Theory of Computation and Formal Languages

Jeena George
Department of Computer Application

Theory of Computation

Lecture 01

Introduction

Theory of Computation: areas

- Formal Language Theory
  - language = set of sentences (strings)
  - grammar = rules for generating strings
  - production/deduction systems
  - capabilities & limitations
  - application: programming language specification
- Automata Theory (Abstract Machines)
  - models for the computing process with various resources
  - characterize ``computable”
  - capabilities & limitations
  - application: parsing algorithms
- Complexity Theory
  - inherent difficulty of problems (upper and lower bounds)
  - time/space resources
  - intractable or unsolvable problems
  - application: algorithm design

Major Themes

- Study models for
  - problems
  - machines
  - languages
- Classify
  - machines by their use of memory
  - grammars and languages by type
  - languages by class hierarchies
  - problems by their use of resources (time, space, …)
- Develop relationships between models
  - *reduce* solution of one problem to solution of another
  - *simulate* one machine by another
  - *characterize* grammar type by machine recognizer

Describing Problems

- Problems described by **functions**
  - functions assign ouputs to inputs
  - described by tables, formulae, circuits, math logic, algorithms

Example: TRAVELLING SALESPERSON Problem

Input: *n(n-1)/2* distances between *n cities*

Output: order to visit cities that gives *shortest tour* (or length of tour)

- Problems described by **languages**
  - ``yes/no'' problems or *decision problems*
  - a language represents a *problem or question*
  - input strings *in* the language: answer is ``yes''; *else* ``*no*''

Example: HALTING Problem

Input: a string w and a Turing Machine M

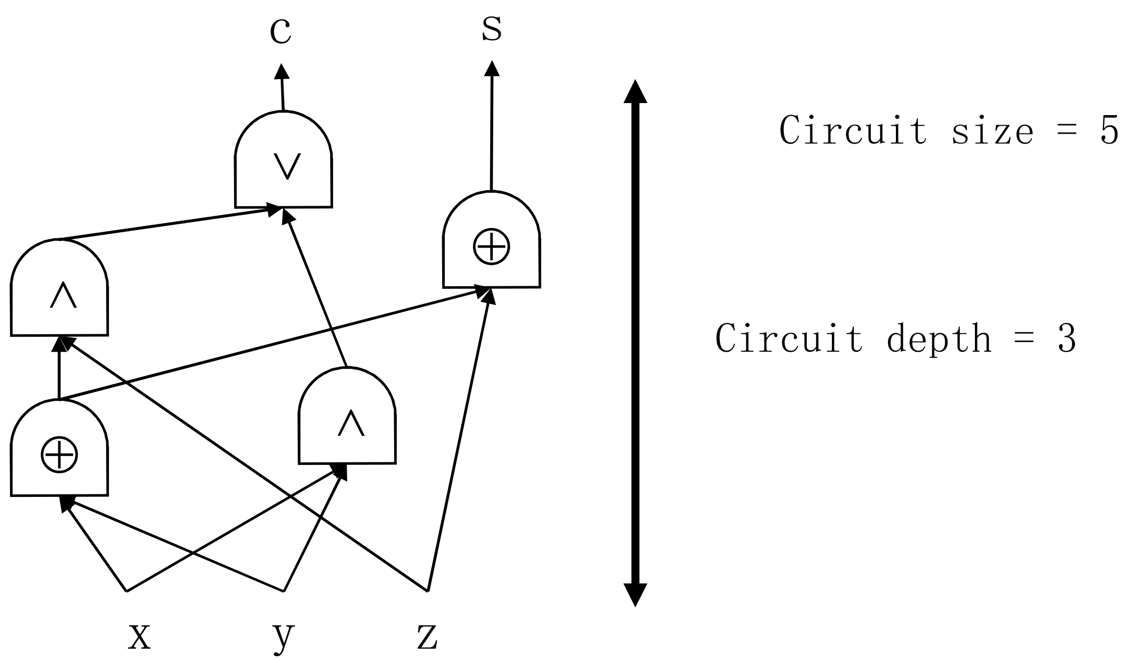
Output: ``yes" if $(e, w) \in L_H$; ``no" otherwise

- equivalent to asking if string $w \in L(M)$

where $L_H = \{(e, w) \mid \text{Turing Machine } M_e \text{ halts on input } w\}$

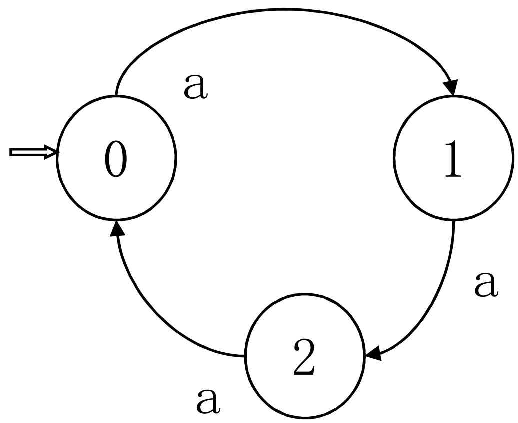
Types of Machines

- Logic circuit
  - memoryless; values combined using gates



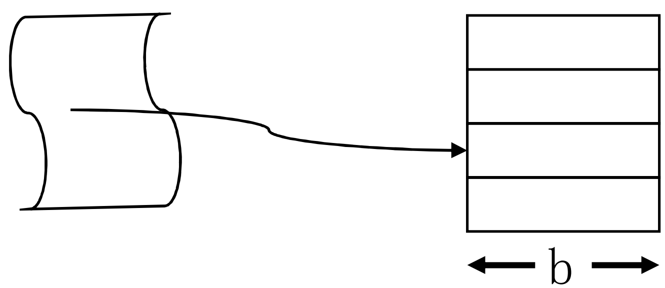
Types of Machines (cont.)

