Theory of Computation

Slides based on Michael Sipser's Textbook

Moses A. Boudourides[1]

Visiting Associate Professor of Computer Science
Haverford College

[1] Moses.Boudourides@gmail.com

Spring 2022

Contents

0. PRELIMINARIES

Strings, I

Basic Definitions

- An **alphabet** is a nonempty finite set, the elements of which would be called **symbols**. Typically, we will use the Greek letter Σ to denote an alphabet. Examples of alphabets:
  $\Sigma_1 = \{a, b\}, \Sigma_2 = \{0, 1\}, \Sigma_3 = \{a, b, c, \cdots, z\}$, etc.
- A **string over an alphabet** is a finite sequence of symbols from that alphabet, usually written next to one another (i.e., *concatenated*) and not separated by commas. Examples of strings: if $\Sigma_1 = \{a, b\}$, then *abaab* is a string over Σ_1; if $\Sigma_2 = \{a, b, c, \cdots, z\}$, then *aloha* is a string over Σ_2.

Strings, II

Basic Definitions, cont.

- For a string x, $|x|$ stands for the **length** (i.e., the number of symbols) of x.
- In addition, for a string x over alphabet Σ and a symbol $\sigma \in \Sigma$,

  $n_\sigma(x) =$ the number of occurrences of the symbol σ in the string x.

- The **null string** is a string over Σ, which is defined as the string with zero length and it is denoted by ε, no matter what the alphabet Σ is. As said, $|\varepsilon| = 0$.
- The **set of all strings over alphabet** Σ will be written Σ^*. For the alphabet $\{a, b\}$, we have

  $$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

Strings, III

Basic Definitions, cont.

- If string x (over alphabet Σ) has length n, we can write $x = x_1 x_2 \cdots x_n$, where each $x_i \in \Sigma$. The **reverse** of x, written x^R, is the string obtained by writing x in the opposite order, i.e., $x^R = x_n x_{n-1} \cdots x_1$. String x is called **palindrome** if $x = x^R$.
- If we have string x of length m and string y of length n, the **concatenation** of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1 \cdots x_m y_1 \cdots y_n$.
- If s is a string and $s = xyz$, for three strings x, y and z, x is called **prefix** of s, z **suffix** of s, and y **substring** of s. Strings x, y, z are called **proper prefix–suffix–substring** of s, respectively, if they are different that s.
- The **lexicographic order** of strings is the same as the familiar dictionary order. The **shortlex order** or simply **string order** is a lexicographic order, in which shorter strings precede longer strings. Thus, for example, the string ordering of all strings over the alphabet $\{a, b\}$ is $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

Strings, IV

Definition of string exponentation

For every string x and integer $k \geq 0$, x^k is a string when defined as:

$$x^k = \begin{cases} \varepsilon, & \text{for } k = 0, \\ x^{k-1}x, & \text{for } k > 0. \end{cases}$$

Operations on Strings

For any strings x, y, z over alphabet Σ, i.e., $x, y, z \in \Sigma^*$,

- $\varepsilon x = x\varepsilon = x$, i.e., ε is the *neutral* or *identity element* of concatenation, considered as a binary relation on Σ^*.
- if either $xy = x$ or $yx = x$, then $y = \varepsilon$,
- $|xy| = |x| + |y|$,
- $(xy)z = x(yz)$, i.e., concatenation is an associative relation and, thus, we may write xyz without specifying how the fractors are grouped.

Languages

Definition

A **language** L over alphabet Σ is a set of strings over Σ, i.e., $L \subseteq \Sigma^*$.

Examples of Languages

- $\varnothing$ is the empty language (since $\{\varnothing\} \subset \Sigma^*$).
- $\{\sigma \mid \sigma \in \Sigma\}$ is the language of all symbols, considered as strings with length 1.
- $\{\varepsilon, a, aab\}$ is a language over $\{a, b\}$ consisting of three strings.
- $Pal(\Sigma)$ is the language of all palindromes over Σ.
- $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$.
- $\{x \in \{a, b\}^* \mid |x| \geq 2$ and x begins and ends with $b\}$.

Remark

As languages, $\{\varepsilon\} \neq \varnothing$. In addition, $\varepsilon \in \Sigma^*$, though other languages $L \subset \Sigma^*$ may or may not contain ε (in the above examples only the third and the fourth do).

Operations on Languages, I

Propositions on Set Operations and Concatenations of Languages

Let L, L_1, L_2 be languages over Σ. Then:

- $L_1 \cup L_2, L_1 \cap L_2, L_1 \setminus L_2$ and the complement of L, denoted $\overline{L}$ and defined as $\overline{L} = \Sigma^* \setminus L$, are all languages over Σ.
- The **concatenation of two languages** L_1 and L_2, denoted $L_1 \circ L_2$ and defined as $L_1 \circ L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$, is a language over Σ.
- $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$. (Notice: $L \circ \varnothing = \varnothing \circ L = \varnothing$.)
- If $L \circ L_1 = L$ (or $L_1 \circ L = L$), it is not always true that $L_1 = \{\varepsilon\}$ (a counterexample is given by $L_1 = \Sigma^*$).
- However, if L_1 is a language such that $L \circ L_1 = L$ (or $L_1 \circ L = L$), for *every* language L, then $L_1 = \{\varepsilon\}$.

Operations on Languages, II

Definition of Language Exponentation

For every language L and integer $k \geq 0$, L^k is a language when defined as:

$$L^k = \begin{cases} \{\varepsilon\}, & \text{for } k = 0, \\ L^{k-1} \circ L, & \text{for } k > 0. \end{cases}$$

Remark

$$\Sigma^k = \{x \in \Sigma^* \mid |x| = k\}.$$

Operations on Languages, III

Definition of Language Closures

For every language L, the **Kleene closure** or **Kleene star** of L and the **positive closure** of L are the languages, denoted L^* and L^+, respectively, which are defined by

$$L^* = \bigcup_{k \geq 0} L^k,$$

$$L^+ = \bigcup_{k \geq 1} L^k.$$

In other words, L^* is the set of strings formed by taking any number of strings (possibly none) from L, possibly with repetitions, and concatenating all of them, and L^+ is the same set, when we should take at least one of such strings. Symbolically:

$$L^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in L\},$$

$$L^+ = \{x_1 x_2 \dots x_k \mid k \geq 1 \text{ and each } x_i \in L\}.$$

Operations on Languages, IV

Remark

$$\varnothing^* = \{\varepsilon\} \text{ and } \varnothing^+ = \varnothing,$$
$$\{\varepsilon\}^* = \{\varepsilon\} \text{ and } \{\varepsilon\}^+ = \{\varepsilon\}.$$

Proposition

For any language L:

- $L^* = \{\varepsilon\} \cup L^+$,
- $\varepsilon \in L^*$ and $\varepsilon \in L^+ \iff \varepsilon \in L$,
- $L^+ = L \circ L^* = L^* \circ L$,
- $(L^+)^+ = L^+$,
- $(L^*)^* = L^*$.

Operations on Languages, V

Example

For $a \in \Sigma$, consider the language $L = \{a\}$. Then:

$$L^* = \{\varepsilon, a, a^2, a^3, \ldots\} = \sum_{k \geq 0} a^k,$$

$$L^+ = \{a, a^2, a^3, \ldots\} = \sum_{k \geq 1} a^k.$$

Example: The case $L^* = L^+ = L$

Let $\Sigma = \{0, 1, 2, 3\}$ and $L = \{x \in \Sigma^* \mid n_3(x) = 0\}$. Clearly, $\varepsilon \in L$. We claim that, for all integers $k \geq 1$, $L^k = L$. Apparently, $L^k \subset L$. In addition, if $x \in L$, then, for any integer $k \geq 1$, $x = \varepsilon^{k-1}x$, i.e., $x \in L^k$, which implies that $L \subset L^k$. Therefore, $L^+ = \bigcup_{k \geq 1} L^k = \bigcup_{k \geq 1} L = L$. Moreover, $L^* = \{\varepsilon\} \cup L^+ = \{\varepsilon\} \cup L = L$ (since $\varepsilon \in L$).

I. REGULAR LANGUAGES

Definition of Deterministic Finite Automata (DFA)

Definition: **A Deterministic Finite Automaton**

A **deterministic finite automaton** (**DFA**) is a 5–tuple $(Q, \Sigma, q_0, F, \delta)$, where

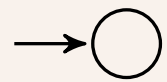
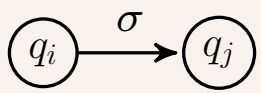
- Q is a finite set called the **states**,
- Σ is a finite set called the **alphabet**,
- $q_0 \in Q$ is the **start state**,
- $F \subseteq Q$ is the **set of accept states**, and
- $\delta : Q \times \Sigma \to Q$ is the **transition function**,

For any element q of Q and any symbol $\sigma \in \Sigma$, we interpret $\delta(q, \sigma)$ as the state to which the DFA moves, when it is in state q and receives the input σ.

Graphical Representation of Finite Automata (FA)

Graph Plots of FAs

A FA is drawn as a **labeled directed graph**, in which:

- vertices, drawn as  or  or  , correspond to states,
- the start state is drawn as  ,
- accept states are drawn as  , and
- transition $\delta(q_i, \sigma) = q_j$ is drawn as  .

Configurations and Yieldings

Definition

Let $M = (Q, \Sigma, q_0, F, \delta)$ be a FA. Any element C of the Cartesian product $Q \times \Sigma^*$ is called **configuration** of M. An **initial configuration** of M is a configuration $C_0 = (q_0, x)$, for $x \in \Sigma^*$, and a **final configuration** of M is a configuration $C_f = (q_f, x)$, for $q_f \in F$ and $x \in \Sigma^*$.

Given two configurations C_i and C_j such that $C_i = (q_i, \sigma y)$ and $C_j = (q_j, y)$, for $q_i, q_j \in Q, y \in \Sigma^*$ and $\sigma \in \Sigma$, we say that configuration C_i **yields in one step** configuration C_j and write

$$C_i \vdash C_j,$$

if

$$q_j = \delta(q_i, \sigma).$$

The Language Accepted by a FA

Definition

Let $M = (Q, \Sigma, q_0, F, \delta)$ be a FA. Given a string $x \in \Sigma^*$, we say that x is **accepted** by M, if there exists a finite sequence of configurations $C_0, C_1, \ldots, C_n$ such that

- $C_0 = (q_0, x), C_n = (q_f, \varepsilon)$, for $q_f \in F$, and
- $C_0 \vdash C_1 \vdash \cdots \vdash C_n$, which is symbolically written as $C_0 \vdash^* C_n$.

The **language accepted** or **recognized** by M is the set

$$L(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}.$$

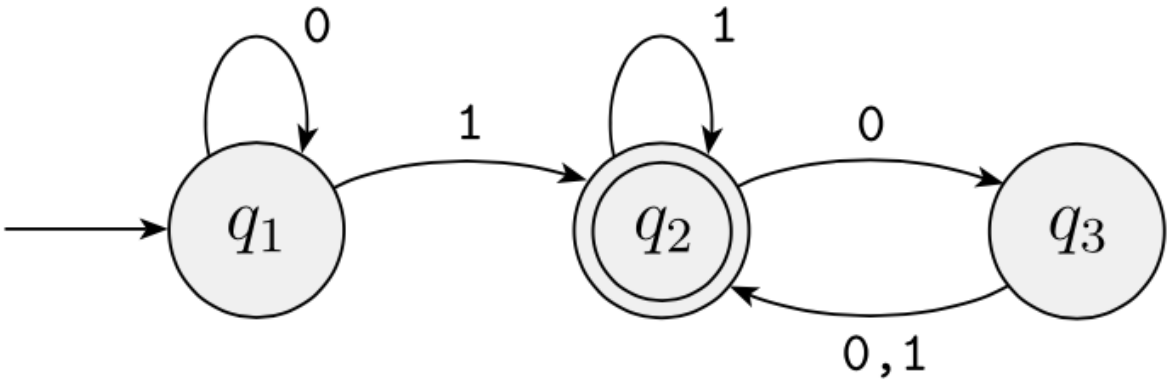
If L is a language over Σ, L is accepted by M if and only if $L = L(M)$.

Definition

A language L over Σ is called **regular language** if there exists a FA $M = (Q, \Sigma, q_0, F, \delta)$ such that $L = L(M)$, i.e., L is accepted (recognized) by M.

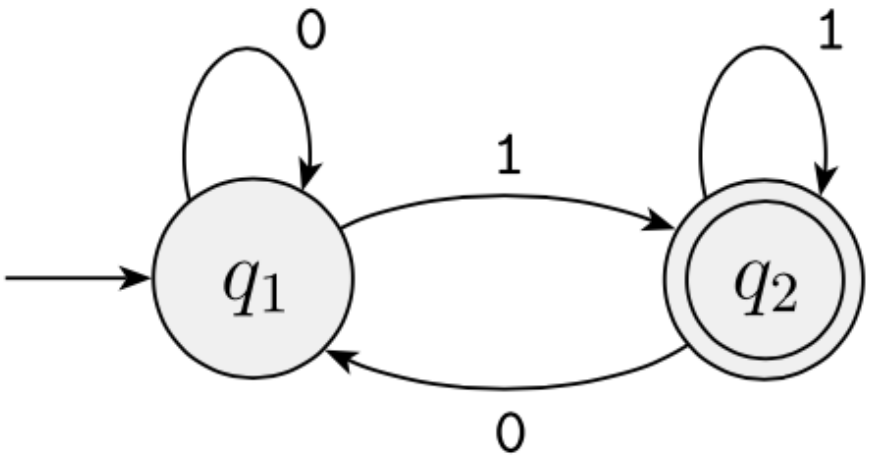
Examples of DFA, I

Example 1:



$L(M) = \{x \mid x$ contains at least one 1 and an even number of 0's follow the last 1$\}$

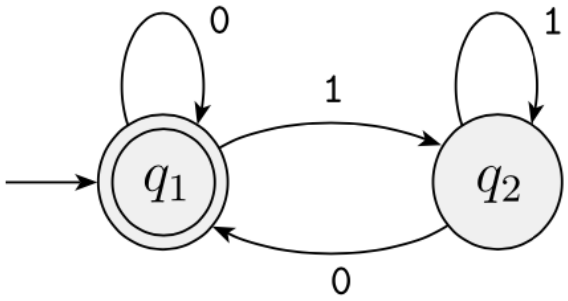
Example 2:



$L(M) = \{x \mid x$ ends in 1$\}$

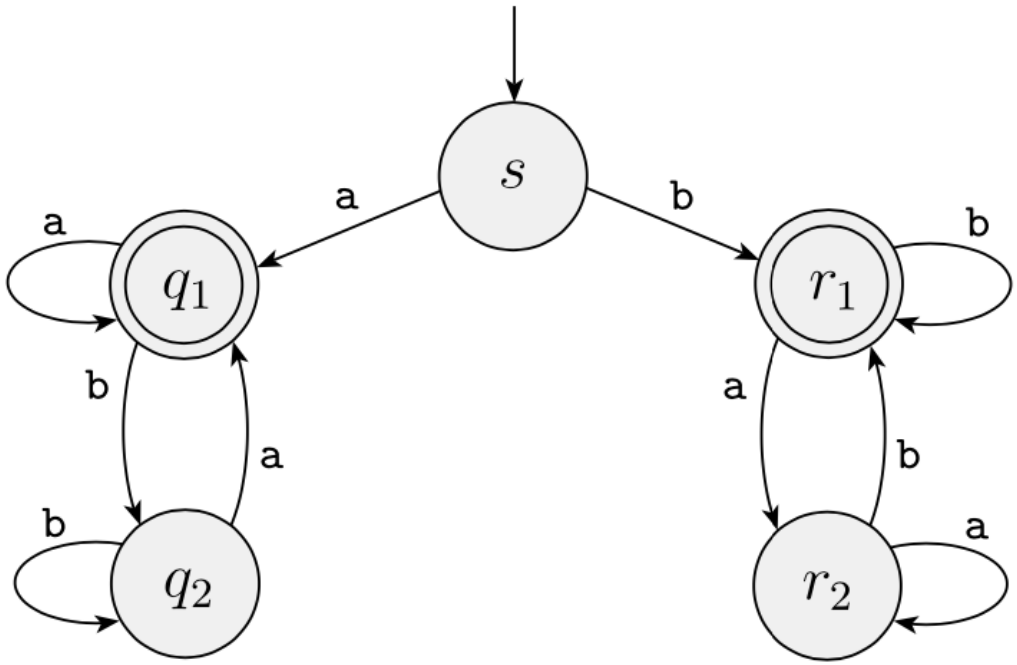
Examples of DFA, II

Example 3:



$L(M) = \{x \mid x = \varepsilon \text{ or ends in a } 0\}$

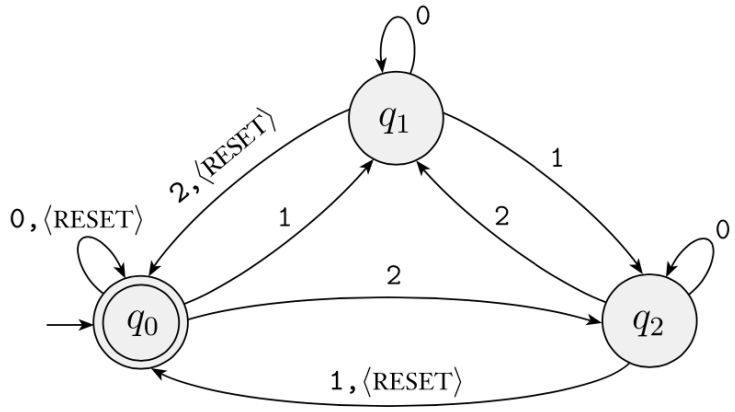
Example 4:



$L(M) = \{x \mid x \text{ starts and ends with the same symbol}\}$

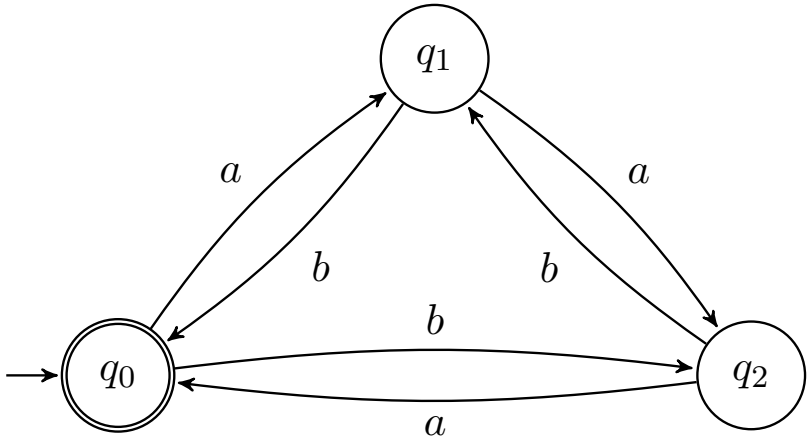
Examples of DFA, III

Example 5:



$$L(M) = \{x \mid x \text{ with sum of symbols equal to } 0 \bmod 3\}$$

Example 6:

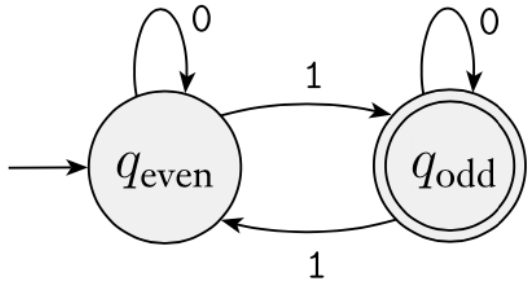


$$L(M) = \{x \mid n_a(x) - n_b(x) = 0 \bmod 3\}$$

Designing DFA

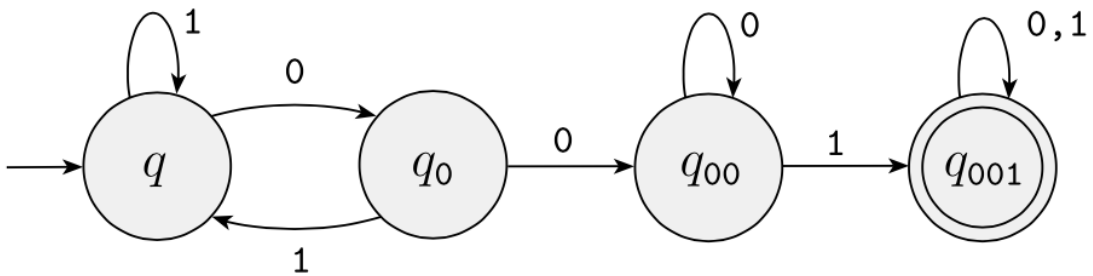
Example 1:

$$L = \{x \in \{0,1\}^* \mid n_1(x) \text{ is odd}\}$$



Example 2:

$$L = \{x \in \{0,1\}^* \mid x \text{ contains the substring } 001\}$$



Regular Operations

Definition

Let L, L_1 and L_2 be languages over the same alphabet Σ. We define three **regular operations** as follows:

- **Union**: $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in B\}$.
- **Concatenation**:
  $L_1 \circ L_2 = \{xy \mid x \in L_1 \text{ andr } y \in B\}$.
- **(Kleene) Star**:
  $L^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in L\}$.

Theorem: Closure of Regular Languages under Regular Operations

The class of regular languages is closed under all three regular operations: (i) union, (ii) concatenation, and (iii) Kleene star.

Definition of Nondeterministic Finite Automata (NFA), I

Definition: **A Nondeterministic Finite Automaton**

A **nondeterministic finite automaton** (**NFA**) is a 5–tuple $(Q, \Sigma, q_0, F, \delta)$, where

- Q is a finite set called the **states**,
- Σ is a finite set called the **alphabet**,
- $q_0 \in Q$ is the **start state**,
- $F \subseteq Q$ is the **set of accepting** or (**final**) **states**, and
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the **transition function**, where $\mathcal{P}(Q)$ denotes the **power set** of the set of states Q.

For any element q of Q and any symbol $\sigma \in \Sigma$, we interpret $\delta(q, \sigma)$ as the set of states to which the NFA moves, when it is in state q and receives the input σ, or, if $\sigma = \varepsilon$, the set of states other than q to which the NFA can move from state q without receiving any input symbol.

Definition of Nondeterministic Finite Automata (NFA), II

Remarks on the Definition of NFA

- **Nondeterminism**: a string may follow more than one paths.
- ε**–moves**: a state can follow from a different state without receiving (reading) any input.
- **Empty states**: since the range of the transition function is the power set, a pair (q, σ) may be mapped to $\varnothing$ and, thus, there would be no arrow moving out of q labeled with σ.

Definition: ε–Closures of States

For any state p of a NFA, the ε–**closure** of p is defined to be a set denoted as $\varepsilon(p)$ consisting of all states q such that there is a path of arrows from p to q such that all the arrows of this path are labeled with ε. Notice that always $p \in \varepsilon(p)$ is true.

