

# Elementary Computation Theory

(Based on [Sipser 2006])

Yih-Kuen Tsay

Department of Information Management  
National Taiwan University

# Preface: What and Why

- 🌐 This lecture provides a quick tour of the basic concepts and results in the theory of computation that are considered essential for the study of formal verification.
- 🌐 Formal verification hinges on precise modeling of the system under verification, and classical **computation models** offer a good starting point of how such precise modeling may be attained.
- 🌐 The common **complexity classes** are also useful when it comes to measure the difficulty of various verification problems.

# Outline

Finite-State Automata

Nondeterminism

Regular Languages

Context-Free Languages

Pushdown Automata

Turing Machines

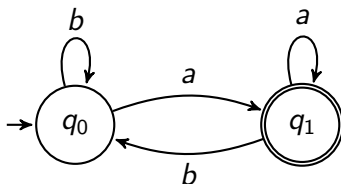
Decidability

Complexity

Reduction and Completeness

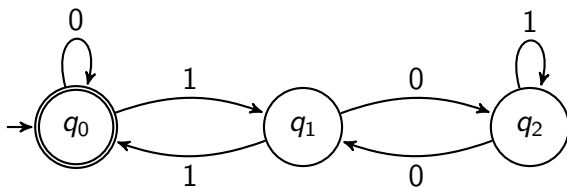
Concluding Remarks

References



- 🌐 This is a finite-state automaton, which recognizes/generates strings over  $\{a, b\}$  that end with an  $a$ .
- 🌐 It is called “finite-state”, as it uses a fixed finite amount (just one bit here) of memory, excluding the machine/program instructions and the input.

# Simple Yet Useful

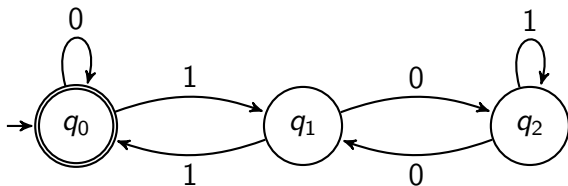


- 🌐 This finite-state automaton recognizes binary numbers (strings over  $\{0, 1\}$ ) that are multiples of 3.
- 🌐 Accepted binary numbers: 0, 11, 110, etc.
- 🌐 Rejected binary numbers: 1, 10, 101, etc.

# Finite-State Automata

- 🌐 Though state diagrams are easier to grasp intuitively, we need the formal definition, too.
- 🌐 A formal definition is precise so as to resolve any uncertainties about what is allowed in a finite-state automaton.
- 🌐 It also provides notation for concise and clear expression.
- 🌐 A **finite(-state) automaton** (DFA) is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , where
  1.  $Q$  is a finite set of *states*,
  2.  $\Sigma$  is a finite set of symbols (the *alphabet*),
  3.  $\delta : Q \times \Sigma \rightarrow Q$  is the *transition function*,
  4.  $q_0 \in Q$  is the *start* state, and
  5.  $F \subseteq Q$  is the set of *accept* (or final) states.

# Finite-State Automata (cont.)



Call the above automaton  $M_3$ . Formally,  $M_3 = (Q, \Sigma, \delta, q_1, F)$ , where

1.  $Q = \{q_0, q_1, q_2\}$ ,
2.  $\Sigma = \{0, 1\}$ ,

3.  $\delta$  is given as

	0	1
$q_0$	$q_0$	$q_1$
$q_1$	$q_2$	$q_0$
$q_2$	$q_1$	$q_2$

4.  $q_0$  is the start state, and
5.  $F = \{q_0\}$ .

- 🌐 We say that  $A$  is the *language* of a machine  $M$ , written as  $L(M) = A$ , if  $A$  is the set of all strings that machine  $M$  accepts.
- 🌐 We also say that  $M$  *recognizes*  $A$  (or  $M$  accepts  $A$ ).
- 🌐 For example,  $L(M_3) = \{w \mid w \text{ is a binary number divisible by } 3\}$ .
- 🌐 A machine is said to accept the empty language  $\emptyset$  if it accepts no strings.
- 🌐 The set of all (finite) strings over an alphabet  $\Sigma$  is conventionally denoted by  $\Sigma^*$ .
- 🌐 So, for a machine  $M$  with  $\Sigma$  as the alphabet,  $L(M) \subseteq \Sigma^*$ .



# Formal Definition of Computation

We already have an informal idea of how a machine computes, i.e., how a machine accepts or rejects a string. Below is a formalization.

- Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a finite automaton and  $w = w_1 w_2 \dots w_n$  be a string over  $\Sigma$ .
- A *run* of  $M$  over  $w$  is a sequence  $r$  of states  $r_0, r_1, \dots, r_n$  such that
  - $r_0 = q_0$  and
  - $\delta(r_i, w_{i+1}) = r_{i+1}$  for  $i = 0, 1, \dots, n - 1$ .
- The run  $r$  is *accepting* if  $r_n \in F$ ; otherwise, it is *rejecting*.
- We say that  $M$  *accepts*  $w$  if the run of  $M$  over  $w$  is accepting.

