View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Saint Mary's University, Halifax: Institutional Repository

CLASSICAL-THEORETICAL

FOUNDATIONS OF COMPUTING

(a concise textbook)

Stavros Konstantinidis, PhD
Professor of Computing Science
Saint Mary's University

http://cs.smu.ca/˜stavros/

Saint Mary's University
May 2011

Preface

Many of the information technology products that we enjoy in our times are founded on theoretical tools of computing science. Some of these tools are presented in this concise textbook at an introductory level. In particular, we discuss basic concepts in the ***classical*** areas of formal languages, logic, and coding and information theory. We call these areas classical as they provided a lot of basic tools in the first few decades of the evolution of computing science. Of course in later stages of this evolution, people developed or utilized additional theoretical tools (such as fuzzy sets and fuzzy logic, neural networks and string distances) that are not covered here. However, the classical tools are so basic that they continue to be of importance at present and most likely in the foreseeable future as well.

Readers are expected to have some basic background in computer programming (in a high level language) and discrete mathematics (e.g., the concepts of set, function and relation, mathematical proof, etc.). This background knowledge is normally acquired after completing a couple of first year related courses in a typical Canadian university. Then, completing a course based on the material of this textbook will provide one with a basic understanding of the following.

- The existence of unsolvable computing problems.
- The role of formal logic in representing and deducing knowledge.
- The paradigm of declarative programming via the Prolog language.
- The role of grammars in specifying the syntax of programming expressions.
- The role of automata in recognizing programming expressions and communication languages.
- The role of codes in communicating information.
- The complexity involved in trying to solve certain important computing problems.

The author invites any comments and corrections that could improve this work – see the author's website for contact information.

GLOSSARY

Sets:

$\mathbb{N} = \{1, 2, 3, 4, \ldots\}$ = positive integers = natural numbers
$\mathbb{N}_0 = \{0, 1, 2, 3, 4, \ldots\}$ = nonnegative integers
$\mathbb{R}$ = real numbers
$\mathcal{B} = \{\mathbf{T}, \mathbf{F}\}$ = Boolean/truth values

Abbreviations:

iff = "if and only if". Used with two statements: A iff B means that statements A and B are either both true, or both false.

Some Greek letters:

α = alpha
β = beta
γ = gamma, Γ = capital gamma
δ = delta, Δ = capital delta
ε = epsilon
θ = theta, Θ = capital theta
λ = lambda, Λ = capital lambda
μ = mu
π = pi, Π = capital pi
σ = sigma, Σ = capital sigma
φ = phi, Φ = capital phi
ω = omega, Ω = capital omega.

Description/Computing methods	***Corresponding Languages***
CFG= Context Free Grammars	context free
CSG= Context Sensitive Grammars	context sensitive
DFA= Determ. Finite Automata	regular
DPDA= Det. Pushdown Automata	deterministic context free
TM= (Determ.) Turing Machines	recursively enumerable
NFA= Nondeterm. Finite Automata	regular
NPDA= Nondet. Pushd. Automata	context free
NTM= Nondeterm. Turing Machines	recursively enumerable
REX= Regular Expressions	regular
UNG= Unrestricted Grammars	recursively enumerable
Decision Turing Machines	recursive

Contents

0 INTRODUCTION

0.1 What and why

As noted in the Preface, we shall study basic concepts in the ***classical*** areas of formal languages, logic, and coding and information theory. Next we give a quick summary of the uses of these areas.

- **Uses of formal language tools**.
  1. Defining the precise syntax of the set of valid expressions/words that are important in some application (e.g., in programming languages, data communications, user input interfaces, mathematical software).
  2. Testing the correctness of, and generally processing, the expressions/words of some application (e.g., compiling programs, decoding transmitted data, processing natural language sentences).
  3. In pattern matching: e.g., finding a certain word in some text possibly allowing a few spelling errors.
  4. Understanding the limits of computing: are there computing problems that cannot be solved?

- **Uses of logic tools**.
  1. Boolean expressions in programming languages.
  2. Theoretical foundation for realizing computing hardware.
  3. Program specification and correctness.
  4. Knowledge representation and natural language processing.
  5. Automated inference and reasoning (expert systems).

- **Uses of coding and information theory tools**.
  1. Error control in data communications and storage.
  2. Data compression.
  3. Machine learning and data analysis.

0.2 Review of Discrete Mathematics Concepts

In this section we review the basic mathematical concepts set, function, relation and graph. Moreover, we discuss the concept of cardinality for infinite sets.

0.2.1 Sets

• **Set**. A set is an ***unordered*** collection of ***distinct*** entities. Here are a few examples of finite sets:

$\emptyset = \{\} =$ the empty set.
$\mathcal{B} = \{\mathbf{T}, \mathbf{F}\} =$ the set of Boolean values,
{Halifax, Sydney, Truro} = {Truro, Halifax, Sydney}.

Ex. 0.1 (Infinite sets)**.**

$$\begin{aligned} \mathbb{N} &= \{1, 2, 3, \ldots\} \text{ (the positive integers = natural numbers)} \\ \mathbb{N}_0 &= \{0, 1, 2, 3, \ldots\} \text{ (the nonnegative integers)} \\ \mathbb{R} &= \text{set of real numbers. E.g., } 3, -3, \frac{3}{7}, \sqrt{7}, \pi, -2.67 \\ (-3.4, \infty) &= \{x \mid x \in \mathbb{R} \text{ and } x > -3.4\}. \end{aligned}$$

• **Subset**. We write $A \subseteq B$ to indicate that A is a subset of B, which means that, if $x \in A$ then $x \in B$ (possibly $A = B$). For example, $\mathbb{N}_0 \subseteq \mathbb{R}$. To indicate that A is a proper subset of B we write $A \subset B$, or $A \subsetneq B$. The empty set is a subset of every set, that is, $\emptyset \subseteq A$ for all sets A.

• **Basic set operations**. The three basic set operations are:

$$\begin{aligned} A \cup B &= \{x \mid x \in A \text{ or/and } x \in B\} \qquad \text{(set union)} \\ A \cap B &= \{x \mid x \in A \text{ and } x \in B\} \qquad \text{(set intersection)} \\ A - B &= A \setminus B = \{x \mid x \in A \text{ and } x \notin B\} \quad \text{(set difference)} \end{aligned}$$

Ex. 0.2. For every two sets A and B, we have
$A \cup B = B \cup A, \quad A \cap B = B \cap A$
$A = A \cup A = A \cap A = A \cup \emptyset$
$A \cap \emptyset = A - A = \emptyset$
If $A \subseteq B$ then $A \cap B = A,\ A \cup B = B,\ A - B = \emptyset.$

• **Power Set and Cartesian (cross) product**. For any set A, its power set is 2^A = the set of all subsets of A, including $\emptyset$ and A. For any sets A, B, their Cartesian (or cross) product is the set of all (ordered) pairs of elements from the two sets, that is,

$$A \times B = \{(a, b) \mid a \in A,\ b \in B\}.$$

• **Cardinality of finite set**. $|A|$ = the number of elements in A. We have that

$$|A \times B| = |A| \cdot |B| \quad \text{and} \quad |2^A| = 2^{|A|}.$$

Ex. 0.3. Let $A = \{-1, 2, 5\}$. Then $|A| = 3$,

$$2^A = \{\emptyset, \{-1\}, \{2\}, \{5\}, \{-1, 2\}, \{-1, 5\}, \{2, 5\}, A\}$$

and

$$\begin{aligned} A \times A &= \{(-1,-1), (-1,2), (-1,5), (2,-1), \\ &\quad (2,2), (2,5), (5,-1), (5,2), (5,5)\}. \end{aligned}$$

• **Method INF1: Show that a subset M of $\mathbb{N}_0$ is infinite**. It is sufficient to show that no integer is the largest element of the given set M. That is, for any positive integer n we pick, there is an element of M that is larger than n.

Ex. 0.4. Show that the set of all even natural numbers is infinite.

Solution. Pick any positive integer n. Then, $2n$ is an even natural number greater than n. Hence, there are infinitely many even numbers. □

0.2.2 Functions

• **Function** $f : A \to B$. Is a mapping associating to each element $a \in A$ exactly one element in B, called the image of a and denoted by $f(a)$. The set A is called the domain of f. Note that:

- different a's can have the same image;
- some $b \in B$ might not be the image of any a.

Ex. 0.5. Let $\textsc{Prov}$ be the set of provinces of Canada and let $\textsc{City}$ be the set of Canada's cities. We can define the function

$$\mathtt{cap}\colon \textsc{Prov} \to \textsc{City}$$

such that $\mathtt{cap}(P)$ is the capital city of the province P. For example, $\mathtt{cap}\,(\text{Ontario}) = \text{Toronto}$. This is an example of a finite function, that is, the set of pairs $(P, \mathtt{cap}\,(P))$ is finite.

Ex. 0.6. The function $f : \mathbb{R} \to \mathbb{R}$ with $f(x) = x^2$ is an example of an infinite function.

• **Method INF2: Show that a set X is infinite**. There are many ways. When possible, we show that X contains another set Y which we know is infinite. Another way is to define a function $f : X \to \mathbb{N}_0$ that assigns a certain value $f(x)$ to every element x of X, and show that the set of values $\{f(x) \mid x \in X\}$ is infinite (that is, these values constitute an infinite subset of $\mathbb{N}_0$).

Ex. 0.7. Show that the set $\textsc{Circle}$ of all circles is infinite. Use the fact that, for every number x, there is a circle whose radius is of length x.

Solution. To every circle C, we assign the length $r(C)$ of the radius of C. As there is a circle of radius n, for every positive integer n, the set of all values $r(C)$ includes all positive integers $\mathbb{N}$. Hence, $\{r(C) \mid C \in \textsc{Circle}\}$ is infinite and, therefore, also the set $\textsc{Circle}$ is infinite. □

• **One-to-one, onto, bijective functions**. A function $f : A \to B$ is called 1-1 (one-to-one), or injective, when all the a's in the domain A have different images, that is, if $a_1 \neq a_2$ then $f(a_1) \neq f(a_2)$. The function `cap` defined in Ex. 0.5 is 1-1. The function f is called onto, or surjective, when all b's are images of a's, that is, for every $b \in B$ there is an $a \in A$ such that $b = f(a)$. The function `cap` is not onto, as there are elements in $\textsc{City}$ that are not capitals of any province. The function is called bijective if it is both, 1-1 and onto.

• **Order of magnitude of a function** $f : \mathbb{N} \to (0, +\infty)$. Here we are interested in comparing real functions on $\mathbb{N}$ in terms of the order of magnitude of their values. In particular, we want to express the order of magnitude of the function f in terms of basic functions like

$$\log n,\ \sqrt{n},\ n,\ n^2,\ n^k,\ 2^n.$$

For example, for $f(n) = 5n^2 + 7n$, we understand that the term n^2 has the highest order of magnitude and we can write $f(n) = O(n^2)$. In other words, f is a quadratic-bounded function. In general, we write $f(n) = O(g(n))$ to mean that there is a constant $c > 0$ such that $f(n) \leq cg(n)$. For example, we verify $5n^2 + 7n = O(n^2)$ as follows:

$$5n^2 + 7n \leq 5n^2 + 7n^2 = 12n^2.$$

Of course, as $n^2 \leq 2^n$, we can also write $f(n) = O(2^n)$, which is an overestimate for the order of magnitude of f. If we have that $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then we write $f(n) = \Theta(g(n))$ and we say that the two functions have the same order of magnitude. For example, $5n^2 + 7n = \Theta(n^2)$.

0.2.3 Cardinalities of Infinite Sets

We now extend the concept of cardinality to infinite sets[1]. The main result here is that there are infinite sets whose cardinalities are different! The set $\mathbb{N}_0$ of nonnegative integers has the ***smallest*** infinite cardinality, which is denoted as $\aleph_0$ (aleph zero), that is,

$$|\mathbb{N}_0| = \aleph_0.$$

• **Comparing cardinalities of sets**. Let A and B be two sets, and let $f : A \to B$.

1. If f is 1-1 (one-to-one) then we write $|A| \leq |B|$, or $|B| \geq |A|$.
2. If f is onto then we write $|B| \leq |A|$, or $|A| \geq |B|$.
3. If f is bijective then $|A| = |B|$.

[1] Although our treatment of these concepts is not thorough from a mathematical point of view, it allows one to compare correctly the cardinalities of two sets.

As usual, $|A| < |B|$ means that $|A| \leq |B|$ and $|A| \neq |B|$, which implies that there is no onto function from A to B, and there is no 1-1 function from B to A.

Ex. 0.8. Consider the case where f is 1-1. Then f is able to assign a different $b \in B$ to each $a \in A$, which means that $|A| \leq |B|$. Similarly, if f is onto then every $b \in B$ is the value $f(a)$ of some $a \in A$, which means that A has to contain at least one element for each b, that is, $|B| \leq |A|$. For example, if f is 1-1 and $A = \{3, 7, 28\}$ then the set B must contain the values $f(3), f(7), f(28)$, which are all different as f is 1-1. Hence, $|B| \geq |A|$. This applies also to the case where A is infinite.

Ex. 0.9. In the case of infinity, there exist situations that are impossible with finite sets. In particular, we can have a situation where a ***proper*** subset of a set S has the same cardinality as S! For example, let B be the set of all even positive integers and let $A = \mathbb{N}$. As B is a subset of A, we have that $|B| \leq |A|$. On the other hand, as the function $f : A \to B$ with $f(n) = 2n$ is 1-1, we have that $|A| \leq |B|$. Hence, $|A| = |B|$. Similarly, we have that

$$|\mathbb{N}| = |\mathbb{N}_0|.$$

Another instance of this phenomenon is that the real interval $(0, 1)$ has the same cardinality as that of the entire set of real numbers.

Definition 0.1. Let A be a set.

1. If $|A| \leq |\mathbb{N}_0|$ then A is called countable (or enumerable).

2. If $|A| = |\mathbb{N}_0|$ then A is called countably infinite.

3. If $|A| > |\mathbb{N}_0|$ then A is called uncountable.

The fact that a set A is countable means that, in principle, we can enumerate all the elements of A in the form of a sequence[2] $x_0, x_1, x_2, \ldots$ such that we can say that x_0 is the first one and, in general, x_{i+1} is the successor of x_i. On the other hand, when the set is not countable then there is no way to arrange all elements of

[2] Indeed, A countable means that there is an *onto* function $g : \mathbb{N}_0 \to A$, which implies that $A = \{g(0), g(1), g(2), \ldots\}$.

the set in a sequence such that we can tell what the successor of a given $x \in A$ is. The next fundamental result gives two examples of uncountable sets.

Theorem 0.1. *(Cantor's Theorem) The set of real numbers is uncountable, that is,* $|\mathbb{R}| > |\mathbb{N}_0|$*. Moreover, the reals have the same cardinality as the set of all subsets of* $\mathbb{N}_0$*, that is,* $|2^{\mathbb{N}_0}| = |\mathbb{R}|$*.*

The cardinality of the real numbers is called the continuum.

• **Proof of** $|\mathbb{R}| > |\mathbb{N}_0|$. It is sufficient to show that the real interval $(0, 1)$ is not a countable set. We assume for the sake of contradiction that $(0, 1)$ is countable. Then we can enumerate all numbers in that interval as $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$. As each number t in the interval has an infinite decimal representation $0.t_0t_1t_2\cdots$, we can enumerate all the elements of $(0, 1)$ as follows:

$$\begin{aligned} x^{(0)} &= 0.x_0^{(0)}x_1^{(0)}x_2^{(0)}\cdots \\ x^{(1)} &= 0.x_0^{(1)}x_1^{(1)}x_2^{(1)}\cdots \\ x^{(2)} &= 0.x_0^{(2)}x_1^{(2)}x_2^{(2)}\cdots \\ &\cdots \end{aligned}$$

Now consider all the diagonal digits $x_0^{(0)}, x_1^{(1)}, x_2^{(2)}, \ldots$ and define a number $t = 0.t_0t_1t_2\cdots$ such that each digit t_i is chosen to be different from 0, 9 and $x_i^{(i)}$. Then, the number t must be in $(0, 1)$ and, by the assumption, t must be equal to the number $x^{(j)}$ for some index $j \in \mathbb{N}$, that is,

$$t \;=\; 0.t_0t_1t_2\cdots \;=\; 0.x_0^{(j)}x_1^{(j)}x_2^{(j)}\cdots.$$

There is a contradiction, however, when we recall that the digit t_j is different from the digit $x_j^{(j)}$. Hence, the real interval $(0, 1)$ must be uncountable.
□

• **Method COUNT1: Show that a set X is countable**. We are given the description of the set X. If the set is finite then it's countable. Else, we prepare an infinite empty array, and we explain ***how*** to insert each element of X into an empty entry of the array:

0	1	2	3	$\cdots$
x_0	x_1	x_2	x_3	$\cdots$

The array indexes correspond to the elements of $\mathbb{N}_0$ and the method of inserting the elements of X into the array corresponds to a function $f : X \to \mathbb{N}_0$. As each element of X goes to a different entry the function f is one-to-one and, therefore, X is countable. ***Note*** that, in general, not all entries of the array must be filled. Moreover, sometimes it might be convenient to index the array with a *start index other than 0*. For example, we might use an array whose entries are indexed as $5, 6, 7, 8, \ldots$.

Ex. 0.10. Is the set of integers $\mathbb{Z}$ countable? Recall,

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

Solution. We insert all the elements of $\mathbb{Z}$ into an array:

0	1	2	3	4	5	6	$\cdots$
0	1	-1	2	-2	3	-3	$\cdots$

We insert 0 into entry 0. Then $1, -1$ go to the next two entries, $2, -2$ to the next entries, and so on. More formally, each positive integer n is inserted into the entry $2n - 1$, and each negative $-n$ into entry $2n$. Hence, $\mathbb{Z}$ is countable. □

0.2.4 Relations and Graphs

• **Relation**. A relation R on two sets A, B is a subset of $A \times B$. The fact that $(a, b) \in R$ is intended to say that a and b are related according to R. For example, let Dept be a set of departments and Emp be a set of employees. We can define a relation

$$\textsc{WorksFor} \subseteq \textsc{Emp} \times \textsc{Dept}$$

such that $(E, D) \in$ WorksFor iff employee E works for department D. An alternate notation for $(a, b) \in R$ is $R(a, b)$, which is normally read as $R(a, b)$ ***is true***. In this case, R is viewed as a function from $A \times B$ into the set of truth values $\mathcal{B} = \{\mathbf{T}, \mathbf{F}\}$. For example, if WorksFor(E, D) is true then E works for department D. This notation is also used with sets. For example, Emp(E) means that $E\ \in$Emp, that is, E is an employee. In general we can have relations on more than two sets. For example, let Date be the set of all dates. We can have a relation

$$\textsc{WorksForSince} \subseteq \textsc{Emp} \times \textsc{Dept} \times \textsc{Date}$$

such that $(E, D, T) \in$WorksForSince means that employee E has been working for department D since date T. In general if a relation is a subset of $A_1 \times \cdots \times A_n$, the cross product of n sets, then we say that the relation is of ***arity*** n, or that it is an n-ary relation. In particular, if $n = 1$ we have a unary relation, if $n = 2$ we have a binary relation, and if $n = 3$ we have a ternary relation.

• **Binary Relation** R **on** A. This is a subset of $A \times A$. This type of relation relates two elements of the same set. For example, the relation "<" on the set $\mathbb{R}$ of real numbers is the "less than" relation such that $(x, y) \in$ "<", or $< (x, y)$, or $x < y$, means that the number x is less than y. Some particular types of a relation R are the following.

- If $(a, a) \in R$ for all a, then R is reflexive. For example, the relation "⊆" on sets is reflexive, as every set is a subset of itself.
- If $(a, a) \notin R$ for all a, then R is irreflexive. For example, the relation "<" on $\mathbb{R}$ is irreflexive.
- If $(b, a) \in R$, whenever $(a, b) \in R$, then R is symmetric. For example, the relation "is sibling of" on people is symmetric.
- If $(b, a) \notin R$, whenever $(a, b) \in R$ and $a \neq b$, then R is antisymmetric. For example, the relation "<" on $\mathbb{R}$ is antisymmetric.
- If $(a, c) \in R$, whenever $(a, b), (b, c) \in R$, then R is transitive. For example, the relation "⊆" on sets is transitive.
- If R is transitive, irreflexive, and such that exactly one of $(a, b) \in R$ and $(b, a) \in R$ is true for all a and b, then R is called a (strict) total order. For example, the relation "<" on $\mathbb{R}$ is a (strict) total order.

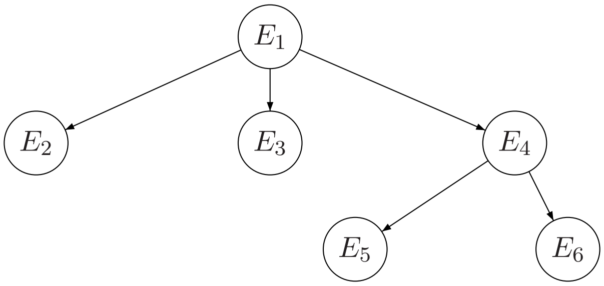


Figure 1: Graph representation of the relation SUPERV.

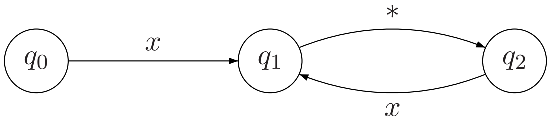


Figure 2: A labeled graph.

• **Directed graph representation of** $R \subseteq A \times A$. We use a vertex (node) for each $a \in A$ and an edge (arc) between two nodes a and b iff $(a, b) \in R$. For example, let SUPERV $\subseteq$ EMP $\times$ EMP be the supervisor relation between employees, that is, $(E_1, E_2) \in$SUPERV means that E_1 is the (direct) supervisor of E_2. If

$$\textsc{Superv} = \{(E_1, E_2), (E_1, E_3), (E_1, E_4), (E_4, E_5), (E_4, E_6)\}$$

then we can represent this relation using the graph of Fig. 1.

• **Graph concepts**. A path in a graph R is a finite sequence

$$(a_0, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n)$$

of consecutive edges in R. For example, $(E_1, E_4), (E_4, E_6)$ is a path in the graph of Fig. 1. A labeled graph (or weighted graph) is a relation $G \subseteq A \times L \times A$ where A is the set of nodes and L is the set of labels (digits, symbols, etc.). An edge of G is a triple $(a, \ell, b) \in A \times L \times A$, and a path of G is a finite sequence of consecutive edges:

$$(a_0, \ell_1, a_1), (a_1, \ell_2, a_2), \ldots, (a_{n-1}, \ell_n, a_n).$$

In this case, the label of the path is the sequence $\ell_1, \ell_2, \ldots, \ell_n$ of the labels appearing in the path. For example,

$$(q_0, x, q_1), (q_1, *, q_2), (q_2, x, q_1), (q_1, *, q_2), (q_2, x, q_1)$$

is a path in the labeled graph in Fig. 2

• **Reflexive and transitive closure of a relation** $R \subseteq A \times A$. This is another binary relation on A:

$R^* = \{(a, b) \mid b = a,$ or there is a path $(a_0, a_1), \ldots, (a_{n-1}, a_n)$ in the graph of R with $n \geq 2, a_0 = a, a_n = b\}$.

For example, (E_1, E_1) and (E_1, E_5) are in $\textsc{Superv}^*$, but (E_2, E_5) is not.

Exercises

Ex. 0.11. Use Method COUNT1, the array method, to show that the following set is countable

$$\{\frac{1}{n} \mid n \in \mathbb{N}_0,\ n \geq 2\} \cup \{\sqrt{7}, -12, 6.47\}.$$

Ex. 0.12. Use Method COUNT1, the array method, to show that the following set is countable.

$$\{\sqrt{n} \mid n \in \mathbb{N},\ n \geq 2,\ n \neq 5\}.$$

Ex. 0.13. Consider the relation $\textsc{Mult} \subseteq \mathbb{N} \times \mathbb{N}$ such that $(m, n) \in \textsc{Mult}$ if m is a multiple of n, that is, $m = pn$ for some $p \in \mathbb{N}$. For example, $(15, 3) \in \textsc{Mult}$, but $(17, 3) \notin \textsc{Mult}$. Is the relation $\textsc{Mult}$ finite? Reflexive? Symmetric? Transitive?

Ex. 0.14. Consider all paths from q_0 to q_1 in Fig. 2. For example,

$$(q_0, x, q_1), (q_1, *, q_2), (q_2, x, q_1)$$

is such a path. Is the set of these paths finite or infinite?

1 LANGUAGES & UNDECIDABILITY: Introduction

In the introductory section we talked briefly about the role of formal language theory in the field of computing. In this section we introduce a few basic elements of formal languages and establish the fact that there are computing problems that are not solvable by any means.

1.1 Alphabet, Words (or strings)

An alphabet is a nonempty set Σ whose elements are called symbols, or letters. Here are some examples of alphabets:

- $\Sigma_2 = \{0, 1\}$, $\mathcal{B} = \{\mathbf{T}, \mathbf{F}\}$ (binary alphabets)
- $\Sigma_q = \{0, 1, \ldots, q-1\}$ (q-ary alphabet)
- $\{a, c, g, t\}$ (the DNA alphabet)
- $\{a, b, c, \ldots, z\} \cup \{A, B, \ldots, Z\}$ (the English alphabet).
- $\Sigma_{\text{calc}} = \{0, 1, \ldots, 9, (,), +, *, -, /\}$ (alphabet for simple calculators)

• **Word or String (over Σ).** A finite sequence of symbols from Σ. For example:

- $00110, 0, 11$ are words over Σ_2
- $accgtt, aaaa$ are DNA words
- $coat, ctrl, www$ are words over the English alphabet.
- $(2 + 27) * 14$, $(12 * +()78$ are words over the alphabet Σ_{calc}.

Words are used to represent the various types of data that can be processed by computing devices. For example, decimal numbers are words over the alphabet Σ_{10}, genes of an organism are words over the DNA alphabet, etc.

• **Length $|w|$ of word w.** Is the number of symbols in w. For example,

$$|01001| = 5, \ \ |acggtac| = 7, \ \ |a| = 1.$$

• **i-th symbol of w.** We write $w(i)$ for the i-th symbol of a word w. For example, if $w = accgt$ then $w(1) = a$ and $w(4) = g$. Moreover, for every word w we have

$$w = w(1) \cdots w(|w|).$$

• **Word concatenation**. For words u and v, uv is the word that consists of the symbols of u followed by the symbols of v. For example, if $u = 011$ and $v = 0001$ then $uv = 0110001$.

• **Empty word** λ. The word containing no symbols. As we cannot see it, we use the special symbol λ. It has the properties

$$|\lambda| = 0, \;\; w\lambda = \lambda w = w, \text{for all words } w.$$

• **All words**. The set of all words is denoted by Σ^*. The set of all nonempty words is denoted by Σ^+, that is, $\Sigma^+ = \Sigma^* - \{\lambda\}$

• **More notation on words**. For any $n \in \mathbb{N}_0$ and $w \in \Sigma^*$, $(w)^n$ is the word consisting of n copies of w. If σ is a single symbol then we write σ^n instead of $(\sigma)^n$. For example

$$(011)^3 = 011011011.$$

Note that $w^0 = \lambda$. We write $(w)^R$ for the reversal of the word w, that is, if w is of length n then $(w)^R = w(n) \cdots w(1)$. For example,

$$(00111)^R = 11100.$$

Ex. 1.1. Show that the set of lengths of all words is infinite.

Solution. The set in question is $\{|w| \mid w \in \Sigma^*\}$, which is a subset of $\mathbb{N}_0$. We use Method INF1 in Section 0.2.1. Pick any positive integer n. We have to find a word whose length is greater than n. Obviously σ^{n+1} is such a word, where σ is any letter. Hence, the set of all word lengths is infinite. □

• **Equal words**. Two words u, v are equal if, $|u| = |v|$ and $u(i) = v(i)$ for every index i.

Ex. 1.2. Find all solutions of the word equation $x1x = 10x01$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Solution. The lengths of the left- and right-hand side words of the equation must be equal:

$$|x1x| = |10x01|,$$

which is equivalent to $2|x| + 1 = |x| + 4$; hence, the length of x must be 3, that is, $x = x(1)x(2)x(3)$. If we align the two sides of the equation we get:

$$\begin{array}{llllllll}
\mathrm{x}(1) & \mathrm{x}(2) & \mathrm{x}(3) & 1 & \mathrm{x}(1) & \mathrm{x}(2) & \mathrm{x}(3) & = \\
1 & 0 & \mathrm{x}(1) & \mathrm{x}(2) & \mathrm{x}(3) & 0 & 1 & = \\
1 & 0 & 1 & 0 & 1 & 0 & 1 &
\end{array}$$

This implies that the symbol $x(2)$ must be 0 and 1, which is impossible. Hence the set of solutions is $\emptyset$. □

Ex. 1.3. Find all solutions of the word equation $0x1 = 01x$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Solution. Here we test a few cases for x based on the length of x. For example, if $|x| = 0$, that is x is empty, then the left-hand side is $0\lambda 1 = 01$ and the right-hand side is $01\lambda = 01$, so the equation is satisfied for $x = \lambda$. Similarly if $|x| = 1$, that is x is a single symbol $\sigma \in \Sigma$, then the equation becomes $0\sigma 1 = 01\sigma$ and is satisfied when $x = \sigma = 1$. In general, if the length of x is $\ell \geq 2$, then the two sides of the equation align as follows:

$$\begin{array}{llllllll}
0 & \mathrm{x}(1) & \mathrm{x}(2) & \mathrm{x}(3) & \cdots & \mathrm{x}(\ell) & 1 & = \\
0 & 1 & \mathrm{x}(1) & \mathrm{x}(2) & \cdots & \mathrm{x}(\ell-1) & \mathrm{x}(\ell) & = \\
0 & 1 & 1 & 1 & \cdots & 1 & 1. &
\end{array}$$

Hence, the equation is satisfied when $x = 1^\ell$. So the set of solutions is $\{1^\ell \mid \ell \in \mathbb{N}_0\}$. □

Ex. 1.4. Show that the set Σ^* of all words is infinite.

Solution. The length function $|\cdot| : \Sigma^* \to \mathbb{N}_0$ assigns the length $|w|$ to each word $w \in \Sigma^*$. By Ex. 1.1, the set of all word lengths is infinite and, therefore, Σ^* must be infinite as well – see also Method INF2 in Section 0.2.1. □

• **A total order on words: the radix order.** Let $\Sigma = \{a_1, \ldots, a_q\}$ be an alphabet. We can define a total order on the set Σ^* of all words provided that we have agreed on a certain total order '$\prec$' on the alphabet Σ, say

$$a_1 \prec a_2 \prec \cdots \prec a_q.$$

Then, for any two different words w, w' we have that $w \prec w'$ if and only if ***either*** w is shorter than w', that is $|w| < |w'|$; ***or*** the words have the same length and w comes before w' in the lexicographic order determined by the order '$\prec$' in Σ, that is, $|w| = |w'|$ and $\sigma \prec \sigma'$, where σ and σ' are the first corresponding letters on which the two words differ. For example, if $\Sigma = \{a, b\}$ and $a \prec b$, then Σ^* is ordered as follows

$$\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots$$

This order is called radix because if we interpret each letter a_i of the alphabet as a nonzero digit $1, 2, \ldots, q$ then every word is mapped to a unique positive integer in radix $(q+1)$. In particular, if $\Sigma = \{a, b\}$ and we interpret a as 1 and b as 2, then every word in Σ^* represents a ternary (radix 3) number with nonzero digits. For example, bab represents the number $2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23$. This 1-1 mapping of words into the nonnegative integers $\mathbb{N}_0$ shows that the set of all words is countable: $|\Sigma^*| \leq |\mathbb{N}_0|$ – where we have assumed that the empty word maps to zero. Moreover, as the set of all words is infinite, and $\mathbb{N}_0$ is the smallest infinite set, we also have $|\mathbb{N}_0| \leq |\Sigma^*|$. Hence, we have established the following result.

Theorem 1.1. *The set of all words is countable. More specifically,*

$$|\Sigma^*| = |\mathbb{N}_0|$$

Exercises of Section 1.1

Ex. 1.5. Find all solutions of the word equation $x01110101 = 10x110x$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Ex. 1.6. Find all solutions of the word equation $xy = 10$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Solution. We have that $|x| + |y| = 2$, so either both lengths $|x|, |y|$ are 1, or one of them is 2 and the other one is 0. If both lengths are 1 then $x = 1$ and $y = 0$. If one of the words has length zero then the other one must be equal to 10. Hence, the set of solutions is $\{(1, 0), (\lambda, 10), (10, \lambda)\}$, where each pair in the set shows the values of x and y. □

Ex. 1.7. Find all solutions of the word equation $x11y1 = 001111$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Ex. 1.8. Find all solutions of the word equation $x01(x)^R = 10x1$ over the alphabet $\Sigma_2 = \{0, 1\}$. *Hint:* Recall, $(x)^R$ is the reverse of x, so $|(x)^R| = |x|$.

Ex. 1.9. Find all solutions of the word equation $yx0x = 1x(y)^R$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Ex. 1.10. Use Ex. 1.3 as a guide to find all solutions of the word equation $x01 = 10x$ over the alphabet $\Sigma_2 = \{0, 1\}$. *Hint:* First try the simple cases where $|x| = 0$ and $|x| = 1$. Then, distinguish two cases: $|x|$ even, and $|x|$ odd. In each case, work as in Ex. 1.3. The set of solutions should be

$$\{(01)^\ell 1 \mid \ell \in \mathbb{N}_0\}.$$

Ex. 1.11. Find all solutions of the word equation $x011x01 = 10x110x$ over the alphabet $\Sigma_2 = \{0, 1\}$. *Hint:* The first $|x| + 2$ symbols in the left- and right-hand sides of the equation must be equal. This implies that every solution to this equation must also be a solution to the equation in Ex. 1.10

Ex. 1.12. Find all solutions of the word equation $x010 = 0y$ over the alphabet $\Sigma_2 = \{0, 1\}$.

Solution. The words in the two sides of the equation must have the same lengths:

$$|x010| = |0y|, \text{ which implies } |y| = |x| + 2.$$

Now we test a few cases based on the length of x. If $|x| = 0$ then $010 = 0y$ and, therefore, y must be 10. So in this case there is one solution $(x, y) = (\lambda, 10)$. If the length of x is $\ell \geq 1$ we have:

$$\begin{array}{llllllll}
\mathrm{x}(1) & \mathrm{x}(2) & \cdots & \mathrm{x}(\ell) & 0 & 1 & 0 & = \\
0 & \mathrm{y}(1) & \cdots & \mathrm{y}(\ell-1) & \mathrm{y}(\ell) & \mathrm{y}(\ell+1) & \mathrm{y}(\ell+2) &
\end{array}$$

Here the symbols $x(2),\ldots,x(\ell)$ must be equal to $y(1),\ldots,y(\ell-1)$ and no other constraint is required. Hence the set of solutions is

$$\{(\lambda,10)\}\cup\{(0w,w010)\mid w\ \text{ is any word}\}$$

□

1.2 Formal Languages

A (formal) language is any set of words, that is, any subset of Σ^*. A language can be finite or infinite. Here are some examples, of languages over the alphabet $\{a,b\}$:

$$\{aa,ba,bba,ba^{43}\},\ \{awbb\mid |w|\geq 65\},\ \{a,b\}^*-\{\lambda,a,b,aa\}.$$

If L is a language and $w\in L$ then we say that w is an L-word.

• **Do not confuse the concepts**. Be sure to distinguish the different concepts introduced so far. A word/string consists of symbols from the alphabet Σ, unless it is the empty word λ. A language is a set consisting of words, unless it is the empty set $\emptyset$. The expression λ denotes the empty word and is different from the expression $\{\lambda\}$, which is the set consisting of the empty word and, therefore, $\{\lambda\}\neq\emptyset$.

• **Some Language operations**. Assume K,L are two languages. As K,L are sets, we can apply the basic set operations: $K\cup L, K\cap L, K-L$. In addition the following language operations are important.

- Complement: $\overline{L}=\Sigma^*-L$
- Concatenation: $KL=\{uv\mid u\in K,\, v\in L\}$
- Power: $L^n=\{u_1\cdots u_n\mid \text{ each } u_i \text{ is in } L\}$: consists of any n concatenated elements from L. Note that $L^0=\{\lambda\}$
- Kleene Closure: $L^*=L^0\cup L^1\cup L^2\cup\cdots=\bigcup_{i=0}^{\infty}L^i$: consists of zero or more concatenated elements from L. Moreover, we have the operation
$$L^+=L^*-\{\lambda\}.$$

Note that L^* always contains the empty word λ. In particular, we have that

$$\emptyset^* = \{\lambda\}.$$

- Reversal: $L^R = \{(u)^R \mid u \in L\}$

Note that, in general, the concatenation operation is not commutative, that is, $KL \neq LK$. Also, if L is not empty then L^+ is an infinite language.

Ex. 1.13. Consider the languages K, L over the DNA alphabet $\{a, c, g, t\}$:

$$K = \{a^i cct^j \mid i \geq 0, j \geq 1\},\ L = \{g^k a \mid k \geq 0\}.$$

For each of the following languages, give an example of a word *in* (if possible), and of a word *not in* (if possible) the language.