- Finite-state automaton (FSA)
  - bounded number of memory states
  - step: input, current state determines next state & output



Mod 3 counter
state/ouput (Moore) machine

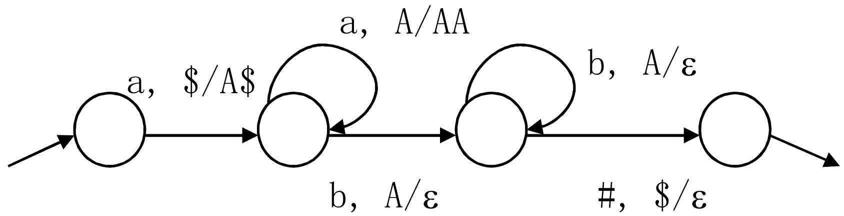
- models programs with a *finite* number of *bounded* registers
  - reducible to 0 registers



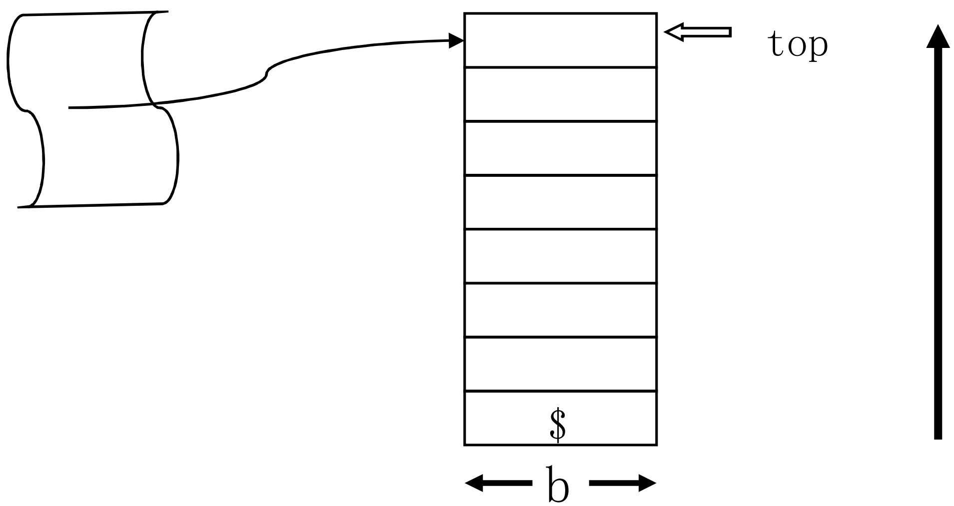
Types of Machines (cont.)

- Pushdown Automaton (PDA)
  - finite control and a single *unbounded* stack

$L = \{a^n b^n \# : n \geq 1\}$



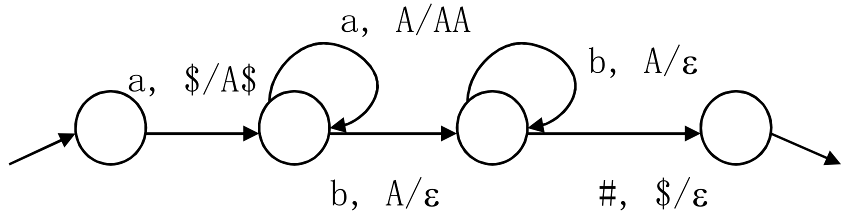
  - models finite program + one *unbounded* stack of *bounded* register



