Elements of Computation Theory

I. ALPHABET AND LANGUAGES

Definition 1 *Alphabet and languages*

- *alphabet Σ: a finite set of symbols*
- *string: a finite sequence of symbols. Empty string is denoted e.*
- *the set of strings over Σ: Σ^**
- *language: any subset of Σ^**

Definition 2 *Operations on languages*

- *concatenation: $L_1L_2 = \{w \in \Sigma^* : w = xy,\ x \in L_1,\ y \in L_2\}$*
- *complement: $\overline{L} = \Sigma^* - L$*
- *Kleene star: $L^* = \{w \in \Sigma^* : w = w_1 \cdots w_k,\ k \geq 0,\ w_1, \cdots, w_k \in L\}$*
- *If $L \neq \emptyset$, define $L^+ = LL^* = \{w \in \Sigma^* : w = w_1 \cdots w_k,\ k \geq 1,\ w_1, \cdots, w_k \in L\}$*

Remark 1 *$\emptyset$ and $\{e\}$ are different languages. The former has no strings, and the latter has only the empty string e. A special case of Kleene star: $\emptyset^* = \{e\}$.*

Theorem 1 *The set of strings over a finite (even countably infinite) alphabet Σ is countably infinite.*

The members of Σ^* can then be enumerated in the following way.

1. For each $k \geq 0$, all strings of length k are enumerated before all strings of length $k+1$.
2. The n^k strings of length exactly k are enumerated lexicographically.

II. FINITE REPRESENTATION OF A LANGUAGE

- Any finite representation must itself be a string, a finite sequence of symbols over some alphabet Γ.
- We require different languages having different representations.
- Possible representations of languages, that is Γ^* , is countably infinite.
- The set of all possible languages over a given alphabet Σ, that is 2^{Σ^*}, is uncountably infinite.

No matter how powerful are the methods we use for representing languages, only countably many languages can be represented. There being uncountably many languages in all, the vast majority of them will inevitably be missed under any finite representational scheme.

Definition 3 *The **regular expressions** over an alphabet Σ are all strings over $\Sigma \cup \{\ \varnothing\ ,\ \cup\ ,\ \star\ ,\ (\ ,\ )\ \}$ that can be obtained as follows. The set of regular expressions over Σ is denoted $\mathsf{reg}(\Sigma)$.*

1. *$\varnothing$ and each member of Σ is a regular expression.*
2. *α and β are regular expressions $\Longrightarrow$ $(\alpha\beta)$ is a regular expression.*
3. *α and β are regular expressions $\Longrightarrow$ $(\alpha \cup \beta)$ is a regular expression.*
4. *α is a regular expression $\Longrightarrow$ $\alpha\star$ is a regular expression.*
5. *Nothing is a regular expression unless following from 1-4.*

Definition 4 *Language function $\mathsf{L} : \mathsf{reg}(\Sigma) \to 2^{\Sigma^*}$*

1. $\mathsf{L}(\varnothing) = \emptyset$, *and* $\forall a \in \Sigma,\ \mathsf{L}(a) = \{a\}$
2. α *and* β *are regular expressions* $\implies \mathsf{L}((\alpha\beta)) = \mathsf{L}(\alpha)\mathsf{L}(\beta)$
3. α *and* β *are regular expressions* $\implies \mathsf{L}((\alpha \cup \beta)) = \mathsf{L}(\alpha) \cup \mathsf{L}(\beta)$
4. α *is a regular expression* $\implies \mathsf{L}(\alpha\star) = \mathsf{L}(\alpha)^*$

The class of regular languages over an alphabet Σ *is defined to be*

$$\{\mathsf{L}(\alpha) : \alpha \in \mathsf{reg}(\Sigma)\}$$

Alternatively, regular languages can be thought of in terms of closures. The class of regular languages over Σ *is precisely the closure of the set of languages*

$$\{\{\sigma\} : \sigma \in \Sigma\} \cup \{\emptyset\}$$

with respect to the functions of union, concatenation, and Kleene star.

Remark 2 *Regular expressions and the languages they represent can be defined formally and unambiguously, we feel free, when no confusion can result, to blur the distinction between the regular expressions and the "mathematical English" we are using for talking about languages.*

1. *When specifying a singleton language such as* $\{a\}$*, we may dispense with the braces and write "a" instead "*$\{a\}$*".*
2. *At another point, we might say that* a^*b^* *is a regular expression representing the set* $\{a\}^*\{b\}^*$*; in this case, to be precise, we should have written* $(a \star b \star)$.

Regular expressions are an inadequate specification method in general. In search of a general method for finitely specifying languages, we might return to our general scheme.

Definition 5 *Finite language specifications*

- ***language recognition device****: An algorithm that is specifically designed for some language* L*, to answer questions of the form "Is string* w *a member of* L*?".*
- ***language generators****: Describe how a generic specimen in the language is produced.*

III. FINITE AUTOMATA

Definition 6 *The notion of computation*

- *configuration : the status of the computing machine*
- *computation : a sequence of configurations at successive moments*

The finite automaton is a severely restricted model of an actual computer. What makes the finite automaton such a restricted model of real computers is the **complete absence of memory** outside its fixed central processor.

Definition 7 *A* ***finite automaton*** *is a quintuple* $M = (K, \Sigma, \Delta, s, F)$ *where*

1. K *is a finite set of states*
2. Σ *is an alphabet*
3. $s \in K$ *is the initial state*
4. $F \subset K$ *is the set of final states*
5. $\Delta \subset K \times (\Sigma \cup \{e\}) \times K$ *is the transition relation*

- *If* Δ *represents a function* $K \times \Sigma \to K$*, then* M *is called a* ***deterministic finite automaton*** *(DFA). Otherwise,* M *is called a* ***nondeterministic finite automaton*** *(NFA).*

- *A **configuration** of a finite automata is an element of $K \times \Sigma^*$.*
- *$\vdash_M$ is binary relation between two configurations $(q, w), (q', w')$ iff the machine M can pass from one to the other as a result of a single move (one configuration **yields** the other in one step).*

$$(q, a, q') \subset \Delta \Longleftrightarrow (q, aw) \vdash_M (q', w) \tag{1}$$

- *The reflexive and transtive closure of $\vdash_M$ is denoted $\vdash_M^*$.*
- *A string $w \in \Sigma^*$ is **accepted** by M iff*

$$\exists q \in F \ (s, w) \vdash_M^* (q, e) \tag{2}$$

- *The language accepted by M, denoted $L(M)$, is the set of all strings accepted by M.*

The differences between DFA and NFA

1. For any states in DFA, a unique "next state" is associated with it for each input symbol.
2. For an NFA, we shall now permit several possible "next states" for a given combination of current state and input symbol. The automaton, as it reads the input string, may choose at each step to go into anyone of these legal next states, or there is no state to be entered with some input symbol; the choice is not determined by anything in our model, and is therefore said to be nondeterministic.