1. K
2. L
3. KL
4. LK
5. K^3
6. LK^*L^2

Solution.
1. In: $cct, aacct, \ldots$. Out: $\lambda, a, cc, \ldots$.
2. In: $a, ga, gga, \ldots$. Out: $g, gg, c, t, \ldots$.
3. In: $ccta, acctga, \ldots$. Out: $\lambda, a, t, \ldots$.
4. In: $acct, aacct, \ldots$. Out: $\lambda, t, a, \ldots$.
5. In: $cctcctcct, \ldots$. Out: $\lambda, a, t, \ldots$.
6. In: $aaa, gacctgaga, \ldots$. Out: $\lambda, t, a, \ldots$. □

- **Property of concatenation**. For any languages K, L, L' we have

$$K(L \cap L') \subseteq KL \cap KL' \quad \text{and} \quad K(L \cup L') = KL \cup KL'.$$

We show the case of the union $\cup$:

$x \in K(L \cup L')$ iff

$x = uv$, for some $u \in K$ and $v \in L \cup L'$ iff

$x = uv$, for some $u \in K$, and $v \in L$ or $v \in L'$ iff

$x = uv$, for some $u \in K$ and $v \in L$, or $u \in K$ and $v \in L'$ iff
$x \in KL$ or $x \in KL'$ iff
$x \in KL \cup KL'$.

Note that there are examples of languages such that $KL \cap KL'$ is not a subset of $K(L \cap L')$.

• **Every language is countable**. As every language is a subset of Σ^*, and Σ^* is a countable set, we get the following corollary.

Corollary 1.2. *Every language is countable.*

• **Method COUNT2: Show that a set X is countable**. Here we use the above corollary as follows. We choose a set of names (= a language) L and explain ***how*** to give a unique name to each element of X. This would show that X is countable because the assignment of unique names is simply a 1-1 mapping of X into a countable set.

Ex. 1.14. To show that $\mathbb{Z}$ is countable using the above method, observe that each integer $x \in \mathbb{Z}$, with $x \neq 0$, can be assigned a *unique* digital word of the form

$$sd_1 \cdots d_m,$$

where s is the sign of x and the d_i's are the digits of x without leading zeros. If x is zero then it gets assigned the word 0. Hence, $\mathbb{Z}$ is countable.

Exercises of Section 1.2

Ex. 1.15. Consider again the languages K, L in Ex. 1.13. Indicate whether each of the following statements is true or not:

1. $acc \in K$
2. $KL \subseteq LK$
3. $aaccta \in (K \cup L)^*$
4. $aacctaa \in (K \cup L)^*$
5. $K \cap L = \emptyset$
6. $K \subseteq L^+K$
7. $K \cap LK = \emptyset$
8. $K \cap LK$ is infinite.

Solution. FALSE, FALSE, TRUE, etc. □

Ex. 1.16. Consider the languages $A = \{0, 10, 11\}$ and $B = \Sigma^* - \{\lambda, 1, 01\}$ over the binary alphabet $\{0, 1\}$. Indicate whether each of the following statements is true or not:

1. $A \subseteq B$
2. $A^* \subseteq B$
3. $A^+ \subseteq B$
4. $A^+\{1\} - \{01\} \subseteq B$
5. $AB \cap BA = \emptyset$

Ex. 1.17. Show that, for any languages K, L, L' we have

$$K(L \cap L') \subseteq KL \cap KL'.$$

Show an example of K, L', L such that $KL \cap KL' \not\subseteq K(L \cap L')$.

Ex. 1.18. Use Method COUNT2 to show that the set of all rational numbers is countable. Use Ex. 1.14 as a guide. Recall, a number is rational if it can be written in the form m/n, where m is an integer and n is a positive integer.

Ex. 1.19. Consider the labeled graph in Fig. 2. Use Method COUNT2 to show that the set of all paths from q_0 to q_1 is countable.

Ex. 1.20. Use Method COUNT2 to show that the set of all finite languages is countable.

Solution. To simplify matters we consider the case of the alphabet $\Sigma = \{a, b\}$ – the general case can be handled analogously. Consider all the finite languages over this alphabet, and define the total order on that alphabet:

$$a \prec b.$$

This order can be extended to the radix order between any two words – see Section 1.1. Thus, for example, we have that $ab \prec abb \prec bba$. Now define the alphabet $\Sigma_\# = \Sigma \cup \{\#\}$, where # is a new symbol. We use Method COUNT 2. To each finite language

$$L = \{w_1, w_2, \ldots, w_n\} \text{ with } w_1 \prec w_2 \prec \cdots \prec w_n,$$

we assign the string $s_L = w_1\#w_2\# \cdots w_n\#$ in $\Sigma_\#^*$, where $s_L = \lambda$ if $L = \emptyset$. Thus, s_L is the concatenation of all words in L using # as a separator, where the words appear in the radix order. One can verify that each finite language L gets assigned a unique string s_L. □

1.3 Undecidable Problems

In this section we establish the fact that some computing problems are not solvable. Informally, a computing problem is a question about any given data, with a well defined set of possible answers, such that the question is to be answered by some computing machine. The data could be an integer, a text, a digital image, a list of integers, etc. In any case, the concept of "word" as defined in Section 1.1 is sufficient for representing data.

• **Decision Problems**. Consider an alphabet Σ. A decision problem is a question about any given word in Σ^* such that there are only two possible answers: {YES, NO}. The given word is called the input word, or input string, and the answer is called the return value. Obviously, a decision problem is specified completely if we describe the set of all YES words, that is, all input words for which the answer is YES. Formally, a decision problem is simply a language L. The language L contains exactly the YES input words. As every language defines a decision problem, we have that

$$\text{the set of ALL decision problems} = 2^{\Sigma^*}.$$

Ex. 1.21. [Pattern-Matching] Consider a word p called the pattern. The problem is to decide whether a given text t contains the pattern p. This problem is important in search engines, for instance, where p is a keyword and the text t is an internet page, and we want to know whether the page is relevant (i.e., contains the keyword). So the decision problem is

Input = a word t (the text)
Return = YES if t contains the word p; NO otherwise.

Mathematically this decision problem is defined by the language

$$L_{\text{pm}} = \{xpy \mid \text{ for some } x, y \in \Sigma^*\}.$$

Indeed, note that for each input t the return value is YES iff $t \in L_{\text{pm}}$.

Ex. 1.22. [Primality testing] Every nonnegative integer can be represented as a binary word, and every binary word w represents a nonnegative integer $\text{num}(w)$, where we assume that $\text{num}(\lambda) = 0$. The primality testing problem is to determine whether a given nonnegative integer is prime, that is, to solve the following decision problem

Input = a binary word w
Return = YES if $\text{num}(w)$ is a prime number; NO otherwise.

Mathematically this decision problem is defined by the language

$$L_{\text{pt}} = \{w \mid \text{num}(w) \text{ is a prime number}\}.$$

• **Uncountably many problems/languages**. We said before that the set of all decision problems is 2^{Σ^*}. As the set Σ^* of all words has cardinality equal to $|\mathbb{N}_0|$ and the set $2^{\mathbb{N}_0}$ is uncountable, it follows that the set of all problems is uncountable as well.

• **Programming Language and Decision Programs**. In Computing Science we solve problems by designing algorithms that are implemented as programs in some programming language. Ultimately, these programs are executed by computing machines, which return the desired answers. A programming language is a set of words, which are the possible programs of the language. Here we ignore the rules defining the syntax of these programs and we assume that a program P acts as a function that takes any word w as input and returns some value $P(w)$, following instructions contained in P. Moreover, we focus on decision programs. A decision program is a program P that, on any given input w, returns YES or NO without producing any output[3], that is, $P(w) \in \{\text{YES, NO}\}$.

• **Decidable/Solvable Problems**. A problem L is called decidable, or computable, or solvable, if there is a (decision) program P that solves L, that is, for any input word w,

$$P(w) = \begin{cases} \text{YES}, & \text{if } w \in L; \\ \text{NO}, & \text{if } w \notin L. \end{cases}$$

Equivalently, we can write $L = \{w \in \Sigma^* \mid P(w) = \text{YES}\}$. We also say that the program P decides (or computes, or solves) the problem L. In anticipation of the types of languages that we shall see later,

[3] We assume that producing any output, in the sense of displaying externally a word, does not contribute to the decision that needs to be made.

we note that when the decidable problem L is viewed as a language then it is called a recursive language.

• **Undecidable/unsolvable problems**. Consider any programming language. The big question that arises now is whether there are undecidable problems, that is, problems L for which no programs P exist deciding L.

Corollary 1.3. *There exists no programming language that computes all problems. In particular, there are undecidable problems with respect to any programming language.*

The above result can be proved using Theorems 0.1 and 1.1. The proof is based on a simple counting argument: being a language, the set of programs is countable, whereas the set of decision problems is equal to 2^{Σ^*}, which is uncountable; that is, there are more problems than programs! Although this result is fundamental, it gives us no concrete examples of undecidable problems. One of the objectives of formal language theory is to provide the tools for addressing this question. In particular, we shall see later several problems that are undecidable.

• **Undescribable Entities**. Using again a counting argument, it turns out that there are problems that are not even describable! More specifically, suppose we want to describe formally entities of interest (problems, numbers, languages, etc) using some language whose words are called descriptions. Let X be the set of entities. A description method for X is an onto function

$$\mathcal{L} : D \to X$$

such that D is a language, which is the description language of the method. As $\mathcal{L}$ is onto[4], every entity in X has at least one description in D. The next statement concerns the case where X is the set of all decision problems. Then, X is uncountable and there can be no language D and description method from D onto X.

Corollary 1.4. *There exists no language that describes all problems. In other words, there are undescribable problems with respect to any language.*

[4] This implies that $|X| \leq |D|$.

Exercises of Section 1.3

Ex. 1.23. Let EP be the set of paragraphs that can be written in the English language. Is the set EP countable? Explain why there are real numbers that cannot be described with any English paragraph.

Ex. 1.24. We say that a real number x is computable if there is a program P_x that takes as input a positive integer n and returns the integral part of x followed by the first n decimal digits of x. Explain why there are real numbers that are not computable.

1.4 Introduction to Context-free Grammars

An important method for describing certain languages is the method of Context-free Grammars (CFGs). These objects involve two alphabets:

$\Sigma =$ the alphabet of ordinary symbols (terminal alphabet).
$V =$ the alphabet of variables or nonterminals.

A variable is a capital English letter $(A, B, C, \ldots)$, possibly with a numeric subscript $(A_1, B_5, \ldots)$.

• **Context-free Rule**. This is an expression of the form $\beta \rightarrow w$, where β is a variable and w is a word in $(\Sigma \cup V)^*$. When a string in $(\Sigma \cup V)^*$ is of the form $x\beta y$ we can *apply the rule* $\beta \rightarrow w$ by replacing the variable β with w:

$$x\beta y \Rightarrow xwy.$$

For example, we can apply the rules $X \rightarrow 01$ and $Y \rightarrow 0$ to the string $1XX0Y$ as follows:

$$1XX0Y \Rightarrow 1X010Y \Rightarrow 1X0100.$$

• **Context-free Grammar** G. Is defined by an expression of the form "$S_G; \text{rules}(G)$" such that S_G is a special variable (the start variable), and rules(G) is a list of context-free rules

$$\text{rules}(G) = \beta_1 \rightarrow w_1;\ \beta_2 \rightarrow w_2; \ldots ; \beta_n \rightarrow w_n.$$

When the particular grammar G is understood we can omit the G from S_G and simply write S for the start symbol of G. When two or more rules have the same left-hand side, as in "$\beta \to w_1; \beta \to w_2; \ldots$", we can group them together using the shorthand notation

$$\beta \to w_1 | w_2 | \cdots,$$

where '|' means 'or', that is, β can be replaced with w_1, or w_2, or...

• **Language generated (=described) by Context-free Grammar** G. The language generated by G is

$$\mathcal{L}(G) = \{u \in \Sigma^* \mid S_G \Rightarrow z_1 \Rightarrow \cdots \Rightarrow z_n \Rightarrow u, \text{ for some } n \geq 0\},$$

that is, $\mathcal{L}(G)$ consists of all terminal words u that can be ***generated*** when we start with the start variable S_G and apply a sequence of one or more rules, one at a time. Note that, in general, the word u can be generated in many different ways.

Ex. 1.25. [Nonnegative integers] The grammar

$$\text{NN} \ = \ \text{"}S; S \to SS; S \to 0|1|\cdots|9\text{"}$$

generates all nonnegative integers, that is, $\mathcal{L}(\text{NN}) = \{0, 1, , \ldots, 9\}^+$. For example, here is one way to generate the number 638

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow 6SS \Rightarrow 63S \Rightarrow 638.$$

Ex. 1.26. [Arithmetic Expressions] We want to represent the language of all valid arithmetic expressions involving nonnegative integers, the operators +, *, and parentheses. Here is a context-free grammar

$$\text{AE} \ = \ \text{"}S;\ S \to (S) \mid S + S \mid S * S \mid S_{\text{NN}};\ \text{rules(NN)"}.$$

For example, the expression $3 * (638 + 2)$ is in $\mathcal{L}(\text{AE})$ because it can be generated by the grammar AE as follows

$$S \Rightarrow S{*}S \Rightarrow S{*}(S) \Rightarrow S{*}(S{+}S) \Rightarrow S_{\text{NN}}{*}(S{+}S) \Rightarrow \cdots \Rightarrow 3{*}(638{+}2)$$

• **The class of context-free languages**. Let CFG be the set of all context free grammars with respect to a certain terminal alphabet.

Each of these grammars G describes a language $\mathcal{L}(G)$. Taking all these languages together defines the class of context-free languages:

$$\mathcal{L}(\text{CFG}) = \{\mathcal{L}(G) \mid G \text{ is in CFG}\}.$$

• **Do not confuse the concepts**. A word (or string) is a finite sequence of zero or more symbols. A language is a set of zero or more words; it can be finite or infinite. A class of languages is a set whose elements are languages. A context-free grammar G is an expression that describes a language $\mathcal{L}(G)$.

Exercises of Section 1.4

Ex. 1.27. Explain why every finite language is context-free.

Solution. Let $F = \{w_1, \dots, w_n\}$ be any finite language. We need a context-free grammar that generates this language. Here it is:

$$S;\ S \to w_1 \mid w_2 \mid \cdots \mid w_n.$$

In case F is empty (thus, $n = 0$ above) the grammar has no rules. □

Ex. 1.28. Let $\Sigma_2 = \{0, 1\}$. Write grammars for generating each of the following languages.

1. Σ_2^*.
2. $\Sigma_2^*\{010\}\Sigma_2^* = \{u010v \mid u, v \in \Sigma_2^*\}$.
3. $\{0^n1^m \mid n \geq 2, m \geq 0\}$.
4. $\{0^n1^n \mid n \in \mathbb{N}_0\}$.
5. $\{w \mid w \text{ contains no } 111\} = \Sigma_2^* - \Sigma_2^*\{111\}\Sigma_2^*$.

Solution.

1. ALL = “S; $S \to \lambda \mid 0 \mid 1 \mid SS$”
2. S; $S \to S_{\text{ALL}}010S_{\text{ALL}}$; rules(ALL)
3. S; $S \to 00ZN$; $Z \to \lambda \mid 0Z$; $N \to \lambda \mid 1N$

□

Ex. 1.29. Modify the grammar in Ex. 1.25 to generate all integers with an optional sign. For example, -13, 27, +27, 0, -32, etc.

Ex. 1.30. Write a grammar to generate all decimal numbers with an optional sign and an optional decimal point. For example, -13, 27.89, +27.09, 0, -32.11, .57, etc. If the decimal point is present then there must be at least one digit following the decimal point.

Ex. 1.31. For each of the following decision problems, describe the language corresponding to that problem. Use a context-free grammar to describe the language. Recall the language that corresponds to a decision problem is the set of all input words for which the answer is YES. Assume that the alphabet is $\{a, b\}$.

1. Input: a word w.
   Return: YES, if w contains only a's (if any); NO, otherwise.

2. Input: a word w.
   Return: YES, if w contains at least two b's; NO, otherwise.

3. Input: a word w.
   Return: YES, if w contains the same number of a's and b's; NO, otherwise.

Solution. The grammar for the language of the second problem is

$$S;\ S \to TbTbT;\ T \to \lambda \mid aT \mid bT.$$

The grammar for the language of the third problem is

$$S;\ S \to \lambda \mid aSb \mid bSa \mid SS.$$

□

We note that proving that a certain grammar generates a given language might be a difficult task. A rigorous proof of correctness for the grammar generating the third language in the above example can be found in [23].

2 PROPOSITIONAL LOGIC (Boolean Algebra)

Propositional Logic is the simplest logic system and involves the following:

- *A formal language* whose words are called propositions (or Boolean expressions). A proposition is intended to express a statement about a real world situation.
- *Semantics*: a method that assigns to every proposition a value in $\{\mathbf{T}, \mathbf{F}\}$, which is called a truth value. One way to realize this method is by using truth tables. Semantics tells us when a proposition can be logically derived from another proposition.
- *Formal deduction (inference rules)*: This is a syntactic method for deducing new true propositions from a set of existing ones. It turns out that the concepts of deduction by inference rules and by semantics are equivalent.

2.1 The language of Propositions (syntax)

The language of propositions involves constants, (logical) variables, parentheses, and the following special *connective* symbols (or operators):

$\neg$: not, negation
$\wedge$: and, conjunction
$\vee$: or, disjunction
$\rightarrow$: implication, conditional, only if
$\leftrightarrow$: bi-conditional, iff

Definition 2.1. The language Prop of propositions (or propositional formulas) is generated by the following grammar:

$$\begin{array}{lll}
P; & & \text{(the start symbol)} \\
P & \rightarrow & \mathbf{T} \mid \mathbf{F}; \qquad \text{(constants: true, false)} \\
P & \rightarrow & L \mid LL; \qquad \text{(logical variables)} \\
L & \rightarrow & a \mid b \mid \cdots \mid y \mid z; \\
P & \rightarrow & (P \wedge P) \mid (P \vee P) \mid (\neg P) \mid \qquad \text{(composite propositions)} \\
 & & (P \rightarrow P) \mid (P \leftrightarrow P)
\end{array}$$

If more variable symbols are needed, we shall use subscripts: $p_1, q_5, \ldots$. We shall use $\textsc{Var}$ to denote the set of all variables.

Ex. 2.1. Here are a few propositions.

$$\mathbf{T},\ \mathbf{F},\ p,\ q,\ r,\ ar,\ be,\ (p \vee r),\ (p \wedge (\neg q)),\ ((\neg ar) \to (be \leftrightarrow ch)).$$

The following strings are not propositions

$$p(q),\ ar \vee \wedge \mathbf{T}.$$

• **Omitting parentheses**. As in the case of algebraic expressions we shall omit parentheses to simplify propositions. In doing so, we shall use the following precedence of operations:

$$\neg,\ \{\wedge, \vee\},\ \{\to, \leftrightarrow\},$$

where the sets indicate operations of equal precedence. Moreover, we shall use the left to right order for multiple operations of the same precedence. For example, $\neg p \wedge q \wedge r \to s \vee p$ is a shorthand for

$$((((\neg p) \wedge q) \wedge r) \to (s \vee p)).$$

2.2 Semantics (meaning) of Propositions

Propositions are syntactic objects that can be used to represent statements about the real world. Depending on the context, these statements could be true or false. In particular, depending on the values assigned to the variables involved, a proposition would evaluate to **T** (true), or **F** (false).

• **Valuation**. Is a function

$$val\ : \textsc{Var} \to \{\mathbf{T}, \mathbf{F}\}$$

assigning truth values to the variables. In practice the particular assignment of values depends on the meaning we give to the variables with respect to a real world application – see examples further below.

• **Evaluating propositions**. A valuation *val* is extended from Var to Prop as follows:

$$\begin{array}{rcl} val\,(\mathbf{T}) & = & \mathbf{T}; \\ val\,(\mathbf{F}) & = & \mathbf{F}; \\ val\,(\neg\beta) & = & \mathbf{T} \text{ iff } val\,(\beta) = \mathbf{F}; \\ val\,(\alpha \wedge \beta) & = & \mathbf{T} \text{ iff } val\,(\alpha) = val\,(\beta) = \mathbf{T}; \\ val\,(\alpha \vee \beta) & = & \mathbf{F} \text{ iff } val\,(\alpha) = val\,(\beta) = \mathbf{F}; \\ val\,(\alpha \to \beta) & = & \mathbf{F} \text{ iff } val\,(\alpha) = \mathbf{T} \text{ and } val\,(\beta) = \mathbf{F}; \\ val\,(\alpha \leftrightarrow \beta) & = & \mathbf{T} \text{ iff } val\,(\alpha) = val\,(\beta). \end{array}$$

To evaluate a proposition γ, first we look at the structure of γ in terms of the grammar rules one can use to generate it – see Definition 2.1. For example, let

$$\gamma = (p \vee q) \wedge r.$$

One can start generating γ using the grammar rule $P \to P \wedge P$. So γ has the structure $\alpha \wedge \beta$, where $\alpha = p \vee q$ and $\beta = r$. Thus $val\,(\gamma) = \mathbf{T}$ iff $val\,(p \vee q) = \mathbf{T}$ and $val\,(r) = \mathbf{T}$. Then, $val\,(p \vee q) = \mathbf{T}$ iff at least one of $val\,(p)$, $val\,(q)$ is true. Thus, we conclude that $val\,(\gamma) = \mathbf{T}$ iff $val\,(p) = val\,(r) = \mathbf{T}$ and/or $val\,(q) = val\,(r) = \mathbf{T}$.

• **Giving meaning to variables**. A proposition (of the language Prop) is intended to express a statement about some real world situation. This is achieved by giving meaning to the variables involved in the proposition. We shall write

$$\alpha \ : \ \text{“statement”}$$

to indicate that α means whatever the “statement” says. Clearly the meaning we give to our variables determines a certain valuation *val* , that is, the assignment of truth values to these variables.

Ex. 2.2. Consider the following variables

$$p: \ \text{“it's cold outside”} \quad q: \ \text{“I'm wearing a coat”}.$$

This meaning determines a valuation *val* and, therefore, also the value of $p \wedge q$, whose meaning is “it's cold outside and I'm wearing a coat". Note that one could give a different meaning to p, q. For example,

$$p: \text{"I like cookies"} \quad q: \text{"I like milk"}.$$

This meaning corresponds to a new valuation val'. Obviously, it could be the case that $val\,(p \wedge q) \neq val'(p \wedge q)$.

One could still wonder about the actual values of p and q. The desired meaning refers to a particular person and, possibly, to a particular time. Thus, with the first meaning referring to the author of this textbook at the time of writing this sentence, we note that it neither was cold outside nor was he wearing a coat and, therefore, $val\,(p) = val\,(q) = \mathbf{F}$. On the other hand, with the second meaning referring to the same person, we have that $val'(p) = val'(q) = \mathbf{T}$.

Ex. 2.3. [Program Specification] Read numbers and compute their sum. Read up to 100 numbers, or until a zero is read.
Logical specification: Define a proposition **stop** that will be used as a stopping condition for the loop that is required to solve the above problem. The loop will have the form

```
do {
      statements
} while (¬ stop);
```

Consider the propositional variables
p : "100 numbers were read" and q : "a zero was read"
Then **stop** $= (p \vee q)$ and the loop becomes

```
do {
      read a number n
      add n to sum
} while (¬(p ∨ q));
```

Implementation in Java/C++: Use the programming variable `count` to count the numbers read and the programming variable `n` for the last number read. Then p : "`(count==100)`" and q : "`(n==0)`", and the implementation is as follows:

```
sum = 0;    count = 0;
do {
    cin >> n;
```

```
        count = count + 1;
        sum = sum + n;
    } while (!  ((count==100) || (n==0)));
```

Ex. 2.4. [Implication] Let p : "it's raining" and q : "there are clouds in the sky". Each of these propositional variables can be true or false depending on the time and place they refer to. In reality we know that, always, if it's raining then there are clouds in the sky. Hence, the proposition $p \to q$ is true. Note that the proposition *$p \to q$ is true even if p is false* (that is, it is not raining); in this case the truth value of q is irrelevant and could be either true or false.

Ex. 2.5. [Bi-conditional] Let p_x : "the number x is greater than 5" and q_x : "the number $x+1$ is greater than 6". Depending on the actual value of x, each of these variables can be true or false, but we know that they always have the same truth value. Hence, the proposition $p_x \leftrightarrow q_x$ is true. Now let r_x : "the number x is greater than 7". If $x = 6$, then p_6 and r_6 have different truth values, so $(p_6 \leftrightarrow r_6)$ would be false.

2.3 Truth tables, Equivalence, Consequence, Model

The possible truth values of a proposition can be determined based on the combinations of values of the variables occurring in the proposition. To this end, we construct a truth table as follows:

1. In the first row, write the different *variables* followed by the *propositions* that will be evaluated.

2. Then, in the column below the *variables*, list all the possible combinations of truth values for these variables (there should be one ***row*** for each combination of values). Note that if the *propositions* involve n different variables then there are 2^n possible combinations of truth values.

3. Then, for each row, compute the truth values of the propositions.

p	q	$\neg p$	$(\neg p) \vee q$	$p \to q$	$(\neg p) \wedge (p \to q)$
T	**T**	**F**	**T**	**T**	**F**
T	**F**	**F**	**F**	**F**	**F**
F	**T**	**T**	**T**	**T**	**T**
F	**F**	**T**	**T**	**T**	**T**

Note that every row in a truth table corresponds to a ***valuation*** function *val* . For example, the second row of the above table corresponds to the valuation *val* such that

$$val\,(p) = \mathbf{T}, \quad val\,(q) = \mathbf{F}.$$

• **Equivalence**. Two propositions α, β are called (logically) equivalent if they always have the same truth values for every combination of values for their variables. Another way to put it is that $val\,(\alpha) = val\,(\beta)$ for ***every*** possible valuation *val* . In this case we write

$$\alpha \equiv \beta.$$

For example, in the above truth table we see that the propositions $p \to q$ and $(\neg p) \vee q$ are equivalent.

• **Tautology, satisfiability, model**. A proposition α is called a tautology, if

$$\alpha \equiv \mathbf{T},$$

that is, all values in the truth column of α are **T**. In other words, $val\,(\alpha) = \mathbf{T}$ for all valuations *val* . A proposition is called a contradiction if

$$\alpha \equiv \mathbf{F},$$

that is, all values in the truth column of α are **F**. A set of propositions A is satisfiable if there is a valuation *val* that makes every proposition in A true, that is, $val\,(\alpha) = \mathbf{T}$ for all $\alpha \in A$. In this case, we say that *val* is a model of A.

• **Logic puzzles**. In a logic puzzle we are given a few statements about a certain event. The problem is to find out the truth about the event in question. In such cases we attempt to express the problem in propositional logic as follows: ***First***, identify the propositional variables and their meaning based on the given statements. ***Then***, using logical connectives, combine the variables, to form a set of propositions corresponding to the given statements. ***Then***, compute a ***model*** for the set of propositions, and use this model to find the desired answers.

Ex. 2.6. Police have investigated the murder case and are convinced of the following statements. At least one of Arthur, Betty, Charles is guilty. In particular, at least one of Betty and Charles must be guilty, and if Betty is guilty then also Charles must be guilty. Moreover, if Arthur is guilty then neither of Betty and Charles can be guilty.

Solution. Consider the following variables

ar : "Arthur is guilty"
be : "Betty is guilty"
ch : "Charles is guilty"

Using the statements in the problem we need to find a model for the following set of propositions:

$$\{ar \vee be \vee ch, be \vee ch, be \to ch, ar \to \neg be \wedge \neg ch\}.$$

If we use the method of truth tables we will find that there are exactly two models for the above set:

Model val_1 with $val_1(ar) = \mathbf{F}$ and $val_1(be) = val_1(ch) = \mathbf{T}$.
Model val_2 with $val_2(ar) = val_2(be) = \mathbf{F}$ and $val_2(ch) = \mathbf{T}$.

In both models, we have that Arthur is not guilty and that Charles is guilty. Betty is guilty in val_1 and not guilty in val_2, but we have no information which model to choose. So we only arrest Charles. □

• **(Logical) Consequence**. A proposition β is said to be a (logical) consequence of α, if the value of β is true whenever the value of α is true, that is, $val(\alpha) = \mathbf{T}$ implies $val(\beta) = \mathbf{T}$, for all possible valuations val. In this case we write

$$\alpha \models \beta.$$

For example p is a (logical) consequence of $p \wedge q$. This can be seen if we construct the truth table for $p \wedge q$. Moreover, this makes sense informally, when we note that if "p and q are both true" then p alone must be true. Another example of consequence is the following

$$\alpha \wedge (\alpha \rightarrow \beta) \models \beta.$$

The concept of logical consequence is extended to sets of propositions. In particular, if $A = \{\alpha_1, \ldots, \alpha_n\}$ is a set of propositions, the notation

$$A \models \beta$$

means that β is a consequence of $\alpha_1 \wedge \cdots \wedge \alpha_n$. Note that when α and β are logical consequences of each other then they are equivalent, and vice versa; that is,

$$\alpha \equiv \beta \quad \text{iff} \quad (\alpha \models \beta \;\text{ and }\; \beta \models \alpha).$$

Ex. 2.7. [*Argument validation*] We are given a set of statements, called premises, about some topic, and a particular statement, called the conclusion, and we want to know whether the conclusion is correct, that is, it follows logically from the premisses. Using the tools of this section, we can translate the premisses into a set A of propositions, and the conclusion into a proposition β, and then test whether

$$A \models \beta.$$

To test the above, we make a truth table that includes all propositions involved, and we look for valuations (rows) in which all propositions of A are true. If β is true in all of those valuations then the conclusion is indeed correct. If there is at least one valuation (row) in which all propositions in A are true and β is false, then the conclusion is incorrect.

2.4 Laws of Propositional logic

The concept of equivalence, $\equiv$, allows one to rewrite a proposition in a different, but equivalent, form. Here we look at some equivalences that we call laws of propositional logic. This is similar to the case of algebraic laws that allow one to manipulate algebraic expressions. For example, the analog of the algebraic expression $x \cdot (y + z) = x \cdot y + x \cdot z$ in logic is

$$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma).$$

The above law says that whenever $\alpha \wedge (\beta \vee \gamma)$ appears in some proposition δ and we replace it with $(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$ then the proposition δ will have the same truth value. This situation is described in the next theorem.

Theorem 2.1. ***(Replacability)*** *Let α, β be two equivalent propositions, namely $\alpha \equiv \beta$. For every proposition δ, if δ' results by replacing some (not necessarily all) occurrences of α in δ with β, then $\delta' \equiv \delta$.*

Proof. We use the method of ***structural induction*** on δ, that is, we prove the statement for the case where δ is simple (**T**, **F**, or a variable) and then for the case where it is composite. So if δ is simple then α must be the only component of δ, that is, $\delta = \alpha$. Then, replacing α with β gives $\delta' = \beta$ and, therefore, $\delta \equiv \delta'$, as required.

Now suppose that δ is composite, that is, of one of the forms $\delta_1 \wedge \delta_2$, $\delta_1 \vee \delta_2$, $\delta_1 \to \delta_2$, $\delta_1 \leftrightarrow \delta_2$, $\neg\delta_1$, such that the statement holds for the simpler components δ_1, δ_2 of δ. If we substitute β for some occurrences of α in δ, then these substitutions will occur in δ_1 and/or δ_2 and we get δ' of the form $\delta_1' \wedge \delta_2'$, $\delta_1' \vee \delta_2'$, etc. By the induction hypothesis, $\delta_1 \equiv \delta_1'$ and $\delta_2 \equiv \delta_2'$. Then one can verify that also $\delta \equiv \delta'$ in all cases. For example, $\mathit{val}\,(\delta_1 \wedge \delta_2) = \mathbf{T}$ iff $\mathit{val}\,(\delta_1) = \mathit{val}\,(\delta_2) = \mathbf{T}$ iff $\mathit{val}\,(\delta_1') = \mathit{val}\,(\delta_1') = \mathbf{T}$ iff $\mathit{val}\,(\delta_1' \wedge \delta_2') = \mathbf{T}$. □

Here are a few more laws:

Negation:
$\neg(\neg\alpha) \equiv \alpha$

Single proposition:
$\alpha \vee (\neg\alpha) \equiv \mathbf{T}$ (excluded middle)
$\alpha \wedge (\neg\alpha) \equiv \mathbf{F}$ (contradiction)
$\alpha \wedge \alpha \equiv \alpha \vee \alpha \equiv \alpha$ (idempotence of $\wedge$ and $\vee$)

Constants:
$\alpha \vee \mathbf{T} \equiv \mathbf{T}$
$\alpha \vee \mathbf{F} \equiv \alpha$
$\alpha \wedge \mathbf{T} \equiv \alpha$
$\alpha \wedge \mathbf{F} \equiv \mathbf{F}$.

Commutativity:
$\alpha \wedge \beta \equiv \beta \wedge \alpha$
$\alpha \vee \beta \equiv \beta \vee \alpha$.

Associativity:
$\alpha \wedge (\beta \wedge \gamma) \equiv (\alpha \wedge \beta) \wedge \gamma$
$\alpha \vee (\beta \vee \gamma) \equiv (\alpha \vee \beta) \vee \gamma$

Distributivity:
$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$.

DeMorgan's laws:
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha) \vee (\neg\beta)$
$\neg(\alpha \vee \beta) \equiv (\neg\alpha) \wedge (\neg\beta)$.

Conditionals:
$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$
$\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$
$\alpha \leftrightarrow \beta \equiv (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$
$\alpha \rightarrow \beta \equiv \neg\beta \rightarrow \neg\alpha$ (contrapositive law)

Subsumption:
$(\alpha \vee \beta) \wedge \alpha \equiv \alpha$.

Exercises of Sections 2.2–2.4

Ex. 2.8. Is the language Prop of all propositions finite, or infinite? Is it countable?

Ex. 2.9. Translate each of the following *natural language* sentences into propositions of Prop, the *propositional logic language*, using appropriately the indicated variables. State ***explicitly*** the meaning of each variable.

1. Either Harper or Ignatieff wins the election, ***but*** not both. (ha, ig)

2. Oscar will not date Anna ***unless*** he breaks up with Mona first. (da, br)

3. Adele and Virginia will go to the beach ***only if*** they can rent a car. (ad, vi, re)

4. If a number is a positive integer, then it is divisible by 11 ***iff*** the alternating sum of its digits is divisible by 11. (po, di, su)

5. A ***necessary condition*** for a number to be prime is that it is ***neither*** divisible by 5 ***nor*** is it an even integer larger than 2. (pr, di, ev)

6. A ***sufficient condition*** for the continuation of Martha's career is that her shares do not lose value and that her jail cell is fashionably decorated. (ca, lo, de)

7. Nobody can run so fast, unless they are trained. Well, Theodore is not trained. (ru, tr)

8. If the floor is wet and you bounce the ball on the floor, then the ball gets wet. (fl, bb, ba)

9. My watch works only if its battery works. (ww, bw)

10. At least one of Helen, Bert and Tom likes chocolate, but not all of them. (he, be, to)

Solution.

1. $(ha \vee ig) \wedge \neg(ha \wedge ig)$, where $ha :$ "Harper wins the election", $ig :$ "Ignatieff wins the election".

2. $da \to br$, where $br :$ "Oscar breaks up with Mona", $da :$ "Oscar will date Anna".

3. $ad \wedge vi \to re$, where $ad :$ "Adele will go to the beach", $vi :$ "Virginia will go to the beach", $re :$ "Adele and Virginia can rent a car".

4. $po \to (di \leftrightarrow su)$, where $po :$ "the number is a positive integer", $di :$ "the number is divisible by 11", $su :$ "the alternating sum of the number's digits is divisible by 11".

5. $pr \to (\neg di \wedge \neg ev)$, where $pr :$ "the number is prime", $di :$ "the number is divisible by 5", $ev :$ "the number is even and larger than 2".

6. $\neg lo \wedge de \rightarrow ca$, where ca : "Martha's career will continue", lo : "Martha's shares lose value", de : "Martha's jail cell is fashionably decorated".

7. $(ru \rightarrow tr) \wedge \neg tr$, where ru: "Theodore can run so fast", tr: "Theodore is trained". We note here that $(ru \rightarrow tr)$ has a more restricted meaning than the given phrase "Nobody can run so fast, unless they are trained". However, we have to specialize "nobody" to "Theodore" as we are forced to use only two variables.

□

Ex. 2.10. Consider the following propositional variables:

p: "last number read is zero"
r: "the last two numbers read are equal"

Write a logical specification for the problem of reading numbers until a zero is read or until the last two numbers read are equal. Then, give an implementation in Java/C++ – show clearly the implementation of p, r.

Solution. Here is the logical specification:

```
do {
    read a number
} while ($\neg(p \vee r)$);
```

Here is the implementation:

```
curr = 0;
do {
    prev = curr;
    cin >> curr;
} while (!( (curr==0) || (prev==curr) ));
```

Note that p : "`(curr==0)`" and r : "`(prev==curr)`". Recall that programming languages evaluate the part r of an expression $(p||r)$ iff the expression p is false. Hence, in this exercise, `(prev==curr)` will be evaluated iff `curr` is nonzero. Moreover, if `curr` is nonzero and also the first number read, then `prev` must be zero, so the loop will not terminate (as required). □

Ex. 2.11. Consider the propositional variables

d : "at least one digit has been read before the char. last read"
nd : "a non-digit character was last read"
ns : "a non-space character was last read"

Use these propositional variables to write a logical specification (in the form of a do-while loop) for the following problem: Read the digits of a number skipping any space characters in the beginning. Stop reading, if a non-digit character is read after some digit was read, or a non-digit and non-space character is read. Then implement your specification in Java/C++ (show clearly the implementation of d, nd, ns). You can assume that there is a function `isdigit(ch)` that returns true/false depending on whether the character `ch` is a digit.