Equivalence of NFAs and DFAs

Theorem: **Equivalence of NFAs and DFAs**

Two FA are said **equivalent** if they recognize the same language.

Every NFA corresponds to an equivalent DFA.

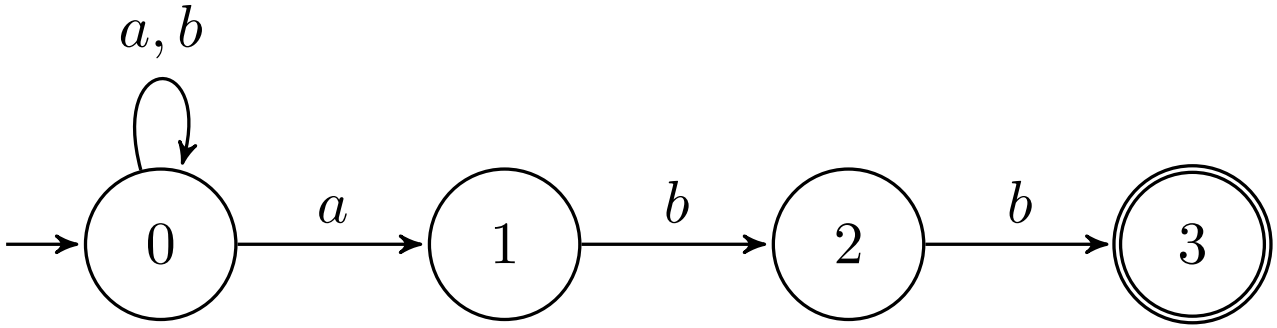
Algorithm for the Reduction of NFA to DFA

- Construct a table with the following columns:

q	$\delta(q,a)$	$\delta(q,b)$	$\delta(q,\varepsilon)$	$\varepsilon(q)$	$\delta(\varepsilon(q),a)$	$\delta(\varepsilon(q),b)$	$\varepsilon(\delta(\varepsilon(q),a))$	$\varepsilon(\delta(\varepsilon(q),b))$

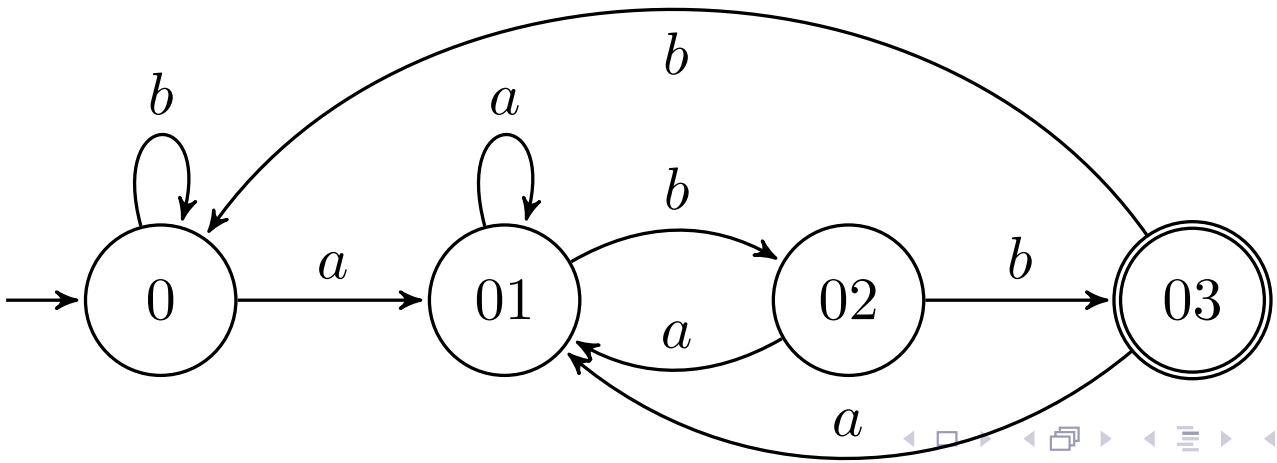
- If the NFA is without ε–moves, use only the first three columns!
- Concatenating notation of states:
  - Use subscripts of states!
  - $\{q_i\}\cup\{q_j\}\cup\{q_k\}=q_i+q_j+q_k=$ (denoted as) ijk (etc.)
- Underline final states in the table.
- Draw the equivalent DFA from the last two columns of the table.

Example: Reduction of NFA without ε–moves to DFA

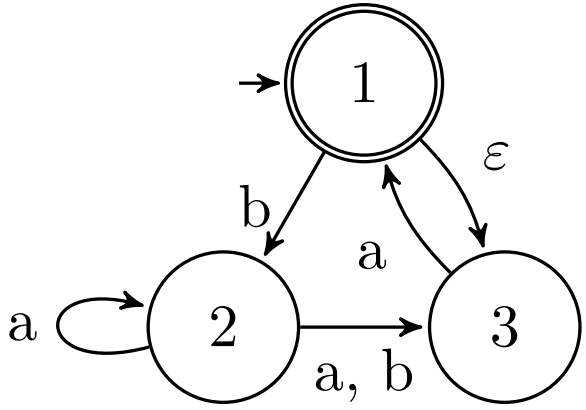


$$L(M) = \{x \in (a+b)^* \mid x \text{ ends in } abb\}$$
$$= (a+b)^* abb.$$

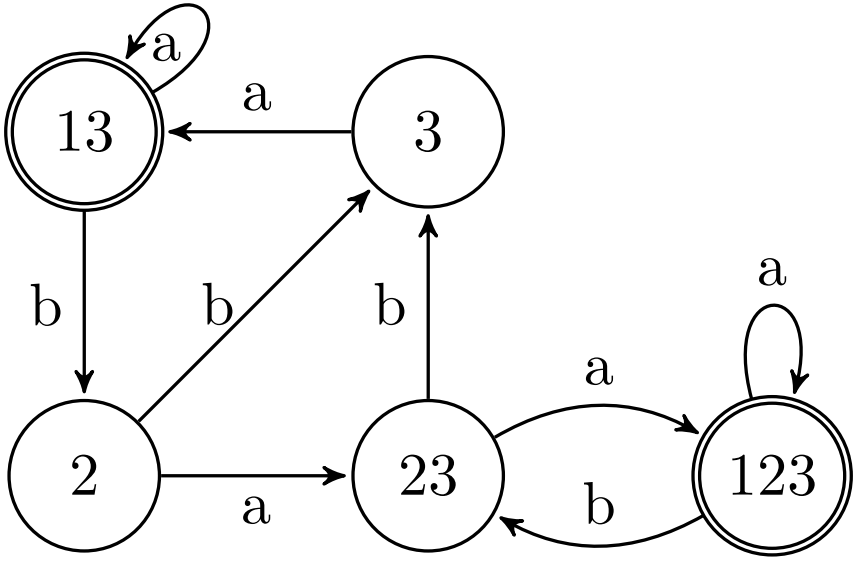
q	$\delta(q,a)$	$\delta(q,b)$
0	01	0
1	$\varnothing$	2
2	$\varnothing$	$\underline{3}$
3	$\varnothing$	$\varnothing$



Example: Reduction of NFA with ε–moves to DFA

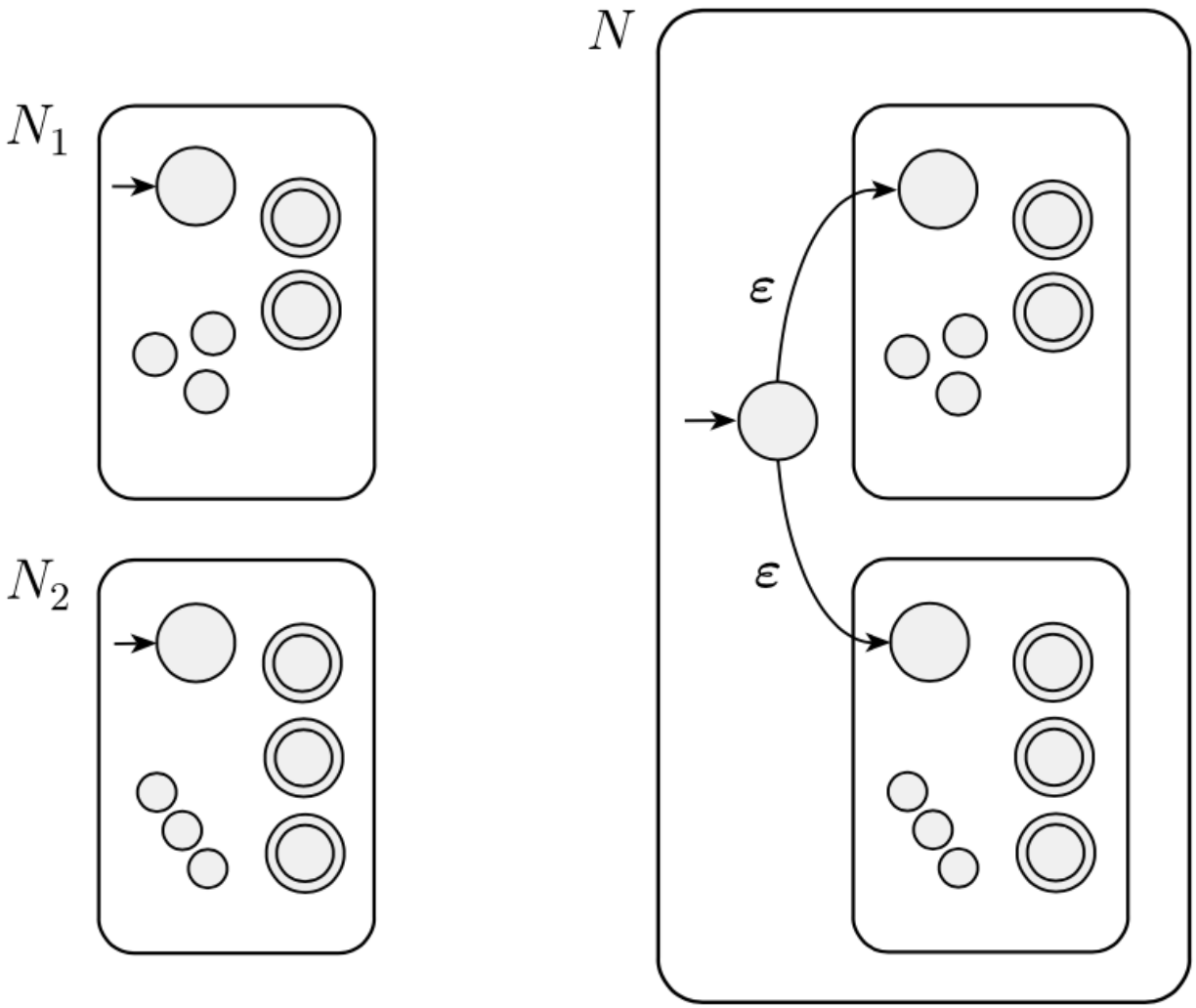


q	$\delta(q,a)$	$\delta(q,b)$	$\delta(q,\varepsilon)$	$\varepsilon(q)$	$\delta(\varepsilon(q),a)$	$\delta(\varepsilon(q),b)$	$\varepsilon(\delta(\varepsilon(q),a))$	$\varepsilon(\delta(\varepsilon(q),b))$
1	$\varnothing$	2	3	13	1	2	$\underline{13}$	2
2	23	3	$\varnothing$	2	23	3	23	3
3	1	$\varnothing$	$\varnothing$	3	1	$\varnothing$	$\underline{13}$	$\varnothing$



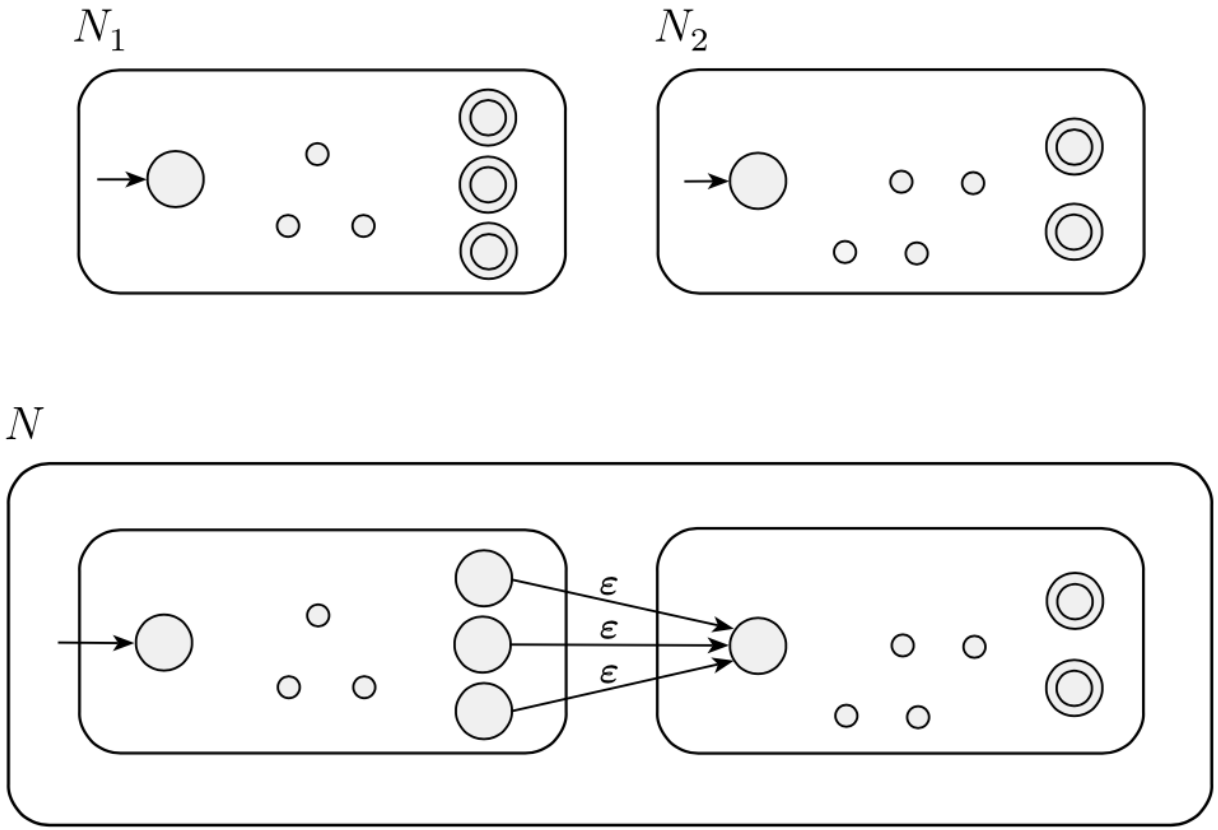
Union of two FAs

If $L_1 = L(N_1)$ and $L_2 = L(N_2)$, then $L_1 \cup L_2$ is recognized (accepted) by the NFA N, i.e., $L_1 \cup L_2 = L(N)$.



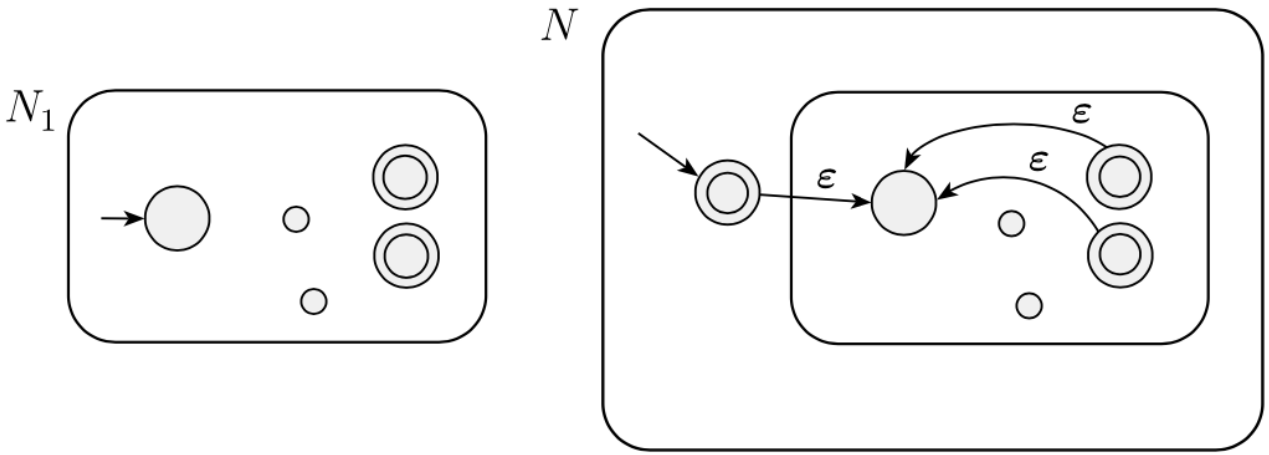
Concatenation of two FAs

If $L_1 = L(N_1)$ and $L_2 = L(N_2)$, then $L_1 \circ L_2$ is recognized (accepted) by the NFA N, i.e., $L_1 \circ L_2 = L(N)$.



The Star of a FA

If $L_1 = L(N_1)$, then L_1^* is recognized (accepted) by the NFA N, i.e., $L_1^* = L(N)$.



Regular Languages and Regular Expressions, I

Definition of Regular Languages over Alphabet Σ

The family $\mathcal{R}$ of **regular languages** (or **regular sets**) over alphabet Σ is defined recursively as follows:

1. The language $\varnothing$ is an element of $\mathcal{R}$ and, for every $\sigma \in \Sigma$, the language $\{\sigma\}$ is in $\mathcal{R}$.
2. For any languages L_1, L_2 in $\mathcal{R}$, the three languages $L_1 \cup L_2, L_1 \circ L_2$ and ${L_1}^*$ are elements of $\mathcal{R}$.

Notation of Regular Expressions

Given a regular language L, the **regular expression** of L is a notational representation of this language, in which

- set delimiters $\{\}$ in regular languages are replaced by parentheses $()$ in regular expressions and they are omitted whenever the rules of precedence allow it, and
- the union symbol $\cup$ in regular languages is replaced by the $+$ symbol in regular expressions.

Regular Languages and Regular Expressions, II

Examples of Regular Expressions of Regular Sets

Regular Set	**Regular Expression**
$\varnothing$	$\varnothing$
$\{\varepsilon\}$	ε
$\{a\}$ or $\{aba\}$ etc.	a or aba etc.
$\{a, b\}^*$	$(a + b)^*$
$\{aa, bb\} \cup \{ab, ba\}$	$aa + bb + ab + ba$

Examples of Regular Expressions of Described Regular Languages

Regular Language	**Regular Expression**
Strings beginning with an a and followed only by b's	ab^*
Strings x with $n_a(x) = 2$	$b^*ab^*ab^*$
Strings x with $n_{aa}(x) \geq 1$ or $n_{bb}(x) \geq 1$	$(a + b)^*(aa + bb)(a + b)^*$
Strings x ending in b with $n_{aa}(x) = 0$	$(b + ab)^+$

Regular Languages and Regular Expressions, III

Definition of Equality among Regular Expressions

Two **regular expressions are equal** if the languages (regular sets) they denote (describe) are equal.

Proposition: Properties of Regular Expressions

Let R, S and T be regular expressions over Σ. Then:

1. $R + S = S + R, R + \varnothing = \varnothing + R, R + R = R$, $(R + S) + T = R + (T + S)$,
2. $R\varepsilon = \varepsilon R = R, R\varnothing = \varnothing R = \varnothing, (RS)T = R(ST)$ (note that generally $RS \neq SR$),
3. $R(S + T) = RS + RT, (S + T)R = SR + TR$,
4. $R^* = R^*R^* = (R^*)^* = (\varepsilon + R)^*, \varnothing^* = \varepsilon^* = \varepsilon$,
5. $R^* = \varepsilon + \sum_{j=1}^{k} R^j + R^{k+1}R^*$, for all $k \geq 1$ (special case: $R^* = \varepsilon + RR^*$),

Regular Languages and Regular Expressions, IV

Proposition: Properties of Regular Expressions (cont.)

8. $(R+S)^* = (R^* + S^*)^* = (R^*S^*)^* = (R^*S)^*R^* = R^*(SR^*)^*$ (note that generally $(R+S)^* \neq R^* + S^*$),
9. $R^*R = RR^*, R(SR)^* = (RS)^*R$,
10. $(R^*S)^* = \varepsilon + (R+S)^*S, (RS^*)^* = \varepsilon + R(R+S)^*$,
11. **Arden Rule**: If $\varepsilon \notin S$, then

$$R = SR + T \text{ if and only if } R = S^*T,$$
$$R = RS + T \text{ if and only if } R = TS^*.$$

Direct Derivation of Regular Expressions from Descriptions of Regular Languages

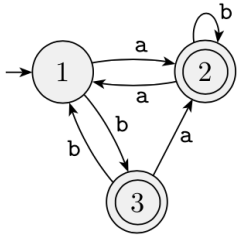
Example

Find the regular expression for the language over $\{a, b\}$ of strings with an odd number of a's.

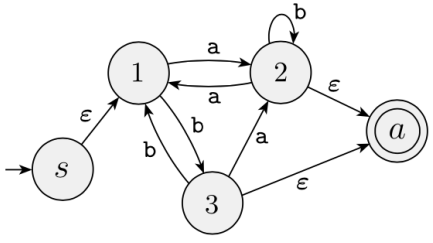
- Apparently, a string with an odd number of a's should contain at least one a and the additional a's grouped into pairs.
- Thus, taking the single a in the beginning of the string, the regular expression is:
$$b^*ab^*(ab^*ab^*)^*.$$
- Another way to group pairs of a's is considering them produced by the star closure of ab^*a and b's, i.e., another correct regular expression is:
$$b^*a(b + ab^*a)^*.$$
- Furthermore, by considering the single a to be placed at the end of these strings, we get two more correct regular expressions:
$$(b^*ab^*a)^*b^*ab^*, (b + ab^*a)^*ab^*.$$

Derivation of Regular Expressions given a DFA: The Method of Removal of States and Replacement of Transitions

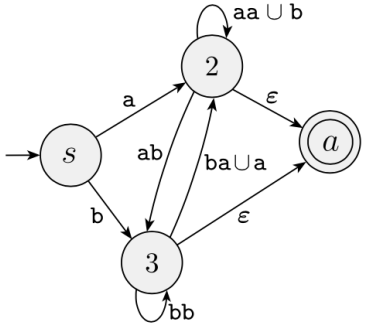
Example



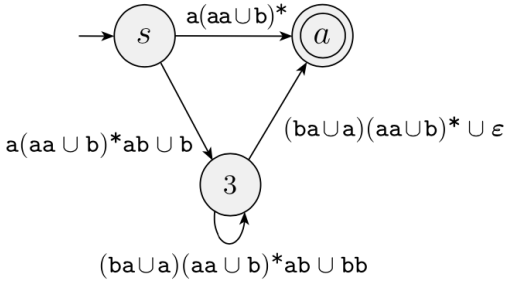
(a)



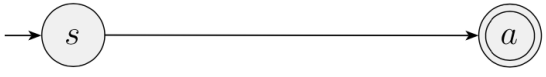
(b)



(c)

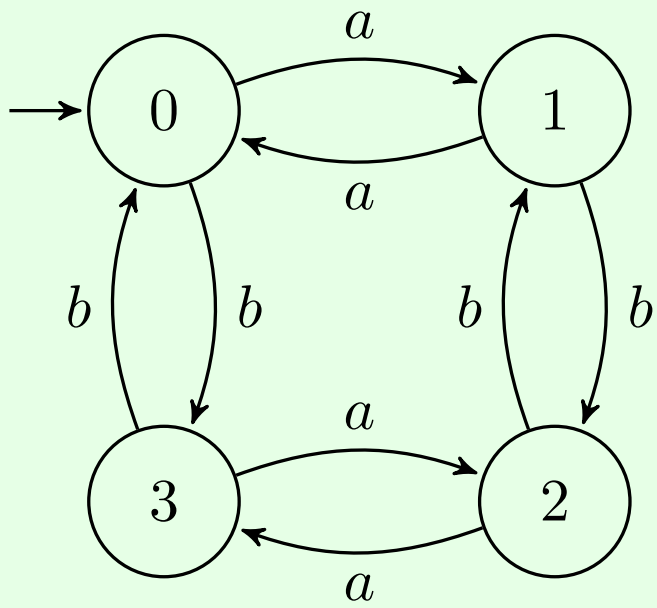


(d)



(a(aa∪b)*ab∪b)((ba∪a)(aa∪b)*ab∪bb)*((ba∪a)(aa∪b)*∪ε)∪a(aa∪b)*

Example



EVEN–EVEN: $F = \{0\}$ accepted are $n_a = 0 \bmod 2$ and $n_b = 0 \bmod 2$.