Such nondeterministic devices are not meant as realistic models of computers. They are simply a useful notational generalization of finite automata, as they can greatly simplify the description of these automata. Moreover, nondeterminism is an inessential feature of finite automata.

Definition 8 *Two FA M_1 and M_2 are equivalent if $L(M_1) = L(M_2)$.*

Theorem 2 *Every NFA is equivalent to a DFA.*

Proof. Let $M = (K, \Sigma, \Delta, s, F)$ be a nondeterministic finite automaton. We shall construct a deterministic finite automaton $M' = (K', \Sigma, \delta, s', F')$ equivalent to M.

The key idea is to view a nondeterministic finite automaton as occupying, at any moment, not a single state but a set of states: namely, all the states that can be reached from the initial state by means of the input consumed thus far.

For any states $q \in K$, $E(q)$ is the closure of the set $\{q\}$ under the relation

$$E(q) = \{p \in K : (q, e) \vdash_M^* (p, e)\} \tag{3}$$

Define the DFA $M' = (K', \Sigma, \delta, s', F')$

$$K' = 2^K \tag{4}$$
$$s' = E(s) \tag{5}$$
$$F' = \{Q \subset K : Q \cap F \neq \emptyset\} \tag{6}$$
$$\delta(Q, a) = \bigcup \{E(p) : p \in K, \ \exists q \in Q \ (q, a, p) \in \Delta\} \tag{7}$$

To prove that M and M' are equivalent, we should show that for any string w

$$\exists q \in F \ (s, w) \vdash_M^* (q, e) \Longleftrightarrow \exists Q \in F' \ (s', w) \vdash_{M'}^* (Q, e)$$

This is indicate by the following claim: $\forall w \in \Sigma^*, \ \forall p, q \in K$

$$(q, w) \vdash_M^* (p, e) \Longleftrightarrow \exists P \in K' \ (p \in P) \wedge [(E(q), w) \vdash_{M'}^* (P, e)]$$

We prove the claim by induction on $|w|$.

1. Basis Step.

$$(q, e) \vdash_M^* (p, e) \Longleftrightarrow p \in E(q)$$

$$\begin{aligned} \exists P \in K' \ (p \in P) \wedge [(E(q), e) \vdash_{M'}^* (P, e)] &\Longleftrightarrow \exists P \in K' \ (p \in P) \wedge (E(q) = P) \\ &\Longleftrightarrow p \in E(q) \end{aligned}$$

2. Induction Hypothesis. Suppose that the claim is true for all strings w of length k or less for some $k \geq 0$.
3. Induction Step.
   - Suppose $|v| = k,\ (q, v) \vdash_M^* (p, e)$, then there exists states r_1, r_2 such that
   $$(q, va) \vdash_M^* (r_1, a) \vdash_M (r_2, e) \vdash_M^* (p, e)$$
   According to induction hypothesis, $\exists R_1 \in K'\ (r_1 \in R_1) \wedge [(E(q), va) \vdash_{M'}^* (R_1, a)]$. And we know
   $$\begin{cases} r_1 \in R_1 \\ (r_1, a, r_2) \in \Delta \\ p \in E(r_2) \end{cases} \implies p \in \delta(R_1, a)$$
   Let $P = \delta(R_1, a)$, then
   $$(E(q), va) \vdash_{M'}^* (R_1, a) \vdash_{M'} (P, e)$$
   - Suppose $\exists P, R_1 \in K'\ (p \in P) \wedge (E(q), va) \vdash_{M'}^* (R_1, a) \vdash_{M'} (P, e)$. Then
   $$\begin{cases} P = \delta(R_1, a) \\ p \in P \end{cases} \implies \exists r_1 \in R_1\ \exists r_2 \in P\ ((r_1, a, r_2) \in \Delta) \wedge (p \in E(r_2))$$
   by the induction hypothesis
   $$(q, va) \vdash_M^* (r_1, a) \vdash_M (r_2, e) \vdash_M^* (p, e)$$

■

The following is an algorithm transferring an NFA to a DFA.

Algorithm 1 Transferring NFA to DFA

```
1:  $states[0] = \emptyset$
2:  $states[1] = E(s)$
3:  $p \leftarrow 1$
4:  $j \leftarrow 0$
5:  while $j \leq p$ do
6:      for all $c \in \Sigma$ do
7:          $tmp \leftarrow \delta(states[j], c)$
8:          if $tmp = states[i]$ for some $i \leq p$ then
9:              $trans[j, c] \leftarrow i$
10:         else
11:             $p \leftarrow p + 1$
12:             $states[p] \leftarrow tmp$
13:             $trans[j, c] \leftarrow p$
14:         end if
15:     end for
16:     $j \leftarrow j + 1$
17: end while
```

Theorem 3 *The class of languages accepted by finite automata is closed under*

1. *union*
2. *concatenation*
3. *Kleene star*
4. *complementation*
5. *intersection*

Proof. Let $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$ and $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$. Without loss of generality, we assume that K_1 and K_2 are disjoint states.

1. Union:

$$\begin{aligned} K &= K_1 \cup K_2 \cup \{s\} \\ F &= F_1 \cup F_2 \\ \Delta &= \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\} \end{aligned}$$

   Then $\forall w \in \Sigma^*$, M accepts w iff M_1 accepts w or M_2 accepts w.

   Or use parallel simulation: (Note that here requires M_1 and M_2 to be DFA.)

$$\begin{aligned} K &= K_1 \times K_2 \\ s &= (s_1, s_2) \\ F &= (F_1 \times K_2) \cup (K_1 \times F_2) \\ \delta &: ((q_1, q_2), a) \mapsto (\delta_1(q_1, a), \delta_2(q_2, a)) \end{aligned}$$

2. Concatenation:

$$\begin{aligned} K &= K_1 \cup K_2 \\ F &= F_2 \\ \Delta &= \Delta_1 \cup \Delta_2 \cup \{(f_1, e, s_2) : f_1 \in F_1\} \end{aligned}$$

   Then $\forall w \in \Sigma^*$, M accepts w iff $w = w_1 w_2$ such that M_1 accepts w_1 or M_2 accepts w_2.

3. Kleene star:

$$\begin{aligned} K &= K_1 \cup \{s_1'\} \\ F &= F_1 \cup \{s_1'\} \\ \Delta &= \Delta_1 \cup \{(f_1, e, s_1) : f_1 \in F_1\} \cup \{(s_1', e, s_1)\} \end{aligned}$$

   Then $\forall w \in \Sigma^*$, M accepts w iff $w = w_1 \cdots w_n$ such that M_1 accepts w_k $(k = 0, \cdots, n)$.

4. Complementation: Let M_1 be a DFA. Define DFA $M = (K_1, \Sigma, \delta_1, s_1, K_1 - F_1)$. Then $\forall w \in \Sigma^*$, M accepts w iff M_1 does not accept w.

5. Intersection. Just recall that

$$L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2)) \tag{8}$$

   and so closedness under intersection follows from closedness under union and complementation.