Ex. 2.12. Arthur, Betty, Charles and Dorothy are the only murder suspects. Their testimonies are as follows:

- Arthur said 'if Betty is guilty then also Dorothy is.'
- Betty said 'Arthur is guilty but Dorothy is not'
- Charles said 'I am not a liar, but Arthur and/or Dorothy are'
- Dorothy said 'if Arthur is not guilty then Charles is guilty'

We want to find out all guilty suspects if possible. We use the following assumption:

- A person tells the truth iff the person is not guilty.

Solution. We use the following propositional variables:

ar : "Arthur is not guilty" : "Arthur tells the truth"
be : "Betty is not guilty" : "Betty tells the truth"
ch : "Charles is not guilty" : "Charles tells the truth"
do : "Dorothy is not guilty" : "Dorothy tells the truth"

We have to find the models of the following set of propositions

$\{$
$\beta_1 = ar \leftrightarrow (\neg be \rightarrow \neg do)$,
$\beta_2 = be \leftrightarrow \neg ar \wedge do$,
$\beta_3 = ch \leftrightarrow ch \wedge (\neg ar \vee \neg do)$,
$\beta_4 = do \leftrightarrow (ar \rightarrow \neg ch)$
$\}$.

We use the truth table in the figure. As we are looking for rows in which ***all*** values of the four propositions are true, we ignore any rows

do	ar	be	ch	β_1	β_2	β_3	β_4
T	**T**	**T**	**T**	**T**	**F**		
T	**T**	**T**	**F**	**T**	**F**		
T	**T**	**F**	**T**	**F**			
T	**T**	**F**	**F**	**F**			
T	**F**	**T**	**T**	**F**			
T	**F**	**T**	**F**	**F**			
T	**F**	**F**	**T**	**T**	**F**		
T	**F**	**F**	**F**	**T**	**F**		
F	**T**	**T**	**T**	**T**	**F**		
F	**T**	**T**	**F**	**T**	**F**		
F	**T**	**F**	**T**	**T**	**T**	**T**	**T**
F	**T**	**F**	**F**	**T**	**T**	**T**	**F**
F	**F**	**T**	**T**	**F**			
F	**F**	**T**	**F**	**F**			
F	**F**	**F**	**T**	**F**			
F	**F**	**F**	**F**	**F**			

Figure 3: Truth table for Ex. 2.12

in which the value **F** appears. We see there is exactly one model in which only Arthur and Charles are guilty. □

Ex. 2.13. This is the same as the previous exercise, except that the assumptions now are as follows:

1. If a person is not guilty then the person tells the truth (the converse might ***not*** hold here).

2. Betty is not guilty.

Ex. 2.14. Read Ex. 2.7 and test whether the following argument is correct. ***Premisses:*** If ghosts exist, they make their presence known. If a ghost makes its presence known, or you dream of a ghost, then you get scared. ***Conclusion:*** If ghosts exist, then you get scared.

Note: When you translate the above statements into propositions use only four logic variables: ex, kn, dr, sc.

2.5 Formal Deduction: Inference rules, Proofs

Here we look at the concept of formal deduction that allows us to derive (deduce/prove) a new proposition (a theorem) α from a set Γ of given propositions, called assumptions. This derivation process uses the available inference rules. An inference rule,

$$A \vdash \beta,$$

consists of a finite set of propositions A, called the premises, and a proposition β called the conclusion. This rule is also written by listing the elements of A one per line, then a horizontal line, and then the conclusion β. We can now define the concept "(formal) proof of $\Gamma \vdash \alpha$." This is a finite sequence of propositions

α_1
α_2
$\ldots$
α_n

such that (i) $\alpha_n = \alpha$, that is, the last proposition is the theorem that we prove, and (ii) every α_k, with $1 \leq k \leq n$, is either an assumption in Γ, or the conclusion α_k of some inference rule $A \vdash \alpha_k$, where

$A \subseteq \{\alpha_1, \ldots, \alpha_{k-1}\}$, that is, each element of A is one of the earlier propositions α_i with $i < k$. When we show that $\Gamma \vdash \alpha$ we say that α is deducible or provable from Γ.

- **The list of inference rules**.

IR01 Modus ponens: $\alpha \to \beta, \alpha \vdash \beta$

IR02 $\leftrightarrow$ introduction: $\alpha \to \beta, \beta \to \alpha \vdash \alpha \leftrightarrow \beta$

IR03 $\leftrightarrow$ elimination: $\alpha \leftrightarrow \beta, \beta \vdash \alpha$ and $\alpha \leftrightarrow \beta, \alpha \vdash \beta$

IR04 Case analysis: $\alpha \vee \beta, \alpha \to \gamma, \beta \to \gamma \vdash \gamma$

IR05 $\wedge$ introduction: $\alpha, \beta \vdash \alpha \wedge \beta$

IR06 $\wedge$ elimination: $\alpha \wedge \beta \vdash \alpha$ and $\alpha \wedge \beta \vdash \beta$

IR07 $\vee$ introduction: $\alpha \vdash \alpha \vee \beta$ and $\alpha \vdash \beta \vee \alpha$

IR08 Contradiction: $\alpha, \neg\alpha \vdash \mathbf{F}$

IR09 $\neg$ introduction: $\alpha \to \mathbf{F} \vdash \neg\alpha$

IR10 $\neg$ elimination: $\neg\alpha \to \mathbf{F} \vdash \alpha$

IR11 Tautology: $\vdash \mathbf{T}$

IR12 Deduction rule: $\vdash \alpha \to \beta$, applicable **if** there is already a proof of "$\Gamma, \alpha \vdash \beta$".

The Deduction rule is used when we want to prove a proposition of the form

$$\alpha \to \beta.$$

In this case, we do a separate proof of β by assuming α (in addition to any propositions in Γ). This proof is normally included as a subproof of the original proof. This is indicated by a line containing the word "SUB:" and then the sequence of propositions (starting with α and ending with β) that constitute the subproof – see the examples below.

Ex. 2.15. [Modus tollens] Give a formal proof of Modus tollens:

$$\alpha \to \beta, \neg\beta \vdash \neg\alpha$$

Solution. Strategy: To prove $\neg\alpha$ we shall use rule IR09, which requires to prove $\alpha \to \mathbf{F}$, which requires to use IR12 (the deduction rule), which requires a subproof of $\alpha \vdash \mathbf{F}$.

1.	$\alpha \to \beta$		given
2.	$\neg\beta$		given
3.	SUB:	α	subproof for $\alpha \vdash \mathbf{F}$
4.		β	Modus ponens: lines 1,3
5.	END	$\mathbf{F}$	Contradiction rule: lines 2,4
6.	$\alpha \to \mathbf{F}$		Deduction rule: line 3
7.	$\neg\alpha$		$\neg$ Introduction: line 6

□

Ex. 2.16. Give a formal proof of $\alpha \vdash \alpha$.

Solution.
1. α given
This is the proof; the given assumption is already the conclusion. □

Ex. 2.17. Give a formal proof of $\vdash \alpha \to \alpha$. You can assume $\alpha \vdash \alpha$.

Solution.
1. $\alpha \to \alpha$ Deduction rule: on Example $\alpha \vdash \alpha$. □

• **Do not confuse** $\alpha \to \beta$ **and** $\alpha \vdash \beta$. The notation $\alpha \to \beta$ is a proposition, that is, a word in the language Prop of Propositional Logic. The notation $\alpha \vdash \beta$ is a natural language expression that says "β is provable from α."

Theorem 2.2. ***(Soundness and Completeness)***
– *If* $\Gamma \vdash \alpha$ *then* $\Gamma \models \alpha$ *(soundness).*
– *If* $\Gamma \models \alpha$ *then* $\Gamma \vdash \alpha$ *(completeness).*

This theorem asserts that (i) Formal Deduction is sound, that is, it proves only propositions that are semantically correct, and (ii) Formal Deduction is complete, that is, every semantically correct proposition is formally provable/deducible. In other words, the concepts of "logical consequence" and "formal deducibility" are equivalent.

Exercises of Section 2.5

Ex. 2.18. Give a formal proof of $\alpha \vdash \neg\neg\alpha$

Solution. To prove $\neg\neg\alpha$ we shall use rule IR09, which requires to prove $\neg\alpha \to \mathbf{F}$, which requires to use IR12 (the deduction rule), which requires a subproof of $\neg\alpha \vdash \mathbf{F}$.

1.	α	given
2.	SUB: $\neg\alpha$	subproof for $\neg\alpha \vdash \mathbf{F}$
3.	END $\mathbf{F}$	Contradiction: lines 1,2
4.	$\neg\alpha \to \mathbf{F}$	Deduction rule: line 2
5.	$\neg\neg\alpha$	$\neg$ Introduction: line 4

□

Ex. 2.19. Give a formal proof of $\neg\neg\alpha \vdash \alpha$

Ex. 2.20 (***Transitivity***)**.** Give a formal proof of

$$p \to q, q \to r \;\vdash\; p \to r$$

Solution. To prove $p \to r$ we shall use the Deduction rule, that is, in addition to the assumptions $p \to q, q \to r$ we shall assume p and then prove r (in a subproof):

1.	$p \to q$	given
2.	$q \to r$	given
3.	SUB: p	subproof for $p \vdash r$
4.	q	Modus ponens: lines 1,3
5.	END r	Modus ponens: lines 2,4
6.	$p \to r$	Deduction rule: line 3

□

Ex. 2.21. Give a formal proof of $\mathbf{F} \vdash \alpha$
Note: This says that, if we use false in our assumptions, then we can prove any proposition! *Hint:* Try to prove $\neg\alpha \to \mathbf{F}$.

Ex. 2.22. Give a formal proof of $\neg\alpha \vdash \alpha \to \beta$.

Ex. 2.23. Give a formal proof of $\alpha \vee \beta \vdash \beta \vee \alpha$

Solution. Strategy: Let $\gamma = \beta \vee \alpha$. We are going to show $\alpha \to \gamma$ and $\beta \to \gamma$, and then apply the "case analysis" rule.

1.	$\alpha \vee \beta$	given	
2.	SUB:	α	subproof for $\alpha \vdash \beta \vee \alpha$
3.	END	$\beta \vee \alpha$	$\vee$-introduction: line 2
4.	$\alpha \to \beta \vee \alpha$	Deduction rule: line 2	
5.	SUB:	β	subproof for $\beta \vdash \beta \vee \alpha$
6.	END	$\beta \vee \alpha$	$\vee$-introduction: line 5
7.	$\beta \to \beta \vee \alpha$	Deduction rule: line 5	
8.	$\beta \vee \alpha$	Case analysis: lines 1,4,7	

□

Ex. 2.24. Give a formal proof of $\neg\alpha \wedge \neg\beta \ \vdash\ \neg(\alpha \vee \beta)$
Hint: Try a subproof of $(\alpha \vee \beta) \ \vdash\ \mathbf{F}$.

Ex. 2.25. Give a formal proof of $\neg(\alpha \vee \beta) \vdash \neg\alpha \wedge \neg\beta$

Ex. 2.26 (***Non-provable proposition***)**.** Show that there is ***no*** formal proof of

$$p \vee q \ \vdash\ p \wedge q.$$

Solution. By the theorem of soundness and completeness, it is sufficient to show that

$$p \vee q \ \not\models\ p \wedge q,$$

that is, there exists a valuation *val* that makes $p \vee q$ true and $p \wedge q$ false. Well, here it is: let *val* be such that $val\,(p) = \mathbf{T}$ and $val\,(q) = \mathbf{F}$. □

Ex. 2.27. Show that there is ***no*** formal proof of $p \to q,\ q \ \vdash\ q \to p$

2.6 Normal forms and Resolution Proving

The concepts of proving or inferring new knowledge from existing one are of central importance in the field of Artificial Intelligence. In particular, many expert systems rely on these concepts and require efficient methods of applying the concepts to real world applications. The method of resolution is a formal proof method that allows one to test a hypothesis h based on a set A of facts. The facts and the hypothesis are expressed as propositions and we wish to know whether

$$A \vdash h,$$

that is, h is provable from A. In resolution proving, there is *only one inference rule:* resolution. Moreover, every proposition must be in conjunctive normal form.

• **Conjunctive normal form**. A literal is a logical variable, or the negation of a variable such as p, $\neg p$, etc. A proposition α is said to be in CNF (conjunctive normal form), if α is a conjunction of clauses, where a clause is a disjunction of literals; that is, α is of the form $\alpha_1 \wedge \cdots \wedge \alpha_n$ and each α_i is of the form $\ell_1 \vee \cdots \vee \ell_m$ with each ℓ_j being a literal. For example, the proposition

$$(\neg p \vee q) \wedge (\neg p \vee r)$$

is in CNF and consists of two clauses: $\neg p \vee q$ and $\neg p \vee r$. The complement of a literal ℓ is denoted by $\bar{\ell}$ and is equal to $\neg p$, if ℓ is the variable p, or p if ℓ is $\neg p$.

Theorem 2.3. *Every proposition is equivalent to one in CNF.*

Ex. 2.28. Using the laws of propositional logic we show how $p \rightarrow (q \wedge r)$ can be written in CNF:

$$\begin{aligned} p \rightarrow (q \wedge r) &\equiv \neg p \vee (q \wedge r) \\ &\equiv (\neg p \vee q) \wedge (\neg p \vee r) \end{aligned}$$

• **Resolution**. This is an inference rule that can be used to test whether $A \vdash h$, where all propositions involved are in CNF. First note that

$$A \vdash h \text{ iff the set } A \cup \{\neg h\} \text{ is unsatisfiable.}$$

Let C be the set of all clauses that appear in $A \cup \{\neg h\}$. Then,

$$A \cup \{\neg h\} \text{ is unsatisfiable iff } C \text{ is unsatisfiable.}$$

We attempt to **resolve** any two clauses $c_1, c_2 \in C$ and, if this is possible, we add the resolvent in C. Here, we say that two clauses

c_1, c_2 can resolve if they contain two complementary literals ℓ and $\bar{\ell}$ and, in this case, the resolvent is the clause containing the literals of c_1 and c_2 except for $\ell, \bar{\ell}$. For example, the resolvent of $p \vee \neg q$ and $r \vee \neg p \vee s$ is $\neg q \vee r \vee s$. If c_1 and c_2 happen to be single complementary literals then their resolvent is $\mathbf{F}$. The process terminates when $\mathbf{F}$ is obtained, or no two clauses in C can be resolved. In the former case we know that C is unsatisfiable, which implies that the hypothesis h is correct.

Ex. 2.29. Recall in Ex. 2.6 we had the following set of propositions:

$$\{ar \vee be \vee ch, be \vee ch, be \to ch, ar \to \neg be \wedge \neg ch\}.$$

We want to test whether any of ar, be, ch is provable from this set. For example, to test ch we add the proposition $\neg ch$ into the set and compute the set of clauses corresponding to the propositions. So we get the following clauses:

$$ar \vee be \vee ch,\ be \vee ch,\ \neg be \vee ch,\ \neg ar \vee \neg be,\ \neg ar \vee \neg ch, \neg ch.$$

The resolution process can be as follows:

1. $be \vee ch$ (given)
2. $\neg be \vee ch$ (given)
3. $\neg ch$ (given)
4. $\neg be$ (resolution on 2,3)
5. be (resolution on 1,3)
6. $\mathbf{F}$ (resolution on 4,5)

We can also perform resolution by arranging the clauses involved as a tree:

```
be ∨ ch      ¬ch      ¬be ∨ ch
        \ /       \ /
        be        ¬be
           \     /
              F
```

Hence, ch must be true.

2.7 A Note on logic-based expert systems

As mentioned in the introductory paragraph of Section 2.6, formal deduction methods are applicable in the area of expert systems, in particular, rule based expert systems. In this context, a ***rule*** is a proposition of the form

$$\alpha \to \beta$$

such that α represents a condition that must be satisfied in order to make the decision β, or take the action β. For example, α could mean "you have adequate savings" and β could mean "you should invest in stocks". The system involves three sets:

$R =$ a set of rules,
$D =$ a set of possible decisions/actions (usually variables),
$F =$ a set of facts about a particular case (usually literals).

Testing whether the action $d \in D$ is appropriate for the case F means deciding whether

$$R \cup F \vdash d,$$

or, equivalently, whether the set $R \cup F \cup \{\neg d\}$ is unsatisfiable (here we assume that already $R \cup F$ is satisfiable). This question can be answered using resolution if we convert the set $R \cup F \cup \{\neg d\}$ into an equivalent set of clauses.

Exercises of Sections 2.6, 2.7

Ex. 2.30. Use the laws of Propositional Logic to show how

$$(p \wedge q) \vee r \to s$$

can be written equivalently in CNF.

Ex. 2.31. Prove the following claim, which we used to establish the correctness of Resolution.

$$A \vdash h \text{ iff the set } A \cup \{\neg h\} \text{ is unsatisfiable.}$$

Ex. 2.32. Translate the following statements into rules:

1. Your income is adequate iff it is steady and above the minimum income.

2. If you have adequate savings and an adequate income then you should invest in stocks.

3. If you have no adequate savings then you should invest in savings.

4. If you have adequate savings but no adequate income then your investment should be a combination of stocks and savings.

Use the variables

mi : "your income is above the minimum income",
si : "your income is steady",
ai : "your income is adequate",
as : "you have adequate savings",
st : "you should invest in stocks",
sv : "you should invest in savings",
co : "you should invest in a combination of stocks and savings".

Convert the set of rules into an equivalent set C of clauses. Then consider the facts $F = \{\text{si},\text{as},\neg\text{mi}\}$ and use resolution to show that, in this case, the person should invest in a combination of stocks and savings.

Ex. 2.33. Use resolution to prove the transitivity of implication – see Ex. 2.20

3 PREDICATE LOGIC (First Order Logic)

Propositional logic is the basic logic for formal specification and modeling of logic problems, but it does not provide a mechanism for dealing with situations involving an unbounded number of entities. For example, we would like to express a general statement of the form "every human is mortal," which could be applied to any particular human using the rule Modus Ponens. More specifically, we would like to have 'propositions' of the form $h(u)$: "u is human" and $m(u)$: "u is mortal," and to be able to apply Modus Ponens for the human Socrates as follows:

$$h(u) \to m(u),\ \ h(\text{Socrates}) \vdash m(\text{Socrates}).$$

To be able to do this in propositional logic we would need to have an unbounded number of variables of the form sh : "Socrates is human" and sm : "Socrates is mortal" (that is, two new variables for each person). Predicate Logic (aka First Order Logic) addresses this limitation by using a richer language.

3.1 The language of Predicate Formulas: syntax

The language of predicate logic involves constants, variables, function and relation names, as well as quantifiers, parentheses and the connective symbols of propositional logic. More specifically:

- CONSTANTS = a set of names[5]: names of objects, digital words such as 235, people or places such as tom, everest, etc.
- FUNCTIONS = a set of function names: `f, g, h, height, capital,` etc.
- RELATIONS = a set of relation (or predicate) names: F, G, H, P, Q, R, IsCapital, Takes, GreaterThan, etc. The relation name == is special and will be used to denote equality. We also have the constant relation names **T** and **F**.
- VARIABLES = a set of variables: u, v, w, x, y, z (possibly with subscripts).

[5] Our syntax here is a little loose. We do assume though that the first letters of names are in lower case.

- QUANTIFIERS: $\forall$ (for all, for every, for each) and $\exists$ (there exists, there is, for some).
- Parentheses and connectives as in Propositional Logic.

Every function and every relation name has a certain arity, which is the number of parameters it accepts – see below. Arity 0 have only the constant relations **T** and **F**. Next we define the concepts term and atom, which are necessary to arrive at the definition of formula.

• **Term**. Any expression that can be formed by the following rules (and only these):

– Every constant is a term

– Every variable is a term

– If f is a function name of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

• **Atom (atomic formula)**. This is the simplest kind of formula: $R(t_1, \ldots, t_n)$, where R is any relation name of some arity n and $t_1, \ldots, t_n$ are terms. The special relation '==' is used with two terms: $== (t_1, t_2)$ or $(t_1 == t_2)$.

Definition 3.1. A formula is any expression that can be formed by the following rules (and only these):

- Every atom is a formula.
- If β is a formula then $(\forall x\, \beta)$ and $(\exists x\, \beta)$ are formulas.
- If α and β are formulas then $(\neg\alpha), (\alpha \vee \beta), (\alpha \wedge \beta), (\alpha \rightarrow \beta), (\alpha \leftrightarrow \beta)$ are formulas.

Ex. 3.1. The following expressions are formulas:
$\exists x\, \textsc{Parent}(x, \text{tom})$,
$\textsc{Parent}(u, \text{tom})$,
$\forall x\, (\textsc{Human}(x) \rightarrow \textsc{Mortal}(x))$,
$\textsc{GreaterThan}(\texttt{height}(u), 50)$.

The following expressions are ***not*** formulas:
$\exists\textsc{Human}(\text{x})$,
$\textsc{Parent}(\textsc{Human}(u), \text{tom})$.

• **Notation**. An expression of the form $\beta(t)$ denotes a formula in which possibly the term t occurs. In this case, the expression $\beta(s)$ denotes the formula that results when we replace all occurrences of t in $\beta(t)$ with s. A similar notation is the following: If β is a formula and t, s are terms then $\beta[t \leftarrow s]$ is the formula that results when we replace in β every occurrence of t with s.

• **Bound and Free variable occurrences**. In a formula of the form $\forall x\, \beta(x)$ or $\exists x\, \beta(x)$, we say that any occurrence of the variable x in $\beta(x)$ is a bound occurrence. If a variable occurrence, say u, is not a bound occurrence then it is a free variable occurrence. For example, in the formula $\exists x\, \textsc{Parent}(x, u)$, x has a bound occurrence and u has a free occurrence. In order to simplify the presentation, in the sequel, we shall use the symbols u, v, w for variables with only free occurrences and the symbols x, y, z for variables with only bound occurrences.

• **Special Notation**. Many authors separate the quantifier and the rest of a formula using ':', that is, $\forall x : \beta(x)$. Moreover, when R is a unary relation, many authors use the notation

$$\forall x \in R : \beta(x) \text{ as a shorthand for } \forall x\, (R(x) \rightarrow \beta(x)).$$

A similar notation is used with the existential quantifier $\exists$:

$$\exists x \in R : \beta(x) \text{ is a shorthand for } \exists x\, (R(x) \wedge \beta(x)).$$

• **Sentence**. Any formula containing ***no*** free occurrences of variables is called a sentence. For example, the formula

$$\exists x\, \textsc{Parent}(x, \mathrm{tom})$$

is a sentence. The formula $\textsc{Parent}(u, \mathrm{tom})$ is not a sentence because it contains a free occurrence of the variable u .

3.2 Semantics (meaning) of Formulas

For the semantics of formulas, we are required to specify the following.

1. A ***universe*** $\mathcal{U}$, which is a nonempty set whose elements are usually called entities.
2. The meaning of the constants, function and relation names with respect to the universe $\mathcal{U}$.
3. An assignment of values in $\mathcal{U}$ to the free variables.

The first requirement is simple: we pick the universe for which we are interested in expressing statements and performing reasoning.

The third requirement is also simple: we pick a value assignment σ such that $\sigma(w)$ is the value in $\mathcal{U}$ of the free variable w – this value could be a person, a number, etc.

The rigorous approach for the second requirement is to pick a meaning function (an interpretation) I assigning meaning to constants, function and relation names. For example, for the constant 50 (which is a string), the meaning $I(50)$ could be the number fifty (which is a concept and an element of the chosen universe $\mathcal{U}$).[6] Here, however, we simplify matters by hiding the function I and specifying the meaning of a constant either implicitly by using a meaningful name for that constant, or explicitly with the ':' notation, which was also used for the meaning of Propositional variables. In either case, we make the ***convention*** that writing a constant c is the same as writing the meaning of c. We apply the same convention also to function and relation names[7]. Thus, when not given explicitly, the meaning of the constant 50 is the number fifty, or we could write explicitly 50 : "the number fifty".

• **Functions and Relations/Predicates**. The meanings of relation names and function names will be relations and functions on the elements of $\mathcal{U}$. An n-ary relation R is a set of tuples $(e_1, \ldots, e_n)$ of elements in $\mathcal{U}$. For example, we can have the unary relation name Person and the binary relation name Taller such that

$$\text{PERSON}(u)\text{: "}u\text{ is a person"},\quad \text{TALLER}(u,v)\text{: "}u\text{ is taller than }v\text{"}.$$

[6] Of course it is possible, for whatever reason, to assign the meaning "my car" to the constant 50 and evaluate formulas based on that meaning!

[7] For example, the meaning of some function name `log` of arity 1 is a function $I(\texttt{log}) : \mathcal{U} \to \mathcal{U}$. Again, however, we assume that `log` is both the name and the meaning of the function.

Then, the meaning of the formula

$$\textsc{Person}(p_1) \wedge \textsc{Person}(p_2) \wedge \textsc{Taller}(p_1, p_2)$$

is "person p_1 is taller than person p_2".

An n-ary function f associates, to each tuple $(e_1, \ldots, e_n)$ of elements in $\mathcal{U}$, a value $f(e_1, \ldots, e_n)$ in the universe $\mathcal{U}$. For example, we can use the unary function name `father` such that

$$\texttt{father}(u) : \text{“the father of } u\text{.”}$$

Thus, if p is a person in $\mathcal{U}$ then $\texttt{father}(p)$ is another person in $\mathcal{U}$, the father of p.

• **Valuation**. In order to assign values to terms and formulas, we use a valuation *val* that involves the universe $\mathcal{U}$, the value assignment σ, and the meaning we have chosen for constants, function and relation names.

• **Value of a term**. The value of a term t is always an entity in the chosen universe $\mathcal{U}$, that is, $\mathit{val}\,(t) \in \mathcal{U}$. In particular, the value of a constant c is the meaning of c in $\mathcal{U}$. Based on the convention we made earlier, we simply write $\mathit{val}\,(c) = c$. The value of a free variable u is $\sigma(u)$, that is, $\mathit{val}\,(u) = \sigma(u)$. The value of a term $f(t_1, \ldots, t_n)$ is

$$\mathit{val}\,(f(t_1, \ldots, t_n)) \;=\; f(\mathit{val}\,(t_1), \ldots, \mathit{val}\,(t_n)).$$

• **Truth value of an Atomic Formula**. A relation name R of some arity n refers to a relation that is a subset of $\mathcal{U}^n = \mathcal{U} \times \cdots \times \mathcal{U}$. With this in mind, we have that $\mathit{val}\,(R(t_1, \ldots, t_n)) = \mathbf{T}$ iff the tuple $(\mathit{val}\,(t_1), \ldots, \mathit{val}\,(t_n))$ is in R. The case of the equality "==" is special: it refers to the relation $\{(e, e) \mid e \in \mathcal{U}\}$; that is, $\mathit{val}\,(t_1 == t_2) = \mathbf{T}$ iff $(\mathit{val}\,(t_1), \mathit{val}\,(t_2)) \in \{(e, e) \mid e \in \mathcal{U}\}$ iff $\mathit{val}\,(t_1) = \mathit{val}\,(t_2)$. We note here that "=" is the symbol for equality in the English language, which is used as a metalanguage to explain the meaning of the symbols (such as '==') and expressions in the formal language of Predicate Logic!

• **Truth value of a Composite formula**. We have that

$$\begin{aligned} \mathit{val}\,(\forall x \beta(x)) = \mathbf{T} &\text{iff } \text{ for } \underline{\text{every}} \text{ entity } e \in \mathcal{U}, \text{ it is } \mathit{val}\,(\beta(e)) = \mathbf{T}. \\ \mathit{val}\,(\exists x \beta(x)) = \mathbf{T} &\text{iff } \text{ for } \underline{\text{some}} \text{ entity } e \in \mathcal{U}, \text{ it is } \mathit{val}\,(\beta(e)) = \mathbf{T} \\ &\text{iff } \text{ there exists } e \in \mathcal{U} \text{ with } \mathit{val}\,(\beta(e)) = \mathbf{T}. \end{aligned}$$

The rules for the rest of the composite formulas are exactly as in Propositional Logic. For example, $\mathit{val}\,(\alpha \wedge \beta) = \mathbf{T}$ iff $\mathit{val}\,(\alpha) = \mathit{val}\,(\beta) = \mathbf{T}$.

We note that we have abused the notation above by using the expression $\mathit{val}\,(\beta(e)) = \mathbf{T}$ with $e \in \mathcal{U}$, as e is not an element of the formal language of Predicate Logic. The rigorous approach is to write $\mathit{val}\,(\beta(u)) = \mathbf{T}$, where u is any unused free variable (not in $\beta(x)$) with $\sigma(u) = e$. Here, however, we settle with this non-rigorous approach without compromising the presentation of subsequent concepts.

• **Equivalence, Consequence, Satisfiability, Model**. These concepts exist in Predicate Logic exactly as in Propositional Logic. For example, the notion of equivalence of two formulas α, β is as follows:

$$\alpha \equiv \beta \text{ iff } \mathit{val}\,(\alpha) = \mathit{val}\,(\beta), \text{ for } \textbf{\textit{every}} \text{ valuation } \mathit{val}\,.$$

Also, the notion of consequence is as follows: "$A \models \beta$" iff, for every valuation val , when $\mathit{val}\,(\alpha)$ is true for all $\alpha \in A$ then also $\mathit{val}\,(\beta)$ must be true. Two important equivalences are

$$\begin{aligned} \exists x\, \beta(x) &\equiv \neg(\forall x\, \neg\beta(x)) \\ \forall x\, \beta(x) &\equiv \neg(\exists x\, \neg\beta(x)). \end{aligned}$$

As usual, when $\mathit{val}\,(\alpha) = \mathbf{T}$ for every $\alpha \in A$, we say that val is a <u>model</u> of the set of formulas A, or that A is <u>satisfiable</u>.

• **No truth tables**. Unfortunately the method of truth tables ***cannot*** be extended to evaluate arbitrary formulas.

Ex. 3.2. Consider the predicate (relation) names P, R, M, L with arities 1, 2, 1, 1, respectively, and a universe in which these predicates have the following meaning

P(u) : "u is a person"
R(u, v) : "u reads story v"
M(v) : "v is a mystery story"
L(v) : "v is a love story".

Consider also the following statements.

β_1 : “There are some people who read mystery stories.”

β_2 : “Everybody who reads mystery stories also reads love stories.”

β_3 : “There are some people who read love stories.”

β_4 : “Peter reads love stories.”

β_5 : “Nobody reads love stories.”

1. Use the given predicates (and no others) and the given meaning to translate the five statements into Predicate Logic sentences $\beta_1, \ldots, \beta_5$.
2. Explain whether or not $\{\beta_1, \beta_2\} \models \beta_3$.
3. Explain whether or not $\{\beta_1, \beta_4\}$ is satisfiable.

Reminder: Recall that, in general, a given set of formulas can have different semantics depending on the meaning we assign to the predicates and terms involved in these formulas.

Solution.

1. The required formulas are:

$$\begin{aligned}
\beta_1 &= \exists x\, \exists y\, (\mathrm{P}(x) \wedge \mathrm{M}(y) \wedge \mathrm{R}(x,y)) \\
&= \exists x \in \mathrm{P}\ \ \exists y \in \mathrm{M} : \mathrm{R}(x,y). \\
\beta_2 &= \forall x \forall y : \mathrm{P}(x) \wedge \mathrm{M}(y) \wedge \mathrm{R}(x,y) \to \exists z (\mathrm{L}(z) \wedge \mathrm{R}(x,z)). \\
&= \forall x \in \mathrm{P}\ \ \forall y \in \mathrm{M} : \mathrm{R}(x,y) \to \exists z (\mathrm{L}(z) \wedge \mathrm{R}(x,z)). \\
\beta_3 &= \exists x\, \exists y\, (\mathrm{P}(x) \wedge \mathrm{L}(y) \wedge \mathrm{R}(x,y)) \\
&= \exists x \in \mathrm{P}\ \ \exists y \in \mathrm{L} : \mathrm{R}(x,y). \\
\beta_4 &= \exists y \in \mathrm{L} : \mathrm{R}(\mathrm{peter}, y). \\
\beta_5 &= \neg\, \exists x \in \mathrm{P}\ \ \exists y \in \mathrm{L} : \mathrm{R}(x,y). \\
&\equiv \forall x \in \mathrm{P}\ \ \forall y \in \mathrm{L} : \neg \mathrm{R}(x,y).
\end{aligned}$$

2. We have to show that, for ***every*** valuation val, if $val\,(\beta_1) = val\,(\beta_2) = \mathbf{T}$ then also $val\,(\beta_3) = \mathbf{T}$.
   Indeed, no matter what *val* is, β_1 says that there are two entities $e_1 \in \mathrm{P}$ and $e_2 \in \mathrm{M}$ such that $(e_1, e_2) \in R$. If we use e_1, e_2 in β_2, we get that $(e_1, e_2) \in R$ implies $(e_1, e_3) \in R$ for some entity $e_3 \in \mathrm{L}$. Thus, there are entities $e_1 \in \mathrm{P}$ and $e_3 \in \mathrm{L}$ such that $(e_1, e_3) \in R$; that is, $val\,(\beta_3) = \mathbf{T}$.

3. It is sufficient to give an example of ***some*** valuation where both β_1 and β_4 are true. We can say that with the meaning given above the two formulas are indeed true, as there are people who read mystery stories and some specific person corresponding to the constant Peter reads love stories. Another example could be as follows: $\mathrm{P}(u)$: "u is a real number", $\mathrm{M}(u)$: "u is an integer", $\mathrm{L}(u)$: "u is a nonnegative integer", $\mathrm{R}(u, v)$: "$u < v$", peter : "5". Then β_1 and β_4 mean that "there is a real number that is smaller than some integer, and that 5 is smaller than some nonnegative integer". Obviously, this is correct and, therefore, the two formulas are again true.

□

Ex. 3.3. A main objective of Predicate Logic is to allow us to formally specify a universe and its properties. For example, it is possible to specify the universes containing exactly two elements using the sentence

$$\beta = \exists x_1 \exists x_2 (\neg(x_1 == x_2) \land \forall x(x == x_1 \lor x == x_2)).$$

According to the semantics of formulas, if *val* is any model of β involving some universe $\mathcal{U}$, that is *val* $(\beta) = \mathbf{T}$, then there are two entities $e_1, e_2 \in \mathcal{U}$, which are different, and every element $e \in \mathcal{U}$ is equal to e_1 or e_2. Obviously, this means that the universe must contain exactly two elements.

• **An impossibility of specification**. The question that arises here is whether it is possible, for any desired type of universe, to write a formula β that specifies exactly this type of universe – recall, in the previous example, this was possible for universes having exactly two elements. In other words whether the formula β is satisfiable (true) exactly in a universe of the desired type. The next theorem implies that it is impossible to write a formula that is satisfiable only in universes that are uncountable – for instance, in the universe of real numbers.

Theorem 3.1. ***(Löwenheim and Skolem)*** *Let A be a set of formulas. If A is satisfiable then A is satisfiable in some countable universe.*

Exercises of Section 3.2

Ex. 3.4. Give a formula that is true exactly when the universe involved consists of at most two elements.

Solution. We give two solutions. One where the formula is a sentence (involves no free variables) and one where the formula is not a sentence:

$$\begin{aligned} \beta_1 &= \exists x_1 \exists x_2 \forall x (x == x_1 \vee x == x_2) \\ \beta_2 &= \forall x (x == u \vee x == v) \end{aligned}$$

When the first formula is true in a universe $\mathcal{U}$ then there are two elements in $\mathcal{U}$ (possibly the same) such that each element in $\mathcal{U}$ is equal to one of these two elements. Thus, $\mathcal{U}$ must contain at most two elements. The second formula involves free variables so we need to consider the possible values of these variables. Any valuation *val* would give to u, v two different values, or the same value, in $\mathcal{U}$; that is, *val* (u) and *val* (v) are two (possibly the same) elements in $\mathcal{U}$. So the formula says that every element of the universe is equal to *val* (u) or to *val* (v). Thus, no matter what *val* is, the universe must have at most two different elements. □

Ex. 3.5. [Nonempty sequence] Consider the following constant and predicate:

first : "the first element of the sequence"

$\textsc{Next}(u, v)$: "u is the successor of v"

Use only the above names to write sentences as follows.

β_1 : "any element is either first or the successor of an element"

β_2 : "the first element is not the successor of any element"

β_3 : "if x is the successor of y then y is not the successor of x"

β_4 : "the sequence is finite"

3.3 Formal Deduction: More Inference rules

The concept of inference rule and proof in Propositional Logic also applies here. Again we assume that proofs and rules are applied with respect to a, possibly empty, set Γ of assumptions (formulas) that are assumed to be true. We have the following additional inference rules that deal with quantifiers and the equality predicate:

IR13 Universal instantiation: $\forall\mathtt{x}\beta(\mathtt{x}) \vdash \beta(t)$, where t is any term.

IR14 Universal introduction: $\beta(\mathtt{u}) \vdash \forall\mathtt{x}\beta(\mathtt{x})$, where $\mathtt{u}$ is any free variable that does not occur in any open[8] assumption.