ODD–EVEN: $F = \{1\}$ accepted are $n_a = 1 \bmod 2$ and $n_b = 0 \bmod 2$.

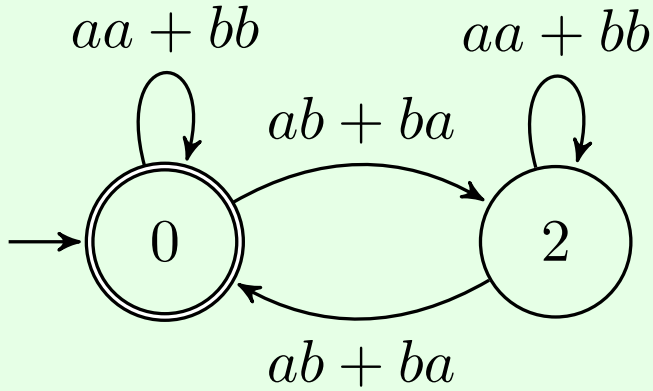
ODD–ODD: $F = \{2\}$ accepted are $n_a = 1 \bmod 2$ and $n_b = 1 \bmod 2$.

EVEN–ODD: $F = \{3\}$ accepted are $n_a = 0 \bmod 2$ and $n_b = 1 \bmod 2$.

Below, we are going to derive the regular expressions of these languages through the **method of removal of states and replacement of transitions by strings**, i.e., through the construction of an equivalent **transition graph**, which is a FA with strings as labels of transitions.

EVEN–EVEN

By removing & replacing nodes 1 and 3 (**both together**), the following transition graph is obtained:

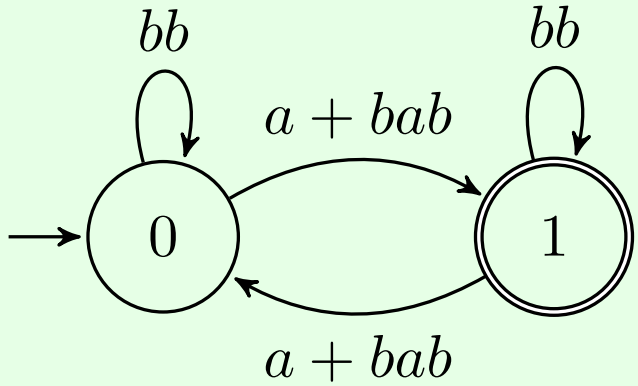


Thus, the regular expression describing the language EVEN–EVEN is:

$$R = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*.$$

ODD–EVEN

By removing & replacing nodes 2 and 3 (**both together**), the following transition graph is obtained:

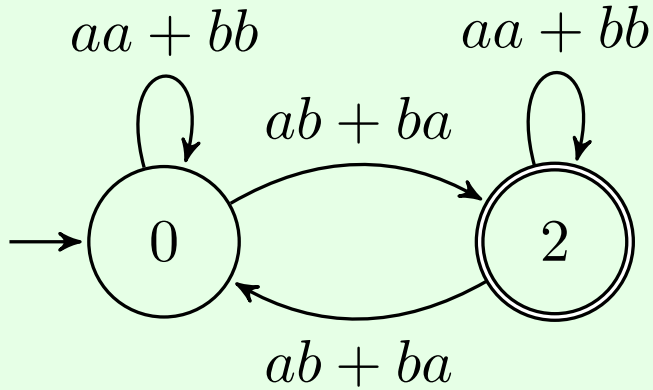


Thus, the regular expression describing the language ODD–EVEN is:

$$R = (bb)^*(a + bab)(bb + (a + bab)(bb)^*(a + bab))^*.$$

ODD–ODD

By removing & replacing nodes 1 and 3 (**both together**), the following transition graph is obtained:

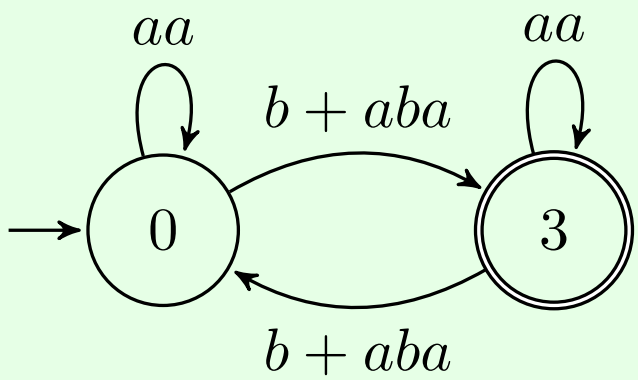


Thus, the regular expression describing the language ODD–ODD is:

$$R = (aa+bb)^*(ab+ba)(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*.$$

EVEN–ODD

By removing & replacing nodes 1 and 2 (**both together**), the following transition graph is obtained:



Thus, the regular expression describing the language EVEN–ODD is:

$$R = (aa)^*(b + aba)(aa + (b + aba)(aa)^*(b + aba))^*.$$

Derivation of Regular Expressions given a DFA: Kleene's Algorithm on Regular Expressions of Paths

Kleene's Algorithm on Regular Expressions of Paths

Let $M = (Q, \Sigma, q_0, F, \delta)$ be a DFA, assuming that $Q = \{1, 2, \ldots, n\}$ and $q_0 = 1$. For any positive integer $k \leq n$, denote by $R(i, j, k)$ a regular expression for the set of strings that M accepts when starting at state i and terminating at state j, only using states in the set $\{1, 2, \ldots, k\}$, i.e., without passing from any state $l > k$. Then, if $k < n$,

$$R(i, j, k+1) = R(i, j, k) + R(i, k+1, k) R(k+1, k+1, k)^* R(k+1, j, k),$$

where

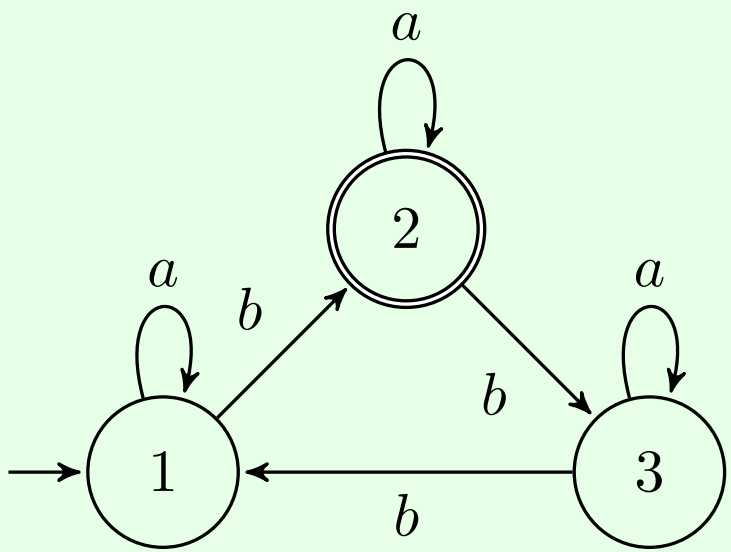
$$R(i, j, 0) = \begin{cases} \{\sigma \in \Sigma \mid \delta(i, \sigma) = j\}, & \text{if } i \neq j, \\ \{\varepsilon\} \cup \{\sigma \in \Sigma \mid \delta(i, \sigma) = j\}, & \text{if } i = j, \end{cases}$$

and, moreover, the regular expression of the language accepted by M is:

$$R = \bigcup_{f \in F} R(1, f, n).$$

Example 1

Let M be the following FA:



Calculating regular expressions from the to down, we get:

$$
\begin{aligned}
R &= R(1,2,3) \\
R(1,2,3) &= R(1,2,2) + R(1,3,2)R(3,3,2)^* R(3,2,2) \\
R(1,2,2) &= R(1,2,1) + R(1,2,1)R(2,2,1)^* R(2,2,1) \\
R(1,3,2) &= R(1,3,1) + R(1,2,1)R(2,2,1)^* R(2,3,1) \\
R(3,3,2) &= R(3,3,1) + R(3,2,1)R(2,2,1)^* R(2,3,1) \\
R(3,2,2) &= R(3,2,1) + R(3,2,1)R(2,2,1)^* R(2,2,1)
\end{aligned}
$$

Example 1 (cont.)

$$
\begin{aligned}
R(1,2,1) &= R(1,2,0) + R(1,1,0)R(1,1,0)^* R(1,2,0) \\
R(2,2,1) &= R(2,2,0) + R(2,1,0)R(1,1,0)^* R(1,2,0) \\
R(1,3,1) &= R(1,3,0) + R(1,1,0)R(1,1,0)^* R(1,3,0) \\
R(2,3,1) &= R(2,3,0) + R(2,1,0)R(1,1,0)^* R(1,3,0) \\
R(3,3,1) &= R(3,3,0) + R(3,1,0)R(1,1,0)^* R(1,3,0) \\
R(3,2,1) &= R(3,2,0) + R(3,1,0)R(1,1,0)^* R(1,2,0)
\end{aligned}
$$

Now, we know the expressions for $k = 0$:

$$
\begin{aligned}
&R(1,1,0) = \varepsilon + a, && R(1,2,0) = b, && R(1,3,0) = \varnothing \\
&R(2,1,0) = \varnothing, && R(2,2,0) = \varepsilon + a, && R(2,3,0) = b \\
&R(3,1,0) = b, && R(3,2,0) = \varnothing, && R(3,3,0) = \varepsilon + a
\end{aligned}
$$

Example 1 (cont.)

Thus, substituting the values in the expressions for $k = 1$, we get:

$$
\begin{aligned}
R(1,2,1) &= a^*b \\
R(2,2,1) &= \varepsilon + a \\
R(1,3,1) &= \varnothing \\
R(2,3,1) &= b \\
R(3,3,1) &= \varepsilon + a \\
R(3,2,1) &= ba^*b
\end{aligned}
$$

which result:

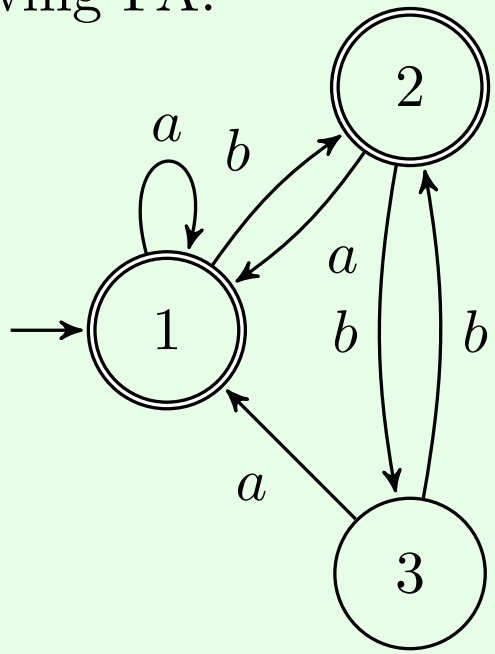
$$
\begin{aligned}
R(3,2,2) &= ba^*ba^* \\
R(3,3,2) &= \varepsilon + a + ba^*ba^*b \\
R(1,3,2) &= a^*ba^*b \\
R(1,2,2) &= a^*ba^*
\end{aligned}
$$

Therefore, the wanted regular expression is:

$$R = R(1,2,3) = a^*ba^* + a^*ba^*b(a + ba^*ba^*b)^*ba^*ba^*.$$

Example 2

Let M be the following FA:



Calculating regular expressions from the to down, we get:

$$R = R(1,1,3) + R(1,2,3)$$
$$R(1,1,3) = R(1,1,2) + R(1,3,2)R(3,3,2)^* R(3,1,2)$$
$$R(1,2,3) = R(1,2,2) + R(1,3,2)R(3,3,2)^* R(3,2,2)$$
$$R(1,1,2) = R(1,1,1) + R(1,2,1)R(2,2,1)^* R(2,1,1)$$
$$R(1,3,2) = R(1,3,1) + R(1,2,1)R(2,2,1)^* R(2,3,1)$$
$$R(3,3,2) = R(3,3,1) + R(3,2,1)R(2,2,1)^* R(2,3,1)$$
$$R(3,1,2) = R(3,1,1) + R(3,2,1)R(2,2,1)^* R(2,1,1)$$
$$R(1,2,2) = R(1,2,1) + R(1,2,1)R(2,2,1)^* R(2,2,1)$$
$$R(3,2,2) = R(3,2,1) + R(3,2,1)R(2,2,1)^* R(2,2,1)$$

Example 2 (cont.)

Thus, from the table of regular expressions for $n = 0$:

i	$R(i,1,0)$	$R(i,2,0)$	$R(i,3,0)$
1	$\varepsilon + a$	b	$\varnothing$
2	a	ε	b
3	a	b	ε

we obtain the values in the following two tables:

i	$R(i,1,1)$	$R(i,2,1)$	$R(i,3,1)$
1	a^*	a^*b	$\varnothing$
2	aa^*	$\varepsilon + aa^*b$	b
3	aa^*	a^*b	ε

i	$R(i,1,2)$	$R(i,2,2)$	$R(i,3,2)$
1	$a^*(baa^*)^*$	$a^*(baa^*)^*b$	$a^*(baa^*)^*bb$
2	$aa^*(baa^*)^*$	$(aa^*b)^*$	$(aa^*b)^*b$
3	$aa^* + a^*baa^*(baa^*)^*$	$a^*b(aa^*b)^*$	$\varepsilon + a^*b(aa^*b)^*b$

Substituting the values in the expressions for $k = 3$ from the above tables, we find the final form of the regular expression for M. (Algebra omitted.)

The Pumping Lemma for Regular Languages

Theorem: The Pumping Lemma for Regular Languages

If L is a regular language, then there is a positive integer n (typically, n is the number of states of the DFA accepting L) such that, if $x \in L$ and $|x| \geq n$, then there exist $u, v, w \in \Sigma^*$ such that $x = uvw$ and:

- $|uv| \leq n$,
- $|v| > 0$ (i.e, $v \neq \varepsilon$), and
- for each integer $m \geq 0, uv^m w \in L$.

Corollary

Let the regular language L be accepted by a DFA with n states. Then L is infinite if and only if there is $x \in L$ such that $n \leq |x| < 2n$.

Theorem

The class of regular languages is closed under complement.

A List of Nonregular Languages, I

I.1 $L = \{a^{i^2} \mid i \in \mathbb{Z}, i \geq 0\}$

I.2 $L = \{a^p \mid p \text{ prime}\}$

I.3 $L = \{a^{i^3+3i^2-2i} \mid i \in \mathbb{Z}, i \geq 0\}$

II.1 $L = \{a^i b^i \mid i \in \mathbb{Z}, i \geq 0\}$

II.2 $L = \{a^i b^{pj+q} \mid i \in \mathbb{Z}, i \geq 0\}$ $(p, q \in \mathbb{Z}, p + q \neq 0)$

II.3 $L = \{a^{pi+q} b^j \mid i \in \mathbb{Z}, i \geq 0\}$ $(p, q \in \mathbb{Z}, p + q \neq 0)$

II.4 $L = \{a^i b^j a^i \mid i, j \in \mathbb{Z}, i, j \geq 0\}$

II.5 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i \neq j\}$

II.6 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i > j\}$

II.7 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i < j\}$

II.8 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i \neq pj + q\}$ $(p, q \in \mathbb{Z}, p + q \neq 0)$

II.9 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, j \neq pi + q\}$ $(p, q \in \mathbb{Z}, p + q \neq 0)$

II.10 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, p_1 j + q_1 \leq i \leq p_2 j + q_2\}$
$(p_k, q_k \in \mathbb{Z}, k = 1, 2)$

II.11 $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, p_1 i + q_1 \leq j \leq p_2 i + q_2\}$
$(p_k, q_k \in \mathbb{Z}, k = 1, 2)$

II.12 $L = \{a^{p_1 i+q_1} b^{p_2 j+q_2} \mid i, j \in \mathbb{Z}, i, j \geq 0\}$ $(p_k, q_k \in \mathbb{Z}, k = 1, 2)$

II.13 $L = \{x \in (a + b)^* \mid x \neq x^R\}$

A List of Nonregular Languages, II

III.1 $L = \{xx \mid x \in (a+b)^*\}$

III.2 $L = \{xyx \mid x, y \in (a+b)^+\}$

III.3 $L = \{xx^R \mid x \in (a+b)^+\}$

III.4 $L = \{xx^R y \mid x, y \in (a+b)^+\}$

III.5 $L = \{x \in (a+b)^* \mid n_a(x) = n_b(x)\}$

III.6 $L = \{x \in (a+b)^* \mid n_a(x) \neq n_b(x)\}$

III.7 $L = \{x \in (a+b)^* \mid n_a(x) = p_1 + q_1 n_b(x)\}$
$(p_1, q_1 \in \mathbb{Z}, p_1 \geq 0, q_1 \neq 1)$

III.8 $L = \{x \in (a+b)^* \mid n_a(x) \neq p_1 + q_1 n_b(x)\}$
$p_1, q_1 \in \mathbb{Z}, p_1 \geq 0, q_1 \neq 1)$

III.9 $L = \{x \in (a+b)^* \mid n_b(x) = p_2 + q_2 n_a(x)\}$
$(p_2, q_2 \in \mathbb{Z}, p_2 \geq 0, q_2 \neq 1)$

III.10 $L = \{x \in (a+b)^* \mid n_b(x) \neq p_2 + q_2 n_a(x)\}$
$p_2, q_2 \in \mathbb{Z}, p_2 \geq 0, q_2 \neq 1)$

III.11 $L = \{x \in (a+b)^* \mid p_1 + q_1 n_a(x) \leq n_b(x) \leq p_2 + q_2 n_a(x)\}$
$(p_k, q_k \in \mathbb{Z}, p_k \geq 0, q_k \neq 1, k = 1, 2)$

III.12 $L = \{x \in (a+b)^* \mid p_1 + q_1 n_b(x) \leq n_a(x) \leq p_2 + q_2 n_b(x)\}$
$(p_k, q_k \in \mathbb{Z}, p_k \geq 0, q_k \neq 1, k = 1, 2)$

III.13 $L = \{x \in (a+b)^* \mid n_a(x) \geq n_b(x)\}$

III.14 $L = \{x \in (a+b)^* \mid n_a(x) \leq n_b(x)\}$

IV.1 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k \geq 0, i = j \text{ or } j \neq k\}$

IV.2 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k \geq 0, i \neq j \text{ or } j \neq k\}$

IV.3 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k \geq 0, k = pi + qj\}$ $(p, q \in \mathbb{Z}, pq \neq 0)$

IV.4 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k \geq 0, k \neq qi + j\}$

IV.5 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k \geq 0, k \neq pi + qj\}$ $(p, q \in \mathbb{Z}, pq \neq 0)$

IV.6 $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{Z}, i, j, k > 0, k = |i - j|\}$

Examples of Pumping Lemma, I

Example 1 (I.1)

Use the pumping lemma to prove that the language $L = \{a^{i^2} \mid i \in \mathbb{Z}, i \geq 0\}$ is not regular.

Assume that $L = \{a^{i^2} \mid i \in \mathbb{Z}, i \geq 0\}$, i.e., the language of all strings which are perfect squares of a, is regular. Then, by the PL, there exists integer $n \geq 1$, and, for $x = a^{n^2}$, since $|x| = n^2 > n$, the PL would necessitate that $x = a^{n^2} = uvw$, for $u, v, w \in \Sigma^*$.

Now, since $|uv^2w| = |uvw| + |v|, |uvw| < |uv^2w|$ and $|v| < |uv| \leq n$, as $|v| > 0$, we get $n^2 = |uvw| \leq n^2 + n < (n+1)^2$. In other words, the length of uv^2w lies between two squares of two consecutive integers, which implies that uv^2w cannot be a perfect square and, thus, $uv^2w \notin L$. But this is a contradiction, because PL implies that $uv^mw \in L$, for any integer $m \geq 2$, and, in particular, for $m = 2$. Consequently, L cannot a regular language.

Examples of Pumping Lemma, II

Example 2

Use the pumping lemma to prove that the language $L = \{a^i b^i \mid i \in \mathbb{Z}, i \geq 0\}$ is not regular.

Assume that $L = \{a^i b^i \mid i \in \mathbb{Z}, i \geq 0\}$ is regular. Then, by the PL, there exists integer $n \geq 1$, and, for $x = a^n b^n$, since $|x| = 2n > n$, the PL would necessitate that $x = a^n b^n = uvw$, for $u, v, w \in \Sigma^*$.

Claim: v contains only a's (at least one a).

Proof of claim: If v contained both a's and b's, $v = a^p b^q$, for some integers p, q such that $p \geq 0, q \geq 1$. Then, since w follows u (as a string in L), w should only contain b's, say $w = b^s$, for some integer $s \geq 0$. Moreover, since u precedes v, u should contain only a's, say $u = a^r$, for some integer $r \geq 0$. Alltogether, u, v and w, are $x = a^{r+p} b^{q+s} = a^n b^n$, which implies that $r + p = n$. In addition, $|uv| = r + p + q = n + q \geq n + 1 > n$ and this contradicts the first consequence of the PL that $|uv| \leq n$. Therefore, the claim is shown.

