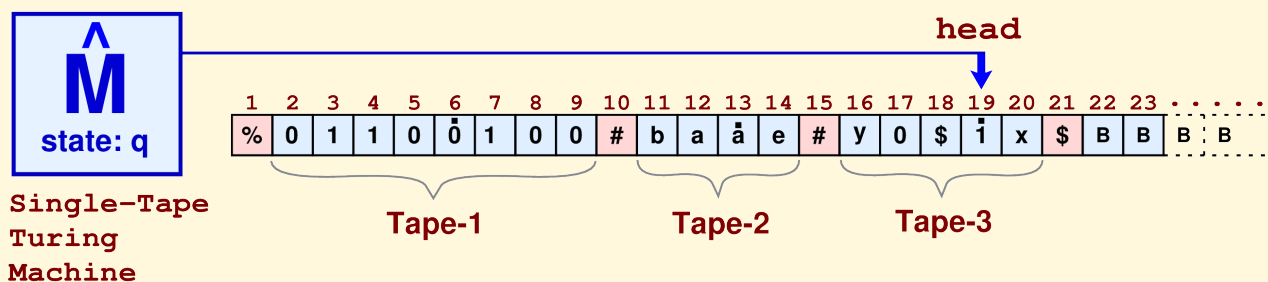
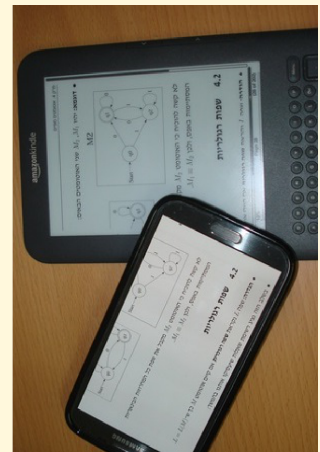
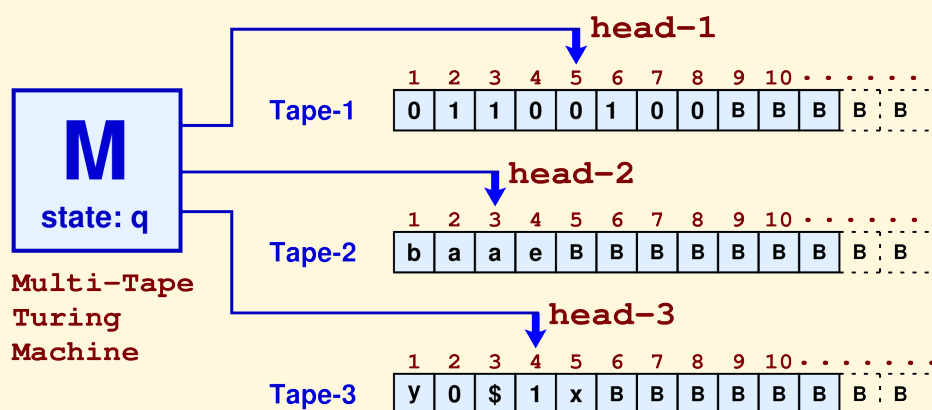


Computational Models and Formal Languages

Introduction to the Theory of Computation



Link to most updated version: https://samyzaf.com/afl_eng.pdf

Introduction

The present booklet was written and developed on the basis of many years of experience teaching courses on finite automata, formal languages, data structures, and algorithms, at Israeli universities and colleges. It was published online around 2015, in Hebrew, and has since undergone many improvements, expansions, and corrections.

It covers the fundamental material in computability theory and may also be suitable for similar courses at American and European universities and colleges. The idea of translating it into English, thereby making it accessible to a much larger audience than Hebrew speakers, had not been realized because of the considerable effort and time required. Recently, artificial intelligence agents such as  ChatGPT have demonstrated an impressive ability to translate text rapidly and efficiently, especially mathematical text, whose syntax is more rigid and structured. Thus, the idea of translating the booklet became feasible.

In light of the outstanding success of translating the [Complex Functions course booklet](#) from Hebrew into English, we decided to translate [our Hebrew booklet on computability](#) into English using  ChatGPT. The translation was completed in less than two hours! Several additional hours were devoted to corrections and adaptations to English, and we are pleased to present the English version of the booklet here. Since this is the first English version, we would be glad to receive feedback from readers.

The booklet aims to cover the main topics taught in a first academic course in computability theory. In the past, such a course was called “Automata and Formal Languages”, but in recent years additional computational models have been added, such as various kinds of Turing machines, logic circuits, register machines, and random-access machines (RAM). Such a course is generally taught in computer science or electrical and electronic engineering faculties and provides an essential foundation for most advanced theoretical courses in algorithms, complexity, programming languages, and various language models in artificial intelligence (LLM). For this reason, the title “Computational Models” is more appropriate.

The purpose of the booklet is to provide concise explanations and canonical examples of computational models using [snoitinfied lacitamehtam esicerp](#). Mathematical models enable us to understand and analyze the concept of **computation** in a precise formal manner, and to prove precise assertions about the nature of **the concept of computation**. Mathematical modeling enables us, for example, to prove that certain problems [devlos eb tonnac](#) by any computing machine, or that the computation time required to solve a problem is the shortest attainable among infinitely many possible methods. The concepts of computation and algorithm have in recent years become (following the expansion of the Internet and the widespread use of mobile phones and tablets) common in many fields, including economic and social discourse, and consequently the need for an introductory course on this subject continues to grow.

In version 2.6 of the booklet, new topics such as a pushdown automaton and Turing machines were added. In versions 3.0–4.2, topics on logic circuits and register machines were added, as were random-access machines. In version 4.6, Jupyter notebooks using Python packages were added for solving various problems involving automata and context-free languages (Chapters 2, 3, and 4). In future versions, an effort will be made to cover more advanced topics in complexity theory.

Adding solutions to the exercises at the end of each chapter will require considerable effort, which is unlikely to be available soon, and readers’ assistance is needed. We would be glad to receive solutions from readers. If a substantial number accumulates, we may be able to publish a solutions appendix.

The format of the booklet is designed primarily for convenient reading on tablets and the many kinds of electronic readers. Most of the text in this booklet is phrased in the masculine gender solely for convenience, but, of course, everything stated is intended equally for women and men, and we regret that we do not have a more satisfactory solution.

Contents

1	Mathematical Background and Basic Concepts	6
1.1	Introduction	6
1.2	Sets	8
1.3	Sequences	13
1.4	Functions (Mappings)	13
1.5	Graphs	14
1.6	Directed Graphs	15
1.7	Labeled Graphs	16
1.8	Proof Techniques	17
1.9	Strings and Languages	20
	Exercises 1	22
2	Finite Automata	26
2.1	Basic Concepts	26
2.2	Transition Diagrams	28
2.3	Regular Languages	30
2.4	Nondeterministic Automata	31
2.5	Equivalence of Deterministic and Nondeterministic Automata	36
2.6	Automata with ϵ -Transitions	39
	Exercises 2	46
3	Regular Languages	54
3.1	Regular Expressions	54
	The Degree of Complexity of a Regular Expression	56
	Equivalence of Regular Expressions and Finite Automata	58
3.2	Generalized Automata	61
3.3	Properties of Regular Languages	69
	The Pumping Lemma for Regular Languages	70
	Proof of the Pumping Lemma	72
	Closure Properties of Regular Languages	74
	Substitutions	76
	Homomorphism	78
3.4	Decision Algorithms for Regular Languages	81
	An Algorithm for Deciding Whether $L(M) \neq \emptyset$	81



An Algorithm for Deciding Whether $L(M)$ Is Infinite or Finite	82
An Algorithm for Deciding Whether $L(M_1) = L(M_2)$	83
3.5 The Myhill-Nerode Theorem	84
3.6 An Algorithm for Finding a Minimal Automaton	92
Isomorphism Between Two Automata	92
Existence and Uniqueness of an Equivalent Minimal Automaton	95
Example	98
Proof of the Correctness of the Algorithm	100
Exercises 3	102
4 Context-Free Languages	108
4.1 Context-Free Grammars	108
Notation Conventions	113
4.2 Additional Properties of the Derivation Relation	115
4.3 Regular Grammars	119
4.4 Derivation Trees and Derivation Sequences	123
4.5 The Relationship Between Derivation Trees and Derivation Sequences	125
4.6 Ambiguity	129
The algorithm:	130
4.7 Useful and Useless Variables	133
The Algorithm	135
Proof of the Correctness of the Algorithm	136
4.8 ϵ -Productions	141
4.9 Unit Productions	143
4.10 Chomsky Normal Form	145
Proof	146
Example	147
The Final Stage	147
4.11 An Algorithm for Deciding Membership	148
4.12 Greibach Normal Form	153
4.13 The Pumping Lemma for Context-Free Languages	153
4.14 Additional Properties of Context-Free Languages	157
4.15 Applications to Programming Languages and Natural Languages	159
Exercises 4	160
5 Nondeterministic Pushdown Automata	169
5.1 Definition	169
Graphical Representation	171
Accepting (accept) and Rejecting (reject) Input	172
5.2 Additional Clarifications	173
5.3 First Example	175
5.4 Second Example	178



5.5	Elementary Pushdown Automata	180
5.6	Equivalence of Context-Free Grammars and Pushdown Automata	181
	Exercises 5	190
6	Turing Machines	194
6.1	Definition	194
6.2	Computation Sequences	199
6.3	Additional Examples	201
6.4	Multitape Turing Machines	209
	Operation of a Multitape Turing Machine	210
	Example 1: Multiplication of Positive Integers	211
	Example 2: Binary Addition	216
6.5	Equivalence of Multitape and Ordinary Turing Machines	220
6.6	The Church-Turing Thesis	224
6.7	Example: An Algorithm for Testing Graph Connectivity	225
	Recursive and Decidable Languages	227
6.8	Encoding a Computational Model	227
6.9	The Halting Problem	232
6.10	Not the End	234
	Exercises 6	234
7	Logic Circuits	240
7.1	Boolean Algebra – A Brief Review	240
7.2	Logic Circuits	248
	Constructing Complex Circuits Using Library Cells	258
7.3	Simulation and Hardware-Description Languages for Logic Circuits	261
	A Logic-Circuit Programming Language (Simulation)	261
	A Hardware-Description Language for Logic Circuits (PyCircHDL)	264
	Constructing New Cells by Parameterization	270
7.4	Complexity of a Logic Circuit	273
7.5	Shannon's Estimate Theorem	281
	Binary Encoding of a Logic Circuit	284
7.6	Finished, but Not Complete	287
	Exercises 7	287
9	Random-Access Machines (RAM)	294
9.1	General Description	294
9.2	Run Tables	299
9.3	Generalized RAM Machine	300
9.4	Equivalence Between a RAM Machine and a Turing Machine	305
9.5	Finished, but Not Complete	308
	Exercises 9	309



Chapter 1

Mathematical Background and Basic Concepts

1.1 Introduction

- **Course Objective:** A first introduction to the fundamental models of computation: finite automata in all their forms, pushdown automata, and the various types of Turing machines. Understanding the concept of a computational model and its connection to formal languages and programming languages.
- **Prerequisites:** Basic academic knowledge of logic, set theory, graph theory, and combinatorics is required.
- **Recommended Reference Books**
 1. Automata and Formal Languages. Professor Shmuel Zaks, Professor Nissim Francez, Technion, Volumes א', ב'. The Open University Press.
 2. Computability Theory, Oded Goldreich, Technion, Haifa.
 3. Introduction to the Theory of Computation. Michael Sipser, PWS Publishing Company, Thomson Publishing, 7625 Empire Drive Florence, KY 41042.
 4. Introduction to Automata Theory, Languages, and Computation. John E. Hopcroft, Jefferey D. Ullman. Addison–Wesley series in computer science, ISBN 0–201–02988–X.
 5. An Introduction to Formal Languages and Automata. Peter Linz, Lexington, Massachusetts : D.C. Heath, c1990, QA267.3, L56, 1990.
 6. Computational Complexity. Papadimitriou C. H., Addison–Wesley 1994, Reading Massachusetts. ISBN/ISN 0–201–53082–1.



- The mathematical theory of the fundamental computational models and their associated formal languages is the first part of computability theory—the mathematical theory of computer science. In the past, the present course was offered at academic institutions as an introductory course in computability theory. More advanced topics (such as computational complexity) are covered in more advanced courses, and the list of recommended books above may be used for a preview or for further study.
- One of the goals of computability theory is to instill habits of rigorous mathematical thinking—the ability to provide precise mathematical justification for the correctness of software or hardware solutions to computational problems. This is particularly essential for the design, analysis, and verification of large and complex software systems.
- The many types of computing machines are hardware implementations of pure mathematical models that can be defined, investigated, and analyzed in a logical–mathematical manner independent of their hardware implementation. The same applies to complex software such as compilers and the many types of scanners. An essential preliminary stage before implementation (in hardware or in software written in a suitable programming language) is the analytical stage of formulating the problem, examining the feasibility of theoretical solutions, and proving the correctness of the optimal solution (if it exists).
- Sometimes, the theoretical analysis reveals that a given problem cannot be solved by a computer (of any type that exists now or will exist in the future), or that a solution exists but its computation time is impractical. Sometimes it turns out that a given problem can be solved in several ways, with one method being more efficient (in terms of computation time or memory usage) than another. In such cases, the question arises whether a particular method is the most efficient possible, and how an assertion of this kind can be proved.
- Computability theory, which provides the broader framework for automata theory, asks more fundamental questions such as:
 - ◆ What is computation?
 - ◆ What can and cannot be computed?
 - ◆ Are there problems that cannot be solved by a computer?
 - ◆ If so, how do we prove that such problems are unsolvable?
 - ◆ How do we prove that a computer program (or algorithm) actually solves the problem it is intended to solve?
 - ◆ How do we prove that a particular software solution is the fastest solution that can be achieved?
 - ◆ What is a formal language? What is a programming language? How are such languages defined and analyzed? How do we prove or verify that a computer program solves the problem it was designed to solve? How do we verify that two computer programs are equivalent?
- All the questions above are illustrated more clearly and precisely, and receive suitable answers, in the theory of finite automata and Turing machines.



- **Motivation:** Most of the topics studied in this course are essential for the design and analysis of complex software or hardware systems.
- For example, the theory of finite automata and regular expressions is needed for the design of logic circuits, compilers, scanners, command processors, text editors, and logic engines (such as Prolog).
- Automata and formal languages provide an important foundation for nonmathematical fields such as natural language processing, linguistics, and brain research.
- The theory of regular expressions has important applications in communications and the Internet. It is used in designing communication protocols and algorithms for the dynamic generation and scanning of HTML pages, and similar tasks. All of these are impossible without a thorough command of automata and regular expressions. Appropriate examples will be presented during the course.
- A sound logical definition of programming languages, compilers, and fast scanners is made possible by the mathematical theory of context-free languages (the second part of the course). It follows that a thorough understanding of the topics in this course is essential (and in fact provides the theoretical foundation) for advanced courses in computer science.

1.2 Sets

- The following is a brief review of the fundamental concepts from set theory, graph theory, and combinatorics that we will need. The purpose is to refresh the reader's memory through short reminders and examples, so we will not dwell on complete and detailed definitions. Readers who require a more thorough review will find an extensive and excellent treatment in the Open University book "Automata and Formal Languages," Volume One.
- Set theory is a relatively new branch of mathematics that was developed at the end of the 19th century, and led to the rigorous foundation and reformulation of older branches of mathematics, as well as the development of new mathematical branches with an incomparably higher degree of precision than had previously been customary.
- Modern set theory enabled the foundation and rapid development of new mathematical theories such as logic, graph theory, and a substantial part of the mathematical theory of computer science (computational models, complexity, formal languages, and so forth).
- General set theory deals with all possible types of sets. We can speak of a set of children, a set of tables, a set of countries, a set of events, and so forth. In computability theory and formal languages, we will be concerned primarily with sets of symbols and formal expressions.
- Sets are conventionally denoted by uppercase letters such as A , B , C . The objects of which a set is composed are called the **elements** of the set, and are usually denoted by lowercase letters: a , b , c , ...



➤ If a is an object that belongs to the set A , we say that a is **an element of A** and write $a \in A$. If b is **not** an element of A , we write $b \notin A$.

➤ There are several ways to describe a set. The following is a brief review of the principal forms of description.

◆ The set A whose elements are the numbers **1**, **3**, **5**, and **7** is described as follows:

$$A = \{1, 3, 5, 7\}$$

◆ The set of all odd numbers from **1** through $n = 99$ is described as follows:

$$B = \{1, 3, 5, \dots, 99\}$$

or in the following form:

$$B = \{2k - 1 : k = 1, 2, 3, \dots, 50\}$$

◆ The elements of a set can also be described verbally. For example, if C is the set of roots of the quadratic equation $x^2 - 9 = 0$, then we can write

$$C = \{x : x^2 - 9 = 0 \text{ noitauqe eht fo toor a si } x\}$$

In this case, a verbal description can be avoided by writing

$$C = \{x : x^2 - 9 = 0\}$$

Since the roots of the equation are $x = \pm 3$, we can simply write $C = \{-3, 3\}$.

◆ The set of all prime numbers P can be described verbally as follows:

$$P = \{n : \text{rebmun emirp larutan a si } n\}$$

We can also define the following set:

$$A = \{x \in \mathbb{R} : x^2 < 0\}$$

Clearly, no real number satisfies the condition $x^2 < 0$, and therefore there is no element belonging to A . In this case, the set A is called the **empty set**, and is denoted by $\emptyset$.

➤ To remove any doubt and avoid pitfalls, we briefly review several basic facts concerning the concept of a set.

◆ A set contains no duplicates! An element a may or may not belong to a set A , but multiple

membership (duplication) of a is impossible (unlike in certain data structures in computer science).

- ◆ The order of the elements in a set is of no importance!

$$\{1, 4, 7\} = \{7, 1, 4\} = \{4, 1, 7\} = \{7, 4, 1\} = \dots$$

- ◆ A distinction must be made between finite sets and infinite sets.
- ◆ Finite sets can be defined by listing their elements (in any order convenient for presentation), as in the following examples:

$$A = \{2, 4, 6, 8\}$$

$$B = \{3, 6, 9, 12, \dots, 300\}$$

$$C = \{n \in \mathbb{N} : 20 \leq n < 100 \vee 200 \leq n < 300\}$$

- ◆ Likewise, **infinite** sets can be defined in various ways, as in the following examples:

$$\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$$

$$A = \{2, 4, 6, 8, \dots\}$$

$$A = \{n \in \mathbb{N} : \text{neve si } n\}$$

$$A = \{n \in \mathbb{N} : \exists k \in \mathbb{N}[n = 2k]\}$$

- A set A is called a **subset** of a set B , and we write $A \subseteq B$, if every element of the set A belongs to the set B . In logical notation, this assertion is expressed by:

$$\forall x [x \in A \implies x \in B]$$

We sometimes also say that the set A is **included** or **contained** in the set B , or that the set B **contains** or **includes** the set A . If the set B is not equal to the set A , we write $A \subset B$ (or, for greater emphasis, $A \subsetneq B$).

- **Equality of Sets:** Two sets A and B are equal if they have exactly the same elements; that is, every element x belonging to the set A also belongs to the set B , and conversely, every element x belonging to the set B also belongs to the set A . This fact can also be expressed by saying that $A \subseteq B$ and $B \subseteq A$. We can write this in logical notation as follows:

$$A = B \iff A \subseteq B \wedge B \subseteq A$$

In logical notation, we can also write

$$A = B \iff \forall x [x \in A \iff x \in B]$$



✦ **Example 1.1:** Let $A = \{x \in \mathbb{R} : x^2 - 9 = 0\}$ and $B = \{-3, 3\}$. Then $A = B$.