IR15 $\exists$-introduction: $\beta(t) \vdash \exists\mathtt{x}\beta'(\mathtt{x})$, where t is any term and $\beta'(\mathtt{x})$ results if we replace some occurrences of t in β with $\mathtt{x}$.

IR16 $\exists$-elimination: $\alpha(\mathtt{u}) \to \beta, \exists\mathtt{x}\alpha(\mathtt{x}) \vdash \beta$, where $\mathtt{u}$ is any free variable that does not occur in any open assumption or in β.

IR17 $\beta(t_1), (t_1 == t_2) \vdash \beta'(t_2)$, where t_1, t_2 are any terms and $\beta'(t_2)$ results if we replace some occurrences of t_1 in β with t_2.

IR18 $\vdash (t == t)$, for any term t.

Ex. 3.6. We go back to the discussion in the beginning of Section 3 regarding the extra capabilities of Predicate Logic compared to Propositional Logic. For any term t (e.g., $t =$ socrates), we prove that

$$\forall x\,(\textsc{Human}(x) \to \textsc{Mortal}(x)),\ \textsc{Human}(t)\ \vdash\ \textsc{Mortal}(t),$$

that is, if every human is mortal and t is human, then t must be mortal.

1.	$\forall x\,(\textsc{Human}(x) \to \textsc{Mortal}(x))$	given
2.	$\textsc{Human}(t)$	given
3.	$\textsc{Human}(t) \to \textsc{Mortal}(t)$	$\forall$-instantiation: line 1
4.	$\textsc{Mortal}(t)$	Modus ponens: lines 2,3

Ex. 3.7. Here is a formal proof of
$\forall x(\alpha(x) \to \beta(x)) \vdash \forall x\alpha(x) \to \forall x\beta(x)$:

1.	$\forall x(\alpha(x) \to \beta(x))$		given
2.	$\alpha(u) \to \beta(u)$		$\forall$-instantiation: line 1: u not in $\alpha(x) \to \beta(x)$
3.	SUB:	$\forall x\alpha(x)$	subproof for $\forall x\alpha(x) \vdash \forall x\beta(x)$
4.		$\alpha(u)$	$\forall$-instantiation: line 3
5.		$\beta(u)$	Modus ponens: lines 2,4

[8] Every assumption is considered open until the end of the proof or subproof for which the assumption is used.

6.	END $\forall x\beta(x)$	$\forall$-introduction: line 5
7.	$\forall x\alpha(x) \to \forall x\beta(x)$	Deduction rule: line 3

Ex. 3.8. Here is a formal proof of $(t_1 == t_2) \vdash (t_2 == t_1)$:

1.	$t_1 == t_2$	given
2.	$t_1 == t_1$	Rule IR18
3.	$t_2 == t_1$	Rule IR17: lines 2, 1

• **Basic Theorems**. The theorems of Replaceability and, Soundness and Completeness, also hold in the case of Predicate Logic. Thus, any formula β that can be formally deduced (proved) from our assumptions Γ, that is $\Gamma \vdash \beta$, is also a logical consequence of Γ: $\Gamma \models \beta$. Conversely, for any formula β that is a logical consequence of Γ there is a formal proof of β using the assumptions in Γ.

• **Gödel's Incompleteness Theorem**. The semantics of Predicate Logic does not provide an algorithmic method for establishing the truth of a formula from the given assumptions Γ. The problem here is that, unlike the case of Propositional Logic, there are infinitely many possible universes[9] and valuations one has to consider. On the other hand, formal deduction is a purely syntactic system and one would hope that there exists a program that decides whether or not a given formula is provable from Γ. This question was raised by David Hilbert in the beginning of the 20th century, using as Γ a set of standard mathematical formulas about the nonnegative integers. In other words the question was whether there exists a program that would be able to prove all mathematical theorems about numbers – a rather scary thought for professional mathematicians! In 1931, Kurt Gödel showed that this is not possible:

Theorem 3.2. *If the set of assumptions Γ is consistent*[10] *and sufficiently rich (e.g. includes the basic statements of arithmetic) then* Γ

[9] In fact, uncountably many.

[10] Consistency of a set of formulas Γ means that Γ cannot prove both, a formula and its negation.

is incomplete, that is, there is a sentence δ such that neither δ nor $\neg\delta$ is provable from Γ:

$$\Gamma \nvdash \delta \quad \textit{and} \quad \Gamma \nvdash \neg\delta.$$

A sentence like the δ above is called independent of Γ. People have now discovered several independent statements. The important issue here is that Γ could be the standard assumptions (axioms) of mathematics and that there are statements whose truth value is independent of formal mathematics! It is interesting to note that very few people initially understood the importance of the incompleteness theorem. It is said that John von Neumann (recall the von Neumann architecture of computers) was among the first who grasped the importance of this result.

The phenomenon of incompleteness in formal mathematics has been explained in [3] in terms of complexity. Roughly speaking, each sentence has a complexity value, which is a positive integer and indicates how difficult it is to describe the sentence. It turns out that, for every formal mathematical system, there is a positive integer N such that the system cannot prove any sentence of complexity greater than N.

• **Proof idea of Gödel's Incompleteness Theorem**. We use [21] as a reference. There are four main parts in the proof. ***First***, Gödel assigned a unique ID to every formula β, which is called the Gödel number of β and denoted as gid(β). With this technique it is possible for a formula α to refer to a formula β via the number gid(β). ***Second*** – this was a really ingenious part – he managed to define a predicate $\textsc{Provable}(u)$ such that

$$\textsc{Provable}(u): \quad \text{“There exists a proof for the formula whose Gödel number is } u\text{”.}$$

Third, he showed that there is a sentence δ such that

$$\Gamma \vdash \neg\delta \leftrightarrow \textsc{Provable}(\text{gid}(\delta)).$$

The ***fourth*** part concludes the proof using contradiction twice as follows: (i) If $\Gamma \vdash \delta$ then δ is provable and, by definition of $\textsc{Provable}()$, we have $\Gamma \vdash \textsc{Provable}(\text{gid}(\delta))$. But, by definition of δ, it follows that $\Gamma \vdash \neg\delta$, which contradicts the consistency of Γ. Hence,

$\Gamma \nvdash \delta$. (ii) Now if $\Gamma \vdash \neg\delta$ then, by definition of δ, it follows that $\Gamma \vdash$ Provable(gid(δ)), and then, by definition of Provable(), we have $\Gamma \vdash \delta$, which contradicts the consistency of Γ. Hence, $\Gamma \nvdash \neg\delta$.
□

We now know that, for a given finite set Γ of formulas (assumptions/axioms), the language of all formulas that are provable from Γ is not recursive – see the terminology of Section 1.3. In other words, we have the following theorem.

Theorem 3.3. *The following problem is undecidable.*
Input: set of formulas Γ and formula β.
Return: YES/NO, depending on whether β is provable from Γ.

Exercises of Section 3.3

Ex. 3.9. Give a formal proof of $\forall x\, \alpha(x) \vdash \exists x\, \alpha(x)$

Ex. 3.10. Give a formal proof of $\forall x \forall y \beta(x,y) \vdash \forall y \forall x \beta(x,y)$

Solution.

1.	$\forall x \forall y \beta(x,y)$	given
2.	$\forall y \beta(u,y)$	$\forall$-instantiation: line 1: u not in $\Gamma, \beta(x,y)$
3.	$\beta(u,v)$	$\forall$-instantiation: line 2: v not in $\Gamma, \beta(u,y)$
4.	$\forall x \beta(x,v)$	$\forall$-introduction: line 3
5.	$\forall y \forall x \beta(x,y)$	$\forall$-introduction: line 4

□

Ex. 3.11. Give a formal proof of $\alpha \to \forall x \beta(x) \vdash \forall x(\alpha \to \beta(x))$

Solution.

1.	$\alpha \to \forall x \beta(x)$		given
2.	SUB:	α	subproof for $\alpha \vdash \beta(u)$: u not in α, $\forall x \beta(x)$
3.		$\forall x \beta(x)$	Modus ponens: lines 1,2
4.	END	$\beta(u)$	$\forall$-instantiation: line 3
5.	$\alpha \to \beta(u)$		Deduction rule: line 2
6.	$\forall x(\alpha \to \beta(x))$		$\forall$-introduction: line 5

□

Ex. 3.12. Give a formal proof of $\neg\forall x\beta(x) \vdash \exists x\neg\beta(x)$

Ex. 3.13. Show that there exists ***no*** formal proof of

$$\vdash\ \forall x\forall y:\ P(x,y) \to P(y,x)$$

Hint: Use the theorem of Soundness and Completeness and the notion of (logical) consequence in Predicate Calculus.

3.4 ∃-free Prenex Normal Form

Definition 3.2. A formula is in *prenex normal form* if it is of the form

$$Q_1x_1\,Q_2x_2\cdots Q_nx_n\;\beta,$$

where each Q_i is a quantifier and the formula β is quantifier free.

A formula with no quantifiers is regarded as a trivial case of a prenex normal form.

• **Algorithm for prenex normal form**. Any formula can be converted to an equivalent one in prenex normal form, as follows:

1. Convert the formula to an equivalent one containing no connectives $\to$ and $\leftrightarrow$.
2. Move all negations inward such that, in the end, no double negations exist and each single negation appears in front of an atomic formula.
3. If necessary, rename bound variables such that no two quantifiers refer to variables with the same name.
4. Move all quantifiers to the front of the formula.

The above method can be made possible using the following laws:

For step 1:
$\alpha \to \beta \equiv \neg\alpha \vee \beta$
$\alpha \leftrightarrow \beta \equiv (\alpha \to \beta) \wedge (\beta \to \alpha)$
$\alpha \leftrightarrow \beta \equiv (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$

For step 2:
$\neg\neg\alpha \equiv \alpha$
$\neg\exists x\, \alpha(x) \equiv \forall x\, \neg\alpha(x)$
$\neg\forall x\, \alpha(x) \equiv \exists x\, \neg\alpha(x)$

For step 3:
$Qx\, \alpha(x) \equiv Qy\, \alpha(y)$ (the name of a variable is immaterial)

For step 4:
$\alpha \wedge Qx\, \beta(x) \equiv Qx\, (\alpha \wedge \beta(x))$, x not occurring in α
$\alpha \vee Qx\, \beta(x) \equiv Qx\, (\alpha \vee \beta(x))$, x not occurring in α
$Q_1x\, \alpha(x) \wedge Q_2y\beta(y) \equiv Q_1x\, Q_2y\, (\alpha(x) \wedge \beta(y))$
$Q_1x\, \alpha(x) \vee Q_2y\beta(y) \equiv Q_1x\, Q_2y\, (\alpha(x) \vee \beta(y))$
$\forall x\, \alpha(x) \wedge \forall x\, \beta(x) \equiv \forall x\, (\alpha(x) \wedge \beta(x))$
$\exists x\, \alpha(x) \vee \exists x\, \beta(x) \equiv \exists x\, (\alpha(x) \vee \beta(x))$
$\forall x\, \forall y\, \alpha(x,y) \equiv \forall y\, \forall x\, \alpha(x,y)$
$\exists x\, \exists y\, \alpha(x,y) \equiv \exists y\, \exists x\, \alpha(x,y)$

Ex. 3.14. Conversion to prenex normal form:

$$\begin{aligned} & \forall x : L(x) \to \neg\forall y M(x,y) \\ \equiv\ & \forall x : \neg L(x) \vee \neg\forall y M(x,y) \\ \equiv\ & \forall x : \neg L(x) \vee \exists y \neg M(x,y) \\ \equiv\ & \forall x \exists y : \neg L(x) \vee \neg M(x,y) \end{aligned}$$

• **∃-free prenex normal form**. A sentence is in ∃-free prenex normal form, if it is in prenex normal form and contains no existential quantifiers. The method of Skolem functions allows one to convert a sentence α to a sentence α' in ∃-free prenex normal form such that α is satisfiable iff α' is. The main idea is that if α is of the form $\forall x_1 \cdots \forall x_n \exists x \beta(x)$ then one can replace all occurrences of x with the term $f(x_1, \ldots, x_n)$, where f is a new function symbol.

• **Algorithm for ∃-free prenex normal form**. Step 1: Transform the given sentence into a sentence α_1 in prenex normal form. Let $i = 1$. Step 2: Repeat until all the existential quantifiers are removed: Assume α_i is of the form

$$\alpha_i = \forall x_1 \cdots \forall x_n \exists x \beta(x)$$

If $n = 0$ then α_i is of the form $\exists x \beta(x)$ and we define α_{i+1} to be the sentence $\beta(c)$ that results when we replace all x's in $\beta(x)$ with a new constant c. If $n > 0$ then we define

$$\alpha_{i+1} = \forall x_1 \cdots \forall x_n \beta(f(x_1, \ldots, x_n)),$$

where $\beta(f(x_1, \ldots, x_n))$ is the sentence obtained from $\beta(x)$ by replacing all x's in $\beta(x)$ with $f(x_1, \ldots, x_n)$ such that f is a new function symbol. Increase i by 1.

Ex. 3.15. The sentence in prenex normal form that was obtained in the previous example can be converted to the following one in $\exists$-free prenex normal form

$$\forall x : \neg L(x) \vee \neg M(x, f(x)).$$

3.5 Unification and Resolution

The concept of literal that was defined in Propositional Logic is also defined here as follows: a literal is an atomic formula or the negation of an atomic formula. Then the concept of a clause is the same as in Propositional Logic: a disjunction of literals.

• **Clauses**. As in propositional logic, using laws of predicate logic, it is possible to obtain from a given sentence h a set C_h of clauses such that h is satisfiable iff C_h is. Then we have that, for a given set $A = \{\alpha_1, \ldots, \alpha_n\}$ of sentences (assumptions) and a given sentence h (the hypothesis), $A \vdash h$ iff the set

$$C_{\neg h} \cup C_{\alpha_1} \cup \cdots \cup C_{\alpha_n}$$

is not satisfiable. To show that a set of sentences is not satisfiable we can use resolution, which requires the concept of resolving two clauses. However, in predicate logic, resolving of clauses takes place via unification.

• **Unification**. We say that two ***atomic*** formulas β_1, β_2 can unify (or are unifiable) if we can substitute terms for some variables in β_1, β_2 that would make the two atomic formulas identical. In this process, all occurrences of the same variable must be substituted by the same term. For example, the atoms $\textsc{Dad}(x, \text{mike})$ and $\textsc{Dad}(\text{doug},$

y) can unify via the substitution [$x \leftarrow$doug, $y \leftarrow$mike]. We note that although this definition of unification is not rigorous, it is adequate for our purposes.

• **Resolution**. We say that two clauses c_1, c_2 can resolve if they contain two complementary literals ℓ and $\bar{\ell}$ such that the atoms in ℓ and $\bar{\ell}$ can unify. In this case, the resolvent of c_1, c_2 is the clause containing the literals of both except for $\ell, \bar{\ell}$. If c_1 and c_2 happen to contain only $\ell, \bar{\ell}$ then their resolvent is **F**. The resolution process is now the same as in propositional logic. We start with a set of clauses; in each step we resolve two clauses, and we terminate when **F** is obtained or when no two clauses can be resolved. In the former case, we have that the original set of clauses is not satisfiable.

Ex. 3.16. If

$\neg H(x, z) \vee \neg \textsc{Dad}(x, \text{mike}) \vee \textsc{Mom}(z, \text{mike}),$
$\neg \textsc{Mom}(\text{pat}, y) \vee \textsc{Dad}(\text{doug}, y)$

are two of the given clauses, we can use the substitution $[x \leftarrow \text{doug}, y \leftarrow \text{mike}]$ for the literals $\neg\textsc{Dad}(x, \text{mike})$ and $\textsc{Dad}(\text{doug}, y)$, to obtain the resolvent

$\neg H(\text{doug}, z) \vee \textsc{Mom}(z, \text{mike}) \vee \neg \textsc{Mom}(\text{pat}, \text{mike}).$

Exercises of Sections 3.4, 3.5

Ex. 3.17. Convert the following formula to one in $\exists$-free prenex normal form.

$$\neg\textsc{Bottom}(\text{mybook}) \leftrightarrow \exists x\, \textsc{Below}(x, \text{mybook}).$$

Ex. 3.18. Use resolution to prove that everybody has a grandparent, provided that everybody has a parent. Use only one predicate:

$P(u, v)$: "u is a parent of v".

Solution. We wish to prove

$$\forall x \exists y P(y, x) \vdash \forall x \exists y \exists z : (P(z, y) \wedge P(y, x)).$$

For this, it is sufficient to prove that the set

$$\{\forall x \exists y P(y, x),\ \neg \forall x \exists y \exists z : (P(z, y) \wedge P(y, x))\}$$

is unsatisfiable. Before we use resolution, we must convert the above sentences into ∃-free prenex form and then into clauses. The two sentences in ∃-free prenex form will be as follows

$$\{\forall x : P(f(x), x),\ \forall y\, \forall z : (\neg P(z, y) \vee \neg P(y, c))\}.$$

The two clauses are obtained by simply dropping the quantifiers:

$$\{P(f(x), x),\ \neg P(z, y) \vee \neg P(y, c)\}.$$

Resolution will work as follows:

1. $P(f(x), x)$, given
2. $\neg P(z, y) \vee \neg P(y, c)$, given
3. $\neg P(z, f(c))$, resolve 1,2 using $[x \leftarrow c, y \leftarrow f(c)]$
4. **F**, resolve 1,3 using $[x \leftarrow f(c), z \leftarrow f(f(c))]$

Hence, the two clauses are unsatisfiable, as required. □

Ex. 3.19. Consider the following predicates

G(w) : "w is a ghost"

U(w) : "w is unhappy"

H(w) : "w is a house in the village"

Lives(u, w) : "u lives in w"

Use only the above names to write sentences as follows.

β_1 : "No ghost is happy"

β_2 : "There is a house in this village where only ghosts live"

β_3 : "There is an unhappy resident in this village"

Use resolution to prove β_3, assuming β_1 and β_2.

Ex. 3.20. Consider the sentences in Ex. 3.2. Use resolution to prove β_3, assuming β_1 and β_2.

4 INTRODUCTION TO PROLOG

• **PROgramming in LOgic**. Unlike imperative languages like C/C++ and Java, Prolog is a ***declarative*** language (in its philosophy), which means that the programmer states/declares *what* the question is, but not *not how* to compute the answer. A Prolog program is a set A of Prolog formulas. Usually the "execution" of A can have one of the two following forms:

1. *Input:* an atomic formula q with no variables.
   Output: `"yes"`, if $A \vdash_{\mathbf{P}} q$. `"no"`, otherwise.

2. *Input:* an atomic formula $q(X)$ with variable(s).
   Output: `"X=c`$_1$`;`$\cdots$ `X=c`$_k$`"`, if there are constants c$_1$,. . .,c$_k$ such that $A \vdash_{\mathbf{P}} q(c_i)$ for all i. `"no"`, otherwise.

In either of the above cases, the input is called a query. The expression $A \vdash_{\mathbf{P}} q$ means that Prolog's inference mechanism can prove q from A, or that q succeeds. If this is not the case then we say that q fails. Note that, if q succeeds then the atomic formula q is provable also in the sense of predicate logic, that is, $A \vdash q$, which implies that q is true. On the other hand, there is no mechanism in Prolog to infer that q is false in the logical sense. This situation is a consequence of the limitation of Prolog formulas – see below.

• **Prolog formulas and programs**. Recall from Section 2.5 that the general problem of deciding, for given formula q and assumptions A, whether $A \vdash q$, that is q is provable from A in Predicate Logic, is undecidable. For this reason, Prolog permits to use only a certain subset of the predicate formulas. In particular, *Prolog does not implement the logical negation connective '¬' and does not support functions.* Prolog allows two types of formulas:

- Atomic formulas without variables, called facts, such as `male(tom)` and `mother(sally,tom)`.

- Clauses of the form "β :- $\alpha_1, \ldots, \alpha_n$" where β and each α_i is an atomic formula possibly containing variables. Any variables occurring in the clause are assumed to be universally quantified. The above clause represents the formula

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta$$

with any quantifiers omitted. The atomic formula β is called the head of the clause.

A Prolog program A consists of a set of facts and a set of rules. A rule is a sequence of consecutive clauses with the same head, and describes the possible ways to prove the head of the clauses that constitute the rule. For this reason, the common head is called the goal of the rule. Given a query q, if q is a fact in the program A then q succeeds. Else Prolog tries to "match" q with the goal β of some clause "β :- $\alpha_1, \dots, \alpha_n$". In this case, q would succeed if all the subgoals $\alpha_1, \dots, \alpha_n$ succeed. If $q = q(X)$ involves a variable X then Prolog would try to find all constants in the formulas of the program A for which the query succeeds. Moreover, Prolog will perform the same process for all the clauses that appear in the rule whose goal is β.

- **Some syntax rules of GNU Prolog**.
  - Prolog is case-sensitive.
  - Comments start with `/*` and end with `*/`. There are also line comments that start with '`%`' and end at the end of the line.
  - Only the names of variables start with capital letters.
  - Constants can be numbers or atoms. The most frequent atoms are those that start with a lower case letter or are strings between single quotes.
  - All facts having the same predicate name must be written in consecutive lines, and similarly, all clauses of the same rule must be written in consecutive lines. For example, in the following program the three `parent` facts are written in consecutive lines.

Ex. 4.1. This is an example of a Prolog program from [8].

```
parent(mary,john).
parent(ann,mary).
parent(mary,susan).
female(mary).
```

```
mother(X,Y) :- parent(X,Y), female(X).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

The program consists of four facts and three rules such that the third rule consists of two clauses. The second rule consists of one clause, which would be written in Predicate Logic as follows

$$\forall x\, \forall y\, \forall z\, (\textsc{Parent}(x,z) \wedge \textsc{Parent}(z,y) \rightarrow \textsc{Grandparent}(x,y)).$$

When the query `grandparent(ann, john)` is given to Prolog, it sees that this query is not one of the four available facts. So Prolog tries to use the clause

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y)
```

by substituting `ann` for `X` and `john` for `Y`, and then trying to satisfy the subgoals `parent(ann,Z)` and `parent(Z,john)`. Both of these subgoals succeed when `mary` is substituted for `Z`.

• **The predicates <u>write</u> and <u>is</u>.** Consider evaluating an arithmetic expression such as `3*5-2`. Prolog can perform this evaluation via the predicate `is` that takes two parameters, a variable and the expression to be evaluated. The predicate `is` assigns the value of the expression to the variable and succeeds. For example,

```
X is 3*5-2, write(X)
```

assigns the value `13` to `X`, prints `X`, and succeeds. Note that attempting to use `write(3*5-2)` would simply print the expression `3*5-2`.

Ex. 4.2. The following rule can be used to print many copies of some expression `E`.

```
writeCopies(_, 0).
writeCopies(E, N) :- N>0, write(E), M is N-1,
                     writeCopies(E, M).
```

• **Watch out for infinite computations!** The way clauses are structured affects how Prolog attempts to answer a given query `q(X)`, that is, how Prolog ***searches*** through the clauses for the values `c` of `X` that would make `q(c)` succeed. If the clauses are not written carefully, then the search could end up in an infinite loop. For example, the following program might seem clear from a purely logical point of view.

```
edge(1,2).
edge(2,1).
edge(2,3).
edge(3,4).
path(X,X).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

The program specifies a graph with four edges and then defines when a path exists between two vertices. In particular, there is always an (empty) path from `X` to `X`, and a path from `X` to `Z` exists, if there is an edge from `X` to some `Y` and a path from `Y` to `Z`. Unfortunately, the query

```
path(1,3)
```

would cause Prolog to perform an infinite search. This is because Prolog would first try to satisfy `edge(1,Y)` and `path(Y,3)` using `Y=2`, and then `path(2,3)` using the subgoals `edge(2,Y)` and `path(Y,3)` with `Y=1`; then it would attempt to satisfy `path(1,3)`, which is the same as the original query! We refer the reader to [2], or any other Prolog book, for more details on Prolog's search mechanism.

• **The cut, '!', predicate**. There are cases where a particular clause "β :- $\alpha_1, \ldots, \alpha_n$" of a certain rule is processed, and we want to tell Prolog that no other clause of the rule should be processed. In this case, we use the predicate '!', called cut, as one of the subgoals, say α_i, of the clause. This predicate always succeeds when reached and, in this case, Prolog will not process another clause of the rule. Moreover, if the current processing of the clause fails, then no subgoal appearing in the clause before the cut (i.e. before α_i) will be tried again with some different values for variables.

Ex. 4.3. A good example of the cut predicate is given in Prolog Tutorial 8 at
`http://www.cs.nuim.ie/~jpower/Courses/PROLOG/`

```
grade(N, first) :- N>=70.
grade(N, two_1) :- N<70, N>=63.
grade(N, two_2) :- N<63, N>=55.
grade(N, third) :- N<55, N>=50.
grade(N, pass)  :- N<50, N>=40.
grade(N, fail)  :- N<40.
```

"While this will work, it is a little inefficient. The query `grade(75,G)` will answer `G=first` as expected but, once this has been satisfied, Prolog will go back to look for any other solutions. In order to do this it will process all of the other options, failing during the body of the rule in each case....To eliminate useless backtracking from the above, (and taking advantage of Prolog's order of execution) we can rephrase the program as:"

```
grade(N,first) :- N>=70, ! .
grade(N,two_1) :- N>=63, ! .
grade(N,two_2) :- N>=55, ! .
grade(N,third) :- N>=50, ! .
grade(N,pass) :- N>=40, ! .
grade(N,fail) :- N<40.
```

In many cases the cut predicate is used in connection with the predicate 'fail', which always fails when reached. A clause of the form

β :- $\alpha_1, \ldots, \alpha_m$, !, fail.

tells Prolog that, if all α_i's have succeeded, then (i) the clause fails, and (ii) no other clause with the same head will be processed.

Ex. 4.4. [Cut-Fail] A useful rule that tells us whether two expressions are different is the following.

```
different(X, X) :- !, fail.
different(_, _).
```

The query `different(ann,tom)` succeeds as follows. Prolog first attempts to match the query with the head of the first clause:

```
different(X, X)
```

As `X` cannot be assigned both `ann` and `tom`, this match fails and then Prolog attempts to match the query with `different(_, _)`. This clause matches any values for the two anonymous variables '_'.

On the other hand, the query `different(ann,ann)` fails as follows. Prolog first matches the query with `different(X, X)` and then attempts to satisfy the subgoals `!, fail`. The subgoal `!` succeeds but tells Prolog not to try another clause after it processes the current one. However, the subgoal `fail` fails and, therefore, the query fails as well.

• **List Processing**. A list in Prolog is a finite sequence of elements and is denoted using an expression of the form

$$[e_1, e_2, \ldots, e_n]$$

where each e_i is a variable, a number, an atom, or a list. If $n = 0$ then the list is empty: [ ]. Else, the element e_1 is called the head of the list, and the remainder $[e_2, ..., e_n]$ is the tail of the list (which could be empty). The Prolog expression `[H|T]` denotes a nonempty list whose head is H and whose tail is T. The following built-in List Processing Predicates are available in GNU-Prolog. The notation `pred/n` means that the predicate `pred` takes `n` parameters.
(see `http://www.gprolog.org/manual/gprolog.html`)

```
append/3
delete/3,
last/2
length/2
max_list/2, min_list/2, sum_list/2
member/2,
nth/3
reverse/2
setof/3
sublist/2
```

Ex. 4.5. Here is an implementation for the predicate
`member(X,L)` : "`X` is an element of the list `L`":

```
member(H,[H|_]).
member(X,[_|T]) :- member(X,T).
```

The first clause says that `H` is a member of any list whose head is `H`. The second clause says that `X` is a member of any list whose tail contains `X`.

Ex. 4.6. The formula `setof(X,` α`, L)` succeeds when α is a formula involving the variable `X`, and the list `L` consists of all unique values of `X` satisfying α. For example,

```
setof(X, parent(mary,X), L)
```

succeeds when the list `L` is the set of `mary`'s children.

• **The predicate <u>name</u>**. This is useful when we want to convert a Prolog atom (especially a string) to the list of characters that constitute this atom. In particular we have that `name(S, L)` succeeds exactly when L is the list of ASCII codes of the characters appearing in S. For example, the query

```
name('3ab4', [51,97,98,52]).
```

succeeds.

Ex. 4.7. The following rules implement the predicate

```
scratchA(S, T)
```

that is true when the string `T` results by replacing every '`A`' in the string `S` with '`X`'.

```
scratchLA([], []).
scratchLA([A|L], [X|M]) :- A=65, X=88, scratchLA(L,M).
scratchLA([B|L], [Y|M]) :- B=\=65, Y=B, scratchLA(L,M).

scratchA(S, T) :- name(S, L), scratchLA(L, M), name(T, M).
```

This works as follows. The clause `name(S, L)` computes the ASCII list `L` corresponding to `S` and then the clause `scratchLA(L, M)` works with ASCII lists. The ASCII codes of the characters '`A`' and '`X`' are `65` and `88`, respectively. In the end, the clause `name(T, M)` converts the ASCII list `M` to the string `T`.

5 REGULAR LANGUAGES & AUTOMATA

5.1 Regular Expressions (REXs)

Recall from Section 1.3 that a description method for a set of entities X is an onto function

$$\mathcal{L} : D \to X$$

such that D is the language used to describe the entities in X. We are interested again in describing languages, that is, X is a class of languages. This is important because a language could be infinite, but a description is always a finite string. Already we have seen the description method of context-free grammars:

$$\mathcal{L}(G) = \text{the language described (generated) by } G.$$

This method defines the class $\mathcal{L}(\text{CFG})$ of context-free languages. Here we define the class of regular languages, which are described by certain simple expressions that resemble algebraic expressions. Let's recall first some examples of operations on languages K, L:

$K \cup L$
KL
L^n
L^*
$KL^5(K \cup \{\lambda\}) = KL^5K \cup KL^5$.

• **Regular expressions (**REXs**)**. We assume a fixed, but arbitrary alphabet Σ. A regular expression is a string over the alphabet $\Sigma \cup \{(,),+,^*,\varnothing,\lambda\}$, where we assume that the symbols in $\{(,),+,^*,\varnothing,\lambda\}$ are not in Σ. More specifically the set REX of regular expressions is defined by the following grammar:

$$\begin{array}{lcl} R; & & \text{(the start variable)} \\ R & \to & \varnothing \mid \lambda \mid \sigma \ \text{ (for all } \sigma \in \Sigma) \\ R & \to & (R^*) \mid (R+R) \mid (RR) \end{array}$$

Here are some examples of regular expressions:

$$\varnothing,\ (0^*),\ ((0+1)^*),\ ((01)+(1^*)).$$

As in the case of algebraic expressions we shall omit parentheses to simplify the expressions. In doing so, we shall use the same precedence of operations as in arithmetic expressions, that is, *, catenation, '+'. Thus, $01 + 1^*$ is a shorthand for $((01) + (1^*))$.

• **Regular Languages**. A regular expression r describes a language, denoted $\mathcal{L}(r)$, as follows:

1. if $r = \varnothing$ then $\mathcal{L}(r) = \emptyset$
2. if $r = \lambda$ then $\mathcal{L}(r) = \{\lambda\}$
3. if $r = \sigma$ then $\mathcal{L}(r) = \{\sigma\}$
4. if $r = t^*$ then $\mathcal{L}(r) = (\mathcal{L}(t))^*$
5. if $r = (r_1 r_2)$ then $\mathcal{L}(r) = \mathcal{L}(r_1)\mathcal{L}(r_2)$
6. if $r = (r_1 + r_2)$ then $\mathcal{L}(r) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

A regular language is defined to be any language that can be described by a regular expression, that is, L is regular if there is a regular expression r such that $L = \mathcal{L}(r)$.

• **Extending Regular Expressions**. It is customary to allow the following two notational extensions to regular expressions:

1. For any $k \geq 0$ and regular expression r, r^k specifies the language $\mathcal{L}(r)^k$ consisting of all the words of the form $w_1 \cdots w_k$, with each $w_i \in \mathcal{L}(r)$.
2. For any regular expression r, r^+ specifies the language $\mathcal{L}(r)^*$ without the empty word λ, that is, $\mathcal{L}(r^+) = \mathcal{L}(r^*) - \{\lambda\}$.

• **The class of Regular languages**. Recall REX is the set of all regular expressions with respect to a certain alphabet Σ. Each of these expressions r describes a language $\mathcal{L}(r)$. Taking all these languages together defines the class of regular languages:

$$\mathcal{L}(\text{REX}) = \{\mathcal{L}(r) \mid r \text{ is in REX}\}.$$

Ex. 5.1. Let $\Sigma = \{0, 1\}$. Is the language

$$\{00w1^n \mid |w| = 5,\, n \in \mathbb{N}_0\}$$

regular? The answer is yes by using the regular expression $00(0+1)^5 1^*$. Indeed, we have:

$\mathcal{L}(00(0+1)^5 1^*) =$
$\mathcal{L}(0)\mathcal{L}(0)\mathcal{L}((0+1)^5)\mathcal{L}(1^*) =$
$\{0\}\{0\}(\{0\} \cup \{1\})^5\{1\}^* =$
$\{00\}\{0,1\}^5\{1\}^* =$
$\{00w1^n \mid |w| = 5,\, n \in \mathbb{N}_0\}.$

• **Equivalent Regular Expressions**. Two regular expressions r_1, r_2 could be different, namely $r_1 \neq r_2$, but they could describe the same language, namely $\mathcal{L}(r_1) = \mathcal{L}(r_2)$. In this case we write

$$r_1 \equiv r_2.$$

Here is a list of useful equivalences

$$\begin{array}{rcl}
r_1 + r_2 & \equiv & r_2 + r_1 \\
r^* r & \equiv & r r^* \equiv r^+ \\
\lambda + r^+ & \equiv & r^* \\
r(r_1 + r_2) & \equiv & r r_1 + r r_2 \\
\lambda r & \equiv & r\lambda \equiv r \\
(r^*)^* & \equiv & r^* \\
r^* r^* & \equiv & r^* \\
\varnothing + r & \equiv & r + \varnothing \equiv r \\
\varnothing^* & \equiv & \lambda
\end{array}$$

Ex. 5.2. We show that $r(r_1 + r_2) \equiv r r_1 + r r_2$:

$\mathcal{L}(r(r_1 + r_2)) =$
$\mathcal{L}(r)(\mathcal{L}(r_1) \cup \mathcal{L}(r_2)) =$
$\{uv \mid u \in \mathcal{L}(r), v \in \mathcal{L}(r_1) \cup \mathcal{L}(r_2)\} =$
$\{uv \mid u \in \mathcal{L}(r), v \in \mathcal{L}(r_1)\} \cup \{uv \mid u \in \mathcal{L}(r), v \in \mathcal{L}(r_2)\} =$
$\mathcal{L}(r r_1) \cup \mathcal{L}(r r_2) =$
$\mathcal{L}(r r_1 + r r_2).$

5.2 UNIX Regular Expressions

`http://www.gnu.org/software/grep/doc/grep.html`

The UNIX operating system provides an implementation of regular expressions, which has been adopted by other operating systems as well. Here we shall give a partial definition of UNIX regular expressions. First, we define the sets

CH = all characters except the new line character
SP = special chars = `. [ ] \ * + ? ^ - $ " | ( )`

• **Definition (partial)**. A UNIX regular expression p is a string over the alphabet CH, and has the following characteristics:

1. If p contains no special characters then p represents the word p.

2. If p contains $\backslash\sigma$, where σ is any character in CH, and $\backslash\sigma$ does not occur between any pair [ ], then $\backslash\sigma$ represents the character σ. In this case we say that σ is quoted.

3. If p contains [**C**], where **C** is a character class, then [**C**] represents the set of all characters specified by **C**. In particular, **C** is any string of characters other than], unless] occurs at the beginning of **C**, and represents exactly the set of these characters except in the following special cases:

   - a caret ^ in the beginning of **C** specifies the set of all characters that are not in the part of **C** following ^.
   - a hyphen - between two characters σ_1 and σ_2 is used to specify the range of characters from σ_1 to σ_2, according to the ASCII order.

4. If p contains the character '.' outside of any pair [ ], then this character represents the set CH (i.e., the period character matches any character in CH).

5. If p contains q* outside of any pair [ ], where q is a single character or an expression of the form (r) or [r], then q* represents the set of words that can be formed by concatenating zero or more words from the set represented by q.

6. If in the above rule we have that p contains q+, then we have one or more concatenations of words from the set represented by q. Similarly, if p contains q?, then q? represents zero or one word from the set of words represented by q.

7. If p contains (q|r) outside of any pair [ ], then (q|r) represents all words represented by q union those represented by r.

• **Command egrep**. UNIX regular expressions are used in egrep:

```
egrep "p" F                 egrep "^p" F
egrep "p$" F                 egrep "^p$ F
```

where `F` is a text file, and `p` is a UNIX regular expression.

1. In the form `"p"`, egrep prints all lines of F ***containing*** a string of the form specified by `p` (that is, a string that belongs to the language described by `p`).

2. In the form `"^p"`, egrep prints all lines ***starting*** with a string that belongs to the language described by `p`.

3. In the form `"p$"`, egrep prints all lines ***ending*** with a string that belongs to the language described by `p`.

4. In the form `"^p$"`, egrep prints all lines that are ***equal to*** a string that belongs to the language described by `p`.

Ex. 5.3. Here are some egrep examples.

`egrep "bb" F`: prints all lines in `F` containing the string `bb`.

`egrep "^a.*3$" F`: prints all lines in `F` starting with `a` and ending with `3`.

`egrep ""  F`: prints all lines in `F`

• **Command sed**. `sed -r 's/E/str/[g]' F`
Here `E` is a UNIX regular expression, `str` is a string, g is an optional flag, and `F` is a text file. The command works as follows:

```
while NOT at end of file F {
  get next line L from F;
  if (L contains a string t matching E) {
    modify L by replacing the first t with str, OR all
           occurrences of t if the flag g is present;
    output the modified line L;
  }
  else output the line L;
}
```

Notes: 1. If str contains &, then sed replaces & with the string t that matched the regular expression E.
2. When a set of overlapping strings in L matches E, then sed selects the longest match.
3. Instead of the three delimiters `/.../.../` one can use `"..."..."`

Ex. 5.4.

`sed -r ’s/ +/ /g’ F`: outputs all lines of F with sequences of one or more spaces replaced with one space.

`sed -r ’s/^./  &/’ F`: outputs all lines of F with the nonempty ones indented by 2 spaces.

Exercises of Sections 5.1, 5.2

Ex. 5.5. Show that the following language over the alphabet $\Sigma_2 = \{0,1\}$ is regular

$$\{x0^n y(10)^m \mid x \in \Sigma_2^* \wedge y \in \Sigma_2^+ \wedge n, m \geq 120\}.$$

Ex. 5.6. Give regular expressions describing the following languages.

All words $w \in \{a,b\}^*$ whose length is even.

All words $w \in \{a,b\}^*$ containing exactly one occurrence of ab.

All words $w \in \{a,b,c\}^*$ containing no occurrence of ab.

Ex. 5.7. Indicate whether each of the following statements is correct.

$$r_1(r_1+r_2)^* + r_2(r_1+r_2)^* \quad \equiv \quad (r_1+r_2)^+$$

$$\begin{aligned} r_1^* + r_2^* &\equiv r_2^* + r_1^* \\ r_1^* + r_2^* &\equiv (r_1 + r_2)^* \\ r + r &\equiv r \end{aligned}$$

Ex. 5.8. Give `egrep` expressions to (i) print all lines in the file `F` containing a string of two or more consecutive b's; (ii) print all the empty lines in `F`.

Solution.