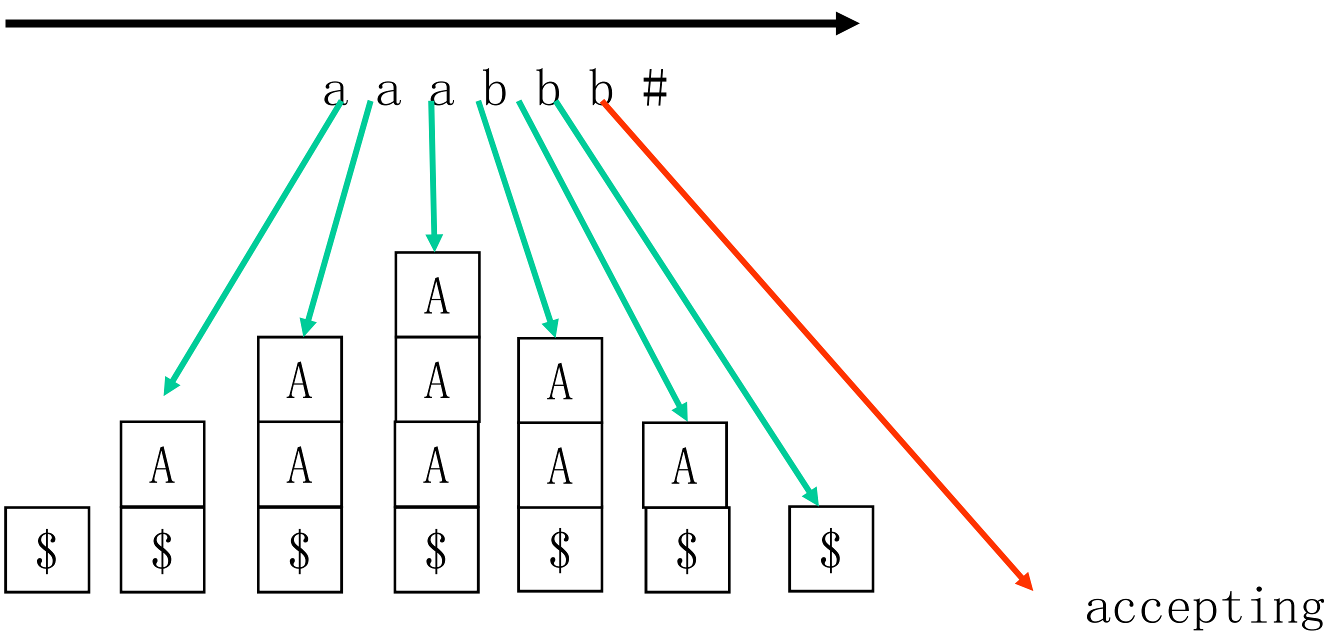
Types of Machines (cont.)

- Pushdown Automaton (PDA)
  - finite control and a single *unbounded* stack

$L = \{a^n b^n \# : n \geq 1\}$



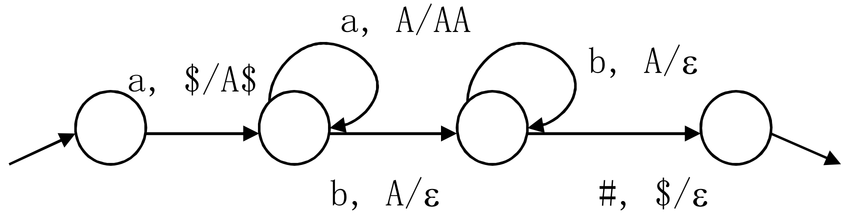
  - models finite program + one *unbounded* stack of *bounded* register



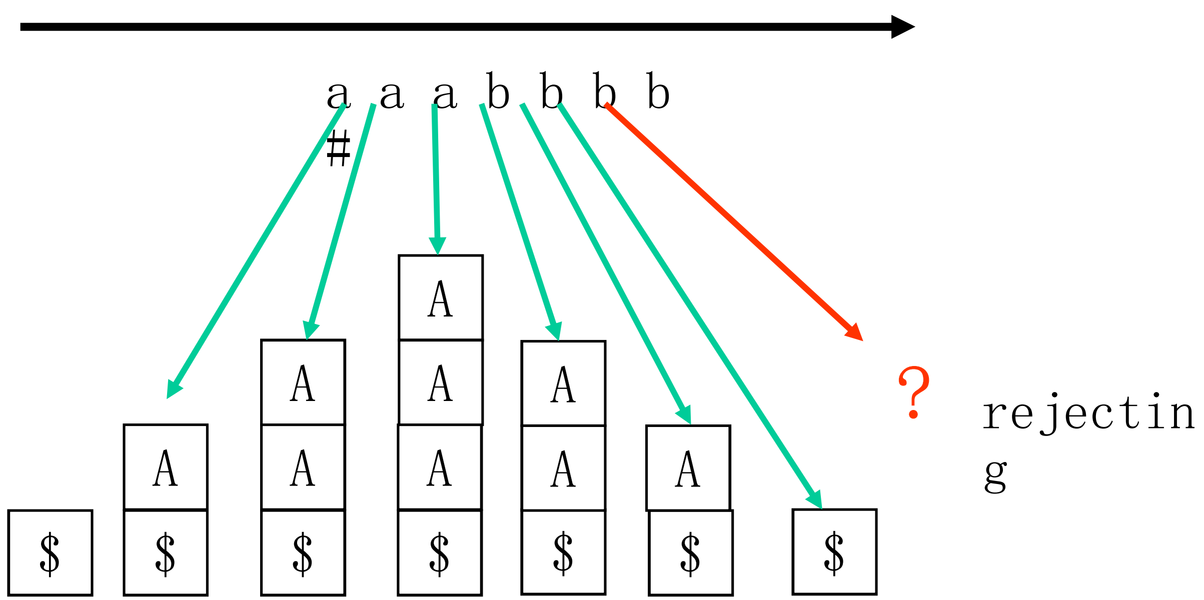
Types of Machines (cont.)