✦ The **union** of two sets A and B is the set containing all the elements of A and all the elements of B .

$$A \cup B = \{x : x \in A \vee x \in B\}$$

✦ The **intersection** of two sets is defined by:

$$A \cap B = \{x : x \in A \wedge x \in B\}$$

✦ **Example 1.2:**

a. Let $A = \{2, 4, 6, 8, 10\}$ and $B = \{1, 4, 7, 10, 13\}$. Then

$$\begin{aligned} A \cup B &= \{1, 2, 4, 6, 7, 8, 10, 13\} \\ A \cap B &= \{4, 10\} \end{aligned}$$

b. If $A = \{1, 3, 5\}$ and $B = \{2, 4, 6\}$, then $A \cap B = \emptyset$.

✦ Two sets whose intersection is the empty set are called **disjoint**.

✦ The following is a list of the fundamental properties of sets and set operations:

- (a) $A \cap B = B \cap A$, $A \cup B = B \cup A$ (commutative law).
- (b) $(A \cap B) \cap C = A \cap (B \cap C)$ (associative law).
- (c) $(A \cup B) \cup C = A \cup (B \cup C)$ (associative law).
- (d) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (distributive law).
- (e) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (distributive law).

✦ The **power set** of a set A is the set of all subsets of A :

$$\mathcal{P}(A) = \{X : X \subseteq A\}$$

✦ **Example 1.3:** If $A = \{q_0, q_1, q_2\}$, then

$$\mathcal{P}(A) = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, A\}$$

✦ The **difference** of the sets A and B is defined by

$$A - B = \{x : x \in A \wedge x \notin B\}$$

The set $A - B$ contains all elements x that belong to the set A but do not belong to the set B .



➤ **Example 1.4:** If $A = \{2, 4, 6, 8, 10\}$ and $B = \{1, 4, 7, 10, 13\}$, then

$$A - B = \{2, 6, 8\}$$

➤ It is customary to use **Venn diagrams** to illustrate pictorially operations and relationships between sets. The following are several simple examples of Venn diagrams for the three basic operations we have defined:

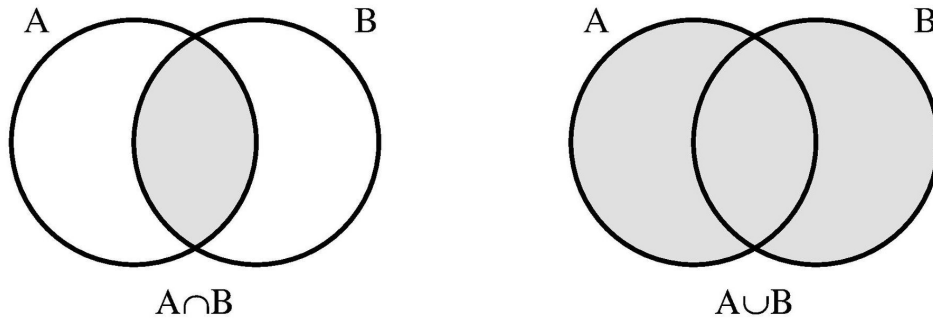


Fig. 1.1: The union of two sets $A \cup B$ and the intersection of two sets $A \cap B$

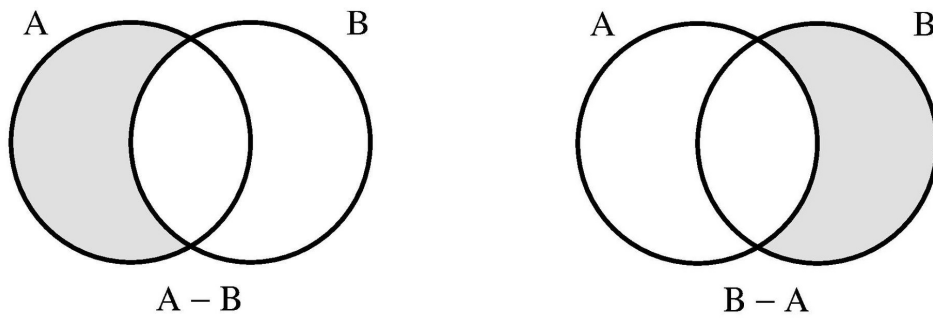


Fig. 1.2: The difference of two sets

- Let U be any set and let A be a subset of U . For the set A .
- In some contexts, a **universal set** U is given, and the discussion concerns its subsets. For every subset $A \subseteq U$, we define the **complement** of A relative to the set U by

$$\overline{A} = \{x \in U : x \notin A\}$$

➤ **De Morgan's Laws**

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

- The **cardinality** of a set A is denoted by $|A|$ and is the number of elements in the set.
- If A is a finite set, then $|\mathcal{P}(A)| = 2^{|A|}$.

1.3 Sequences

- Unlike a set, in a sequence the order in which the elements appear is of primary importance. Moreover, repetitions are allowed in sequences: a particular element may appear several times in a sequence. $(7, 3, 7, 0, 0, 7, 5, 3, 2, 0)$.
- Order matters: $(1, 4, 7) \neq (7, 1, 4)$.
- A finite sequence containing n elements is called an **n -tuple**.
- A sequence containing two elements is called an **ordered pair**, or simply a **pair**.
- The **Cartesian product** of two sets A , B is defined as the set of all pairs (a, b) where $a \in A$, $b \in B$.

$$A \times B = \{ (a, b) : a \in A \wedge b \in B \}$$

- Similarly, we define the **power** of a set A by a natural number n : $A^2, A^3, \dots, A^n$.

$$A^2 = A \times A = \{ (a, b) : a, b \in A \}$$

$$A^3 = A \times A \times A = \{ (a, b, c) : a, b, c \in A \}$$

$$A^n = \{ (a_1, a_2, \dots, a_n) : a_1, a_2, \dots, a_n \in A \}$$

- If A , B are finite sets, then $|A \times B| = |A||B|$, $|A^n| = |A|^n$.

1.4 Functions (Mappings)

- A function, in its complete form, is specified by three components:

$$f : A \rightarrow B$$

f **Function name**

A **Domain of the function** – The set of all arguments on which the function can operate.

B **Codomain of the function** – A set containing every element that may be obtained as an output of the function (but it may contain additional elements).

- The domain of the function is the set of all input values of f , and the codomain of the function is the set of values that may be obtained as outputs of f .

Remark: Not every element of B must occur as an output of f !

- In many contexts in computability theory, we encounter functions $f : \mathbb{N} \rightarrow \mathbb{N}$ from the set of natural numbers to itself. In these cases, we will be interested in the **growth rate** of $f(n)$ for



large natural numbers $n \in \mathbb{N}$.

- Let f, g be two functions on the natural numbers (their codomain may be $\mathbb{Z}$ or $\mathbb{R}$). We say that f is **of order at most g** if there exists a real constant c such that $|f(n)| \leq c|g(n)|$ for all sufficiently large n . We denote this by the expression $f(n) = O(g(n))$.
- We say that f and g are **of the same order** if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.
- **Example 1.5:** The function $f(n) = 5n^2 - 48n + 1000$ is of the same order as the function $g(n) = n^2$.
Explanation: Clearly, $g(n) \leq f(n)$ for every $n \geq 48$. Therefore, $g(n) = O(f(n))$. It is easy to see that $f(n) \leq 5g(n)$ for every $n \geq 22$, and therefore $f(n) = O(g(n))$.

1.5 Graphs

- A simple graph (or undirected graph) $G = (V, E)$:
 V – the set of **vertices** (or **nodes**)
 E – the set of **edges**, $E \subseteq \{ \{a, b\} : a, b \in V \}$

- **Example 1.6:**

$$\begin{aligned} V &= \{1, 2, 3, 4, 5\} \\ E &= \{ \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\} \} \end{aligned}$$

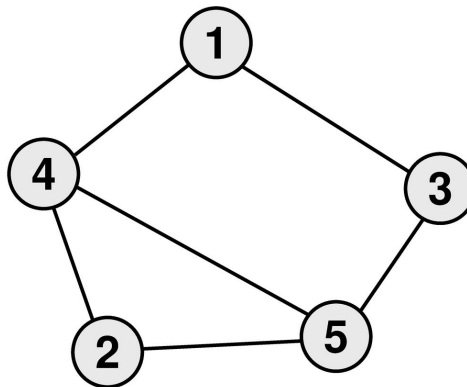


Fig. 1.3: An undirected graph with vertex set $V = \{1, 2, 3, 4, 5\}$ and edge set $E = \{ \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\} \}$

- We say that an edge $e = \{a, b\}$ **connects** the vertex a to the vertex b , and that the vertex a is **connected** to the vertex b by e .
- The **degree** of a vertex is the number of edges incident with it.
- A **path (path)** is a sequence of distinct vertices $(v_1, v_2, \dots, v_n)$ such that every two consecutive vertices form an edge in the graph. That is, for every $i < n$, $\{v_i, v_{i+1}\} \in E$.
- A **walk (walk)** is a sequence of vertices $(v_1, v_2, \dots, v_n)$ such that, for every $i < n$, $\{v_i, v_{i+1}\} \in E$. The vertices need not be distinct.

- A **cycle** is a path $(v_1, v_2, \dots, v_n)$ in which the last vertex is connected to the first vertex, $\{v_n, v_1\} \in E$, and every edge in the path (including $\{v_n, v_1\}$) appears exactly once¹.
- A **connected graph** is a graph in which, between every two distinct vertices, there is a path that starts at one vertex and ends at the other.
- A **tree** (undirected) is a connected graph with no cycles.

◆ **Example 1.7:**

$$V = \{0, 1, 2, 3, 4, 5, 7, 8, 9\}$$

$$E = \{\{4, 5\}, \{5, 1\}, \{5, 3\}, \{5, 8\}, \{4, 2\}, \{2, 7\}, \{7, 0\}, \{7, 9\}, \{0, 6\}\}$$

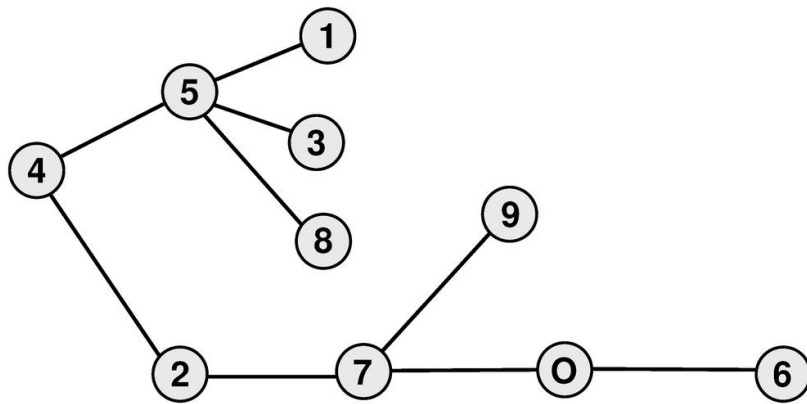


Fig. 1.4: An undirected tree with vertex set $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and edge set $E = \{\{4, 5\}, \{5, 1\}, \{5, 3\}, \{5, 8\}, \{4, 2\}, \{2, 7\}, \{7, 0\}, \{7, 9\}, \{0, 6\}\}$

1.6 Directed Graphs

- A directed graph $G = (V, E)$:
 - V – the set of nodes (or vertices)
 - E – the set of directed edges, $E \subseteq V \times V$

◆ **Example 1.8:**

$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 3), (1, 4), (4, 2), (2, 4), (5, 2), (3, 3), (3, 5), (5, 4)\}$$

- The following concepts are defined similarly:
directed path, directed walk, directed cycle, and directed tree.
- A directed graph is called **connected** if, between every two distinct vertices, there is a directed path connecting them.

¹ In an undirected graph, this requires the path to contain at least three distinct vertices, since $\{v_2, v_1\} = \{v_1, v_2\}$ is the same edge.

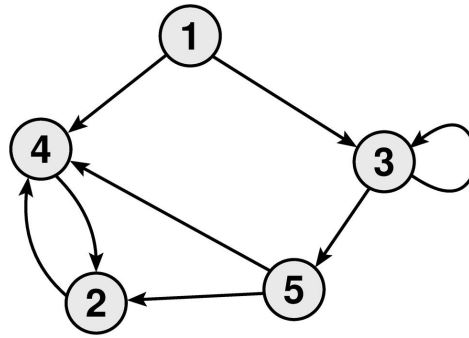


Fig. 1.5: A directed graph with vertex set $V = \{1, 2, 3, 4, 5\}$ and edge set $E = \{(1, 3), (1, 4), (4, 2), (2, 4), (5, 2), (3, 3), (3, 5), (5, 4)\}$

✦ **Exercise 1.1:** Examine the directed graph G in Figure 1.5 and answer the following questions:

- Is there more than one directed path from vertex 1 to vertex 4?
- Is there a directed cycle in the graph G ?
- Is the directed graph G connected?
- Is the directed graph G also a directed tree?

1.7 Labeled Graphs

- ✦ A labeled graph $G = (V, E, \Sigma)$ consists of the following:
 - V – the set of nodes (or vertices)
 - Σ – the set of symbols used to label the edges of the graph
 - E – the set of labeled edges, $E \subseteq (V \times \Sigma \times V)$
- ✦ Every edge in the graph is represented by an ordered triple (v_1, a, v_2) , where v_1, v_2 are two vertices from V , and a is a symbol labeling the edge.
- ✦ Unlike an ordinary directed graph, a labeled graph may have several distinct edges between two vertices! The edges are distinguished by their symbols.
- ✦ Moreover, a labeled graph may have several distinct edges from a vertex to itself! They are distinguished by the different symbols on the edges. In the following example, we see a labeled graph in which the vertex q_1 has two self-loops: one labeled 0 and the other labeled 1 .

❖ **Example 1.9:**

$$V = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{0, 1, 2\}$$

$$E = \{(q_0, 1, q_1), (q_1, 0, q_1), (q_1, 1, q_1), (q_1, 2, q_2), \\ (q_1, 2, q_3), (q_2, 0, q_1), (q_2, 1, q_3), (q_3, 1, q_3), \\ (q_3, 0, q_2), (q_3, 1, q_4)\}$$

Graphical representation:

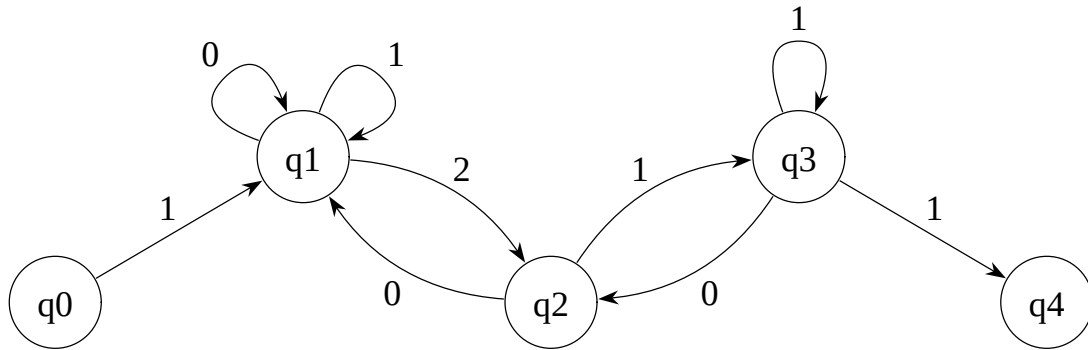


Fig. 1.6: Labeled graph

❖ To describe walks in a labeled graph, the symbols labeling the edges of the walk must be included! Otherwise, ambiguity may arise.

❖ Examples of walks in the graph above:

a. $q_0 \xrightarrow{1} q_1 \xrightarrow{2} q_2 \xrightarrow{1} q_3$

b. $q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{2} q_2$

c. $q_1 \xrightarrow{2} q_2 \xrightarrow{0} q_1 \xrightarrow{2} q_2 \xrightarrow{0} q_1$

1.8 Proof Techniques

❖ During the course, we will prove various theorems using the following methods of proof:

a. **Proof by induction**

b. **Proof by construction**

c. **Proof by contradiction**

❖ Many proofs in computability theory are based on the method of induction. We present a preliminary example of each proof method.

❖ A string of symbols is called an **even string** if each symbol of the alphabet appears in it exactly

twice. For example, if our alphabet is $\Sigma = \{a, b, c\}$, then the following strings are even strings:

$$abbacc, \quad cabcba, \quad cbcaba$$

Question: How many even strings are there over an alphabet with n symbols?

Theorem: For every alphabet with n symbols, the number of even strings is

$$\frac{(2n)!}{2^n}$$

Proof by induction:

Let $F(n)$ denote the number of even strings over an alphabet with n symbols.

Basis of induction: ($n = 1$)

Clearly, over an alphabet containing a single symbol, only one even string can be formed. That is, $F(1) = 1$. This agrees with the formula: $1 = \frac{(2 \cdot 1)!}{2^1}$.

Induction hypothesis: $F(n) = \frac{(2n)!}{2^n}$

To prove: $F(n+1) = \frac{(2n+2)!}{2^{n+1}}$

Note that every even string over an alphabet of n symbols must have length $2n$. Over an alphabet of $n+1$ symbols, every even string has length $2n+2$, and it is clear that it is obtained from a even string over an alphabet of n symbols by inserting two copies of a new symbol into the slots shown in Figure 1.7.

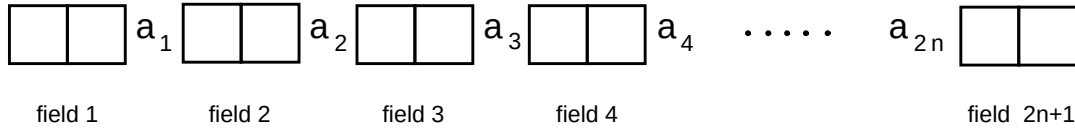


Fig. 1.7: Proof by induction

To obtain a even string over an alphabet of $n+1$ symbols from a even string $a_1 a_2 \cdots a_{2n}$ over an alphabet of n symbols, the new symbol, which we denote by x , must be inserted into two empty slots. There are two principal ways to do this:

Method A:

Insert x twice into the same slot, as shown in Figure 1.8. Clearly, there are exactly $2n+1$ ways to do this.

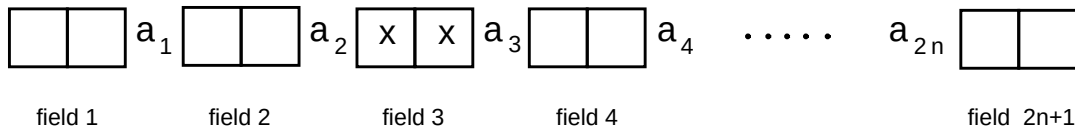


Fig. 1.8: Proof by induction

Method B:

Insert x into two different slots, as shown in Figure 1.9. Clearly, there are exactly $\binom{2n+1}{2}$ ways to do this.