# Nondeterminism

- In a *nondeterministic* machine, several choices may exist for the next state after reading the next input symbol in a given state.
- The difference between a deterministic finite automaton (DFA) and a nondeterministic finite automaton (NFA):

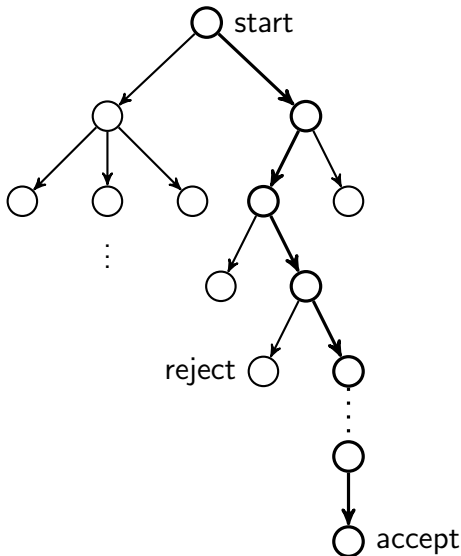
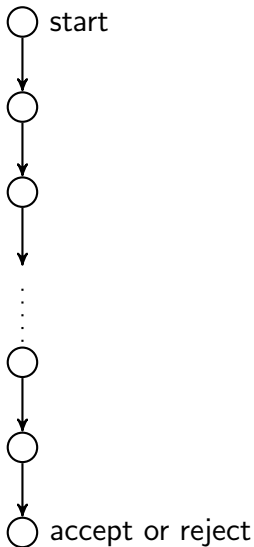
	# of next states (per symbol)	input symbols
DFA	1	from $\Sigma$
NFA	0, 1, or more	from $\Sigma$ or $\Sigma \cup \{\epsilon\}$

Note: sometimes we allow an NFA to make a transition without consuming any input symbol, which is indicated by labeling the transition with an  $\epsilon$ . Such “ $\epsilon$ -transitions” are convenient but do not add expressive power. In this case, the set of input symbols becomes  $\Sigma \cup \{\epsilon\}$ .

# How Does an NFA Compute?

1. If there are multiple choices for the next state, given the next input symbol, the machine **splits into multiple copies**, all moving to their respective next states in parallel.
2. If *any* copy is in an accept state at the end of the input, the machine accepts the input string.
3. If there are input symbols remaining, the preceding steps are repeated.

# Determinism vs. Nondeterminism



# Definition of an NFA

- 🌐 The transition function of an NFA takes a state and an input symbol and produces *a set of possible next states*.
- 🌐 Let  $\mathcal{P}(Q)$  be the power set of  $Q$ .
- 🌐 A **nondeterministic finite automaton** (NFA) is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , where
  1.  $Q$  is a finite set of states,
  2.  $\Sigma$  is a finite alphabet,
  3.  $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$  is the transition function, (alternatively,  $\delta \subseteq Q \times \Sigma \times Q$ , called the transition relation)
  4.  $q_0 \in Q$  is the start state, and
  5.  $F \subseteq Q$  is the set of accept states.
- 🌐 We have chosen to disallow “ $\varepsilon$ -transitions” in this definition.

# Equivalence of NFAs and DFAs

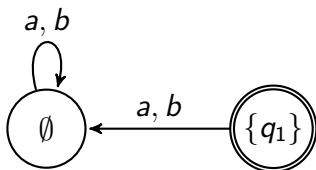
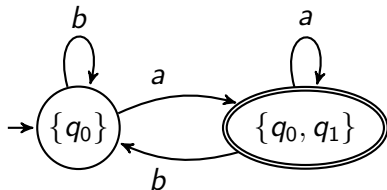
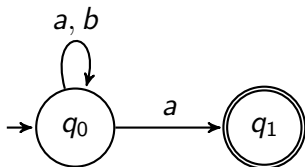
Two machines are *equivalent* if they recognize the same language.

## Theorem

*Every NFA has an equivalent DFA.*

- 🌐 Let  $N = (Q, \Sigma, \delta, q_0, F)$  be an NFA.
- 🌐 Construct a DFA  $M = (Q', \Sigma, \delta', q'_0, F')$  to recognize  $L(N)$  as follows:
  1.  $Q' = \mathcal{P}(Q)$ .
  2. For  $R \in Q'$  and  $a \in \Sigma$ , let  $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$ .
  3.  $q'_0 = \{q_0\}$ .
  4.  $F' = \{R \in Q' \mid R \text{ contains some element of } F\}$ .
- 🌐 The above construction is conventionally referred to as the *subset construction*.

# An NFA and Its Equivalent DFA



# Regular Languages

- 🌐 A language is called a **regular language** if it can be recognized (is recognizable) by some DFA.
- 🌐 There are a few alternatives for defining regular languages.
- 🌐 We will see some of them and show that they are all equivalent.
- 🌐 From the equivalence of NFAs and DFAs, we can immediately conclude that a language recognizable by an NFA is regular.



# Myhill-Nerode Theorem

- 🌐 Given a language  $L \subseteq \Sigma^*$ , define a binary relation  $R_L$  over  $\Sigma^*$  as follows:

$$xR_Ly \text{ iff } \forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$$

- 🌐  $R_L$  can be shown to be an equivalence relation.

## Theorem (Myhill-Nerode)

*With  $R_L$  defined as above, the following are equivalent:*

1.  $L$  is regular.
2.  $R_L$  is of finite index.