```
egrep "b(b)+" F
egrep "^$" F
```

□

Ex. 5.9. Give an `egrep` expression to print from the file `F` each line that consists of a, possibly signed, integer allowing spaces before and after the integer.

Ex. 5.10. Show that every finite language is regular.

Solution. Let F be a finite language. If F is empty then it is described by the regular expression $r = \varnothing$, that is, $F = \mathcal{L}(r)$. If F is not empty then $F = \{u_1, \dots, u_n\}$, for some $n \in \mathbb{N}$. In this case, consider the regular expression $r = u_1 + u_2 + \cdots + u_n$. Then

$$\mathcal{L}(r) = \mathcal{L}(u_1) \cup \cdots \cup \mathcal{L}(u_n) = \{u_1\} \cup \cdots \cup \{u_n\} = F.$$

Thus, $F = \mathcal{L}(r)$ for some regular expression r and, therefore, F is a regular language. □

5.3 Finite Automata (DFAs, NFAs)

Finite automata constitute another method for describing languages. Informally, we define a finite automaton by drawing a labeled graph (state transition diagram). Such a graph consists of a set of nodes, called states, and a set of edges between states, called transitions. Each transition is labeled with a letter of the alphabet. One of the states is special and called the start, or initial state. This is denoted using a short incoming arrow without an origin state. Some of the states are called final, or accepting, and denoted by double lines. The automaton represents the set of all words that are formed along the paths of the graph from the start state to a final state. These words

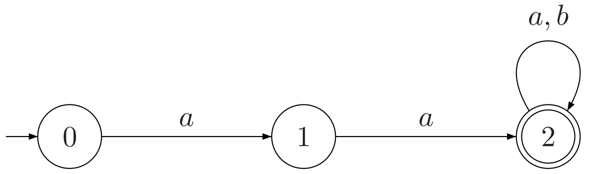


Figure 4: A finite automaton accepting $\{aa\}\Sigma^*$.

constitute the language accepted, or recognized, by the automaton. The automaton in Fig. 4 accepts the language $\{aa\}\{a,b\}^*$. For example, the word *aaba* is accepted because it is formed by the labels a, a, b, a in the path

$$(0, a, 1), (1, a, 2), (2, b, 2), (2, a, 2) \tag{1}$$

such that the path begins with the initial state 0 and ends with 2, which is a final state. On the other hand, *aba* is not accepted, as there is no path from state 0 to a final state having a, b, a as labels.

• **Automata as computing machines (or decision programs)**. The important aspect of an automaton M is that it can be viewed as a computing machine that processes input words and decides whether or not to accept these words. In particular, given any input word $w = \sigma_1 \cdots \sigma_n$ at the initial state q_0, the automaton ***reads*** (or consumes) each symbol $\sigma_1, \sigma_2, \ldots$ by following the available transitions

$$(q_0, \sigma_1, q_1), (q_1, \sigma_2, q_2), \ldots$$

This computation ends when all symbols are read, or there are no available transitions that can be used to read the entire input – this happens, for instance, when the automaton in Fig. 4 processes the input *aba*. In the former case (when all input symbols are read) we have a computation as shown in (1) above when the input is *aaba*. In this case, as the last state in the computation is a final state, the automaton M accepts *aaba*, or we can say that M returns YES:

$$M(aaba) = \text{YES}.$$

On the other hand, it is easy to see that $M(aba) = \text{NO}$.

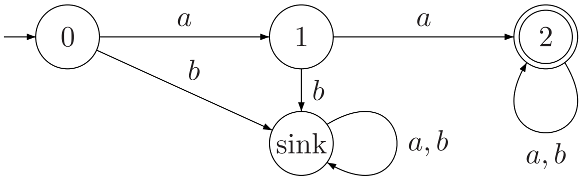


Figure 5: Complete automaton accepting $\{aa\}\Sigma^*$.

5.4 Deterministic Automata (DFAs)

A finite automaton is called deterministic if no state has two different transitions with the same letter going out of the state. The automaton in Fig. 4 is deterministic. In practice, this means that there is always exactly one path in the automaton that one can use to recognize a word of the language.

• **Complete automaton**. This is an automaton in which, for each state, every letter of the alphabet appears on a transition from this state. The automaton of Fig. 4 is not complete. We can complete an automaton without affecting the language it recognizes by introducing a ***sink*** state (or ***trap*** state) and transitions to it from all other states with the missing alphabet letters. This is shown on Fig. 5.

• **Rigorous definition**. A (complete) deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, q_0, \delta, F)$ such that

Q is a finite nonempty set of states,
Σ is an alphabet,
$q_0 \in Q$ is the start state,
$F \subseteq Q$ is the set of final (accepting) states, and
$\delta : Q \times \Sigma \to Q$ is the transition function.

The transition function δ takes as input a state q and a letter σ, and returns the next state $\delta(q, \sigma)$. The closure δ^* of δ extends δ by mapping a state and a word to a state such that:

$$\delta^*(q, \lambda) = q \text{ and } \delta^*(q, \sigma x) = \delta^*(\delta(q, \sigma), x),$$

that is, $\delta^*(q, w)$ returns the state that will be reached after the word w has been read starting at state q. The language $\mathcal{L}(M)$ accepted,

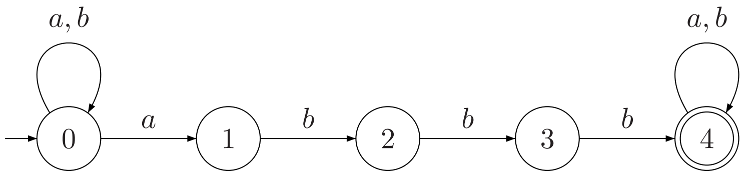


Figure 6: Nondeterministic automaton accepting $\Sigma^* abbb\Sigma^*$.

or recognized, by M is

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

Ex. 5.11. For the automaton of Fig. 5, the transition function δ is as follows:

$\delta =$

	a	b
0	1	sink
1	2	sink
2	2	2
sink	sink	sink

We have: $\delta(0, a) = 1 = \delta^*(1, \lambda)$ and $\delta^*(0, aaba) = \delta^*(1, aba) = \delta^*(2, ba) = \delta^*(2, a) = \delta^*(2, \lambda) = 2$.

• **The class of** DFA **languages**. The class of DFA languages consists of all languages that are accepted by deterministic finite automata:

$$\mathcal{L}(\text{DFA}) = \{\mathcal{L}(M) \mid M \text{ is a DFA}\}.$$

5.5 Nondeterministic Automata (NFAs)

Here we allow situations where two or more transitions with the ***same*** label go out of one state. Thus, at such a state, when the next letter of the input word matches two or more transitions the automaton works nondeterministically, that is, it chooses/guesses one transition and continues from there. Moreover, we assume that for every input word w, if there is at least one accepting path (from the start state to a final state), then the automaton will always guess an accepting path for w!

Ex. 5.12. Consider the word *abbb*, which we consider to be a pattern, and another longer word t, which we consider to be a text. The question is whether the pattern *abbb* occurs in the text t. Note that the set of all words containing the pattern *abbb* is equal to

$$\Sigma^* abbb \Sigma^*.$$

Thus, our question is equivalent to whether the given text t is an element of $\Sigma^* abbb \Sigma^*$. A nondeterministic automaton accepting the language $\Sigma^* abbb \Sigma^*$ is given in Fig. 6.

• **Rigorous definition**. A (nondeterministic) finite automaton (NFA) is a 5-tuple $M = (Q, \Sigma, q_0, T, F)$ such that

1. Q is a finite nonempty set of states,
2. Σ is an alphabet,
3. $q_0 \in Q$ is the start state,
4. $F \subseteq Q$ is the set of final (accepting) states,
5. T is the finite set of transitions, where a transition is a triple $(p, \sigma, q) \in Q \times \Sigma \times Q$.

A transition (p, σ, q) specifies that the automaton can go from state p to state q when the current input symbol is σ. In this case, we say that the automaton reads, or consumes, the symbol σ. A ***computation*** of the automaton is a sequence of transitions of the form

$$(p_0, \sigma_1, p_1),\ (p_1, \sigma_2, p_2),\ \ldots,\ (p_{n-1}, \sigma_n, p_n).$$

If p_0 is the start state and p_n is a final state then we have an accepting computation. The language $\mathcal{L}(M)$ accepted, or recognized, by M is the set of all words $w = \sigma_1 \cdots \sigma_n$ such that the word is formed in the transitions of an accepting computation of M, as shown above. In this case we say that the automaton returns YES on input w. In general, we have $M(w) \in \{\text{YES, NO}\}$ such that

$$M(w) = \text{ YES iff } M \text{ has an accepting computation on input } w.$$

Ex. 5.13. We show three different computations of the NFA in Fig. 6 on the same input word *abbbaabbb* such that the first two are accepting computations.

1st comput.	2nd comput.	3rd comput.
0, a, 1	0, a, 0	0, a, 0
1, b, 2	0, b, 0	0, b, 0
2, b, 3	0, b, 0	0, b, 0
3, b, 4	0, b, 0	0, b, 0
4, a, 4	0, a, 0	0, a, 0
4, a, 4	0, a, 1	0, a, 0
4, b, 4	1, b, 2	0, b, 0
4, b, 4	2, b, 3	0, b, 0
4, b, 4	3, b, 4	0, b, 0
accept	accept	reject

• **The class of** NFA **languages**. The class of NFA languages consists of all languages that are accepted by deterministic finite automata:

$$\mathcal{L}(\text{NFA}) = \{\mathcal{L}(M) \mid M \text{ is a NFA}\}.$$

• **Method: Designing DFAs**. Given a language L, we want to design a DFA recognizing L. One approach is to assign a certain meaning to each state of the DFA such that the transitions can be defined by simply preserving the meaning of the states. In many cases, the meaning of a state is the set of words accepted when the state is viewed as final. For example, a DFA accepting all words over $\{a, b\}$ with an even number of a's has two states:

ev : "all words with even number of a's"
od : "all words with odd number of a's"

such that ev is the only final state. The transition function is very simple:

$$\delta(\text{ev}, a) = \text{od}, \quad \delta(\text{ev}, b) = \text{ev}, \quad \delta(\text{od}, a) = \text{ev}, \quad \delta(\text{od}, b) = \text{od}.$$

Equivalently, we can write the above as a set of four transitions:

$$\{(\text{ev}, a, \text{od}), (\text{ev}, b, \text{ev}), (\text{od}, a, \text{ev}), (\text{od}, b, \text{od})\}.$$

A state diagram is shown in Fig. 7

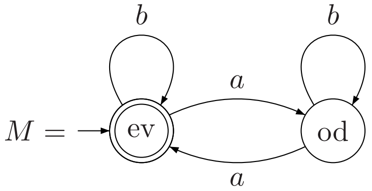


Figure 7: DFA accepting all words with an even number of a's.

- **Quick Facts**.

1. Every finite language is a DFA language.
2. Every DFA can be viewed as an NFA. Thus every DFA language is also an NFA language.
3. The language recognized by an automaton is empty iff there is no path from the start state to a final state.
4. The language recognized by an automaton contains λ iff the start state is also a final state.
5. The language recognized by an automaton is infinite iff there is a useful cycle (loop) in the automaton, that is, there is a state and a path that starts and ends at that sate, such that this state can be reached from the start state and can reach a final state.

Exercises of Sections 5.3, 5.5

Ex. 5.14. Draw the state diagram of a DFA accepting all words over $\{a, b\}$ having no occurrence of bb.

Ex. 5.15. Draw the state diagram of a DFA accepting all words w in $\{a, b\}^*$ with the following property: each b that occurs in w (if any) must be followed immediately by at least one a. Note that a regular expression describing this language is $(a + ba)^*$.

Ex. 5.16. Draw the state diagram of a DFA accepting all expressions for decimal integers. These consist of decimal digits with an

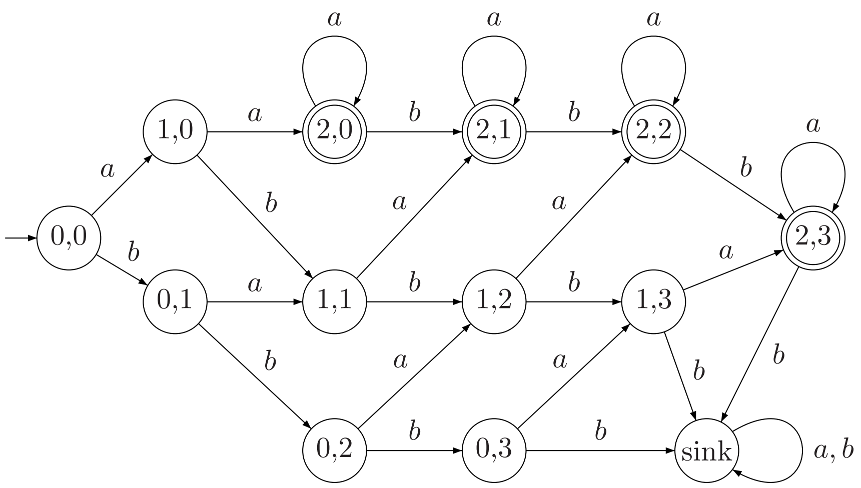


Figure 8: DFA accepting words with ≥ 2 a's and ≤ 3 b's.

optional sign and any number of spaces before and after the integer. For example, your DFA should accept the following three strings

```
"+123 "   "  34"   " -5   "
```

Ex. 5.17. Draw the state diagram of a DFA accepting all words over $\{a, b\}$ containing at least two a's and at most three b's.

Solution. We shall use 13 states as follows.
– (i, j) : "i a's and j b's were read, with $0 \leq i \leq 1,\ 0 \leq j \leq 3$".
– $(2, j)$: "j b's and at least 2 a's were read, with $0 \leq j \leq 3$".
– sink : "a sink state".
– Start state = (0,0).
– Final states = {(2,j): j = 0,1,2,3}.
The set of transitions can be defined easily based on the meaning of the states. For example, $((1, 2), a, (2, 2)$ is a transition that says, if the DFA is at state (1,2) – hence, it has read one a and two b's – and the next input is a then the DFA goes to state (2,2). The state diagram is shown in Fig. 8. □

Ex. 5.18. Draw the state diagram of a DFA accepting all words over $\{a, b\}$ containing an even numbers of a's and an even number of b's.

Ex. 5.19. Explain why every finite language is a DFA language.

5.6 A simple DFA Implementation

Consider a DFA $M = (Q, \Sigma, q_0, \delta, F)$. In our implementation, we assume that the alphabet is $\Sigma = \{0, 1, \ldots, n-1\}$ and the set of states is $Q = \{0, 1, \ldots, m-1\}$, with 0 being the initial state. The transition function δ is implemented as a two dimensional (dynamic) array `delta_arr[][]` such that the value $\delta(q, i)$ can be found by accessing the array entry `delta_arr[q][i]`, which takes constant time $O(1)$, as an array is a random access data structure. Now, if we wish to run the automaton on some input word `t`, we shall need a function `run(t)`. This will access the array once for every letter in `t` and, therefore, this takes time $O(n)$, where n is the length of the input string `t`. In particular, here is how `run(t)` works, where the variable `c` is used to keep track of the current state.

```
c = 0;
for (i=0; i<n; i++)  c = delta_arr[c][t[i]];
if (c is final) return YES;
else return NO;
```

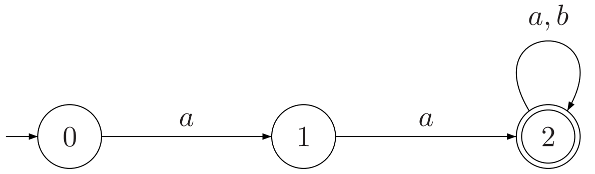
An important application of DFAs is in the pattern-matching problem.

Theorem 5.1. *The pattern-matching problem can be decided in time $O(n + k)$, where n is the length of the text and k is the length of the pattern.*

Proof idea. Let t be the given text of length n and p be the given pattern of length k. We assume that both are given as strings (in the programming sense). Obviously, p occurs in t iff the word t is an element of the language $\Sigma^*\{p\}\Sigma^*$. The main steps of the algorithm are as follows:

```
1. Construct a DFA M accepting Sigma*{p}Sigma*
2. Run M using  t  as input
3. If t is accepted return YES, else return NO
```

From the discussion in this section we know that Step 2 can be performed in time $O(n)$. It can be shown that Step 1 can be performed in time $O(k)$ – see for example, [4]. □



5.7 NFA Implementation in Prolog

Here the automaton is defined as a set of Prolog facts specifying the start state, the final states and the transitions. Moreover, it is convenient to have a predicate `states(N)` defining the number `N` of states in the automaton. Sometimes it might be useful to include the predicate `alphabet(`L`)` defining the alphabet as a list L. The automaton in the picture is defined as follows:

```
  states(3).
  alphabet([a,b]).
  start(0).
  final(2).
  tr([0,a,1]).
  tr([1,a,2]).
  tr([2,a,2]).
  tr([2,b,2]).
```

• **Predicates to run the** NFA. We implement words as Prolog lists. In particular the empty word is implemented as the empty list `[]`. Predicate `runNFA(W)` runs the current automaton on input `W`. The predicate `accept(Q,W)` succeeds when the automaton accepts the word `W` when it starts at state `Q`. The predicate `runNFA(F,W)` first reads the file `F` containing the definition of the automaton and then tests whether the automaton accepts `W`.

```
accept(Q,[]) :- final(Q).
accept(Q,[S|T]) :- tr([Q,S,R]), accept(R,T).

runNFA(W) :- start(Q), accept(Q,W), !.
runNFA(F,W) :- consult(F), runNFA(W).
```

• **Decision problems**. Here are some decision problems considered in automaton theory.

Emptiness. Given an automaton M, decide whether $\mathcal{L}(M) = \emptyset$.

Finiteness. Given an automaton M, decide whether $\mathcal{L}(M)$ is infinite.

Empty word. Given an automaton M, decide whether $\lambda \in \mathcal{L}(M)$

A useful predicate in this context is `pathAut(Q1,Q2,N)` : "there is a path of length at most `N` from state `Q1` to state `Q2`, where `N` is a nonnegative integer." Here is a definition of this predicate.

```
pathAut(Q1,Q1,N) :- N>=0, !.
pathAut(Q1,Q2,N) :- N>0, tr([Q1,S,Q2]), !.
pathAut(Q1,Q2,N) :- N>1, tr([Q1,S,Q]),
                    M is N-1,
                    pathAut(Q,Q2,M), !.
```

The term `N` is necessary to bound the amount of computation.

Ex. 5.20. [Empty word] We define the predicate `emptyword(F)` : "the language accepted by the automaton defined in the file `F` contains the empty word".

```
emptyword(F) :- consult(F), start(Q), final(Q).
```

Ex. 5.21. [Emptiness] We define the predicate `nonempty(F)` : "the language accepted by the automaton defined in the file `F` is not empty".

```
nonempty(F) :- consult(F), start(Q), final(R),
            states(N), pathAut(Q,R,N), !.
```

Ex. 5.22. [Finiteness] The following predicates are useful in deciding the finiteness of an NFA language: `reachable(Q)` : "there is a path from the start state to state `Q`," `tofinal(Q)` : "there is a path from state `Q` to a final state," `useful(Q) :- reachable(Q), tofinal(Q)`. It can be shown that the language of an NFA is infinite iff there is a useful state `Q` and a path from `Q` to `Q`.

```
reachable(Q) :- start(Q).
reachable(Q) :- tr([P,_,Q]), states(N),
                start(Q0), pathAut(Q0,P,N).
...
```

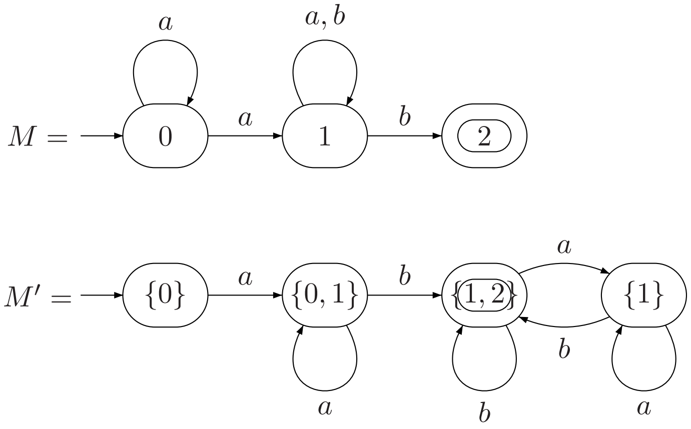


Figure 9: Subset Construction as in Theorem 5.2. The sink state is omitted.

5.8 Basic Theorems on Automata and λ-NFAs

In this section we state a few basic results of automata theory. In particular the fact that the classes of NFA, DFA and regular languages coincide. Thus, we can speak about regular languages with the understanding that there are three ways to describe these languages. We also state a few basic results about ***closure properties*** of regular language operations. For example, we show that regular languages are closed under the intersection operation $\cap$, that is, for every regular languages L, L' we have that $L \cap L'$ is a regular language.

Theorem 5.2. *Every* NFA *language is a* DFA *language and vice versa:*

$$\mathcal{L}(\text{DFA}) = \mathcal{L}(\text{NFA}).$$

Proof. The direction $\mathcal{L}(\text{DFA}) \subseteq \mathcal{L}(\text{NFA})$ is obvious as every DFA is a particular type of NFA. The direction $\mathcal{L}(\text{NFA}) \subseteq \mathcal{L}(\text{DFA})$ can be shown using the Subset Construction as follows. Given NFA

$$M = (Q, \Sigma, q_0, T, F),$$

we construct a DFA $M' = (Q', \Sigma, \{q_0\}, \delta', F')$ accepting the same language. The states $\varphi \in Q'$ of M' are ***subsets of*** Q. The transition function δ' of M' is such that $\delta'(\varphi, \sigma)$ is the set of states q in all transitions of the form (p, σ, q) for every state $p \in \varphi$; that is, the next state $\delta'(\varphi, \sigma)$ of M' consists of all states that can be reached in M from any state in φ with input σ. The final states of M' are all the subsets of Q that contain at least one final state of M. The process starts with initializing the current state $\varphi = \{q_0\}$. Then new states are derived by completing the transitions of the current state, that is computing $\delta'(\varphi, \sigma)$, for all symbols σ. This process stops when no new states can be derived. Finally, the resulting DFA gets completed by adding a sink state. □

Theorem 5.3. *If L and L' are* NFA *languages then so is $L \cap L'$.*

Proof. As L and L' are NFA languages, there are NFAs M, M' accepting L, L', respectively. We construct an NFA $M \cap M'$ accepting $L \cap L'$, as follows.

- The states of the automaton $M \cap M'$ are pairs (q, q') of states from M and M', respectively.
- The start state is (q_0, q_0'), made of the start states of M, M'.
- The final states are all pairs (f, f') of final states from the two NFAs.
- For any two transitions with the ***same*** label, (q, σ, p) from M and (q', σ, p') from M', the tuple $((q, q'), \sigma, (p, p'))$ is a transition of the new automaton.

Fig. 10 demonstrates the above construction.

For the correctness of the construction, we have to show that the language $\mathcal{L}(M \cap M')$ accepted by $M \cap M'$ is exactly $L \cap L'$. Equivalently, we show that a computation of $M \cap M'$ accepts a word $\sigma_1 \cdots \sigma_n$ iff there are two computations of M and M', both accepting the word $\sigma_1 \cdots \sigma_n$.

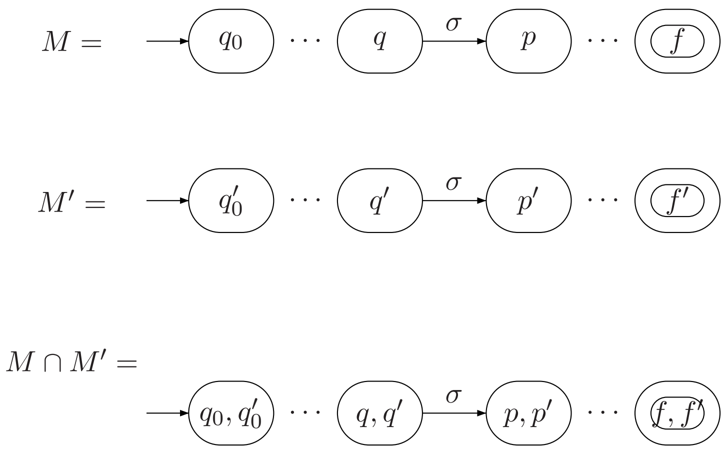


Figure 10: Cartesian Product Construction in Theorem 5.3

So first assume that

$$\mathcal{P} = ((q_0, q'_0), \sigma_1, (q_1, q'_1)), \ldots, ((q_{n-1}, q'_{n-1}), \sigma_n, (q_n, q'_n))$$

is a computation of $M \cap M'$ accepting the word $\sigma_1 \cdots \sigma_n$ – hence, both q_n and q'_n are final states in M and M', respectively. By the above construction, for each transition $((q_{i-1}, q'_{i-1}), \sigma_i, (q_i, q'_i))$ in the computation $\mathcal{P}$, there must be two transitions (q_{i-1}, σ_i, q_i) and $(q'_{i-1}, \sigma_i, q'_i)$ in M and M', respectively. Thus, when we consider all these transitions for $i = 1, \ldots, n$, we can form

$$\mathcal{C} = (q_0, \sigma_1, q_1), \ldots, (q_{n-1}, \sigma_n, q_n),$$

which is an accepting computation of M, and

$$\mathcal{C}' = (q'_0, \sigma_1, q'_1), \ldots, (q'_{n-1}, \sigma_n, q'_n),$$

which is an accepting computation of M'. Hence, the word $\sigma_1 \cdots \sigma_n$ must be in both $L = \mathcal{L}(M)$ and $L' = \mathcal{L}(M')$, as required.

Conversely, suppose that there are two computations of M and M' accepting the same word $\sigma_1 \cdots \sigma_n$. Then, these computations have the form of $\mathcal{C}$ and $\mathcal{C}'$ shown above. By the construction of $M \cap M'$, for each pair of transitions (q_{i-1}, σ_i, q_i) and $(q'_{i-1}, \sigma_i, q'_i)$ in $\mathcal{C}$ and $\mathcal{C}'$, respectively, we have that $((q_{i-1}, q'_{i-1}), \sigma_i, (q_i, q'_i))$ is a transition of $M \cap M'$. Hence we can form a computation of $M \cap M'$ of the form $\mathcal{P}$ shown above such that $\mathcal{P}$ accepts the word $\sigma_1 \cdots \sigma_n$, as required. □

Note: The type of construction in Theorem 5.3 is known as *(Cartesian) Product* construction.

Theorem 5.4. *If L and L' are* NFA *languages then so are $\overline{L}$ and $L' - L$.*

Proof idea. We are given automata M, M' recognizing L, L', respectively. If M is not a (complete) DFA, use Theorem 5.2 to make it so. Then construct the automaton $\overline{M}$ that is exactly the same as M except that every final state of M is non-final in $\overline{M}$, and vice versa. We have that

$$\mathcal{L}(\overline{M}) = \overline{\mathcal{L}(M)}.$$

To show that $L' - L$ is an NFA language, observe that

$$L' - L = L' \cap \overline{L}.$$

Hence, we can use Theorem 5.3 and the above method to construct the NFA $M' \cap \overline{M}$ such that

$$\mathcal{L}(M' \cap \overline{M}) = L' - L.$$

□

Ex. 5.23. The previous theorems could be useful when designing automata for languages whose words satisfy multiple constraints. For example, consider the language L in Ex. 5.15. If we want to describe all L-words that contain no pattern aaa, then the set of these words is $L - L'$, where $L' = \Sigma^*\{aaa\}\Sigma^*$. So if M and M' are DFAs accepting L and L', respectively, then we can use the above theorem to construct a DFA for $L - L'$.

• **More closure properties via λ-NFAs**. It can be shown that LL', L^* and L^R are NFA languages when L and L' are. The technique

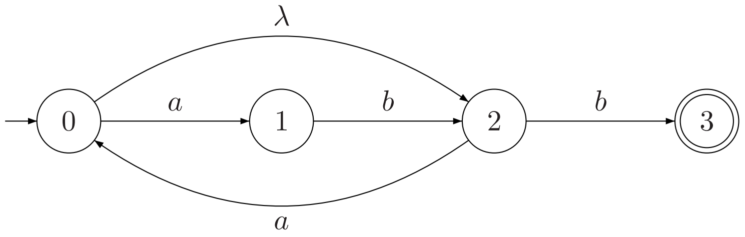


Figure 11: A λ-NFA recognizing the language $(\lambda + ab)(a + aab)^*b$.

to show this uses an extended type of automaton, the λ-NFA. The definition is exactly the same as that of NFAs with the additional capability that some of the transitions could be $\underline{\lambda\text{-transitions}}$:

$$(q, \lambda, p).$$

Then, when the automaton is at state q it could go to state p ***without reading*** the next input symbol. Fig. 11 shows an example of a λ-NFA M accepting the language $(\lambda+ab)(a+aab)^*b$. In particular, the word $abab$ is accepted by M via the following accepting computation:

$$(0, a, 1), (1, b, 2), (2, a, 0), (0, \lambda, 2), (2, b, 3).$$

Note that the 4th transition in the above computation is a λ-transition where the machine goes to state 2 without reading the next input symbol b. The next theorem says that λ-NFAs do not accept more languages than the ordinary NFAs do.

Theorem 5.5. $\mathcal{L}(\lambda\text{-NFA}) = \mathcal{L}(\text{NFA})$.

Theorem 5.6. *If L and L' are* λ-NFA *languages then so are $L \cup L'$, LL' and L^*.*

Proof idea. Consider λ-NFAs M, M' accepting L, L' respectively. We shall construct λ-NFAs $M \cup M'$, MM' and M^* accepting $L \cup L'$, LL' and L^*, respectively. The construction of $M \cup M'$ is very simple: it is made of M and M', a new initial state s and two λ-transitions from s to the initial states of M and M'.

The construction of MM' is simple as well: it is made of M and M' such that the intial state of MM' is that of M, the final states of

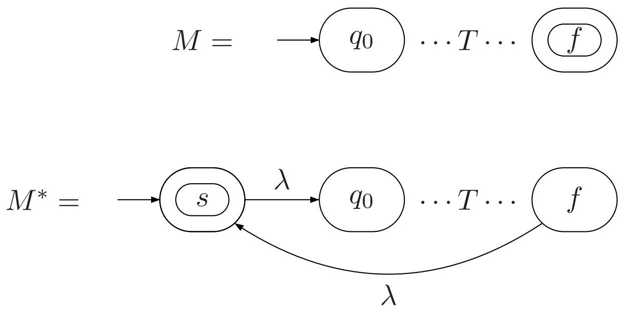


Figure 12: Construction in Theorem 5.6

MM' are those of M', and every final state of M has a λ-transition to the start state of M'.

The construction of M^* is as follows – see also Fig. 12: Consider the automaton M and add a new state s which is the start state of M^* and also the only final state of M^*. Add the following λ-transitions: (s, λ, q_0), where q_0 is the start state of M, and (f, λ, s) for every final state of M. □

Theorem 5.7. *If L is a* λ-NFA *language then so is L^R.*

Proof idea. Given a λ-NFA M accepting L, we construct a λ-NFA M^R accepting L^R as follows – see also Fig. 13: Consider the NFA M and add a new state s, which is the start state of M^R. Reverse every transition of M: if (q, σ, p) is a transition of M then (p, σ, q) is added in M^R. The start state q_0 of M is the only final state of M^R. Add the following λ-transitions: (s, λ, f) for every final state f of M. One verifies that all words accepted by M^R are reverses of those accepted by M, and vice versa. □

5.9 REX **matching via** NFA **membership**

In this section we establish another fundamental result, that the class of NFA languages coincides with the class of regular languages. In other words, for every regular expression r we can construct an NFA M such that $\mathcal{L}(M) = \mathcal{L}(r)$ (and vice versa) – see Theorem 5.8.

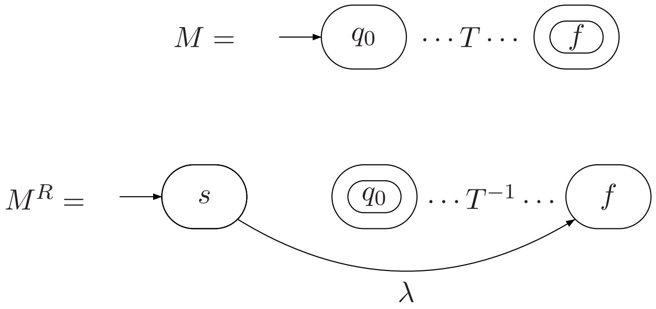


Figure 13: Construction in Theorem 5.7

A consequence of this is that the REX matching problem can be reduced to the NFA membership problem. More specifically, consider the UNIX command