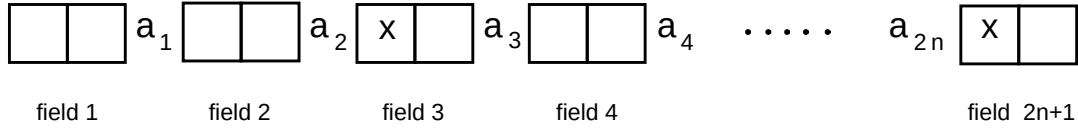


Fig. 1.9: Proof by induction

Recall that

$$\binom{2n+1}{2} = \frac{(2n+1)!}{2 \cdot (2n-1)!} = \frac{(2n+1)2n}{2} = (2n+1)n$$

In summary, from each even string over an alphabet of n symbols, we can create

$$(2n+1)n + (2n+1)$$

even strings over an alphabet of $n+1$ symbols. The sum is $(2n+1)(n+1)$. Therefore

$$\begin{aligned} F(n+1) &= (2n+1)(n+1)F(n) = (2n+1)(n+1)\frac{(2n)!}{2^n} \\ &= \frac{(2n+2)(2n+1) \cdot (2n)!}{2 \cdot 2^n} \\ &= \frac{(2n+2)!}{2^{n+1}} \end{aligned}$$

- **Exercise 1.2:** How many triple strings are there over an alphabet with n symbols?
- The following theorem illustrates a proof by construction:

Theorem: For every even n , there exists a simple graph in which every vertex has degree 3

Proof by construction:

We prove the theorem by constructing the required graph.

$$\begin{aligned} V &= \{v_1, v_2, \dots, v_n\} \\ E &= \left\{ \{v_i, v_j\} : j = i + 1 \vee (j = n \wedge i = 1) \vee j = i + \frac{n}{2} \right\} \end{aligned}$$

For example, for $n = 8$, the graph appears as shown in Figure [1.10](#).

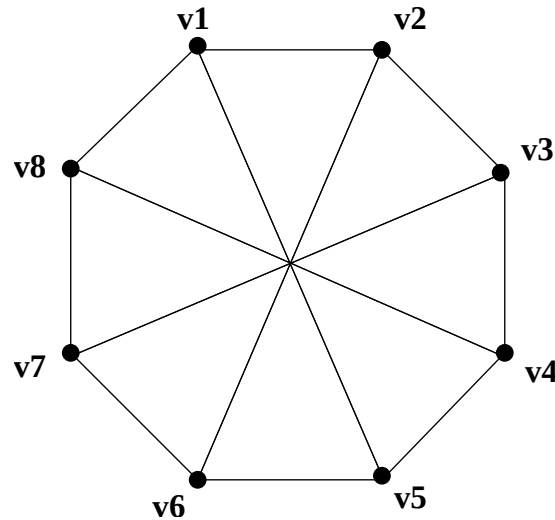


Fig. 1.10: Proof by construction

1.9 Strings and Languages

- An **alphabet** is a finite set Σ of characters (symbols).
- A **string** is a finite sequence of symbols. For example, if $\Sigma = \{a, b, c\}$, then

$$w = aacbcca$$

is a **string** (or **word**) over the alphabet Σ .

- **Length of a string:** The number of symbols in a string is denoted by $|w|$.
In the example above, $|w| = 7$.
- **The empty string:** The empty string is the string containing zero symbols (that is, it contains no symbol at all). It is of theoretical importance and is denoted by the Greek letter ϵ . The length of the empty string is 0 .
- **Reversal:** The reversal of the string w is $accbcaa$, and is usually denoted by w^R .
- **Substring:** If $|w| = n$, then for every $1 \leq i \leq n$, we define $w[i]$ to be the symbol in position i . For every $1 \leq i \leq j \leq n$, we define the **substring**

$$w[i : j] = w[i]w[i + 1] \cdots w[j]$$

Example: The string **abb** is a substring of $w = caabbbbc$. It is easy to see that $abb = w[3 : 5]$.

- **Prefix:** When $i = 1$, $w[1 : j]$ is called a **prefix** of w . Example: The string **aab** is a **prefix** of **aabbbbc**.
- **Suffix:** When $j = n$, $w[i, n]$ is called a **suffix** of w . Example: The string **bbc** is a **suffix** of **caabbbbc**.
- Prefixes and suffixes are two special cases of substrings!

- **Concatenation or joining of strings:** The concatenation of the strings aab and bc is $aabbc$.
- The string $aaaaa$ is denoted briefly by a^5 . The string a^3bc^4 is therefore $aaabcccc$.
- A **language** is a set L of strings over an alphabet Σ .

➤ **Example 1.10:**

a.

$$\begin{aligned}\Sigma &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ L &= \{1, 151, 15151, 1515151, \dots\}\end{aligned}$$

b.

$$\begin{aligned}\Sigma &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ L &= \{3, 31, 314, 3141, 31415, 314159, \dots\}\end{aligned}$$

c.

$$\begin{aligned}\Sigma &= \{0, 1, +, =\} \\ L &= \{1 + 0 = 1, 1 + 1 = 0, 1 + 1 + 1 = 1, 0 + 1 = 1\}\end{aligned}$$

d.

$$\begin{aligned}\Sigma &= \{A, C, G, T\} = \text{sdica onima 4 eht} \\ L &= \{w : \text{ecneuqes DNA a } w\}\end{aligned}$$

- Symbols from the alphabet are usually denoted by the letters $a, b, c, \dots$.
Strings are denoted by the letters $u, v, w, x, y, z, \dots$.
 Σ^* – the set of all strings over the alphabet Σ .
 Σ^+ – the set of all strings over the alphabet Σ except for the empty string.

➤ **Operations on Languages:**

a. **Union**

$$L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}$$

b. **Intersection**

$$L_1 \cap L_2 = \{w : w \in L_1 \text{ and } w \in L_2\}$$

c. **Concatenation**

$$L_1 L_2 = \{uv : u \in L_1 \text{ and } v \in L_2\}$$

d. **Power**

$$\begin{aligned}L^0 &= \{\varepsilon\} \\ L^1 &= L \\ L^2 &= LL \\ L^{n+1} &= L^n L\end{aligned}$$

If Σ is our alphabet, what are the languages $\Sigma^1, \Sigma^2, \dots$?



e. Closure under Concatenation

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

$$L^+ = \bigcup_{n=1}^{\infty} L^n$$

The only difference between L^* and L^+ is that $\varepsilon \in L^*$, whereas $\varepsilon \in L^+$ only if $\varepsilon \in L$. In other words, if $\varepsilon \notin L$, then $L^+ = L^* - \{\varepsilon\}$.

♦ **Example 1.11:** If $L_1 = \{10, 101\}$, $L_2 = \{1010, 010\}$, then

$$L_1 \cup L_2 = \{10, 101, 1010, 010\}$$

$$L_1 \cap L_2 = \emptyset$$

$$L_1 L_2 = \{101010, 10010, 1011010\}$$

$$L_1^2 = \{1010, 10101, 10110, 101101\}$$

If $L = \{00, 11\}$, then

$$L^* = \{\varepsilon, 00, 11, 0000, 0011, 1100, 1100, 000000, \dots\}$$

If the language L contains at least one string, then the languages L^* and L^+ are infinite.

Exercises for chapter 1

Remark: The following are several basic exercises in set theory and graph theory. They are intended primarily for students who have forgotten the material, and also as warm-up exercises for the remainder of the course. The fundamentals of set theory and graph theory (or any course in discrete mathematics) are prerequisites for the present course, and therefore it is recommended that you try to solve at least some of the following exercises before continuing.

1. Draw the Venn diagrams of each of the following sets:
 - (a) $(A - B) \cup (B - A)$
 - (b) $A \cap (B \cup C)$
 - (c) $(A \cap B) \cup (A \cap C)$
 - (d) $A - (B \cup C)$
 - (e) $(A - B) \cap (A - C)$
 - (f) $[(A \cap B) - C] \cup [(A \cap C) - B] \cup [(B \cap C) - A]$
2. Prove the following assertions about sets using Venn diagrams.
 - (a) If $A \subseteq B$, then $A \cap B = A$.
 - (b) If $A \subseteq B$, then $A \cup B = B$.
 - (c) $A - (B \cup C) = (A - B) \cap (A - C)$.
 - (d) $A - (B \cap C) = (A - B) \cup (A - C)$.
 - (e) If $A \cup B = B$, then $A \subseteq B$.



- (f) If $A \cap B = A$, then $A \subseteq B$.
 (g) If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.

3. Prove or disprove by means of a counterexample:

- (a) If $A \cup B = A \cup C$, then $B = C$
 (b) If $A \cap B = A \cap C$, then $B = C$
 (c) If $A - B = B - A$, then $A = B$
 (d) If $A \subseteq B \subseteq C$, then $A \cup B = B \cap C$



4. Define the following two sets of real numbers:

$$A = \left\{ x : \frac{4}{1+x} > 1 \right\}, \quad B = \left\{ x : x^2 - 3x \leq 10 \right\}$$

Determine which of the following assertions are true and which are false:

- (a) $A \subseteq B$ (b) $B \subseteq A$ (c) $A \cap B \neq \emptyset$
 (d) $A \cap B = (3, 5]$ (e) $A \cup B = [-2, 5]$ (f) $A \cup B = [-2, \infty)$
 (g) $A - B = (5, \infty)$ (h) $B - A = [-2, 3]$

5. Let $f: A \rightarrow B$, $g: B \rightarrow C$, $h: C \rightarrow D$, be three given functions. Prove or disprove each of the following assertions:

- (a) $f \circ g = g \circ f$.
 (b) If f , g are one-to-one functions, then $g \circ f$ is also one-to-one.
 (c) $(h \circ g) \circ f = h \circ (g \circ f)$.



6. Let $G = (V, E)$ be a graph whose vertex set is $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and whose edge set is defined by: $\{u, v\} \in E$ if and only if $|u - v| \in \{2, 5, 7\}$.

- (a) Is G a connected graph?
 (b) Is G a tree? If your answer is negative, find at least one cycle in the graph G .
 (c) What is the subgraph of G induced by the vertex set $\{0, 2, 4, 6, 8\}$?
 (d) What is the subgraph of G induced by the vertex set $\{1, 3, 5, 7\}$?
 (e) Is there a walk connecting 0 and 9? If so, how many such walks are there?
 (f) Does the graph G have a Hamiltonian path? If so, is there more than one?
 (g) Does the graph G have an Eulerian trail?

Hint: Before answering the questions, it is advisable to draw a diagram of the graph on paper (or with suitable drawing software). If $G = (V, E)$ is a given graph, and if $U \subseteq V$, the **induced subgraph** on U is the graph whose vertex set is U and whose edges are all pairs $\{u, v\} \in E \cap (U \times U)$. A **Hamiltonian path** is a path that passes through every vertex of the graph. An **Eulerian trail** is a walk that traverses every edge of the graph exactly once (but it may visit the same vertex more than once).



7. Let $G = (V, E)$ be the directed graph in Figure 1.11.

- a. Write the vertex set and edge set of G formally.
 b. Draw the subgraph of G induced by the vertex set $\{v_1, v_3, v_5, v_7\}$. Is this subgraph



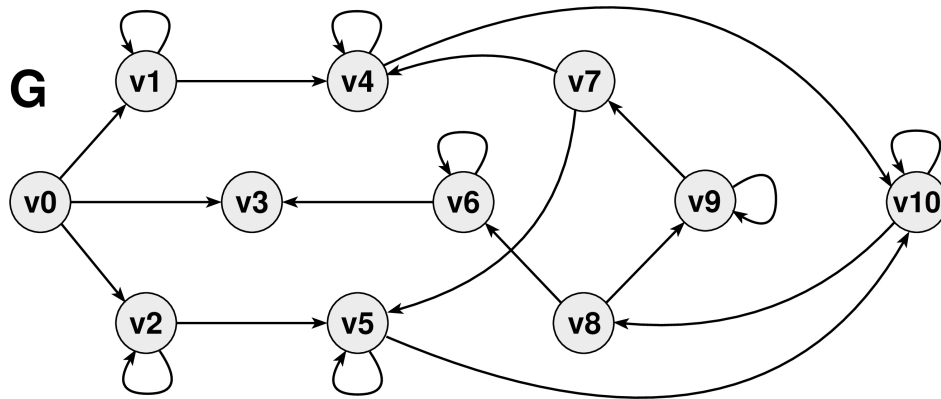


Fig. 1.11: A diagram of the graph $G = (V, E)$ for Exercise 7

connected?

- c. Draw the subgraph of G induced by the vertex set $\{v_0, v_2, v_4, v_6, v_8\}$. Is this subgraph connected?
- d. Is there a path connecting v_6 to v_7 ? If so, how many such paths are there?
- e. Is there a path connecting v_7 to v_6 ? If so, how many such paths are there?
- f. Does the graph G have a Hamiltonian path? If so, is there more than one?
- g. Does the graph G have an Eulerian trail?
- h. Is G a tree? If your answer is negative, find at least one cycle in the graph G .
- i. Is G a connected graph? If so, find a **maximal spanning tree** for G . If not, give an example of two vertices with no path between them.

Hint: The list of edges must also include loops (an edge from a vertex to itself). A Hamiltonian path and an Eulerian trail were defined in the guidance for Exercise 6, but in this case it must be remembered that the graph is directed! A **maximal spanning tree** of G is a subgraph $T = (V, E')$ of G that is a tree whose vertex set is V , and adding any other edge of G to it creates a cycle (so it ceases to be a tree). A theorem in graph theory guarantees the existence of a maximal spanning tree in a connected graph.

8. Prove that the sum of the degrees of the vertices in a finite simple graph is always even.
Hint: Try induction on the size of the graph.
9. Prove that every simple graph with at least two vertices contains two distinct vertices of the same degree.
10. Prove that if there is a walk between two distinct vertices of a graph (of any type: simple, directed, or labeled), then there is also a path between those two vertices.
11. A graph G is called **regular** of degree k if every vertex has degree k .
 - a. Draw an example of a regular graph of degree 3 with six vertices.

- b.** Prove that for every $k \geq 2$, there exists a regular graph of degree k .
- c.** Prove that if G is a regular graph of odd degree, then the number of its vertices must be even.



Chapter 2

Finite Automata

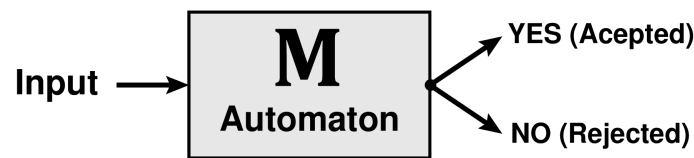


Fig. 2.1: The minimal architecture of every finite automaton

2.1 Basic Concepts

i **Definition 2.1:** A **finite automaton** is a theoretical computing machine characterized by the following five components:

Q A set of **states** in which the machine may be.

Σ The alphabet over which the strings fed to the automaton are defined.

δ **Transition function** $\delta : Q \times \Sigma \rightarrow Q$

F A set of **accepting states** (or final states), $F \subseteq Q$.

q_0 The **start state** of the automaton. This is the state in which the automaton is found at the beginning of every computation.

Abbreviated notation:

$$M = (Q, \Sigma, \delta, F, q_0)$$

- Whenever discussing an automaton, it is necessary to state clearly (unless this is clear from the context) all five components above.
- We may model a finite automaton as a computing machine with a tape of unlimited length and a read head that moves over the tape cells in one direction (from left to right). When the machine is in state q and its head reads a particular symbol a , the head moves to the next cell (on the right), and the machine moves to a new state q' according to the transition function $q' = \delta(q, a)$ (see Figure 2.2).

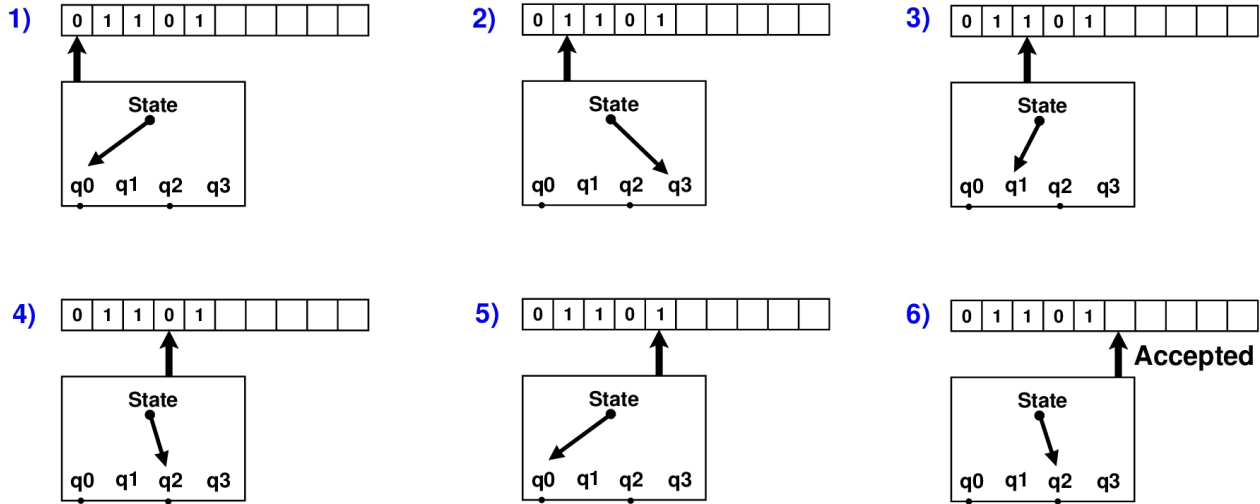


Fig. 2.2: A finite automaton is a type of state machine

- For example, Figure 2.2 shows six stages of a computation in a state machine. The input fed to the machine is **01101**. At the fourth stage of the computation, for example, the machine head reads the fourth input symbol, **0**. At this stage, the machine is in state **q_2** . The machine's transition function specifies: if the machine is in state **q_2** and the current symbol is **0**, it moves to state **q_0** , and its head advances to the next symbol.
- A string fed to an automaton is **accepted** by the automaton if, after its head has scanned all the symbols of the string, the automaton is in one of its accepting states. Figure 2.2 shows that the machine has finished scanning all the symbols of the input and is in state **q_2** , which is one of its accepting states. We therefore say that the machine **accepts** the input **01101**.
- Mathematically, the following definition suffices to describe the automaton **M** above:

$$\begin{aligned}
 Q &= \{q_0, q_1, q_2, q_3\} \\
 \Sigma &= \{0, 1\} \\
 q_0 &= \text{Start} \\
 F &= \{q_0, q_2\}
 \end{aligned}$$

δ	0	1
q_0	q_3	q_2
q_1	q_1	q_2
q_2	q_0	q_3
q_3	q_1	q_1

Note that the machine has two accepting states: **q_0** , **q_2** .

i **Definition 2.2:** An automaton **M** **accepts** a string **w** if, after scanning all the symbols of **w** , the automaton reaches one of its accepting states. We also say that the string **w** is accepted by the automaton **M** .

- For example, the automaton **M** above accepts the string **101**. The computation sequence is:

$$q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_0 \xrightarrow{1} q_2 \in F$$

The word **100**, on the other hand, is not accepted by M

$$q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_0 \xrightarrow{0} q_3 \notin F$$

The word **101010100** is also not accepted by M (verify this using the diagram below).

i **Definition 2.3:** *The language accepted by an automaton M is the set of all strings accepted by M . This language is denoted by $L(M)$.*

➤ The complexity of the definition of a finite automaton requires us to understand it through additional mathematical models. The closest model is a **labeled graph**.