*Moreover, the index of  $R_L$  equals the number of states in the smallest DFA that recognizes  $L$ .*

Note: the *index* of an equivalence relation is the number of equivalence classes it induces.

# Regular Operations

- 🌐 In arithmetic, the basic objects are numbers and the tools for manipulating them are operations such as  $+$  and  $\times$ .
- 🌐 In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We consider three operations called **regular operations**.
- 🌐 Let  $A$  and  $B$  be languages. The three regular operations are defined as follows:
  - ☀️ **Union:**  $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$ .
  - ☀️ **Concatenation:**  $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$ .
  - ☀️ **Star:**  $A^* = \{x_1x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$ .
- 🌐 We will use these operations to study the properties of finite automata and regular languages.

# Closure Properties

## Theorem

*The class of regular languages is closed under the union operation (i.e., if  $A_1$  and  $A_2$  are regular languages, so is  $A_1 \cup A_2$ ).*

Note: in fact, the class of regular languages is closed under negation and intersection and hence all Boolean operations.

## Theorem

*The class of regular languages is closed under the concatenation operation.*



## Theorem

*The class of regular languages is closed under the star operation.*

# Regular Expressions

- 🌐 We can use the regular operations (union, concatenation, star) to build up expressions, called **regular expressions**, to describe languages.
- 🌐 The *value* of a regular expression is a *language*.
- 🌐 For example, the value of  $(0 \cup 1)0^*$  is the language consisting of all strings starting with a 0 or 1 followed by any number of 0's. (The symbols 0 and 1 are shorthands for the sets  $\{0\}$  and  $\{1\}$ .)
- 🌐 Regular expressions have an important role in computer science applications involving text.

# Formal Definition of a Regular Expression

-  We say that  $R$  is a **regular expression** if  $R$  is
1.  $a$  for some  $a \in \Sigma$ ,
  2.  $\varepsilon$ ,
  3.  $\emptyset$ ,
  4.  $(R_1 \cup R_2)$ , where  $R_1$  and  $R_2$  are regular expressions,
  5.  $(R_1 \circ R_2)$ , where  $R_1$  and  $R_2$  are regular expressions, or
  6.  $(R_1^*)$ , where  $R_1$  is a regular expression.
- (Note: a definition of this type is called an *inductive definition*).
-  We write  $L(R)$  to denote the language of  $R$ .

# Example Regular Expressions

- Let  $\Sigma$  be  $\{0, 1\}$ .
- $0^*10^* = \{w \mid w \text{ has exactly a single } 1\}$ .
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$ .
- $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains } 001 \text{ as a substring}\}$ .
- $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$ .
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$ .
- $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$ .
- $\emptyset^* = \{\varepsilon\}$ .
- $R \cup \emptyset = R$ ,  $R \circ \varepsilon = R$ ,  $R \circ \emptyset = \emptyset$ , but  $R \cup \varepsilon$  may not equal  $R$ .

# Regular Expressions vs. Finite Automata

## Theorem

*A language is recognizable by an NFA if and only if some regular expression describes it.*

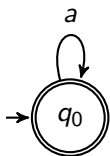
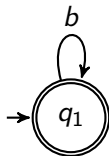
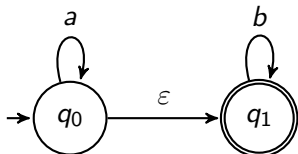
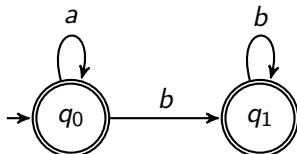
The “if” part can easily be proven by induction:

- 🌐 Languages described by the basic regular expressions ( $a$ ,  $\varepsilon$ , and  $\emptyset$ ) are clearly recognizable by NFAs.
- 🌐 For the compound regular expressions ( $(R_1 \cup R_2)$ ,  $(R_1 \circ R_2)$ , and  $R_1^*$ ), the proof is similar to that for the closure of regular languages under regular operations.

## Corollary

*A language is regular if and only if some regular expression describes it.*

## Regular Expressions vs. Finite Automata (cont.)

(a) NFA for  $a^*$ (b) NFA for  $b^*$ (c) NFA with an  $\epsilon$ -transition(d) NFA for  $a^*b^*$ 

Note:  $\epsilon$ -transitions make it easier to “connect” two automata.



# Nonregular Languages

- 🌐 To understand the power of finite automata we must also understand their limitations.
- 🌐 Consider the language  $B = \{0^n 1^n \mid n \geq 0\}$ .
- 🌐 To recognize  $B$ , a machine will have to remember how many 0's have been read so far. This cannot be done with any finite number of states, since the number of 0's is not bounded.
- 🌐  $C = \{w \mid w \text{ has an equal number of 0's and 1's}\}$  is not regular, either.
- 🌐 But,  $D = \{w \mid w \text{ has equal occurrences of } 01 \text{ and } 10 \text{ as substrings}\}$  is regular.

# The Pumping Lemma


## Lemma


If  $A$  is a regular language, then there is a number  $p$  (the pumping length) such that, if  $s$  is any string in  $A$  and  $|s| \geq p$ , then  $s$  may be divided as  $s = xyz$  satisfying:


1. for each  $i \geq 0$ ,  $xy^i z \in A$ ,
2.  $|y| > 0$ , and
3.  $|xy| \leq p$ .


- Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a DFA that recognizes  $A$ .
- We assign the pumping length  $p$  to be  $|Q|$ .
- Any string  $s$  in  $A$  of length at least  $p$  has an accepting run with two identical states; the input consumed between two closest identical states may serve as the needed  $y$ .

# Example Nonregular Languages

  $B = \{0^n 1^n \mid n \geq 0\}.$

  $C = \{w \mid w \text{ has an equal number of 0's and 1's}\}.$

  $D = \{0^i 1^j \mid i > j\}.$

  $E = \{1^{n^2} \mid n \geq 0\}.$

  $F = \{w \# w \mid w \in \{0, 1\}^*\}.$

# Context-Free Languages

- 🌐 We have seen languages that cannot be described by any regular expression (or recognized by any finite automaton).
- 🌐 *Context-free grammars* (CFGs) are a more powerful method for describing languages; they were first used in the study of natural languages.
- 🌐 They play an important role in the specification and compilation of programming languages.
- 🌐 The collection of languages associated with context-free grammars are called the *context-free languages* (CFLs).

# Context-Free Grammars

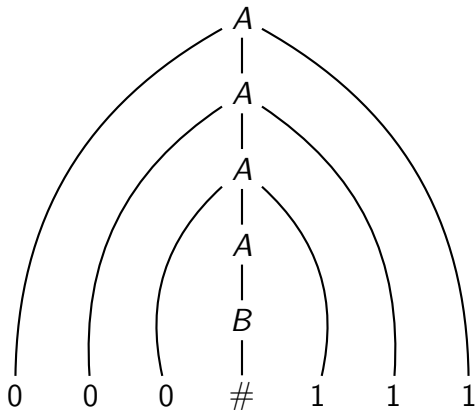
- Basically, a *context-free grammar* (CFG) consists of a collection of *substitution rules* (or *productions*) such as:

$$\begin{array}{l}
 A \rightarrow 0A1 \\
 A \rightarrow B \\
 B \rightarrow \#
 \end{array}
 \quad \text{or alternatively} \quad
 \begin{array}{l}
 A \rightarrow 0A1 \mid B \\
 B \rightarrow \#
 \end{array}$$

- Symbols  $A$  and  $B$  here are called *variables*; the other symbols  $0$ ,  $1$ , and  $\#$  are called *terminals*.
- A grammar describes a language by *generating* each string of the language through a *derivation*.
- For example, the above grammar generates the string  $000\#111$ :  
 $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$ .

# A Parse Tree

The parse tree for  $000\#111$  in grammar  $G_1$ :



# Definition of a CFG

- 🌐 A **context-free grammar** is a 4-tuple  $(V, \Sigma, R, S)$ :
  1.  $V$  is a finite set of *variables*.
  2.  $\Sigma$  ( $\Sigma \cap V = \emptyset$ ) is a finite set of *terminals*.
  3.  $R$  is a finite set of *rules*, each of the form  $A \rightarrow w$ , where  $A \in V$  and  $w \in (V \cup \Sigma)^*$ .
  4.  $S \in V$  is the *start* symbol.
- 🌐 If  $A \rightarrow w$  is a rule, then  $uAv$  *yields*  $uwv$ , written as  $uAv \Rightarrow uwv$ .
- 🌐 We write  $u \Rightarrow^* v$  if  $u = v$  or a sequence  $u_1, u_2, \dots, u_k$  ( $k \geq 0$ ) exists such that  $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$ .
- 🌐 The *language of the grammar* is  $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$ .

# Example CFGs

🌐  $G_3 = (\{S\}, \{(, )\}, R, S)$ , where  $R$  contains

$$S \rightarrow (S) \mid SS \mid \varepsilon.$$

$L(G_3)$  is the language of all strings of properly nested parentheses.

🌐  $G_4 =$   
 $(\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}, \{a, +, \times, (, )\}, R, \langle \text{EXPR} \rangle)$ ,  
 where  $R$  contains

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



# Pushdown Automata

- 🌐 *Pushdown automata* (PDAs) are like nondeterministic finite automata but have an extra component called a *stack*.
- 🌐 A stack is valuable because it can hold an *unlimited* amount of information.
- 🌐 In contrast with the finite automata situation, *nondeterminism* adds power to the capability that pushdown automata would have if they were allowed only to be deterministic.
- 🌐 Pushdown automata are equivalent in power to context-free grammars.
- 🌐 To prove that a language is context-free, we can give either a context-free grammar *generating* it or a pushdown automaton *recognizing* it.

# Recognize a CFL by a PDA


🌐  $B = \{0^n 1^n \mid n \geq 0\}$ .

- ☀️ Push the 0's one by one to the stack.
- ☀️ For each 1 read, pop out one 0.
- ☀️ When the input is exhausted, accept if the stack is empty.
- ☀️ Reject, otherwise.

🌐  $C = \{w \mid w \text{ has an equal number of 0's and 1's}\}$ .

- ☀️ Push the input symbol to the stack if it is the same as top of the stack or if the stack is empty.
- ☀️ Pop out the symbol on top of the stack if it is opposite to the input symbol.
- ☀️ When the input is exhausted, accept if the stack is empty.
- ☀️ Reject, otherwise.