   Or use parallel simulation: (Note that here requires M_1 and M_2 to be DFA.)

$$\begin{aligned} K &= K_1 \times K_2 \\ s &= (s_1, s_2) \\ F &= F_1 \times F_2 \\ \delta &: ((q_1, q_2), a) \mapsto (\delta_1(q_1, a), \delta_2(q_2, a)) \end{aligned}$$

■

Theorem 4 *A language is regular if and only if it is accepted by a finite automaton.*

Proof.

1. Only if. Recall that The class of regular languages over Σ is precisely the closure of the set of languages

$$\{\{\sigma\} : \sigma \in \Sigma\} \cup \{\emptyset\}$$

   with respect to the functions of union, concatenation, and Kleene star. It is evident that the empty set and all singletons are indeed accepted by finite automata; and by Theorem 3 the finite automaton languages are closed under union, concatenation, and Kleene star. Hence every regular language is accepted by some finite automaton.

2. If. Let $M = (K, \Sigma, \Delta, s, F)$ be a finite automaton. We shall construct a regular expression R such that $\mathsf{L}(R) = L(M)$. Let $K = \{q_1, \cdots, q_n\}$ and $s_1 = q_1$. Define $R(i, j, k)$ as the set of strings driving the automaton M from q_i to q_j with intermediate states choosing from $\{q_1, \cdots, q_k\}$. We know that

$$L(M) = \bigcup\{R(1, j, n) : q_j \in F\} \tag{9}$$

Then we show by induction on k that all of these sets $R(i, j, k)$ are regular, and hence so is $L(M)$.

   (a) Basic Step. For $k = 0$, if $i \neq j$, then

   $$R(i, j, 0) = \{a \in \Sigma \cup \{e\} : (q_i, a, q_j)\}$$

   if $i = j$, then

   $$R(i, i, 0) = \{a \in \Sigma \cup \{e\} : (q_i, a, q_j)\} \cup \{e\}$$

   Each of these sets is finite and therefore regular.

   (b) Induction Hypothesis. $R(i, j, n - 1)$ is regular.

   (c) Induction Step.

   $$R(i, j, n) = R(i, j, n - 1) \cup R(i, n, n - 1)R(n, n, n - 1)^* R(n, j, n - 1)$$

   Thus $R(i, j, n)$ is regular.

Therefore language $R(i, j, k)$ is regular for all i, j, k, thus completing the induction.

■

Method: *Construct regular expressions from finite automata.*

1. Every finite automaton has a "special form"
   (a) It has a single final state, $F = \{f\}$.
   (b) Furthermore, if $(q, u, p) \in \Delta$, then $q \neq f$ and $p \neq s$; that is, there are no transitions into the initial state, nor out of the final state.
2. Denote the states as $q_1, \cdots, q_n$ such that $s = q_{n-1}$ and $f = q_n$.
3. Eliminate the states $q_1, \cdots, q_{n-2}$.

See the example in Figure 1.

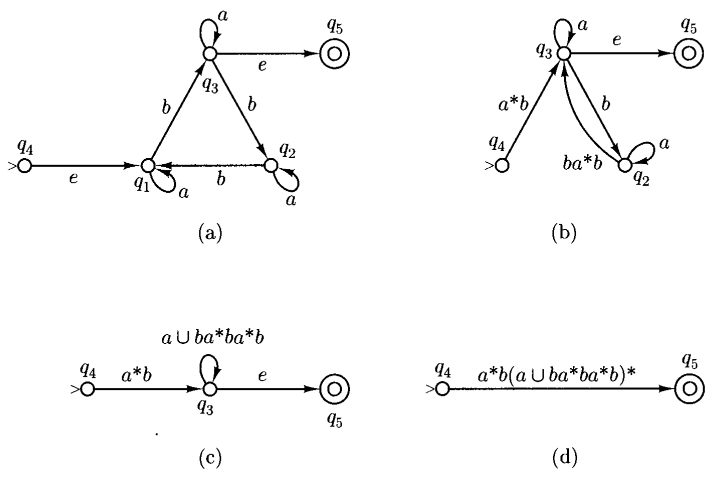

FIG. 1. Construct regular expressions from finite automata.

IV. REGULAR LANGUAGES

Theorem 5 *(Pumping theorem) Let L be a regular language.*

$$\exists n \in \mathbb{N}^* \ \forall w \in L \ \left(|w| \geq n \implies \exists x, y, z \in \Sigma^* \ \forall i \in \mathbb{N}^* \ \begin{cases} w = xyz \\ y \neq e \\ |xy| \leq n \\ xy^iz \in L \end{cases} \right) \tag{10}$$

Proof. Since L is regular, L is accepted by a DFA M. Suppose that n is the number of states of M, and let w be a string of length n or greater.

Consider now the first n steps of the computation of M on w:

$$(q_0, w_1 \cdots w_n) \vdash_M (q_1, w_2 \cdots w_n) \vdash_M \cdots \vdash_M (q_n, e)$$

By the pigeonhole principle there exist i and j, $0 \leq i < j \leq n$, such that $q_i = q_j$. Then the string $y = w_i \cdots w_j$ could be removed from w, or repeated any number of times in w just after the j-th symbol of w, and M would still accept this string. ■

Corollary 1

$$\forall n \in \mathbb{N}^* \ \exists w \in L \ |w| \geq n \wedge \left(\begin{cases} w = xyz \\ y \neq e \\ |xy| \leq n \end{cases} \implies \exists i \geq 0 \ xy^iz \notin L \right) \tag{11}$$

Then L is not a regular.

Applying the theorem correctly can be subtle. It is often useful to think of the application of this result as a game between yourself

- Prover: L is not regular
- Adversary: L is regular

The theorem states that, once L has been fixed,

1. the adversary provides a number n
2. you come up with a string $w \in L$ that is longer than n
3. the adversary supplies an appropriate decomposition of w into xyz
4. you point out i for which xy^iz is not in the language.

If you have a strategy that always wins, no matter how brilliantly the adversary plays, then you have established that L is not regular.

Example 1 *1. $L = \{a^ib^i : i \geq 0\}$ is not regular. $\forall n \in \mathbb{N}^*$, let $w = a^nb^n$, then let $w = xyz$ be a decomposition, then $y = a^i$ $(i > 0)$ but $a^{n-i}b^n \notin L$.*

2. $L = \{a^n : n \text{ is prime}\}$ is not regular. Let $w = a^pa^qa^r$ be a decomposition and take $i = (p + 2q + r + 2)$, then $a^p(a^q)^ia^r = a^{(q+1)(p+2q+r)} \notin L$.

3. Sometimes it pays to use closure properties to show that a language is not regular.

$$L = \{w \in \{a, b\} : w \text{ has an equal number of a's and b's}\} \tag{12}$$

$L \cap a^*b^* = \{a^ib^i : i \geq 0\}$ is not regular.