- Pushdown Automaton (PDA)
  - finite control and a single *unbounded* stack

$L = \{a^n b^n \# : n \geq 1\}$



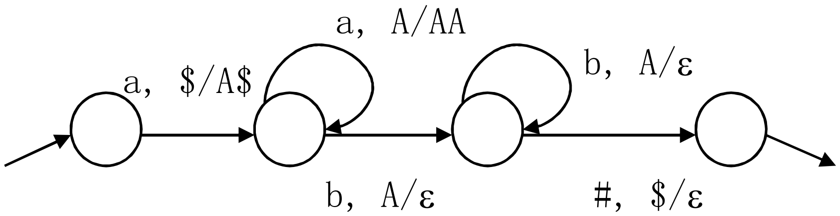
  - models finite program + one *unbounded* stack of *bounded* register



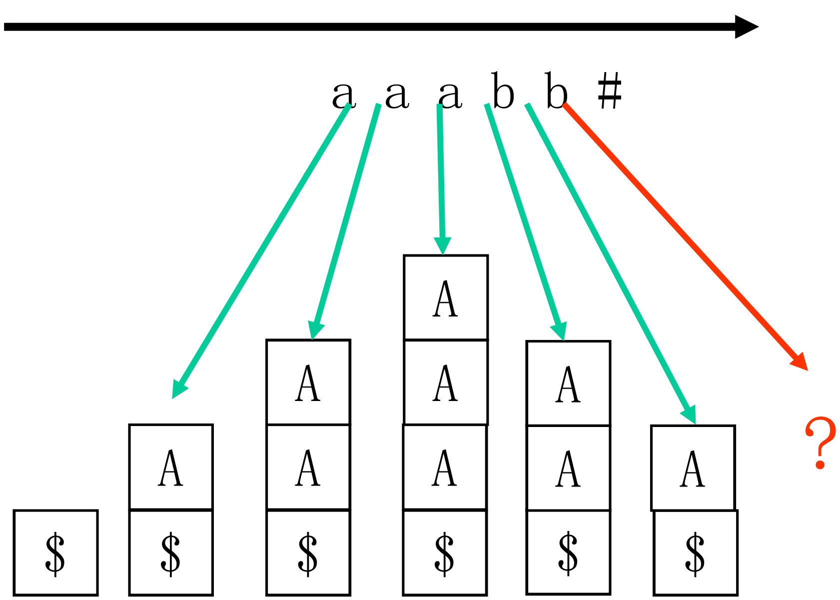
Types of Machines (cont.)

- Pushdown Automaton (PDA)
  - finite control and a single *unbounded* stack

$L = \{a^n b^n \#: n \geq 1\}$



  - models finite program + one *unbounded* stack of *bounded* register



rejectin
g

Types of Machines (cont.)

- Random access machine (RAM)
  - finite program and an *unbounded, addressable* random access memory of ``registers"
  - models general programs
    - unbounded # of bounded registers

Example:

$$\frac{R_0 \leftarrow R_0 + R_1}{}$$

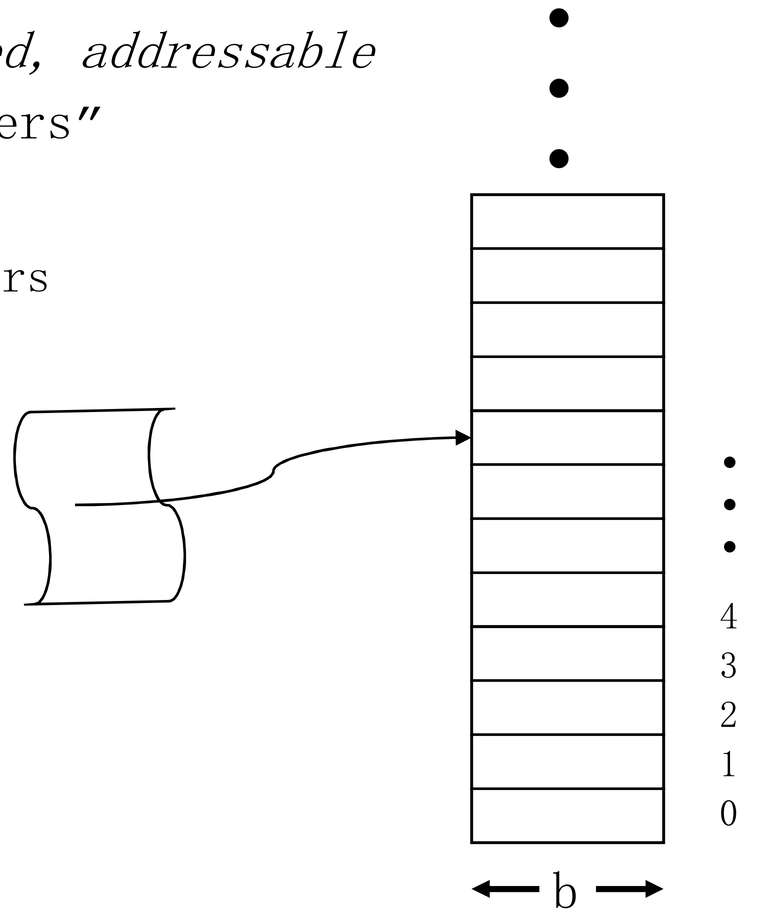
$L_0 : JMPZ\ R_1\ L_1$

$\quad INC\ R_0$

$\quad DEC\ R_1$

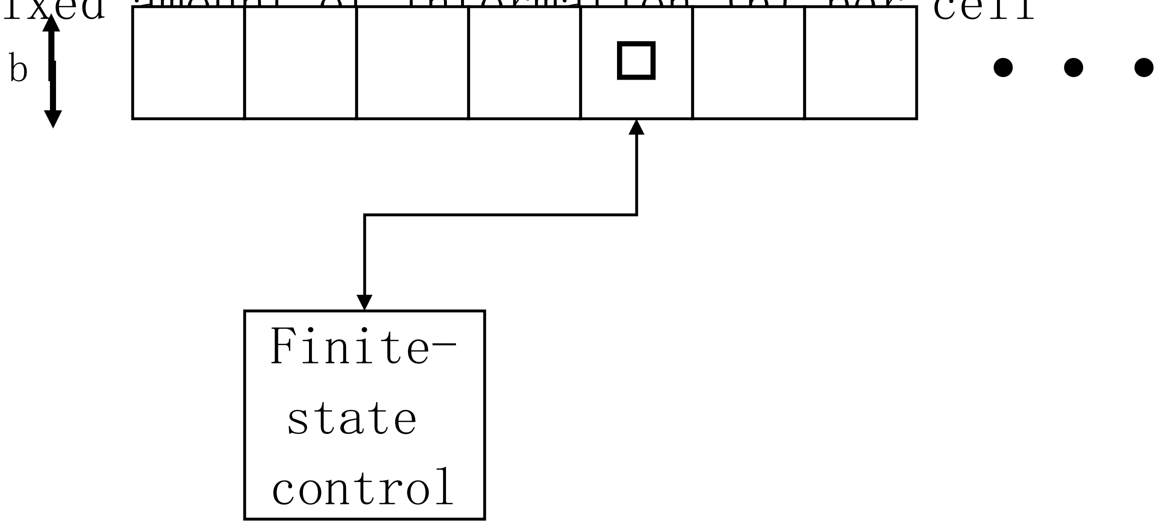
$\quad JMP\ L_0$

$L_1 : CONTINUE$



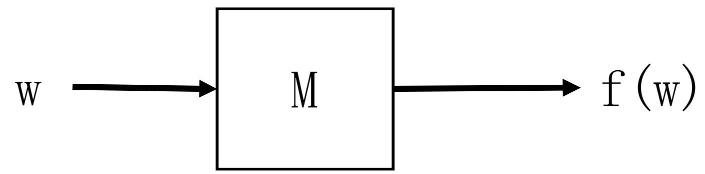
Types of Machines (cont.)

- Turing Machine (TM)
  - finite control & tape of *bounded cells* unbounded in # to R
  - current state, cell scanned determine next state & overprint symbol
  - control writes over symbol in cell and moves head 1 cell L or R
  - models simple ``sequential'' memory; no addressability
  - fixed amount of information (b) per cell

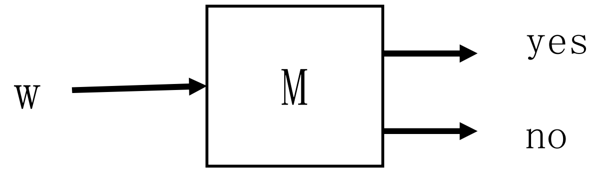


How Machines are Used

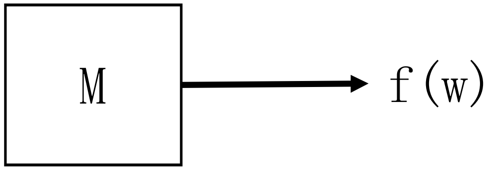
- To specify *functions or sets*
  - Transducer – maps string to string



  - Acceptor – ``recognizes" or ``accepts'' a set of strings



    - M *accepts* strings which cause it to enter a final state
  - Generator – ``generates" a set of strings

