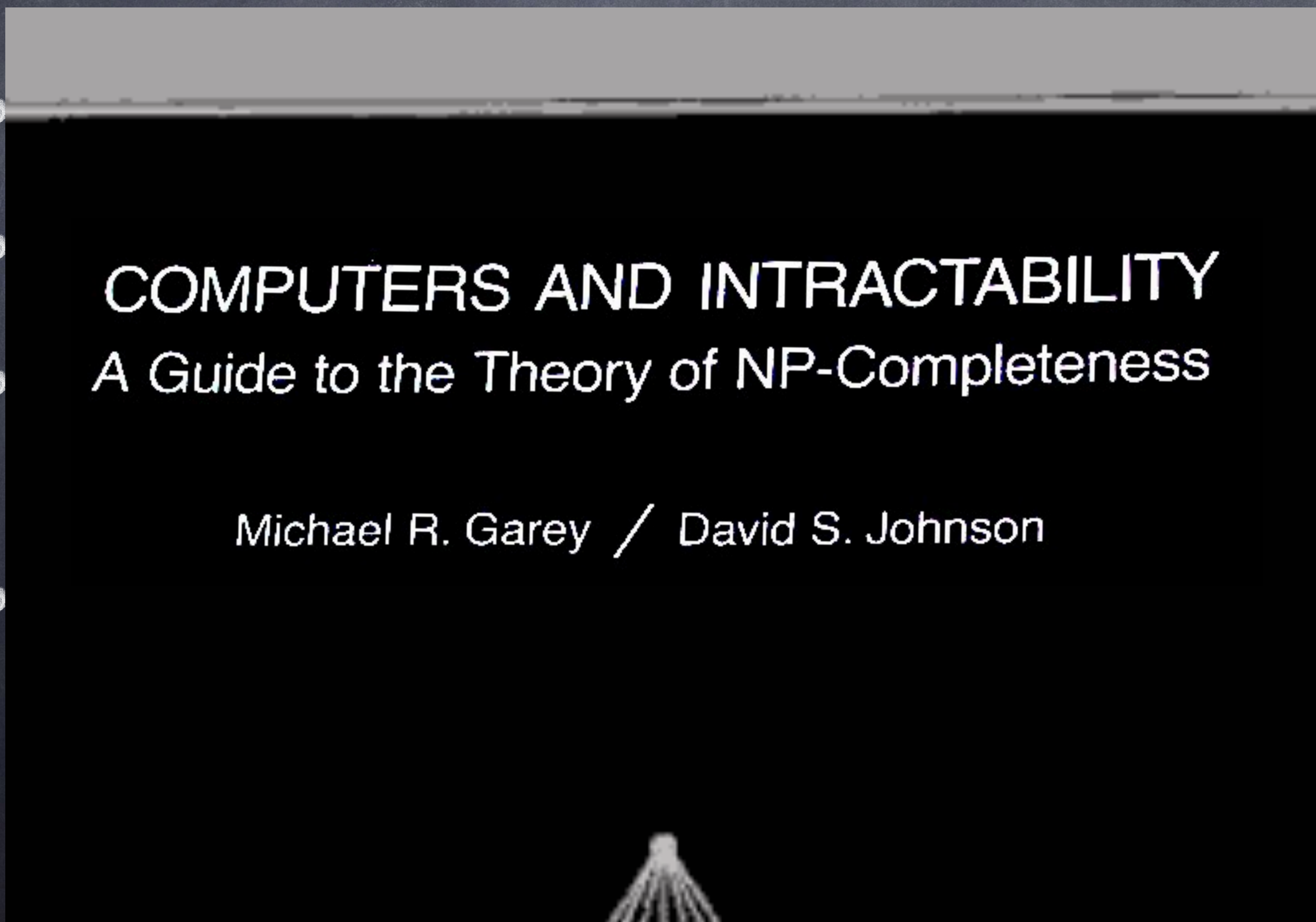
3-colouring of Maps

- Seems hard to solve in general,
- Is easy to verify when a solution is given, (is in NP : guess a solution and verify it)
- Is a special type of problem (NP-complete) because an efficient solution to it would yield efficient solutions to ALL problems in NP!

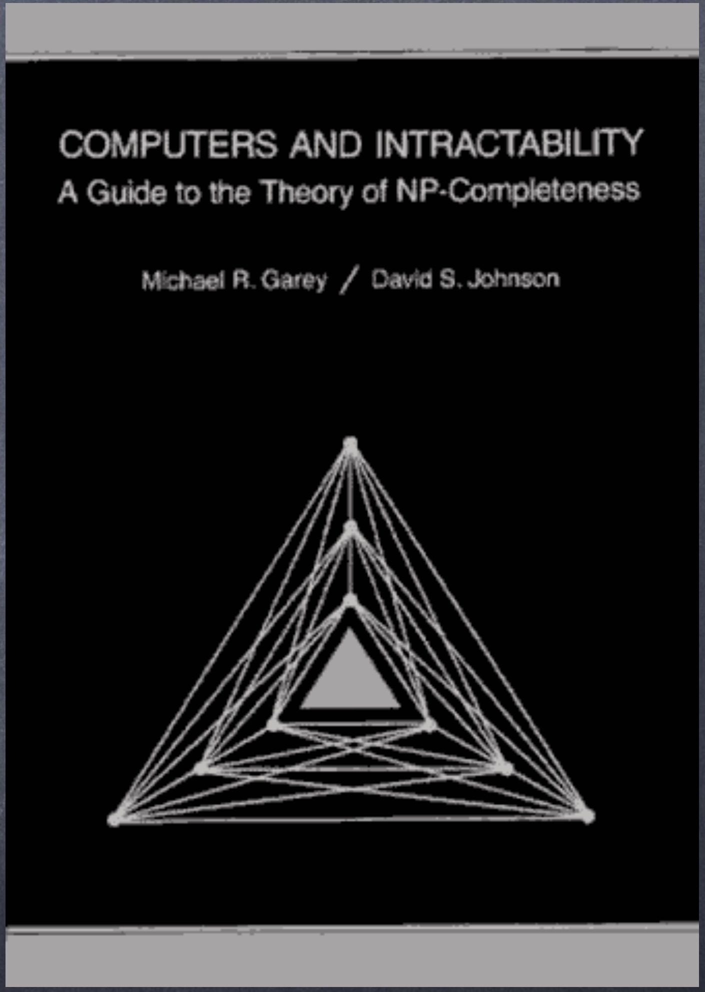
Examples of NP-Complete Problems

- SAT: given a boolean formula, is there an assignment of the variables making the formula evaluate to true ?
- Travelling Salesman: given a set of cities and distances between them, what is the shortest route to visit each city once.
- KnapSack: given items with various weights, is there of subset of them of total weight K.

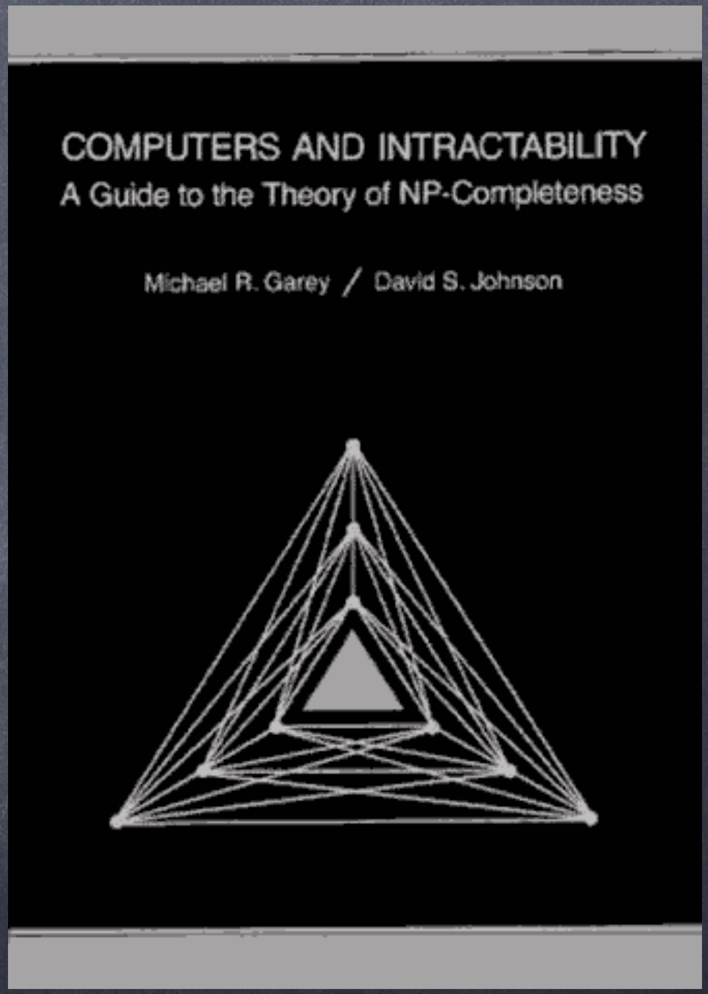
NP-Complete Problems



NP-Complete Problems



NP-Complete Problems



100 pages
1979 !!!

Complexity Theory

P = NP ?

P vs NP

P vs NP

DEFINITION 7.7

Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(\boldsymbol{t(n)})$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