```
egrep "r" f
```

and a line `t` of the text file `f`. We want to decide whether the line 'matches' the regular expression `r`, that is, whether `t` contains a string that belongs to $\mathcal{L}(r)$. This is equivalent to whether

$$t \in \mathcal{L}(r'),$$

where r' is a regular expression with $\mathcal{L}(r') = \Sigma^* \mathcal{L}(r) \Sigma^*$ – for example, in UNIX, r' could be equal to "`(.*)r(.*)`". So we can convert r' to an equivalent NFA M and then simply run M on the input `t` to decide whether $t \in \mathcal{L}(M)$.

Theorem 5.8. *Every regular language is an* NFA *language, and vice versa; that is,* $\mathcal{L}(\text{REX}) = \mathcal{L}(\text{NFA})$.

• **Proof of** $\mathcal{L}(\text{REX}) \subseteq \mathcal{L}(\text{NFA})$. Given a regular expression r, we construct an NFA M such that $\mathcal{L}(r) = \mathcal{L}(M)$. By Theorem 5.5, it is sufficient that M be a λ-NFA. We use ***structural induction*** on r:

Induction basis: r is a simple regular expression of the form $\varnothing$, λ, or $\sigma \in \Sigma$. In each of these three cases we can construct an equivalent NFA as shown in Fig. 14.

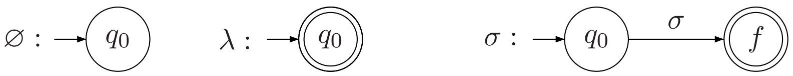


Figure 14: Induction basis in Theorem 5.8.

Induction step: r is of the form (t^*), (tt'), or $(t + t')$, and we assume that there are NFAs M, M' such that $\mathcal{L}(M) = \mathcal{L}(t)$ and $\mathcal{L}(M') = \mathcal{L}(t')$. We need to show that in each of the three cases there is an NFA N accepting the language $\mathcal{L}(r)$. We have that $\mathcal{L}(r) = \mathcal{L}(M)^*$ or $\mathcal{L}(r) = \mathcal{L}(M)\mathcal{L}(M')$ or $\mathcal{L}(r) = \mathcal{L}(M) \cup \mathcal{L}(M')$, and the required NFA N is M^* or MM' or $M \cup M'$, as defined in Theorem 5.6. □

Exercises of Sections 5.7, 5.8, 5.9

Ex. 5.24. Write a Prolog rule `test(F,L)` that runs the automaton in the file `F` on each word w in the list `L` and prints, for each w, `yes/no` depending on whether the automaton accepts w.

Ex. 5.25. Show the product construction of Theorem 5.3 for intersecting the languages of the two automata M and M' in Fig. 16. The first one accepts all words with an even number of a's and the second one all words ending with b.

Solution. The required automaton is shown in Fig. 17. □

Ex. 5.26. What changes would you make to the automaton you constructed in Ex. 5.25 so that you get a new automaton accepting the language $\mathcal{L}(M) - \mathcal{L}(M')$?

Ex. 5.27. Convert the following regular expression to an equivalent λ-NFA

$$a^*(aba + a).$$

Ex. 5.28. In Theorem 5.6, the automaton $M \cup M'$ contains λ-transitions. If M and M' contain no such transitions then we can define directly another automaton N accepting $\mathcal{L}(M) \cup \mathcal{L}(M')$ and

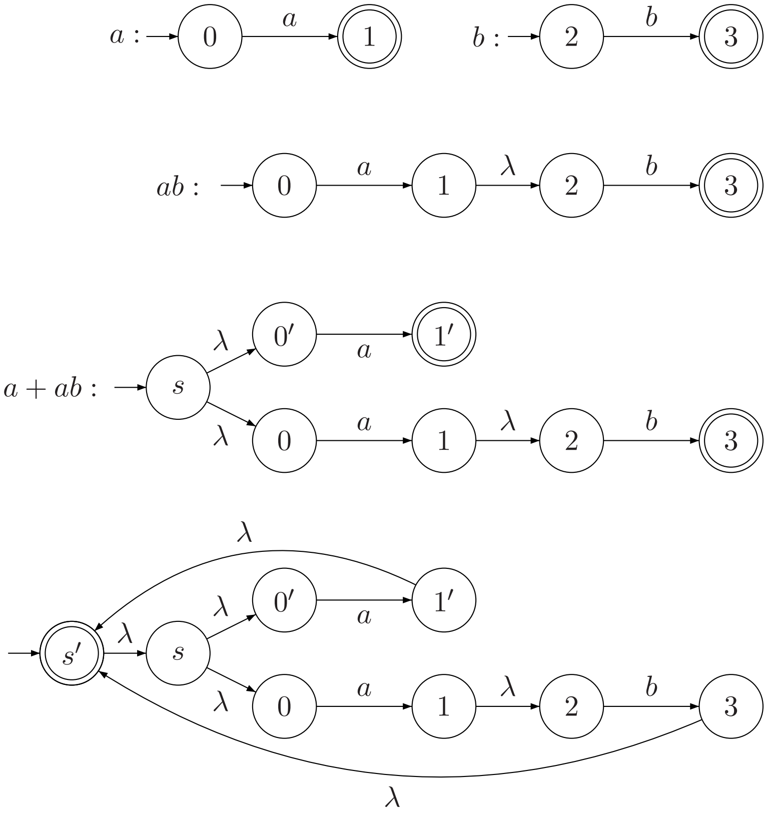


Figure 15: Induction step in Theorem 5.8 for $r = (a + ab)^*$.

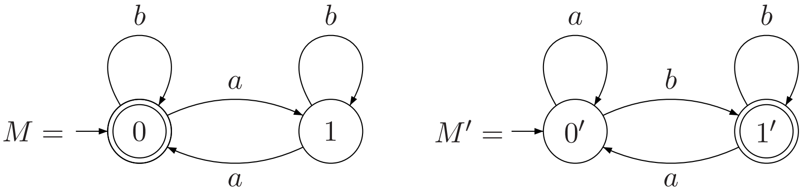


Figure 16: Two automata.

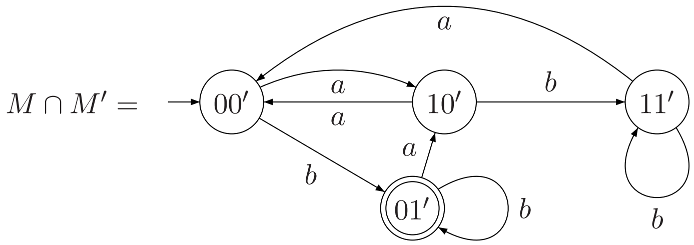


Figure 17: Product construction for the automata in Fig. 16.

containing no λ-transitions. This can be achieved using the product construction in Theorem 5.3 modified as follows: M and M' must be complete NFAs, and the set of final states in the product automaton N consists of all state pairs (f, f') where *at least one* of f and f' is a final state in M, M'. Apply this construction on the automata in Fig. 16. Prove the correctness of the construction using as a guide the proof of correctness in Theorem 5.3.

5.10 Sequential Transducers

• **Motivation**. The machines (automata) we have seen so far receive an input word and return a value in {YES, NO}, that is, they make a decision whether or not to accept the input word. Here we are interested in machines that would allow us to model situations where a given input word is processed and some word is returned. In particular in data communications, a binary message at the site of the sender is sent through a channel and arrives at the site of the receiver such that the channel has possibly changed some of the bits in the message – transmission errors. For example, the channel [Sub(1,∞)] permits at most one substitution in any symbol of the input message, that is, at most one bit in the input word will be changed (a 0 will become 1, or a 1 will become 0). So if the input word is 0000 then the output of the channel could be 0000 (no error), or one of 1000, 0100, 0010, 0001 (one substitution error). Obviously, a channel behaves in a nondeterministic manner.

• **Definition**. A (nondeterministic) sequential transducer (NST) is like an NFA whose transitions have two labels: input and output. More formally, an NST is a 6-tuple

$$M = (Q, \Sigma, \Gamma, q_0, T, F),$$

where Q, q_0, F are as in the definition of NFAs, Σ and Γ are the input and output alphabets, respectively, and T is the set of transitions of the form $(p, \sigma/w, q)$ such that p, q are states in Q, σ is a symbol in Σ, and w is a word in Γ^*. A computation of M is a sequence of transitions of the form

$$(p_0, \sigma_1/w_1, p_1), (p_1, \sigma_2/w_2, p_2), \ldots, (p_{n-1}, \sigma_n/w_n, p_n).$$

A computation is ***accepting*** if p_0 is the start state and p_n is a final state. Given an input word $u \in \Sigma^*$ the transducer returns a word $w \in \Gamma^*$, if there is an accepting computation as above such that

$$u = \sigma_1\sigma_2 \cdots \sigma_n \quad \text{and} \quad w = w_1w_2 \cdots w_n.$$

In general, as an NST is a nondeterministic machine, for a given input word there might be *zero or more possible return values.*

- We have $M(u) =$ the set of possible return values of M on input u. Note that if there is no accepting computation on input u then $M(u) = \emptyset$.

Ex. 5.29. Channel [Sub(1,∞)] Fig. 18 shows the channel that permits up to one substitution error in any input word. For example,

$$[\text{Sub}(1, \infty)](000) = \{000, 100, 010, 001\}.$$

In particular, we have that $010 \in [\text{Sub}(1, \infty)](000)$ as a result of substituting the 2nd bit of the input word 000 with 1. The transducer allows this via the accepting computation

$$(0, 0/0, 0), (0, 0/1, 1), (1, 0/0, 1)$$

The meaning of the states is as follows: State 0 means that there have been no error so far, that is, for each input symbol σ, the corresponding output is σ. State 1 means that one error has occurred. Thus, at state 1 there can be no further errors.

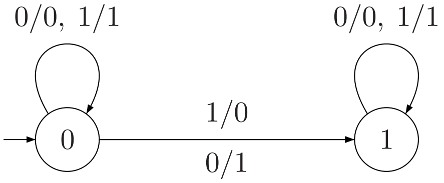


Figure 18: Channel [Sub(1,∞)]. All states are ***final***.

• **Deterministic Sequential Transducer**. If, for every state p and input symbol σ, there is at most one transition of the form $(p, \sigma/\_, \_)$, then M is a deterministic sequential transducer (DST) and realizes a word function. In this case, for each input word w, the set $M(w)$ contains at most one word. In particular if $M(w)$ contains w' then we write $M(w) = w'$ to mean that $M(w) = \{w'\}$. Examples of DSTs are given in the chapter on coding theory.

• **Combinatorial Channel**. This is an NST, like the one in Fig. 18, with the property that, for every accepting input u of M – that is, $M(u) \neq \emptyset$ – we have $u \in M(u)$. Such a machine models a communication channel in the sense that when an input word u is transmitted we receive an output word $u' \in M(u)$. If $u' \neq u$ then we say that u was received with errors – note that this is a typical situation in data communications. Possible errors at a symbol $u(i)$ of the input word can be: substitution of $u(i)$ by another symbol, deletion of $u(i)$, insertion of one or more new symbols to the left of $u(i)$. Although we focus on channels that allow ***only*** substitution errors, we do show in Fig. 19 a channel allowing deletion and substitution errors.

Ex. 5.30. Channels of the form $[\mathrm{Sub}(m, \infty)]$ are appropriate for communication languages of some known maximum length. This is because these channels allow up to m errors no matter how long the input word is. For infinite languages, however, it is more appropriate to consider channels of the form $[\mathrm{Sub}(m, l)]$ that allow up to m errors in any segment of length l of the input word. Fig. 20 shows the channel M = [Sub(1,4)] that permits at most one substitution error in any segment of length 4 of the input word. For example we have

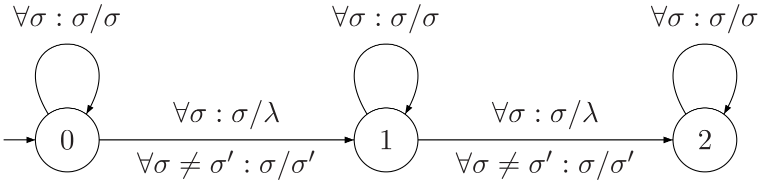

$\forall\sigma : \sigma/\sigma$
$\forall\sigma : \sigma/\sigma$
$\forall\sigma : \sigma/\sigma$
$\forall\sigma : \sigma/\lambda$
$\forall\sigma : \sigma/\lambda$
0
1
2
$\forall\sigma \neq \sigma' : \sigma/\sigma'$
$\forall\sigma \neq \sigma' : \sigma/\sigma'$

Figure 19: Channel [SubDel(2,∞)] allowing two substitutions/deletions in any input message. All states are ***final***. The shorthand notation "$\forall\sigma : \sigma/\lambda$" indicates a set of transitions between the two states involved, one for each letter σ of the alphabet.

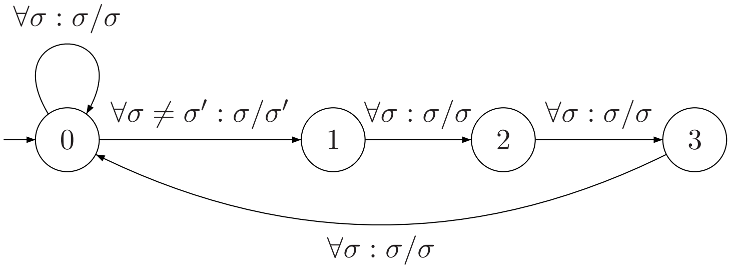

$\forall\sigma : \sigma/\sigma$
$\forall\sigma \neq \sigma' : \sigma/\sigma'$
$\forall\sigma : \sigma/\sigma$
$\forall\sigma : \sigma/\sigma$
0
1
2
3
$\forall\sigma : \sigma/\sigma$

Figure 20: Channel [Sub(1,4)]. All states are ***final***.

that

$$M(00000) = \{00000, 10000, 01000, 00100, 00010, 00001, 10001\}.$$

Note that $01001 \notin M(00000)$ because we cannot have two errors in a block of four zeros.

• **More examples**. More transducer examples are given in the chapter on coding theory.

6 CODING & INFORMATION THEORY

6.1 Coding Systems

• **Some motivation**. In many applications of information processing, the data words involved need to be encoded into a certain format that is appropriate for storage or transmission. For example, storing an English text into a recording device requires encoding English words into binary words that are specific to the device in question [14]. Similarly, sending an image over a network requires encoding the image as a certain binary word (or sequence of binary words) whose bits are interpreted as signals that can be transmitted over the network. Fig. 6.1 shows the scenario of communication considered here.

Definition 6.1. A coding system is a triple (e, D, C) such that D and C are languages over some possibly different alphabets, and e is a 1-1 function such that $e(D) \subseteq C$, that is, the image $e(w)$ of any $w \in D$ must be in C. The language D is called the data language, the language C is called the communication (or channel) language, and the function e is called the encoding function.

We shall normally call the words of D data words and the words of C messages.

• **Why 1-1 function?** When a data word $w \in D$ is encoded as $e(w) \in C$, there should be no other data word $u \in D$ mapped to the same message: $e(u) \neq e(w)$. This implies that the inverse function e^{-1} is well-defined and allows one to decode a message in $e(D)$ correctly, that is, $e^{-1}(e(w)) = w$. The function e^{-1} is called the decoding function.

• **Finite Coding System** (e, D, K). When D and K are finite. If all the words of D are of the same length then we call it a block coding system. For example,

$$(e, \{00, 01, 10, 11\}, \{0, 10, 110, 111\})$$

is a finite block system, where

$$e: \ 00 \mapsto 0, \ 01 \mapsto 10, \ 10 \mapsto 110, \ 11 \mapsto 111.$$

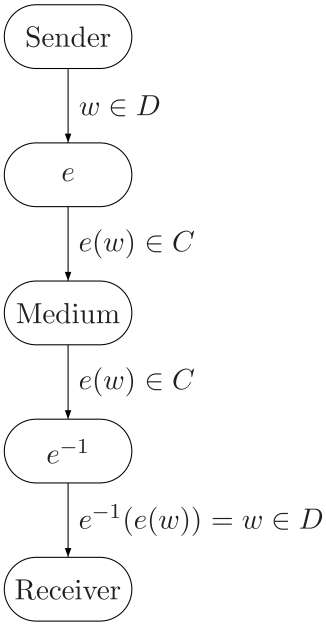


Figure 21: The sender and receiver communicate using a common language D. The communication medium only accepts messages (words) of some language C. The function e maps every data word $w \in D$ to a message $e(w) \in C$.

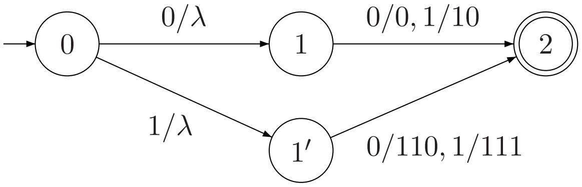


Figure 22: Encoder from $\{0,1\}^2$ to $\{0,10,111,110\}$.

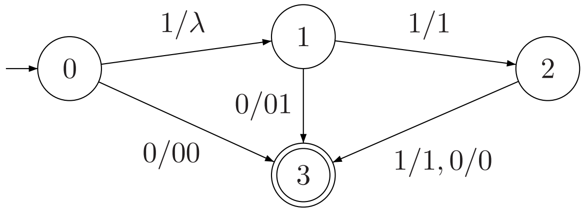


Figure 23: Decoder from $\{0, 10, 111, 110\}$ to $\{0, 1\}^2$.

• **Finite-state Coding**. When the encoding function e can be realized by a deterministic sequential transducer M. In this case, the transducer M is called an encoder – see Fig. 22. Moreover, if there is a deterministic sequential transducer M' that realizes the decoding function then the transducer M' is called a decoder – see Fig. 23. Obviously, we have that $w' = M(w)$ iff $w = M'(w')$.

• **Homomorphic Block Coding** $(e^*, (\Gamma^m)^*, K^*)$. Here Γ is an alphabet and $m \geq 1$, and (e, Γ^m, K) is a coding system. Thus, every data word consists of zero or more blocks of length m, and K is some language, such that
– each block $v \in \Gamma^m$ is mapped to a unique word $e(v) \in K$;
– for every data word $v_1 v_2 \cdots v_n$, with each $|v_i| = m$,

$$e^*(v_1 v_2 \cdots v_n) \ = \ e(v_1)e(v_2) \cdots e(v_n).$$

Moreover, K must be a (uniquely decodable) code, that is, for every word in K^*, there is a exactly one way to break it as a sequence of codewords in K – this ensures that $(e^*)^{-1}$ is well-defined (see Section 6.2). An example of such a system is (e^*, D^*, K^*), where e, D, K are as shown in the previous example of a finite coding system. A decoder for this system is shown in Fig. 24.

Ex. 6.1 (Bit-stuffing Coding)**.** A basic task in data communications is to identify the beginning (and possibly the end) of a received message. This is usually achieved by using a special word, called flag, at the beginning of the message [20]. Moreover, the message should not contain any occurrences of the special flag. An example of a flag is the word 1^{k+1}, for some $k \geq 2$. In this case, the data words must

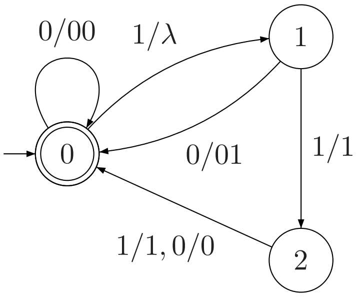


Figure 24: Decoder from $\{0, 10, 111, 110\}^*$ to $(\{0,1\}^2)^*$.

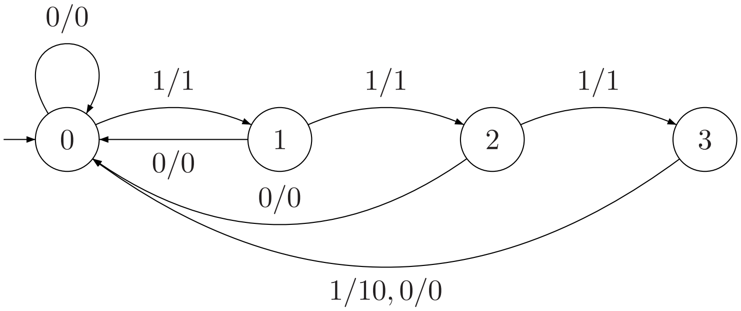


Figure 25: Bit-stuffing encoder with $k = 4$ (all states final).

be encoded into a binary language whose words contain no run of $k + 1$ successive 1s. We define the 1-1 encoding function [BS-k] on $\Sigma_2^* = \{0,1\}^*$ such that:

– [BS-k](w) = the word w' that results when we read w left-to-right and insert a 0 immediately after a run of k ones.

– [BS-k]$^{-1}(w')$ = the word w that results when we read w' left-to-right and delete the zero in every subword 1^k0 of w'.

An example of this method is shown in Fig. 25. So the coding system here is ([BS-k]$, D, C)$, where $D = \Sigma_2^*$ and $C = \{w \in \Sigma_2^* : w$ contains no $1^{k+1}\}$. The decoder for the case of $k = 4$ is shown in Fig. 26.

□

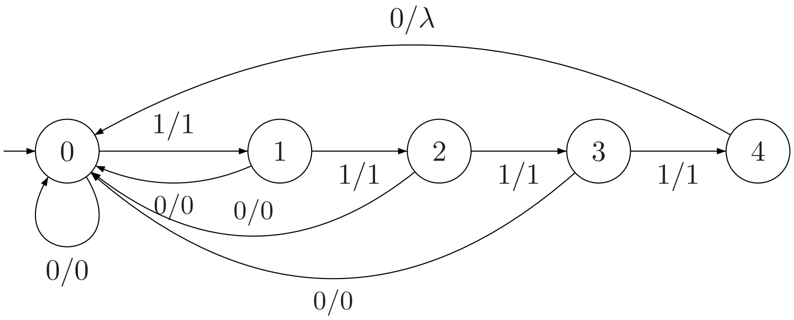


Figure 26: Bit-stuffing decoder with $k = 4$ (all states final).

6.2 Basic Code Properties

Here we introduce a few basic properties of a language K, which is intended to be used for encoding data. The topic of code properties is quite large. The interested reader is referred to [10] for further information on this topic.

6.2.1 Unique decodability

Before providing the formal definition of this section, we give the motivation via an ***incorrect*** example of homomorphic block coding:

$$(e^*, \{a, c, g, t\}^*, K^*), \quad \text{where} \quad K = \{0, 01, 10, 11\},$$

and $e(a) = 0, e(c) = 01, e(g) = 10, e(t) = 11$. Here, the data word agg is encoded as $e^*(agg) = 01010 \in K^*$. However, the message $01010 \in K^*$ can be broken into K-words in two different ways:

$$0, 10, 10 \;\text{ and }\; 01, 01, 0,$$

which implies that $01010 = e^*(agg) = e^*(cca)$ and, therefore, $(e^*)^{-1}$ is not a 1-1 function. The problem here lies on the choice of the language K, as explained next.

Given a language K and a message $w \in K^+$, we say that a word $v \in K$ is a correct first codeword of w, if $w \in vK^*$. For example, using again $K = \{0, 01, 10, 11\}$, we see that in the message $w = 01101$, the first codeword 01 is not correct because the remainder 101 is not in K^*. On the other hand, the first codeword 0 is a correct first codeword of w because the remainder 1101 is in K^*. A language K is called a (uniquely decodable) code if, for every message $w \in K^+$, there is exactly one correct first codeword of w, that is,

$$\forall u_1, u_2 \in K : \ w \in u_1 K^* \wedge w \in u_2 K^* \ \rightarrow \ u_1 = u_2 .$$

Note that this predicate is equivalent to the condition that every message $w \in K^+$ can be broken into K-words in exactly one way.

Ex. 6.2. The language $\{0, 01, 10, 11\}$ is not a code:

$$01010 \ \text{ has two correct first codewords: } \ 0 \text{ and } 01.$$

The languages $\{0, 01, 110\}$ and $\{0, 01, 11\}$ are codes.

• **The decoding problem for a code** K. Given a message $w \in K^+$, the question is to *find the correct first codeword of* w, that is, the correct K-word v_1 such that $w = v_1 w_1$, for some $w_1 \in K^*$. In this case, the remainder w_1 is a message in K^* as well, for which the correct first codeword, say v_2, can also be found. Thus, the message w will be broken into some K-words $v_1, \ldots, v_n$ and then these will be decoded as data words: $e^{-1}(v_1), \ldots, e^{-1}(v_n)$.

• **Block codes**. A language is a block code if all of its words have the same length. Obviously, every block code K is indeed a code. Moreover, the decoding problem here is trivial: If the codewords of K have length l, then the correct first codeword of any message $w \in K^+$ is simply the block that consists of the first l symbols of w.

6.2.2 Prefix property

The decoding problem for a code K becomes complex when there are two codewords $u, v \in K$ such that v is a prefix of u, that is, u is of the form vx. For example, in the code $\{0, 01, 110\}$, the codeword 0 is a prefix of the codeword 01, and when a message of K^+ starts as

$$01 \cdots$$

it is not clear whether the correct first codeword is 0 or 01. This observation motivates the study of prefix codes.

Definition 6.2. A language K is called a prefix code, if no word of K is a prefix of another word of K.

Obviously every message w over a prefix code K can start with exactly one codeword, which must be the correct first codeword of w. The code $\{0, 01, 110\}$ considered above is not a prefix code. On the other hand, $\{0, 10, 11\}$ is a prefix code. If K is a block code all the words of K have the same length, and, therefore, no codeword is a proper prefix of another codeword. Hence, every block code is a prefix code.

Theorem 6.1. *(Kraft-McMillan inequality) Let Σ be an alphabet of r symbols. If $K = \{u_1, \ldots, u_n\}$ is a code over Σ then*

$$\sum_{i=1}^{n} \frac{1}{r^{\ell_i}} \leq 1,$$

where each $\ell_i = |u_i|$. Moreover, for every sequence $(\ell_1, \ldots, \ell_n)$ of positive integers satisfying the above inequality, there exists a prefix code of n codewords whose lengths equal to these integers.

• **Why is this important?** An important objective of coding theory is to find a code of n, say, words for a certain application such that the word lengths $\ell_1, \ldots, \ell_n$ of the code are as small as possible. This would allow for an efficient communication of the messages of this code. Now one would think that the best prefix code might not be as good as certain non-prefix codes in terms of the word length criterion. However, the above theorem says that this is not the case, that is, if $\ell_1, \ldots, \ell_n$ are appropriate code lengths (satisfying the above inequality) then there is always a prefix code with exactly these word lengths. Hence, in trying to find a code for an application, if we ***restrict*** our attention from general codes to prefix codes then we do not sacrifice anything in terms of efficiency.

• **Constructing the Kraft code**. We show how to construct the prefix code K with a given sequence of word lengths $(\ell_1, \ldots, \ell_n)$ according to Theorem 6.1. First prepare a complete binary tree of height ℓ_n – assuming $\ell_i \leq \ell_{i+1}$ and ℓ_n is the largest length. Insert in every node a unique binary word, starting from the root with the

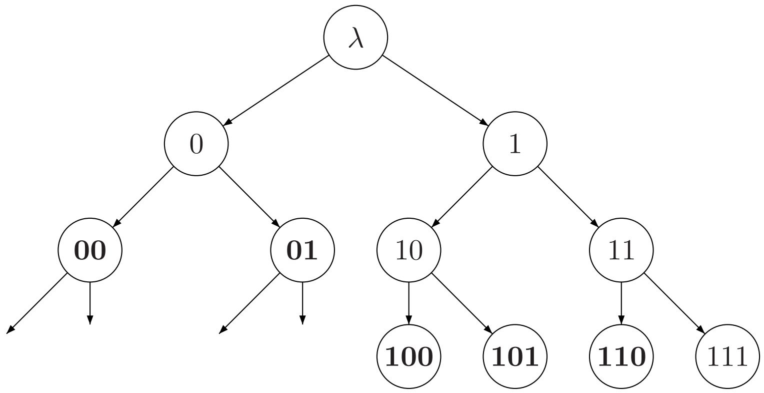


Figure 27: The Kraft construction for given sequence of word lengths.

empty word and going left to right on each level of the tree such that the words are used in the radix order:

$$\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots$$

Then, initialize the code K to empty, and for each $i = 1, \ldots, n$, do the following three tasks: (1) Pick the first (left-to-right) node N_i at level ℓ_i; (2) Add in K the word contained in the node N_i; (3) Remove the subtree whose root is the node N_i. Figure 27 shows this construction for the length sequence $(2, 2, 3, 3, 3)$. The construction gives the code $\{00, 01, 100, 101, 110\}$.

6.2.3 Error detection

A major objective in data communication systems is to process reliably a message that was transmitted via some channel M capable of introducing substitution errors – see Fig. 28. In this context, error detection is a fundamental property of the communication language C. It ensures that, if a word w' is received via the channel M and $w' \in C$, then w' must be correct, that is, equal to the C-word that was sent in M. If, however, the received word w' is not in C then a transmission error is detected and w' is not processed. In other words, the channel cannot turn a C-word to another C-word. For-

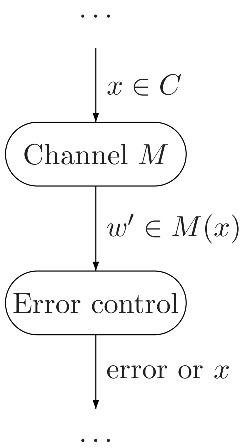


Figure 28: Refining the action of the medium (channel) in Fig. 6.1. The channel is called M. The set $M(x)$ consists of all possible outputs of the channel M on input x. When the channel M introduces errors in x, the output of the channel is $w' \neq x$. The error control module attempts to detect, or even correct, errors in x.

mally, C is error detecting for M, if[11]

$$\forall w', x:\ x \in C \ \wedge\ w' \in M(x)\ \wedge\ w' \in C\ \ \rightarrow\ \ x = w'.$$

• **Testing for error detection**. To test whether a ***finite*** language $K = \{u_1, \ldots, u_n\}$ is error detecting for some channel M we can do the following:

– Compute the sets $M(u_1), M(u_2), \ldots, M(u_n)$;
– If there are $u_i \neq u_j$ with $u_i \in M(u_j)$
  then K is not error detecting for M.
– Else K is error detecting for M.

Ex. 6.3. The language $K = \{11, 011, 101\}$ is error detecting for the channel M = [Sub(1,3)]:

– $M(11) = \{11, 01, 10\}$
– $M(011) = \{011, 111, 001, 010\}$

[11] Recall we assume that the channel M allows only substitution errors and, therefore, if $w' \in M(x)$ then w' has the same length as x.

– $M(101) = \{101, 001, 111, 100\}$.

However, K^* is not error detecting for that channel:

$$111111 \in K^3 \text{ and } 011011 \in K^2 \text{ and } 111111 \in [\text{Sub}(1,3)](\underline{0}11\underline{0}11),$$

where underlined 0s indicate substitution errors.

• **Even parity code** E_n. This is the most famous code for detecting errors of $[\text{Sub}(1,\infty)]$, that is, at most one error in any codeword. It is a binary block code such that each word contains an even number of 1s. For example, E_3 consists of the following codewords

$$000\underline{0}, 001\underline{1}, 010\underline{1}, 011\underline{0}, 100\underline{1}, 101\underline{0}, 110\underline{0}, 111\underline{1}.$$

In each codeword, the first 3 bits are arbitrary and the 4th bit is chosen such that the total number of 1s in the codeword is even. In general the even parity code E_n is of length $n+1$, for some $n > 1$, where the first n bits of a codeword in E_n are arbitrary and the last bit is either 0 or 1 such that the total number of 1s is even:

$$E_n = \{ub \mid |u| = n,\ ub \text{ has an even \# of 1s}\}.$$

Note that E_n is error detecting for the channel [Sub(1,∞)] because if an error occurs in a codeword x then the resulting word w' has an odd number of ones and, therefore, w' cannot be in E_n.

• **n-block encoder into** E_n. This is a deterministic sequential transducer M that returns, for each input word u of length n, one output word w in E_n, that is, $M(u) = w$. For example, the 3-block encoder of Figure 29 encodes every data word of length 3 into a codeword of the even parity code E_3.

• **Decoder for** E_n. This is a deterministic sequential transducer that maps the words of E_n into the corresponding length n data words. For example, the 4-block decoder of Figure 30 maps every even parity word ub of E_3 into the data word u. If the input word ub is not of even parity (due to a transmission ***error***) then there is no output.

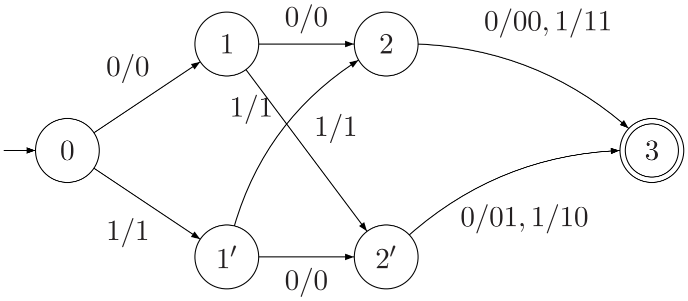


Figure 29: 3-block encoding into E_3.

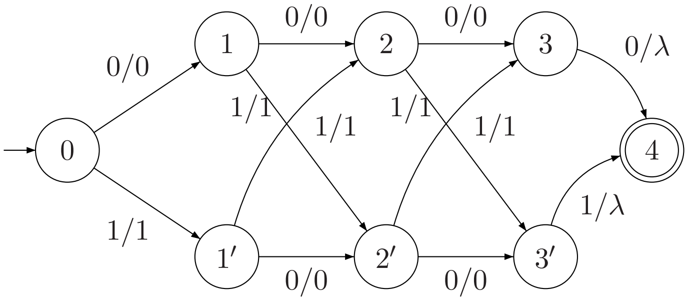


Figure 30: 4-block decoding.

• **Hamming distance**. Error detection has been studied extensively for channels of the form [Sub(m, ∞)], which permit up to m substitutions in any input word. In this case, the concept of Hamming distance is important. The Hamming distance between two words w, w' of the same length, denoted as Hamm(w, w'), is the number of corresponding positions in which the words differ. For example,

$$\text{Hamm}\,(00000, 10010) = 2,$$

because the words 00000 and 10010 differ in two positions: their 1st and 4th bits are different. The Hamming distance Hamm(K) of a block code K is the smallest distance between any two ***different*** words of K.

Theorem 6.2. *A block code K is error detecting for the channel [Sub(m, ∞)] iff Hamm(K) $> m$.*

6.2.4 Error correction

Recall that, if the communication language C is error detecting for the channel M, then an error is detected exactly when a word w' that was received via M is not in C. In this case, the original message $x \in C$ must be retransmitted with the hope that this time it will be received with no errors. If, however, the communication language C is error-correcting for the channel M, then it is possible to find out the original message x even when it is received via M with errors. Formally, C is error correcting for M, if

$$\forall w'\, \forall x, w \in C:\ w' \in M(x) \wedge w' \in M(w)\ \rightarrow\ x = w.$$

Thus, for every received word w', there is exactly one message in C that can be received as w'. One can verify that, if C is error correcting for M, it must be error detecting for M.

• **Testing for error correction**. To test whether a ***finite*** language $K = \{u_1, \ldots, u_n\}$ is error correcting for a channel M we can do the following:

– Compute the sets $M(u_1), M(u_2), \ldots, M(u_n)$;
– If there are $u_i \neq u_j$ with $M(u_i) \cap M(u_j) \neq \emptyset$
    then K is not error correcting for M.
– Else K is error correcting for M.

Ex. 6.4. The language $K = \{11, 011, 101\}$ of Ex. 6.3 is not error correcting for the channel $M =$[Sub(1,3)], because $001 \in M(011) \cap M(101)$. Thus, when the word 001 is received, we know that an error occurred (as $001 \notin K$), but we cannot correct the error, that is, we cannot tell whether the original message was 011 or 101.

Theorem 6.3. *A block code K is error correcting for the channel [Sub(m, ∞)] iff Hamm$(K) > 2m$.*

• **The Hamming Code [7].** Hamming was one of the first to discover an efficient error correcting code for the channel [Sub(1,7)]. The Hamming code is a binary block code of length 7 such that, in every codeword, the bits at positions 3,5,6,7 are arbitrary and the bits at positions 1,2,4 are called parity bits whose values depend on the values of the arbitrary bits. Formally, the code is as follows:

$$\begin{aligned}\{p_1p_2b_3p_4b_5b_6b_7 \mid\; & b_3, b_5, b_6, b_7 \in \{0,1\},\\ & p_4 + b_5 + b_6 + b_7 =_2 0,\\ & p_2 + b_3 + b_6 + b_7 =_2 0,\\ & p_1 + b_3 + b_5 + b_7 =_2 0\ \},\end{aligned}$$

where $=_2$ indicates that the sum of bits is computed modulo 2. When an unknown codeword u is received via the channel as z, one can find the unique set [Sub(1,7)](u) containing z, and then decode z as u. However, here there is a quick method to decode z. Compute (modulo 2) the sums

$$c_0 = z(4) + z(5) + z(6) + z(7),$$
$$c_2 = z(2) + z(3) + z(6) + z(7),$$
$$c_1 = z(1) + z(3) + z(5) + z(7).$$

Each of these sums is a value in $\{0,1\}$. If the binary word $c_0c_2c_1$ represents the number zero then there is no error in z. Else, the error is at the position represented by $c_0c_2c_1$.

• K **vs** K^* **substitution error correction**. Substitution errors have the property that they do not alter the start positions of the codewords in a message. This implies that if a block code K of some length ℓ is error correcting for the channel [Sub(m, ℓ)] then K^* is error correcting for the channel [Sub(m, ℓ)]. Unfortunately, this is not the case when the channel involves insertions and/or deletions.

Exercises of Sections 6.1, 6.2

Ex. 6.5. Can you construct a binary prefix code with word lengths

2, 2, 3, 3, 3, 4, 4, 4 ?

2, 2, 3, 4, 4, 4, 4, 4, 4 ?

Explain your answers. When your answer is yes show the required code.

Ex. 6.6. Let $D = \{00, 01, 10, 11\} = \{0, 1\}^2$ and

$$K = \{101, 10^2 1, 10^3 1, 10^4 1\}.$$

We need a finite coding system (e, D, K). Show the diagram of a 3-state encoder for e. Then show the diagram of the corresponding decoder.

Ex. 6.7. Change your diagrams for the previous exercise such that the new encoder and decoder will work for a homomorphic block coding system (e^*, D^*, K^*).

Ex. 6.8. For each of the following languages indicate whether it is a code, and whether it is a prefix code.

{01, 22, 012, 201}
{1, 20, 21, 012}
{20, 021, 202}

Ex. 6.9. Find the Hamming distance of the code

$$\{0102, 1010, 2221, 0011\}.$$

Is the code error detecting for the channel [Sub(1,4)]? Is it error correcting for that channel? If not find a subset of this code (as large as possible) that is error correcting for [Sub(1,4)]. Is the code error detecting for the channel [Sub(1,3)]?

6.3 Information Theory

• **What is it?** A theory that provides limits on the efficiency of coding techniques. It finds applications in data analysis (e.g., bioinformatics) and machine learning. It is also important from a philosophical point of view as it defines mathematically the concept of information which is a fundamental concept in human activities.

• **Efficiency of codes – Average word length**. Suppose we need a code to encode n different objects, to which we refer by simply using the integers $1, 2, \ldots, n$. There are many codes consisting of n words. We wish to choose a code K whose words are as short as possible, as this would allow for an efficient communication of messages in K^+. More specifically, we wish to choose a code $K = \{v_1, \ldots, v_n\}$ of n elements such that the *average word length* of K

$$\ell(K) = \frac{|v_1| + \cdots + |v_n|}{n}$$

is as small as possible. In many cases each object $i \in \{1, 2, \ldots, n\}$ is used with a known probability $\tilde{p}(i)$, which implies that the corresponding K-codeword v_i occurs in messages with probability $\tilde{p}(i)$. Thus, instead of the term object, we use the term ***event***. More specifically, we have n events that occur according to a probability distribution

$$\tilde{p} = (\tilde{p}(1), \ldots, \tilde{p}(n)),$$

that is, each number $\tilde{p}(i) \in (0, 1]$ and $\sum_{i=1}^{n} \tilde{p}(i) = 1$. In this more general scenario, the *average word length of* K *(with respect to)* $\tilde{p}$ is

$$\ell_{\tilde{p}}(K) \;=\; \tilde{p}(1) \cdot |v_1| + \cdots + \tilde{p}(n) \cdot |v_n| \;=\; \sum_{i=1}^{n} \tilde{p}(i)|v_i|.$$

Obviously, if $\tilde{p}$ is the uniform distribution $(1/n, \ldots, 1/n)$ then $\ell_{\tilde{p}}(K) = \ell(K)$. In any case, the objective is to find a code of n words such that the average word length is as small as possible.

• **Recall a simple coding fact**. To encode n objects using a binary block code, the code length must be at least $\lceil \log n \rceil$.[12] We shall see that encoding n objects can be done more efficiently if we allow *variable-length* codes.

Ex. 6.10. [**A motivating example**] Consider the following statistical data about the daily weather conditions in Atlantis:

sunny: $0.3 = 30\%$
partly cloudy: $0.3 = 30\%$

[12] Recall that, for any real number x, $\lceil x \rceil$ is the smallest integer $\geq x$. For example, $\lceil 2.18 = 3 \rceil$. Of course, if x is an integer then $\lceil x \rceil = x$.

overcast: $0.1 = 10\%$
rain: $0.2 = 20\%$
snow: $0.1 = 10\%$

We want to communicate in binary the weather conditions in the last 10,000 days. What is an efficient way to encode this information?

The simplest method is to encode the five events {sunny, partly-cloudy, overcast, rain, snow} using a block code K_1. This means that the words of K_1 must be of length 3 and, therefore, this requires 3 bits per event. Thus, to encode 10,000 events we need a total of 30,000 bits.

Another approach is to use a (variable-length) prefix code instead of a block code. In particular one can argue that since sunny is a most frequent event, we can assign to this event the codeword 0 and then the codewords 100, 101, 110, 111 to the rest of the events. In this case, the average word length (= average number of bits per event) is

$$1 \times 0.3 + 3 \times 0.3 + 3 \times 0.1 + 3 \times 0.2 + 3 \times 0.1 = 2.4 \text{ bits}$$

and, therefore, we need (on average) 24,000 bits to encode 10,000 events.

The second method is obviously more efficient. We shall see later in this section whether we can do better than that.

• **Definition of Information/Entropy**. The most successful approach to defining mathematically the concept of information was proposed in [19] by Shannon. In this approach, information exists in connection with a source of ***possible events*** in the sense that the occurrence of *an event brings some information* to the observer. Moreover, the lower the chance of occurrence of an event is, the higher the information content of that event is.

An (information) source is a function $\tilde{p} : E \to (0, 1]$ such that E is a nonempty finite set, called the set of events, and $\tilde{p}(i)$ is the probability of the event i such that

$$\sum_{i=1}^{n} \tilde{p}(i) = 1,$$

where n is the number of events, that is, $n = |E|$. For any source $\tilde{p}$ we shall write $E_{\tilde{p}}$ for the event set of $\tilde{p}$, but we shall drop the subscript $\tilde{p}$ and simply write E instead of $E_{\tilde{p}}$ if there is no risk of confusion. Moreover, unless stated otherwise, we shall use positive integers $1, 2, \ldots$ to denote the events in $E = E_{\tilde{p}}$. Hence, we can say that a source $\tilde{p}$ is a probability distribution

$$\tilde{p} = (\tilde{p}(1), \ldots, \tilde{p}(n)).$$

Here we shall assume that the source is memoryless[13].

Definition 6.3. Let $\tilde{p}$ be an (information) source of n events. The information content, or entropy, or uncertainty of the event i is the value $(-\log \tilde{p}(i))$, and the <u>information content</u>, or <u>entropy</u>, or <u>uncertainty</u> of the source is the value

$$H(\tilde{p}) = -\sum_{i=1}^{n} \tilde{p}(i) \log \tilde{p}(i).$$

Thus, the information content of a source is the average of the information contents of its events. The unit for measuring information content is (usually) the ***bit***. This choice comes from our custom of writing information down in words. Here in particular, suppose we want to communicate/describe events of the source using a binary code. When we say that the information content of the source $\tilde{p}$ is, for instance, 5 bits, it is meant that it takes an *average* of about 5 bits to describe in binary an event of the source – this interpretation is justified mathematically in Theorem 6.7.

The quantity $H(\tilde{p})$ can also be interpreted as the amount of ***uncertainty*** of the source. To see this, consider the time instance ***before*** the occurrence of the next event. As the next event is one of the n possible ones, there is some uncertainty as to what the event will be. This interpretation is consistent with our original one because at the time instance ***after*** the occurrence of the event there is no uncertainty but there is some information brought by the occurrence of the event.

Ex. 6.11. Consider again the source (0.3, 0.3, 0.1, 0.2, 0.1) in the previous example about the daily weather in Atlantis. We compute

[13] The occurrence of an event does not depend on any of the previous event occurrences.

the entropy of this source. We use the facts $\log(xy) = \log x + \log y$, $\log(1/x) = -\log x$ and $\log x = \log_b x / \log_b 2$:

$$\begin{aligned}
H &= -(\frac{3}{10}\log\frac{3}{10} + \frac{3}{10}\log\frac{3}{10} + \frac{1}{10}\log\frac{1}{10} + \frac{2}{10}\log\frac{2}{10} + \frac{1}{10}\log\frac{1}{10}) \\
&= -\frac{6}{10}\log\frac{3}{10} - \frac{2}{10}\log\frac{1}{10} - \frac{2}{10}\log\frac{2}{10} \\
&= \frac{6}{10}(\log 10 - \log 3) + \frac{2}{10}\log 10 + \frac{2}{10}(\log 10 - \log 2) \\
&= \log 10 - 0.6\log 3 - 0.2 = 1 + \log 5 - 0.6\log 3 - 0.2 \\
&= 0.8 + 2.3219 - 0.6 \cdot 1.58496 = 2.1709...
\end{aligned}$$

• **Maximum entropy for n-event sources**. Consider all possible sources involving exactly n events, for some fixed $n \geq 2$. The source that maximizes the entropy function is the one with the uniform probability distribution:

$$\tilde{u}_n = (\frac{1}{n}, \dots, \frac{1}{n}).$$

It can be shown that $H(1/n, \dots, 1/n) = \log n$. So we have that for every source $\tilde{p}$

$$H(\tilde{p}) \leq H(1/n, \dots, 1/n) = \log n.$$

6.3.1 Entropy and Average codeword length

Here we discuss the importance of entropy in evaluating the average word length of a code. The proof of the next result can be found in the appendix.

Lemma 6.4. *Let K be a code for some source $\tilde{p}$. Then*

$$\ell_{\tilde{p}}(K) \geq H(\tilde{p}).$$

Moreover, there is a prefix code K_0 such that $\ell_{\tilde{p}}(K_0) < H(\tilde{p}) + 1$.

According to the above result, we cannot encode the events of the source $\tilde{p}$ using a code whose average length is smaller than the entropy of $\tilde{p}$. On the other hand, we can always encode the events of $\tilde{p}$ using a prefix code whose average length is one plus the entropy of the source! This can be done as follows, given the source

$$\tilde{p} = (\tilde{p}(1), \ldots, \tilde{p}(n)).$$

– For each event i, define the Shannon length $\ell_i = \lceil -\log \tilde{p}(i) \rceil$
– Construct the Kraft code K_0 with lengths $(\ell_1, \ldots, \ell_n)$.

In the proof of the above lemma it is shown that

$$\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1$$

and the average length of the constructed Kraft code is indeed smaller than $H(\tilde{p}) + 1$.

• **Huffman coding**. Here we consider the question of finding optimal (in terms of average word length) codes for a given source $\tilde{p}$. In particular we present the algorithm of Huffman that is based on binary trees – the method also applies to higher order trees. The algorithm is nondeterministic and could result, in general, to different codes. Any resulting code, however, has the same average word length. We note that the Kraft code discussed before is not always optimal.

1. Input: source $\tilde{p}$ of n events.

2. Construct n tree roots $R_1, \ldots, R_n$ and assign to each R_i the value $v(R_i) = \tilde{p}(i)$.

3. Choose two trees T_1, T_2 with smallest root values, and replace them with one new tree whose root has value $v(T_1)+v(T_2)$ and T_1, T_2 as left and right subtrees.

4. If at least two trees are left, repeat the above step.

5. For each node of the tree, assign the bit 0 to its left edge and the bit 1 to its right edge.

6. For each leaf node R_i, with $i = 1, \ldots, n$, define $e(i)$ to be the binary word that is formed along the path from the root of the tree to R_i.

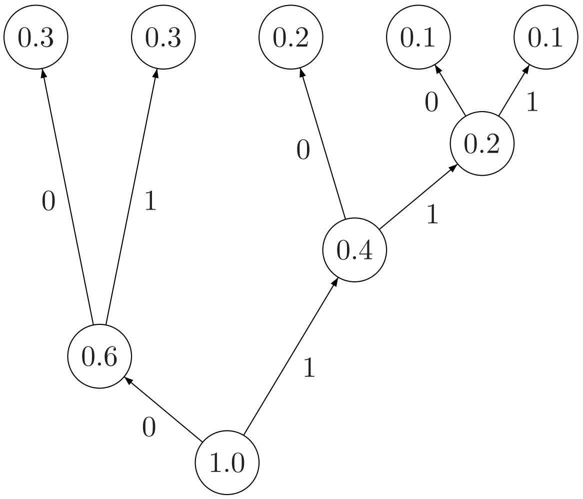


Figure 31: A Huffman tree for the source $(0.3, 0.3, 0.2, 0.1, 0.1)$.

7. Return the code $\{e(1), \ldots, e(n)\}$.

• Note that all Huffman codes are ***prefix*** codes. This is because each codeword corresponds to a leaf of the tree, that is a node with no children and, therefore, that codeword cannot be a prefix of another codeword.

Theorem 6.5. *Let $\tilde{p}$ be a source. There is no code for $\tilde{p}$ whose average word length is smaller than that of some Huffman code for $\tilde{p}$.*

Ex. 6.12. Fig. 31 shows a Huffman tree for the Atlantis weather source $(0.3, 0.3, 0.2, 0.1, 0.1)$ in Ex. 6.10. The code is {00, 01, 10, 110, 111} and has an average word length

$$2 \times 0.3 + 2 \times 0.3 + 2 \times 0.2 + 3 \times 0.1 + 3 \times 0.1 = 2.2 \text{ bits.}$$

Thus, the Huffman method has produced a better code than the ones we tried in the introductory paragraphs of this section – now it would require 22,000 bits to encode 10,000 weather events of Atlantis. Of course, as stated in Theorem 6.5, this is the best we can do for single-event to codeword coding.

• **Entropy of the k-power source**. Recall from Ex. 6.10 that the entropy of the Atlantis weather source is about 2.1709 bits, and the Huffman algorithm gives a code of average length 2.2 bits. This suggests the possibility of improving the encoding efficiency for this source if we somehow change our requirement of assigning one codeword to one event. More specifically, we can try to define a code for pairs (i, j) of events. In the case again of the Atlantis weather source, there are $5 \times 5 = 25$ pairs of events such that the probability of the 2-event (i, j) is $\tilde{p}(i) \times \tilde{p}(j)$. For example, the probability of the 2-event (sunny, rain) is $0.3 \times 0.2 = 0.06$. In effect we have defined the new source $\tilde{p}^2$ of 25 events such that

$$\tilde{p}^2(i, j) = \tilde{p}(i) \times \tilde{p}(j).$$

In general, let $\tilde{p}$ be a source and $k \geq 1$. The k-power of $\tilde{p}$ is the source $\tilde{p}^k$ such that each event in $E_{\tilde{p}^k}$ is a sequence $(i_1, \ldots, i_k)$ of k events from $E_{\tilde{p}}$, and the probability $\tilde{p}(i_1, \ldots, i_k) = \tilde{p}(i_1) \cdots \tilde{p}(i_k)$. The following holds.

Lemma 6.6. *For every source $\tilde{p}$ and integer $k \geq 1$, $H(\tilde{p}^k) = kH(\tilde{p})$.*

The next result follows easily from Lemma 6.4 and the concept of k-power source.

Theorem 6.7. ***(Fundamental Coding Theorem)*** *Let C be a code for the k-sequences of some source $\tilde{p}$. Then*

$$\ell_{\tilde{p}^k}(C) \geq kH(\tilde{p}).$$

Moreover, there is a prefix code C_0 such that

$$kH(\tilde{p}) \leq \ell_{\tilde{p}^k}(C_0) < kH(\tilde{p}) + 1.$$

• **What does this mean?** In order to encode the events of a source $\tilde{p}$, we need on average no less than $H(\tilde{p})$ bits per event and we can get close to that limit. In particular, there are cases where we can encode each single event with a codeword such that the average word length is exactly $H(\tilde{p})$ bits. If this is not possible, however, we can find a code C_k for the k-sequences of the source (for some $k > 1$) which has an average word length of less than $kH(\tilde{p}) + 1$ bits and, therefore, this corresponds to $H(\tilde{p}) + 1/k$ bits per single event. Thus, as $k \to \infty$, we have that $1/k \to 0$, so the average number of bits per event tends to $H(\tilde{p})$.

Exercises of Section 6.3

Ex. 6.13. Here are the statistics for the accuracy of rain prediction in Wonderland by the weather station [17]:

	actual rain	actual no-rain
rain prediction	1/12	1/6
no-rain prediction	1/12	2/3

For instance, 1/12 of the time the weather station predicts rain when in fact it does rain.

1. How often is the weather station correct?

2. Find the amount of uncertainty about the (actual) rain in Atlantis?

3. Find the amount of uncertainty in the station's rain predictions.

4. An unemployed listener observed that one would be correct 5/6 of the time by simply always predicting no-rain, and so the listener applies for a job. Why does the station manager decline to hire the listener?

Ex. 6.14. Find a Huffman code for the source

$$(1/4,\ 1/4,\ 1/8,\ 1/8,\ 1/8,\ 1/8).$$

Would we gain anything if we used a code for the k-sequences of this source, for some $k > 1$?

Ex. 6.15. Consider the source $\tilde{p} = (1/10, 2/10, 7/10)$. Find the entropy of the source and the average length of a Huffman code for that source. Then, construct the Huffman code for the source $\tilde{p}^2$ and find the average length of that code. Did this help?

7 LANGUAGE TYPES & PUSHDOWN AUTOMATA

Recall the description method of regular expressions defines the class of regular languages. On the other hand, there is no description method for the set of all languages – see Corollary 1.4. Hence, there must be languages that are not regular. Using a similar argument we see that there must be languages that are not context-free. In this section we discuss briefly the four types (classes) of languages that constitute the well-known Chomsky hierarchy of languages. Two of these types we have seen: the regular (or type-3) and context-free (or type-2) languages. The other two are the context-sensitive (or type-1) and the unrestricted-grammar (or type-0, or recursively enumerable) languages.

7.1 Non-regular languages

The language

$$\{a^n b^n \mid n \in \mathbb{N}_0\} = \{\lambda, ab, a^2b^2, a^3b^3, \ldots\}$$

can be described by a context-free grammar, but according to the following result it cannot be described by any regular expression or NFA! The proof can be found in the appendix.

Theorem 7.1. *The language $\{a^n b^n \mid n \in \mathbb{N}_0\}$ is not regular.*

• **Guessing that L is non-regular**. If L is regular there is an NFA recognizing L with N, say, states. Then, at any step of its computation, the NFA can 'remember' up to N facts about the part of the input it has read so far. Thus, it is not possible to have an NFA for a language that contains arbitrarily long words with two 'matching' parts. For example, to recognize $a^n b^n$, an NFA would have to remember the number of a's it has read, so that when the first b is read, the NFA tries to match the number of b's.

The language of arithmetic expressions is non-regular because of the unbounded number of matching parentheses '(', ')' that could occur in such expressions. Similarly C++ and Java are not regular languages.

Note that the above explanations are not mathematical proofs. Proofs of such statements in the mathematical sense are based on the Pumping Lemma of regular languages, which is omitted here. We refer the interested reader to [24, 23], for instance. Another method for non-regularity is described next.

• **A method for proving that an infinite L is non-regular**. The method relies on a language K that is known to be non-regular. First, we assume that the given L is regular. Then, we find a language L' that is regular and satisfies $K = L \cap L'$. As the intersection of two regular languages is regular (recall Theorem 5.3), K must be regular as well, which is a contradiction. ***Note*** The same method applies if, instead of the intersection '$\cap$', we use another operation such as union: '$\cup$', set-difference: '$-$', concatenation, etc.

Ex. 7.1. Using the fact that $K = \{a^n b^n \mid n \in \mathbb{N}_0\}$ is not regular, we show that the language

$$L = \{a^n b^n \mid n \in \mathbb{N}_0,\ n \geq 2\}$$

is not regular as well. Observe that L is equal to $K - \{\lambda, ab\}$.

1. ***Assume:*** L is regular.

2. $\{\lambda, ab\}$ is regular: recall all finite languages are regular.

3. $L \cup \{\lambda, ab\}$ is regular: by Theorem 5.6 and lines 1, 2.

4. $L \cup \{\lambda, ab\}$ is not regular: by Theorem 7.1.

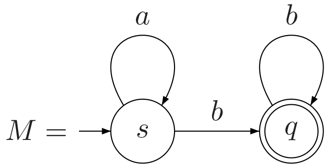
5. ***Contradiction:*** lines 3, 4. Hence L is not regular.

7.2 Back to Context-free Languages

We have seen languages that are not regular and we know that there are languages that are not context-free. In this section we establish the relationship between regular and context-free languages, and show an example of a language that is not context-free.

Theorem 7.2. *Every regular language is context-free. Moreover, there is a context-free language that is not regular. Hence,*

$$\mathcal{L}(\text{REX}) \subsetneq \mathcal{L}(\text{CFG}).$$



$$\Longrightarrow \quad G = \begin{array}{l}\text{“}X_s;\ X_s \to aX_s \mid bX_q;\\ X_q \to \lambda \mid bX_q\text{”}\end{array}$$

Figure 32: Construction in Theorem 7.2: NFA to context-free grammar.

Proof. The language in Theorem 7.1 is context-free, but not regular. Next we show that every regular language L is context-free. More specifically, we assume that L is accepted by some NFA $M = (Q, \Sigma, s, T, F)$ and we construct a context-free grammar G such that $\mathcal{L}(G) = \mathcal{L}(M) = L$. The required grammar consists of the following: (i) variables

$$\{X_q \mid q \in Q\},$$

that is, one variable X_q for each state of M, (ii) start variable X_s, where s is the start state of M, and (iii) rules

$$\{X_p \to \sigma X_q \mid (p, \sigma, q) \in T\} \cup \{X_f \to \lambda \mid f \in F\},$$

that is, one rule for each transition (p, σ, q) and one rule for each final state f. Fig. 32 shows an example of this construction. Now we see that, for each word $w = \sigma_1 \cdots \sigma_n$, there is a computation of M

$$(s, \sigma_1, q_1), (q_1, \sigma_2, q_2), \ldots, (q_{n-1}, \sigma_n, q_n)$$

accepting the word w iff there is a derivation of the grammar G

$$X_s \Rightarrow \sigma_1 X_{q_1} \Rightarrow \sigma_1 \sigma_2 X_{q_2} \Rightarrow \cdots \Rightarrow (\sigma_1 \cdots \sigma_n) X_{q_n} \Rightarrow \sigma_1 \cdots \sigma_n$$

generating the word w. Hence, $\mathcal{L}(G) = \mathcal{L}(M)$, as required. □

The next theorem gives the most famous non context-free language.

Theorem 7.3. *There exists no context-free grammar generating the language*

$$\{a^n b^n c^n \mid n \in \mathbb{N}_0\}.$$

• **Some closure and non-closure properties**. Let L, L' be two context-free languages. Then
– $L \cup L'$ and L^* are context-free languages.
– $L \cap L'$ and $\overline{L}$ are not necessarily context-free.
– If R is regular then $L \cap R$ is (always) context-free.

7.3 Context-sensitive & Unrestricted Grammars

Here we define another class of languages that properly contains the class of context-free languages. For this class, we extend the concept of grammar by allowing, in addition to context-free rules, rules in which the variable of the rule can be replaced with a word only if there is a certain pattern to the left and/or to the right of the variable.

• **Context-sensitive rule**. This is an expression of the form

$$\alpha \to \beta$$

such that $\alpha, \beta \in (\Sigma \cup V)^*$, that is, α, β are any strings involving terminal symbols and/or variables, and α contains ***at least one variable***. Obviously, if α is a single variable, then the rule is simply a context-free rule. The context-sensitive rule

$$0X11 \to 01011$$

says that X can be replaced with 10 if there is a 0 on the left of X and 11 on the right of X. This ***rule can be applied*** to strings containing $0X11$. For example,

$$01\underline{0X11}X11 \Rightarrow 01\underline{01011}X11.$$

• **Context-sensitive grammar (CSG)**.
This is a quadruple (V, Σ, S, R) such that V, Σ, S, are the alphabet of variables, the terminal alphabet and the start symbol, as in the case of CF grammars, and R is a set of context sensitive rules. Moreover, in every rule $\alpha \to \beta$ we have that $|\alpha| \leq |\beta|$. The possible exception to that is to have the rule $S \to \lambda$, in which case S does not occur in the right side of any rule. As usual, we write $\mathcal{L}(\text{CSG})$ for the class of languages generated by context-sensitive grammars – these languages are simply called context-sensitive languages.

Ex. 7.2. The language $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$ is context-sensitive. Here is a context-sensitive grammar from [15]:

$$\begin{aligned} S &\rightarrow \lambda \mid T \\ T &\rightarrow aTBc \\ T &\rightarrow abc \\ cB &\rightarrow Bc \\ bB &\rightarrow bb \end{aligned}$$

When the second rule is applied repeatedly, the grammar generates equal numbers of a's, B's and c's. Then, the third rule generates one extra a, a b and a c. However, the B's have to move to the left of c's. This is achieved using the fourth rule. The last rule can be applied only if there is a b to the left of a B and, when this happens, there can be no c's to the left of any b.

• **Unrestricted grammar (UNG)**. This is a quadruple (V, Σ, S, R) such that V, Σ, S, are the alphabet of variables, the terminal alphabet and the start symbol, as in the case of CF grammars, and R is a set of context sensitive rules. (Note there is no restriction on the rules as in the case of context-free, or context-sensitive, grammars.) The grammar in the previous example is an unrestricted.

• **Chomsky Hierarchy of Languages**. The following hierarchy of language classes holds

$$\mathcal{L}(\text{REX}) \subsetneq \mathcal{L}(\text{CFG}) \subsetneq \mathcal{L}(\text{CSG}) \subsetneq \mathcal{L}(\text{UNG}).$$

Each of the various description methods for languages (regular expressions, context-free grammars, etc) defines a certain class of languages. A question that arises is whether there are other formal methods that are capable of describing languages and, in particular, languages that cannot be described by unrestricted grammars. We investigate this question in the section on Turing Machines.

Exercises of Sections 7.1–7.3

Ex. 7.3. Draw the state diagram of a DFA accepting the following language

$$\{(ab)^n a \mid n \in \mathbb{N}_0\}.$$

Then, use the method in Theorem 7.2 to construct a context-free grammar generating the same language.

Ex. 7.4. Is the following language regular? Prove your answer.

$$\{a^{2n}b^{2n} \mid n \in \mathbb{N}_0\}.$$

Hint: Observe that the given language consists of all words that belong to the language of Theorem 7.1 ***and*** have an even number of a's followed by an even number of b's.

Ex. 7.5. Is the following language regular? Prove your answer.

$$\{wab^n \mid n \in \mathbb{N} \land |w| = n - 1\}.$$

Ex. 7.6. Using the fact that $\{a^n b^n \mid n \in \mathbb{N}_0\}$ is not regular, show that the language

$$L = \{a^n b^m \mid m \geq n \geq 0\}$$

is not regular as well.

Solution. 1. ***Assume:*** L is regular.

2. $L^R = \{b^m a^n \mid m \geq n \geq 0\}$ is regular: by Theorem 5.7.
3. $L' = \{a^m b^n \mid m \geq n \geq 0\}$ is regular: rename a as b and b as a.
4. $L \cap L'$ is regular: by Theorem 5.3 and lines 1, 3.
5. $L \cap L' = \{a^n b^n \mid n \geq 0\}$: $a^n b^m \in L \cap L'$ iff $n = m$.
6. $L \cap L'$ is not regular (the known non-regular language).
7. Contradiction on lines 4, 6.
8. ***Hence,*** L is not regular.

□

Ex. 7.7. The language $\{ww \mid w \in \{a, b\}^*\}$ is not context-free. Give a context-sensitive grammar generating this language.

7.4 Pushdown Automata

We describe here a type of finite-state machines that are appropriate for parsing (recognizing) complex strings such as arithmetic expressions. The machines are called *deterministic pushdown automata* (DPDA). We shall omit the general case of nondeterministic pushdown automata (NPDA), but we note that (i) the class of languages recognized by NPDA is exactly the class of context-free languages, and (ii) the class of languages recognized by DPDA is a proper subset of the class of context-free languages.

A (deterministic) PDA

$$M = (Q, \Sigma, \Gamma, q_0, F, T)$$

is an NFA equipped with a ***stack*** data structure (a string) where symbols of the stack alphabet Γ can be pushed (added) on, or popped (deleted) from, the top of the stack (= beginning of the string), depending on the current state, current input symbol, and the top symbol of the stack. Thus, a transition of M is a tuple of the form

$$(p, \sigma, t/z, q)$$

and says that, at state $p \in Q$, if the current input symbol is $\sigma \in \Sigma$ and the top stack symbol is $t \in \Gamma$, then M can replace t with $z \in \Gamma^*$ and go to state q. Note that replacing the top t of the stack with λ is equivalent to popping (deleting) from the stack, and replacing t with a string of the form yt is equivalent to pushing (adding) y on the top of the stack. The machine is deterministic: given state p, input σ and top of stack t, there is at most one transition that matches these characteristics, that is, a transition of the form $(p, \sigma, t/\_, \_)$.

As usual, there is a start state q_0 and a set F of final states. Initially, the stack consists of one special bottom symbol \$. Moreover we assume that every input word ends with the special end of input symbol # that is not in Σ. A computation of M is a sequence of consecutive transitions

$$(p_0, \sigma_1, t_1/z_1, p_1), (p_1, \sigma_2, t_2/z_2, p_2), \ldots, (p_{n-1}, \sigma_n, t_n/z_n, p_n),$$

such that when a transition with label "$\sigma_i, t_i/z_i$" is performed the top of the stack is t_i and the next input symbol is σ_i. In each transition, the machine reads the next input symbol and possibly alters the top

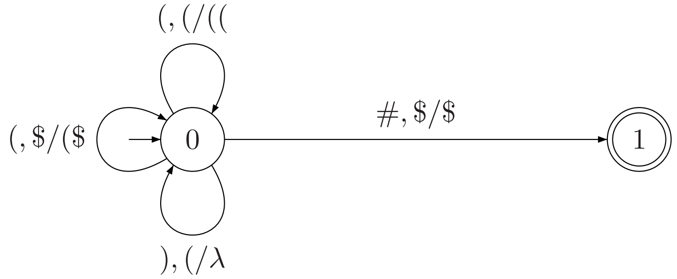


Figure 33: Deterministic PDA accepting the words of balanced parentheses.

of the stack. The computation is accepting if p_0 is the start state and p_n is a final state. The language accepted, or recognized, by M is the set of all words $\sigma_1 \cdots \sigma_n$ formed by concatenating the input labels in an accepting computation of M.

Ex. 7.8. We show a computation of the DPDA in Fig. 33 accepting the word `(())()`. We also show the stack contents before each transition. We use `$` for the bottom of the stack, and `^` for the empty word.