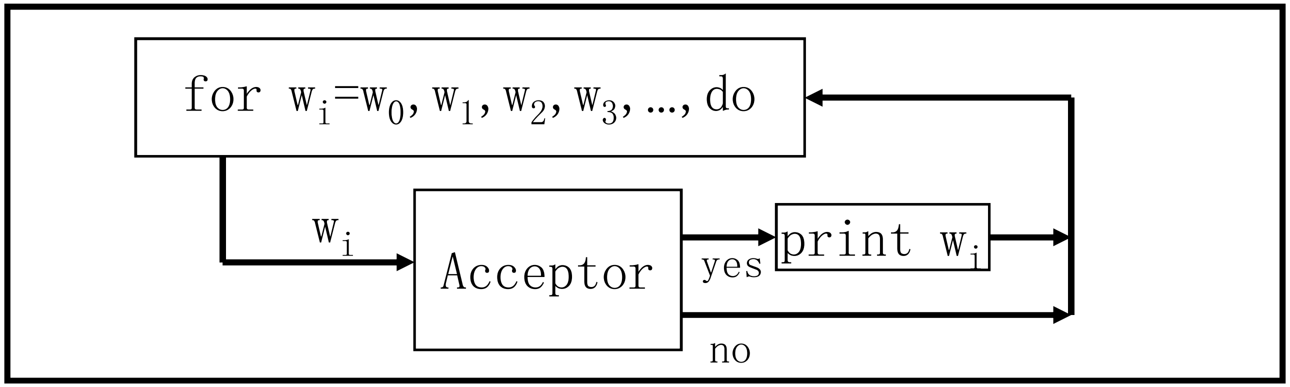


    - M generates all values generated during computation

How Machines are Used

The equivalence between acceptor and generator

- Using an acceptor to implement a generator

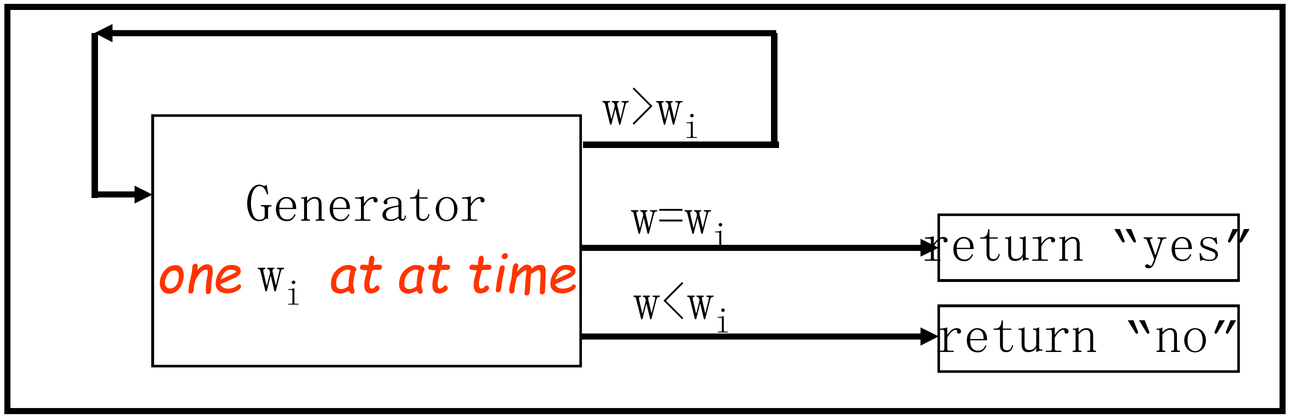



```
#include “acceptor.h”

generator() {

  for wi=w0, w1, w2, w3, …, do if accept(wi)
print(wi);

}
```

How Machines are Used

The equivalence between acceptor and generator

- Using a generator to implement an acceptor




```
#include "generator.h"

acceptor(w) {

  while (wi=generator) ≠ null, do

    if w=wi, return "yes"; else if w<wi,
return "no";

}
```

Types of Language

- A *language* is a set of strings over an alphabet

$$L_1 = \{ a^n b^n c^n \mid n \geq 0 \}$$

$$L_2 = \{ a^i b^j c^k \mid i + j = k \ \& \ i, j, k \geq 0 \}$$

- *Grammars* are rules for generating a set of strings

$$G_1 : S \rightarrow aBSc \mid abc \mid \varepsilon \qquad S \Rightarrow aBSc \Rightarrow aBaBScc \Rightarrow aBaBabccc \Rightarrow aBaaBbccc \Rightarrow aaBaBbccc$$

$$Ba \rightarrow aB \qquad \Rightarrow aaBabbccc \Rightarrow aaaBbbccc \Rightarrow aaabbbccc$$

$$Bb \rightarrow bb$$

$$G_2 : S \rightarrow aSc \mid T \mid \varepsilon$$

$$T \rightarrow bTc \mid \varepsilon$$

- Machines can *accept* languages (sets)

Types of Language (cont.)

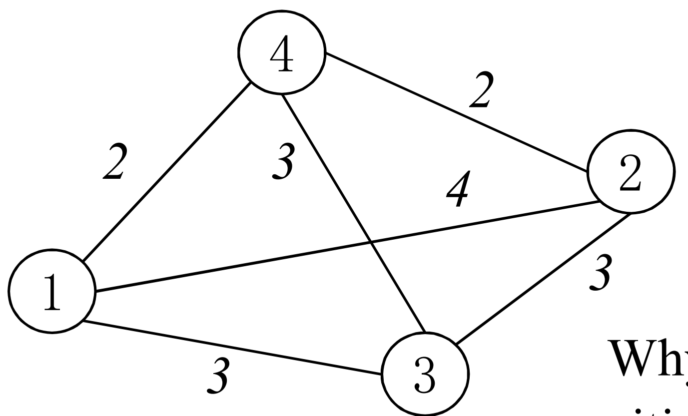
- Languages can represent complex ``problems"

Example: Traveling Salesperson Language (TSL)

strings describe *instances* of TSP having tours of length $\leq k$

here is one string in TSL when $n=4$:

$$(d_{12}, d_{13}, d_{14}, d_{23}, d_{24}, d_{34} \ ; \ k) = (4,3,2,3,2,3 \ ; \ 11)$$



Why?

cities visited in order

1,4,2,3,1 $\Rightarrow$ tour distance = 10

Types of Language (cont.)