To continue with the proof, we have that $u = a^\alpha, v = a^\beta$ and $w = a^\gamma b^\delta$ and, totally, $\alpha+\beta+\gamma = \delta = n$. On the other side, the third consequence of the PL implies that, for all integer $m \geq 0$, $uv^m w \in L$, which gives that $\alpha + m\beta + \gamma = n_a(uv^m w) = n_b(uv^m w) = \delta$. Therefore, $\alpha + m\beta + \gamma = \delta = \alpha + \beta + \gamma$ or $\beta(m - 1) = 0$, where $\beta > 0$ (because $|v| > 0$), which generates the contradiction $m = 1$ (because the previous ought to be true for all integer $m \geq 0$). Consequently, L is not a regular language.

Examples of Pumping Lemma, III

Example 3

Use the pumping lemma to prove that the language $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i > j\}$ is not regular.

Assume that $L = \{a^i b^j \mid i, j \in \mathbb{Z}, i, j \geq 0, i > j\}$ is regular. Then, by the PL, there exists integer $n \geq 1$, and, for $x = a^{n+1}b^n$, since $|x| = 2n+1 > n$, the PL would necessitate that $x = a^{n+1}b^n = uvw$, for $u, v, w \in \Sigma^*$.

Similarly to how the claim in the proof of Example 2 was proved (*please, repeat the proof of such a claim in this example and in any similar problem that you are solving!*), it can be shown that v contains only a's (at least one a). Thus, we have that $u = a^\alpha, v = a^\beta$ and $w = a^\gamma b^\delta$ and, totally, $\alpha + \beta + \gamma = n + 1$ and $\delta = n$. On the other side, the third consequence of the PL implies that, for all integer $m \geq 0$, $uv^m w \in L$, and, hence, for the particular value $m = 0$, $uw \in L$, which implies that $\alpha + \gamma = n_a(uw) > n_b(uw) = \delta$. However, since $\alpha + \gamma = n + 1 - \beta$ and $\delta = n$, the last inequality yields that $n+1-\beta > n$, i.e., or $\beta < 1$, which is a contradiction, because β is the length of v and by the PL it is assumed that $|v| = \beta > 1$. Consequently, L is not a regular language.

Examples of Pumping Lemma, IV

Example 4

Use the pumping lemma to prove that the language $L = \{xx \mid x \in (a+b)^*\}$ is not regular.

Assume that $L = \{xx \mid x \in (a+b)^*\}$ is regular. Then, by the PL, there exists integer $n \geq 1$, and, for $x = a^n b a^n b$, since $|x| = 2n + 2 > n$, the PL would necessitate that $x = a^n b a^n b = uvw$, for $u, v, w \in \Sigma^*$.

Similarly to how the claim in the proof of Example 2 was proved (*please, repeat the proof of such a claim in this example and in any similar problem that you are solving!*), it can be shown that v contains only a's (at least one a). Thus, the third consequence of the PL implies that, for all integer $m \geq 0$, $uv^m w \in L$, and, hence, for the particular value $m = 2$, $uvvw \in L$, which implies that $uv = vw$. However, since v contains only a's (at least one a), the only case that it could be $uv = vw$ would be that the suffix of w was a, which is a contradiction of the fact that $uvw = a^n b a^n b$, i.e., the fact that, by construction of x, w ends in b. Consequently, L is not a regular language.

Examples of Pumping Lemma, V

Example 5

Use the pumping lemma to prove that the language $L = \{x \in (a+b)^* \mid n_a(x) = n_b(x)\}$ is not regular.

Assume that $L = \{x \in (a+b)^* \mid n_a(x) = n_b(x)\}$ is regular. Then, by the PL, there exists integer $n \geq 1$, and, for $x = a^n b^n$, since $|x| = 2n > n$, the PL would necessitate that $x = a^n b^n = uvw$, for $u, v, w \in \Sigma^*$.

Similarly to how the claim in the proof of Example 2 was proved (*please, repeat the proof of such a claim in this example and in any similar problem that you are solving!*), it can be shown that v contains only a's (at least one a). Thus, we have that $u = a^\alpha, v = a^\beta$ and $w = a^\gamma b^\delta$ and, totally, $\alpha + \beta + \gamma = \delta = n$. On the other side, the third consequence of the PL implies that, for all integer $m \geq 0$, $uv^m w \in L$, which gives that $\alpha + m\beta + \gamma = n_a(uv^m w) = n_b(uv^m w) = \delta$. Therefore, $\alpha + m\beta + \gamma = \delta = \alpha + \beta + \gamma$ or $\beta(m-1) = 0$, where $\beta > 0$ (because $|v| > 0$), which generates the contradiction $m = 1$ (because the previous ought to be true for all integer $m \geq 0$). Consequently, L is not a regular language.

The Myhill–Nerode Theorem

Definition of Indistinguishable Strings in a Language

Let L a language over Σ and let $x, y \in \Sigma^*$. We say that x and y are **indistinguishable with respect to** L and we write $x \approx_L y$ if, for all $z \in \Sigma^*$, either both xz and $yz \in L$ or neither is. Furthermore, $\approx_L$ can be proved to be an equivalence relation on Σ^*.

The Myhill–Nerode Theorem

A language L is regular if and only if the number of equivalence classes of $\approx_L$ is finite.

II. CONTEXT–FREE LANGUAGES

Context–Free Grammars, I

Definition: **Context-Free Grammars**

A **context–free grammar** (**CFG**) is a 4–tuple $G = (V, \Sigma, S, P)$, where

- V is a finite set of **variables**, $S \in V$ is the **start variable**,
- Σ is a finite set of **terminal symbols** or **terminals** such that $V \cap \Sigma = \varnothing$, and
- P is a finite set, the elements of which are called **grammar rules** or **productions** and they are of the form

$$A \to \alpha,$$

where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

Context–Free Grammars, II

Notation

Given a CFG $G = (V, \Sigma, S, P)$ and two strings $\alpha \in (V \cup \Sigma)^* V (V \cup \Sigma)^*$ and $\beta \in (V \cup \Sigma)^*$, a **derivation** from α to β, denoted as

$$\alpha \implies \beta,$$

is an one–step process to obtain β from α by using a rule in P in order to replace the single occurrence of S in α by the right–hand side of that rule. Furthermore, the notations

$$\alpha \implies^n \beta \text{ or } \alpha \implies_G^n \beta$$

refer to a sequence of (exactly) n steps of derivations and the notations

$$\alpha \implies^* \beta \text{ or } \alpha \implies_G^* \beta$$

refer to a sequence of 0 or more steps of derivations.

Context–Free Grammars, III

Definition: **The Language Generated by a CFG**

If $G = (V, \Sigma, S, P)$ is a CFG, the **language generated by** G is

$$L(G) = \{x \in \Sigma^* \mid S \Longrightarrow_G^* x\}.$$

A language L is a **context–free language** (**CFL**) if there is a CFG G with $L = L(G)$.

Notation

The symbol "|" inside the set of productions denotes a disjunction (or). For instance, $P = \{S \to aSb \mid ab\}$ includes

$$S \to aSb,$$

$$S \to ab.$$

Example 1

If $G = (\{S\}, \{a, b\}, S, \{S \to aSb \mid ab\})$, then $L(G) = \{a^n b^n \mid n \in \mathbb{Z}, n \geq 1\}$.

Context–Free Grammars, IV

Example 2

If G is a CFG with productions $S \to aSa \,|\, aBa, B \to bB \,|\, b$, then $L(G) = \{a^i b^j a^i \mid i, j \in \mathbb{Z}, i \geq 0, j > 0\}$.

For every nonnegative integers n, m, k,

$$S \Longrightarrow^n a^n S a^n \Longrightarrow^m a^n \left(a^m B a^m\right) a^n \Longrightarrow^k a^{n+m} b^k B a^{n+m}$$
$$\Longrightarrow a^{n+m} b^{k+1} a^{n+m}.$$

Example 3

Find the CFG generating $L = \{a^i b^{2i} \mid i \in \mathbb{Z}, i \geq 0\}$.

$$P = \{S \to aSbb \mid \varepsilon\}$$

Example 4

Find the CFG generating $L = \{x \in (a+b)^* \mid n_a(x) = n_b(x) \geq 0\}$.

$$P = \{S \to SS \mid aSb \mid bSa \mid \varepsilon\}$$

Derivation Trees, I

Definition

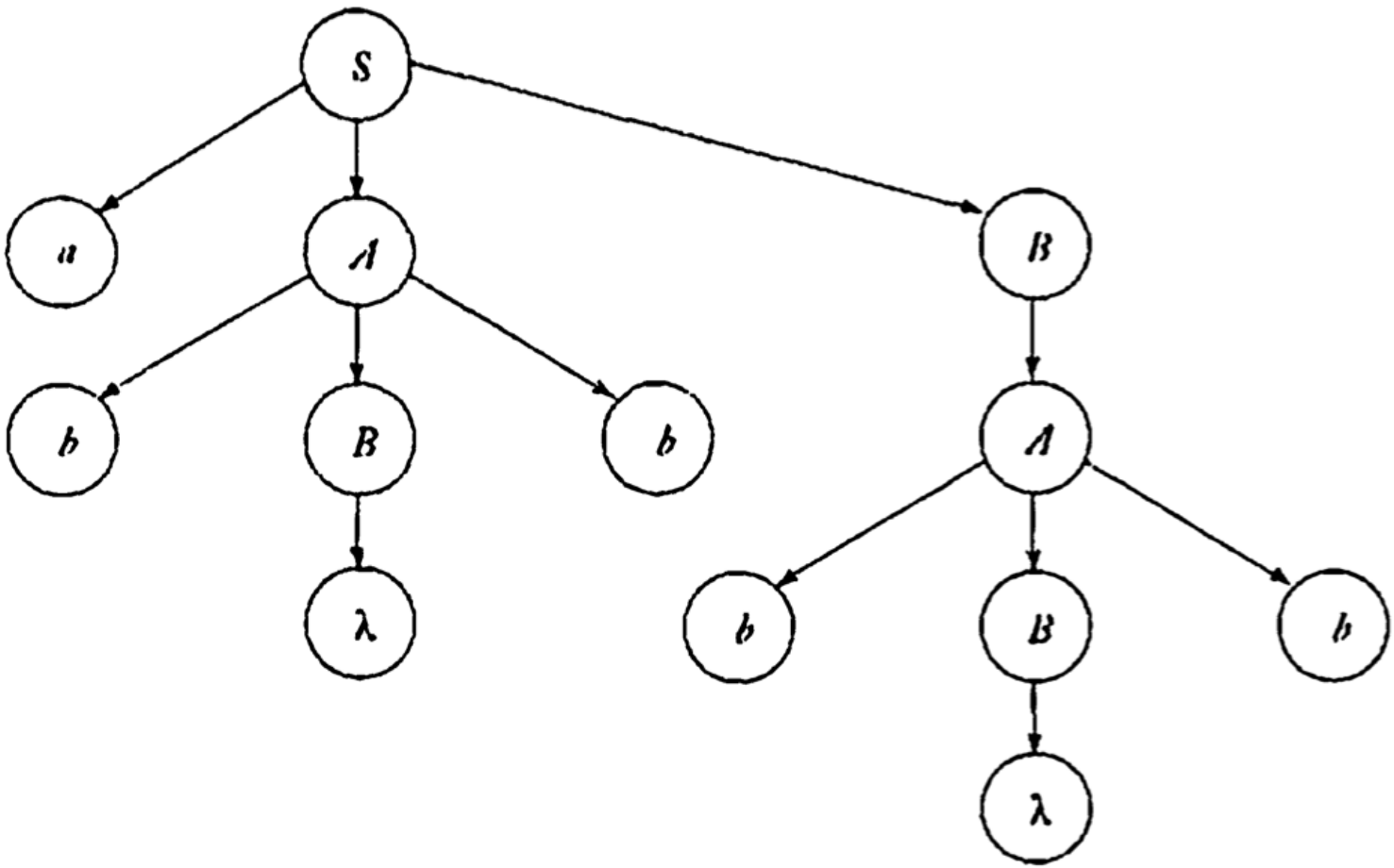
Let $G = (V, \Sigma, S, P)$ a CFG. A **derivation tree** (or **parse tree**) for G is an ordered tree such that:

- the root is labeled S;
- every leaf has a label from $\Sigma \cup \{\varepsilon\}$;
- every interior vertex has a label from V;
- if a vertex has label A, and its children are labeled (from left to right) $a_1, a_2, \ldots, a_n$, where $a_j \in V \cup \Sigma \cup \{\varepsilon\}$, for $j = 1, 2, \ldots, n$, then P contains a production of the form $P \to a_1 a_2 \cdots a_n$.

Derivation Trees, II

Example of Derivation Tree

The CFG G with productions $S \to aAB, A \to bBb$, $B \to A \mid \varepsilon$ has the following derivation tree (among other derivation trees):



Derivation Trees, III

Definition

The **yield** of a tree is the string of terminals (symbols) obtained by reading the leaves of the tree from left to right, omitting any ε's encountered. The precise meaning of the ordering "from left to right" is that terminals are yielded in the order they are encountered when the tree is traversed in a depth–first manner, always taking the leftmost ununexplored branch.

Example of a Yield

The yield of the derivation tree of the previous example is the string $abbbb$. Notice that the corresponding CFL is the set of all strings over $\{a, b\}$ starting with a and followed by an even positive number of b's.

Theorem

If G is a CFG, then, for every $x \in L(G)$, there exists a derivation tree of G whose yield is x. Conversely, the yield of any derivation tree is in $L(G)$.

Ambiguity

Definition

A derivation in a CFG is a **leftmost derivation** (**LMD**) if, at each step, a production is applied to the leftmost variable-occurrence in the current string. A **rightmost derivation** (**RMD**) is defined similarly.

Theorem

If G is a CFG, then, for every $x \in L(G)$, the following three statements are quivalent:

1. x has more than one derivation tree;
2. x has more than one leftmost derivation;
3. x has more than one rightmost derivation.

Definition

A CFG G is **ambiguous** if, for at least one $x \in L(G)$, x has more than one derivation tree (or, equivalently, more than one leftmost derivation).

Equivalent Context–Free Grammars, I

Definition

Let $G_1 = (V_1, \Sigma, S_1, P_1)$ and $G_2 = (V_2, \Sigma, S_2, P_2)$ be two CFGs over the same set of terminals (alphabet) Σ. The grammars G_1 and G_2 are called **equivalent CFGs** if they are generating the same language, i.e., if $L(G_1) = L(G_2)$.

Notation

Let $L_{a=b}$ denote the (nonregular) language of strings with equal number of a's and b's, i.e.,

$$L_{\sharp a=\sharp b} = L\{x \in (a+b)^* \mid n_a(x) = n_b(x) \geq 0\}.$$

Proposition

The following two CFGs $G_1 = \{\{S_1\}, \Sigma, S_1, P_1\}$ and $G_2 = \{\{S_2, A, B\}, \Sigma, S_2, P_2\}$ are equivalent, both generating the language $L_{\sharp a=\sharp b}$, when

$P_1 = \{S_1 \to S_1S_1 \mid aS_1b \mid bS_1a \mid \varepsilon\}$,

$P_2 = \{S_2 \to aB \mid bA \mid \varepsilon, A \to aS_2 \mid bAA, B \to bS_2 \mid aBB\}$.

Equivalent Context–Free Grammars, II

Proposition

Prove that $L(G_1) = L_{\sharp a = \sharp b}$.

Proposition

Prove that $L(G_1) \subseteq L_{\sharp a = \sharp b}$.

Using **induction on the number of derivations** n, we are going to show that: If, for every integer $n \geq 1$, and $x \in L(G_1)$, $S_1 \Longrightarrow^n x$, then $n_a(x) = n_b(x)$, i.e., $x \in L_{\sharp a = \sharp b}$.

Base case: True, for $n = 1$, because $S_1 \Longrightarrow \varepsilon \in L_{\sharp a = \sharp b}$.
Inductive hypothesis: Suppose that there exists integer $k \geq 1$ such that, for all nonnegative integers $i \leq k$, if $x \in L(G_1)$ such that $S_1 \Longrightarrow^i x$, then $x \in L_{\sharp a = \sharp b}$.
Inductive step: We need to show that, if $y \in L(G_1), S_1 \Longrightarrow^{k+1} y$, then $y \in L_{\sharp a = \sharp b}$. Notice that, in this case, there exists $z \in (V_1 \cup \Sigma)^*$ such that $S_1 \in z$ and $S_1 \Longrightarrow^{k+1-j} z \Longrightarrow^j y$, where $j \geq 1$ (because we need at least one rule of P_1 in order to replace a variable by a terminal). However, since $k + 1 - j < k$, the inductive hypothesis implies that $n_a(z) = n_b(z)$ and the subsequent application of (any) rules of P_1 does not change the equality in the number of a's and b's, meaning that $y \in L_{\sharp a = \sharp b}$.

Equivalent Context–Free Grammars, III

Proposition

Prove that $L_{\sharp a=\sharp b} \subseteq L(G_1)$.

Using **induction on the length of strings** n, we are going to show that: For every integer $n \geq 0$, if $x \in L_{\sharp a=\sharp b}$ with $|x| = n$, then $S_1 \implies^* x$, i.e., $x \in L(G_1)$.

Base case: True, for $n = 0$, because $n_a(\varepsilon) = n_b(\varepsilon) = 0$ and $S_1 \implies \varepsilon$.
Inductive hypothesis: Suppose that there exists integer $k \geq 1$ such that, for all i, $0 \leq i \leq k$, if $x \in L_{\sharp a=\sharp b}$ and $|x| = i$, then $S_1 \implies^* x$.
Inductive step: We need to show that, if $y \in L_{\sharp a=\sharp b}$ with $|y| = k + 1$, then $S_1 \implies^* y$. Depending on how it starts and ends, y can be one of the following four forms: (i) $y = azb$, (ii) $y = bza$, (iii) $y = aza$, (iv) $y = bzb$, where in all cases $z \in \Sigma^*$ has $|z| = k + 1 - 2 = k - 1$. Actually, it suffices to do cases (i) and (iii) (because (ii) can be treated similarly to (i) and (iv) similarly to (iii)).

(i) If $y = azb$, since $n_a(z) = n_b(z)$ and $|z| = k-1$, the inductive hypothesis implies that $S_1 \implies^* z$. Hence, $S_1 \implies aS_1b \implies^* azb = y$, which was what we needed to show.

(iii) If $y = aza$, since $n_b(z) = n_a(z) + 2$ and $|z| = k - 1$, z should be represented as $z = u_1bu_2bu_3$, for three strings $u_p \in L_{\sharp a=\sharp b}$, for $p = 1, 2, 3$, such that $0 \leq |u_p| \leq |u_1| + |u_2| + |u_3| = |z| - 2 = k - 3$. Then, by the inductive hypothesis, $S_1 \implies^* u_1|u_2|u_3$ and, moreover, $S_1 \implies S_1S_1 \implies S_1S_1S_1 \implies^2 aS_1bS_1bS_1a \implies^* aS_1bu_2bS_1a \implies^* au_1bu_2bS_1a \implies^* au_1bu_2bu_3a = aza = y$ and that was what we wanted to show.

Equivalent Context–Free Grammars, IV

Proposition

Prove that $L(G_2) = L_{\sharp a = \sharp b}$.

Proposition

Prove that $L(G_2) \subseteq L_{\sharp a = \sharp b}$.

Using **induction on the number of derivations** n, we are going to show that: If, for every integer $n \geq 1$, $S_2 \Longrightarrow^n x \in \Sigma^*$, i.e., $x \in L(G_2)$, then $n_a(x) = n_b(x)$, i.e., $x \in L_{\sharp a = \sharp b}$.

Base case: True, for $n = 1$, because $S_2 \Longrightarrow \varepsilon \in L_{\sharp a = \sharp b}$.
Inductive hypothesis: Suppose that there exists integer $k \geq 1$ such that, for all nonnegative integers $i \leq k$, if $S_2 \Longrightarrow^i x \in \Sigma^*$, then $x \in L_{\sharp a = \sharp b}$.
Inductive step: We need to show that, if $S_2 \Longrightarrow^{k+1} y \in \Sigma^*$, then $y \in L_{\sharp a = \sharp b}$. Notice that, in this case, there exists $z \in (V_2 \cup \Sigma)^*$ such that $S_2, A, B \in z$ and $S_2 \Longrightarrow^{k+1-j} z \Longrightarrow^j y$, where $j \geq 1$ (because we need at least one rule of P_2 in order to replace a variable by a terminal). However, since $k + 1 - j < k$, the inductive hypothesis implies that $n_a(z) = n_b(z)$ and the subsequent application of (any) rules of P_2 does not change the equality in the number of a's and b's, meaning that $y \in L_{\sharp a = \sharp b}$.

Equivalent Context–Free Grammars, V

Proposition

Prove that $L_{\sharp a=\sharp b} \subseteq L(G_2)$.

Using **induction on the length of strings** n, we are going to show that: For every integer $n \geq 0$, if $x \in L_{\sharp a=\sharp b}$ with $|x| = n$, then $S_2 \implies^* x$, i.e., $x \in L(G_2)$.

Base case: True, for $n = 0$, because $n_a(\varepsilon) = n_b(\varepsilon) = 0$ and $S_2 \implies \varepsilon$.
Inductive hypothesis: Suppose that there exists integer $k \geq 1$ such that, for all i, $0 \leq i \leq k$, if $x \in L_{\sharp a=\sharp b}$ and $|x| = i$, then $S_2 \implies^* x$. Actually, k should be taken even, because any string with equal number of a's and b's has even length.
Inductive step: We need to show that, if $y \in L_{\sharp a=\sharp b}$ with $|y| = k + 2$, then $S_2 \implies^* y$. Depending on how it starts and ends, y can be one of the following three forms: (i) $y = abz$, (ii) $y = baz$, (iii) y does not start either with ab or ba, where in the first two cases $|z| = k + 1 - 2 = k - 1$. Actually, it suffices to do cases (i) and (iii) (since (ii) can be treated similarly to (i)).

(i) If $y = abz$, since $n_a(z) = n_b(z)$ and $|z| = k - 1$, the inductive hypothesis implies that $S_2 \implies^* z$. Hence, $S_2 \implies aB \implies abS_2 \implies^* abz = y$, which was what we needed to show.

Equivalent Context–Free Grammars, V (cont.)

Proof of Proposition (cont.)

(iii) If y does not start either with ab or ba, then necessarily y is represented as $y = u_1u_2$, where $u_1, u_2 \in L_{\sharp a=\sharp b}$ and $2 \leq |u_1|, |u_2| < |u_1| + |u_2| = |y| = k + 2$, i.e., $0 \leq |u_1|, |u_2| \leq k$. Then the inductive hypothesis implies that $S_2 \Longrightarrow^* u_1 \mid u_2$. Moreover, since all the grammar rules in the CFG G_2 have a variable at the rightmost place and the only variable terminating (to a terminal) is S_2, accoding to the Theorem in the previous section about ambiguity, every $u \in L_{\sharp a=\sharp b}$ with $|u| \leq k$ can be produced as $S_2 \Longrightarrow^* uS_2$. Therefore, $S_2 \Longrightarrow^* u_1S_2 \Longrightarrow^* u_1u_2 = y$, which was what we needed to show.