# Definition of a PDA

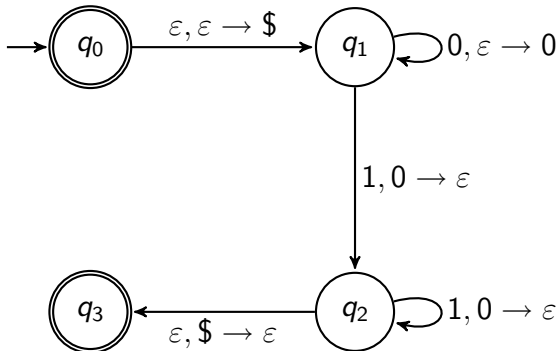
 A **pushdown automaton** (PDA) is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$ , where  $Q$ ,  $\Sigma$ ,  $\Gamma$ , and  $F$  are all finite sets, and

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

 Note that a PDA as defined is nondeterministic.

# State Diagram of a PDA

The state diagram of  $M_B$  for  $B = \{0^n 1^n \mid n \geq 0\}$ :



# Computation of a PDA

- Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a PDA and  $w$  be a string over  $\Sigma$ .
- We say that  $M$  *accepts*  $w$  if we can write  $w = w_1 w_2 \dots w_n$ , where  $w_i \in \Sigma_\epsilon$ , and sequences of states  $r_0, r_1, \dots, r_n \in Q$  and strings  $s_0, s_1, \dots, s_n \in \Gamma^*$  exist such that:
  - $r_0 = q_0$  and  $s_0 = \epsilon$ ,
  - for  $i = 0, 1, \dots, n-1$ ,  $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$  and  $s_i = at$  and  $s_{i+1} = bt$  for some  $a, b \in \Gamma_\epsilon$  and  $t \in \Gamma^*$ .
  - $r_n \in F$ .

# Equivalence of PDAs and CFGs

## Theorem

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

- 🌐 Recall that a context-free language is one that can be described with a context-free grammar.
- 🌐 The proof is by converting any context-free grammar into a pushdown automaton that recognizes the same language and vice versa.


# The Pumping Lemma for CFL

## Lemma

If  $A$  is a context-free language, then there is a number  $p$  such that, if  $s$  is a string in  $A$  and  $|s| \geq p$ , then  $s$  may be divided into five pieces,  $s = uvxyz$ , satisfying the conditions: (1) for each  $i \geq 0$ ,  $uv^i xy^i z \in A$ , (2)  $|vy| > 0$ , and (3)  $|vxy| \leq p$ .

- Let  $G$  be a CFG that generates  $A$ .
- Consider a “sufficiently long” string  $s$  in  $A$  that satisfies the following condition:  
*The parse tree for  $s$  is very tall so as to have a long path on which some variable symbol  $R$  of  $G$  repeats.*
- Take  $p$  to be  $b^{|V|+1}$ , where  $V$  is the set of variables of  $G$ . A string of length at least  $p$  is sufficiently long.

# Non-Context-Free Languages

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

  $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}.$

  $D = \{w#w \mid w \in \{0,1\}^*\}.$



- 🌐 Finite and pushdown automata are too restricted to serve as models of general-purpose computers.
- 🌐 A *Turing machine* (TM) is similar to a finite automaton but with an unlimited and unrestricted memory—*an infinite tape*. It has a tape head that can read and write symbols and move around on the tape.
- 🌐 A Turing machine can do everything that a real computer (as we know it) can do.
- 🌐 Nonetheless, there are problems that no Turing machines, and hence no real computers, can solve.

# An Example Turing Machine

Let  $D = \{w\#w \mid w \in \{0,1\}^*\}$ . A Turing machine  $M_{w\#w}$  for  $D$  (with input on the infinite tape) may work as follows:

1. Zig-zag across the tape to corresponding positions on either side of the  $\#$  symbol to check whether these positions contain the same symbol. If they do not, or if no  $\#$  is found, *reject*. Cross off symbols as they are checked.
2. When all symbols to the left of  $\#$  have been crossed off, check for any remaining symbols to the right of the  $\#$ . If any symbols remain, *reject*; otherwise, *accept*.

# Formal Definition of a TM

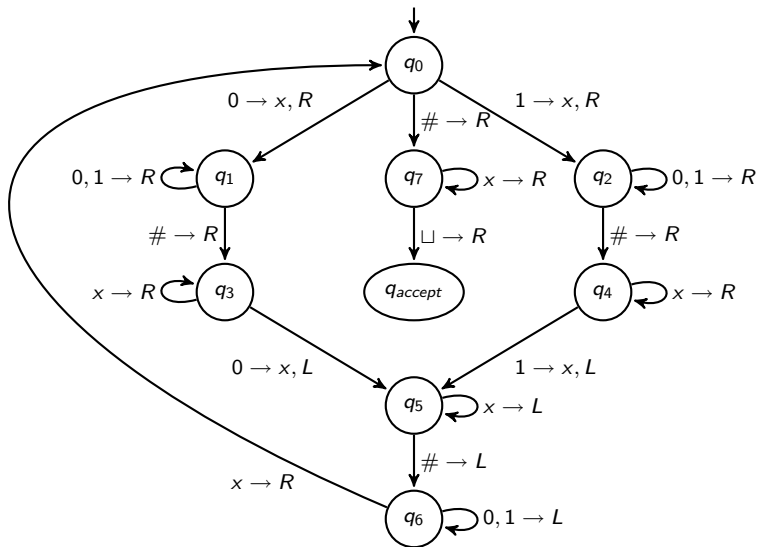
A **Turing machine** (TM) is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ , where  $Q$ ,  $\Sigma$ , and  $\Gamma$  are all finite sets and

1.  $Q$  is the set of states,
2.  $\Sigma$  is the input alphabet, where the *blank* symbol  $\sqcup \notin \Sigma$ ,
3.  $\Gamma$  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.  $q_{\text{accept}} \in Q$  is the accept state, and
7.  $q_{\text{reject}} \in Q$  is the reject state.