Theorem 6 *If L is a regular language, then $L^R = \{w : w^R \in L\}$ is also a regular language.*

Proof. Consider the automaton M in the special form. Converse the transition relation to get the new automata M' whose start point is the end point of M' and end point is the start point of M'. ■

V. STATE MINIMIZATION

Identifying the reachable states is easy to do in polynomial time, because the set of reachable states can be defined as the closure of $\{s\}$ under the relation $\{(p,q) : a \in \Sigma, \delta(p,a) = q\}$. Therefore, the set of all reachable states can be computed by this simple algorithm:

Algorithm 2 Computing reachable states

```
1: $R \leftarrow \{s\}$
2: while there is a state $p \in R$ and $a \in \Sigma$ such that $\delta(p,a) \notin R$ do
3:     $R \leftarrow R \cup \{\delta(p,a)\}$
4: end while
```

Definition 9 *Two equivalence relations on languages.*

Let $L \subset \Sigma^*$ be a language, and let $x, y \in \Sigma^*$. We say that x and y are equivalent with respect to L, denoted $x \approx_L y$, if

$$\forall z \in \Sigma^* \ (xz \in L \Leftrightarrow yz \in L)$$

Let $M = (K, \Sigma, \delta, s, F)$ be a DFA. We say that two strings $x, y \in \Sigma^*$ are equivalent with respect to M, denoted $x \sim_M y$, if

$$\exists q \in K \ ((s,x) \vdash_M^* (q,e)) \wedge ((s,y) \vdash_M^* (q,e))$$

Note that $\sim_M$ is less fundamental than $\approx_L$.

Theorem 7 *Both $\approx_L$ and $\sim_M$ are equivalence relations. The equivalence class of $\sim_M$ can be identified by the states of M.*

Example 2

- *For the regular language $L = \{$the strings that do not contain occurrences of every symbol of the alphabet$\}$, the number of equivalence class is $2^{|\Sigma|}$. Let A and B be two distinct subset of Σ and assume that A does not contain B. Let L_X be the language consisting of strings which contains every occurrence of symbols of X. For $x \in L_A$, $y \in L_B$ and $z \in L_{\Sigma - B}$, $xz \in L$ but $yz \notin L$.*

  Recall that there is a nondeterministic finite automaton with $|\Sigma| + 1$ states that accepts the same language. Although deterministic automata are exactly as powerful as nondeterministic ones in principle, determinism comes with a price in the number of states which is, at worst, exponential.

  *To put it in a different way, and in fact a way that anticipates the important issues of computational complexity: When the number of states is taken into account, **non-determinism is exponentially more powerful than determinism in the domain of finite automata**.*

- *For the language $L = (ab|ba)^*$, it is not hard to see that $\approx_L$ has four equivalence classes:*

  1. *$[e] = L$*
  2. *$[a] = La$*
  3. *$[b] = Lb$*
  4. *$[aa] = L(aa|bb)\Sigma^*$*

  Let E_q denote the equivalence class corresponding to state q of M. Then for the automaton in Figure 2

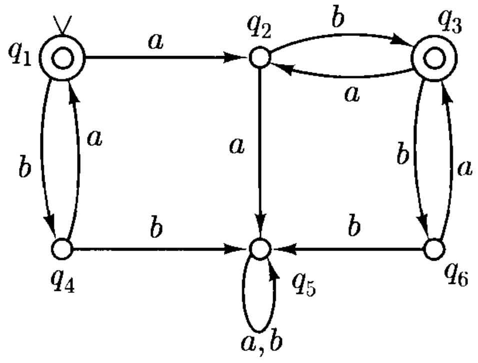

FIG. 2. Minimization

1. $E_{q_1} = (ba)^*$
2. $E_{q_2} = La$
3. $E_{q_3} = (ba)^*abL$
4. $E_{q_4} = b(ab)^*$
5. $E_{q_5} = L(aa|bb)\Sigma^*$
6. $E_{q_6} = (ba)^*abLb$

Definition 10 *An equivalence relation $\sim$ is a refinement of another $\approx$ if*

$$\forall x, y \; x \sim y \implies x \approx y \tag{13}$$

Each equivalence class of $\approx$ is the union of one or more equivalence classes of $\sim$.

Theorem 8 *$\sim_M$ is a refinement of $\approx_{L(M)}$, that is*

$$\forall x, y \in \Sigma^* \; x \sim_M y \implies x \approx_{L(M)} y \tag{14}$$

Theorem 9 *(The Myhill-Nerode Theorem): Let L be a regular language. Then there is a DFA M, $L = L(M)$, and the number of states of M is equal to the number of equivalence classes in $\approx_L$.*

Proof. Construct the following standard DFA M:

$$\begin{aligned} K &= \{[x] : x \in \Sigma^*\} \\ s &= [e] \\ F &= \{[x] : x \in L\} \\ \delta([x], a) &= [xa] \quad \forall [x] \in K \end{aligned}$$

L is regular $\implies$ K is a finite set.
By induction on $|y|$, we have

$$([x], y) \vdash_M^* ([xy], e) \tag{15}$$

Then

$$\begin{aligned} x \in L(M) &\Leftrightarrow \exists q \in F \; ([e], x) \vdash_M^* (q, e) \\ &\Leftrightarrow [x] \in F \\ &\Leftrightarrow x \in L \end{aligned}$$

Therefore, $L = L(M)$. ■

Corollary 2 *A language L is regular if and only if L has finitely many equivalence classes.*

How to compute the standard automaton from a given DFA?
Define the relation $A_M \subset K \times \Sigma^*$

$$A_M = \{(q, w) \in K \times \Sigma^* : \exists f \in F \; (q, w) \vdash_M^* (f, e)\}$$

Define the equivalence relation $\equiv_n \subset K \times K$

$$\equiv_n = \{(p, q) \in K \times K : \forall z \in \Sigma^* \; (|z| \leq n) \wedge ((p, z) \in A_M \Leftrightarrow (q, z) \in A_M)\} \tag{16}$$

Define the equivalence relation $\equiv \subset K \times K$

$$\equiv = \{(p, q) \in K \times K : \forall z \in \Sigma^* \; (p, z) \in A_M \Leftrightarrow (q, z) \in A_M\} \tag{17}$$

The equivalence classes of $\equiv$ are precisely those sets of states of M that must be clumped together in order to obtain the standard automaton of $L(M)$.
For the example in Figure 2

$$[e] = E_{q_1} \cup E_{q_3} \qquad [b] = E_{q_4} \cup E_{q_6}$$

Example 3 *The standard automaton corresponding to the language $L = (ab|ba)^*$ is*

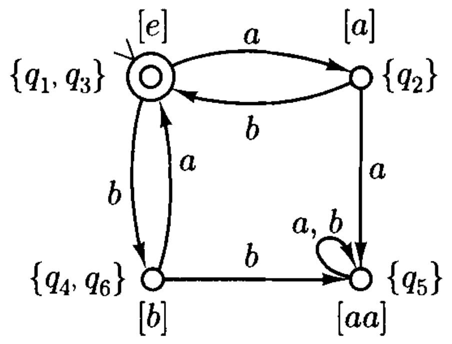

FIG. 3. standard automaton

We can compute the equivalence classes of $\equiv$ step by step.