- ◆ Vertex set: Q
- ◆ Edge set: $E = \{ (q, a, q') : \delta(q, a) = q' \}$
- ◆ The set of symbols used to label the graph's edges is, of course, Σ .
- ◆ Because δ is a function, our graph has the following special property: **From every vertex q , and for every symbol a , there is exactly one outgoing edge labeled a .**

2.2 Transition Diagrams

➤ It is easier to describe and understand automata by means of transition diagrams. Our automaton M is depicted in Figure 2.3.

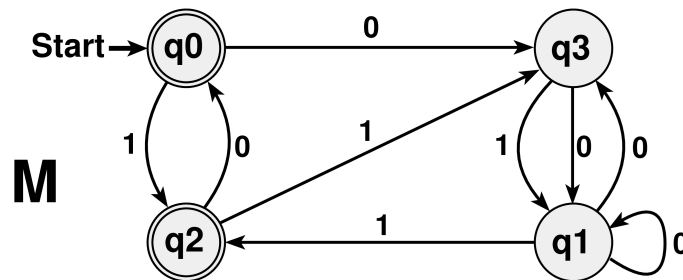


Fig. 2.3: Transition diagram of the automaton M

Each state q of the automaton M corresponds to one vertex of the graph, represented by a circle with the name of the state written at its center. If q is an **accepting state** of the automaton, we indicate this fact by a double circle.

i **Definition 2.4:** *The extended transition function δ^**

$$\delta^* : Q \times \Sigma^* \rightarrow Q$$

is defined as follows (a recursive definition!)

- a. $\delta^*(q, \varepsilon) = q$
- b. $\delta^*(q, a) = \delta(q, a), \quad a \in \Sigma$

$$c. \delta^*(q, wa) = \delta^*(\delta^*(q, w), a), \quad w \in \Sigma^*$$

❖ **Example 2.1:** We compute several values of δ^* for the automaton M above

$$\delta^*(q_0, 0) = q_1$$

$$\delta^*(q_0, 01) = \delta^*(\delta^*(q_0, 0), 1) = \delta^*(q_1, 1) = q_0$$

$$\delta^*(q_0, 010) = \delta^*(\delta^*(q_0, 01), 0) = \delta^*(q_0, 0) = \delta^*(q_0, 0) = q_1$$

➤ The function δ^* is very important for defining additional concepts and analyzing automata.

➤ For example, we can define the language accepted by an automaton M more precisely as

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

➤ We can also use the function δ^* to prove assertions.

❖ **Example 2.2:** Let M be the following automaton

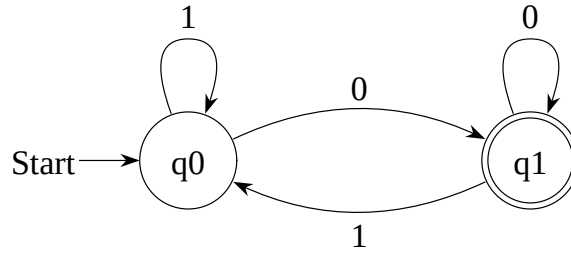


Fig. 2.4: Transition diagram of a finite automaton M

Then

$$L(M) = \{ w \in \{0, 1\}^* : w \text{ ends with the symbol } 0 \}$$

✔ **Proof:** The proof has two directions.

First direction: Let w be a binary string ending with the symbol 0 . Then $w = u0$, where u is some binary string.

$$\delta^*(q_0, w) = \delta^*(q_0, u0) = \delta^*(\delta^*(q_0, u), 0) = \begin{cases} \delta^*(q_0, 0) = q_1 \\ \delta^*(q_1, 0) = q_1 \end{cases}$$

The value of $\delta^*(q_0, u)$ may be q_0 or q_1 , and in either case the automaton moves to the accepting state q_1 .

Second direction: Assume that $w \in L(M)$. Write $w = ua$, where $u \in \{0, 1\}^*$, $a = 0, 1$. We prove that necessarily $a = 0$. Suppose, by way of contradiction, that $a = 1$. Then

$$\delta^*(q_0, w) = \delta^*(q_0, u1) = \delta^*(\delta^*(q_0, u), 1) = \begin{cases} \delta^*(q_0, 1) = q_0 \\ \delta^*(q_1, 1) = q_0 \end{cases}$$



Definition 2.5: Two automata M_1 , M_2 are called **equivalent** if

$$L(M_1) = L(M_2)$$

Notation: $M_1 \equiv M_2$.

Example 2.3: Let M_1 , M_2 be the two automata depicted in Figure 2.5.

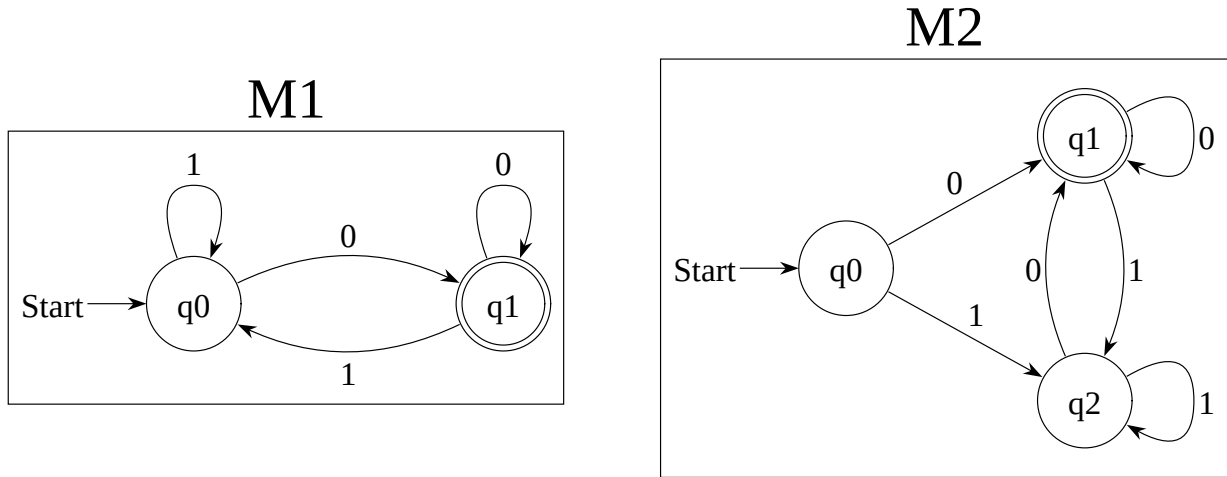


Fig. 2.5: Transition diagrams for the automata M_1 , M_2

It is not difficult to prove that the automaton M_2 accepts the language of all binary strings ending in zero, and therefore $M_1 \equiv M_2$.

2.3 Regular Languages

Definition 2.6: A language L is called a **regular language** if there is an automaton M such that $L = L(M)$.

At this stage, we present a list of questions that will occupy us later:

- Is there a method (an algorithm) for determining whether a given language L is regular? If so, is there an algorithm for finding an automaton M that accepts the language L ?
- Is there an algorithm by which one can determine whether $L(M_1) = L(M_2)$ for any two automata M_1 , M_2 ?
- Given a complex automaton M , is there a simpler automaton equivalent to it? And is there a practical algorithm for finding a smallest equivalent automaton?



2.4 Nondeterministic Automata

- An ordinary finite automaton is sometimes also called a **deterministic automaton**, because from every state q , for every symbol a , there is exactly one way to move to the next state. This restriction does not exist in a nondeterministic automaton. That is, from a given state q on a symbol a , there may be zero or more states to which the automaton can move, as shown in the transition diagram in Figure 2.6.

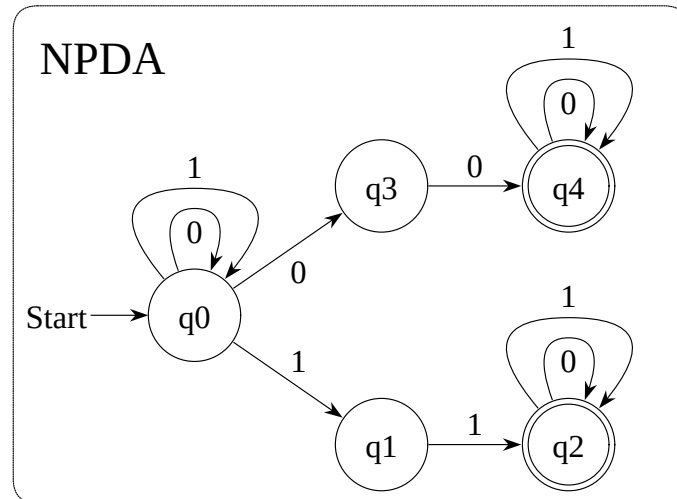


Fig. 2.6: A nondeterministic automaton

Definition 2.7: A **nondeterministic finite automaton** is a theoretical computing machine characterized by the following components:

- Q A set of **states** in which the machine may be.
- Σ The alphabet over which the strings fed to the automaton are defined.
- δ **Transition function** $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$
- F A set of **accepting states** (or final states), $F \subseteq Q$.
- q_0 The **start state** of the automaton. This is the state in which the automaton is found at the beginning of every computation.

Reminder: $\mathcal{P}(Q)$ is the set of all subsets of Q .

- The definition of a nondeterministic automaton is identical to that of a deterministic automaton, except for the transition function, which maps a pair (q, a) to a set of states $\delta(q, a) \subseteq Q$. This means that if a nondeterministic automaton is in state q while reading a symbol a , it may move to any of the states in the set $\delta(q, a)$.
- Abbreviated notation:

$$M = (Q, \Sigma, \delta, F, q_0)$$

Whenever discussing a nondeterministic automaton, all five components must be stated clearly (unless they are clear from the context).

- ⊛ Unlike a deterministic automaton, for every string w there are usually a large number of distinct computation sequences, and in some cases there is not even a single computation sequence.
- ⊛ For example, in the automaton above there are four distinct computation sequences for the string $w = 11001$:

$$\begin{aligned}
 q_0 &\xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \\
 q_0 &\xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_3 \xrightarrow{0} q_4 \xrightarrow{1} q_4 \\
 q_0 &\xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \\
 q_0 &\xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \xrightarrow{0} q_2 \xrightarrow{1} q_2
 \end{aligned}$$

i Definition 2.8: A string w is said to be **accepted** by a nondeterministic automaton M if at least one of the computation sequences for w ends in an accepting state.

- ⊛ The string **11001** is accepted by M .
- ⊛ Mathematically, the following definition suffices to describe the automaton M above:

$$\begin{aligned}
 Q &= \{q_0, q_1, q_2, q_3, q_4\} \\
 \Sigma &= \{0, 1\} \\
 q_0 &= \text{Start} \\
 F &= \{q_2, q_4\}
 \end{aligned}$$

δ	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

i Definition 2.9: The language accepted by a nondeterministic automaton M is the set of all strings accepted by M . This language is denoted by $L(M)$.

- ⊛ A common problem: find as simple a verbal description of $L(M)$ as possible. For example, exactly what language is accepted by the automaton M in the example above? (Figure 2.6)
- ⊛ As in the deterministic case, it is very convenient to represent an NFA by a labeled graph whose
 - ◆ vertex set is: Q ,
 - ◆ edge set is: $E = \{(q, a, q') : q' \in \delta(q, a)\}$,
 - ◆ and set of symbols used to label the graph's edges is, of course, Σ .
- ⊛ In a deterministic automaton, for every state q and every symbol a , there is a unique state q' such that $(q, a, q') \in E$.
- ⊛ In a nondeterministic automaton, there may be more than one such state q' , or even no such state q' at all.



i **Definition 2.10:** *The extended transition function δ^**

$$\delta^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

is defined as follows

- a. $\delta^*(q, \varepsilon) = \{q\}$
- b. $\delta^*(q, a) = \delta(q, a), \quad a \in \Sigma$
- c. $\delta^*(q, wa) = \bigcup_{r \in \delta^*(q, w)} \delta(r, a), \quad w \in \Sigma^*$

or, equivalently:

$$\delta^*(q, wa) = \{p \in Q : \exists r \in \delta^*(q, w) [p \in \delta(r, a)]\}$$

Again, this is a recursive definition!

- Condition a' forbids a change of state without input!
- In simple terms: $\delta^*(q, w)$ is the set of all states at which computation sequences for the string w end, provided that the computation begins in state q . Normally, every computation sequence begins in the start state q_0 , and therefore we will usually encounter the expression $\delta^*(q_0, w)$.
- In our example, the computation sequences for the string **11001** end in the states $\{q_0, q_1, q_2, q_4\}$, and therefore

$$\delta^*(q_0, 11001) = \{q_0, q_1, q_2, q_4\}$$

But this is not the shortest way to compute $\delta^*(q_0, 11001)$!

- Another useful extension of the transition function δ is:

$$\delta : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q), \quad \delta(P, a) = \bigcup_{q \in P} \delta(q, a)$$

There is no risk of ambiguity in using the symbol δ for this extension¹. For every set of states $P \subseteq Q$, and every symbol a , $\delta(P, a)$ is the set of all states q belonging to $\delta(p, a)$ for $p \in P$.

- This extension allows us to give an elegant formula for δ^* :

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a) \tag{2.1}$$

¹ Anyone familiar with operator overloading in object-oriented programming understands exactly why.

➤ **Example 2.4:** We compute $\delta^*(q_0, 11001)$ for the automaton M above

$$\begin{aligned}
 \delta^*(q_0, 1) &= \{q_0, q_1\} \\
 \delta^*(q_0, 11) &= \delta(\delta(q_0, 1), 1) = \delta(\{q_0, q_1\}, 1) \\
 &= \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_1\} \cup \{q_2\} \\
 &= \{q_0, q_1, q_2\} \\
 \delta^*(q_0, 110) &= \delta(\delta(q_0, 11), 0) = \delta(\{q_0, q_1, q_2\}, 0) \\
 &= \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) \\
 &= \{q_0, q_3\} \cup \emptyset \cup \{q_2\} \\
 &= \{q_0, q_2, q_3\} \\
 \delta^*(q_0, 1100) &= \delta(\delta(q_0, 110), 0) = \delta(\{q_0, q_2, q_3\}, 0) \\
 &= \delta(q_0, 0) \cup \delta(q_2, 0) \cup \delta(q_3, 0) \\
 &= \{q_0, q_3\} \cup \{q_2\} \cup \{q_4\} \\
 &= \{q_0, q_2, q_3, q_4\} \\
 \delta^*(q_0, 11001) &= \delta(\delta(q_0, 1100), 1) = \delta(\{q_0, q_2, q_3, q_4\}, 1) \\
 &= \delta(q_0, 1) \cup \delta(q_2, 1) \cup \delta(q_3, 1) \cup \delta(q_4, 1) \\
 &= \{q_0, q_1\} \cup \{q_2\} \cup \emptyset \cup \{q_4\} \\
 &= \{q_0, q_1, q_2, q_4\}
 \end{aligned}$$

- The function δ^* is very important for defining additional concepts and analyzing automata.
- For example, the language accepted by an automaton M can be defined more concisely by

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \emptyset\} \quad (2.2)$$

- That is: A string w is **accepted** by a nondeterministic automaton M if and only if the set $\delta^*(q_0, w)$ contains at least one accepting state.
- We can also use the function δ^* to prove various assertions.

◆ **Exercise 2.1:** Prove that for any two strings $u, v \in \Sigma^*$,

$$\delta^*(q, uv) = \delta^*(\delta^*(q, u), v)$$

* **Hint:** Induction on the length of the string v .

- **Notation:** We write $q \xrightarrow{w} q'$ if there is a computation sequence for the string w that begins in state q and ends in state q' . For example, in our example, $q_3 \xrightarrow{001} q_4$.

i Proposition 2.1: If $q \xrightarrow{u} q'$ and $q' \xrightarrow{v} q''$, then $q \xrightarrow{uv} q''$.

➤ Let us return to our example automaton M (Figure 2.6).

i Proposition 2.2:

$$L(M) = \left\{ w \in \{0, 1\}^* : w \text{ contains the substring } 00 \text{ or } 11 \right\}$$

✓ Proof: The proof has two directions.

First direction: Let w be a binary string containing the substring 00 (or 11). We may write $w = u00v$. Clearly, for every string u , $q_0 \xrightarrow{u} q_0$ holds. Also, $q_0 \xrightarrow{00} q_4$ and $q_4 \xrightarrow{v} q_4$. Therefore $q_0 \xrightarrow{w} q_4$, and hence $w \in L(M)$.

The same applies to $w = u11v$.

Second direction: Assume that $w \in L(M)$. Then $q_2 \in \delta^*(q_0, w)$ or $q_4 \in \delta^*(q_0, w)$. Suppose, for example, that $q_4 \in \delta^*(q_0, w)$. Then there is at least one computation sequence for w that begins in state q_0 and ends in state q_4 . This computation sequence must contain the path $q_0 \xrightarrow{0} q_3 \xrightarrow{0} q_4$. ■

i Definition 2.11: Two automata M_1, M_2 (of any kind) are called **equivalent** if

$$L(M_1) = L(M_2)$$

Notation: $M_1 \equiv M_2$.

- The definition of a nondeterministic automaton includes the ordinary automaton as a special case. An ordinary automaton is a nondeterministic automaton with the following special property: for every state q and every symbol a , the set of states $\delta(q, a)$ contains exactly one state.
- Therefore, every language definable by an ordinary automaton can also be defined by a nondeterministic automaton.
- **Question:** Are there languages that can be defined by nondeterministic automata but cannot be defined by ordinary automata?
- The answer to this question is surprising: every language that can be defined by nondeterministic automata can also be defined by an ordinary automaton!
- If so, why do we need nondeterministic automata?
 - a. Theoretical reasons: certain concepts are easier to define and understand using nondeterministic automata. Many proofs in computability theory are simpler when nondeterministic automata are used.
 - b. It is easier to prove that a given language is regular by constructing a nondeterministic automaton. In many cases, languages defined by complicated deterministic automata can be defined by far simpler nondeterministic automata.



2.5 Equivalence of Deterministic and Nondeterministic Automata

i Theorem 2.3: Let L be a language accepted by a nondeterministic finite automaton M . Then there is a deterministic automaton M' that accepts L .

✓ Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic automaton accepting the language L . We present a method for transforming any nondeterministic automaton M into a deterministic automaton $M' = (Q', \Sigma, \delta', q'_0, F')$ that accepts exactly the same language! The deterministic automaton obtained by this method is usually complicated (an excessively large number of states), but for now, its existence is important to us from a theoretical perspective. After constructing it, we can simplify it by removing (redundant states) from it.

- a. The state set Q' of M' consists of all subsets of Q . That is, $Q' = P(Q)$. This means that if the automaton M has n states, then the automaton M' has 2^n states. Usually, many of these states are redundant, and in practice the number of states of M' is much smaller. Each state of Q' is denoted by $[q_1, q_2, \dots, q_i]$ (a subset of Q). We emphasize again that $[q_1, q_2, \dots, q_i]$ is a single state of Q' ! The idea behind this choice is that the automaton M' must remember, in a single state, all the possible states in which M may be at any given moment.
- b. The symbol set Σ is the same for both automata
- c. The start state of M' is $q'_0 = [q_0]$.
- d. The set of accepting states F' of the automaton M' consists of all subsets of Q that contain at least one accepting state of M :

$$F' = \{P \subseteq Q : P \cap F \neq \emptyset\}$$

- e. The transition function δ' of M' is defined as follows:

$$\delta'([q_1, q_2, \dots, q_i], a) = [p_1, p_2, \dots, p_j]$$

if and only if

$$\delta(\{q_1, q_2, \dots, q_i\}, a) = \{p_1, p_2, \dots, p_j\}$$

In greater detail,

$$\delta'([q_1, q_2, \dots, q_i], a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \dots \cup \delta(q_i, a)$$

The last formula will be important for practical examples.