Remark

In the previous Example, both $G1$ and G_2 are ambiguous. Exampes:

$S_1 \Longrightarrow S_1S_1 \Longrightarrow aS_1bS_1 \Longrightarrow abS_1abS_1 \Longrightarrow abab$

$S_1 \Longrightarrow aS_1b \Longrightarrow abS_1ab \Longrightarrow abab$

$S_2 \Longrightarrow bA \Longrightarrow bbAAA \Longrightarrow bbaS_2A \Longrightarrow bbabAA \Longrightarrow bbabbAAA \Longrightarrow bbabbaaa$

$S_2 \Longrightarrow bA \Longrightarrow bbAAA \Longrightarrow bbaS_2A \Longrightarrow bbaA \Longrightarrow bbabAA \Longrightarrow bbabbAAA \Longrightarrow bbabbaaa$

Can you find a nonambiguous CFG for $L_{\sharp a=\sharp b}$?

Definition of Pushdown Automata (PDA)

Definition: A (Nondeterministic) Pushdown Automaton

A (nondeterministic) **pushdown automaton** (**PDA**) is a 7–tuple $(Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$, where

- Q is the (finite) set of **states**,
- Σ is the (finite) **input alphabet**,
- Γ is the (finite) **stack alphabet**,
- $q_0 \in Q$ is the (input) **start state**,
- $Z_0 \in \Gamma$ is the **stack start symbol**,
- $F \subseteq Q$ is the **set of accepting** or (**final**) **states**, and
- $\delta \colon Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times \Gamma^*)$ is the **transition function** (or better said **multi–function** or **relation**) (where $\mathcal{P}(X)$ denotes the **power set** of set of X).

Transitions of a PDA, I

How a PDA computes

Let us denote $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. Writing the transition function of a PDA M as:

$$\delta(p, \sigma, Z) = \{(q_1, \gamma_1), (q_2, \gamma_2), \ldots, (q_m, \gamma_m)\},$$

where $p, q_1, q_2, \ldots q_m \in Q$ are states, $\sigma \in \Sigma_\varepsilon$ is an input symbol, $Z \in \Gamma_\varepsilon$ is a stack symbol, and $\gamma_1, \gamma_2, \ldots, \gamma_m \in \Gamma^*$ are stack strings, means that:

- when M is on state p and it is reading the **leftmost** input symbol σ (notice that, since $\sigma \in \Sigma_\varepsilon$, it is possible that $\sigma = \varepsilon$), while the **top of the stack** is occupied by stack symbol Z (notice that, since $Z \in \Sigma_\varepsilon$, one may consider that $Z = \varepsilon$),
- then M transitions to states $q_1, q_2, \ldots q_m$ by removing σ from the input string and replacing Z with γ_i, for each $i = 1, 2, \ldots, m$ (respectively).

Transitions of a PDA, II

How a PDA computes (cont.)

For $p, q \in Q, \sigma \in \Sigma_\varepsilon, Z \in \Gamma_\varepsilon, \gamma \in \Gamma^*$,

$$p \xrightarrow{\sigma \mid Z,\gamma} q \text{ means } \delta(p, \sigma, Z) \ni (p, \gamma),$$

i.e., when a PDA M is on state p, reads the rightmost input symbol σ, while symbol Z is on top of the stack, then M transitions to state q, removes σ from the input and replaces Z with γ on the top of the stack. Particular cases:

- $\sigma \mid \varepsilon, \gamma$ means reading input or stack symbol σ, adding string γ on the top of the stack.
- $\sigma \mid Z, \varepsilon$ means reading input symbol σ, removing symbol Z from the top of the stack.
- $\sigma \mid \varepsilon, \varepsilon$ means reading input symbol σ, without making any change on the stack.
- $\varepsilon \mid Z, \gamma$ means without reading any input symbol, replacing Z with γ on the top of the stack.
- $\varepsilon \mid \varepsilon, \gamma$ means without reading any input or stack symbol, adding string γ on the top of the stack.
- $\varepsilon \mid Z, \varepsilon$ means without reading any input symbol, removing symbol Z from the top of the stack.

Configurations and Moves

Definition: **Configurations and Moves**

Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

- A **configuration** (or **instantaneous discription**) of M is a triplet (p, w, γ) such that, when M is on state $p \in Q$, the part of the input string that is about to be read is string $w \in \Sigma^*$ and, at the same instance, the contents of the (whole) stack are given by string $\gamma \in \Gamma^*$.
- Two configurations $(p, \sigma w, Z\alpha)$ and $(q, w, \gamma\alpha)$ (for $p, q \in Q, \sigma \in \Sigma_\varepsilon, w \in \Sigma^*, Z \in \Gamma_\varepsilon, \alpha, \gamma \in \Gamma^*$) are said to form a **move in one step**, written as

$$(p, \sigma w, Z\alpha) \vdash (q, w, \gamma\alpha),$$

whenever $\delta(p, \sigma, Z) \ni (q, \gamma)$, i.e., whenever M is on state p, reads the rightmost input symbol σ and at the stack's top is symbol Z, then M transitions to state q replacing Z with (string) γ on the stack.

Chains of Moves

Definition: Chains of Moves

Let $C_0, C_1, \ldots, C_n$ be a sequence of configurations such that every successive two configurations form a move in one step. Then these configurations are said to form (a **chain of**) **moves in n steps**, $C_0 \vdash C_1 \vdash \cdots \vdash C_n$, which is symbolically written as

$$C_0 \vdash^n C_n.$$

As previously, the notation

$$C_0 \vdash^* C_n.$$

will refer to a move of 1 or more steps.

Languages Accepted by PDAs

Definition: **Languages accepted by PDAs**

Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ be a PDA.

- The **language accepted by empty stack** by M is the set

  $$L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, \varepsilon), \text{ for } q_f \in F\}.$$

- The **language accepted by final state** by M is the set

  $$L_{FS}(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, \gamma), \text{ for } q_f \in F \text{ and } \gamma \in \Gamma^*\}.$$

Theorem 1

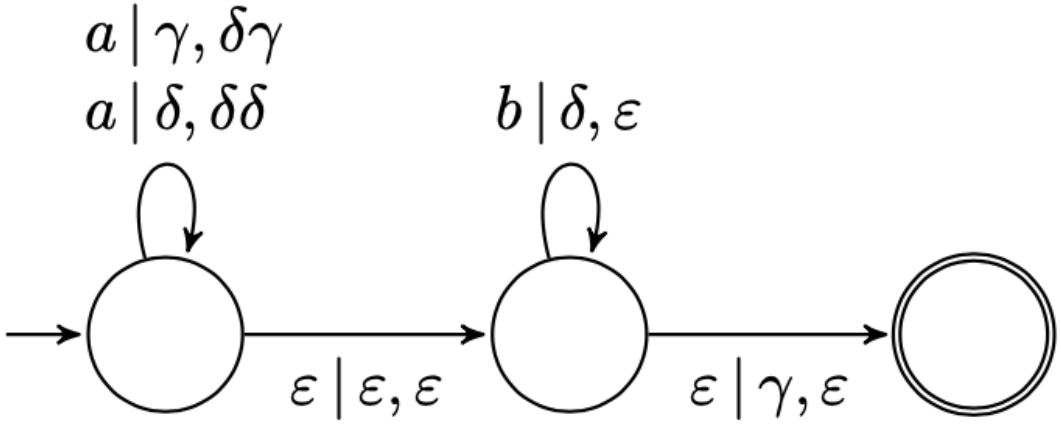
$L = L_{FS}(M_1)$, for some PDA M_1, if and only if there exists PDA M such that $L = L(M)$.

Theorem 1

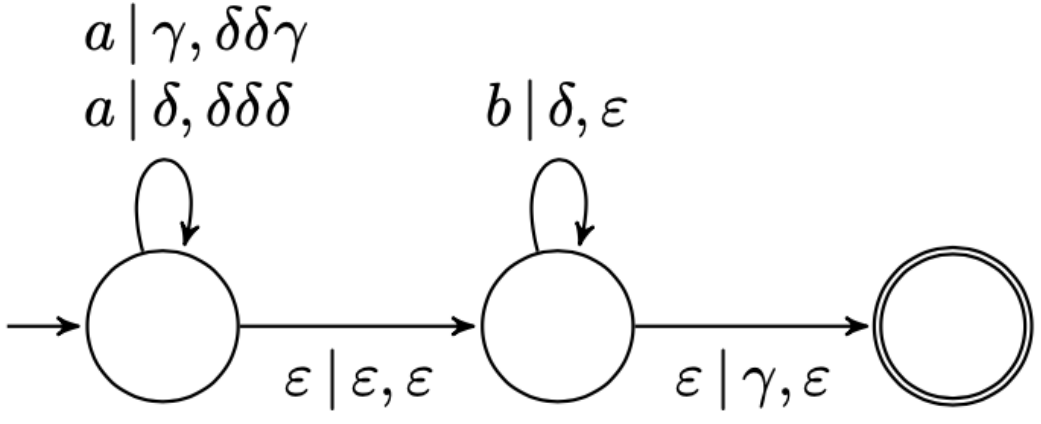
L is a context–free language (CFL) if and only if $L = L(M)$, for some PDA M.

Examples of Pushdown Automata, I

Example 1: $L = \{a^i b^i \mid i \geq 0\}$

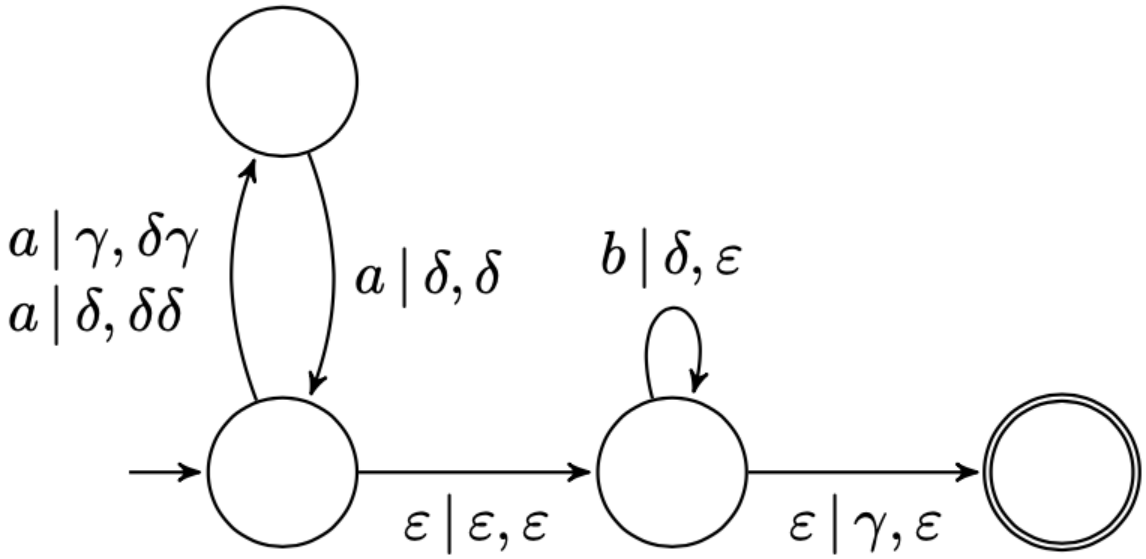


Example 2: $L = \{a^i b^{2i} \mid i \geq 0\}$

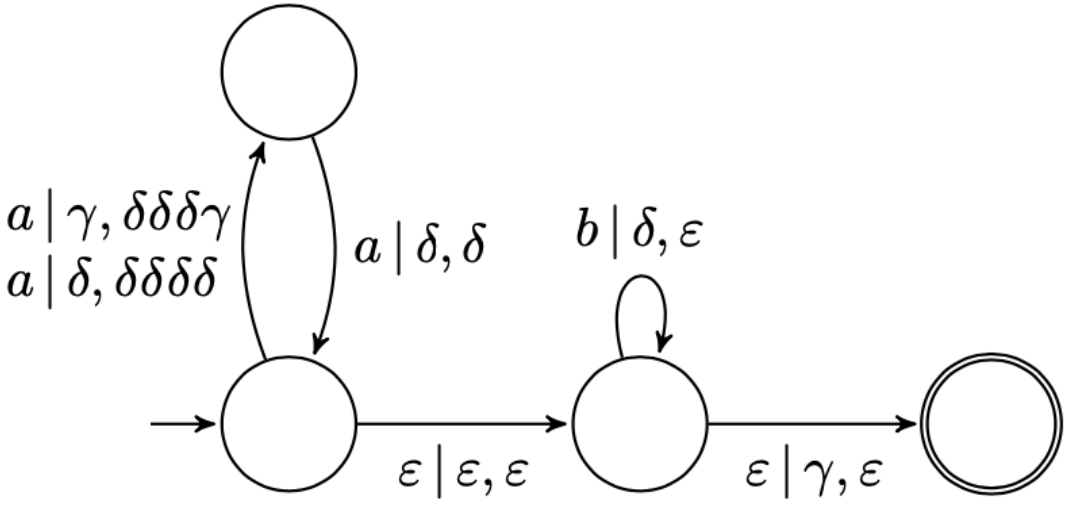


Examples of Pushdown Automata, II

Example 3: $L = \{a^{2i}b^i \mid i \geq 0\}$

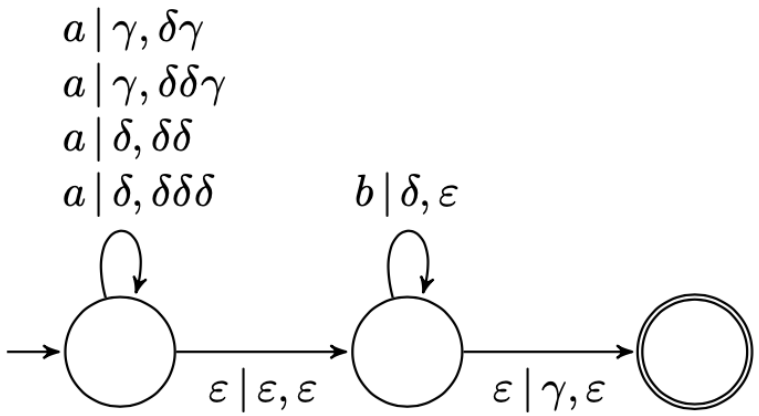


Example 4: $L = \{a^{2i}b^{3i} \mid i \geq 1\}$

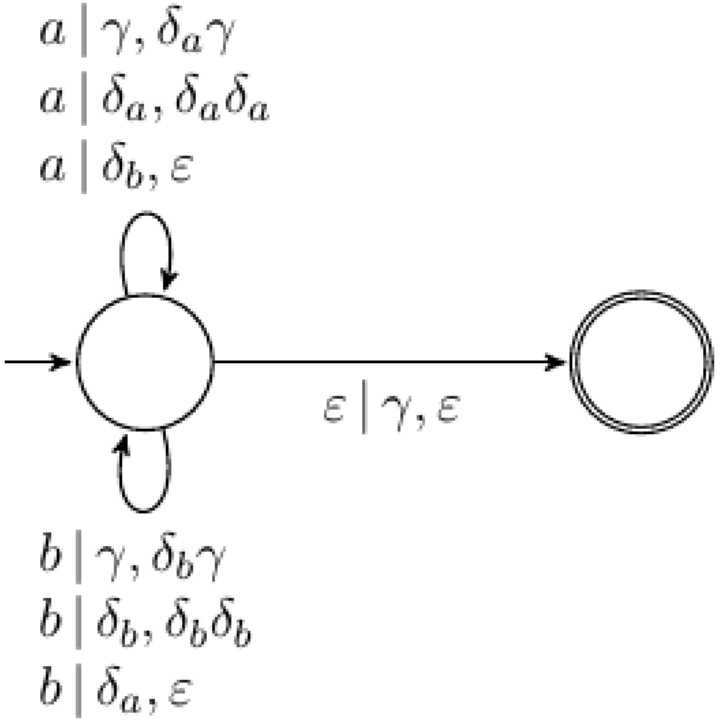


Examples of Pushdown Automata, III

Example 5: $L = \{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$



Example 6: $L = \{w \in (a+b)^\star \mid n_a(w) = n_b(w) \geq 0\}$



Regular Grammars, I

Definition: **Right–, Left–, Regular and Linear Grammars**

Let $G = (V, \Sigma, S, P)$ a CFG.

- G is called **right–linear** if all productions are of one of the two forms

$$A \to xB,$$

$$A \to x,$$

  where $A, B \in V$ and $x \in \Sigma^*$.
- G is called **left–linear** if all productions are of one of the two forms

$$A \to Bx,$$

$$A \to x.$$

- G is called **regular grammar** if it is either right–linear or left–linear.
- G is called **linear grammar** if at most one variable can ocur on the right side of any production, independently of its position.

Regular Grammars, II

Example

Consider the following CFGs:

$G_1 = (\{S\}, \{a, b\}, S, \{S \to abS \mid a\})$,
$G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, \{S \to S_1ab, S_1 \to S_1ab \mid S_2, S_2 \to a\})$,
$G_3 = (\{S, A, B\}, \{a, b\}, S, \{S \to A, A \to aB \mid \varepsilon, B \to Ab\})$.

Then G_1 is right–linear, G_2 is left–linear, and G_3 is linear. Notice that G_1 generates the regular language $(ab)^*a$, G_2 generates the regular language $aab(ab)^*$, and G_3 generates the nonregular language $\{a^n b^n \mid n \geq 0\}$.

Theorem

If G is a right–linear (or left–linear) CFG, then $L(G)$ is a regular language.

Theorem

If L is a regular language over Σ, then there exists a right–linear (or left–linear) CFG $G = (V, \Sigma, S, P)$ such that $L = L(G)$.

Regular Grammars, III

Correspondence of Regular Grammar to Finite Automaton

Assume that G is a right–linear CFG with variables $V = \{V_0, V_1, \ldots\}, S = V_0$, and productions of the form $V_0 \to v_i V_i, V_i \to v_2 V_j, \ldots$ or $V_n \to v_k$.

- Each variable in V is considered to be a state with the start state being the start variable.
- Each production $V_i \to \sigma V_j$ (where $\sigma \in \Sigma$) creates the transition $\delta(V_i, \sigma) = V_j$.
- Each production $V_i \to \sigma_1 \sigma_2 \cdots \sigma_m V_j$, where $\sigma_1 \sigma_2 \cdots \sigma_m \in \Sigma^*, m \geq 2$, creates the additional states $P_1, P_2, \ldots, P_{m-1}$ (other than states V_i and V_j) and the transitions $\delta(V_i, \sigma_1) = P_1, \delta(P_1, \sigma_2) = P_2, \ldots, \delta(P_{m-1}, \sigma_m) = V_j$.
- Each production $V_i \to \sigma$ (where $\sigma \in \Sigma$) creates a final state F and the transition $\delta(V_i, \sigma) = F$.
- Each production $V_i \to \tau_1 \tau_2 \cdots \tau_m$, where $\tau_1 \tau_2 \cdots \tau_m \in \Sigma^*, |\tau_1 \tau_2 \cdots \tau_m| > 0$ (necessarily $m > 1$), creates the additional states $Q_1, Q_2, \ldots, Q_{m-1}$ (other than the state V_i), a final state F, and the transitions $\delta(V_i, \tau_1) = Q_1, \delta(Q_1, \tau_2) = Q_2, \ldots, \delta(Q_{m-1}, \tau_m) = F$.
- Each production $V_i \to \varepsilon$ creates a final state F and a transition $\delta(V_i, \varepsilon) = F$.

Regular Grammars, IV

Example

Construct a finite automaton that accepts the regular language generated by the right–linear CFG $(\{S, V\}, \{a, b\}, S, \{S \to aV, V \to abS \mid b\})$.

This is the finite automaton with states $Q = \{S, V, P, F\}$ and transitions converted by productions as follows:

Productions	States	Transitions
$S \to aV$	S, V	$\delta(S, a) = V$
$V \to abS$	S, P, V	$\delta(V, a) = P$ $\delta(P, b) = S$
$V \to b$	V, F	$\delta(V, b) = F$

Apparently, the regular language is $(aab)^*ab$.

Correspondence of Finite Automaton to Regular Grammar

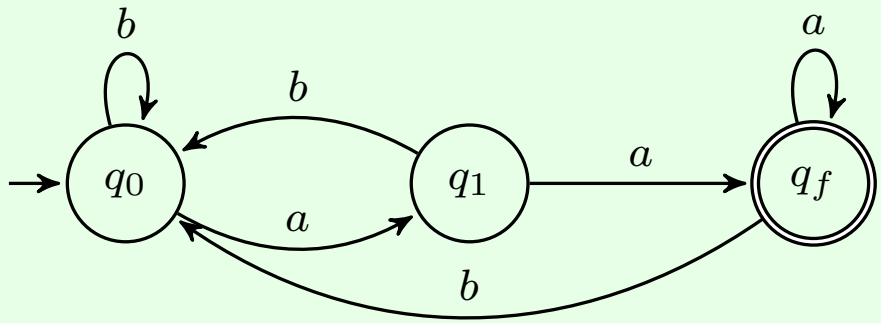
Assume that M is finite automaton over Σ with set of states Q, start state q_0 and (set of) final states F, and transitions given by function δ.

- Each state in Q other than final states is considered to be a variable with the start variable being the start state.
- Each transition $\delta(p, \sigma) = q$ creates the production $p \to \sigma q$.
- Each final state $q_f \in F$ creates the production $q_f \to \varepsilon$.

Regular Grammars, V

Example

Find the right–linear CFG corresponding to the language accepted by the following finite automaton:



The corresponding CFG has variables (q_0, q_1, q_f) (with q_0 the start variable) and productions converted by transitions as follows:

Transitions	Variables	Productions
$\delta(q_0, a) = q_1$	q_0, q_1	$q_0 \to aq_1$
$\delta(q_0, b) = q_0$	q_0	$q_0 \to bq_0$
$\delta(q_1, a) = q_f$	q_1, q_f	$q_1 \to aq_f$
$\delta(q_1, b) = q_0$	q_1, q_0	$q_1 \to bq_0$
$\delta(q_f, a) = q_f$	q_f	$q_f \to aq_f$
$\delta(q_f, b) = q_0$	q_f, q_0	$q_f \to bq_0$
Final state	q_f	$q_f \to \varepsilon$

Apparently, the regular language is $(a + b)^* aa$, i.e., strings ending in aa.

Simplification of Context–Free Grammars: Removing ε–Productions, I

Definition: ε–Productions and Nullable Variables

In a CFG, any production of the form

$$A \to \varepsilon$$

is called ε–**production** and any variable A, for which the derivation

$$A \Longrightarrow^* \varepsilon$$

is possible, is called **nullable**.