Note:  $L$  instructs the tape head to move left and  $R$  to move right.

# State Diagram of a TM

The state diagram of  $M_{w\#w}$  (partial):



# Snapshots/Configurations of a TM

A few snapshots/configurations of  $M_{w\#w}$  on input 011000#011000:

	0	1	1	0	0	0	#	0	1	1	0	0	0	□	...
$q_0$	x	1	1	0	0	0	#	0	1	1	0	0	0	□	...
		$q_1$													
x	1	1	0	0	0	0	#	0	1	1	0	0	0	□	...
								$q_3$							
x	1	1	0	0	0	0	#	x	1	1	0	0	0	□	...
							$q_5$								
x	1	1	0	0	0	0	#	x	1	1	0	0	0	□	...
$q_6$															
x	x	1	0	0	0	0	#	x	1	1	0	0	0	□	...
		$q_0$													
x	x	x	x	x	x	x	#	x	x	x	x	x	x	□	...
															$q_{\text{accept}}$

# Variants of Turing Machines

- Alternative definitions of Turing machines abound, including versions with *multiple tapes* or with *nondeterminism*. They are called *variants* of the Turing machine model.
- The original model and its reasonable variants all have the same power—they *recognize the same class of languages*.
- To show that two models are equivalent, we simply need to show that we can *simulate* one by the other.

# The Definition of Algorithm

- 🌐 All models of a general-purpose computer turn out to be at best equivalent in power to the Turing machine, as long as they satisfy certain reasonable requirements.
- 🌐 This has an important philosophical corollary: Even though there are many different computational models, *the class of algorithms that they describe is unique*.
- 🌐 The **Church-Turing thesis** says that *the intuitive notion of an algorithm corresponds to the formal definition of a Turing machine*.








# Turing-Recognizability and Decidability



- 🌐 A language is **Turing-recognizable** (also called *recursively enumerable*) if some Turing machine recognizes it.
- 🌐 A Turing machine can fail to accept an input by entering the  $q_{\text{reject}}$  state and rejecting, or by looping (not halting).
- 🌐 A machine is called a *decider* if it halts on all inputs. A decider that recognizes some language is said to *decide* the language.
- 🌐 A language is **Turing-decidable**, or simply **decidable** (also called *recursive*), if some Turing machine decides it.



# Example Decidable Languages/Problems

-   $A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$ .
-   $A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts } w \}$ .
-   $A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates } w \}$ .
-   $E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$ .
-   $EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$ .
-   $A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$ .
-   $E_{\text{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \}$ .

Note:  $\langle O_1, O_2, \dots, O_k \rangle$  denotes the encoding of objects  $O_1, O_2, \dots, O_k$  as a string.

# Classes of Languages







Chomsky Hierarchy	Grammar	Language	Computation Model
Type-0	Unrestricted	R.E.	Turing Machine
N/A	(no common name)	Recursive	Decider
Type-1	Context-Sensitive	Context-Sensitive	Linear Bounded
Type-2	Context-Free	Context-Free	Pushdown
Type-3	Regular	Regular	Finite

- 🌐 Recall that Recursively Enumerable (R.E.)  $\equiv$  Turing-recognizable and Recursive  $\equiv$  Decidable (Turing-decidable).
- 🌐 Linear Bounded Automata is a restricted type of Turing machine wherein the tape head is not permitted to move off the portion of the tape containing the input.

# Undecidability

- 🌐 An important finding in computation theory is that *there are specific problems that are algorithmically unsolvable.*
- 🌐 One such problem is *testing whether a Turing machine accepts a given input string.*
- 🌐 This result demonstrates that computers are limited in a very fundamental way.
- 🌐 Unsolvable problems are not necessarily esoteric. Some ordinary problems that people want to solve may turn out to be unsolvable.
- 🌐 For example, the general problem of software verification is not solvable by computer.

# Example Undecidable Languages/Problems

-   $A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ .
-   $HALT_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on } w\}$ .
-   $E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ .
-   $ALL_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ .
-   $OVERLAP_{\text{CFG}} = \{\langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are CFGs where } L(G_1) \cap L(G_2) \neq \emptyset\}$ .
-   $CONTAIN_{\text{CFG}} = \{\langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are CFGs where } L(G_1) \subseteq L(G_2)\}$ .