P vs NP

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

P vs NP

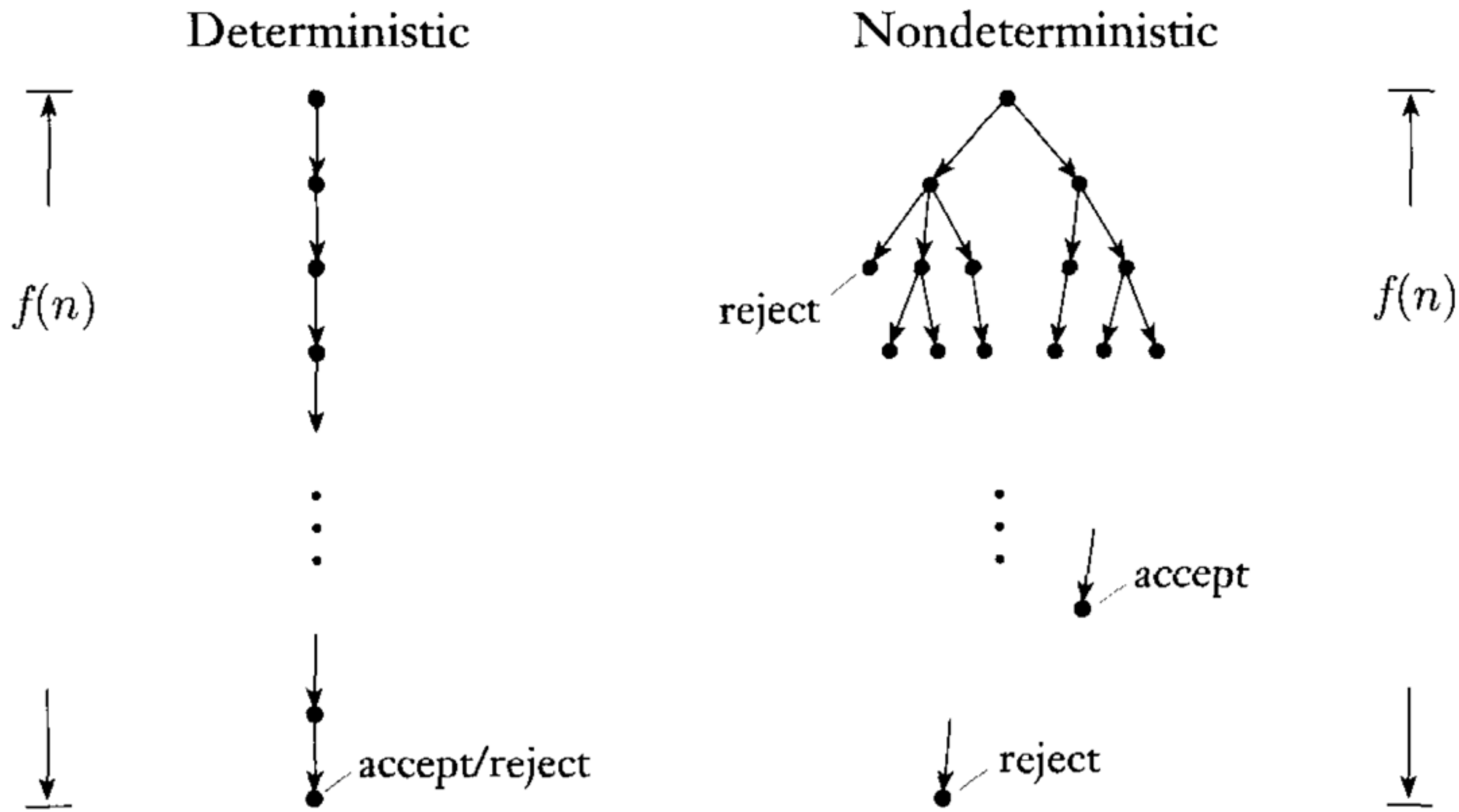


FIGURE 7.10
Measuring deterministic and nondeterministic time

P vs NP

DEFINITION 7.21

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

COROLLARY 7.22

$\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

P vs NP

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

P vs NP

A ***clique*** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A ***k*-*clique*** is a clique that contains k nodes. Figure 7.23 illustrates a graph having a 5-clique

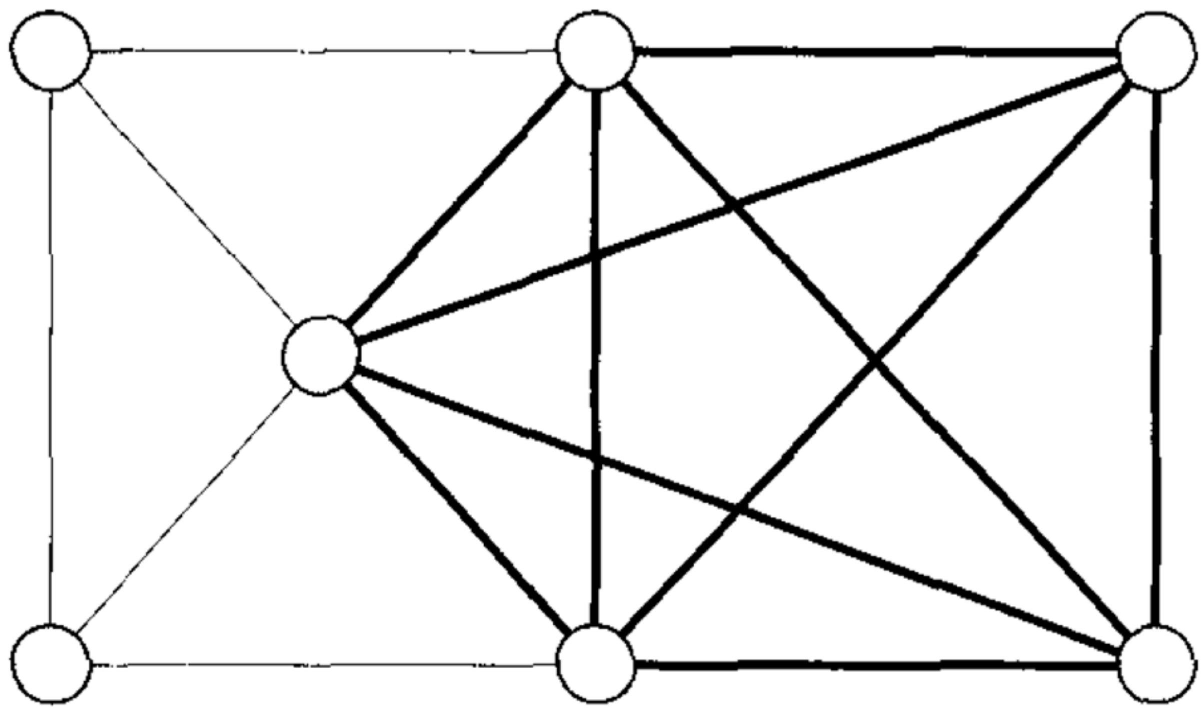


FIGURE 7.23
A graph with a 5-clique

P vs NP

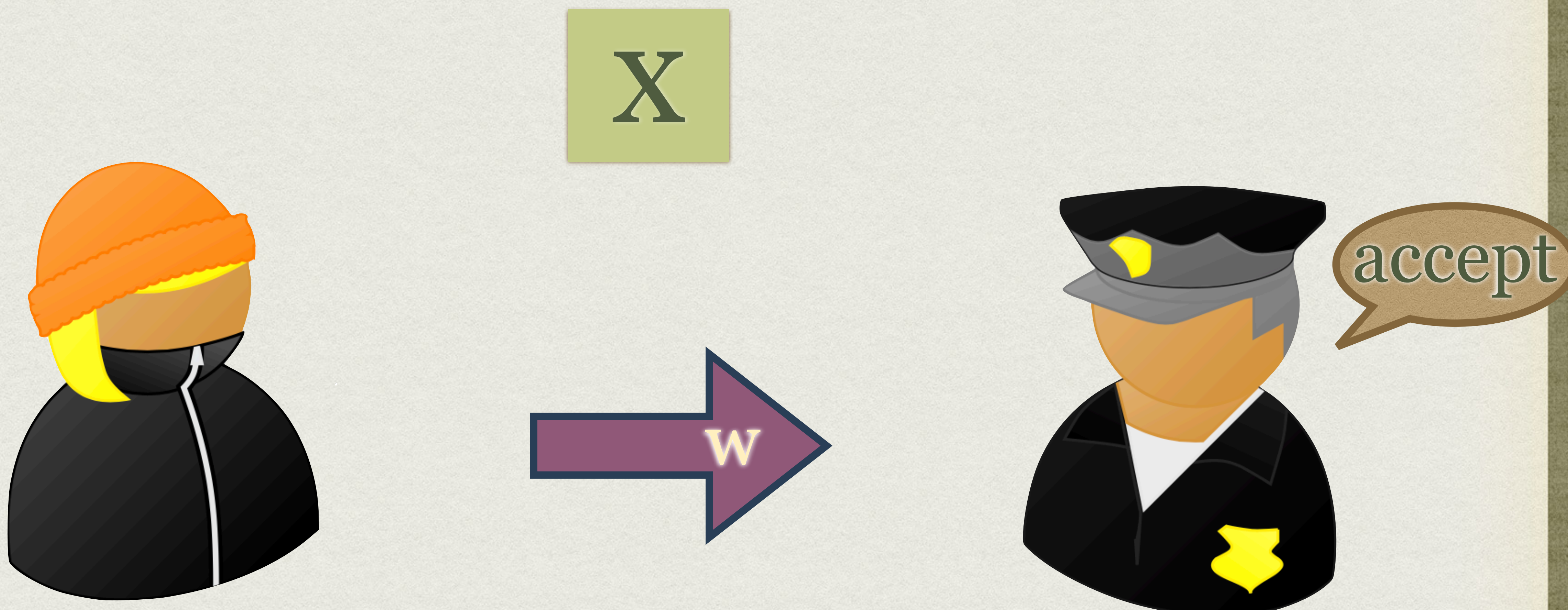
The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}.$$

Completeness

x∈L

$\exists$  , $\forall x \in L, \exists w,$ [ (x, w) accepts]



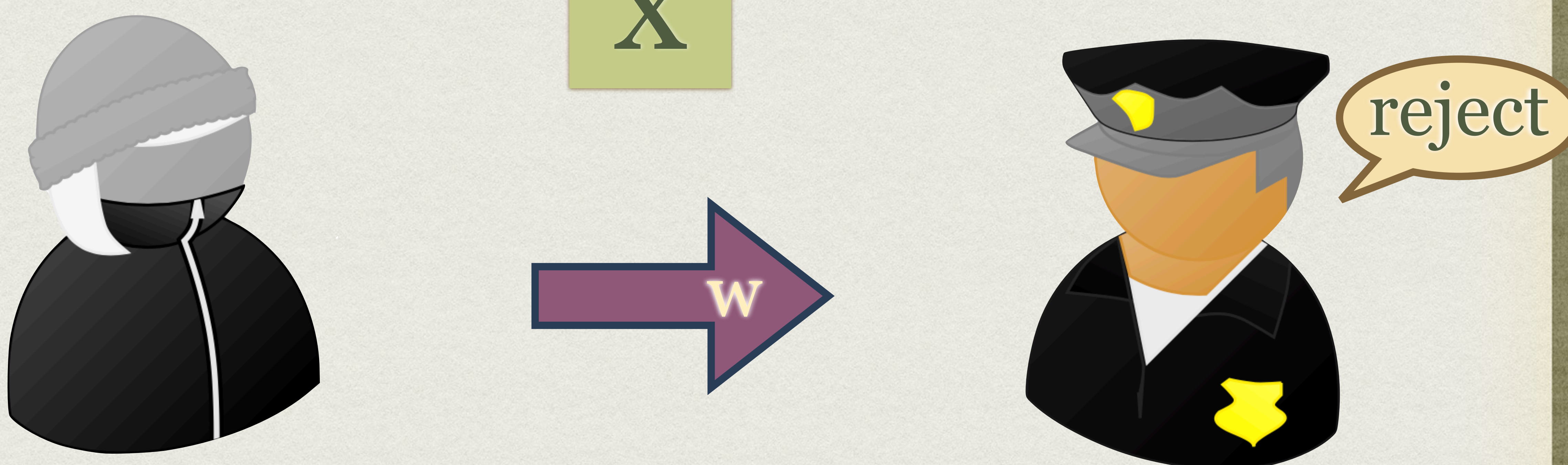
Soundness

x∉L

∃, $\forall x \in L, \exists w,$ [(x, w) accepts]

and $\forall x \notin L, \forall w,$ [(x, w) rejects]

X



THEOREM 7.24

CLIQUE is in NP.

PROOF IDEA The clique is the certificate.

PROOF The following is a verifier V for *CLIQUE*.

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

$N =$ "On input $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

SAT

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

Now we state the Cook–Levin theorem, which links the complexity of the *SAT* problem to the complexities of all problems in NP.

THEOREM 7.27

Cook–Levin theorem $SAT \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$.