- ◆ We now turn to the proof itself.



Auxiliary claim: For every string w ,

$$\delta'(q'_0, w) = [q_1, q_2, \dots, q_i] \iff \delta(q_0, w) = \{q_1, q_2, \dots, q_i\} \quad (*)$$

Proof: By induction on the length of the string.

Base case: the claim is clear if $|w| = 0$, because then $w = \varepsilon$, and by the definition of the extension of δ ,

$$\delta(q_0, \varepsilon) = \{q_0\}$$

On the other hand, by the definition of δ' ,

$$\delta'(q'_0, \varepsilon) = \delta(\{q_0\}, \varepsilon) = \{\delta(q_0, \varepsilon)\} = \{q_0\}$$

Induction hypothesis: Assume that claim $(*)$ holds for every string w of length at most n . We prove that it holds for every string of length $n + 1$.

◆ Let u be a string of length $n + 1$. Then $u = wa$, $|w| = n$, $a \in \Sigma^*$

$$\delta'(q'_0, wa) = \delta'(\delta'(q'_0, w), a)$$

By the induction hypothesis,

$$\delta'([q_1, q_2, \dots, q_i], w) = [p_1, p_2, \dots, p_j]$$

if and only if

$$\delta(\{q_1, q_2, \dots, q_i\}, w) = \{p_1, p_2, \dots, p_j\}$$

But by the definition of δ' ,

$$\delta'([p_1, p_2, \dots, p_j], a) = [r_1, r_2, \dots, r_k]$$

if and only if

$$\delta(\{p_1, p_2, \dots, p_j\}, a) = \{r_1, r_2, \dots, r_k\}$$

It follows that

$$\delta'(q'_0, wa) = [r_1, r_2, \dots, r_k]$$

if and only if

$$\delta(q_0, wa) = \{r_1, r_2, \dots, r_k\}$$

This completes the proof of the claim.

◆ Let L' be the language accepted by the automaton M' . To complete the proof of the theorem,



we must prove that $L = L'$. For every string $w \in \Sigma^*$,

$$w \in L' \iff \delta'(q'_0, w) \in F'$$

Suppose that

$$\delta'(q'_0, w) = [q_1, q_2, \dots, q_i] \quad (**)$$

The definition of F' implies that

$$\{q_1, q_2, \dots, q_i\} \cap F \neq \emptyset$$

By the claim above, equality $(**)$ is equivalent to the assertion

$$\delta(q_0, w) = \{q_1, q_2, \dots, q_i\}$$

We have obtained that $w \in L'$ if and only if the set of states $\delta(q_0, w)$ contains an accepting state of M . That is, $w \in L'$ if and only if $w \in L$. Therefore $L = L'$. ■

➤ **Example 2.5:** We transform the nondeterministic automaton in Figure 2.7 into the deterministic automaton in Figure 2.8.

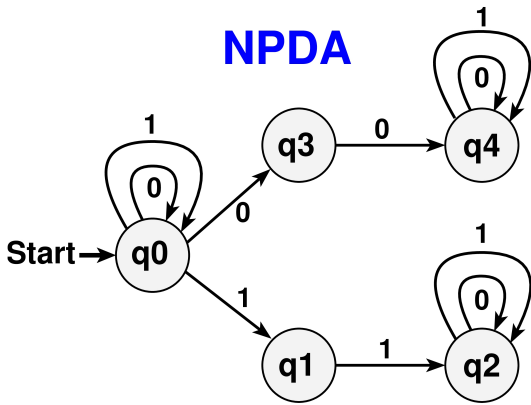


Fig. 2.7: A nondeterministic automaton

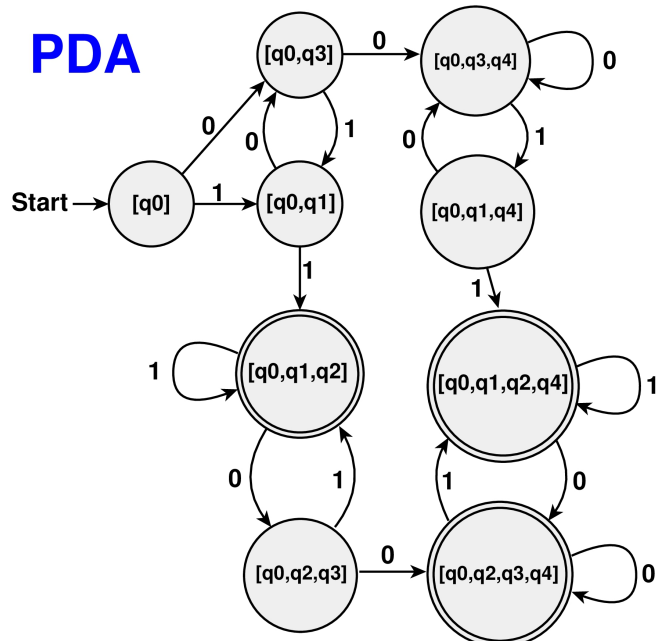


Fig. 2.8: An equivalent deterministic automaton

2.6 Automata with ϵ -Transitions

- Any commitment to a single model of computation quickly proves to be a serious mistake. A variety of theoretical models of computation is generally useful for understanding the complex nature of the concept of computation and provides us with many alternatives for addressing different problems. A particular model of computation may be naturally suited to a certain type of problem, but unsuitable for another type.
- In this section, we present an extended model of a nondeterministic automaton that is particularly suitable for handling regular expressions, which are very common in shell languages and scripting languages such as: `csh`, `awk`, `grep`, `perl`, `python`.

Definition 2.12: A *nondeterministic automaton with ϵ -transitions* is characterized by the usual five components

$$M = (Q, \Sigma, \delta, q_0, F)$$

with a small (but significant) difference in the transition function:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$$

The symbol set Σ is augmented by the empty string ϵ .

- This means that, in certain states, it is possible to move from one state to another without reading a symbol – such a move is called an ϵ -*transition*. Graphically, this means that we are permitted to label edges in the transition diagram with the empty string ϵ . The transition between the two states along such an edge is made without reading a symbol.

Example 2.6:

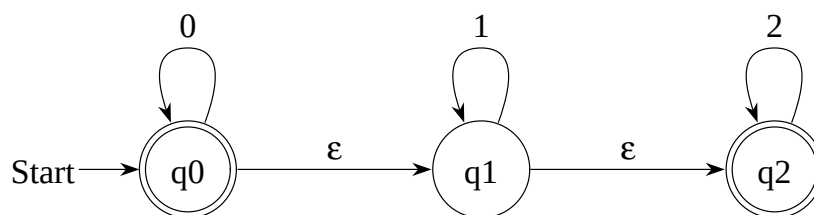


Fig. 2.9: A nondeterministic automaton with ϵ -transitions

This automaton includes two ϵ -transitions, and it is not difficult to see that it defines the language of all strings over the alphabet $\Sigma = \{0, 1, 2\}$ consisting of a sequence of zeros followed by a sequence of 1's and finally a sequence of 2's (each of these sequences may be empty!). The automaton accepts strings such as **000112**, **00222**, ϵ , **2222**, etc'

- As before, we will find that automata with ϵ -transitions cannot define new languages that we could not define using an ordinary automaton or a nondeterministic automaton, but an ordinary nondeterministic automaton without ϵ -transitions that defines the same language may be much more complicated!

➤ Here are several examples of computation sequences

$$\begin{aligned}
 q_0 &\xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{1} q_1 \xrightarrow{\epsilon} q_2 \\
 q_0 &\xrightarrow{\epsilon} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_1 \xrightarrow{\epsilon} q_2 \\
 q_0 &\xrightarrow{\epsilon} q_1 \xrightarrow{\epsilon} q_2
 \end{aligned}$$

All the sequences end in the accepting state q_2 . The first sequence proves that M accepts the string **001**, the second accepts **111**, and the last accepts ϵ .

➤ Here is a complete formal definition of M :

$Q = \{q_0, q_1, q_2\}$	δ	0	1	2	ϵ
$\Sigma = \{0, 1, 2\}$	q_0	$\{q_0\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
$q_0 = \text{Start}$	q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
$F = \{q_2\}$	q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

Note that a new column for ϵ has been added to the transition table!

➤ Our main objective in this section is to prove that every nondeterministic automaton with ϵ -transitions is equivalent to an ordinary nondeterministic automaton. For this purpose, we need the following concepts.

➤ ϵ -closure

For every state q , we denote by $C_\epsilon(q)$ the set of all states p such that there is a path from state q to state p all of whose edges are labeled ϵ .

$$C_\epsilon(q) = \left\{ p \in Q : \text{there is a path } q \xrightarrow{\epsilon} r_1 \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} r_i \xrightarrow{\epsilon} p \right\}$$

The set $C_\epsilon(q)$ always contains q .

➤ For example, in the automaton above,

$$\begin{aligned}
 C_\epsilon(q_0) &= \{q_0, q_1, q_2\} \\
 C_\epsilon(q_1) &= \{q_1, q_2\} \\
 C_\epsilon(q_2) &= \{q_2\}
 \end{aligned}$$

➤ We now extend the definition of the operator C_ϵ in a completely natural way, so that it operates not only on a single state but also on a set of states $P \subseteq Q$

$$C_\epsilon(P) = \bigcup_{q \in P} C_\epsilon(q)$$

➤ Extending the transition function δ to a function δ^*



As we did for an ordinary automaton and a nondeterministic automaton, in this case too we need to define the extension δ^* . Here, however, the ε -transitions introduce a certain difficulty; to overcome it, we first had to define the operator C_ε .

- a. For every state q , define

$$\delta^*(q, \varepsilon) = C_\varepsilon(q)$$

- b. For every string w in Σ^* , and every symbol a ,

$$\delta^*(q, wa) = C_\varepsilon(P)$$

where

$$P = \bigcup_{r \in \delta^*(q, w)} \delta(r, a)$$

In other words: If, during the recursion, we find that

$$\delta^*(q, w) = \{r_1, r_2, \dots, r_k\}$$

then we first perform the intermediate step

$$P = \delta(r_1, a) \cup \delta(r_2, a) \cup \dots \cup \delta(r_k, a)$$

and finally take the ε -closure of P .

◆ **Example 2.7:**

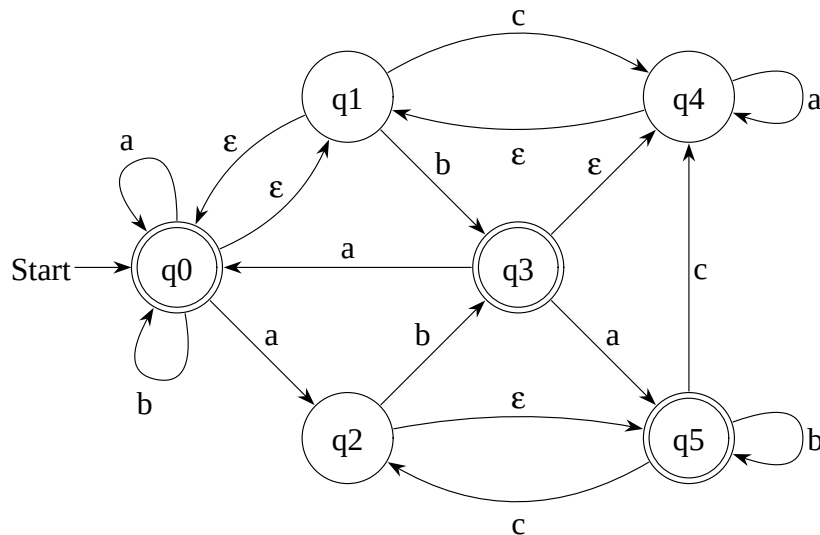


Fig. 2.10: A nondeterministic automaton with ε -transitions

The strings **babb**, **acb**, **aaa** are accepted by this automaton, as the following computation

sequences demonstrate:

$$\begin{aligned}
 q_0 &\xrightarrow{\epsilon} q_1 \xrightarrow{b} q_3 \xrightarrow{a} q_5 \xrightarrow{b} q_5 \xrightarrow{b} q_5 \\
 q_0 &\xrightarrow{a} q_2 \xrightarrow{\epsilon} q_5 \xrightarrow{c} q_2 \xrightarrow{b} q_3 \xrightarrow{\epsilon} q_4 \xrightarrow{\epsilon} q_1 \xrightarrow{\epsilon} q_0 \\
 q_0 &\xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0
 \end{aligned}$$

We can also illustrate the function δ^* in this automaton.

For example, let us compute $\delta^*(q_0, ac)$:

$$\begin{aligned}
 \delta^*(q_0, \epsilon) &= \{q_0, q_1\} \\
 P &= \delta(q_0, a) \cup \delta(q_1, a) \\
 &= \{q_0, q_2\} \cup \emptyset \\
 &= \{q_0, q_2\} \\
 \delta^*(q_0, a) &= C_\epsilon(\{q_0, q_2\}) \\
 &= C_\epsilon(q_0) \cup C_\epsilon(q_2) \\
 &= \{q_0, q_1\} \cup \{q_2, q_5\} \\
 &= \{q_0, q_1, q_2, q_5\} \\
 P &= \delta(q_0, c) \cup \delta(q_1, c) \cup \delta(q_2, c) \cup \delta(q_5, c) \\
 &= \emptyset \cup \{q_4\} \cup \emptyset \cup \{q_2, q_4\} \\
 &= \{q_2, q_4\} \\
 \delta^*(q_0, ac) &= C_\epsilon(\{q_2, q_4\}) \\
 &= C_\epsilon(q_2) \cup C_\epsilon(q_4) \\
 &= \{q_2, q_5\} \cup \{q_4, q_1, q_0\} \\
 &= \{q_0, q_1, q_2, q_4, q_5\}
 \end{aligned}$$

➤ For convenience, we also extend the functions δ , δ^* to sets of states

$$\delta : \mathcal{P}(Q) \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q), \quad \delta^* : \mathcal{P}(Q) \times \Sigma^* \rightarrow \mathcal{P}(Q)$$

in the following natural way:

$$\begin{aligned}
 \delta(P, a) &= \bigcup_{q \in P} \delta(q, a) \\
 \delta^*(P, w) &= \bigcup_{q \in P} \delta^*(q, w)
 \end{aligned}$$

➤ **Exercise 2.2:** Compute $\delta(P, a)$ and $\delta^*(P, \epsilon)$ where $P = \{q_0, q_3\}$ and M is the automaton in the second example.

➤ **Question:** Is $\delta(P, a) = \delta^*(P, a)$?

➤ **Remark:** Unlike the preceding cases, for an automaton with ε -transitions, $\delta^*(q, a)$ need not be identical to $\delta(q, a)$. This is because $\delta^*(q, a)$ is the set of all states that can be reached from state q along edges labeled a or ε , whereas $\delta(q, a)$ is the set of all states that can be reached from state q along edges labeled a only! Similarly, $\delta^*(q, \varepsilon)$ is not necessarily equal to $\delta(q, \varepsilon)$. Therefore, when dealing with an automaton with ε -transitions, one must carefully distinguish between δ and δ^* .

❗ **Definition 2.13:** The language $L(M)$ accepted by a nondeterministic automaton with ε -transitions is

$$L(M) = \{ w \in \Sigma^* : \delta^*(q_0, w) \text{ contains a state in } F \}$$

❗ **Theorem 2.4:** If L is a language accepted by a nondeterministic automaton with ε -transitions, then L is also accepted by a nondeterministic automaton without ε -transitions.

✔ **Proof:** We present a useful method for converting an automaton with ε -transitions into one without ε -transitions. Let

$$M = (Q, \Sigma, \delta, q_0, F)$$

be an automaton with ε -transitions accepting the language L . We construct a new automaton

$$M' = (Q, \Sigma, \delta', q_0, F')$$

without ε -transitions that accepts the same language L . The automaton M' is identical to the automaton M in every respect except for the transition function δ' and the set of accepting states.

◆ **Definition of F' :**

If $C_\varepsilon(q_0)$ contains an accepting state in F , define $F' = F \cup \{q_0\}$. Otherwise, define $F' = F$.

◆ **Definition of the transition function δ' :**

We use the extended transition function δ^* of M , which we defined in the preceding sections. For every $q \in Q$ and every $a \in \Sigma$, define

$$\delta'(q, a) = \delta^*(q, a)$$

◆ Note that the automaton M' has no ε -transitions! This is because at no stage did we define $\delta(q_0, \varepsilon)$. Therefore, there is no risk in also using the symbol δ' for the extended transition function δ'^* of M' (but one must still distinguish between δ and δ^* !).

◆ At this stage, we have finished presenting the algorithm by which we convert an automaton with ε -transitions into an equivalent automaton without ε -transitions. We now prove the correctness of the algorithm. That is, we must prove that $L(M') = L$. For this purpose, we would like to



prove, by induction on the length of a string w , the stronger assertion:

$$\forall w \in \Sigma^* : \quad \delta'(q_0, w) = \delta^*(q_0, w)$$

Our theorem would follow immediately from this assertion. But this assertion may fail for $w = \epsilon$, because $\delta'(q_0, \epsilon) = \{q_0\}$ whereas $\delta^*(q_0, \epsilon) = C_\epsilon(q_0)$. We therefore begin our induction at the stage $n = 1$.

- ◆ **Base case:** If $|w| = 1$, then $w = a$ for some symbol $a \in \Sigma$. This follows immediately from the definition of δ' :

$$\delta'(q, a) = \delta^*(q, a)$$

- ◆ **Induction hypothesis:** For every string u of length n ,

$$\delta'(q, u) = \delta^*(q, u)$$

- ◆ **Inductive step:** Let w be a string of length $n + 1$. We must prove that

$$\delta'(q, w) = \delta^*(q, w)$$

First write w as

$$w = ua$$

where u is a string of length n , and a is the last symbol of w . Then

$$\delta'(q_0, w) = \delta'(q_0, ua) = \delta'(\delta'(q_0, u), a)$$

By the induction hypothesis, $\delta'(q_0, u) = \delta^*(q_0, u)$. Denote

$$P = \delta^*(q_0, u)$$

Thus, we must prove that

$$\delta'(P, a) = \delta^*(q_0, ua)$$

But

$$\delta'(P, a) = \bigcup_{q \in P} \delta'(q, a) = \bigcup_{q \in P} \delta^*(q, a) = C_\epsilon(P)$$

By the definition of δ^* ,

$$C_\epsilon(P) = \delta^*(q_0, w)$$

We have therefore obtained

$$\delta'(q_0, w) = \delta^*(q_0, w)$$

- ◆ To complete the proof, we prove the following assertion:

For every string $w \in \Sigma^*$, $\delta'(q_0, w)$ contains a state in F' if and only if $\delta^*(q_0, w)$ contains a state in F .

- ◆ If $w = \varepsilon$, then the assertion follows immediately from the definition of F' .

In detail: $\delta'(q_0, \varepsilon) = \{q_0\}$, and if $q_0 \in F$, then by the definition of F' , $q_0 \in F'$. Conversely, if $\delta^*(q_0, \varepsilon)$ contains a state in F , then this state automatically belongs to F' because $F \subseteq F'$.

- ◆ Now suppose that $w \neq \varepsilon$. Then $w = ua$ for suitable $u \in \Sigma^*$, $a \in \Sigma$.