Theorem

Let $G = (V, \Sigma, S, P)$ a CFG containing ε–productions. Removing in P all ε–productions and adding new productions obtained by substituting ε for an ε–production wherever else the latter occurs in P, a CFG $\hat{G}$ is created such that $L(\hat{G}) = L(G) \setminus \{\varepsilon\}$.

Modifying ε–Productions

- If B is ε–production and $A \to BB$, then the latter production is modified as $A \to B$.
- If B, C are ε–productions and $A \to aBbCa$, then the latter production is modified as $A \to abCa \mid aBba$.

Removing ε–Productions, II

Example

Let G the CFG with ε–productions at the left column of the following table. The productions of the corresponding ε–productions–free CFG $\hat{G}$ are converted at the right column of the table:

Productions of G	Productions of $\hat{G}$
$S \to XY$	$S \to XY$
$X \to Zb$	$X \to Zb \mid b$
$Y \to bW$	$Y \to bW \mid b$
$Z \to AB$	$Z \to AB \mid A \mid B$
$W \to Z$	$W \to Z$
$A \to aA$	$A \to aA \mid a$
$A \to bA$	$A \to bA \mid b$
$A \to \varepsilon$	—
$B \to Ba$	$B \to Ba \mid a$
$B \to Bb$	$B \to Bb \mid b$
$B \to \varepsilon$	—

ε–productions: $A \to \varepsilon$ and $B \to \varepsilon$

Nullable variables $= \{A, B, Z, W\}$

Simplification of Context–Free Grammars: Removing Unit–Productions, I

Definition: Unit–Productions

Any production of a CFG $G = (V, \Sigma, S, P)$ of the form

$$A \to B,$$

where $A, B \in V$, is called a **unit–production**. Moreover, for any variable $C \in V$, we define the **unit set** of C as

$$\text{Unit}(C) = \{D \in V \mid C \implies^* D \text{ only through unit–productions}\}.$$

Notice that, by default, $C \in \text{Unit}(C)$.

Theorem

Let $G = (V, \Sigma, S, P)$ a CFG without ε–productions. Removing in P all unit–productions and adding new productions obtained from the unit set of the latter, a CFG $\hat{G}$ is created such that $\hat{G}$ does not have any ε–productions nor unit–productions and $L(\hat{G}) = L(G) \setminus \{\varepsilon\}$.

Modifying Unait–Productions

For each $B \in \text{Unit}(A), B \neq A$, such that $B \to x$, for some $x \in \Sigma^*$ (non–unit–production of G), we are adding in $\hat{G}$ the production $A \to x$.

Removing Unit–Productions, II

Example

Let the CFG G, which includes the productions:

$$S \to A \mid Aa, A \to B, B \to C \mid b, C \to D \mid ab, D \to b.$$

Clearly,

$$\text{Unit}(S) = \{S, A, B, C, D\}.$$

The productions of the corresponding unit–productions–free CFG $\hat{G}$ are converted in the table:

Productions of G	Productions of $\hat{G}$
$S \to Aa$	$S \to Aa$
$B \to b$	$B \to b$
$C \to ab$	$C \to ab$
$D \to b$	$D \to b$
$\text{Unit}(S) = \{S, A, B, C, D\}$	$S \to b \mid ab$
$\text{Unit}(A) = \{A, B, C, D\}$	$A \to b \mid ab$
$\text{Unit}(B) = \{B, C, D\}$	$B \to b \mid ab$
$\text{Unit}(C) = \{C, D\}$	$C \to b$

Chomsky Normal Form, I

Definition: Chomsky Normal Form

A CFG $G = (V, \Sigma, S, P)$ is in **Chomsky normal form** if all productions are of the form

$$A \to BC,$$

or

$$A \to a,$$

where $A, B, C \in V$ and $a \in \Sigma$. (Notice that $a \neq \varepsilon$.)

Theorem

Let $G = (V, \Sigma, S, P)$ a ε–productions–free CFG. Modifying G's productions, an equivalent CFG $\hat{G}$ in Chomsky normal form can be created.

Chomsky Normal Form Conversion

- G–productions of the form $A \to BC$ or $A \to a$ are preserved in $\hat{G}$.
- G–productions of the form $A \to B_1 B_2 \cdots B_m$, for $m \geq 2, B_1, \ldots, B_m \in V \cup \Sigma$, preserve the (old) variables A, B_1, B_{m-1}, B_m and create the (new) $\hat{G}$–variables $D_1, \ldots, D_{m-1}$ into the (new) $\hat{G}$–productions $A \to B_1 D_1, D_1 \to B_2 D_2, \ldots,$ $D_{m-3} \to B_{m-2} D_{m-2}, D_{m-2} \to B_{m-1} B_m$.

Chomsky Normal Form, II

Example

Let the CFG G, which includes the productions:

$$S \to bA \mid aB, A \to bAA \mid aS \mid a, B \to aBB \mid bS \mid b.$$

The productions of the corresponding Chomsky normal form CFG $\hat{G}$ are converted in the table:

Productions of G	Productions of $\hat{G}$
$S \to bA \mid aB$	$S \to C_bA \mid C_aB$ $C_a \to a, C_b \to b$
$A \to bAA$	$A \to C_bD_1$ $D_1 \to AA$
$B \to aBB$	$B \to C_aD_2$ $D_2 \to BB$
$D \to b$	$D \to b$
$A \to aS$	$A \to C_aS$
$B \to bS$	$B \to C_bS$
$A \to a$	$A \to a$
$B \to b$	$B \to b$

Pushdown Automata from Context–Free Grammars, I

Theorem

For every CFG G, there is a PDA M such that $L(M) = L(G)$.

Construction of a PDA from a CFG

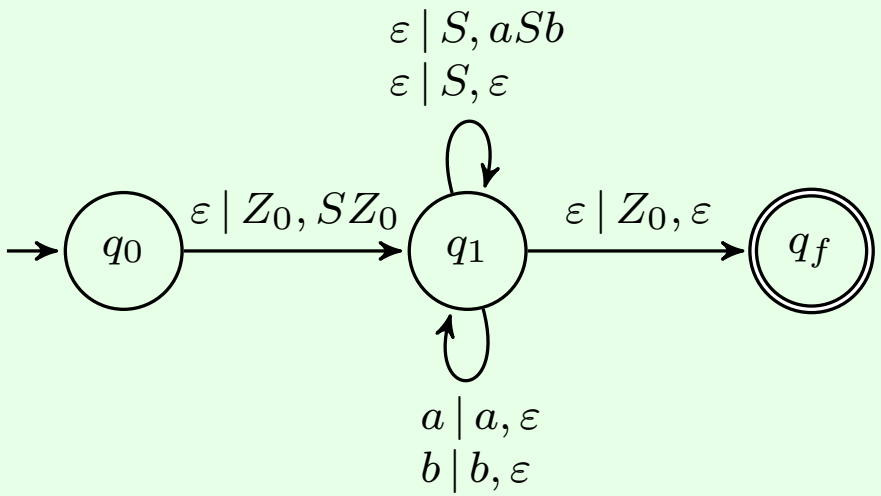
Let $L = L(G)$, where $G = (V, \Sigma, S, P)$ is a CFG. We are constructing a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$, which accepts language L, as follows:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_f\}, \text{ three (arbitrary) states,} \\
q_0 &= \text{the start state,} \\
F &= \{q_f\}, \\
\Gamma &= V \cup \Sigma \cup \{Z_0\}, \\
Z_0 &= \text{the start stack symbol,} \\
\delta(q_0, \varepsilon, Z_0) &= (q_1, SZ_0), \\
\delta(q_1, \varepsilon, A) &\ni (q_1, w), \text{ whenever } A \to w \text{ is a } G\text{–production,} \\
\delta(q_1, a, a) &\ni (q_1, \varepsilon), \text{ whenever } a \in \Sigma, \\
\delta(q_1, \varepsilon, Z_0) &= (q_f, \varepsilon).
\end{aligned}
$$

Pushdown Automata from Context–Free Grammars, II

Example

The PDA corresponding to the CFG $G = (\{S\}, \{a, b\}, S, \{S \to aSb \mid \varepsilon\})$, i.e., generating the language $L = \{a^i b^i \mid i \geq 0 \text{ integer}\}$, with three states $\{q_0, q_1, q_f\}$, input alphabet $\{a, b\}$, and stack alphabet $\{S, a, b, Z_0\}$:



Context–Free Grammars from Pushdown Automata, I

Theorem

For every PDA M, there is a CFG G such that $L(G) = L(M)$.

Construction of a CFG from a PDA

Let the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ such that, for any $p, q \in Q, \sigma \in \Sigma$, $\gamma, \delta \in \Gamma$,

- $F = \{q_f\}$ (and M terminates only with empty stack),
- every transition is of one the two forms

$$\delta(q, \sigma, \gamma) \ni (p, \varepsilon) \text{ or } \delta(p, \sigma, \gamma) \ni (p, \delta\gamma),$$

  which means that every transition either decreases or increases the stack content by a single symbol.

We are constructing a CFG $G = (V, \Sigma, S, P)$, which generates the language $L(G)$, as follows:

$$V = \{[q\gamma p] \,|\, p, q \in Q, \gamma \in \Gamma\},$$
$$S = [q_0 Z_0 q_f],$$
$$P = \{[q\gamma p] \to \sigma, \text{ whenever } \delta(q, \sigma, \gamma) \ni (p, \varepsilon)\}.$$

Context–Free Grammars from Pushdown Automata, II

Example

Let the PDA $M = (\{q_0, q_1, q_f\}, \{a, b\}, \{X, Z_0\}, q_0, Z_0, \{q_f\}, \delta)$ possess the following transitions:

$$\begin{aligned}
\delta(q_0, a, Z_0) &= (q_0, XZ_0),\\
\delta(q_0, a, X) &= (q_0, XX),\\
\delta(q_0, b, X) &= (q_1, \varepsilon),\\
\delta(q_1, b, X) &= (q_1, \varepsilon),\\
\delta(q_1, \varepsilon, X) &= (q_1, \varepsilon),\\
\delta(q_1, \varepsilon, Z_0) &= (q_f, \varepsilon).
\end{aligned}$$

We are constructing the corresponding CFG $G = (V, \Sigma, S, P)$ with the following variables:

$$\begin{aligned}
V = \{&[q_0Xq_0], [q_0Xq_1], [q_0Xq_f], [q_0Z_0q_0], [q_0Z_0q_1], [q_0Z_0q_f] \equiv S,\\
&[q_1Xq_0], [q_1Xq_1], [q_1Xq_f], [q_1Z_0q_0], [q_1Z_0q_1], [q_1Z_0q_f],\\
&[q_fXq_0], [q_fXq_1], [q_fXq_f], [q_fZ_0q_0], [q_fZ_0q_1], [q_fZ_0q_f].
\end{aligned}$$

Context–Free Grammars from Pushdown Automata, III

Example (cont.)

Moreover, the productions of G are created as follows:

Transversal Transitions of M	Productions of G
$\delta(q_0, b, X) = (q_1, \varepsilon)$	$[q_0 X q_1] \to b$
$\delta(q_1, b, X) = (q_1, \varepsilon)$	$[q_1 X q_1] \to b$
$\delta(q_1, \varepsilon, X) = (q_1, \varepsilon)$	$[q_1 X q_1] \to \varepsilon$
$\delta(q_1, \varepsilon, Z_0) = (q_f, \varepsilon)$	$[q_1 Z_0 q_f] \to \varepsilon$

Productions of G created by transition $\delta(q_0, a, Z_0) = (q_0, XZ_0)$

$$[q_0 Z_0 q_0] \to a[q_0 X q_0][q_0 Z_0 q_0] \,|\, a[q_0 X q_1][q_1 Z_0 q_0] \,|\, a[q_0 X q_f][q_f Z_0 q_0]$$
$$[q_0 Z_0 q_1] \to a[q_0 X q_0][q_0 Z_0 q_1] \,|\, a[q_0 X q_1][q_1 Z_0 q_1] \,|\, a[q_0 X q_f][q_f Z_0 q_1]$$
$$[q_0 Z_0 q_f] \to a[q_0 X q_0][q_0 Z_0 q_f] \,|\, a[q_0 X q_1][q_1 Z_0 q_f] \,|\, a[q_0 X q_f][q_f Z_0 q_f]$$

Productions of G created by transition $\delta(q_0, a, X) = (q_0, XX)$

$$[q_0 Z_0 q_0] \to a[q_0 X q_0][q_0 X q_0] \,|\, a[q_0 X q_1][q_1 X q_0] \,|\, a[q_0 X q_f][q_f X q_0]$$
$$[q_0 Z_0 q_1] \to a[q_0 X q_0][q_0 X q_1] \,|\, a[q_0 X q_1][q_1 X q_1] \,|\, a[q_0 X q_f][q_f X q_1]$$
$$[q_0 Z_0 q_f] \to a[q_0 X q_0][q_0 X q_f] \,|\, a[q_0 X q_1][q_1 X q_f] \,|\, a[q_0 X q_f][q_f X q_f]$$

Notice that the only variables which are used in the previous productions are:

$$V = \{[q_0 Z_0 q_f] \equiv S, [q_0 X q_0], [q_0 X q_1], [q_0 X q_f], [q_0 Z_0 q_0], [q_0 Z_0 q_1], [q_1 Z_0 q_f], [q_1 X q_1]\},$$

while all the other variables may be dropped from the corresponding productions.

The Pumping Lemma for Context–Free Languages

Theorem: **The Pumping Lemma for Context–Free Languages**

If L is a context–free language (**CFL**) over alphabet Σ, then there is a positive integer n so that, for every $x \in L$ with $|x| \geq n$, x can be written as $x = uvwxy$, for some strings $u, v, w, x, y \in \Sigma^*$ satisfying:

- $|vwx| \leq n$,
- $|vx| \geq 1$, i.e., $v \neq \varepsilon$ or $x \neq \varepsilon$,
- for every integer $m \geq 0, uv^m wx^m y \in L$.

Corollary

Let L be a CFL and n the positive integer of the Pumping Lemma. Then:

- $L \neq \varnothing$ if and only if there exists a $w \in L$ with $|w| < n$, and
- L is infinite if and only if there exists a $z \in L$ such that $n \leq |z| < 2n$.

Examples of Pumping Lemma for CFL, I

Example 1

Show that the language $L = \{a^i b^i c^i \mid i \geq 0\}$ is not context–free.

Assume that L is context–free and denote by n the natural number obtained by using the PL. Let $z = a^n b^n c^n$. Then $|z| = 3n > n$. Thus, according to PL, we may write $z = uvwxy$, where, setting $p = n_a(vx), q = n_b(vx), r = n_c(vx)$, we should have $1 \leq p + q + r \leq n$. Apparently, one of the following five cases should hold:

Case (1): vx contains only a's ($p \geq 1, q = 0, r = 0$);

Case (2): vx contains only b's ($p = 0, q \geq 1, r = 0$);

Case (3): vx contains only c's ($p = 0, q = 0, r \geq 1$);

Case (4): vx contains both a's and b's ($p \geq 1, q \geq 1, r = 0$);

Case (5): vx contains both b's and c's ($p = 0, q \geq 1, r \geq 1$).

Let $z_0 = uwy \in L$ and $z_2 = uv^2wx^2y$ (both generated from the third implication of the PL for $m = 0$ and $m = 2$, respectively).

In case (1), $n_a(z_0) = n_a(z) - p = n - p, n_b(z_0) = n_b(z) = n$ and $n_c(z_0) = n_c(z) = n$. But then (since $p > 0$), $n_a(z_0) = n - p \neq n = n_b(z_0) = n_c(z_0)$, which means that $z_0 \notin L$, a contradiction. Similar contradictions are derived in cases (2) and (3).

In case (4), since vx contains a and b (at least one from both), $vx = a^p b^q$, for some integers $p \geq 1, q \geq 1$. Hence, $z_2 = u(a^p b^q)(a^p b^q)wx^2y \in L$, which is a contradiction, because strings of L cannot have b's followed by a's. Similar contradiction is derived in case (5) too.

Examples of Pumping Lemma for CFL, II

Example 1

Show that the language $L = \{a^i b^j \mid i = j^2\}$ is not context–free.

Assume that L is context–free and denote by n the natural number obtained by using the PL. Let $z = a^{n^2} b^n$. Then $|z| = n^2 + n > n$. Thus, according to PL, we may write $z = uvwxy$, where, setting $p = n_a(vx), q = n_b(vx)$, we should have $1 \leq p + q \leq n$. Apparently, one of the following three cases should hold:

Case (1): vx contains only a's ($p \geq 1, q = 0$);

Case (2): vx contains only b's ($p = 0, q \geq 1$);

Case (3): vx contains both a's and b's ($p \geq 1, q \geq 1$).

Let $z_0 = uwy \in L$ (generated from the third implication of the PL for $m = 0$).

In case (1), $n_a(z_0) = n_a(z) - p = n^2 - p$ and $n_b(z_0) = n_b(z) = n$. But then $(n_b(z_0))^2 = n^2 > n^2 - p = n_a(z_0)$, i.e., $(n_b(z_0))^2 > n_a(z_0)$, which means that $z_0 \notin L$, a contradiction.

In case (2), $n_a(z_0) = n_a(z) = n^2$ and $n_b(z_0) = n_b(z) - q = n - q$. But then $(n_b(z_0))^2 = (n-q)^2 < (n-1)^2 = n^2 - 2n + 1 < n^2 = n_a(z_0)$, i.e., $(n_b(z_0))^2 < n_a(z_0)$, which means that $z_0 \notin L$, a contradiction.

In case (3), $n_a(z_0) = n_a(z) - p = n^2 - p$ and $n_b(z_0) = n_b(z) - q = n - q$. But then $(n_b(z_0))^2 = (n - q)^2 < (n - 1)^2 = n^2 - 2n + 1 < n^2 - p = n_a(z_0)$, i.e., $(n_b(z_0))^2 < n_a(z_0)$, which means that $z_0 \notin L$, a contradiction.

Ogden's Lemma

Theorem: **Ogden's Lemma**

If L is a context–free language (**CFL**) over alphabet Σ, then there is a positive integer n so that, for every $x \in L$ with $|x| \geq n$, if we mark at least n symbols of x (i.e., if we choose n or more "distinguished" positions in the string x), x can be written as $x = uvwxy$, for some strings $u, v, w, x, y \in \Sigma^*$ satisfying:

- string vwx contains at most n marked symbols,
- string vx contains at least one marked symbol, i.e., there exists (at least) one marked symbol either in string v or in string x, and
- for every integer $m \geq 0, uv^m wx^m y \in L$.

Example of Ogden's Lemma

Example

Show that the language $L = \{a^i b^i c^j \mid i \neq j\}$ is not context–free.

Using the PL, we might start with the string $z = a^n b^n c^{n+n!}$, so that if vx contains both a's and b's, say $p = n_a(vx), q = n_b(vx)$, where $\max(p, q) \leq n$, this would get a contradiction when we "pump" v and x for $m = n!/\max(p, q)$ times. However, if vx contains only c's, this approach is not going to work. This is why we need to use Ogden's lemma now and mark all symbols a in z to force the case that vx contains both a's and b's. The details are as follows:

Assume that L is context–free and denote by n the natural number obtained by using Ogden's Lemma (OL). Let $z = a^n b^n c^{n+n!}$ with all symbols a marked (i.e., considered "distinguished"). Since $|z| = 3n + n! > n$, z can be decomposed as $z = uvwxy$ satisfying the three implications of OL.

First, we observe that neither v nor x may contain more than one type of symbols. For instance, if vx contains both symbols a and b, then the string $z_2 = uv^2wx^2y \in L$ would have a b occurring before an a and, hence, it would not belong to L, which is a contradiction. Similarly, it is not possible that vx contains either a and b or a and c. Therefore, either b (case 1) or c (case 2) is not in vx.

In case (1), $n_a(vx) > 0$, while $n_b(vx) = 0$, implies that $z_2 = uv^2wx^2y \in L$ would have more a's than b's, which is a contradiction.

In case (2), if $n_a(vx) = p$, where $1 \leq p \leq n$, while $n_c(vx) = 0$, then, for $m = 1 + n!/p$, the string $z_m = uv^m wx^m y \in L$ would have as many a's as c's and, hence, it cannot be in L, which is a contradiction.

A Pumping Lemma for Linear Languages

Definition

A CFL L is said to be **linear language** if there exists a linear CFG G such that $L = L(G)$.

Theorem: The Pumping Lemma for Linear Languages

If L is an infinite linear language over alphabet Σ, then there is a positive integer n so that, for every $x \in L$ with $|x| \geq n$, x can be written as $x = uvwxy$, for some strings $u, v, w, x, y \in \Sigma^*$ satisfying:

- $|uvxy| \leq n$,
- $|vx| \geq 1$, i.e., $v \neq \varepsilon$ or $x \neq \varepsilon$,
- for every integer $m \geq 0, uv^m wx^m y \in L$.

Example of the Pumping Lemma for Linear Languages

Example

Show that the language $L = \{w \in (a+b)^n \,|\, n_a(w) = n_b(w)\}$ is not linear.

Assume that L is linear and denote by n the natural number in the PL for linear languages. Let $z = a^n b^{2n} a^n$ with $|z| = 4n > n$.

Since now we should have $|uvxy| \leq n$, strings u, v, x, y must all consist entirely of a's. In particular say $p = n_a(vx)$, where $1 \leq p < n$.

However, if we "pump" z, for any $m \neq 1$, we obtain the string $z_m = a^{n+(m-1)p} b^{2n} a^n \in L$ with $n_a(z_m) = 2n + (m-1)p \neq 2n = n_b(z_m)$, which is a contradiction.

Closure Properties of Context–Free Languages

Theorem 1

The class of CFLs is closed under union, concatenation, and Kline star.

Theorem 2

The class of CFLs is not closed under intersection and complementation.

Theorem 3

The intersection of a CFL with a regular language is a CFL.

Example of How to Use Closure Properties of CFLs

Example

Show that the language $L = \{a^i b^i \,|\, i \geq 0, i \neq 100\}$ is context–free.

It is possible to construct a PDA or a CFG for the above language, but the process would be tedious. We can get a much neater argument using the last Theorem in the previous slide.

Let us consider the language:

$$L_1 = \{a^{100} b^{100}\},$$

which is obviously regular (being finite). Furthermore, we have

$$L = \{a^i b^i \mid i \geq 0\} \cap \overline{L_1}.$$

Therefore, by the closure of regular languages under complementation and the closure of CFL under regular intersection, L is a CFL.