Poly-time Reducibility

DEFINITION 7.28

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Poly-time Reducibility

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

Poly-time Reducibility

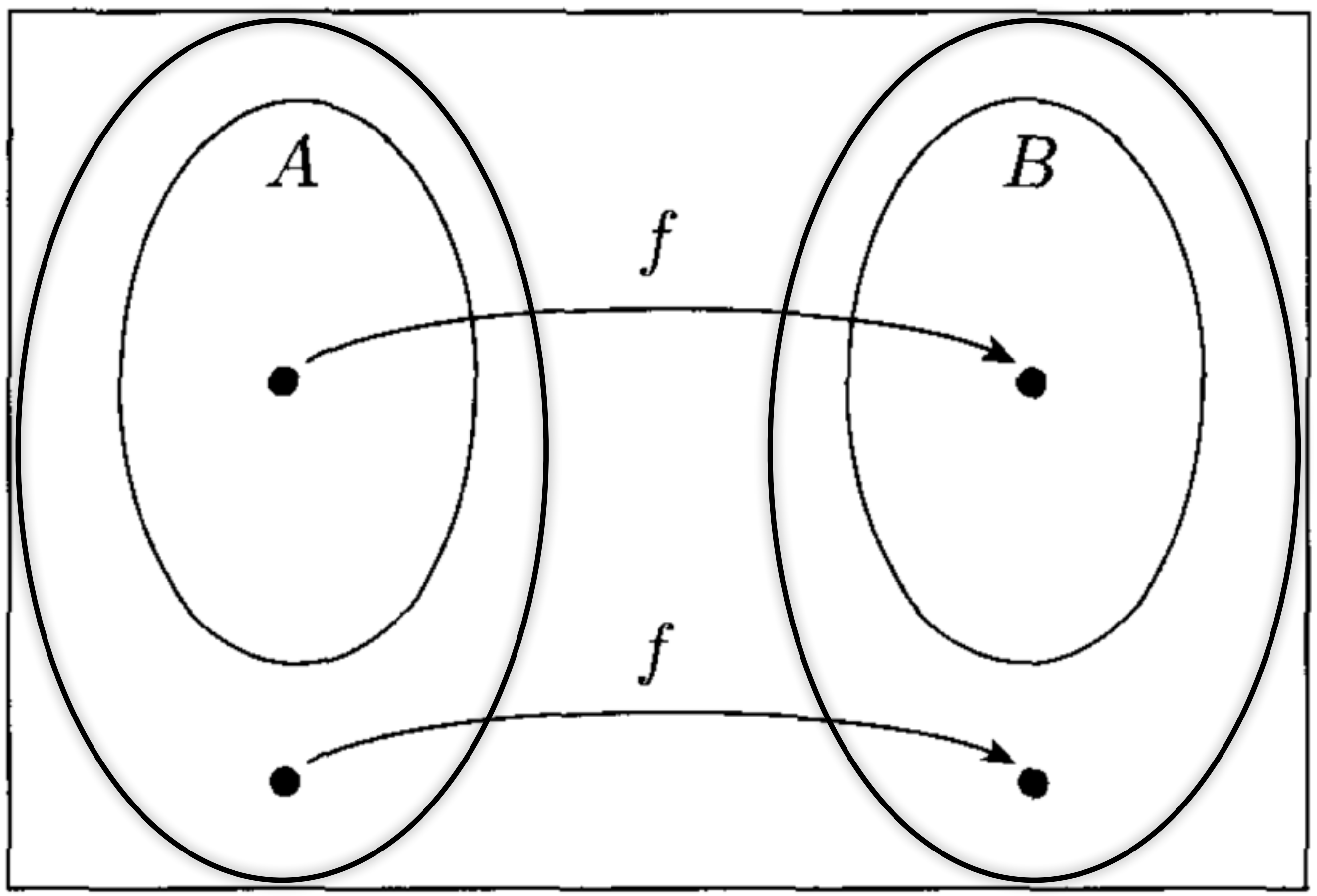


FIGURE 7.30
Polynomial time function f reducing A to B

Poly-time Reducibility

THEOREM 7.31

If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

NP-completeness

DEFINITION 7.34

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

THEOREM 7.35

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

PROOF This theorem follows directly from the definition of polynomial time reducibility.

NP-completeness

THEOREM 7.36

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

PROOF We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in NP is polynomial time reducible to C.

Cook-Levin Theorem

Cook-Levin Theorem

THEOREM 7.37 ....

SAT is NP-complete.[2]

This theorem restates Theorem 7.27, the Cook–Levin theorem, in another form.

Cook-Levin Theorem

PROOF First, we show that *SAT* is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Next, we take any language* A in NP and show that A is polynomial time reducible to *SAT*. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (For convenience we actually assume that N runs in time $n^k - 3$, but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

*"any language A in NP" really means:

"any language A *provably* in **NP**".

A ***tableau*** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w, as shown in the following figure.

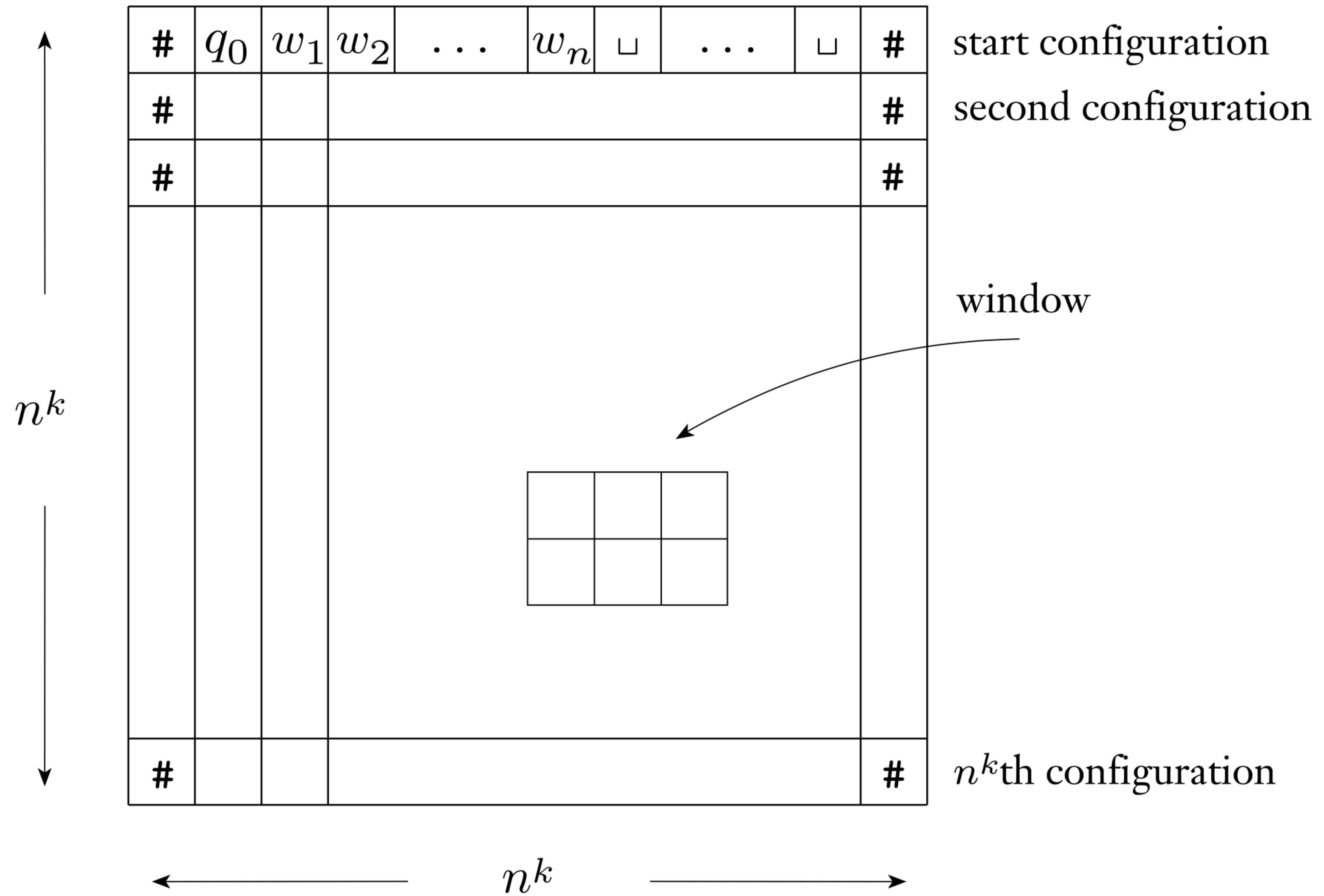
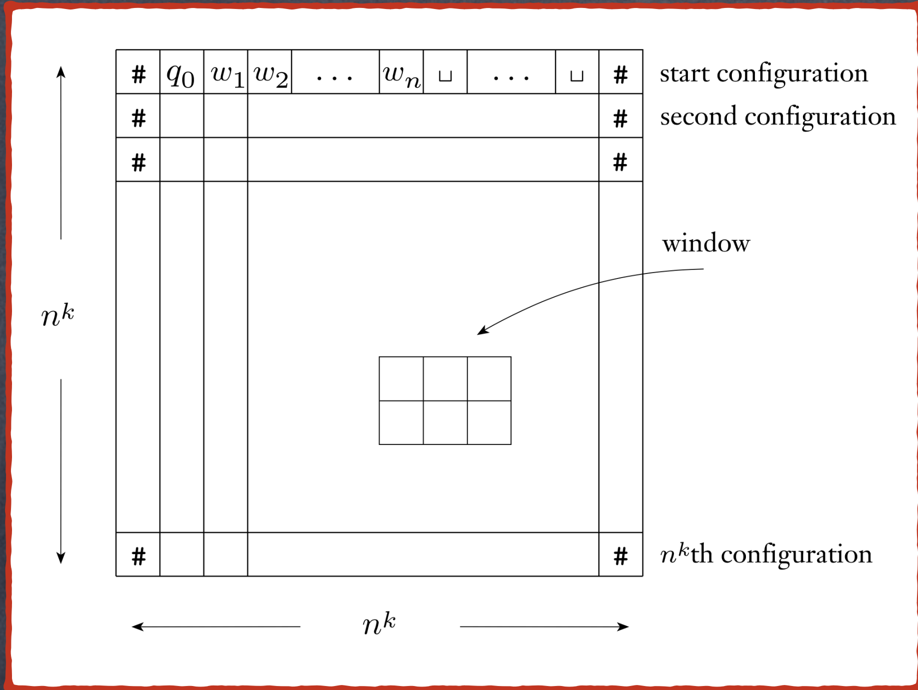