If $\delta^*(q_0, w)$ contains a state in F , then since $\delta'(q_0, w) = \delta^*(q_0, w)$, $\delta'(q_0, w)$ also contains that state.

Conversely, if $\delta'(q_0, w)$ contains a state in F' other than q_0 (and hence in F), then the equality $\delta'(q_0, w) = \delta^*(q_0, w)$ implies that this state also belongs to $\delta^*(q_0, w)$. If this state is q_0 but $q_0 \notin F$ (that is, $q_0 \in \delta'(q_0, w) \cap F'$ but $q_0 \notin F$), then denote

$$P = \delta(\delta^*(q_0, u), a)$$

By the definition of δ^* ,

$$\delta^*(q_0, w) = C_\varepsilon(P)$$

The equality $\delta'(q_0, w) = \delta^*(q_0, w)$ implies that $q_0 \in C_\varepsilon(P)$. Therefore $C_\varepsilon(q_0) \subseteq C_\varepsilon(P)$, and hence also $C_\varepsilon(q_0) \subseteq \delta'(q_0, w)$. The only reason that state q_0 was included in F' is that the set $C_\varepsilon(q_0)$ contains a state $q_f \in F$; therefore, necessarily $q_f \in \delta^*(q_0, w)$. ■

- ◆ **Example 2.8:** We convert the following automaton into an automaton without ε -transitions.

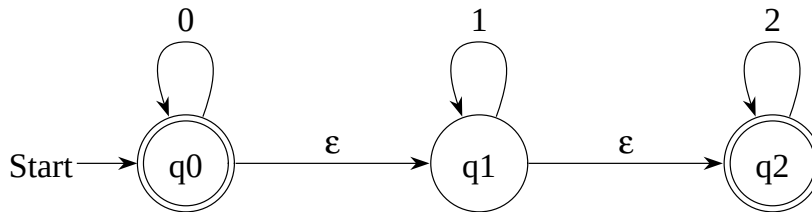


Fig. 2.11: A nondeterministic automaton with ε -transitions

We compute the transition table for δ' using the formula $\delta'(q, a) = \delta^*(q, a)$.

$Q = \{q_0, q_1, q_2\}$	δ'	0	1	2
$\Sigma = \{0, 1, 2\}$	q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_1\}$
$q_0 = \text{Start}$	q_1	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
$F' = \{q_0, q_1, q_2\}$	q_2	$\emptyset$	$\emptyset$	$\{q_2\}$

Here is the resulting automaton:

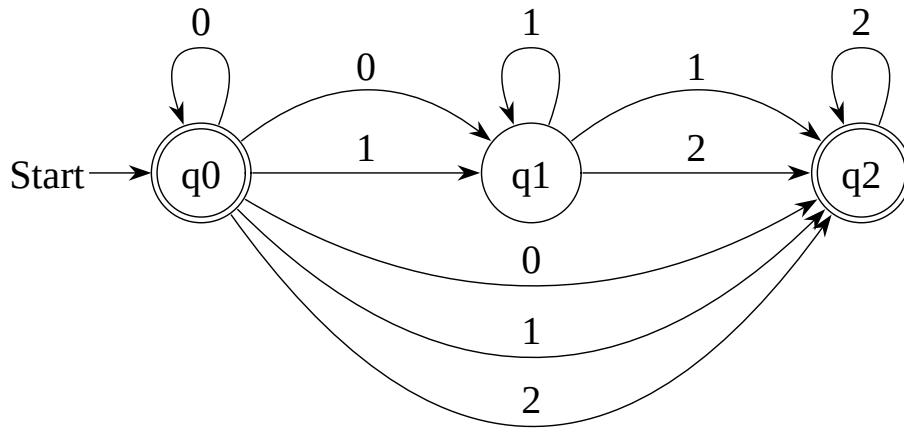


Fig. 2.12: An equivalent nondeterministic automaton without ϵ -transitions

Exercises for chapter 2

Guidance: Some of the exercises can be solved, or their results verified, by using a simple simulation program that can be run in an Internet browser. We would be pleased to receive code from students for solutions to additional exercises.

<https://colab.research.google.com/github/samyzaf/notebooks/blob/main/fa2.ipynb>

1. Consider the automaton M in Figure 2.13. Briefly explain why the automaton M is deterministic. What are the accepting states of M ? Write a transition table for M . Also write the computation sequences for the strings $u = 0010110$, $v = 100111011$, $w = 1101110100$. Which strings are accepted by M ?

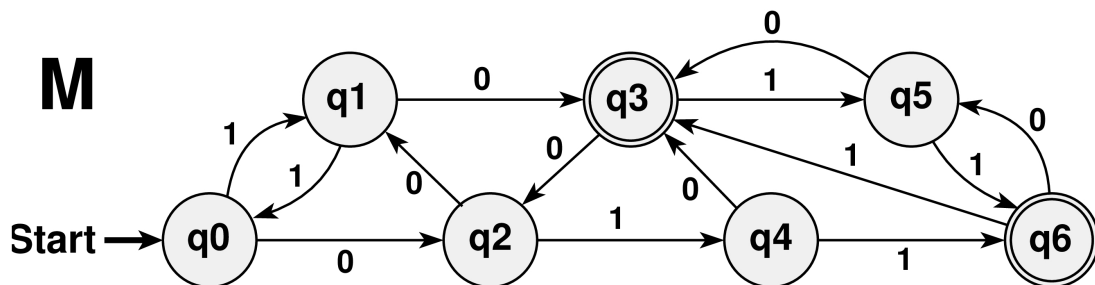


Fig. 2.13: A deterministic automaton M with seven states $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ over the alphabet $\{0, 1\}$

2. Construct a deterministic automaton over the alphabet $\Sigma = \{a, b, c\}$ that accepts the language defined by
 - a. The set of all words containing exactly one occurrence of a and exactly one occurrence of b .
 - b. The set of all words containing exactly one occurrence of a or exactly one occurrence of b .
 - c. The set of all words containing exactly one occurrence of a or exactly one occurrence of b , but not both.
 - d. The set of all words containing exactly one occurrence of a and at least two occurrences of c .
 - e. $L = \{ac^3wb^2 : w \in \{a, b, c\}^*\}$

f. $L = \{ab^n : n \geq 1, n \neq 5\}$

g. The set of all words w in which the number of occurrences of the symbol a is divisible by 5.



3. Let L be a language over an alphabet Σ . Prove that $L = L^*$ if and only if $LL \subseteq L$.
4. Let L be the language of all strings over the alphabet $\Sigma = \{1, 2, 3\}$ whose sum of digits is divisible by 5. Construct a deterministic automaton for L .
5. Let r, s be two regular expressions. The expression $r \equiv s$ indicates that the two languages generated by the expressions are equal. Prove or disprove by a counterexample
 - (a) $r(r + s) \equiv rr + rs$
 - (b) $(r^*)^* \equiv r^*$
 - (c) $(rs)^* \equiv r^*s^*$
 - (d) $(r + s)^* \equiv (r^*s^*)^*$
 - (e) $(rs^*r)^* \equiv r^*s^*r^*$
6. Convert the nondeterministic automaton in Figure 2.14 into a deterministic automaton.

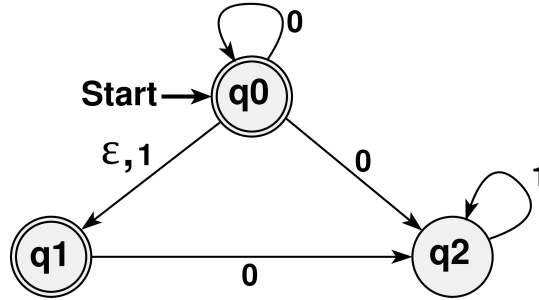


Fig. 2.14: Figure for Exercise 6: a nondeterministic automaton with three states $\{q_0, q_1, q_2\}$ over the alphabet $\Sigma = \{0, 1\}$

7. Convert the nondeterministic automaton in Figure 2.15 into a deterministic automaton

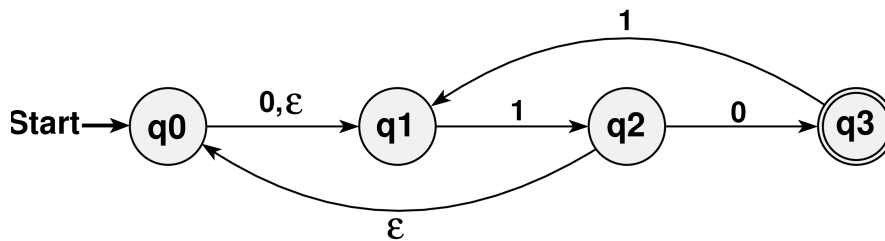


Fig. 2.15: Figure for Exercise 7: a nondeterministic automaton with four states $\{q_0, q_1, q_2, q_3\}$ over the alphabet $\Sigma = \{0, 1\}$

8. Let

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$

$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_1)$$

be two deterministic automata accepting the languages L_1, L_2 , respectively. The **Cartesian**



product of the two automata M_1 , M_2 is defined by

$$M_1 \times M_2 = (Q_1 \times Q_2, \Sigma, \delta, [q_1, q_2], F_1 \times F_2)$$

where for every $p_1 \in Q_1$, $p_2 \in Q_2$, $a \in \Sigma$,

$$\delta([p_1, p_2], a) = [\delta_1(p_1, a), \delta_2(p_2, a)]$$

Prove that the language accepted by the automaton $M_1 \times M_2$ is $L_1 \cap L_2$.

That is: $L(M_1 \times M_2) = L_1 \cap L_2$.

9. Figure 2.16 presents two deterministic automata M_1 , M_2 . Using the product automaton (see Exercise 8), construct an automaton M that accepts the language $L = L(M_1) \cap L(M_2)$. Draw a diagram and write a transition table for $M_1 \times M_2$.



Fig. 2.16: Construct the automaton $M_1 \times M_2$ that accepts the intersection language $L(M_1) \cap L(M_2)$

10. Using the idea of the product of automata (Exercises 8, 9), construct a deterministic automaton for the language of all binary strings that begin with 001 and end with 101.

Note: The problem can be solved more simply by using a nondeterministic automaton. See Exercise 12.

11. Construct a deterministic automaton accepting the language of all strings w with the property: every substring of length 4 of w must contain the substring 01.

Hint: It is easier to construct an automaton M for the complement language $\bar{L}$: $w \in \bar{L}$ if and only if w contains one of the strings: 0000, 1000, 1100, 1110. After constructing M , we obtain $\bar{M}$ by changing accepting states into ordinary states, and ordinary states into accepting states.

12. Construct a nondeterministic automaton with ϵ -transitions for the language of all binary strings that begin with 001 and end with 101.

13. Let M be a nondeterministic automaton with ϵ -transitions over an alphabet Σ , and let w be a string over the alphabet Σ . The **derivation tree** of a string w is a directed labeled graph whose maximal paths are all the possible computation sequences for w . For example, if M is the automaton

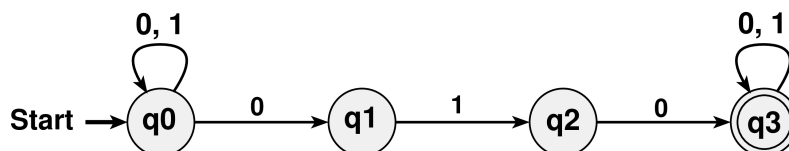


Fig. 2.17: A nondeterministic automaton for the language of strings containing 010

- a. Draw a diagram for the automaton M .
- b. Find the ε -closure of the state sets $\{q_0\}$, $\{q_0, q_1\}$, $\{q_0, q_1, q_2\}$.
- c. Compute $\delta^*(q_0, aaaa)$ and $\delta^*(bbccc)$, where δ^* is the extended transition function of the automaton M .
- d. Compute the derivation trees of the strings $u = aaaa$, $v = bbccc$.



15. We define an addition table for modular arithmetic on the digits $\{1, 2, 3\}$

+	1	2	3
1	2	3	1
2	3	1	2
3	1	2	3

For every string w over the alphabet $\Sigma = \{1, 2, 3\}$, define the sum $\text{sum}(w)$ to be the sum of the digits in the string $w = w_1w_2 \cdots w_n$ according to the modular addition defined in the table. For example, if $w = 221323$, then

$$\text{sum}(w) = 2 + 2 + 1 + 3 + 2 + 3 = 1$$

Construct a nondeterministic automaton (with ε -transitions if necessary) for each of the following languages

$$\begin{aligned} L_1 &= \{w \in \Sigma^* : \text{sum}(w) = 1\} \\ L_2 &= \{w \in \Sigma^* : \text{sum}(w) = \text{sum}(w^R)\} \\ L_3 &= \{uv \in \Sigma^* : \text{sum}(u) = \text{sum}(v)\} \end{aligned}$$

16. What is the language L accepted by the nondeterministic automaton in Figure 2.20? Try to give as simple a description of the language L as possible.

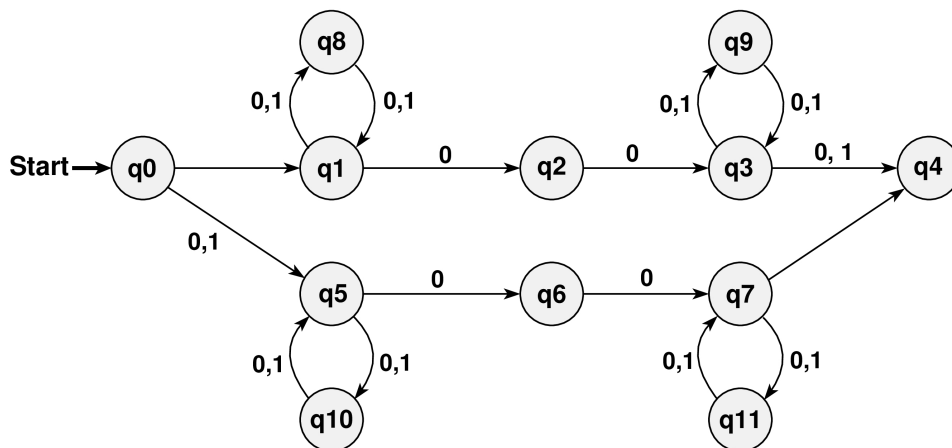


Fig. 2.20: A nondeterministic automaton with ε -transitions and 12 states

17. Internet search engines are sometimes required to find matches for an expression that may contain errors, such as an extra symbol or a missing symbol. In this exercise, we consider two



simple cases. Let L be a given language over an alphabet Σ , and let L_1 be the language consisting of all the words obtained from L by adding some symbol $a \in \Sigma$ to some word $w \in L$ at any position in the word w (including at the beginning or end of the word). Similarly, define the language L_{-1} as the set of all words obtained by deleting some symbol from a word belonging to the language L . Here is a formal definition of the two languages

$$\begin{aligned} L_1 &= \{ uav : uv \in L, a \in \Sigma \} \\ L_{-1} &= \{ uv : uav \in L, a \in \Sigma \} \end{aligned}$$

Given a nondeterministic automaton for the language L , explain how to construct a nondeterministic automaton for the language L_1 (or the language L_{-1}).

18. Let L_1, L_2 be two regular languages over the binary alphabet $\Sigma = \{0, 1\}$. For any two strings of the same length $u = u_1u_2 \cdots u_n, v = v_1v_2 \cdots v_n$, define the operation

$$u \vee v = (u_1 \vee v_1)(u_2 \vee v_2) \cdots (u_n \vee v_n)$$

For example, if $u = 1100, v = 0101$, then $u \vee v = 1101$. Now define the language

$$L_3 = \{ u \vee v : u \in L_1, v \in L_2, |u| = |v| \}$$

Prove that if L_1, L_2 are regular languages, then L_3 is also regular.

19. Prove or disprove by a counterexample: Let L be a given language. If LL is a regular language, then L is regular.
20. Let L be a regular language, and let M be a deterministic automaton with n states that accepts L . Prove that L is infinite if and only if L contains a string w satisfying $n \leq |w| \leq 2n - 1$.
21. Prove or disprove: if L_1, L_2 are regular languages, then $(L_1 \cup L_2)^* = L_1^*(L_2L_1^*)^*$



22. Let L be a given language over an alphabet Σ . Define a new language by

$$L_{rot} = \{ w \in \Sigma^* : w = uv, vu \in L \}$$

In simple terms: $w \in L_{rot}$ if and only if one can divide the string w into two strings, $w = uv$, in such a way that $vu \in L$. Prove that if L is regular, then L_{rot} is also regular.



23. A **finite-state transducer** is a theoretical computing machine characterized by the following components:

Q A set of **states** in which the transducer may be.



- Σ The alphabet over which the input strings of the transducer are defined.
- Δ The alphabet over which the output strings of the transducer are defined.
In most common cases, $\Delta = \Sigma$.
- δ **Transition function** $\delta : Q \times \Sigma \rightarrow Q \times \Delta$
- q_0 The **start state** of the transducer.
This is the state in which the transducer is found at the beginning of every computation.

Unlike a finite automaton, a finite-state transducer receives a string over the alphabet Σ and outputs a string over the alphabet Δ . The transition function δ receives a pair (q, a) , where q denotes the current state of the transducer and a the input symbol (from the alphabet Σ). The value of $\delta(q, a)$ is also a pair (r, b) , where r is the new state of the transducer and b is the output symbol (from the alphabet Δ).

Here is an example of a transducer with 4 states $Q = \{q_0, q_1, q_2, q_3\}$ that converts strings from the alphabet $\Sigma = \{0, 1\}$ to the alphabet $\Delta = \{a, b, c\}$.

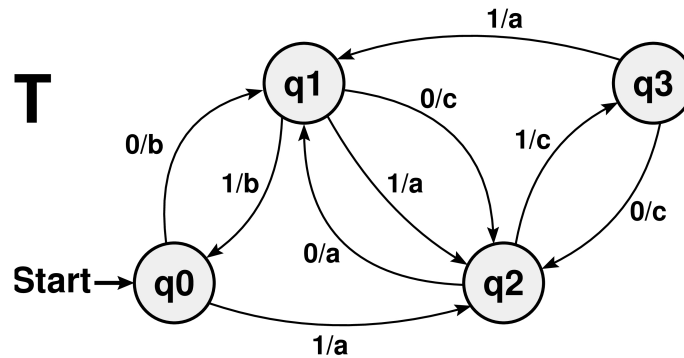


Fig. 2.21: A finite-state transducer T

- Compute the output for the words $w = 000$, $w = 1010$.
- Give an example of a word over the alphabet $\{a, b, c\}$ that is not obtained as the output of any word over the alphabet $\{0, 1\}$.
- Write a suitable transition table for the transition function δ .
- If w is any word over the alphabet Σ , denote the corresponding output by $T(w)$. Give a formal recursive definition of $T(w)$ based on the length of w .
- If L is any language over the alphabet Σ , define the language

$$T(L) = \{T(w) : w \in L\}$$

This is, of course, a language over the alphabet Δ . Give a regular description of the language $T(L)$ for the languages corresponding to the following regular expressions

- (a) 1^* (b) $(10)^*$ (c) $(00)^*$ (d) $(00 + 11)^*$

- Prove: if L is a regular language over the alphabet Σ , then $T(L)$ is a regular language over the alphabet Δ .



24. Formal languages are also used by navigation systems to describe travel routes. One of the simplest examples is shown in Figure 2.22. Our alphabet consists of four symbols

$$\Sigma = \{l, r, u, d\}$$

whose meanings are to move one unit of distance to the left, right, up, or down.