III. COMPUTABILITY THEORY

Turing Machines: Operation

Turing Machine Operation: Tape, Head, States

A **Turing Machine** operates in a finite number of **states** using the following two components:

- a **tape**, which is an **infinite** string of **squares**, on each of which **tape symbols** are engraved, and
- a **tape head**, which at each state is pointing to a tape symbol that (the tape head) can control by:
  - reading or writing (on it) and then
  - move one symbol **left** (**L**) or **right** (**R**) on the tape.
- **Initially**, the tape contains only an **input string** and it is **blank** everywhere else (where "␣" is the tape symbol for blank).
- Finally, the machine either reaches a **halt state**, in which the outputs **accept** and **reject** are obtained, or it never halts falling into an (infinite) **loop**.

Turing Machines: Formal Definition

Formal Definition of a Turing Machine

A **Turing machine** (**TM**) M is a 7–tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets such that:

- Q is the set of **states**,
- Σ is the **input alphabet**,
- $\Gamma \cup \{\sqcup\}$ is the **tape alphabet**, where $\Sigma \subseteq \Gamma$,
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the **transition function**,
- $q_0 \in Q$ is the **start state**,
- $q_{\text{accept}} \in Q$ is the **accept state**, and
- $q_{\text{reject}} \in Q$ is the **reject state**, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Notice that, when $\delta(q_i, b) = (q_j, c, D)$, M goes from state q_i, in which the tape head points to (tape) symbol b, to a unique state q_j (and, thus, M is a **determinstic** TM), replacing b by c on the tape and moving one cell in the direction $D = \text{L}$ or R.

Turing Machines: Computation

Computation of a Turing Machine

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a TM.

- Initially, M receives an input $w = w_1 w_2 \dots w_n \in \Sigma^*$ on the **leftmost** n **squares** of the tape and the rest of the tape is filled with the blank symbol.
- The head starts operating according to the rule assigned to the start state.
- Once M has started, the computation proceeds according to the rules described by the transition function.
- If M ever tries to move its head to left off the left–hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L.
- The computation continues until it enters either the accept or reject states, at which point it halts. Otherwise, the computation goes for ever in a loop.

Turing Machines: Configurations and Yields, I

Definition: **Configurations and Yields (or Moves)**

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a TM.

- When, at time/step k (of the computation implemented by M), M is at state $q \in Q$, while the tape contents are the string uv, for $u, v \in \Gamma^*$, and the tape head location is the first symbol of v, the **configuration** of M is denoted as $C_k = u\,q\,v$.
- At two successive times/steps i and $j = i + 1$, we say that configuration C_i **yields** configuration C_j (or that M **moves** from C_i to C_j in one step), written as $C_i \vdash C_j$, if the transition function δ maps the states and tapes of C_i and C_j as follows (depending on the direction that the tape head moves):

  $C_i = ua\,q_i\,bv \vdash u\,q_j\,acv = C_j$ whenever $\delta(q_i, b) = (q_j, c, \text{L})$,

  $C_i = ua\,q_i\,bv \vdash uac\,q_j\,v = C_j$ whenever $\delta(q_i, b) = (q_j, c, \text{R})$.

Turing Machines: Configurations and Yields, II

Special Cases of Configurations and Yields

- When the head is at the left–hand end of the tape:

  $C_i = q_i\, bv \vdash q_j\, cv = C_j$ whenever $\delta(q_i, b) = (q_j, c, \mathrm{L})$,

  $C_i = q_i\, bv \vdash c\, q_j\, v = C_j$ whenever $\delta(q_i, b) = (q_j, c, \mathrm{R})$.

- When the head is at the right–hand end of the tape:

  $C_i = ua\, q_i \sqcup \vdash u\, q_j\, ac\sqcup = C_j$ whenever $\delta(q_i, \sqcup) = (q_j, c, \mathrm{L})$,

  $C_i = ua\, q_i \sqcup \vdash uac\, q_j \sqcup = C_j$ whenever $\delta(q_i, \sqcup) = (q_j, c, \mathrm{R})$.

- The **start configuration** on input w is $q_0\, w$ (i.e., at q_0, the head points to the first symbol of w).

- The **halting configurations** are two: the **accepting configuration** corresponding to the state q_{accept}, and the **rejecting configuration** corresponding to q_{reject}. Both halting configurations do not yield further configurations (i.e., $q_{\text{accept}}, q_{\text{reject}}$ do not occur in the domain of δ).

Turing Machines: Example of a Looping Computation

Example

Let the TM $M = (\{q_0, q_1\}, \{a, b\}, \{a, b, \sqcup\}, \delta, q_0, q_{\text{accept}} = \varnothing, q_{\text{reject}} = \varnothing)$ and transition function:

$$\begin{aligned}
\delta(q_0, a) &= (q_1, a, \mathrm{R}),\\
\delta(q_0, b) &= (q_1, b, \mathrm{R}),\\
\delta(q_0, \sqcup) &= (q_1, \sqcup, \mathrm{R}),\\
\delta(q_1, a) &= (q_0, a, \mathrm{L}),\\
\delta(q_1, b) &= (q_0, b, \mathrm{L}),\\
\delta(q_1, \sqcup) &= (q_0, \sqcup, \mathrm{L}).
\end{aligned}$$

Here are the sequence of configurations and yields on the input *abba* producing a loop:

$$\begin{aligned}
q_0\, abba\sqcup &\vdash a\, q_1\, bba\sqcup\\
&\vdash a\, q_0\, bba\sqcup\\
&\vdash ab\, q_1\, ba\sqcup\\
&\vdash a\, q_0\, bba\sqcup\\
&\vdash ab\, q_1\, ba\sqcup\\
&\vdash a\, q_0\, bba\sqcup\\
&\dots
\end{aligned}$$

Turing Machines: Example of a Halted Computation

Example

Let the TM $M = (\{q_0, q_1, q_{\text{accept}}\}, \{a, b\}, \{a, b, \sqcup\}, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} = \varnothing)$ and transition function:

$$\begin{aligned}
\delta(q_0, \sqcup) &= (q_1, \sqcup, \mathrm{R}),\\
\delta(q_1, a) &= (q_1, b, \mathrm{R}),\\
\delta(q_1, b) &= (q_1, a, \mathrm{R}),\\
\delta(q_1, \sqcup) &= (q_{\text{accept}}, \sqcup, \mathrm{R}).
\end{aligned}$$

Here is an example of the sequence of configurations and yields on the input $abaab$:

$$\begin{aligned}
q_0\, abaab\sqcup &\vdash b\, q_1\, baab\sqcup\\
&\vdash ba\, q_1\, aab\sqcup\\
&\vdash bab\, q_1\, ab\sqcup\\
&\vdash babb\, q_1\, b\sqcup\\
&\vdash babba\, q_1\, \sqcup\\
&\vdash babb\sqcup\ q_{\text{accept}}.
\end{aligned}$$

Apparently, this TM switches all a's and b's in the input.

Turing Machines: Recognized and Decided Languages

Definition: **Language Recognized by TM**

A TM M is said to **accept** an input string w if there exists a sequence of configurations $C_1, C_2, \ldots, C_k$ such that:

- C_1 is the start configuration of M on input w,
- $C_i \vdash C_{i+1}$, for all $i = 1, 2, \ldots, k-1$,
- C_k is an accepting configuration.

The **language recognized by** M is the set of all strings that M accepts. Moreover, a language is called **Turing–recognizable** (or **recursively enumerable language**) if there is a TM that recognizes it.

Definition: **Language Decided by TM**

A TM M is called **decider** if any input can be either accepted or rejected by M, but never looping. Moreover, a language is called **Turing–decidable** or simply **decidable** (or **recursive language**) if there is a decider TM that decides it.

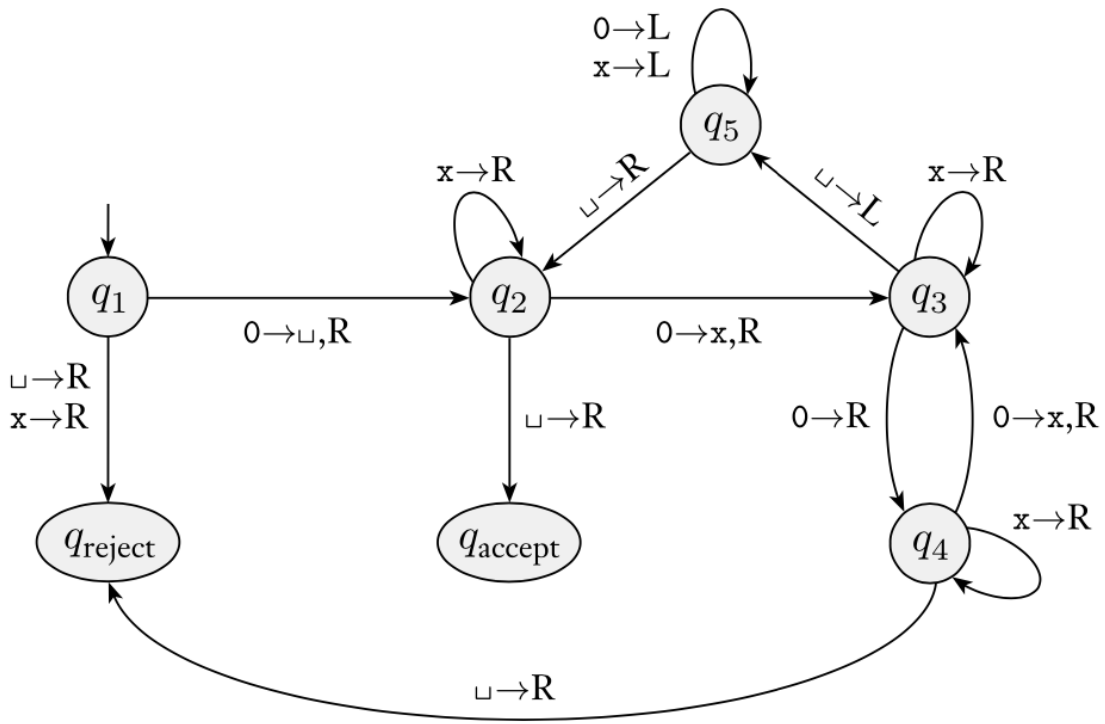
Turing Machines: Example 1

Example 1

TM that decides $L = \{0^{2^n} \mid n \geq 0\}$.

"On input string w:

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



Turing Machines: Example 1 (cont.)

Below are two sequences of configurations and yields, the first for an accepting and the second for a rejected input:

$$
\begin{aligned}
q_1\,0000\sqcup &\vdash \sqcup\, q_2\,000\sqcup \\
&\vdash \sqcup x\, q_3\, 00\sqcup \\
&\vdash \sqcup x0\, q_4\, 0\sqcup \\
&\vdash \sqcup x0x\, q_3\, \sqcup \\
&\vdash \sqcup x0\, q_5\, x\sqcup \\
&\vdash^4 q_5\ \sqcup\, x0x\sqcup \\
&\vdash \sqcup\, q_2\, x0x\sqcup \\
&\vdash \sqcup x\, q_2\, 0x\sqcup \\
&\vdash \sqcup xx\, q_3\, x\sqcup \\
&\vdash \sqcup xxx\, q_3\, \sqcup \\
&\vdash \sqcup xx\, q_5\, x\sqcup \\
&\vdash^4 q_5\ \sqcup\, xxx\sqcup \\
&\vdash \sqcup\, q_2\, xxx\sqcup \\
&\vdash^3 \sqcup xxx\, q_2\, \sqcup \\
&\vdash \sqcup xxx\, \sqcup\ q_{\text{accept}}.
\end{aligned}
$$

$$
\begin{aligned}
q_1\,000000\sqcup &\vdash \sqcup\, q_2\,00000\sqcup \\
&\vdash \sqcup x\, q_3\, 0000\sqcup \\
&\vdash \sqcup x0\, q_4\, 000\sqcup \\
&\vdash \sqcup x0x\, q_3\, 00\sqcup \\
&\vdash \sqcup x0x0\, q_4\, 0\sqcup \\
&\vdash \sqcup x0x0x\, q_3\, \sqcup \\
&\vdash \sqcup x0x0\, q_5\, x\sqcup \\
&\vdash^5 q_5\ \sqcup\, x0x0x\sqcup \\
&\vdash \sqcup\, q_2\, x0x0x\sqcup \\
&\vdash \sqcup x\, q_2\, 0x0x\sqcup \\
&\vdash \sqcup xx\, q_3\, x0x\sqcup \\
&\vdash \sqcup xxx\, q_3\, 0x\sqcup \\
&\vdash \sqcup xxx0\, q_4\, x\sqcup \\
&\vdash \sqcup xxx0x\, q_4\, \sqcup \\
&\vdash \sqcup xxx0x\, \sqcup\ q_{\text{reject}}.
\end{aligned}
$$

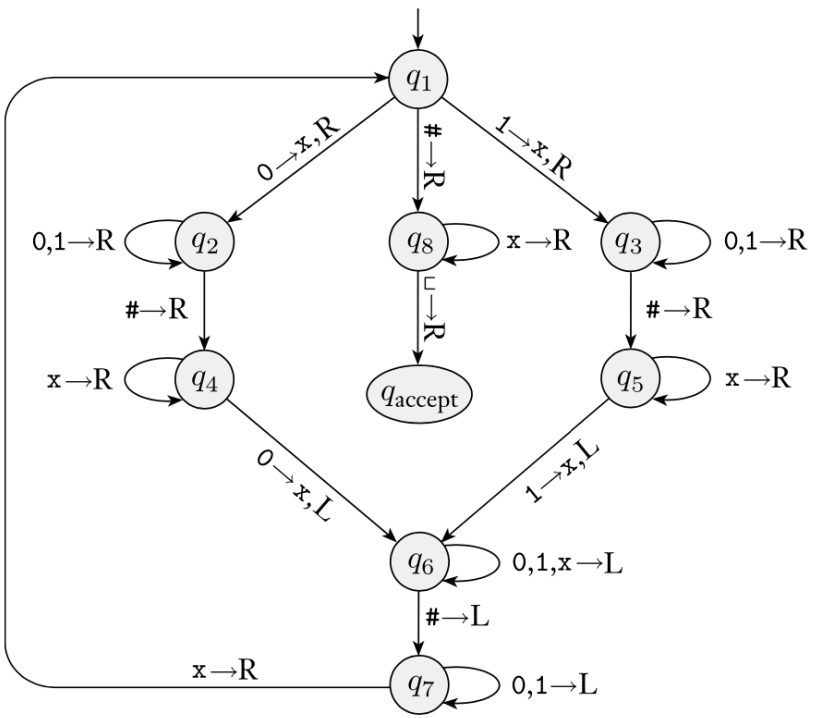
Turing Machines: Example 2

Example 2

TM that decides $L = \{w\#w \mid w \in (a+b)^*\}$.

"On input string w:

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 to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*."



Turing Machines: Example 3

Example 3

TM that decides $L = \{a^i b^j c^k \mid k = ij, i, j, k \geq 1\}$.

"On input string w:

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

Turing Machines: Example 4

Example 4

TM that decides $L = \{\#w_1\#w_2\#\cdots\#w_k \mid w_i \in (a+b)^*,$ $w_i \neq w_j,$ for all $i \neq j\}$ (*element distinctness problem*).

"On input w:

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

Multitape Turing Machines

Definition of a Multitape TM

A **multitape Turing machine** is still described by a 7–tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, but it contains several tapes, each one with its own tape alphabet Γ^k, and correspondingly its own head for reading and writing. Initially the input appears on tape 1, and the other tapes start out blank. Thus, the transition function is modified as follows:

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

where k is the number of tapes and S denotes no move option for the head.

Theorem

Every multitape TM has an equivalent single–tape TM.

Corollary

A language is Turing–recognizable if and only if some multitape TM recognizes it.

Nondeterministic Turing Machines

Definition of a Nondeterministic TM

A **nondeterministic Turing machine** is defined in the expected way, in which the transition (multi)function has the form:

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

Theorem

Every nondeterministic TM has an equivalent deterministic TM.

Corollary

A language is Turing–recognizable if and only if some nondeterministic TM recognizes it.

Corollary

A language is decidable if and only if some nondeterministic TM decides it.

Turing Machines as Enumerators

Definition of an Enumerator

An **enumerator** is a TM, defined in the expected way, in which Q contains an extra **print state** q_{print} and the transition function has the form:

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\} \times \Sigma_\varepsilon,$$

where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Theorem

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

Decidable Languages

Definitions: **The case of subsets of natural numbers**

- Let $S \subseteq \mathbb{N}$ be a (sub)set of natural numbers $\mathbb{N}$ (i.e., nonnegative integers). The **characteristic function** of S is the function $\chi_S : \mathbb{N} \to \mathbb{N}$ defined as:

$$\chi_S(x) = \begin{cases} 1, & \text{if } x \in S, \\ 0, & \text{if } x \notin S. \end{cases}$$

- Informally, an **algorithm** for a function $f : \mathbb{N} \to \mathbb{N}$ is a finite set of instructions, which given an input $x \in \mathbb{N}$, yields after a finite number of steps an output $y = f(x)$ and correctly decides whether $y \in \mathbb{N}$ or not.
- S is called **decidable** or **computable** or **recursive** if there exists an **algorithm** for the characteristic function of S.
- S is called **undecidable** or **noncomputable** set if S is not decidable.
- S is called **semidecidable** or **computably** or **recursively enumerable** if there exists an algorithm that, for any input $x \in S$, correctly decides when the output $y = \chi_S(x) = 1$, though it may give no answer (but not the wrong answer) for outputs $y = \chi_S(x) = 0$.

Decidable Languages

Definitions: **The case of languages as subsets of strings**

Let A be a language over an alphabet Σ.

- A is called **decidable** or **Turing–decidable** or **recursive** if there exists a **decider Turing machine** (otherwise called **total Turing machine**) that halts for every given input in such a way that, when given a string over Σ as input, it accepts it if it belongs to the language and rejects it otherwise.
- A is called **undecidable** or **Turing–undecidable** set if A is not decidable.
- A is called **semidecidable** or **Turing–recognizable** or **recursively enumerable** if there exists a **recognizing Turing machine**, that accepts all input strings in the language and rejects or loops all strings outside the language.

Decidable Problems for Regular Languages and CFLs

Theorem(s): **Decidable Languages**

All the following languages are decidable (the notation "$< \cdots >$" is used for the corresponding *encoding*; notice that $<X> = <X, \varepsilon>$):

1. $A_{\text{DFA}} = \{<B, w> \mid B \text{ is a DFA that accepts input string } w\}$,
2. $A_{\text{NFA}} = \{<B, w> \mid B \text{ is an NFA that accepts input string } w\}$,
3. $A_{\text{REX}} = \{<R, w> \mid R \text{ is a regular expression that generates input string } w\}$,
4. DFA emptiness testing:
   $E_{\text{DFA}} = \{<A> \mid A \text{ is a DFA and } L(A) = \varnothing\}$,
5. Testing of equivalent regular languages:
   $EQ_{\text{DFA}} = \{<A, B> \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$,
6. $A_{\text{CFG}} = \{<G, w> \mid G \text{ is a CFG that generates string } w\}$,
7. CFG emptiness testing:
   $E_{\text{CFG}} = \{<G> \mid G \text{ is a CFG and } L(G) = \varnothing\}$,
8. Testing of equivalent CFGs:
   $EQ_{\text{CFG}} = \{<G, H> \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$,
9. Context–Free Language.

Decidability Proofs

Proofs

1. $A_{\text{DFA}} = \{<A> \mid A \text{ is a DFA and } L(A) = \varnothing\}$

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

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

2. $A_{\text{NFA}} = \{<B, w> \mid B \text{ is an NFA that accepts input string } w\}$

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

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

3. $A_{\text{REX}} = \{<R, w> \mid R \text{ is a regular expression that generates input string } w\}$

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

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

Decidability Proofs (cont.)

Proofs

4. $E_{\text{DFA}} = \{<A> \mid A \text{ is a DFA and } L(A) = \varnothing\}$

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

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

5. $EQ_{\text{DFA}} = \{<A, B> \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$

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

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

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

6. $A_{\text{CFG}} = \{<G, w> \mid G \text{ is a CFG that generates string } w\}$. Notice that, if G is a CFG in Chomsky normal form, then it can be proved that, for any $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps are required for any derivation of w.

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

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

Decidability Proofs (cont.)

Proofs

7. $E_{\text{CFG}} = \{<G> \mid G \text{ is a CFG and } L(G) = \varnothing\}$

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

   1. Mark all terminal symbols in G.
   2. Repeat until no new variables get marked:
   3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
   4. If the start variable is not marked, *accept*; otherwise, *reject*."

8. $EQ_{\text{CFG}} = \{<G, H> \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$

   "On input w:

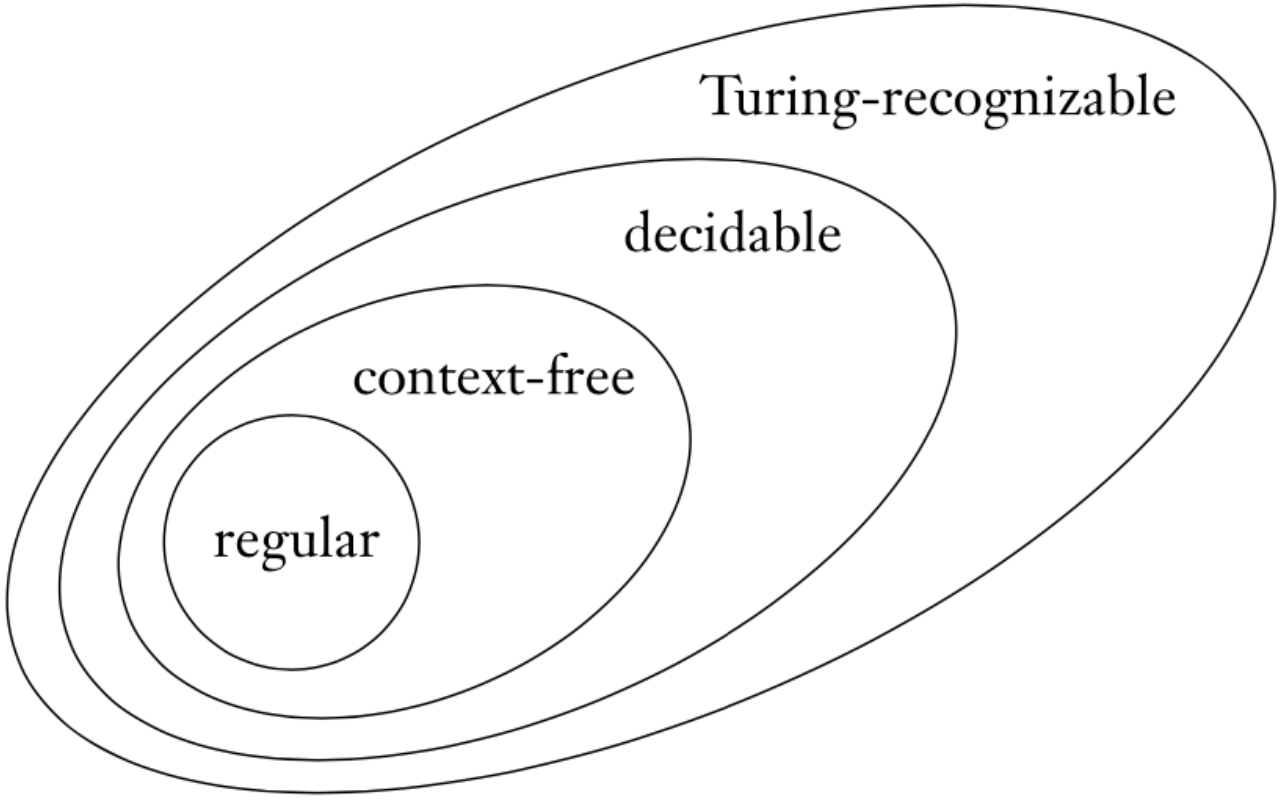
   1. Run TM S on input $\langle G, w \rangle$.
   2. If this machine accepts, *accept*; if it rejects, *reject*."