FIGURE 7.38
A tableau is an $n^k \times n^k$ table of configurations

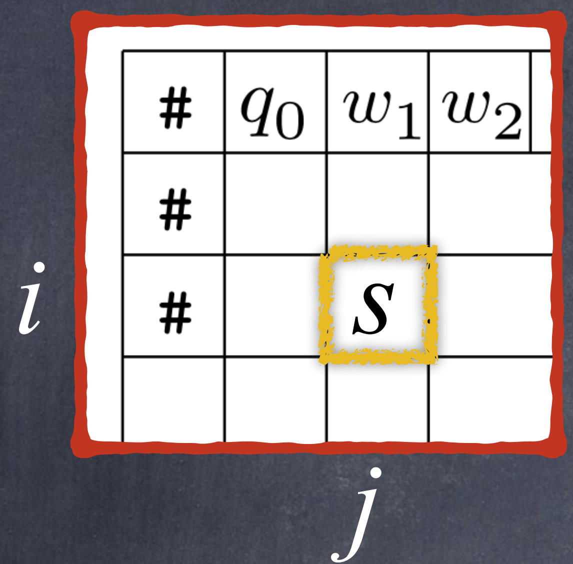


Cook-Levin Theorem

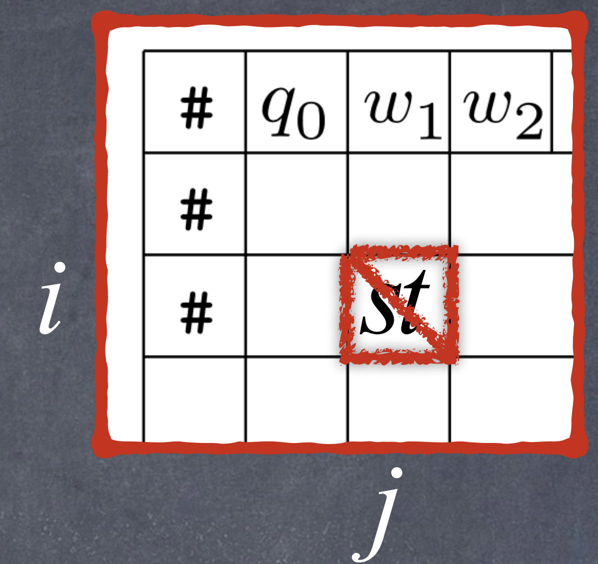
Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.

$$\phi = \phi_{cell} \cup \phi_{start} \cup \phi_{accept} \cup \phi_{move}$$



Cook-Levin Theorem: ϕ_{cell}



turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

$$C = Q \cup \Gamma \cup \{\#\}.$$

Cook-Levin Theorem: ϕ_{cell}

The symbols $\bigwedge$ and $\bigvee$ stand for iterated AND and OR. For example, the expression in the preceding formula

$$\bigvee_{s \in C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

where $C = \{s_1, s_2, \ldots, s_l\}$. Hence ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k.

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

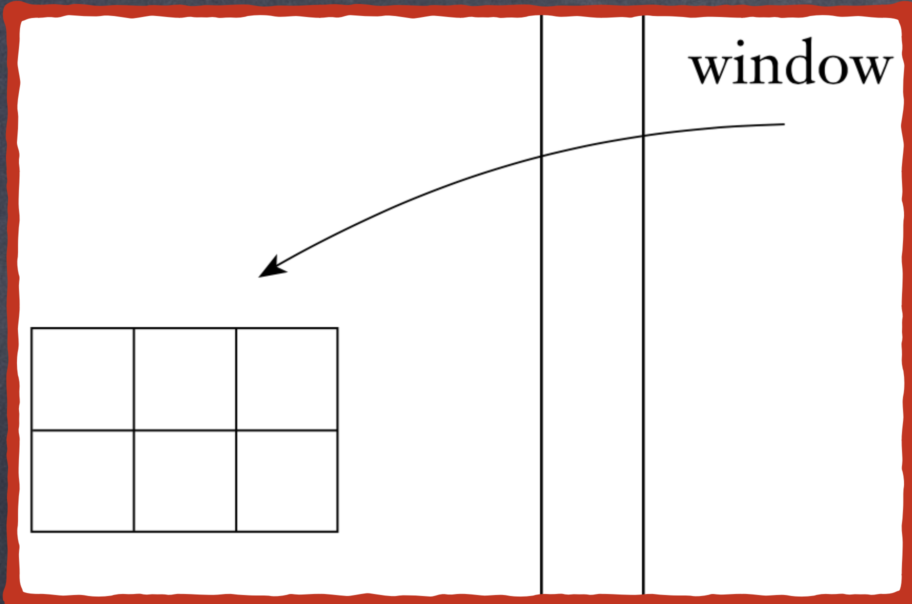
$$\phi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge$$
$$x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge$$
$$x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} .$$

Cook-Levin Theorem: ϕ_{accept}

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}} .$$

Cook-Levin Theorem: ϕ_{move}



(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

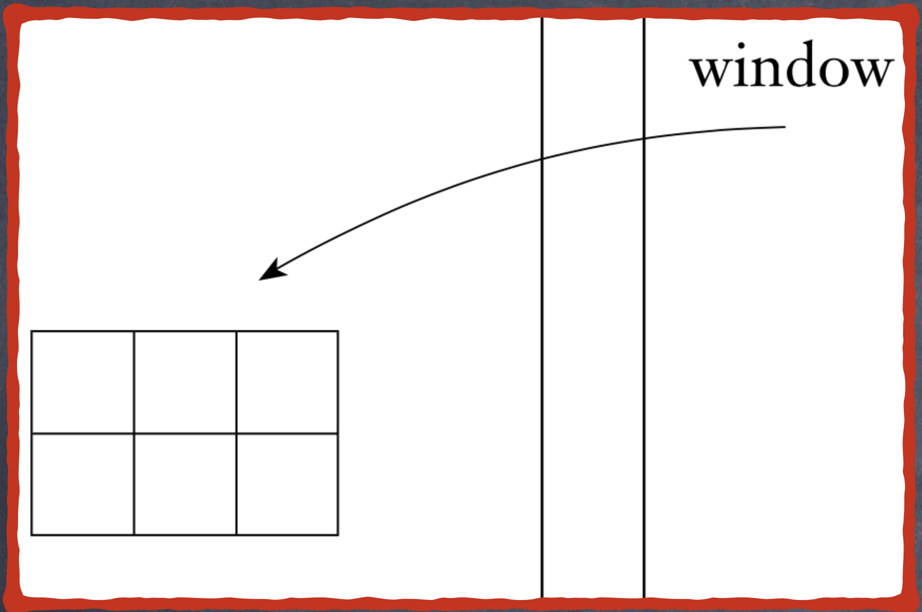
(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

FIGURE 7.39
Examples of legal windows



Cook-Levin Theorem: ϕ_{move}

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

$\boldsymbol{\delta}(q_1,b)=(q_1,c,L)$

FIGURE 7.40
Examples of illegal windows

Cook-Levin Theorem: ϕ_{move}

CLAIM 7.41

If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

Cook-Levin Theorem: ϕ_{move}

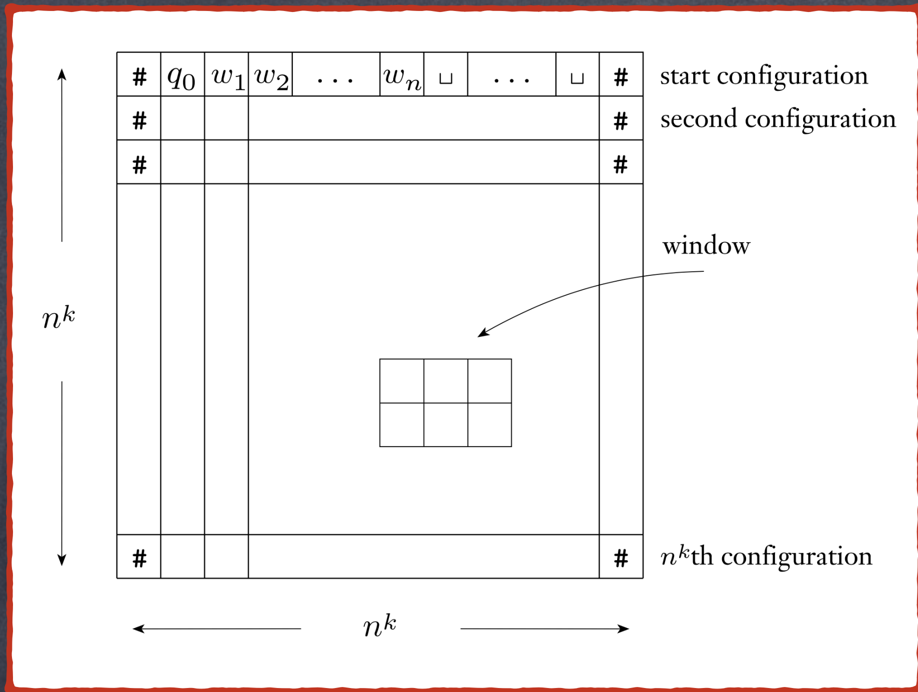
Now we return to the construction of ϕ_{move}. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula ϕ_{move} says that the settings of those six cells must be one of these ways, or

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \left(\text{the } (i,j) \text{ window is legal}\right)$$

Cook-Levin Theorem: ϕ_{move}

We replace the text "the (i, j) window is legal" in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1, \ldots, a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \right)$$



Cook-Levin Theorem

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.

$\langle \phi \rangle \in SAT$

iff

N accepts w within n^k steps.

NP-Complete Problems

3SAT is NP-Complete

literal is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A ***clause*** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in ***conjunctive normal form***, called a ***cnf-formula***, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

It is a ***3cnf-formula*** if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Let *3SAT* $= \{\langle\phi\rangle \mid \phi$ is a satisfiable 3cnf-formula$\}$. In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

3SAT is NP-Complete

COROLLARY 7.42

3SAT is NP-complete.

PROOF Obviously *3SAT* is in NP, so we only need to prove that all languages in NP reduce to *3SAT* in polynomial time. One way to do so is by showing that *SAT* polynomial time reduces to *3SAT*. Instead, we modify the proof of Theorem 7.37 so that it directly produces a formula in conjunctive normal form with three literals per clause.

Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right].$$

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} .\end{aligned}$$

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}} .$$

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

$$\phi_{\text{cell}} \;\; \phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\; 1<j<n^k} \text{(the } (i,j) \text{ window is legal)} \;\; \text{;})) \Big] .$$

Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \text{(the } (i,j) \text{ window is legal)}$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

3SAT is NP-Complete

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} (\text{the } (i,j) \text{ window is legal})$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} (x_{i,j-} \ldots {}_{i5} \wedge x_{i+1,j+1,a_6})$$

- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

Now that we have written the formula in cnf, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original.

3SAT is NP-Complete

For example, we replace clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, wherein z is a new variable. If some setting of the a_i's satisfies the original clause, we can find some setting of z so that the two new clauses are satisfied. In general, if the clause contains l literals,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

we can replace it with the $l - 2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

We may easily verify that the new formula is satisfiable iff the original formula was, so the proof is complete.

CLIQUE is NP-Complete

THEOREM 7.32

3SAT is polynomial time reducible to *CLIQUE*.

PROOF IDEA The polynomial time reduction f that we demonstrate from *3SAT* to *CLIQUE* converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.

CLIQUE is NP-Complete

PROOF Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \ \wedge \ (a_2 \vee b_2 \vee c_2) \ \wedge \quad \cdots \quad \wedge \ (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

The nodes in G are organized into k groups of three nodes each called the ***triples***, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$. The following figure illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \ \wedge \ (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \ \wedge \ (\overline{x_1} \vee x_2 \vee x_2)$.