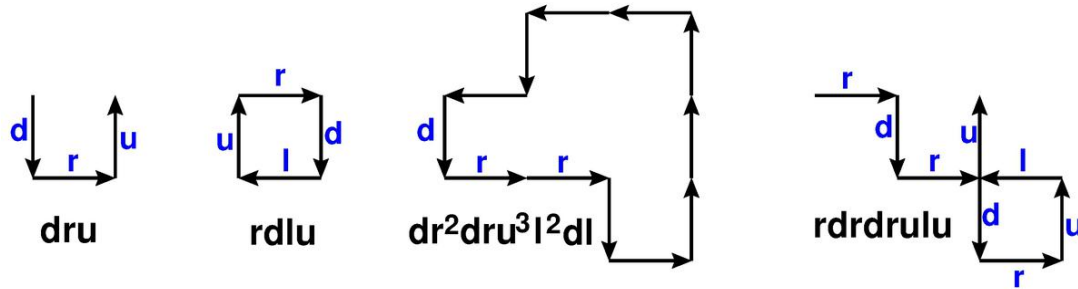


Fig. 2.22: A formal language for describing rectangular paths: d=down, u=up, l=left, r=right

Figure 2.22 shows two examples of open paths, dru , $rdrdrulu$, and two examples of closed paths, $rdlu$, $dr^2dru^3l^2dl$. Let L be the set of all expressions that describe closed paths only. Try to give a simple characterization of the language L , and then construct an automaton that accepts L .



Chapter 3

Regular Languages

In this chapter, we present another way to describe (or generate) the regular languages, but this time the model is **syntactic**, rather than computational. The fact that all the models of computation we have encountered so far have proved to be equivalent and define the same class of languages – the regular languages – indicates that this class is natural in a certain sense and deserves special attention.

3.1 Regular Expressions

It turns out that every regular language can be characterized by a short expression with a simple syntactic form. These expressions are called **regular expressions**, and they are very useful in programming languages and scripting languages such as `csh`, `awk`, `perl`, `python`, and others.

Definition 3.1: Let Σ be a given alphabet (a finite set of symbols), and let L, L_1, L_2 be languages over this alphabet. We recall several operations by which new languages can be generated from given languages, as follows:

- ◆ The **union** of the languages L_1, L_2 is defined by

$$L_1 \cup L_2 = \{ w : w \in L_1 \text{ or } w \in L_2 \}$$

- ◆ The **intersection** of the languages L_1, L_2 is defined by

$$L_1 \cap L_2 = \{ w : w \in L_1 \text{ and } w \in L_2 \}$$

- ◆ The **concatenation** of the languages L_1, L_2 is defined by

$$L_1 L_2 = \{ uv : u \in L_1 \text{ and } v \in L_2 \}$$

- ◆ A **power** of a language L is defined by

$$L^0 = \{\varepsilon\}$$

$$L^1 = L$$

$$L^2 = LL$$

$$L^{n+1} = L^n L$$

- ◆ The **closure** (Kleene Closure) is defined by

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

- ◆ The **positive closure** (Positive Closure) is defined by

$$L^+ = \bigcup_{n=1}^{\infty} L^n$$

- Note that the only difference between L^* and L^+ is that $\varepsilon \in L^*$, whereas $\varepsilon \in L^+$ only if $\varepsilon \in L$.

i Definition 3.2: Definition of regular expressions

- ◆ $\emptyset$ is a regular expression describing the empty language.
- ◆ ε is a regular expression describing the language $\{\varepsilon\}$ (this is not an empty language!).
- ◆ For every symbol $a \in \Sigma$, a is a regular expression describing the language $\{a\}$.
- ◆ If r and s are two regular expressions describing the languages L_1 and L_2 , respectively, then $(r + s)$ is a regular expression describing the language $L_1 \cup L_2$.
- ◆ If r and s are two regular expressions describing the languages L_1 and L_2 , respectively, then (rs) is a regular expression describing the language $L_1 L_2$.
- ◆ If r is a regular expression for the language L , then r^* is a regular expression describing the language L^* .

- The order of the concatenation, union, and closure operations is important. For example, there is a substantial difference between the language described by the regular expression ab^* and the language described by $(ab)^*$! The same applies to the expressions $b + a^*$ and $(b + a)^*$. Therefore, to avoid confusion, parentheses must be used carefully wherever they are called for.
- Nevertheless, excessive use of parentheses may make regular expressions cumbersome and difficult to read. It is customary to omit parentheses when the closure operation $*$ takes precedence over concatenation, and concatenation takes precedence over union.
- We did not include the positive closure operation r^+ in the list of regular expressions, because it can easily be defined by rr^* .
Likewise, we can define a power of a regular expression as follows: if r is a regular expression describing the language L , then r^n is a regular expression describing the language L^n .
- For example, the regular expression $(b + (aa(b^*)))$ can be written more conveniently as $b + aab^*$.

❖ **Example 3.1:** Further examples:

- a. The regular expression $(0 + 1)^*000$ describes the language of all binary strings ending in 000 .
- b. The regular expression $(0 + 1)^*000(0 + 1)^*$ describes the language of all binary strings containing the sequence 000 (anywhere in the string).
- c. The regular expression $(0 + 1)^*000 + 11(1 + 0)^*$ describes the language of all binary strings that end in 000 or begin with 11 .
- d. The regular expression $(a + ab)^*$ describes the language of all strings over the alphabet $\Sigma = \{a, b\}$ that begin with a but do not contain bb .
It is easy to prove by induction on n that the regular expression $(a + ab)^n$ contains no occurrence of the string bb .
- e. The regular expression $(0 + 1)^*(01)^{99}$ describes the language of all binary strings that end with the string $(01)^{99}$.
- f. The regular expression $0^*1^*2^*$ describes the language of all strings over the alphabet $\Sigma = \{0, 1, 2\}$ that begin with any number of zeros, followed by any number of 1 's, and finally any number of 2 's.
- g. The regular expression $00^*11^*22^*$ describes the language of all strings over the alphabet $\Sigma = \{0, 1, 2\}$ that begin with a sequence of zeros, followed by a sequence of 1 's, and finally a sequence of 2 's. Each of the sequences contains at least one element.

The Degree of Complexity of a Regular Expression

- As we saw in the examples above, regular expressions may be simple or complex. In the following definition, we associate with every regular expression a natural number n , $n = 0, 1, 2, \dots$, which reflects the degree of complexity of the expression.

i Definition 3.3: We assign a **degree** to every regular expression according to the following scheme

- a. Regular expressions of degree of complexity 0 : $\emptyset$, ε , a
- b. If r , s are regular expressions with degrees of complexity m , n , respectively, then the degree of complexity of rs and $r + s$ is $\max\{m, n\} + 1$.
- c. If r is a regular expression with degree of complexity n , then the degree of complexity of r^* is $n + 1$.

Notation: The degree of complexity of a regular expression r is denoted by $\deg[r]$.



➤ Using this notation, we can write the following rules:

$$\begin{aligned} \deg[r^*] &= \deg[r] + 1 \\ \deg[rs] &= \max(\deg[r], \deg[s]) + 1 \\ \deg[r + s] &= \max(\deg[r], \deg[s]) + 1 \end{aligned} \tag{3.1}$$

◆ **Example 3.2:**

- a. The degree of the regular expression $(a + ab)^*$ is 3.
- b. The degree of the regular expression bba is 2.
- c. The degree of the regular expression $(bbaa)^*$ is 4.

Proof:

$$\begin{aligned} \deg[(bbaa)^*] &= \deg[bbaa] + 1 \\ &= (\max(\deg[bba], \deg[a]) + 1) + 1 \\ &= (\max(2, 0) + 1) + 1 = 3 + 1 = 4 \end{aligned}$$

- d. The degree of the regular expression $(a + ab)^*(bbaa)^*$ is 5.

Proof:

$$\begin{aligned} \deg[(a + ab)^*(bbaa)^*] &= \\ &= \max\{\deg[(a + ab)^*], \deg[(bbaa)^*]\} + 1 \\ &= \max(3, 4) + 1 = 5 \end{aligned}$$

It is easy to draw a derivation tree for our regular expression – see Figure 3.1. The degree of the expression can then be computed by assigning a degree to every expression in the tree, proceeding upward from the leaves to its root (which is our expression; see Figure 3.2).

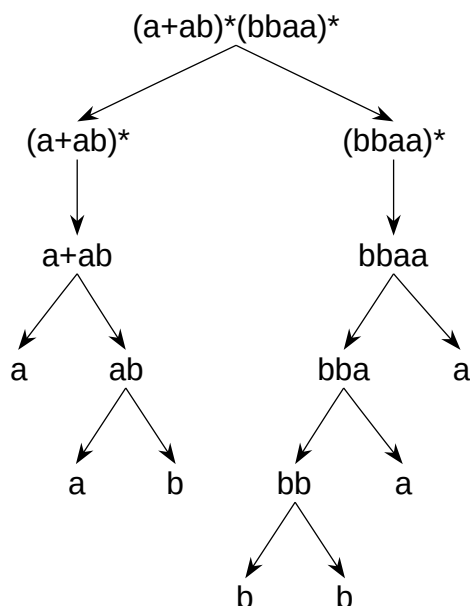


Fig. 3.1: Derivation tree of a regular expression

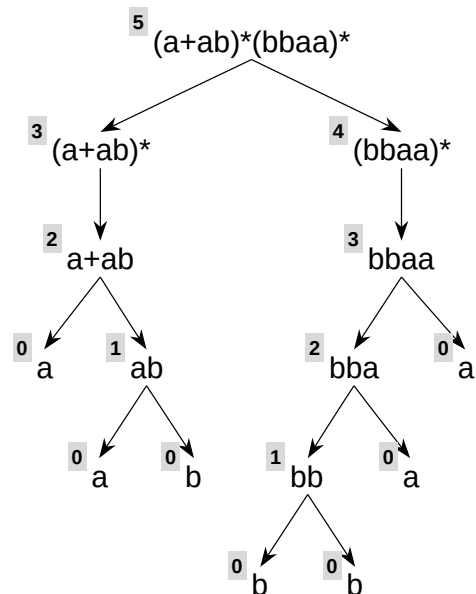


Fig. 3.2: Complexity level (degree) of a regular expression

Equivalence of Regular Expressions and Finite Automata

i Theorem 3.1: Every language L that can be described by a regular expression can also be defined by a nondeterministic automaton with ϵ -transitions.

- The theorem is proved by induction on the complexity degree of a regular expression r . As in previous cases, the proof of the theorem is in fact a useful algorithm for converting a regular expression into an automaton; understanding it is therefore very important for other practical problems as well.
- To prove the theorem above, we prove a more general theorem from which it follows immediately: **For every regular expression r , there is a normal automaton that accepts the language $L(r)$.**

i Definition 3.4: A **normal automaton** is a nondeterministic automaton with ϵ -transitions for which

- a. there is exactly one start state, into which no edge enters.
- b. there is exactly one accepting state, from which no edge leaves.

➤ **Base case:**

If $\deg[r] = 0$, then $r = \emptyset$ or $r = \epsilon$ or $r = a$, $a \in \Sigma$. For each of these expressions, it is easy to construct a normal automaton

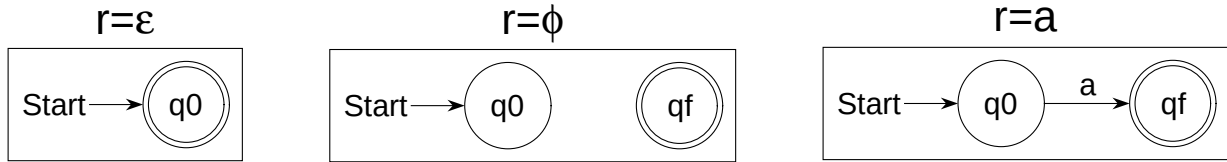


Fig. 3.3: Elementary normal automata

➤ **Induction hypothesis:**

For every regular expression of complexity degree at most n , there is a normal automaton accepting the language $L(r)$.

➤ **We must prove:**

For every regular expression of complexity degree $n + 1$, there is a normal automaton accepting the language $L(r)$.

- Let r be a regular expression of complexity degree $n + 1$. Then one of the following cases must hold:
 - a. $r = r_1 + r_2$, where r_1, r_2 are regular expressions of degree at most n .
 - b. $r = r_1 r_2$, where r_1, r_2 are regular expressions of degree at most n .
 - c. $r = r_1^*$, where r_1 is a regular expression of degree n .
- **Case a': $r = r_1 + r_2$**

By the induction hypothesis, there are normal automata

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, f_1)$$

$$M_2 = (Q_2, \Sigma, \delta_2, q_2, f_2)$$

accepting the languages $L(r_1)$, $L(r_2)$, respectively. Here q_1 , q_2 are the start states of M_1 , M_2 , and f_1 , f_2 are the accepting states of M_1 , M_2 . We may assume that the state sets Q_1 , Q_2 are disjoint, because the states can always be renamed.

By the definition of regular expressions,

$$L(r) = L(r_1) \cup L(r_2)$$

We use the automata M_1 , M_2 as building blocks to construct a normal automaton M for the language $L(r_1) \cup L(r_2)$, according to the scheme in Figure 3.4.

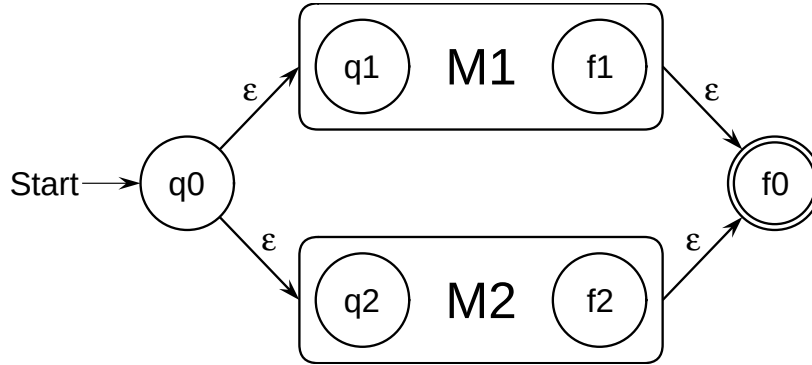


Fig. 3.4: A normal automaton for the language $L(r_1) \cup L(r_2)$

The formal definition of M is

$$M = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma, \delta, q_0, f_0)$$

The state set of M is the union of the states of M_1 and M_2 , together with two new states q_0 , f_0 . The state q_0 is the start state of M , and the state f_0 is the accepting state of M .

The states q_1 , f_1 , q_2 , f_2 are ordinary states of the automaton M .

The transition function δ of M is defined as follows:

$$\delta(q_0, \varepsilon) = \{q_1, q_2\}$$

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & q \in Q_1 - \{f_1\} \\ \delta_2(q, a), & q \in Q_2 - \{f_2\} \end{cases}$$

$$\delta(f_1, \varepsilon) = \{f_0\}$$

$$\delta(f_2, \varepsilon) = \{f_0\}$$

where $a \in \Sigma \cup \{\varepsilon\}$.

➤ **Case b': $r = r_1 r_2$**

Let M_1, M_2 be as in the preceding case. Define the normal automaton M :

$$M = (Q_1 \cup Q_2, \Sigma, \delta, q_1, f_2)$$

In this case:

- ◆ The state set of M is the union of the states of M_1, M_2 only.
- ◆ The start state of M is q_1 (which is also the start state of M_1).
- ◆ The accepting state of M is f_2 (which is also the accepting state of M_2).
- ◆ The transition function δ of M is defined as follows:

$$\begin{aligned} \delta(q, a) &= \begin{cases} \delta_1(q, a), & q \in Q_1 - \{f_1\} \\ \delta_2(q, a), & q \in Q_2 \end{cases} \\ \delta(f_1, \varepsilon) &= \{q_2\} \end{aligned}$$

where $a \in \Sigma \cup \{\varepsilon\}$.

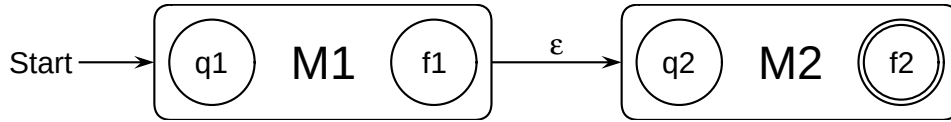


Fig. 3.5: A normal automaton for the language $L(r_1)L(r_2)$

It is not difficult to see that

$$L(M) = \{uv : u \in L(M_1) \text{ and } v \in L(M_2)\}$$

That is,

$$L(M) = L(M_1)L(M_2) = L(r_1)L(r_2) = L(r)$$

➤ **Case c': $r = r_1^*$**

Let M_1 be as before. Define the normal automaton M :

$$M = (Q_1 \cup \{q_0, f_0\}, \Sigma, \delta, q_0, f_0)$$

where the transition function δ is given by

$$\begin{aligned} \delta(q_0, \varepsilon) &= \{q_1, f_0\} \\ \delta(f_1, \varepsilon) &= \{q_1, f_0\} \\ \delta(q, a) &= \delta_1(q, a), \quad q \in Q_1 - \{f_1\}, \quad a \in \Sigma - \{f_1\} \end{aligned}$$

By Figure 3.6, it is easy to determine that $w \in L(M)$ if and only if $w \in L(M_1)^*$. Since

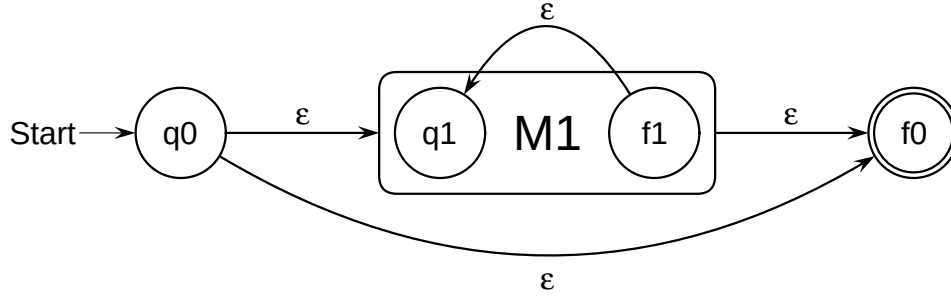


Fig. 3.6: A normal automaton for the language $L(r_1^*)$

$L(M_1) = L(r_1)$, we obtain $L(M) = L(r_1)$. This completes the proof of the theorem. ■

◆ **Example 3.3:** Using the algorithm presented in the proof of the theorem, we find the corresponding automaton for the regular expression $(ab)^* + bb$ (see Figure 3.7).