Lemma 1 *For any two states $q, p \in K$ and any integer $n \geq 1$*

$$q \equiv_n p \Longleftrightarrow \begin{cases} q \equiv_{n-1} p \\ \forall a \in \Sigma \; \delta(q, a) \equiv_{n-1} \delta(p, a) \end{cases} \tag{18}$$

Algorithm 3 Computing the standard automata

```
1: Initially the equivalence classes of $\equiv_0$ are $F$ and $K - F$
2: repeat
3:     for $n = 1, 2, \cdots$ do
4:         compute the equivalence classes of $\equiv_n$ from those $\equiv_{n-1}$
5:     end for
6: until $\equiv_n$ is the same as $\equiv_{n-1}$
```

VI. CONTEXT-FREE LANGUAGES

Definition 11 *A context-free grammar G is a quadruple (V, Σ, R, S), where*

1. *V is an alphabet*
2. *$\Sigma \subset V$ is the set of terminals*
3. *$R \subset (V - \Sigma) \times V^*$*
4. *$S \in V - \Sigma$ is the start symbol*

- *The members of $V - \Sigma$ are called nonterminals.*
- *For any $A \in V - \Sigma$ and $u \in V^*$,*

$$A \rightarrow_G u \iff (A, u) \in R \tag{19}$$

- *For any strings $u, v \in V^*$*

$$u \Rightarrow_G v \iff \exists x, y, w \in V^*,\ A \in V - \Sigma \begin{cases} u = xAy \\ v = xwy \\ A \rightarrow_G w \end{cases} \tag{20}$$

- *The relation $\Rightarrow_G^*$ is the reflexive, transitive closure of $\Rightarrow_G$.*
- *$L(G)$, the language generated by G, is*

$$L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\} \tag{21}$$

- *A language L is said to be a context-free language if $L = L(G)$ for some context-free grammar G.*

Definition 12 *We call any sequence of the form*

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n \tag{22}$$

an n-step derivation in G of w_n from w_0.

Example 4 *The following grammar $G = (V, \Sigma, R, S)$ generates the irregular language $\{a^i b^i : i \in \mathbb{N}\}$*

$$\begin{aligned} V &= \{S, a, b\} \\ \Sigma &= \{a, b\} \\ R &= \{S \rightarrow aSb, S \rightarrow e\} \end{aligned}$$

Theorem 10 *All regular languages are context-free.*

Proof. Consider the regular language accepted by the deterministic finite automaton $M = (K, \Sigma, \delta, s, F)$. The same language is generated by the grammar $G(M) = (V, \Sigma, R, S)$, where $V = K \cup \Sigma$, $S = s$, and R consists of these rules:

$$R = \{q \rightarrow ap : \delta(q, a) = p\} \cup \{q \rightarrow e : q \in F\} \tag{23}$$

■

Let G be a CFG. A string $w \in L(G)$ may have many derivations in G. Intuitively, parse trees are ways of representing derivations of strings in $L(G)$ so that the superficial differences between derivations, owing to the order of application of rules, are suppressed. To put it otherwise, parse trees represent equivalence classes of derivations.

Definition 13 *For a context-free grammar $G = (V, \Sigma, R, S)$, we define its* ***parse trees*** *and their roots, leaves, and yields, as follows.*

1. *$\forall a \in \Sigma$ is a parse tree. The single node of this parse tree is both the root and a leaf. The yield of this parse tree is a.*
2. *If $A \rightarrow a$ is a rule, then there is a parse tree whose root is the node labeled A, its sole leaf is the node labeled e, and its yield is e.*

3. *Given n parse trees, where $n > 1$, with roots labeled $A_1, \cdots, A_n$ respectively, and with yields $Y_1, \cdots, Y_n$, and $A \to A_1 \cdots A_n$ is a rule in R, then we can construct a parse tree. Its root is the new node labeled A, its leaves are the leaves of its constituent parse trees, and its yield is $Y_1 \cdots Y_n$.*

4. *Nothing else is a parse tree.*

*We say that two derivations D and D' are **similar** if the pair (D, D') belongs in the reflexive, symmetric, transitive closure of $\prec$.*

Definition 14 *Let D, D' be derivations in a CFG G. $D \prec D'$ iff*

1. $x_i = x'_i \quad \forall i \neq k$

2. $x_{k-1} = x'_{k-1} = uAvBw$ *where* $u, v, w \in V^*$ *and* $A, B \in V - \Sigma$

3. $x_k = vyvBw$ *where* $A \to y \in \mathbb{R}$

4. $x'_k = uAvzw$ *where* $B \to z \in \mathbb{R}$

5. $x_{k+1} = x'_{k+1} = uyvzw$

Theorem 11 *Let $G = (V, \Sigma, R, S)$ be a CFG, and let $A \in V - \Sigma$, and $w \in \Sigma^*$. Then the following statements are equivalent*

1. $A \Rightarrow^* w$

2. *There is a parse tree with root A and yield w*

3. $A \Rightarrow^*_L w$

4. $A \Rightarrow^*_R w$

Definition 15 *Grammars with strings that have two or more distinct parse trees, are called **ambiguous**. In fact, there are context-free languages with the property that all context-free grammars that generate them must be ambiguous. Such languages are called **inherently ambiguous**.*

VII. PUSHDOWN AUTOMATON

Definition 16 ***Pushdown automaton*** *is defined to be a sextuple $M = (K, \Sigma, \Gamma, \Delta, s, F)$, where*

1. *K is a finite set of states*

2. *Σ is an alphabet (the input symbols)*

3. *Γ is an alphabet (the stack symbols)*

4. *$s \in K$ is the initial state*

5. *$F \in K$ is the set of final states*

6. *$\Delta \subset (K \times (\Sigma \cup \{e\}) \times \Gamma^*) \times (K \times \Gamma^*)$ is a finite transition relation.*

- *If $((p, a, \beta), (q, \gamma)) \in \Delta$ then M, whenever it is in state p with β at the top of the stack, may read a from the input tape, replace β by γ, on the top of the stack, and enter state q. Such a pair $((p, a, \beta), (q, \gamma))$ is called a **transition** of M. To **push** a symbol is to add it to the top of the stack; to **pop** a symbol is to remove it from the top of the stack.*

- *A **configuration** of a pushdown automaton is defined to be a member of $K \times \Sigma^* \times \Gamma^*$: The first component is the state of the machine, the second is the portion of the input yet to be read, and the third is the contents of the pushdown store, read top-down.*

- ***Yield***

$$((p, a, \beta), (q, \gamma)) \in \Delta \Longleftrightarrow (p, ay, \beta\eta) \vdash_M (q, y, \gamma\eta) \tag{24}$$