CLIQUE is NP-Complete

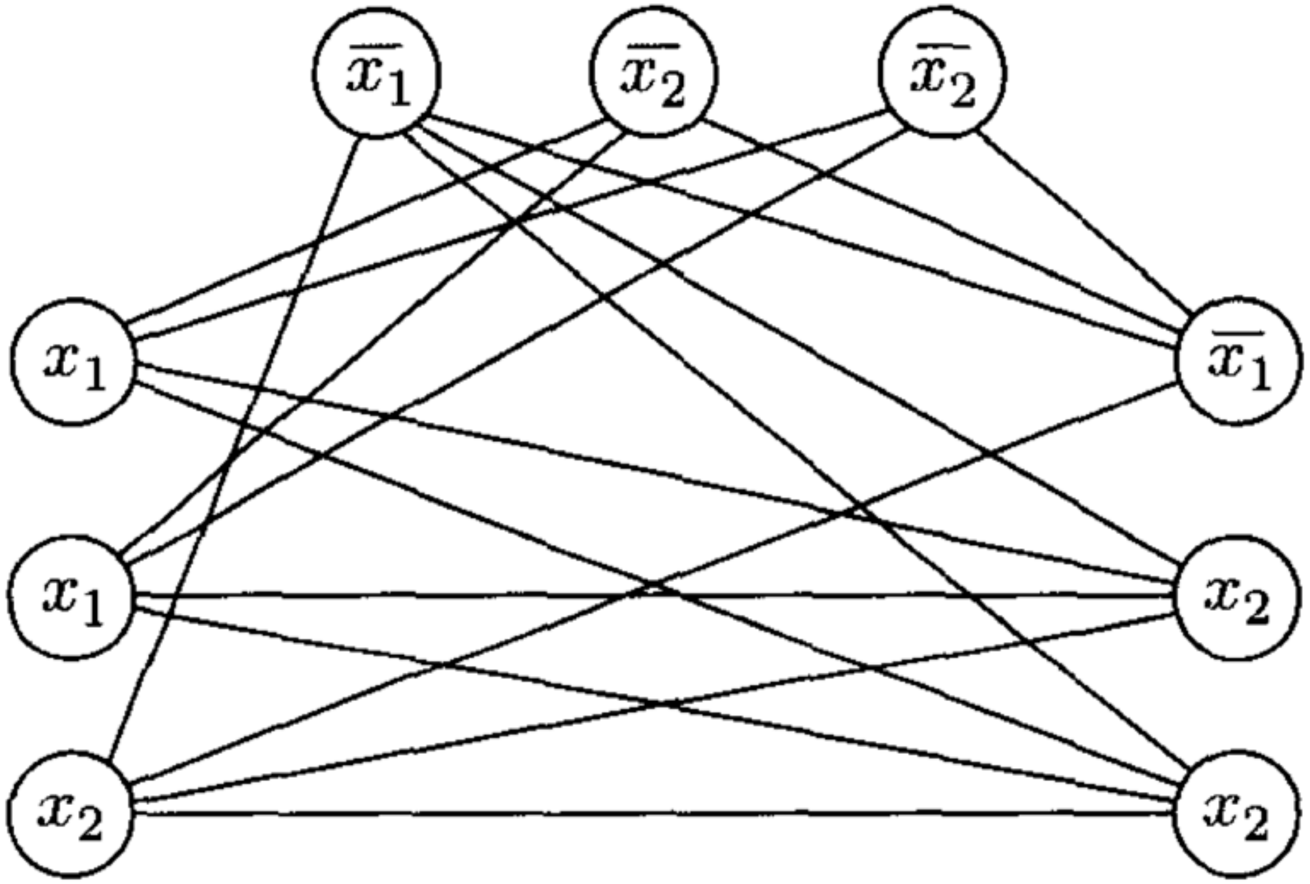


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

CLIQUE ∈ NP-Complete: ⟨ϕ⟩∈3SAT -> ⟨G,k⟩∈CLIQUE

Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a k-clique. The number of nodes selected is k, because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore G contains a k-clique.

CLIQUE ∈ NP-Complete: ⟨G,k⟩∈CLIQUE -> ⟨ϕ⟩∈3SAT

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal that is assigned TRUE. Therefore ϕ is satisfiable.

Vertex-Cover is NP-Complete

THE VERTEX COVER PROBLEM

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

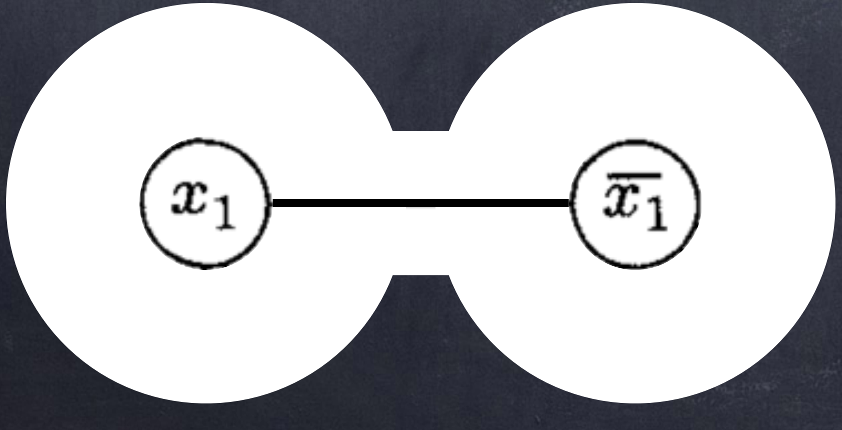
$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

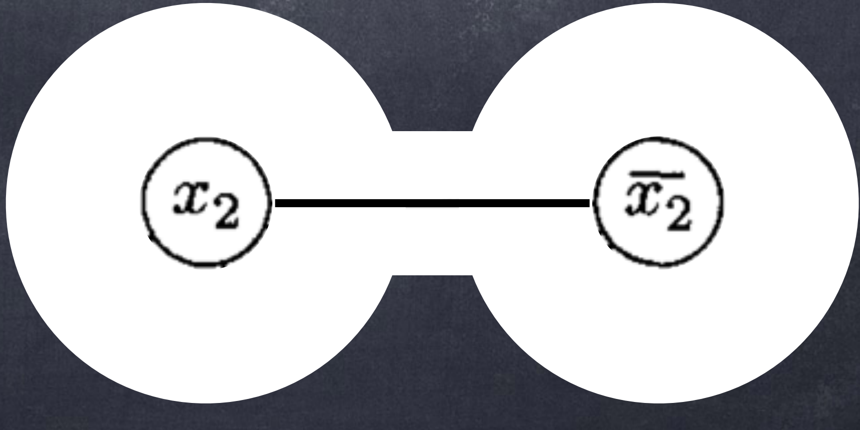
THEOREM 7.44

VERTEX-COVER is NP-complete.

Vertex-Cover is NP-Complete

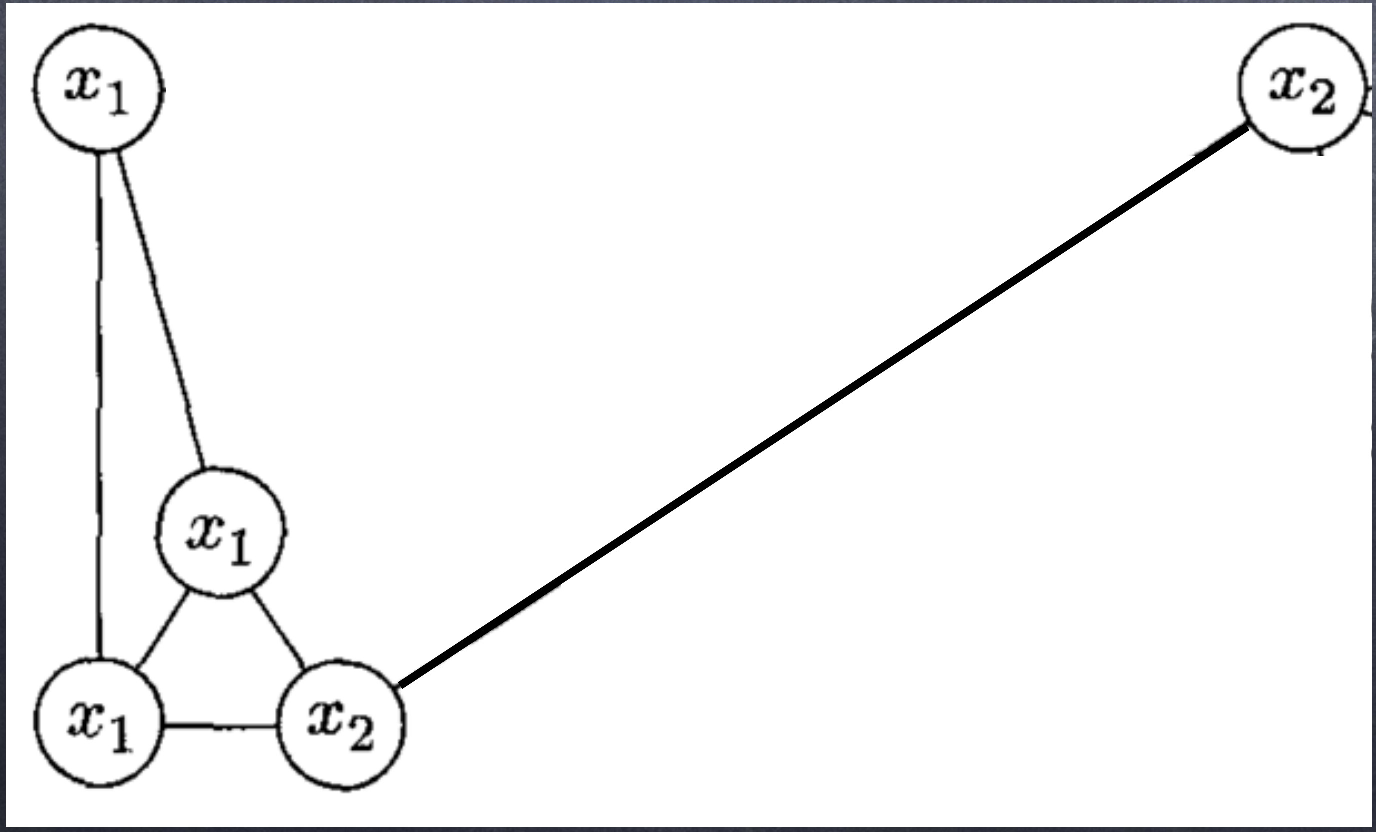
PROOF Here are the details of a reduction from *3SAT* to *VERTEX-COVER* that operates in polynomial time. The reduction maps a Boolean formula ϕ to a graph G and a value k. For each variable x in ϕ, we produce an edge connecting two nodes. We label the two nodes in this gadget x and $\overline{x}$. Setting x to be TRUE corresponds to selecting the left node for the vertex cover, whereas FALSE corresponds to the right node.





Vertex-Cover is NP-Complete

The gadgets for the clauses are a bit more complex. Each clause gadget is a triple of three nodes that are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variables gadgets that have the identical labels. Thus the total number of nodes that appear in G is $2m + 3l$, where ϕ has m variables and l clauses. Let k be $m + 2l$.



For example, if $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, the reduction produces $\langle G, k \rangle$ from ϕ, where $k = 8$ and G takes the form shown in the following figure.