9. Context–Free Language A: Let G a CFG for A and M_G a TM that decides A. Build a copy of G into M_G as follows:

   "On input w:

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

Undecidability



Theorem: A_{TM} is Undecidable

The following language (encoded together with strings) is undecidable:

- $A_{\mathrm{TM}} = \{<M, w> \mid M \text{ is a TM that recognizes input string } w\}$,

Universal Turing Machine

Theorem: **Universal Turing Machine definition**

There is a Turing machine U_{TM} called **universal Turing machine** (**UTM**) such that when it runs on $< M, w >$, for a Turing machine M and a string w, U_{TM} simulates M running on w. In other words, U_{TM} recognizes the language A_{TM} with a high–level description:

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

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

Summary of Cantor's Set Theory

Definitions

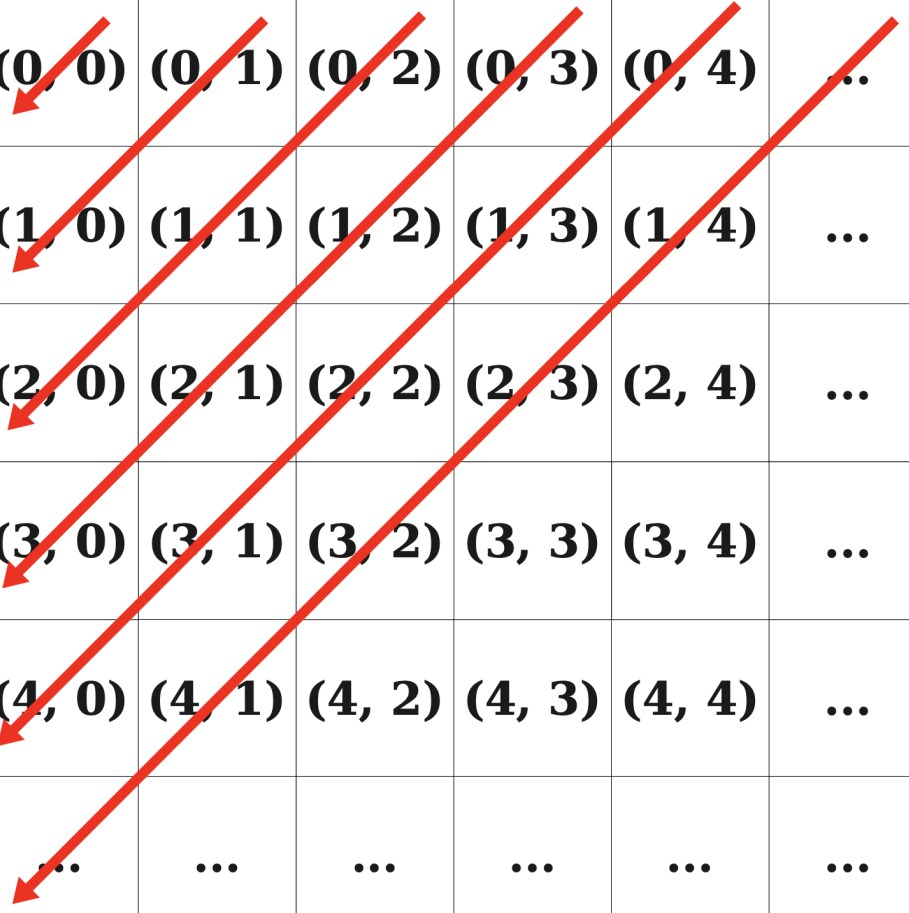
- Let X, Y sets. A **function** $f: X \to Y$ is a mapping such that every element of X (the **domain** of f) is associated with a *single* element of Y (the **codomain** of f).
- f is **one–to–one** (or **injection**) if, $\forall x_1, x_2 \in X$, if $f(x_1) = f(x_2)$, then $x_1 = x_2$.
- f is **onto** (or **surjection**) if, $\forall y \in Y, \exists x \in X$ such that $f(x) = y$.
- f is **bijective** if it is one–to–one (injective) and onto (surjective).
- Roughly speaking, the **cardinality** of a set X, denoted as $|X|$, is the number of elements it contains.
- $|X| = |Y| \iff \exists$ a bijection $f: X \to Y$.
- Defining cardinality on $X \times Y$ makes cardinality an **equivalence relation**.

Theorem: **Cantor's Diagonalization Method**

$$|\mathbb{N}| = |\mathbb{N}^2|.$$

Cantor's Pairing Function

	0	1	2	3	4	...
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	...
1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	...
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	...
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	...
4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	...
...	...	...	...	...	...	...



(0, 0)

(0, 1)
(1, 0)

(0, 2)
(1, 1)
(2, 0)

(0, 3)
(1, 2)
(2, 1)
(3, 0)

(0, 4)
(1, 3)
(2, 2)
(3, 1)
(4, 0)

...

$$f(m,n) = \frac{1}{2}(m+n)(m+n+1)+m$$

Uncountability of $\mathbb{R}$ to Turing-Nonrecognizability

Theorem: **Uncountability of Real Numbers**

$\mathbb{R}$ is uncountable.

Corollary: **Turing-Nonrecognizability of Languages**

Some languages are not Turing–recognizable.

Proof of Corollary

First, notice that Σ^* is countable for any (finite) alphabet Σ. Second, the set of all Turing machines is countable, since each Turing machine M has an encoding into a string $<M>$. Furthermore, let us consider the set $\mathcal{B}$ of *infinite binary sequences*, i.e., countable (unending) sequences of 0s and 1s. Similarly to the proof of the uncountability of $\mathbb{R}$, Cantor's diagonalization argument may show that $\mathcal{B}$ is uncountable. Let $\mathcal{L}$ the set of all languages over Σ. Then we claim that $\mathcal{L}$ is uncountable by constructing a correspondence with $\mathcal{B}$. For this purpose, for any $A \in \mathcal{L}$, we are considering the **characteristic sequence** of A, denoted as $\chi_A \colon \mathcal{L} \to \mathcal{B}$, such that, for any string $s \in \mathcal{L}$, the string $\chi_A(s) \in \mathcal{B}$ is generated by taking $\chi_A(s)_i = 1$, if $s \in A$, and $\chi_A(s)_i = 0$, if $s \notin A$. For example:

$$
\begin{array}{rccccccccccc}
\Sigma^* = \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots & \} \; ; \\
A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots & \} \; ; \\
\chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & .
\end{array}
$$

Clearly, the mapping χ_A is surjective and, thus, since $\mathcal{B}$ is uncountable, $\mathcal{L}$ is uncountable too. Hence, the set of all languages cannot be put in a correspondence with the set of all Turing machines, meaning that some languages are not recognized by any Turing machine.

Proof that A_{TM} is undecidable

Proof that A_{TM} is undecidable:

We assume that A_{TM} is decidable and obtain a contradiction. Let H be a decider for A_{TM}:

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

Next, construct a new Turing machine D with H as subroutine such that D calls H to determine what M does when the input to M is its own description $<M>$, in which case D does the opposite:

> "On input $\langle M\rangle$, where M is a TM:
> 1. Run H on input $\langle M, \langle M\rangle\rangle$.
> 2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

Thus,

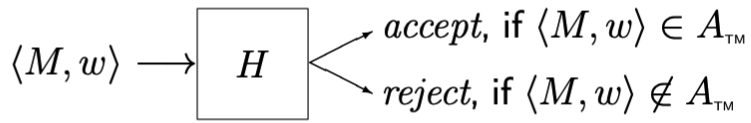
$$D(\langle M\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M\rangle. \end{cases}$$

However, when we run D with its own description $<D>$, we are getting exactly the opposite, which is a contradiction:

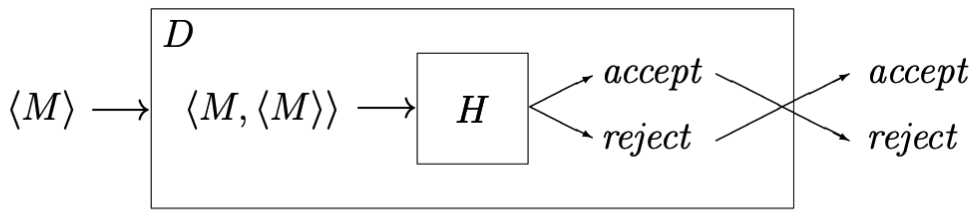
$$D(\langle D\rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

Proof that A_{TM} is undecidable (cont.)

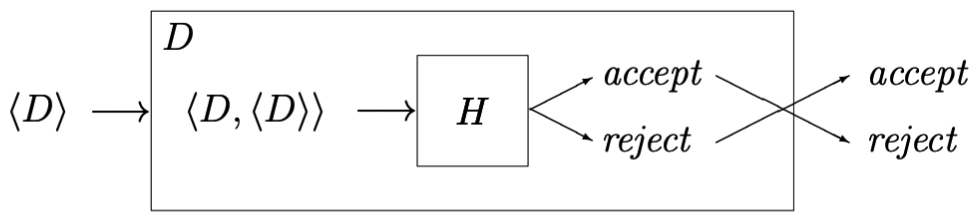
Overview of Proof: We use a proof by contradiction. Suppose A_{TM} is decided by some TM H, so H accepts $\langle M, w\rangle$ if TM M accepts w, and H rejects $\langle M, w\rangle$ if TM M doesn't accept w.



Define another TM D using H as a subroutine.



So D takes as input any encoded TM $\langle M\rangle$, then feeds $\langle M, \langle M\rangle\rangle$ as input into H, and finally outputs the opposite of what H outputs. Because D is a TM, we can feed $\langle D\rangle$ as input into D. What happens when we run D with input $\langle D\rangle$?



Note that D accepts $\langle D\rangle$ iff D doesn't accept $\langle D\rangle$, which is impossible. Thus, A_{TM} must be undecidable.

Proof that A_{TM} is undecidable (cont.)

Where does diagonalization occur in the proof of A_{TM}'s undecidability?

The following is a made-up table of behavior for TMs H and D. Apparently, D being a TM, it is listed as a row. However, D computes the opposite of the diagonal entries and, thus, a contradiction occurs at the point of the question mark, where the entry must be the opposite of itself.

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

Turing–Unrecognizable Languages

Definition

A language is called **co–Turing–recognizable** if it is the complement of a Turing–recognizable language.

Theorem

A language is decidable if and only if it is Turing–recognizable and co–Turing–recognizable.

Corollary

$\overline{A_{\mathrm{TM}}}$ is not Turing–recognizable.

Mapping Reducibility

Intuitive definition

A **reduction** is a way of converting problem A to problem B in such a way that a solution to problem B can be used to solve problem A. When this is true, solving problem A cannot be harder than solving problem B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is called **computable** (or **recursive**) if there exists some Turing machine M such that, on input w, M halts with just $f(w)$ on the tape.

Examples of computable functions

- Arithmetic operations on integers.
- Transformations of machine descriptions.

Formal Definition of Mapping Reducibility

Definition

Language A is said to be **mapping reducible** to language B, written $A \leq_{\mathrm{m}} B$, if there exists a computable function $f : \Sigma^* \to \Sigma^*$, called the **reduction** of A to B, such that, for every w,

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

Theorem

If $A \leq_{\mathrm{m}} B$ and B is decidable, then A is decidable.

Proof: Let M be the decider for B and f be the reduction from A to B. Then the following is a decider for A (because f is a reduction from A to B and M accepts $f(w)$, whenever $w \in A$):

N = "On input w:

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

Corollary

If $A \leq_{\mathrm{m}} B$ and A is undecidable, then B is undecidable.

Examples of Mapping Reducibilities

Theorem: The Halting Problem

Let $HALT_{\text{TM}}$ be the problem of determining whether a TM halts (by accepting or rejecting) on a given input:

$HALT_{\text{TM}} = \{<M, w> \mid M \text{ is a TM and } M \text{ halts on input } w\}$.

Then $A_{\text{TM}} \leq_{\text{m}} HALT_{\text{TM}}$ and, thus, $HALT_{\text{TM}}$ is undecidable.

Proof: The following TM F computes a reduction f:

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

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

Notice that improperly formed inputs are assumed to map to strings outside $HALT_{\text{TM}}$.

Examples of Mapping Reducibilities (cont.)

Theorem: **The Emptying Problem**

Let $E_{\text{TM}} = \{<M, w> \mid M \text{ is a TM such that } L(M) = \varnothing\}$. Then $A_{\text{TM}} \leq_{\text{m}} E_{\text{TM}}$ and, thus, E_{TM} is undecidable.

Proof: For each instance of $<M, w>$, we construct an instance of E_{TM} M_1 as follows. On input $x \neq w$, M_1 rejects x. Otherwise, M_1 simulates M on w. If M accepts (or rejects) w, then M_1 accepts (or rejects) w too. Hence, $<M, w> \in A_{\text{TM}}$ implies that $M_1 \in E_{\text{TM}}$ and, thus, E_{TM} is undecidable.

Theorem: **Equivalence of TMs to FA**

Let $REGULAR_{\text{TM}} = \{<M> \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$. Then $A_{\text{TM}} \leq_{\text{m}} REGULAR_{\text{TM}}$ and, thus, $REGULAR_{\text{TM}}$ is undecidable.

Proof: Let $<M, w> \in A_{\text{TM}}$. We construct a TM M_1 such that $L(M_1)$ is regular if and only if M accepts w in the following way. On input x, M_1 accepts x, if it is of the form 0^n1^n, for some $n \in \mathbb{N}$, and, otherwise M_1 simulates M on w, and accepts x if and only if M accepts w. Thus, $L(M_1) = \Sigma^*$ if and only if M accepts w, and $L(M_1) = \{0^n1^n \mid n \in \mathbb{N}\}$, unless $L(M_1)$ is not regular. Hence, $L(M_1) \in REGULAR_{\text{TM}}$ if and only if $<M, w> \in A_{\text{TM}}$, which implies that $REGULAR_{\text{TM}}$ is undecidable.

Rice's Theorem and Complementation

Theorem: Rice's Theorem

Any nonempty proper subset of the set of Turing–recognizable languages is undecidable.

Theorem

If $A \leq_{\mathrm{m}} B$ and B is Turing–recognizable, then A is Turing–recognizable.

Corollary

If $A \leq_{\mathrm{m}} B$ and A is not Turing–recognizable, then B is not Turing–recognizable.

Theorem

If EQ_{TM} is neither Turing–recognizable nor co–Turing–recognizable.

The Church–Turing Thesis

The Church–Turing Thesis

Any algorithm that can be performed on any computing machine can be performed on a Turing machine.

III. COMPLEXITY THEORY

Analyzing Algorithms

Time Complexity of $A = \{0^k 1^k \mid k \geq 0\}$

- $A \in \text{TIME}(n^2)$: By the following TM M_1:

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

- $A \in \text{TIME}(n \log n)$: By the following TM M_2:

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

- $A \in \text{TIME}(n)$: By the following TM M_3 with two tapes:

M_3 = "On input string w:

1. Scan across tape 1 and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*."

The Class P and Examples

Definition

The **polynomial time complexity class**, denoted P, is defined as:

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k).$$

Example 1: $PATH \in \mathrm{P}$.

The $PATH$ problem is defined as follows:

$$PATH = \{<G, u, v> \mid G \text{ is a directed graph } G = (V, E), u, v \in V \text{ and } G \text{ has a path from } u \text{ to } v\}.$$

By the following TM (breadth–first algorithm):

M = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Examples of Problems in Class P (cont.)

Example 2: $RELPRIME \in \mathrm{P}$

The $RELPRIME$ problem is defined as follows:

$$RELPRIME = \{(a, b) \in \mathbb{Z}^+ \times \mathbb{Z}^+ \mid \gcd(a, b) = 1\}.$$

By the following TM (Euclidean algorithm):

E = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

using the following subroutine:

R = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. Run E on $\langle x, y \rangle$.
2. If the result is 1, *accept*. Otherwise, *reject*."

Examples of Problems in Class P (cont.)

Example 3: $CFL \in \mathrm{P}$

Theorem: If A is a context–free language, then $A \in \mathrm{P}$.

Proof: Using a **dynamic programming** technique (Sipser, pages 290–1).

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

1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. [[$w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: [[i is the start position of the substring]]
8. Let $j = i + l - 1$. [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$: [[k is the split position]]
10. For each rule $A \to BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

The Class NP

The original definition of the NP class:

The **nondeterministic polynomial time complexity class**, denoted NP, is defined as the set of all languages that are decidable by polynomial time *nondeterministic* Turing machines. In other words, a language $A \in \text{NP}$ if there exists a nondeterministic Turing machine M and a polynomial p such that, for any $w \in A$, M accepts w in $p(|w|)$ time.

Definition: **Verifier**

A **verifier** for language A over Σ is a TM V (i.e., an algorithm) that halts on all inputs where:

$$A = \{w \in \Sigma^* \mid V \text{ accepts } <w, c> \text{ for some string } c \in \Sigma^*\}.$$

We refer to the string c as the **witness** or **certificate**. The verifier V is a **polynomial time verifier** if it runs in $p(|w|)$ time, for a fixed polynomial p and every input $w \in A$, which implies that $|c| < p(|w|)$. A language A is **polynomially verifiable** if it has a polynomial verifier.

The Class NP (cont.)

The modern definition of the NP class:

The **nondeterministic polynomial time complexity class** NP is the set of all languages that have polynomial time verifiers.

Theorem

A language $A \in \text{NP}$ if and only if A is decided by some nondeterministic polynomial time Turing machine.

Definition

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

Corollary

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k).$$

Examples of Problems in Class NP

Example 1: $CLIQUE \in$ NP, where

$CLIQUE = \{<G, k> \mid G \text{ is a graph that has a } k\text{–clique } \}$.

Example 2: $HP \in$ NP, where

$HP = \{<G> \mid G \text{ is a graph that has a Hamiltonian path } \}$.

Example 3: $COMPOSITE \in$ NP, where

$COMPOSITE = \{n \in \mathbb{Z} \mid \exists\, a, b \in [n-1] \text{ s.t. } n = ab\}$.

Example 4: $SUBSET\text{–}SUM \in$ NP, where

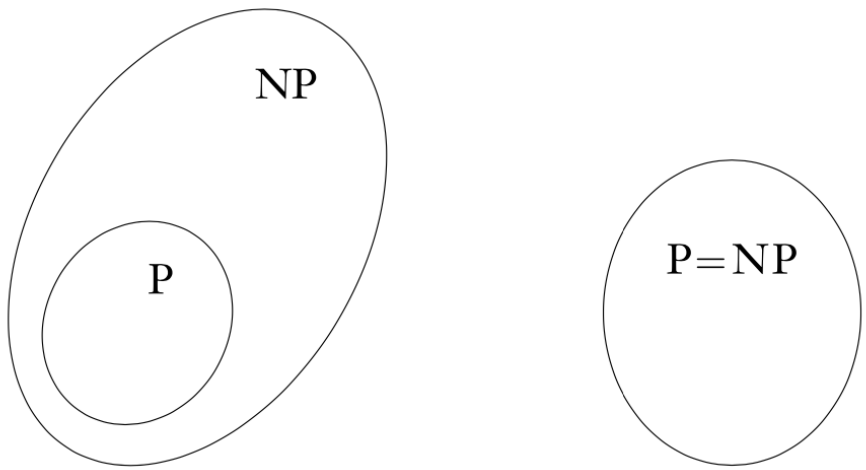
$SUBSET\text{–}SUM = \{<S, t> \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t\}$.

The P versus NP Question

Proposition

$$\mathrm{P} \subset \mathrm{NP}$$

One of these two possibilities is correct:



The best that we can currently prove is:

$$\mathrm{NP} \subset \mathrm{EXPTIME} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(2^{n^k}),$$

but we do not know whether NP is contained in a smaller deterministic time complexity class.

Polynomial Time Reducibility

Definition

A function $f : \Sigma^* \to \Sigma^*$ is called **polynomial time computable** if there exists some polynomial time Turing machine M such that, on input w, M halts with just $f(w)$ on the tape.

Definition

Language A is said to be **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_{\mathrm{p}} B$, if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, called the **polynomial time reduction** of A to B, such that, for every w,

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

Theorem

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

NP–Hardness and NP–Completeness

Definition

A problem B is called NP–**hard** if, for every $A \in \text{NP}$, $A \leq_{\text{p}} B$.

Definition

A language B is called NP–**complete** if $B \in \text{NP}$ and B is NP–hard.

Theorem

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

Theorem

If B is NP–complete and $B \leq_{\text{p}} C$, for some $C \in \text{NP}$, then C is NP–complete.

The Boolean Satisfiability Problem (SAT)

Definition: Boolean Satisfiable Function

Let $\phi : \{0,1\}^n \to \{0,1\}$ be a **Boolean function** (or **formula**), i.e., a composite expression involving **Boolean variables** (taking values TRUE, represented by 1, and FALSE, represented by 0) and **Boolean operations** (AND, OR, NOT). Then, the Boolean function ϕ is called **satisfiable** if there exists (assignment) $x \in \{0,1\}^n$ such that $\phi(x) = 1$.

Definition: Boolean Satisfiability Problem

The **satisfiability problem** (SAT) is to test whether a Boolean function is satisfiable, i.e.,

$$SAT = \{<\phi> \mid \phi \text{ is a satisfiable Boolean function}\}.$$

The Cook–Levin Theorem

SAT is NP–complete.

Time Complexity of Various Problems

- $SUBSET - SUM$ is NP–complete.
- $CLIQUE$ is NP–complete.
- $VERTEX - COVER$ is NP–complete. (The **vertex cover** of a graph is a subset of nodes such that every edge is incident to them.)
- $INDEPENDENT - SET$ is NP–complete. (A set of graph nodes is an **independent set** if its elements are pairwise non–adjacent in the graph.)
- $HAMPATH$ is NP–complete.
- $UHAMPATH$ is NP–complete. (This is the undirected graph version of the Hamiltonian path problem.)
- $HAMCYCLE$ is NP–complete.
- $TSP - OPT$ is NP–complete. (This is the **Traveling Salesman Optimization** problem.)