- *We denote the reflexive, transitive closure of $\vdash_M$ by $\vdash_M^*$.*
- *A string $w \in \Sigma^*$ is accepted by M iff*

$$\exists q \in F \; (s, w, e) \vdash_M^* (q, e, e) \tag{25}$$

- *The language accepted by M, denoted $L(M)$, is the set of all strings accepted by M.*

Example 5 *The PDA $M = (K, \Sigma, \Gamma, \Delta, s, F)$ to accept the language $L = \{wcw^R : w \in \{a, b\}^*\}$.*

$$\begin{array}{l}
K = \{s, f\} \quad \Sigma = \{a, b, c\} \quad \Gamma = \{a, b\} \quad F = \{f\} \\
\Delta = \{ \\
\quad ((s, a, e), (s, a)), \\
\quad ((s, b, e), (s, b)), \\
\quad ((s, c, e), (f, e)), \\
\quad ((f, a, a), (f, e)), \\
\quad ((f, b, b), (f, e)) \\
\;\}
\end{array}$$

Example 6 *The PDA $M = (K, \Sigma, \Gamma, \Delta, s, F)$ to accept the language $L = \{ww^R : w \in \{a, b\}^*\}$.*

$$\begin{array}{l}
K = \{s, f\} \quad \Sigma = \{a, b\} \quad \Gamma = \{a, b\} \quad F = \{f\} \\
\Delta = \{ \\
\quad ((s, a, e), (s, a)), \\
\quad ((s, b, e), (s, b)), \\
\quad ((s, e, e), (f, e)), \\
\quad ((f, a, a), (f, e)), \\
\quad ((f, b, b), (f, e)) \\
\;\}
\end{array}$$

Example 7 *The PDA $M = (K, \Sigma, \Gamma, \Delta, s, F)$ to accept the language*

$$L = \{w \in \{a, b\}^* : w \text{ has the same number of } a\text{'s and } b\text{'s}\}$$

$$\begin{array}{l}
K = \{s, q, f\} \quad \Sigma = \{a, b\} \quad \Gamma = \{a, b, c\} \quad F = \{f\} \\
\Delta = \{ \\
\quad ((s, e, e), (q, c)), \\
\quad ((q, a, c), (q, ac)), \\
\quad ((q, a, a), (q, aa)), \\
\quad ((q, a, b), (q, e)), \\
\quad ((q, b, c), (q, bc)), \\
\quad ((q, b, b), (q, bb)), \\
\quad ((q, b, a), (q, e)), \\
\quad ((q, e, c), (f, e)) \\
\;\}
\end{array}$$

Theorem 12 *Every finite automaton can be trivially viewed as a PDA that never operates on its stack.*

Proof.

$$M = (K, \Sigma, \Delta, s, F) \tag{26}$$

$$M' = (K, \Sigma, \emptyset, \Delta', s, F) \tag{27}$$

where

$$\Delta' = \{((p,u,e),(q,e)) : (p,u,q) \in \Delta\} \tag{28}$$

We have $L(M) = L(M')$. ■

Definition 17 *A PDA $M = (K, \Sigma, \Gamma, \Delta, s, F)$ is simple if*

$$\begin{cases} q \neq s \\ ((q,a,\beta),(p,\gamma)) \in \Delta \end{cases} \Longrightarrow \begin{cases} |\beta| = 1 \\ |\gamma| \leq 2 \end{cases} \tag{29}$$

Theorem 13 *If a language is accepted by an unrestricted PDA, then it is accepted by a simple PDA.*

Proof. Let $M = (K, \Sigma, \Gamma, \Delta, s, F)$ be a PDA. We shall construct a simple PDA $M' = (K', \Sigma, \Gamma \cup \{Z\}, \Delta', s', \{f'\})$ that also accepts $L(M)$, where $K' = K \cup \{s', f'\}$ and Δ' consists of

1. start transition: $((s', e, e), (s, Z))$
2. final transitions: $\forall f \in F,\ ((f, e, Z), (f', e))$
3. the transitions transformed from those in Δ

■

Theorem 14 *Each context-free language is accepted by some PDA.*

Proof. Let $G = (V, \Sigma, R, S)$ be a context-free grammar; we must construct a PDA M such that $L(M) = L(G)$. The machine we construct has only two states, p and q, and remains permanently in state q after its first move. Also, M uses V, the set of terminals and nonterminals, as its stack alphabet. We let

$$M = (\{p,q\}, \Sigma, V, \Delta, p, \{q\}) \tag{30}$$

where Δ contains the following transitions:

1. $((p,e,e),(q,S))$
2. $((q,e,A),(q,x))$ for each rule $A \to x$ in R
3. $((q,a,a),(q,e))$ for each $a \in \Sigma$

To continue the proof, in order to establish that $L(M) = L(G)$, we prove the following claim: Let $w \in \Sigma^*$ and $\alpha \in (V - \Sigma)V^* \cup \{e\}$. Then

$$S \overset{L}{\Rightarrow}^* w\alpha \Longleftrightarrow (q,w,S) \vdash_M^* (q,e,\alpha) \tag{31}$$

Taking $\alpha = e$ completes the proof.

The following is proof of the claim.

1. To prove $S \overset{L}{\Rightarrow}^* w\alpha \implies (q,w,S) \vdash_M^* (q,e,\alpha)$, induction on the number of derivations.

   Basis Step. If the derivation is of length 0. then $w = e, \alpha = S$ and hence $(q,w,S) \vdash_M^* (q,e,\alpha)$.

   Induction Hypothesis. Suppose $S \overset{L}{\Rightarrow}^* w\alpha \implies (q,w,S) \vdash_M^* (q,e,\alpha)$ holds when the derivation is of length n or less.

   Induction Step. Consider the derivation of length $n+1$:

   $$S = u_0 \Rightarrow \cdots \Rightarrow u_n \Rightarrow u_{n+1} = w\alpha \tag{32}$$

   Let $u_n = xA\beta$, then

   $$(q,x,S) \vdash_M^* (q,e,A\beta) \tag{33}$$

   Suppose the $(n+1)$-th derivation use the rule $A \to \gamma$, then

   $$x\gamma\beta = w\alpha = xy\alpha \tag{34}$$

   $$(q,w,S) = (q,xy,S) \vdash_M^* (q,y,A\beta) \vdash_M (q,y,\gamma\beta) \vdash_M^* (q,e,\alpha) \tag{35}$$

2. To prove $(q, w, S) \vdash_M^* (q, e, \alpha) \implies S \overset{L}{\Rightarrow}^* w\alpha$, induction on the number of type 2 transitions.

   Basis Step: if there are no type-2-transtion, then $w = e$ and $\alpha = S$, thus $S \overset{L}{\Rightarrow}^* w\alpha = S$.

   Induction Hypothesis. Suppose $(q, w, S) \vdash_M^* (q, e, \alpha) \implies S \overset{L}{\Rightarrow}^* w\alpha$ holds when there are no more that n type-2-transtions.