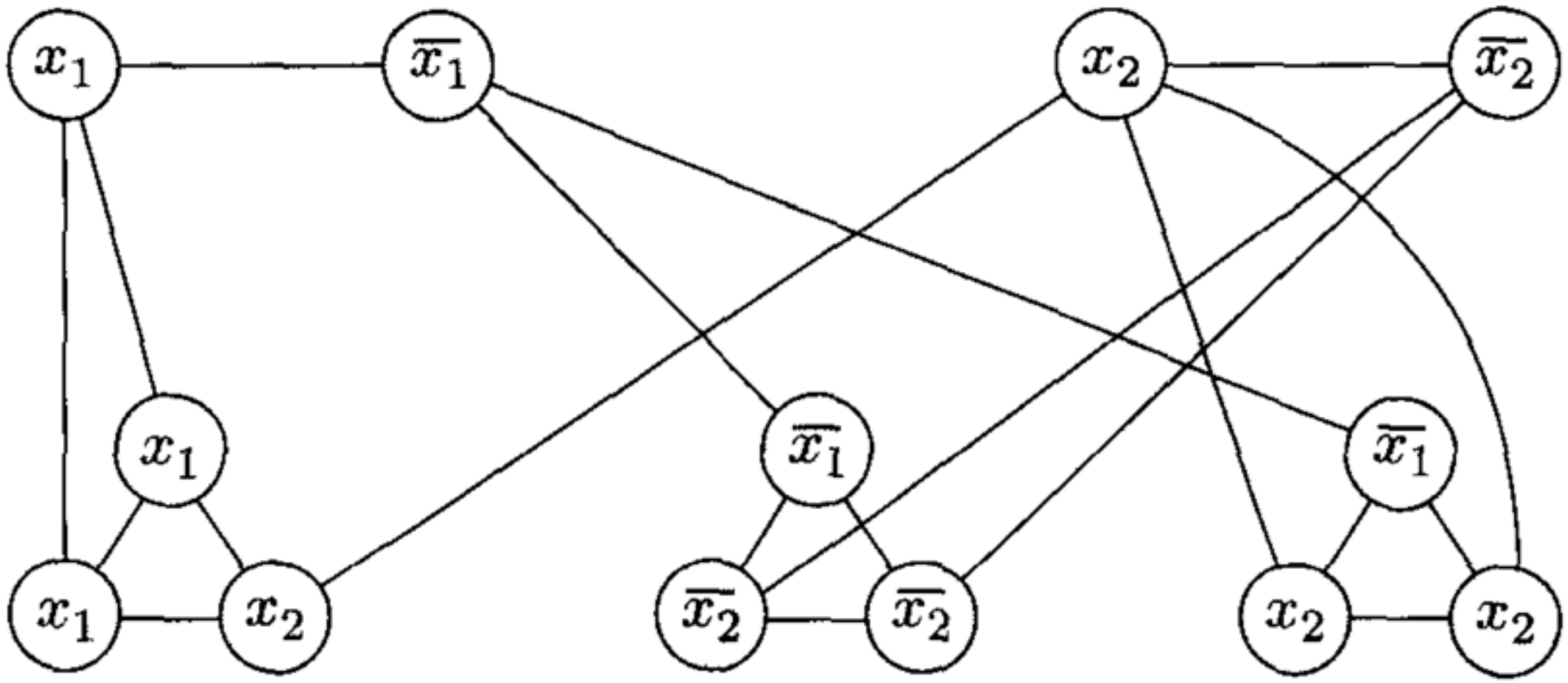
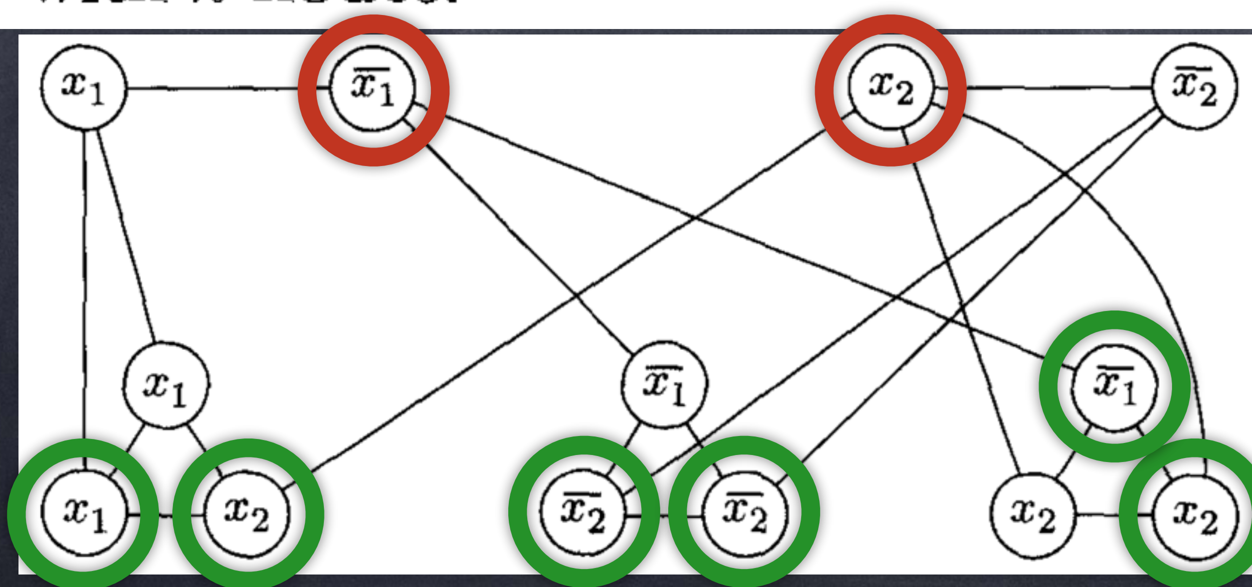


FIGURE 7.45
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Vertex-Cover ∈ NP-Complete: ⟨ϕ⟩∈ 3SAT -> ⟨G,k⟩∈ V-C

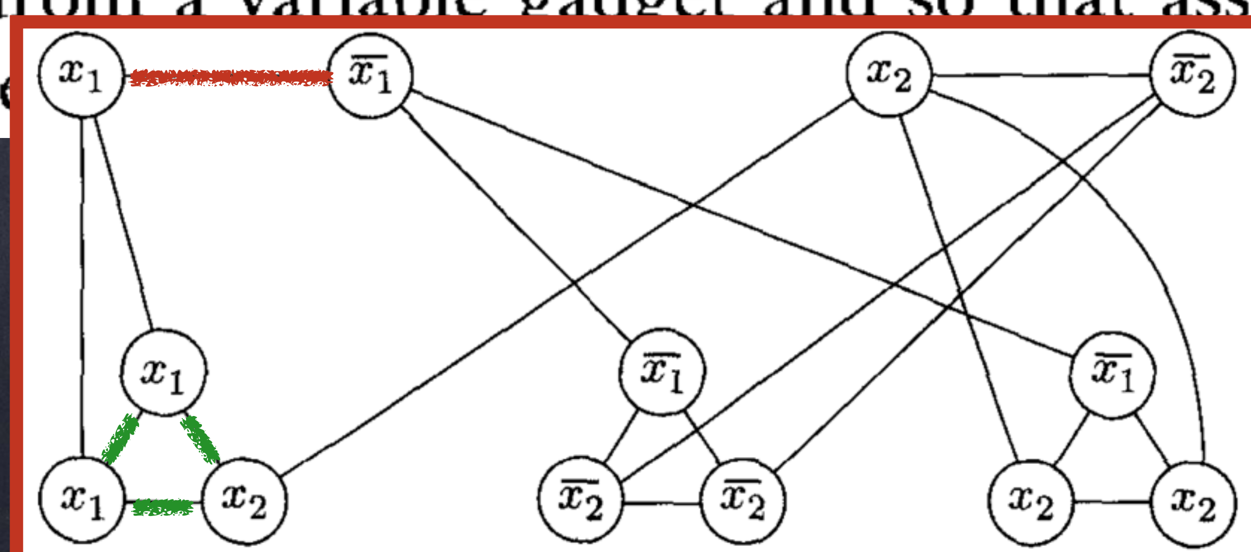
To prove that this reduction works, we need to show that ϕ is satisfiable if and only if G has a vertex cover with k nodes. We start with a satisfying assignment. We first put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. Now, we have a total of k nodes. They cover all edges because every variable gadget edge is clearly covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered. Hence G has a vertex cover with k nodes.



Vertex-Cover ∈ NP-Complete:

⟨G,k⟩ ∈ V-C -> ⟨ϕ⟩ ∈ 3SAT

Second, if G has a vertex cover with k nodes, we show that ϕ is satisfiable by constructing the satisfying assignment. The vertex cover must contain one node in each variable gadget and two in every clause gadget in order to cover the edges of the variable gadgets and the three edges within the clause gadgets. That accounts for all the nodes, so none are left over. We take the nodes of the variable gadgets that are in the vertex cover and assign the corresponding literals TRUE. That assignment satisfies ϕ because each of the three edges connecting the variable gadgets with each clause gadget is covered and only two nodes of the clause gadget are in the vertex cover. Therefore one of the edges must be covered by a node from a variable gadget and so that assignment satisfies the corresponding clause



Beyond NP-Completeness

Beyond NP-Completeness

- PSpace Completeness: problems that require a reasonable (Poly) amount of space to be solved but may use very long time though.
- Many such problems. If any of them may be solved within reasonable (Poly) amount of time, then all of them can.

Beyond NP-Completeness

DEFINITION 8.1

Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f\colon \mathcal{N}\longrightarrow\mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

Space Complexity

DEFINITION 8.2

Let $f\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. The ***space complexity classes***, **SPACE($f(n)$)** and **NSPACE($f(n)$)**, are defined as follows.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Turing machine}\}.$$

THEOREM 8.5

Savitch's theorem For any[1] function $f\colon \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

Space Complexity

DEFINITION 8.6

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes. However, PSPACE = NPSPACE by virtue of Savitch's theorem, because the square of any polynomial is still a polynomial.

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$

Space/Time Complexity

NP = PSpace ?

Space/Time Complexity

PSpace=EXPTime ?

Space/Time Complexity

P≠EXPTime

Space Complexity

DEFINITION 8.8

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

PSpace Completeness

- Geography Game:

  Given a set of country names: Aruba, Cuba, Canada, Equador, France, Italy, Japan, Korea, Nigeria, Russia, Vietnam, Yemen.

- A two player game: One player chooses a name and crosses it out. The other player must choose a name that starts with the last letter of the previous name and so on. A player wins when his opponent cannot play any name.