```
stack   transitions
  $     0, (, $ / ($, 0
 ($     0, (, ( / ((, 0
(($     0, ), ( / ^,  0
 ($     0, ), ( / ^,  0
  $     0, (, $ / ($, 0
 ($     0, ), ( / ^,  0
  $     0, #, $ / $,  1
  $     end
```

• **Designing DPDAs**. Many simple context-free languages can be recognized by DPDAs having a few states. For more complex languages L, a starting point in designing a DPDA could be as follows. First, find the set F_λ of all possible symbols that appear in the beginning of the words of L. Then, for each symbol $\sigma \in \Sigma$, find the set F_σ of all possible ***follow-up symbols***, that is, the symbols that immediately follow σ in the words of L. Then use this information to construct the states and transitions of the DPDA.

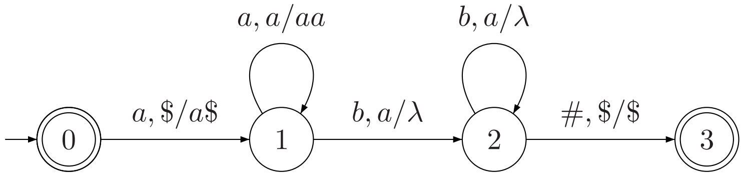


Figure 34: Deterministic PDA accepting $\{a^i b^i \mid i \geq 0\}$.

Ex. 7.9. Design a DPDA accepting all function calls of the form

`f() f(x) f(x,f(x)) f(f(),f(x,f())) ....`

More specifically, we want the function calls generated by the following grammar

$$S;\ S \to \mathtt{f()} \mid \mathtt{f(}L\mathtt{)};\ L \to P \mid P\mathtt{,}L;\ P \to \mathtt{x} \mid S$$

where '`,`' is part of the terminal alphabet and is used to separate the parameters in a function call.

Solution. The terminal alphabet is { `f x , ( )` }. We observe that the "follow-up symbols" are as follows:

F_λ : `f`

$F_{\mathtt{f}}$: `(`

$F_{\mathtt{(}}$: `) x f`

$F_{\mathtt{)}}$: `) , #`

$F_{\mathtt{x}}$: `) ,`

$F_{\mathtt{,}}$: `f x`

Then, we use this information to design the states and transitions of the required DPDA – see Fig. 35. □

• **Designing DPDAs (cont'd)**. The language L for which we want to design a DPDA could be given via a context-free grammar. Recall here that DPDAs cannot recognize all context-free languages. If, however, we are convinced that the given language L is a DPDA language, we can use the rules of the grammar to help us in determining the possible symbols appearing next to a given σ. In particular we

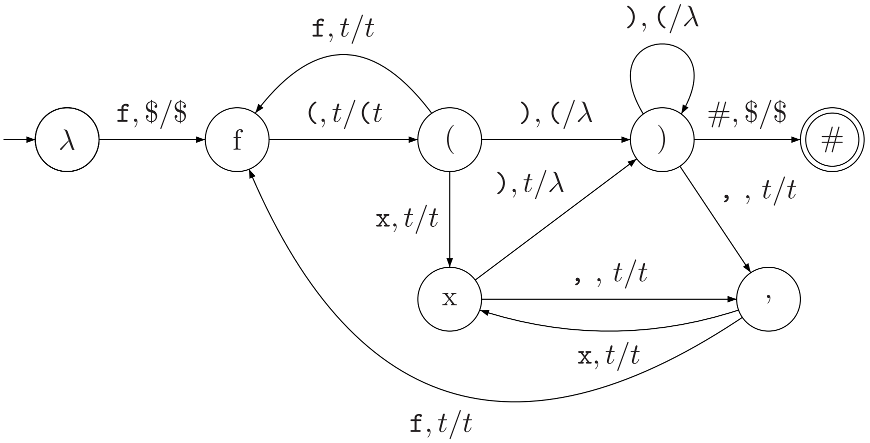


Figure 35: Deterministic PDA accepting function calls.

like to see rules of the form $X \to \sigma\beta$. In this process, we try to eliminate left recursive rules. These are rules of the form $X \to X\beta$, that is, the right-hand side of the rule starts with the variable of the left-hand side. More specifically, we replace all left recursive rules with a set of rules that are not left recursive such that the new grammar generates the same language.

• **Eliminating left recursion**. When we are given a context-free grammar containing a variable X with left recursive rules, we can replace them with non left recursive ones as follows. Let

$$X \to X\beta_1 \mid \cdots \mid X\beta_n$$

be all the left recursive rules for X, and let

$$X \to \alpha_1 \mid \cdots \mid \alpha_m$$

be all the other rules for X. We introduce a new variable B and replace all the above rules with the following ones

$$\begin{array}{lcl} X & \to & \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha_1 B \mid \cdots \mid \alpha_m B \\ B & \to & \beta_1 \mid \cdots \mid \beta_n \mid \beta_1 B \mid \cdots \mid \beta_n B. \end{array}$$

Of course this process is meaningful when all α_i's are nonempty. It can be shown that we can always convert the given grammar to one with no rules of the form $Z \to \lambda$, without changing the language it recognizes except possibly for the empty word [23].

7.5 Implementing PDAs in Prolog

We use an approach similar to that in the Implementation of NFAs – see Section 5.7. We use `$` as the stack bottom symbol. The transition predicate is `tr([Q,`β`,Z,P,R])` such that `Q` is the current state, β is a symbol or the empty word `[]`, `Z` is the stack top, `P` is the string that replaces `Z`, and `R` is the next state. Our implementation works also for nondeterministic PDAs, as it allows one to include λ-transitions. These are transitions of the form $(q, \lambda, z/p, r)$ that are implemented as `tr([Q,[],Z,P,R])`.

```
accept(Q, [], _) :- final(Q).
accept(Q,[C|T1],[Z|T2]) :- tr([Q,C,Z,P,R]),
                           append(P,T2,Y),
                           accept(R,T1,Y).

accept(Q,T1,[Z|T2]) :- tr([Q,[],Z,P,R]),
                       append(P,T2,Y),
                       accept(R,T1,Y).

runPDA(W) :- start(Q), accept(Q,W,[$]), !.
runPDA(F,W) :- consult(F), runPDA(W).
```

Here is the Prolog description of the DPDA shown in Fig. 34.