The equivalence of language recogintion and problem solving

- An algorithm is known that will accept length *n* strings in TSL in time *exponential* in *n* -- *BUT* none known for polynomial time
- TSL is an NP-*complete* language:
  - NP is an acronym for *recognizable in nondeterministic polynomial time*
  - NP-complete languages are ``maximally hard" among all in NP
  - the best known recognition algorithms need time exponential in *n*
  - all NP-complete languages either <u>require</u> exponenital time OR can be done efficiently in poly-time, *but we do not (yet) know which!*
  - we will use *reduction* to show that a language is NP-complete; major subject later on

Languages: Characterize Grammar by Machine

Language	*Grammar*		*Automaton*
Computably Enumerable (c.e.) or Recursively Enumerable (r.e.)	type 0 (unrestricted) $\alpha \rightarrow \beta$ □ $\alpha \neq \varepsilon$	⟷	TM (det. or non-det.)
∪	∪		∪
Context-Free (CFL)	type 2 $A \rightarrow \beta$ CFG	⟷	*non*-det. PDA
∪	∪		∪
Regular (Reg)	type 3 $A \rightarrow aB$ $A \rightarrow a$ right-linear	⟷	FSA (det. or non-

∪ = strict inclusion

⟷ = 2-way conversion algorithm

"Chomsky Hierarchy"

Classifying Problems by Resources

- What is the smallest *size* circuit (fewest gates) to add two binary numbers?
- What is the smallest *depth* circuit for binary addition? (low depth $\Rightarrow$ fast)
- can a FSA be used to recognize all binary strings with = numbers of 1s & 0s? Must a stronger model be used?
- How quickly can we determine if strings of length n are in TSL?
- How much space is needed to decide if boolean formulae of length n are satisfiable?

Relationships between problems

Reduction

- the most powerful technique available to compare the complexity of two problems
- reducing the solution of problem A to the solution of problem $B \Rightarrow$ *A is no harder than B* (written $A \leq B$)
  - *The translation procedure should be no harder than the complexity of A.*

Example: *squaring ≤ multiplication: suppose we have an algorithm **mult(x,y)** that will multiply two integers. Then **square(x)** = **mult(x,x)*** [trivial reduction]

Relationships between problems

Reduction

- the most powerful technique available to compare the complexity of two problems
- reducing the solution of problem A to the solution of problem $B \Rightarrow$ *A is no harder than B* (written $A \leq B$)
  - *The translation procedure should be no harder than the complexity of A.*

Example: *2-PARTITION* $\leq$ *MAKESPAN SCHEDULING*

2-PARTITION: Given a set H of natural numbers, is there a subset H' of H such that $\Sigma_{a \in (H-H')} a = \Sigma_{a \in H'} a'$?

MAKESPAN SCHEDULING: Given n processor and m tasks with execution time $c_1, c_2, \ldots, c_m$, *find a shortest schedule of executing these m tasks on this n processors.*

Both problems are NP-complete!

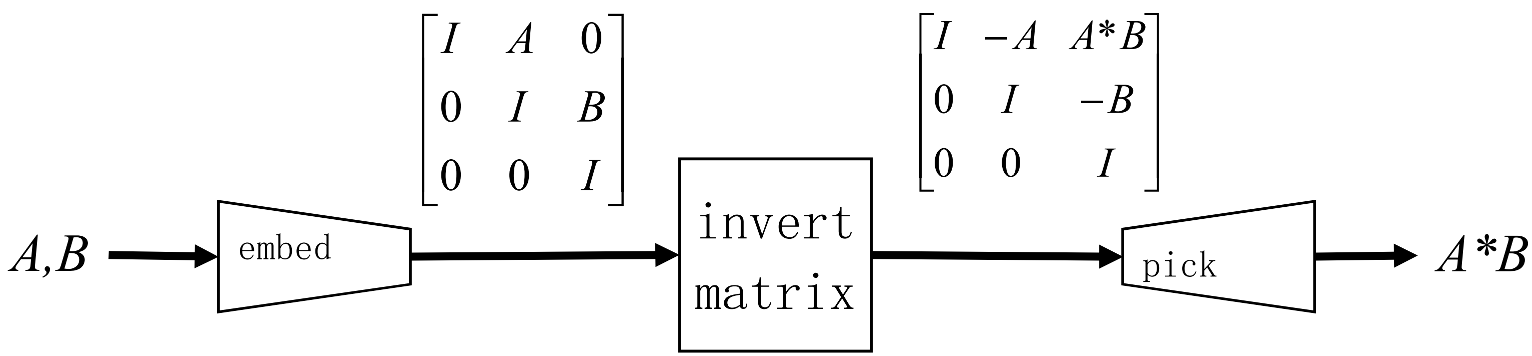
Relationships between problems

Reduction

- the most powerful technique available to compare the complexity of two problems
- reducing the solution of problem A to the solution of problem B $\Rightarrow$ *A is no harder than B* (written $A \le B$)
  - *The translation procedure should be no harder than the complexity of A.*