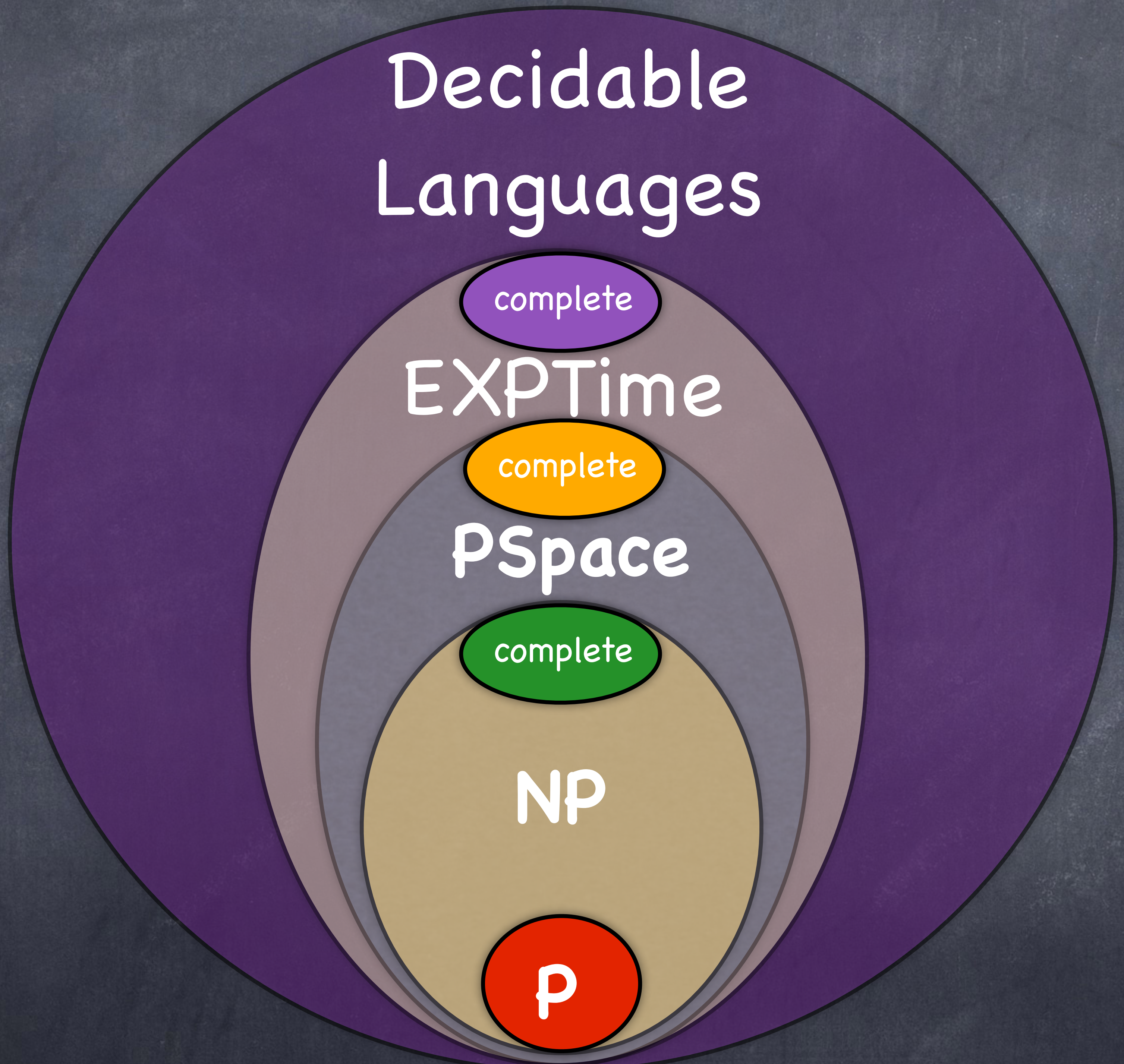
Generalized Geography

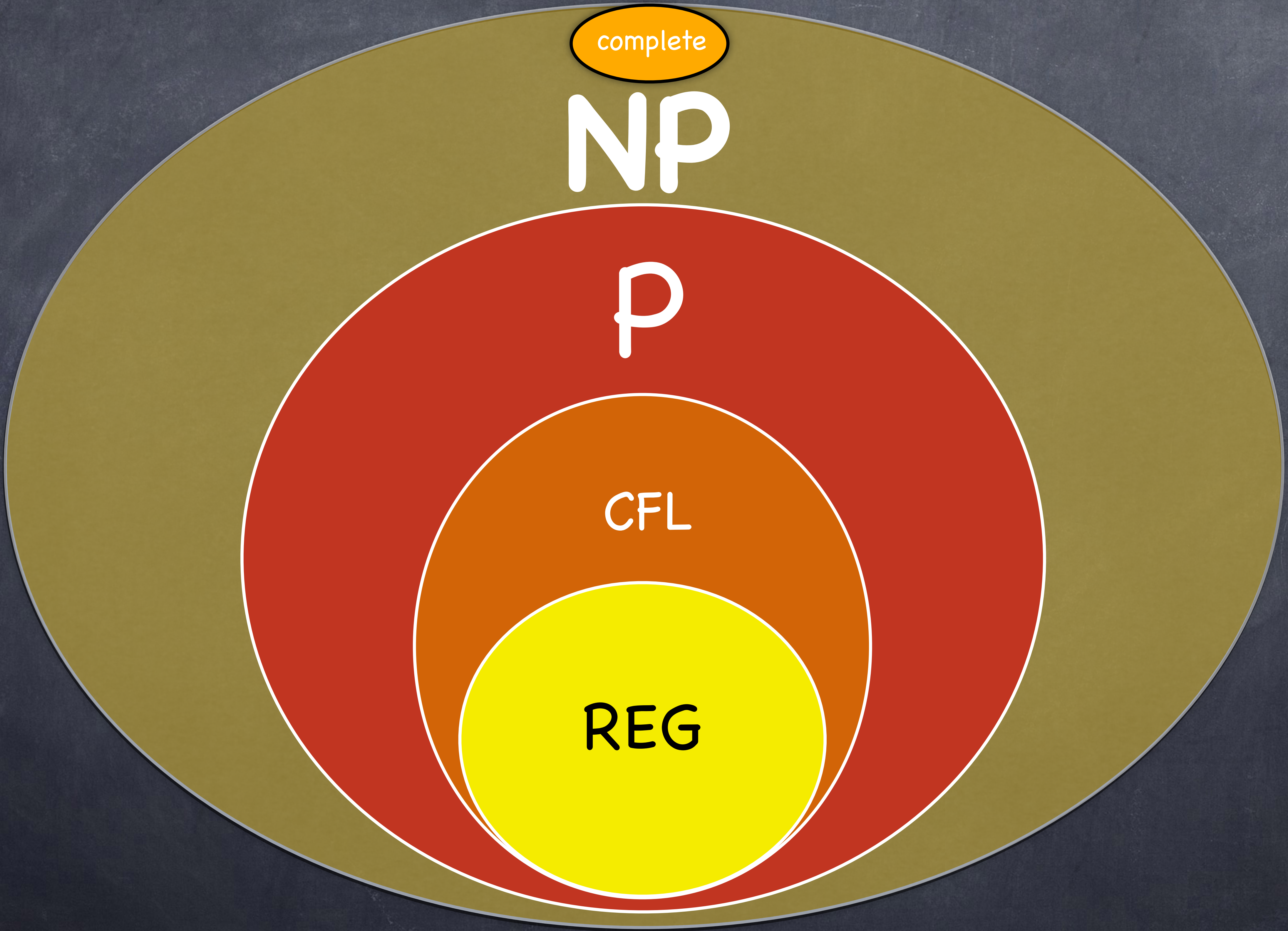
- Given an arbitrary set of names: $w_1, \ldots, w_n$.
- Is there a winning strategy for the first player to the previous game ?

Theoretical Computer Science

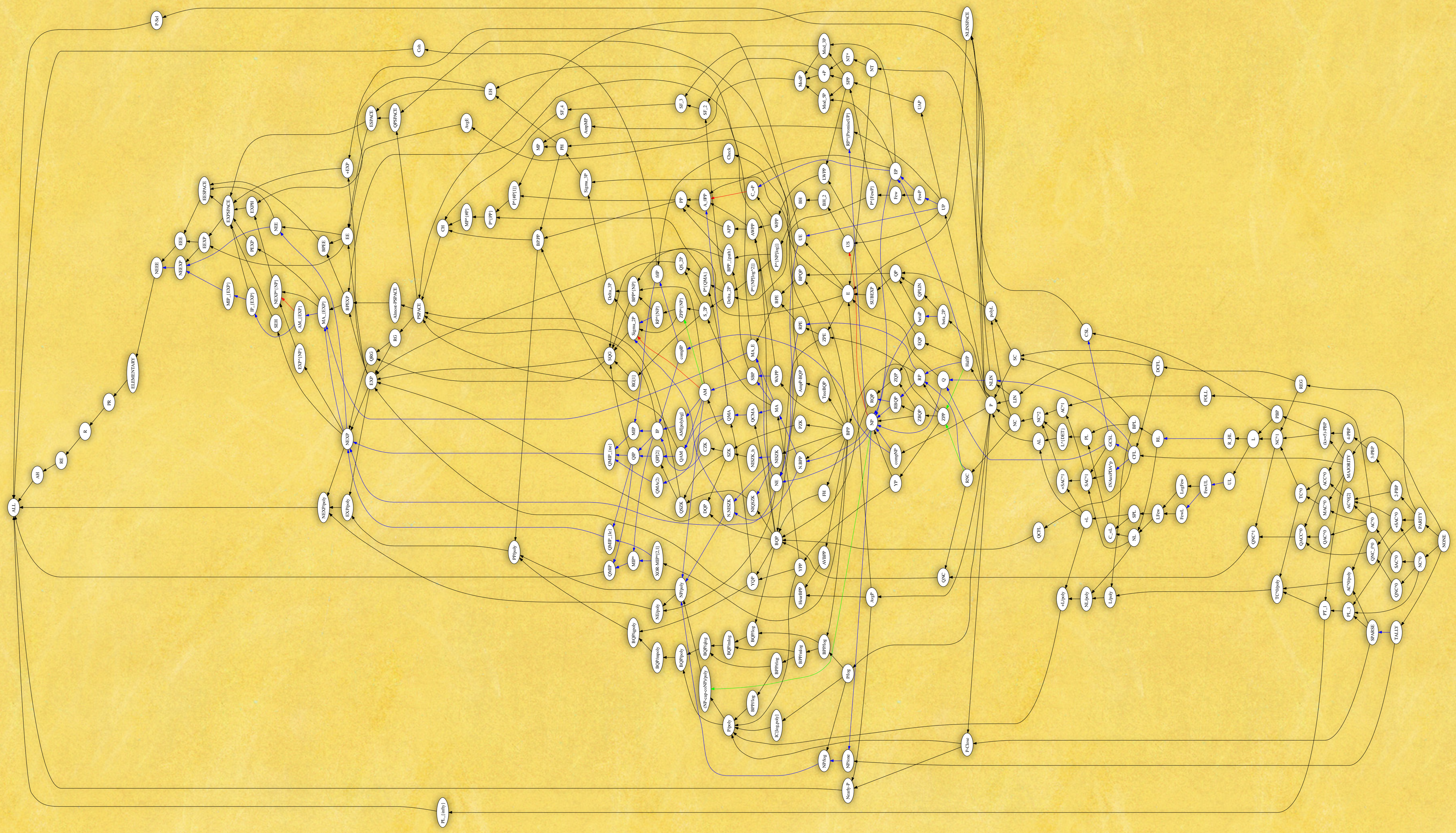
- Challenges of TCS:
- FIND efficient solutions to many problems. (Algorithms and Data Structures)
- PROVE that certain problems are NOT computable within a certain time or space.
- Consider new models of computation. (Such as a Quantum Computer)