# Turing-Unrecognizable Languages

## Theorem

*If a language  $L$  and its complement  $\bar{L}$  both are Turing-recognizable, then  $L$  is decidable.*

- 🌐 A TM that simulates the TM for  $L$  and that for  $\bar{L}$  in parallel is a decider for  $L$ .
- 🌐 Let  $\overline{A_{\text{TM}}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ does not accept } w\}$ .
- 🌐 Since  $A_{\text{TM}}$  is Turing-recognizable but not decidable, a corollary follows:

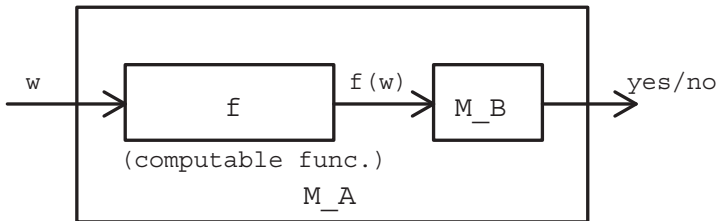
## Corollary

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

# Mapping (Many-One) Reducibility

- Language  $A$  is **mapping reducible** (many-one reducible) to language  $B$ , written  $A \leq_m B$ , if there is a computable function  $f : \Sigma^* \rightarrow \Sigma^*$ , where for every  $w$ ,

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



- This provides a way to convert questions about membership testing in  $A$  to membership testing in  $B$ .

# Reducibility and Decidability

## Theorem

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

- 🌐 Let  $M$  be a decider for  $B$  and  $f$  a reduction from  $A$  to  $B$ . A decider  $N$  for  $A$  works as follows.
- 🌐  $N =$  “On input  $w$ :
  1. Compute  $f(w)$ .
  2. Run  $M$  on input  $f(w)$  and output whatever  $M$  outputs.”

## Corollary

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

# Rice's Theorem

- Let  $P$  be a set of Turing machine descriptions.
- $P$  is *nontrivial* if it contains some, but not all, TM descriptions.
- $P$  is said to be a *property about the languages recognized by Turing machines* if, for any two TMs  $M_1$  and  $M_2$  such that  $L(M_1) = L(M_2)$ ,  $\langle M_1 \rangle \in P$  iff  $\langle M_2 \rangle \in P$ , i.e.,  $P$  does not distinguish TMs that recognize the same language.

## Theorem (Rice)

*Any nontrivial property about the languages recognized by Turing machines is undecidable.*



# Time Complexity

- Let  $M$  be a deterministic TM that halts on all inputs.
- The **running time** or **time complexity** of  $M$  is the function  $f : \mathcal{N} \rightarrow \mathcal{N}$ , where  $f(n)$  is the *maximum* number of steps that  $M$  uses on any input of length  $n$ .
- If  $f(n)$  is the running time of  $M$ , we say that  $M$  *runs in time*  $f(n)$  or that  $M$  *is an*  $f(n)$  *time Turing machine*.
- The running time of a **nondeterministic** TM  $N$  is the function  $f : \mathcal{N} \rightarrow \mathcal{N}$ , where  $f(n)$  is the *maximum* number of steps that  $N$  uses on *any* branch of its computation on any input of length  $n$ .

# Time Complexity Classes

- Let  $t : \mathcal{N} \rightarrow \mathcal{R}^+$  be a function.
- Define the **time complexity classes** as follows:
  - TIME** $(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time Turing machine}\}$ .
  - NTIME** $(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}$ .
- P** =  $\bigcup_k \mathbf{TIME}(n^k)$ .
- EXP** =  $\bigcup_k \mathbf{TIME}(2^{n^k})$ .
- NP** =  $\bigcup_k \mathbf{NTIME}(n^k)$ .
- Apparently, **P**  $\subseteq$  **EXP** and **P**  $\subseteq$  **NP**.

# Space Complexity

- Let  $M$  be a deterministic TM that halts on all inputs.
- The **space complexity** of  $M$  is the function  $f : \mathcal{N} \rightarrow \mathcal{N}$ , where  $f(n)$  is the *maximum* number of tape cells that  $M$  scans on any input of length  $n$ .
- If  $f(n)$  is the space complexity of  $M$ , we say that  $M$  *runs in space*  $f(n)$ .
- The space complexity of a **nondeterministic** TM  $N$  is the function  $f : \mathcal{N} \rightarrow \mathcal{N}$ , where  $f(n)$  is the *maximum* number of tape cells that  $N$  scans on *any* branch of its computation on any input of length  $n$ .

# Space Complexity Classes

- Let  $f : \mathcal{N} \rightarrow \mathcal{R}^+$  be a function.
- Define the **space complexity classes** as follows:
  - SPACE**( $f(n)$ ) = { $L$  |  $L$  is a language decided by an  $O(f(n))$  space Turing machine}.
  - NSPACE**( $f(n)$ ) = { $L$  |  $L$  is a language decided by an  $O(f(n))$  space nondeterministic Turing machine}.
- PSPACE** =  $\bigcup_k \mathbf{SPACE}(n^k)$ .
- NPSPACE** =  $\bigcup_k \mathbf{NSPACE}(n^k)$ .
- Apparently, **PSPACE**  $\subseteq$  **NPSPACE**.

# Complexity Classes of Complements

- The complement of a language  $L \subseteq \Sigma^*$ , written as  $\bar{L}$ , is defined to be  $\Sigma^* - L$ ; in other words,  $x \in \bar{L}$  iff  $x \notin L$ .
- For a complexity class  $\mathcal{C}$ , define

$$\mathbf{co}\mathcal{C} = \{L \mid \bar{L} \in \mathcal{C}\}.$$

- For a deterministic complexity class  $\mathcal{C}$ , we have  $\mathbf{co}\mathcal{C} = \mathcal{C}$ .
- Consider  $\mathbf{coNP} = \{L \mid \bar{L} \text{ is in } \mathbf{NP}\}$ .