   Induction Step. Consider the computation with $n + 1$ type 2 transitions. The last type-2-transtion is operated on $(q, y, A\beta)$ and the rule is $A \to \gamma$.

$$(q, w, S) \vdash_M^* (q, y, A\beta) \vdash_M (q, y, \gamma\beta) \vdash_M^* (q, e, \alpha) \tag{36}$$

$$\gamma\beta = y\alpha \tag{37}$$

   According to induction hypothesis

$$S \overset{L}{\Rightarrow}^* xA\beta \overset{L}{\Rightarrow}^* x\gamma\beta = xy\alpha = w\alpha \tag{38}$$

■

Theorem 15 *If a language is accepted by a PDA, it is a context-free language.*

Proof. Let $M = (K, \Sigma, \Gamma, \Delta, s, F)$ be a PDA. There is a simple PDA $M' = (K', \Sigma, \Gamma \cup \{Z\}, \Delta', s', \{f'\})$ that also accepts $L(M)$, where $K' = K \cup \{s', f'\}$. Let $G = (V, \Sigma, R, S)$ where V consists of

1. S and each symbol in Σ
2. $\langle q, A, p \rangle \quad q, p \in K' \; A \in \Gamma \cup \{e, Z\}$

■

Theorem 16 *The context-free languages are closed under union, concatenation, and Kleene star.*

Theorem 17 *The intersection of a context-free language with a regular language is a context-free language.*

VIII. TURING MACHINES

Definition 18 *A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$, where*

1. *K is a finite set of states*
2. *Σ is an alphabet such that $\{\sqcup, \rhd\} \subset \Sigma$ and $\{\leftarrow, \rightarrow\} \not\subset \Sigma$.*
3. *$s \in K$ is the initial state*
4. *$H \subset K$ is the set of halting states*
5. *$\delta : (K - H) \times \Sigma \to K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ is the transition function which satisfies*
   - *(a) $\forall q \in K - H,\; \delta(q, \rhd) = (p, \rightarrow)$ for some $p \in K$*
   - *(b) $\forall q \in K - H,\; p \in K$ and $a \in \Sigma,\; \delta(q, a) \neq (p, \rhd)$*

Definition 19 *A configuration of a Turing machine $M = (K, \Sigma, \delta, s, H)$ is a member of $K \times \rhd\Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\})$.*

Example 8 *Consider the Turing machine $M = (K, \Sigma, \delta, s, \{h\})$ where*

$$\begin{aligned} K &= \{q_0, q_1, h\} \\ \Sigma &= \{a, \sqcup, \rhd\} \\ s &= q_0 \end{aligned}$$

The transition function δ is defined as

$$\begin{aligned} \delta(q_0, a) &= (q_1, \sqcup) \\ \delta(q_0, \sqcup) &= (h, \sqcup) \\ \delta(q_0, \rhd) &= (q_0, \rightarrow) \\ \delta(q_1, a) &= (q_0, a) \\ \delta(q_1, \sqcup) &= (q_0, \rightarrow) \\ \delta(q_1, \rhd) &= (q_1, \rightarrow) \end{aligned}$$

When M is started in its initial state q_0, it scans its head to the right, changing all a's to $\sqcup$'s as it goes, until it finds a tape square already containing $\sqcup$; then it halts.

Example 9 *Basic Turing machines:*

1. *Move left:* $L = (\{s, h\}, \Sigma, \delta, s, \{h\})$, $\delta(s, \sigma) = \begin{cases} (h, \leftarrow) & \sigma \in \Sigma - \{\rhd\} \\ (s, \rightarrow) & \sigma = \rhd \end{cases}$
2. *Move right:* $R = (\{s, h\}, \Sigma, \delta, s, \{h\})$, $\delta(s, \sigma) = \begin{cases} (h, \rightarrow) & \sigma \in \Sigma - \{\rhd\} \\ (s, \rightarrow) & \sigma = \rhd \end{cases}$
3. *Write:* $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$, $\delta(s, b) = \begin{cases} (h, a) & b \in \Sigma - \{\rhd\} \\ (s, \rightarrow) & b = \rhd \end{cases}$

Definition 20 *The Rules for Combining Machines. Turing machines will be combined in a way suggestive of the structure of a finite automaton. Individual machines are like the states of a finite automaton, and the machines may be connected to each other in the way that the states of a finite automaton are connected together. However, the connection from one machine to another is not pursued until the first machine halts; the other machine is then started from its initial state with the tape and head position as they were left by the first machine. So if M_1 , M_2 and M_3 are Turing machines, the machine operates as follows: Start in the initial state of M_1; operate as M_1 would operate until M_1 would halt; then, if the currently scanned symbol is an a, initiate M_2 and operate as M_2 would operate; otherwise, if the currently scanned symbol is a b, then initiate M_3 and operate as M_3 would operate.*

A formal definition is

$$\begin{aligned} K &= K_1 \cup K_2 \cup K_3 \\ s &= s_1 \\ H &= H_2 \cup H_3 \\ \delta(q, \sigma) &= \begin{cases} \delta_1(q, \sigma) & q \in K_1 - H_1 \\ \delta_2(q, \sigma) & q \in K_2 - H_2 \\ \delta_3(q, \sigma) & q \in K_3 - H_3 \\ (s_2, \sigma) & q \in H_1,\ \sigma = a \\ (s_3, \sigma) & q \in H_1,\ \sigma = b \\ (h, \sigma) & h \in H,\ \sigma \neq a, b \end{cases} \end{aligned}$$

Definition 21 *Let $M = (K, \Sigma, \delta, s, \{y, n\})$ be a TM.*

- *accepting configuration: the halting configuration whose state component is y*
- *rejecting configuration: the halting configuration whose state component is n*
- *M accepts $w \in (\Sigma - \{\sqcup, \rhd\})^*$ if $(s, \rhd\underline{\sqcup}w)$ yields an accepting configuration*

- *M rejects $w \in (\Sigma - \{\sqcup, \rhd\})^*$ if $(s, \rhd\underline{\sqcup}w)$ yields an rejecting configuration*

Let $\Sigma_0 \subset \Sigma - \{\sqcup, \rhd\}$ be the input alphabet. We say that M decides a language $L \subset \Sigma_0^*$ if

$$\begin{cases} w \in L \implies M \text{ accepts } w \\ w \notin L \implies M \text{ rejects } w \end{cases}$$

*A language L is **recursive** if there is a Turing machine that decides it.*

Remark 3 *There is a subtle point in relation to Turing machines that decide languages: With the other language recognizers that we have seen so far in this book (even the nondeterministic ones), one of two things could happen: either the machine accepts the input, or it rejects it. A Turing machine, on the other hand, even if it has only two halt states y and n, always has the option of evading an answer ("yes" or "no"), by **failing to halt**. Given a Turing machine, it might or it might not decide a language – and there is no obvious way to tell whether it does. The far-reaching importance – and necessity – of this deficiency will become apparent later in this chapter and in the next.*

Definition 22 *Let $M = (K, \Sigma, \delta, s, \{h\})$ be a TM. Suppose*

$$(s, \rhd\underline{\sqcup}x) \vdash_M^* (h, \rhd\underline{\sqcup}y)$$

where $x, y \in \Sigma_0^*$ and $\Sigma_0 \subset \Sigma - \{\sqcup, \rhd\}$ is the input alphabet.