Computability Theory


Decidable
Languages
complete
EXPTime
complete
PSpace
complete
NP
P

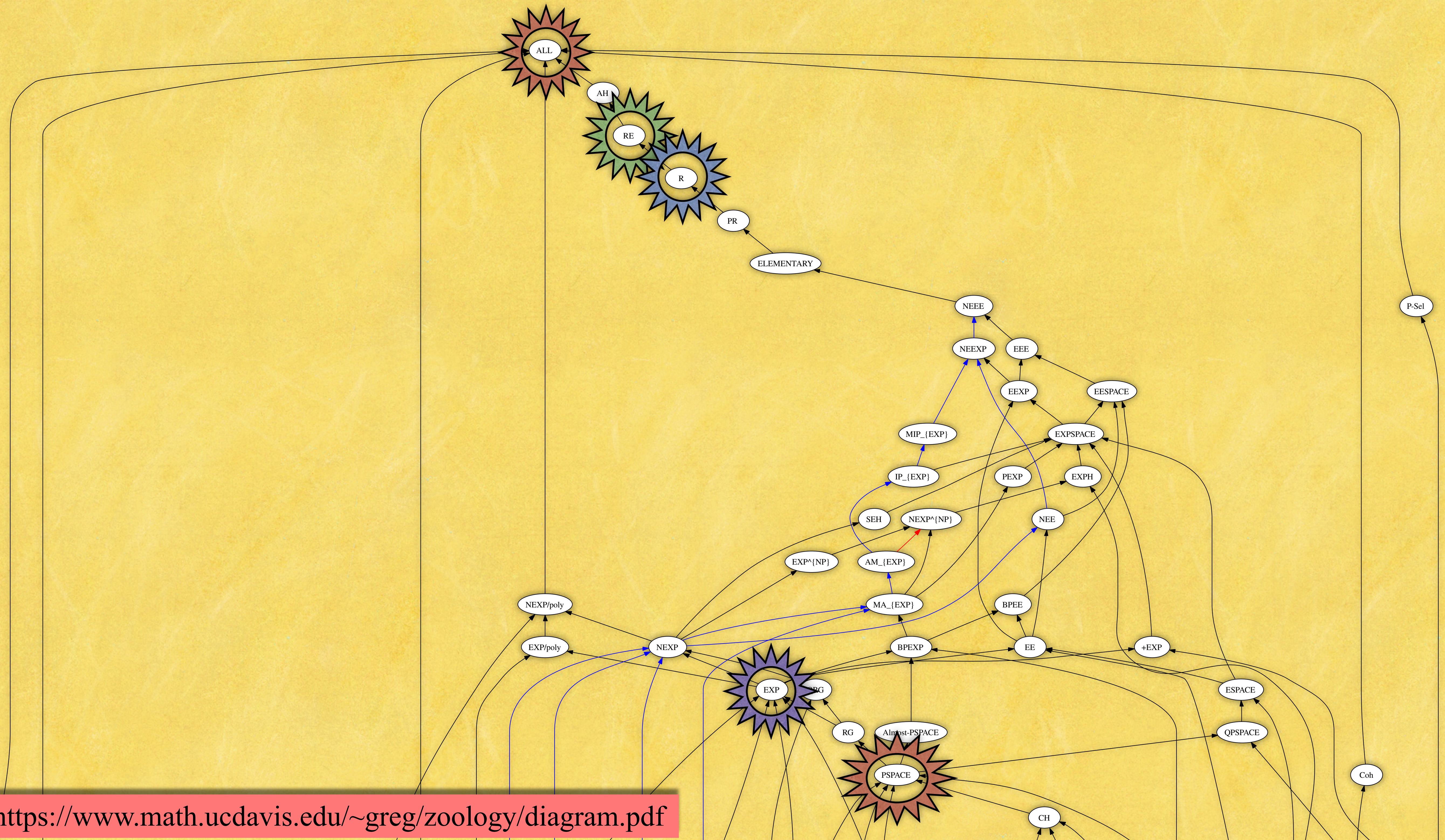

complete
NP
P
CFL
REG

Computability/Complexity



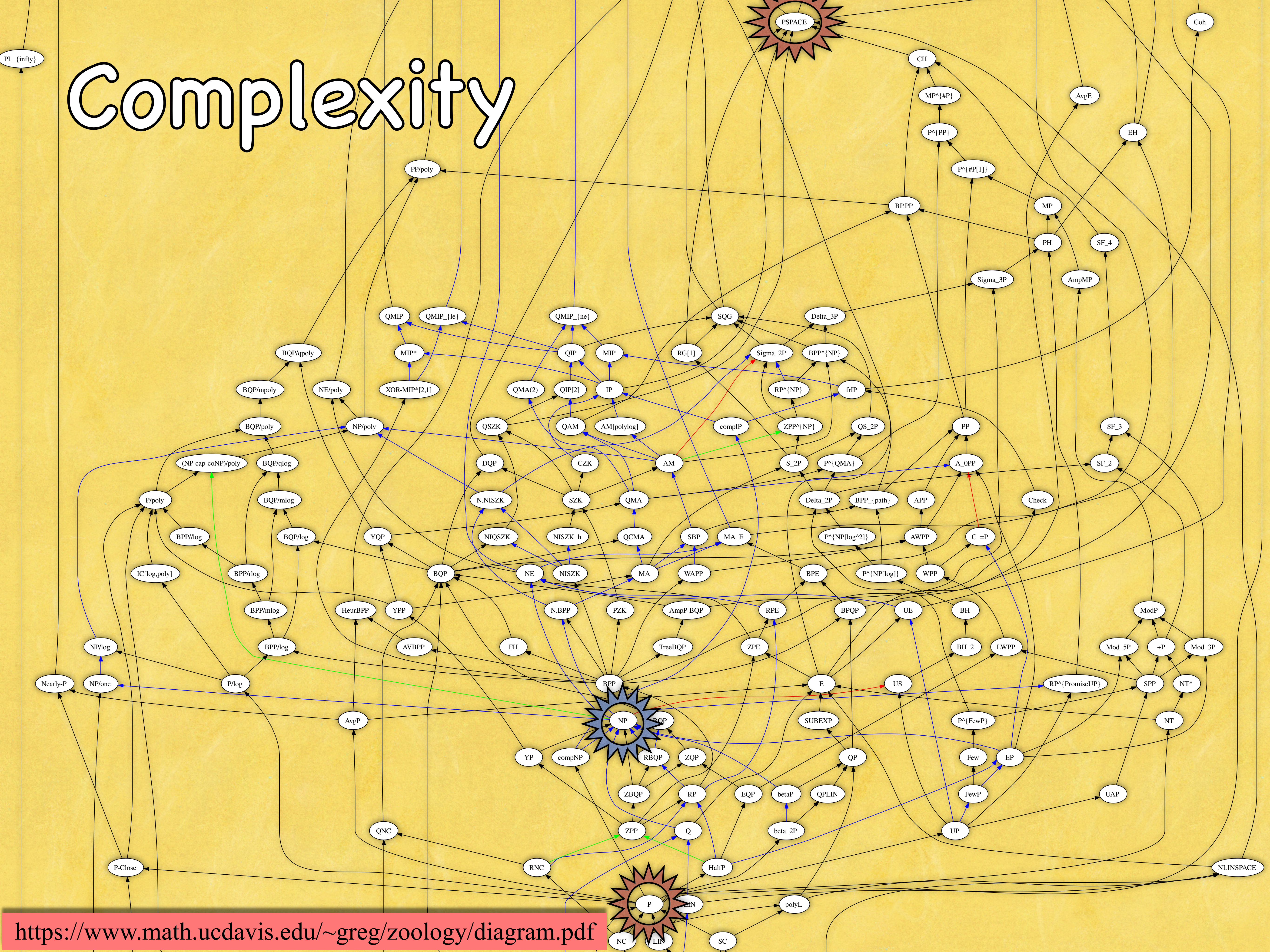
https://www.math.ucdavis.edu/~greg/zoology/diagram.pdf

Computability/Complexity

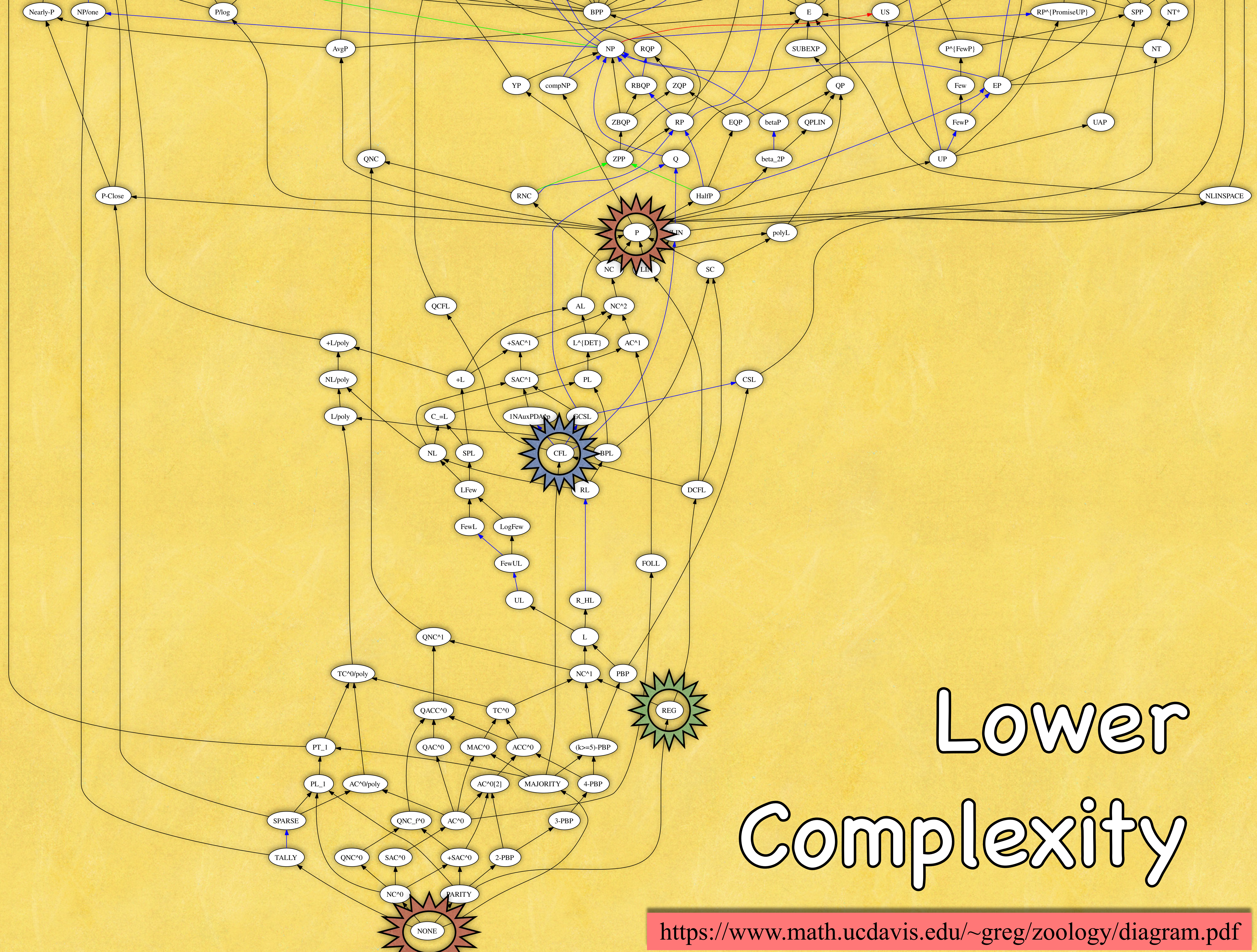


https://www.math.ucdavis.edu/~greg/zoology/diagram.pdf

Complexity



https://www.math.ucdavis.edu/~greg/zoology/diagram.pdf



Lower Complexity

https://www.math.ucdavis.edu/~greg/zoology/diagram.pdf

COMP-330
Theory of Computation

Fall 2019 -- Prof. Claude Crépeau

Part II : Lec. 10-23