```
states(4).
start(0).
final(0).
final(3).
tr([0, a, $, [a,$], 1]).
tr([1, a, a, [a,a], 1]).
tr([1, b, a, [],    2]).
tr([2, b, a, [],    2]).
tr([2, #, $, [$],   3]).
```

Exercises of Sections 7.4,7.5

Ex. 7.10. Show the computation of the DPDA in Fig. 34 on the word `aabb`. Show the stack contents before each transition.

Ex. 7.11. Design a DPDA accepting the language

$$\{a^n b^{2n} \mid n \in \mathbb{N}_0\}.$$

Ex. 7.12. Design a DPDA accepting arithmetic expressions involving only integers, the operations in $\{+, -, *, /\}$ and parentheses. For example, your DPDA should accept the expressions

```
   -27 -((+35)) (23*(-7+49))/(5-32) ....
```

Ex. 7.13. Write the Prolog description of the DPDA shown in Fig. 33.

8 TURING MACHINES & UNDECIDABILITY

In the 1930's there were several attempts to give a rigorous mathematical definition of the informal concept of ***algorithm***. The most successful attempt was made by Alan Turing in [22]. He defined a set of "machines" that are now called Turing Machines (TMs). A TM M is deterministic (unless noted otherwise) and accepts a certain language $\mathcal{L}(M)$. Unlike the previous machines we have seen, an arbitrary TM can be of the type that, for some input word, the computation is infinite, that is, the machine performs transitions forever, and never returns any value.

Informally, an arbitrary TM can be viewed as a program that attempts to make a decision on the given input word. The particular type of decision TMs are exactly those whose computations are finite for all input words. So, a decision TM can be viewed as a decision program (see Section 1.3) that always halts and returns an answer in {YES, NO}, for all possible input words. It is this type of TM that constitutes the mathematical definition of the concept of algorithm.

8.1 Turing Machines (TMs)

A (deterministic) Turing machine (TM) is a tuple

$$M = (Q, \Sigma, \Gamma, q_0, T, F)$$

such that the following hold.

- Q is a finite nonempty set of states, which includes the start, or initial, state q_0, and F is a subset of Q called the set of final states.
- The machine has a tape consisting of cells such that the first cell is on the left end of the tape, and the tape is unbounded on the right (conceptually, there are infinitely many cells on the right). Initially, each cell contains the blank symbol $\mathbf{B}$, except for the cells containing the ***input word*** over the input alphabet Σ, which is written starting on the second cell of the tape (one input symbol per cell). The alphabet Σ is a subset of the tape alphabet Γ and $\mathbf{B} \in \Gamma - \Sigma$. There is a read/write

Head that is placed initially on the ***second*** cell of the tape. So when the input is *abbab* the initial tape contents are

$$\mathbf{B}\, a'\, b\, b\, a\, b\, \mathbf{B}$$

where the prime indicates the position of the Head. Conceptually, there are infinitely many blank cells following the last blank cell.

- T is the set of transitions. Each one is of the form $(q, t/s, \mu, p)$ and has the following meaning: if at state q the Head reads the tape symbol t, then t is replaced with the tape symbol s, the Head performs the one cell move $\mu \in \{\text{Left, Right}\}$, and the machine goes to state p. As the machine is deterministic, each state q has at most one transition of the form $(q, t/\_, \_, \_)$ where t is a tape symbol. In particular, there are no transitions going out of the final states, that is, if f is a final state then there are ***no*** transitions of the form $(f, \_/\_, \_, \_)$. We note that an attempt to move the Head to the left of the first cell would have no effect, that is, the Head would remain on the first cell. On the other hand, the Head can always move to the right.

• **Language accepted by a TM** M.
Given an input word $w \in \Sigma^*$, a Turing machine M starts at the start state with the input w and the Head initialized as noted above, and performs transitions. Then there are three cases:

1. The machine ***halts*** at a final state. In this case, we say that M accepts the input w and returns YES:

$$M(w) = \text{YES}.$$

2. The machine ***halts*** at a non-final state. In this case, we say that M does not accept (or rejects) w and returns NO:

$$M(w) = \text{NO}.$$

3. The machine ***loops***, that is, it continuously performs transitions never reaching a final state. In this case, we say that again M does not accept (or rejects) the input w, but now M does not return any value:

$$M(w) = \infty.$$

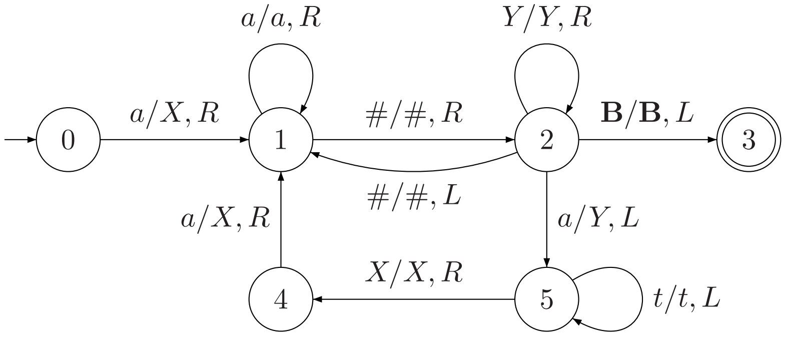


Figure 36: Turing machine accepting $\{a^n\#a^m \mid n > m \geq 0\}$. The machine loops on certain non-accepted input words.

A computation of M is a sequence of consecutive transitions

$$(p_0, t_1/s_1, \mu_1, p_1), (p_1, t_2/s_2, \mu_2, p_2), \ldots, (p_{n-1}, t_n/s_n, \mu_n, p_n),$$

such that when a transition with label "$t_i/s_i, \mu_i$" is performed the Head is on a cell containing the symbol t_i. If p_0 is the start state, p_n is a final state, and the computation starts with the tape containing initially some input $w \in \Sigma^*$, then the computation is called an accepting computation. Obviously, M accepts a word w iff the computation of M on input w is accepting.

Definition 8.1. The language $\mathcal{L}(M)$ accepted by a TM M is the set of all input words accepted by M, or equivalently the set of all words w such that $M(w) = \text{YES}$, or equivalently the set of all words w such that the computation of M on w is accepting. In mathematical notation:

$$\begin{aligned}\mathcal{L}(M) &= \{w \in \Sigma^* \mid M(w) = \text{YES}\} \\ &= \{w \in \Sigma^* \mid \text{the computation of } M \text{ on input } w \text{ is accepting}\}\end{aligned}$$

• **State diagrams for TMs**. As in the case of automata, we shall draw state diagrams for TMs using the same notation for drawing states and transitions.

Ex. 8.1. The TM in Fig. 36 accepts input words of the form $a^n\#a^m$, with $n > m \geq 0$, where the input alphabet is $\{a, \#\}$. The tape alphabet also includes the symbols X, Y. The machine expects to see first the symbol a, which it replaces with X, and then skips any a's and the # until it, either finds a blank in which case it halts at a final state, or finds an a, which it replaces with Y and goes to state 5 where then goes back left to find more a's – there should be more a's on the left of # than on the right.

Ex. 8.2. Next we show the accepting computation of the TM of Fig. 36 on input $aa\#a$. Initially, the tape contents are

```
   B a’ a # a B
```

where `B` is the blank symbol and `’` indicates the position of the Head (shown here on the second cell).

```
    Tape        Transitions
  B a’a # a B    0 a/X,R 1
  B X a’# a B    1 a/a,R 1
  B X a #’a B    1 #/#,R 2
  B X a # a’B    2 a/Y,L 5
  B X a #’Y B    5 #/#,L 5
  B X a’# Y B    5 a/a,L 5
  B X’a # Y B    5 X/X,R 4
  B X a’# Y B    4 a/X,R 1
  B X X #’Y B    1 #/#,R 2
  B X X # Y’B    2 Y/Y,R 2
  B X X # Y B’   2 B/B,L 3
  B X X # Y’B     halts
```

Note that if the input is of the form $a^n\#\#x$, for some $n \in \mathbb{N}$ and any word x, then the machine loops due to the transitions from states 1 to 2 and from 2 to 1. The next computation demonstrates this for the case where the input is $a\#\#$.

```
  Tape       Transitions
  B a’# # B   0 a/X,R 1
  B X #’# B   1 #/#,R 2
  B X # #’B   2 #/#,L 1
  B X #’# B    loops...(repeats forever the last 2 transitions)
```

8.2 Recursively enumerable languages

The next theorem asserts that unrestricted grammars and Turing machines describe the same class of languages. These languages are called recursively enumerable.

Theorem 8.1. $\mathcal{L}(\mathrm{UNG}) = \mathcal{L}(\mathrm{TM})$.

• **Why called "recursively enumerable" languages?** The following theorem explains the use of the terms 'recursively' and 'enumerable'. The second term has to do with enumerating, or listing, the words of the language in question. Of course, we have seen that every language is enumerable (= countable) in the mathematical sense. However, the term 'recursively' says that the enumeration is doable algorithmically – in the first days of computing theory one approach to defining algorithms was the recursive functions of Kleene [12].

Theorem 8.2. *If a language L is recursively enumerable then there is a program that (ultimately) prints w, for each $w \in L$.*

We note that the words of a recursively enumerable language are printed, in general, in no particular order.
□
People have proposed other formal methods for describing languages (including nondeterministic TMs – see the next sections). None of these formal methods can describe languages that are not recursively enumerable. On the other hand, we know that there exist languages that are not recursively enumerable (recall, the set of languages is uncountable and the set of grammars is countable). So the question is whether we can at least describe informally some non-recursively enumerable languages.

• **Turing Machine Encodings and the Diagonal Language**. A TM M is a mathematical object that can be expressed as a binary word $\langle M\rangle$ using a certain encoding method – we do something similar with numbers: a number is a mathematical concept which can be expressed in binary notation. This will allow us to give the encoding $\langle M\rangle$ as input to another Turing machine. This idea should not be surprising. For example a compiler, which is a program, takes as input a program P and tells us whether P contains any syntax errors.

Theorem 8.3. *The following language, called diagonal language, is not recursively enumerable*

$$L_{\mathrm{diag}} = \{\langle M\rangle \mid M \textit{ is a TM and } \langle M\rangle \notin \mathcal{L}(M)\}.$$

The above language consists of all TM encodings $\langle M\rangle$ such that M does not accept its own encoding as input!

Proof. By definition of L_{diag}, for any TM encoding $\langle M\rangle$, we have

$$\langle M\rangle \in L_{\text{diag}} \;\leftrightarrow\; \langle M\rangle \notin \mathcal{L}(M). \tag{2}$$

Assume for the sake of contradiction that L_{diag} is recursively enumerable, that is, there is a TM M_d accepting L_{diag}, hence, $L_{\text{diag}} = \mathcal{L}(M_d)$. Then in (2), if we instantiate M as M_d we get

$$\langle M_d\rangle \in \mathcal{L}(M_d) \leftrightarrow \langle M_d\rangle \notin \mathcal{L}(M_d),$$

which is a contradiction. Hence, the language L_{diag} is not recursively enumerable. □

• **Sample Encoding of Turing Machines**. Consider the prefix code[14]

$$C = \{11, 0^m 1 \mid m \in \mathbb{N}\} = (11 + 0^+1).$$

Given a TM $M = (Q, \Sigma, \Gamma, q_0, T, F)$ we shall encode the states Q, the tape symbols Γ, and the move symbols {Left, Right} using the codewords in 0^+1. The codeword 11 will be used as a separator. In particular,
– The start state q_0 is encoded as 01.
– The rest of the states are encoded using 001, 0001, etc.
– The blank is encoded as 01.
– The rest of the tape symbols are encoded using 001, 0001, etc.
– Left, Right are encoded as 01, 001, respectively.

If α is any of the above objects, we shall write $\bar{\alpha}$ for the codeword for α. For example, $\bar{q}_0 = 01$ and $\overline{\text{Right}} = 001$. Each transition $(p, t/s, \mu, q)$ is encoded as $\bar{p}\bar{t}\bar{s}\bar{\mu}\bar{q}$. If the sets of final states and transitions are

$$F = \{f_1, \ldots, f_m\} \text{ and } T = \{(p_i, t_i/s_i, \mu_i, q_i) \mid i = 1, \ldots, N\},$$

respectively, then we encode the machine M as

$$\langle M\rangle = \bar{f}_1 \cdots \bar{f}_m\, 11\, \bar{p}_1\bar{t}_1\bar{s}_1\bar{\mu}_1\bar{q}_1 \cdots \bar{p}_N\bar{t}_N\bar{s}_N\bar{\mu}_N\bar{q}_N 11,$$

[14] Recall a language C is called a prefix code, if no C-word is a prefix of another C-word, that is, C contains no two distinct words of the form w and wx.

that is, the prefix $\bar{f}_1 \cdots \bar{f}_m\, 11$ indicates the final states terminated with 11, and the rest of the encoding is the list of transitions followed by 11. For example, with respect to the TM M in Fig. 36, assuming that the codewords for states 1, 2, 3 are 001, 0001, 00001, respectively, for a is 001, for # is 0001, and for X is 00001, then the first few bits of $\langle M \rangle$ are as follows

$$00001\,11\,0100100001001001\,001001001001001$$

Using the same notation we can encode any input word $w = \sigma_1 \cdots \sigma_n$ as a binary codeword

$$\langle w \rangle = \bar{\sigma}_1 \cdots \bar{\sigma}_n.$$

For example, $\langle a\#a \rangle = 0010001001$.

• **Universal Language / Universal Turing Machine**. The following language is of particular importance in computation theory:

$$\mathcal{L}(U) = \{\langle M \rangle \langle w \rangle \mid M \text{ halts on input } w\}.$$

This language is called the universal language and consists of all words of the form $\langle M \rangle \langle w \rangle$ such that $\langle M \rangle$ is the encoding of a TM M that halts on input w.

Theorem 8.4. *The universal language $\mathcal{L}(U)$ is recursively enumerable.*

The proof of the above theorem is based on the existence of a TM U accepting the universal language – the machine U is called *universal* as well. This machine simulates the execution of any TM M on any input w, that is,

$$U(\langle M \rangle \langle w \rangle) = M(w).$$

The universal machine resembles a modern general purpose computer that is capable of executing any program on a given input.

8.3 Decidable Problems / Recursive Languages

It is evident that the description of a TM can be viewed as a computer program and, in fact, it appears that TMs constitute the correct mathematical model for the concept of algorithm. However,

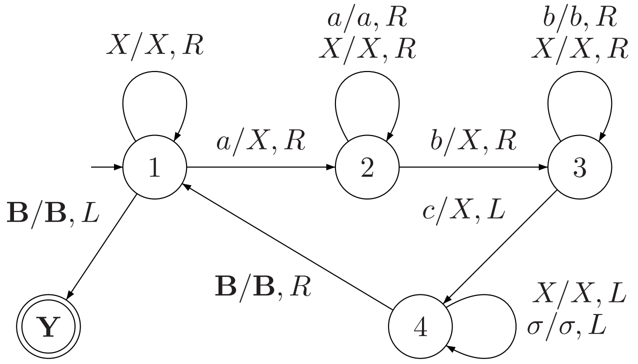


Figure 37: Decision Turing machine.

most people expect that an algorithm should behave nicely on all possible inputs, that is, the algorithm should always halt. For this reason, we restrict here our attention to TMs that halt on every input word. These machines formalize the concept of decision algorithm, or decision program, and can be used to solve decision problems[15].

• **Decision TMs and Recursive languages**. A decision TM is a Turing machine M that *halts on every input* w. A language L is called recursive if it is accepted by a decision TM.

Ex. 8.3. The decision TM in Figure 37 accepts the non-context-free language $L = \{a^n b^n c^n \mid n \in \mathbb{N}_0\}$. The input alphabet is $\Sigma = \{a, b, c\}$, and the tape alphabet is $\Gamma = \{a, b, c, \mathbf{B}, X\}$. The machine replaces the 1st a with X and then attempts to do the same with the 1st b and the 1st c. It goes back to the 1st cell (which is blank – see transition from state 4 to state 1) and repeats the previous action until there are no a's left. If the input is in the given language L then the machine halts at state **Y** and every input symbol has been replaced with an X.

Theorem 8.5. *If a language L is recursive then $\overline{L}$ is recursive.*

[15] Recall that a decision problem is a computing problem with exactly two possible answers, YES and NO, and that, mathematically, a decision problem is defined to be a language such that any word in the language corresponds to a YES-input and any word outside of the language corresponds to a NO-input.

• **Decidability and the Church-Turing thesis**. Recall from Section 1.3, a problem L is decidable if there is a decision program such that, on every input w, the program returns YES if w is in L, or returns NO if w is not in L.

Theorem 8.6. [16] *A problem L is decidable iff there is a decision TM accepting L (that is, L is recursive as a language).*

In practice, the above theorem says that anything we can decide using modern computers can also be accepted by decision TMs, and vice versa! In fact, there has been plenty of evidence that decision TMs can accept any languages that can be decided by any other means. For this reason the concept of decision TM is used as a rigorous mathematical definition for the informal concept of algorithm. This interpretation of TMs is known as the Church-Turing thesis. Here is another way to phrase it:

- For any possible algorithmic process that solves a decision problem, there is a decision TM solving that problem.

8.4 Undecidable problems / Non-recursive languages

Recall again in Section 1.3 we established that there are undecidable problems (or non-recursive languages) using the counting argument that there are more decision problems than programs. We shall give now concrete examples of these objects.

Theorem 8.7. *Every recursive language is recursively enumerable.*

Recall Theorem 8.3 says that the diagonal language L_{diag} is not recursively enumerable. Hence, the above theorem implies that L_{diag} is not recursive either. We give next more examples of non-recursive languages that are more relevant when viewed as decision problems.

• **The Halting problem**. The first undecidable problem was discovered by Turing [22]. It is called the Halting problem. In terms of languages, this is the universal language $\mathcal{L}(U)$ and the question is whether, given $\langle M \rangle$ and $\langle w \rangle$, the TM M halts on input w. In modern

[16] The statement of this theorem relies on a mathematical definition of 'decision program'. Such a definition can be found, for instance, in [13] under the name Random Access Turing Machine.

terminology, the input consists of[17] a program P and a string w, and the question is whether the program P will halt on input w, that is,

$$P(w) \neq \infty\ ?$$

(or P contains no reachable infinite loop).

Theorem 8.8. *The universal language $\mathcal{L}(U)$ is not recursive. Equivalently, the Halting problem is undecidable (or, there is no decision program that computes the Halting problem).*

• **Informal Proof**[18]. We argue by contradiction, that is, we assume there is a decision program H that computes the Halting problem, that is, H behaves as follows when given a program P and string w:

$$H(P,w) = \begin{cases} \text{YES}, & \text{if } P(w) \neq \infty \\ \text{NO}, & \text{if } P(w) = \infty \end{cases}$$

Now we can use H to make another program D as follows.

```
D(X)  {
  while (H(X,X)==YES) do nothing;
  return YES;
}
```

If D is given a program P as input, then D loops exactly when $H(P,P)$ returns YES. Thus, D behaves as follows

$$\text{For every program } P,\quad D(P) = \infty \text{ iff } P(P) \neq \infty.$$

Then, if we take $P = D$, we have that

$$D(D) = \infty \text{ iff } D(D) \neq \infty$$

which is impossible. Hence, the program H cannot exist.
□

[17] To be more precise, the input could also be a string that does not consist of a program P and string w. However, in this case, we simply agree that the answer to the problem is YES.

[18] This proof is not rigorous, as it does not refer to a concrete definition of decision program. It can be made rigorous, however, if we argue in terms of decision TMs.

• **The Post Correspondence Problem (PCP)**. This is a surprisingly simple looking problem that is undecidable and has been used to prove that numerous problems about grammars are undecidable. A domino is a pair of words (u, v) written vertically:

$$\binom{u}{v}$$

Given a set of at least two dominos the question is whether we can write a finite sequence of such dominos so that the word formed on the top part of the dominos is equal to the word formed on the bottom part. For example, for the following set of dominos

$$\binom{a}{aa}, \binom{bb}{b}, \binom{a}{bb},$$

we can see that the answer is YES:

$$\binom{a}{aa}\binom{a}{bb}\binom{bb}{b}\binom{bb}{b}.$$

Theorem 8.9. *The Post Correspondence problem is undecidable.*

• **The Program Equivalence Problem**. Here the input consists of two programs P_1 and P_2, and the question is whether, for each input w, the two programs will return the same value.

Theorem 8.10. *The Program Equivalence problem is undecidable.*

We note that the above problem remains undecidable even when one of the programs is fixed, that is, when the input consists of only one program.

Exercises of Section 8

Ex. 8.4. Show the computation of the Turing machine in Ex. 8.3 on the input word `abca`.

Ex. 8.5. Draw the state diagram of a decision Turing machine accepting the language $\mathcal{L}(a^*b^+a^*)$.

Solution. The machine is shown in Fig. 38. □

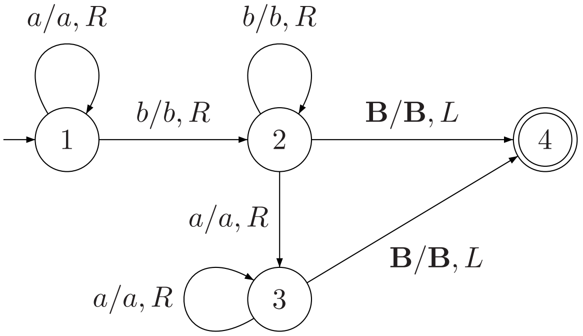


Figure 38: Decision Turing machine accepting $\mathcal{L}(a^*b^+a^*)$

Ex. 8.6. Draw the state diagram of a decision Turing machine accepting the language

$$\{w \mid w \in \{0,1\}^* \land w = w^R\}.$$

Ex. 8.7. With respect to Fig. 38, assume that
– $\mathbf{B}, a, b$ are encoded as 01, 001, 0001 (respectively).
– 1, 2, 3, 4 are encoded as 01, 001, 0001, 00001 (respectively).
– Left, Right are encoded as 01, 001 (respectively).
Show the binary encoding of the TM shown in Fig. 38.

Ex. 8.8. Draw the state diagram of a Turing machine that takes as input a binary word and replaces the word with its reversal. For example, if the initial tape contents are `B 0 1 B` the machine will change them to `B 1 0 B`.

Ex. 8.9. Is there a PCP solution for the following set of dominos

$$\binom{b}{a}, \binom{a}{ab}, \binom{ba}{ab}?$$

9 COMPUTATIONAL COMPLEXITY

In this section, we consider only problems that are decidable, that is, solvable via decision Turing machines. However, now we are interested in the time complexity of decision TM computations, in the first place, and also in the time *complexity of decision problems*. The time for a TM computation is simply the ***length*** of the computation, that is, the number of transitions that occur in the computation.

It turns out that, for some important problems that are decidable, nobody has ever come up with Turing machines (algorithms) to solve these problems ***efficiently***. Efficiency here has to do with the time and space resources required to compute problems. On the other hand, if we define nondeterministic Turing machines, ***NTM***s, these problems can be solved efficiently. So the question then becomes whether we can convert efficiently a nondeterministic Turing machine to an equivalent deterministic one. These considerations are discussed in the next sections.

9.1 Polynomial Complexity

Let M be a decision TM. For each nonnegative integer n, we define

- $T(n)$ to be the length of the longest computation of M on any input word of length n. The function $T(n)$ is called the (worst case) ***time*** complexity of M.

In most cases, instead of the exact values of the function $T(n)$, we are interested in the order of magnitude of this function. We say that the (worst case) time *complexity of a decision problem* is $O(f(n))$, if there is a decision TM that decides the language of the problem in time $T(n)$, with $T(n) = O(f(n))$.

Ex. 9.1. Consider the TM M in Fig. 37. Let w be any input word of length n. If w does not start with an a then M halts immediately. If w starts with exactly m a's, for some $m \geq 1$, then w is of the form $a^m z$. Obviously w will be accepted iff $z = b^m c^m$. In particular, for each letter a, the machine M attempts to read the tape contents from left to right replacing one a, one b, and one c with X, and then M moves the Head to the left end of the tape. If z is not $b^m c^m$ then M would not be able to complete this task and would halt at a nonfinal state. Else the machine would read the tape contents two times for each letter a. So the tape content is read $\leq 2m$ times. In the end, M could read once again the tape contents to make sure that only X's are present. So the tape content is read $\leq 2m + 1 = O(m)$ times. As the tape always contains exactly n non-blank cells, the machines performs $O(n)$ transitions times $O(m)$, in the worst case. Moreover, as $m \leq n$, we have that, in total, the machine performs $O(n^2)$ transitions on any input of length n, that is, $T(n) = O(n^2)$.

• **Efficiency = Polynomial complexity**. In practice, it is difficult to give a satisfactory definition of the concept ***efficient*** TM (or algorithm). The most successful definition is via polynomial complexities. In particular, in terms of time, a decision TM is efficient if the time complexity $T(n)$ of the machine is $O(n^k)$, for some constant nonnegative integer k. This definition makes sense when we realize that algorithms with exponential time complexities of the form 2^n, or higher, are too expensive in practice. For example, even for a small input of size $n = 40$, the number 2^{40} is greater than 1 trillion. Of course, for a large constant k such as $k = 100$, an algorithm of time complexity n^k is impractical as well. However, in practice most known polynomial time algorithms have a small value for k.

Language class $\mathcal{P} =$ All problems L for which there is a decision TM accepting L whose time complexity is $O(n^k)$, for some $k \in \mathbb{N}_0$, (i.e., polynomially bounded).

Note: Problems in $\mathcal{P}$ are also called *tractable*. Those not in $\mathcal{P}$ are called *intractable*. Again, recall that, in both cases, the problems are decidable.

• **Important type of question**. Given a decidable problem L, we would like to know whether L is in $\mathcal{P}$ (i.e., whether there is an efficient algorithm to compute the problem).

• **An intractable language**. Generally it is difficult to find examples of practical problems that are provably intractable (i.e., not in $\mathcal{P}$). A somewhat artificial problem is given next.

Theorem 9.1. *The following problem is decidable but intractable, that is, the language is recursive but not in the class* $\mathcal{P}$

$$\{\langle M\rangle\langle w\rangle \mid \textit{the TM } M \textit{ accepts } w \textit{ using} \leq 2^{|w|} \textit{ transitions}\}$$

9.2 Decision problems of unknown time-complexity

There are many decidable problems of practical significance for which nobody has ever exhibited polynomial time algorithms to decide such problems. Such problems are usually called ***combinatorial problems***. Next we list a few of these problems.

• **SAT = the Satisfiability problem**. Given a set of propositional clauses involving n variables, decide whether the set is satisfiable (i.e., whether there is a *model* = a valuation for which all clauses are true). The problem is decidable because we can construct the truth table for the given clauses and find whether one of the valuations in the table is a model. However, this is an exponential process, as the table involves 2^n rows. Nobody has ever found a polynomial-time algorithm for this problem.

• **3SAT**. This problem is the same as **SAT** with the restriction that each clause contains exactly three literals. Despite this restriction, the time-complexity of this problem is not known to be polynomial.

• **The Travelling Salesman problem**. Given a number of cities and the costs of travelling from any city to any other city, what is the least-cost round-trip route that visits each city exactly once and then returns to the starting city? This is not a decision problem. However, we can modify it to a decision problem as follows: given a bound B and the cities data, decide whether there is a round-trip route whose cost is at most B.

• **The Hamilton Cycle problem**. Given a graph, decide whether there is a cycle that goes through every vertex exactly once.

• **The integer partition problem**. Given a set of integers, decide whether we can partition the set into two parts such that the sums of the integers in the two parts are equal.

• **Nondeterministic algorithms**. The previous problems have the following common characteristic. There are many possible *combinations of tests* we can perform on the given input in order to decide whether to return YES. Usually this number is exponential with respect to the size of the input. For example in the 3SAT problem, if the input involves n different variables, there are 2^n possible valuations, and each valuation is *tested* to see whether it makes the clauses true. A *nondeterministic algorithm* would perform all tests in parallel and return YES if at least one test succeeds, or NO otherwise. We also say that the nondeterministic algorithm always ***guesses*** the right test for the given input, if indeed it is a YES-input. The time required to perform a test is usually polynomially bounded.

Ex. 9.2. *[A nondeterministic algorithm for 3SAT]* Given the input clauses $C_1, \ldots, C_k$ involving n different variables $p_1, \ldots, p_n$, do the following

```
- Guess a combination val $= (v_1, \ldots, v_n)$ of
truth values for the $n$ variables.
    – For each clause $C_i$, test whether val$(C_i)$ is $\mathbf{T}$.
    – If all $C_i$'s are true under val, output YES.
- Output NO.
```

Being nondeterministic, the above algorithm guesses a valuation val that makes all clauses true, if indeed such a valuation exists. Evaluating whether C_i is true requires looking up the truth values of the three variables in C_i. This is an $O(n)$-time task. Thus, evaluating k clauses takes time $O(kn)$, which is polynomially bounded. If the guessed valuation does not make all clauses true, then the output is NO.

9.3 NP-completeness

Many combinatorial problems like 3SAT can be solved in polynomial time using nondeterministic algorithms. The concept of nondeterministic algorithm is formalized next using nondeterministic TMs.

• **Nondeterministic TMs (NTMs).** In a nondeterministic TM M, for a given state q and tape symbol t, there could be two different transitions of the form $(q, t/\_, \_, \_)$. A word w belongs to $\mathcal{L}(M)$ iff there is at least one accepting computation of M on input w. A decision NTM is a NTM M that halts on every input.

Theorem 9.2. *We have that $\mathcal{L}(NTM) = \mathcal{L}(TM)$, that is, a language is accepted by some NTM, iff it is accepted by some (ordinary) TM. Moreover, a language L is accepted by some decision NTM iff L is accepted by some decision TM (that is, L is recursive).*

Language class $\mathcal{NP}$ = All problems L for which there is a *nondeterministic* decision TM M accepting L and the time complexity of M is $O(n^k)$, for some $k \in \mathbb{N}_0$, (i.e., polynomially bounded).

• **Problems in $\mathcal{NP}$.** All the problems listed in Section 9.2 belong to the class $\mathcal{NP}$, that is, there are NTMs deciding these problems in polynomial time. Unfortunately, as noted already, we know of no ordinary (deterministic) algorithms deciding these problems in polynomial time.

• **The most famous open problem in CS**.

Is $\mathcal{P} = \mathcal{NP}$?

We know for fact that $\mathcal{P} \subseteq \mathcal{NP}$, but nobody has ever proved the converse. In fact, most people ***believe*** that the two classes are ***not equal***. The most promising approach to address the $\mathcal{P} = \mathcal{NP}$ question is the theory of NP-completeness, which requires the concept of polynomial reduction.

• **Problem reduction**. Informally, we say that problem L reduces to L' to mean that if I know how to decide L' then also L can be decided by mapping somehow L to L'. More formally, we say that L is (effectively) reducible to L', if there is a program P that halts on all inputs w such that "$w \in L$ iff $P(w) \in L'$." Thus, if I can decide L' then I can also decide L as follows: Given an input w, run P to get the value $P(w)$. Then, decide whether $P(w) \in L'$. If YES then also $w \in L$ so return YES; else return NO.

• **Polynomial reductions**. We say that problem L is polynomially reducible to L' if there is a program as above whose time complexity is polynomially bounded.

• **NP-complete problem**. This is a decision problem L that (i) belongs to the class $\mathcal{NP}$, and (ii) every problem in $\mathcal{NP}$ is polynomially reducible to L.

• **Why is this important?**. The significance of this concept is that if we can solve an NP-complete problem L deterministically in polynomial time then every other problem in $\mathcal{NP}$ can also be solved deterministically in polynomial time. Hence, this would imply that $\mathcal{P} = \mathcal{NP}$. Obviously, if L is proved to be not in $\mathcal{P}$ then $\mathcal{P} \neq \mathcal{NP}$.

Theorem 9.3. *(Cook-Levin Theorem) There exist NP-complete problems.*

• **How to show that a problem L is NP-complete using a known NP-complete problem L'.** *First* we show that L is in NP, that is, there is a nondeterministic algorithm (or NTM) deciding L in polynomial time (at most). *Second*, we show that there is a polynomial reduction from L' to L, that is, a polynomially bounded algorithm P such that "$w \in L'$ iff $P(w) \in L$."

Ex. 9.3. Explain how you would show that the Hamilton Cycle problem is NP-complete using the fact that SAT is NP-complete.

Solution. First I would show a polynomially bounded nondeterministic algorithm that decides whether a given graph contains a Hamilton cycle. Then I would show a polynomially bounded algorithm P that maps every set of clauses C to a graph $P(C)$ such that C is satisfiable iff the graph contains a Hamilton cycle. □

10 APPENDIX: various proofs

Here we list the proofs of a few of the theorems in the preceding sections.

• **Proof of Lemma 6.4**. We follow the presentation in [5]. We assume that the source involves n events. For the first part we shall use the Kraft inequality (see Theorem 6.1) and the fact that, for every real $x > 0$,

$$\log x \leq (x-1)\log e,$$

where e is the constant $2.718281\cdots$. We assume that $\ell_1, \ldots, \ell_n$ are the lengths of the words in K. We have

$$\begin{aligned}
H(\tilde{p}) - \ell_{\tilde{p}}(K) &= -\sum_i \tilde{p}(i)\log\tilde{p}(i) - \sum_i \tilde{p}(i)\ell_i \\
&= -\sum_i \tilde{p}(i)(\log\tilde{p}(i) + \ell_i) \\
&= -\sum_i \tilde{p}(i)(\log\tilde{p}(i) + \log 2^{\ell_i}) \\
&= \sum_i (\tilde{p}(i)\log\frac{1}{\tilde{p}(i)2^{\ell_i}}) \\
&\leq \sum_i \tilde{p}(i)(\frac{1}{\tilde{p}(i)2^{\ell_i}} - 1)\log e \\
&= \log e \sum_i (2^{-\ell_i} - \tilde{p}(i)) \\
&= \log e(\sum_i 2^{-\ell_i} - \sum_i \tilde{p}(i)) \\
&\leq \log e\,(1-1) = 0.
\end{aligned}$$

For the second part we use again the Kraft inequality and the concept of Shannon length, which is the quantity $\ell_i = \lceil -\log\tilde{p}(i)\rceil$. This gives that $\ell_i \geq -\log\tilde{p}(i)$ and $2^{-\ell_i} \leq \tilde{p}(i)$. If we take the sum over all the i's we get

$$\sum_i 2^{-\ell_i} \leq \sum \tilde{p}(i) = 1.$$

By Theorem 6.1, there is a prefix code K_0 of n codewords whose words have lengths $\ell_1, \ldots, \ell_n$. As $\ell_i < -\log \tilde{p}(i) + 1$, we have

$$\ell_{\tilde{p}}(K_0) = \sum_i \tilde{p}(i)\ell_i < \sum_i \tilde{p}(i)(-\log \tilde{p}(i) + 1) = H(\tilde{p}) + 1.$$

□

• **Proof of Lemma 6.6**. If we show that, for $k \geq 2$, $H(\tilde{p}^k) = H(\tilde{p}) + H(\tilde{p}^{k-1})$ then the claim will follow easily. We use the term P_{k-1} as a shorthand for $\tilde{p}(i_1) \cdots \tilde{p}(i_{k-1})$. So we have

$$\begin{aligned}
H(\tilde{p}^k) &= -\sum_{i_1,\ldots,i_k \in E_{\tilde{p}}} \tilde{p}(i_1) \cdots \tilde{p}(i_k) \log(\tilde{p}(i_1) \cdots \tilde{p}(i_k)) \\
&= -\sum_{i_k \in E_{\tilde{p}}} \tilde{p}(i_k) \sum_{i_1,\ldots,i_{k-1} \in E_{\tilde{p}}} P_{k-1}[\log(P_{k-1}) + \log \tilde{p}(i_k)] \\
&= -\sum_{i_k \in E_{\tilde{p}}} \tilde{p}(i_k) \sum_{i_1,\ldots,i_{k-1} \in E_{\tilde{p}}} P_{k-1} \log P_{k-1} \\
&\quad -\sum_{i_k \in E_{\tilde{p}}} \tilde{p}(i_k) \sum_{i_1,\ldots,i_{k-1} \in E_{\tilde{p}}} P_{k-1} \log \tilde{p}(i_k) \\
&= (\sum_{i_k \in E_{\tilde{p}}} \tilde{p}(i_k)) \cdot H(\tilde{p}^{k-1}) + H(\tilde{p}) \sum_{i_1,\ldots,i_{k-1} \in E_{\tilde{p}}} P_{k-1} \\
&= 1 \cdot H(\tilde{p}^{k-1}) + H(\tilde{p}) \cdot 1 = H(\tilde{p}^{k-1}) + H(\tilde{p}).
\end{aligned}$$

□

• **Proof of Theorem 7.1**. Suppose the language is regular. Then there is an NFA with N, say, states accepting the language. As the word $a^N b^N$ is in the language there is an accepting computation

$$(q_0, a, q_1), \ldots, (q_{N-1}, a, q_N), (q_N, b, q_{N+1}), \ldots, (q_{2N-1}, b, q_{2N}).$$

As the NFA has N states and the first N transitions above contain $N+1$ states: $q_0, q_1, \ldots, q_N$, it follows that at least one of these states is repeated, that is, there are two indices i and j, such that

$$i < j \tag{3}$$

and $q_i = q_j$. Now we modify the above computation by removing the transitions $(q_i, a, q_{i+1}), \ldots, (q_{j-1}, a, q_j)$ and we get a new ***accepting*** computation

$$(q_0, a, q_1), \ldots, (q_{i-1}, a, q_i), (q_j, a, q_{j+1}), \ldots, (q_{N-1}, a, q_N),$$
$$(q_N, b, q_{N+1}), \ldots, (q_{2N-1}, b, q_{2N})$$

whose label is the word $a^{N+i-j}b^N$ and, therefore, that word must be in the language. So we must have $N + i - j = N$, which implies $i = j$, contradicting (3). Hence, there can be no NFA accepting the language and, therefore, the language is not regular.

□

References

[1] J. Berstel and D. Perrin. *Theory of Codes*. Academic Press, Orlando, 1985.

[2] P. Brna. *Prolog Programming*. Accessed in Nov. 2010, http://homepages.inf.ed.ac.uk/pbrna/prologbook/, 2001.

[3] C. Calude and H. Jürgensen. Is complexity a source of incompleteness?. *Advances in Applied Mathematics* **35** (2005), 1–15.

[4] M. Crochemore and C. Hancart. Automata for matching patterns. In [18], Vol. 2, pp 399–462.

[5] A. Drozdek. *Elements of Data Compression*. Brooks/Cole, Pacific Grove, CA, 2002.

[6] J. Duske and H. Jürgensen. *Codierungstheorie*. BI Wissenschaftsverlag, Manheim, 1977.

[7] R.W. Hamming. *Coding and Information Theory*. Prentice Hall, Englewood Cliffs, NJ, 1980.

[8] H. Hamburger and D. Richards. *Logic and Language Models for Computer Science*. Prentice Hall, New Jersey, 2002.

[9] H. Jürgensen. *Logic for Computer Science – Lecture Notes*. The University of Western Ontario, 1994.

[10] H. Jürgensen and S. Konstantinidis. Codes. In [18], Vol. 1, pp 511–607.

[11] L. Kari. *Logic for Computer Science – Lecture Notes*. The University of Western Ontario, 2007.

[12] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen* **112** (1936), 727–742.

[13] H. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation, 2nd ed*. Prentice Hall, 1998.

[14] B.H. Marcus, R.M. Roth and P.H. Siegel. Constrained Systems and Coding for Recording Channels. In [16], pp 1635–1764.

[15] A. Mateescu and A. Salomaa. Aspects of classical language theory. In [18], Vol. 1, pp 175–251.

[16] V.S. Pless and W.C. Huffman (eds). *Handbook of Coding Theory.* Elsevier Science, 1998.

[17] S. Roman. *Coding and Information Theory.* Springer-Verlag, New York, 1992.

[18] G. Rozenberg and A. Salomma (eds). *Handbook of Formal Languages.* Springer-Verlag, Berlin, 1997.

[19] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* Univ. of Illinois Press, Urbana, 1949.

[20] W.A. Shay. *Understanding Data Communications & Networks – 2nd ed.* Brooks/Cole, Pacific Grove, CA, 1999.

[21] C. Smorynski. *Self-Reference and Modal Logic.* Springer-Verlag, New York, 1985.

[22] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceed. London Math. Society* **2**.42 (1936), 230–265. See also ibid. 2.43, 544–546.

[23] D. Wood. *Theory of Computation.* John Wiley, New York, NY, 1987.

[24] S. Yu. Regular Languages. In [18], Vol. 1, pp 41–110.