Then y is called the output of M on input x, and is denoted $M(x)$. Notice that $M(x)$ is defined only if M halts on input x.

Let $f : \Sigma_0^* \to \Sigma_0^*$. We say that M computes function f if

$$M(x) = f(x) \qquad \forall x \in \Sigma_0^*$$

*A function f is called **recursive**, if there is a Turing machine M that computes f.*

Definition 23 *Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, let $\Sigma_0 \subset \Sigma - \{\sqcup, \rhd\}$ be an alphabet, and let $L \subset \Sigma_0^*$ be a language. M semidecides L if*

$$w \in L \iff M \text{ halts on } w$$

*A language L is **recursively enumerable** if and only if there is a Turing machine M that semidecides L.*

Theorem 18 *If a language is recursive, then it is recursively enumerable.*

Theorem 19 *If L is a recursive language, then its complement $\overline{L}$ is also recursive.*

Definition 24 *Extentions of Turing machines*

1. *k-tape Turing machine: the transition function is from $(K - H) \times \Sigma^k$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k$*
2. *Two-way Infinite Tape*
3. *Multiple Heads*
4. *Two-Dimensional Tape*

Theorem 20 *Any language decided or semidecided, and any function computed by Turing machines with several tapes, heads, two-way infinite tapes, or multi-dimensional tapes, can be decided, semidecided, or computed, respectively, by a standard Turing machine.*

Definition 25 *A random access Turing machine is a pair $M = (k, \Pi)$ where $k > 0$ is the number of registers, and $\Pi = (\pi_1, \pi 2, \cdots, \pi_p)$, the program, is a finite sequence of instructions. We assume that the last instruction, π_p, is always a halt instruction (the program may contain other halt instructions as well).*

A configuration of a random access Turing machine (k, Π) is a $(k+2)$-tuple $(\kappa, R_0, R_1, \cdots, R_{k-1}, T)$, where κ is the program counter. The configuration is called a halted configuration is $\kappa = 0$.

T, the tape contents, is a finite set of pairs of positive integers – that is, a finite subset of $(\mathbb{N} - \{0\}) \times \{\mathbb{N} - \{0\}\}$ – such that $\forall i \geq 1$ there is at most one pair of the form $(i, m) \in T$.

Theorem 21 *Any language decided or semidecided by a random access Turing machine, and any function computable by a random access Turing machine, can be decided, semidecided, and computed, respectively, by a standard Turing machine. Furthermore, if the machines halt on an input, then the number of steps taken by the standard Turing machine is bounded by a polynomial in the number of steps of the random access Turing machine on the same input.*

IX. GRAMMER

Example 10 *The grammar that generates the language* $\{ww : w \in \{a, b\}\}$

X. NUMERICAL FUNCTIONS

XI. UNDECIDABILITY

A. the Church-Turing thesis

Our notion of an algorithm must exclude Turing machines that may not halt on some inputs. We therefore propose to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm". Nothing will be considered an algorithm if it cannot be rendered as a Turing machine that is guaranteed to halt on all inputs, and all such machines will be rightfully called algorithms. This principle is known as the **Church-Turing thesis**. It is a thesis, not a theorem, because it is not a mathematical result: It simply asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine). Not being a mathematical statement, the Church-Turing thesis cannot be proved. It is theoretically possible, however, that the Church-Turing thesis could be disproved at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as a plausible and reasonable model of computation, and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

Adopting a precise mathematical notion of an algorithm opens up the intriguing possibility of formally proving that certain computational problems cannot be solved by any algorithm. We already know enough to expect this. In Chapter 1 we argued that if strings are used to represent languages, not every language can be represented: there are only a countable number of strings over an alphabet, and there are uncountably many languages. Finite automata, pushdown automata, context-free grammars, unrestricted grammars, and Turing machines are all examples of finite objects that can be used for specifying languages, and that can be themselves described by strings (in the next section we develop in detail a particular way of representing Turing machines as strings). Accordingly, there are only countably many recursive and recursively enumerable languages over any alphabet. So although we have worked hard to extend the capabilities of computing machines as far as possible, in absolute terms they can be used for semideciding or deciding only an infinitesimal fraction of all the possible languages.

Using cardinality arguments to establish the limitation of our approach is trivial; finding particular examples of computational tasks that cannot be accomplished within a model is much more interesting and rewarding. In earlier chapters we did succeed in finding certain languages that are not regular or context-free; in this chapter we do the same for the recursive languages. There are two major differences, however.

1. First, these new negative results are not just temporary setbacks, to be remedied in a later chapter where an even more powerful computational device will be defined: according to the Church-Turing thesis, computational tasks that cannot be performed by Turing machines are impossible, hopeless, undecidable.

2. Second, our methods for proving that languages are not recursive will have to be different from the "pumping" theorems we used for exploiting the weaknesses of context-free grammars and finite au-tomata. Rather, we must devise techniques for exploiting the considerable power of Turing machines in order to expose their limitations.

B. The halting problem

The Turing machine (program) halts is defined as the Turing machine that takes P, an arbitrary Turing machine (program), and X, the input of P, as input, and decides whether P halts with input X.

Theorem 22 halts *does not exists.*

Proof. Suppose halts, define the following program:

```
diag(X):
    if halts(X, X) then
        loop infinitely
    else
        halt
```

Then consider the process diag(diag):

$$\begin{aligned}
&\text{diag(diag) halts} \Longleftrightarrow \text{halts(diag, diag)} \implies \text{diag(diag) does not halt}\\
&\text{diag(diag) does not halt} \Longleftrightarrow \text{!halts(diag, diag)} \implies \text{diag(diag) halts}
\end{aligned}$$

This comes to a contradiction. Hence there must not be a halts program. ■

Theorem 23 *(Rice's Theorem): Suppose that $\mathcal{C}$ is a proper, nonempty subset of the class of all recursively enumerable languages. Then the following problem is undecidable: Given a Turing machine M, is $L(M) \in \mathcal{C}$?*

We can assume that $\emptyset \in \mathcal{C}$ and $L \in \mathcal{C}$. We can construct T_M such that the language semidecided by T_M is either the language L or $\emptyset$.

Let M_L be a TM that semidecides L. T_M first simulate UTM on input "M". If M halts on e, then T_M goes on to simulate the M_L on input x, if it halts, then accept. Otherwise, T_M rejects input x.

It is easy to see that $L(T_M) \in \mathcal{C}$ iff M halts on input e. Thus we reduce H_e to the problem. ■