☀  $\mathbf{P} \subseteq \mathbf{coNP}$ .

If  $L \in \mathbf{P}$ , then  $\bar{L} \in \mathbf{P} \subseteq \mathbf{NP}$  and hence  $L \in \mathbf{coNP}$ .

- ☀ There exist languages in  $\mathbf{NP} \cap \mathbf{coNP}$  (e.g., the problem of checking whether a number is prime), i.e.,  $\mathbf{NP} \cap \mathbf{coNP} \neq \emptyset$ .
- ☀ Whether  $\mathbf{coNP} = \mathbf{NP}$  is open.

# Relation between Complexity Classes

## Theorem

$$\mathbf{P} \subseteq \begin{matrix} \mathbf{NP} \\ \mathbf{coNP} \end{matrix} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXP}.$$

- 🌐 **PSPACE = NPSPACE** follows from Savitch's Theorem, which says  $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$ .
- 🌐 We also know that  $\mathbf{P} \neq \mathbf{EXP}$ .
- 🌐 So, at least one of the containments must be proper, but we do not know which one.
- 🌐 Most researchers believe all the containments are proper.

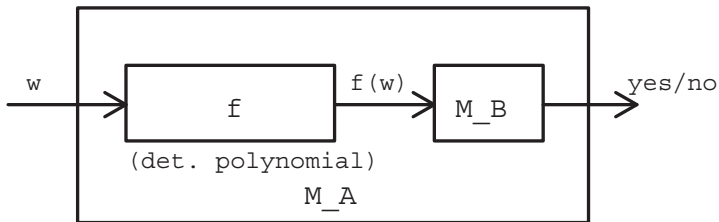
# Polynomial Time Reducibility

- When problem  $A$  is *efficiently* reducible to problem  $B$ , an efficient solution to  $B$  can be used to solve  $A$  efficiently.
- A function  $f : \Sigma^* \rightarrow \Sigma^*$  is a **polynomial time computable function** if some polynomial time Turing machine  $M$ , on every input  $w$ , halts with just  $f(w)$  on its tape.
- Language  $A$  is **polynomial time mapping reducible** (polynomial time reducible) to language  $B$ , written  $A \leq_p B$ , if there is a polynomial time computable function  $f : \Sigma^* \rightarrow \Sigma^*$ , where for every  $w$ ,

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

- Polynomial time reducibility is a *transitive* relation.

# Polynomial Time Reducibility (cont.)





# Hardness and Completeness

- Let  $\mathcal{C}$  be a complexity class such that  $\mathbf{P} \subseteq \mathcal{C}$ .
- A language  $B$  is  $\mathcal{C}$ -hard if every  $A$  in  $\mathcal{C}$  is polynomial time reducible to  $B$ .
- A language is  $\mathcal{C}$ -complete if it is in  $\mathcal{C}$  and is also  $\mathcal{C}$ -hard.
- $\mathcal{C}$ -complete problems are the hardest in the class  $\mathcal{C}$ , ignoring polynomial time differences.
- From the transitivity of polynomial time reducibility, a theorem follows:

## Theorem

*If  $A$  is  $\mathcal{C}$ -complete and  $A \leq_p B$  for some  $B \in \mathcal{C}$ , then  $B$  is  $\mathcal{C}$ -complete.*

- We have restricted to  $\mathcal{C}$  that contains  $\mathbf{P}$ , as the above concepts/results would make little sense for complexity classes beneath  $\mathbf{P}$ .

# NP-Completeness and coNP-Completeness

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

## Theorem

$SAT$  is **NP**-complete.

🌐 Let  $VALIDITY = \{\langle \phi \rangle \mid \phi \text{ is a valid Boolean formula}\}$ .

## Theorem

$VALIDITY$  is **coNP**-complete.

# PSPACE-Completeness

- 🌐 A *fully quantified* Boolean formula is a quantified Boolean formula where each Boolean variable is bound, e.g.,  $\forall x \exists y ((x \vee y) \wedge (\bar{x} \vee \bar{y}))$ .
- 🌐 Every fully quantified Boolean formula is either true or false, e.g., the formula  $\forall x \exists y ((x \vee y) \wedge (\bar{x} \vee \bar{y}))$  is *true*, but  $\exists y \forall x ((x \vee y) \wedge (\bar{x} \vee \bar{y}))$  is *false*.
- 🌐 Let  $TQBF = \{ \langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula} \}$ .




## Theorem

$TQBF$  is **PSPACE**-complete.

# Concluding Remarks

- 🌐 The material summarized here, if fully expanded, would require one full semester to cover.
- 🌐 In particular, we have omitted almost all proofs.
- 🌐 If you have never taken a course or done a self-study on theory of computation, you should try to look up some of the proofs.

# References

-  M. Sipser. *Introduction to the Theory of Computation*, Thomson Course Technology, 2006.
-  C.H. Papadimitriou. *Computational Complexity*, Addison-Wesley, 2003.
-  J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.