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & -A & A*B \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix} = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix}$$

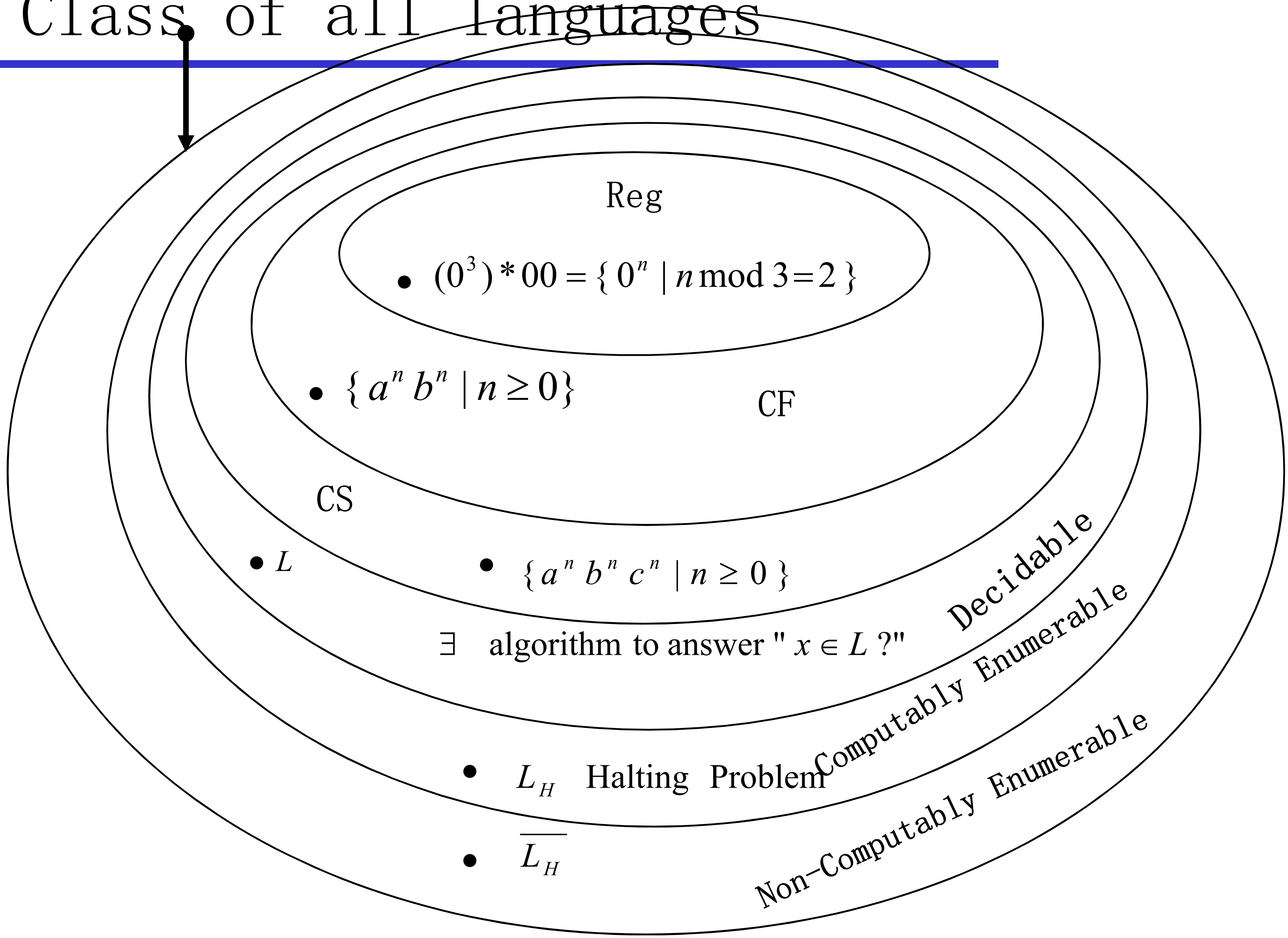
Example: matrix-mult ≤ matrix-inversion



Obtaining Results

- Use definitions, theorems and lemmas, with proofs
- Proofs use construction, induction, reduction, contradiction
  - *Construction:* design an algorithm for a problem, or to build a machine from a grammar, etc.
  - *Induction:* a base case and an induction step imply a conclusion about the general case. Main tool for showing algorithms or constructions are *correct.*
  - *Reduction:* solve a new problem by using the solution to an old problem + some additional operations or transformations
  - *Contradiction:* make an assumption, show that an absurd conclusion follows; conclude the negation of the assumption holds ("reductio ad absurdum")

Class of all languages



Summary: Theory of Computation

- Models of the computing process
  - circuits
  - finite state automata
  - pushdown automata
  - Turing machines
  - capabilities and limitations
- Notion of ``effectively computable procedure''
  - universality of the notion
  - Church's Thesis
  - what is algorithmically computable
- Limitations of the algorithmic process
  - unsolvability (undecidability) & reducibility
- Inherent complexity of computational problems
  - upper and lower bounds: classification by resource use
  - NP-completeness & reducibility