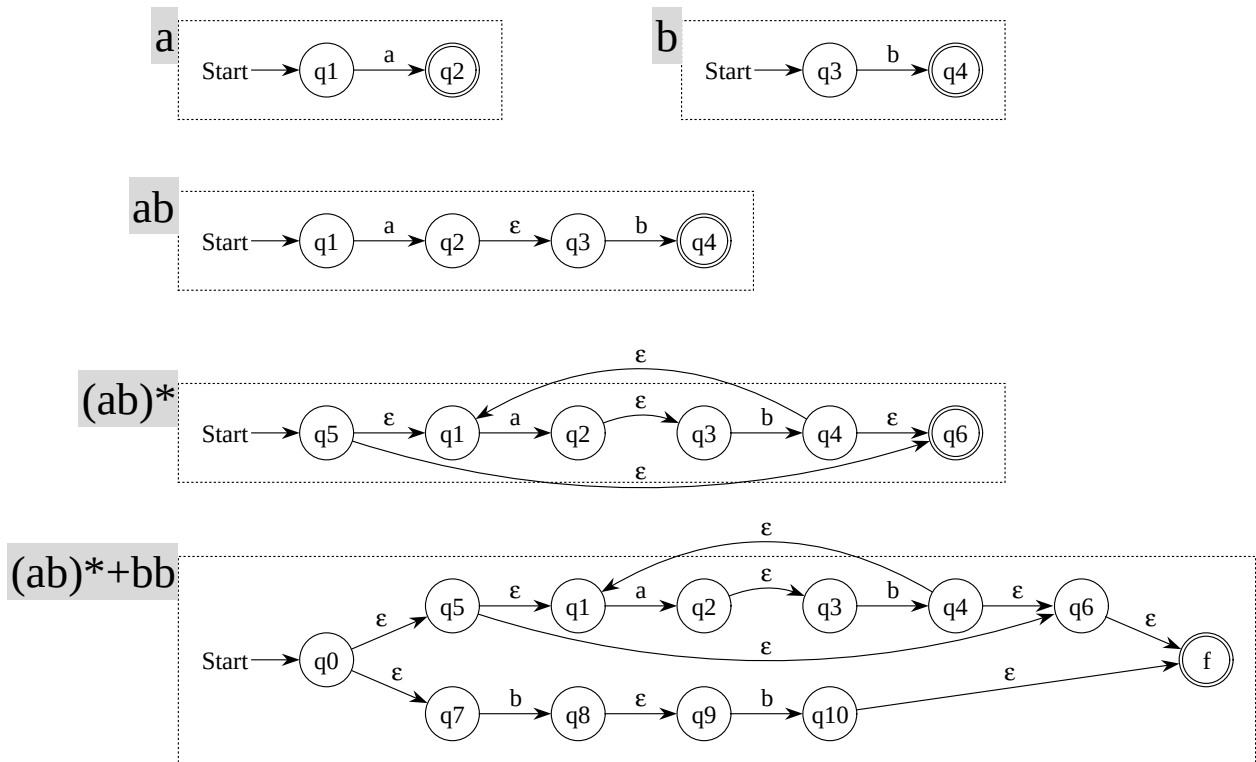


Fig. 3.7: A normal automaton for the regular expression $(ab)^* + bb$

3.2 Generalized Automata

➤ To complete our program, it remains to prove the converse of the preceding theorem.

❗ **Theorem 3.2:** Every language L accepted by a deterministic automaton can be described by a regular expression.

➤ For this purpose, we introduce another automaton model: a **generalized automaton**.

- i Definition 3.5:** A *generalized automaton* is an automaton in which
- every edge is labeled by a regular expression over the alphabet Σ .
 - there is exactly one start state q_s , into which no edge enters.
 - there is exactly one accepting state q_f , from which no edge leaves.
 - a transition from state q_i to state q_j is possible only by reading a string that matches the regular expression labeling the edge from q_i to q_j .
 - A string w is accepted by the automaton if and only if there is a decomposition of w into k strings $w = w_1 w_2 \cdots w_k$ and there is a computation sequence

$$q_s = p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \cdots p_{k-1} \xrightarrow{w_k} q_f = p_k$$

such that at every stage $1 \leq i \leq k$, w_i matches the regular expression labeling the edge from state q_{i-1} to state q_i .

Example 3.4: Let M be the automaton whose transition diagram is shown in Figure 3.8.

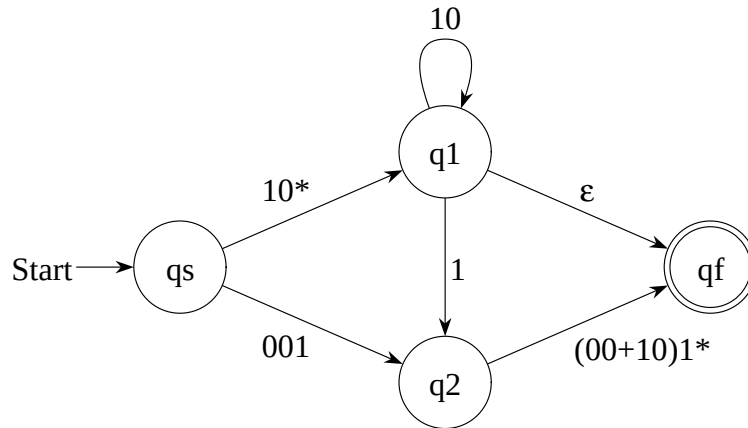


Fig. 3.8: A generalized automata

Some examples of strings accepted by the automaton M are

00100, 100000, 10001010, 1110111

For example, a computation sequence for $w = 10001010$ is

$$q_s \xrightarrow{1000} q_1 \xrightarrow{10} q_1 \xrightarrow{10} q_1 \xrightarrow{\varepsilon} q_f$$

- Although at first glance this type of automaton appears capable of defining languages more complicated than the regular languages, it turns out that this is not the case. We prove that every generalized automaton can be transformed (by a suitable algorithm) into a regular expression.
- We proved that every language generated by a regular expression is also accepted by a suitable deterministic automaton. Therefore, every language accepted by a generalized automaton is also

accepted by an ordinary deterministic automaton.

- On the other hand, it is clear that every ordinary deterministic automaton is in fact a generalized automaton in which all the regular expressions are simple symbols (recall that every symbol a is a regular expression describing itself alone).
- The preceding assertion is not entirely accurate! In an ordinary deterministic automaton, there may be edges entering the start state q_0 , more than one accepting state, or edges leaving one of the accepting states. None of these is permitted in a generalized automaton!
- But all these problems can easily be overcome by adding two new states, q_s and q_f , as shown in Figures 3.9, 3.10.

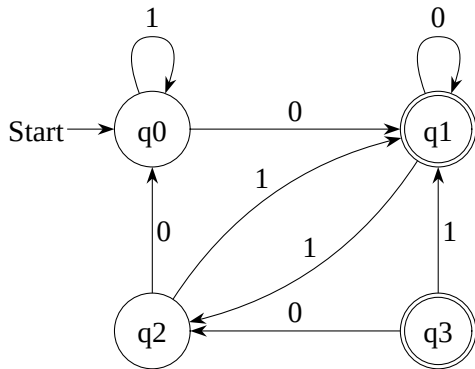


Fig. 3.9: A generalized automaton with an edge entering the start state and an edge leaving a final state

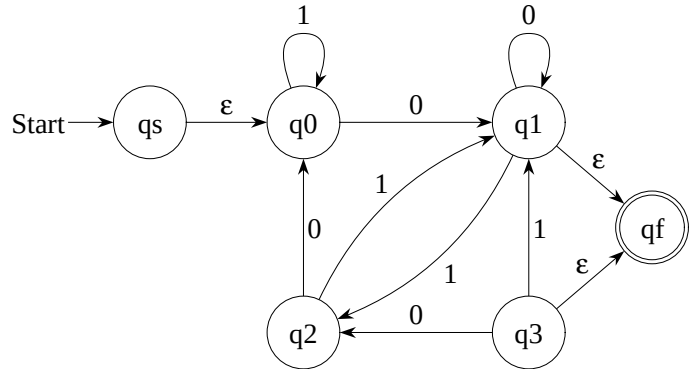


Fig. 3.10: A generalized automaton with standard start and final states

- It follows that every ordinary deterministic automaton can be transformed into an equivalent generalized automaton.
- We prove that every generalized automaton can be transformed into a regular expression, thereby proving the second half of the equivalence theorem (between deterministic automata and regular expressions).
- After proving that every generalized automaton can be converted into an equivalent regular expression, we can summarize all the results we have proved in the following diagram:

i Theorem 3.3: There is an algorithm by which every generalized automaton with n states, $n > 2$, can be converted into an equivalent generalized automaton with $n - 1$ states.

- ✓ **Proof:** Let M be a generalized automaton with n states, $n > 2$. We present an algorithm for reducing the number of states of M by one. Arbitrarily choose a state q_k distinct from the start and accepting states.

For every pair of states q_i, q_j ($i, j \neq k, i \neq j$) whose relationship to state q_k matches the case on the left side of the following diagrams, perform the operation on the right side:

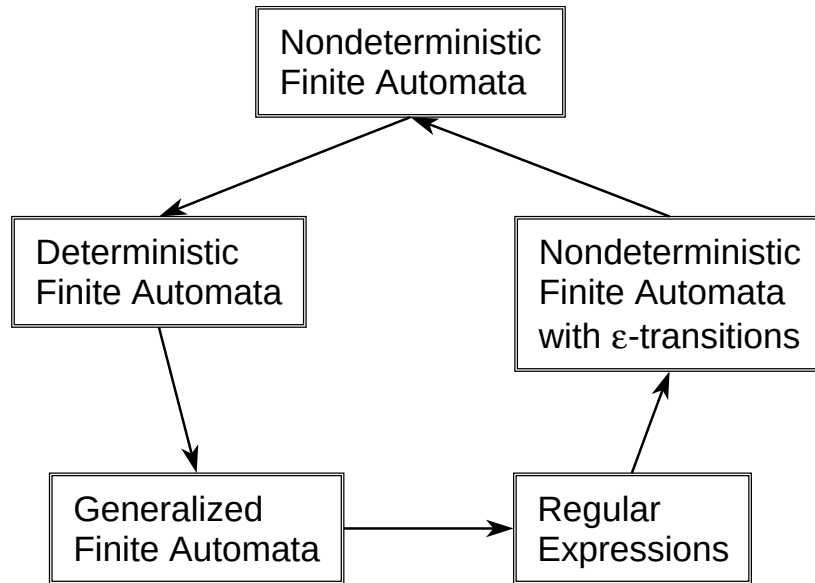
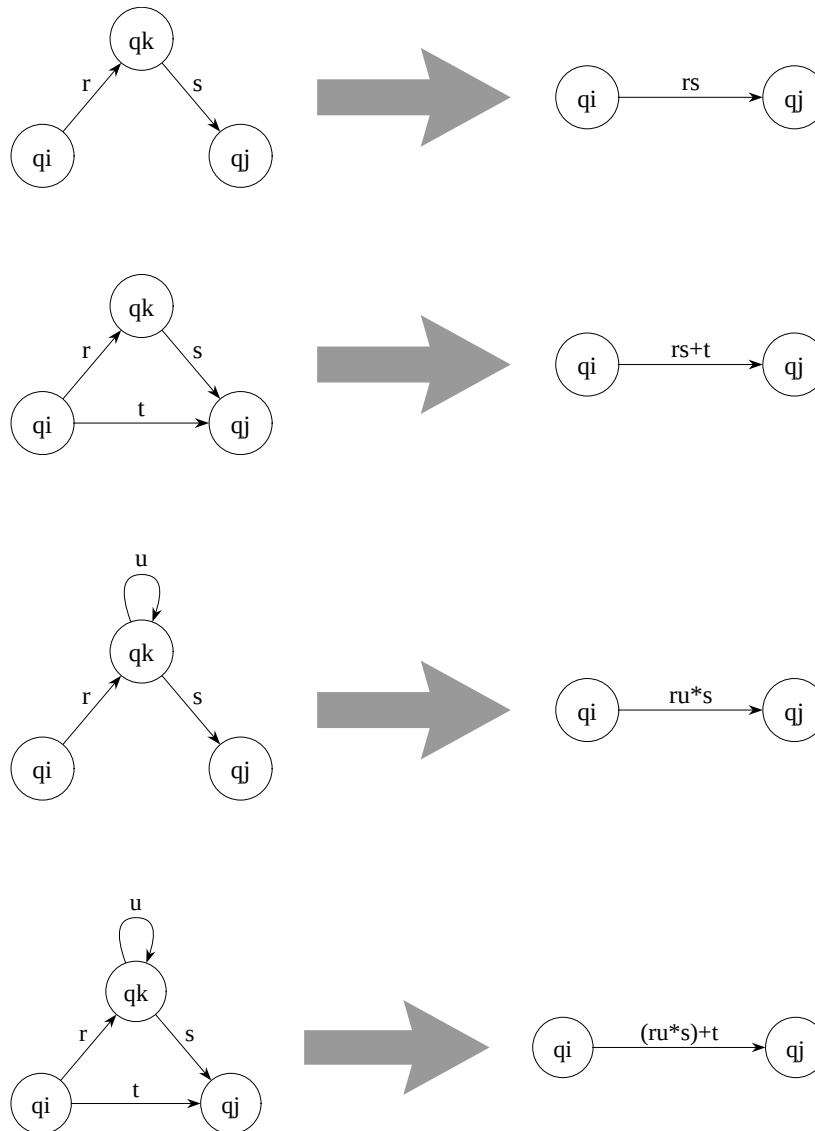


Fig. 3.11: Summary of results



Likewise, for every state q_i , $i \neq k$, whose relationship with state q_k matches the left side of

the following diagrams, perform the operation on the right

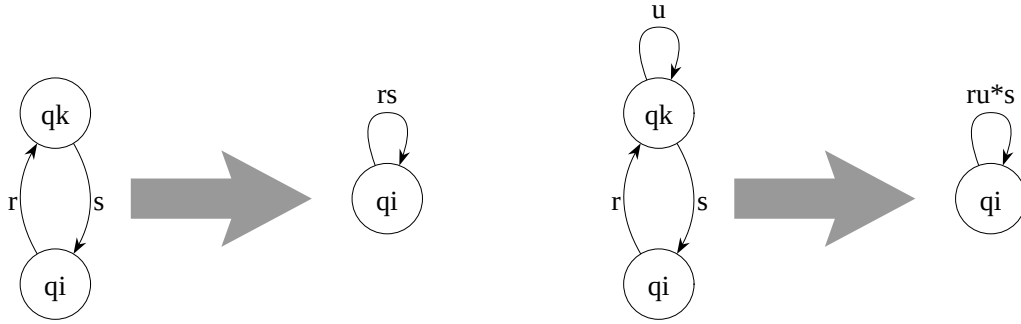


Fig. 3.16

- We emphasize that eliminating state q_k from the automaton M may require a very large number of operations of the types above. One must ensure that all are carried out with the utmost care.
- Let M' denote the generalized automaton obtained from the procedure above. It is not difficult to prove that for every string $w \in \Sigma^*$, w is accepted by M if and only if w is accepted by M' .
- Therefore, the automaton M' is equivalent to the automaton M .
- We call the above procedure **Reduce** for short. Thus, if M' is the automaton obtained from the automaton M by eliminating state q_k , we can write this more conveniently as

$$M' = \text{Reduce}(M, q_k)$$

- ◆ **Example 3.5:** For example, eliminate state q_1 from the generalized automaton in Figure 3.17. The result is shown in Figure 3.18.

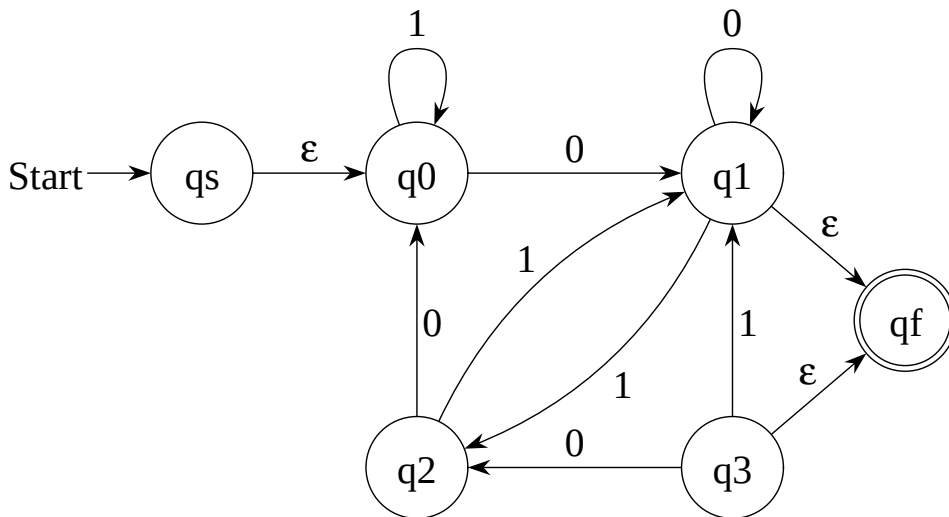


Fig. 3.17: A generalized automaton

Note that eliminating the single state q_1 significantly affected the entire structure of the automaton – the labels of most of the edges changed.

- An immediate consequence of the preceding theorem is the following theorem.

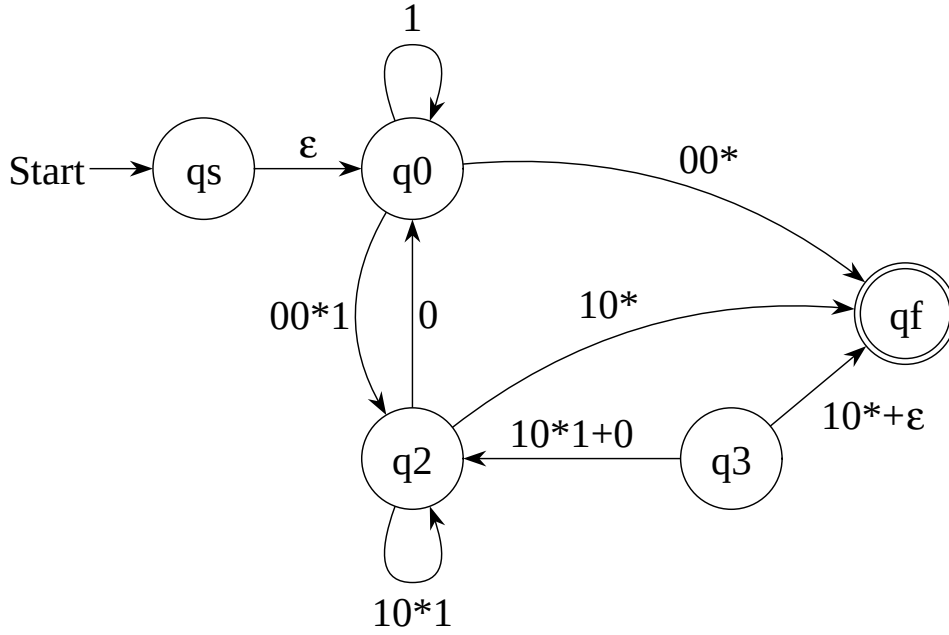


Fig. 3.18: An equivalent generalized automaton without state q_1

Theorem 3.4: Every generalized automaton is equivalent to a generalized automaton with only two states – q_s and q_f .

Proof: Let M be a generalized automaton with n states. If $n = 2$, we are done. If $n > 2$, applying the procedure **Reduce** to the automaton M a total of $n - 2$ times eventually produces a minimal generalized automaton with only two states. ■

We can now formulate the algorithm for converting an ordinary deterministic automaton into a regular expression:

1. Convert M into a generalized automaton by adding two new states q_s and q_f and adding suitable edges (labeled by ϵ).
2. Choose a state q_k distinct from q_s and q_f , and eliminate q_k :

$$M := \text{Reduce}(M, q_k)$$

Continue to call the resulting automaton M .

3. If in the automaton M only the two states q_s and q_f remain, proceed to the next step. Otherwise, return to the preceding step.
4. Combine all the regular expressions labeling all edges from state q_s to state q_f to obtain a minimal generalized automaton.
5. The regular expression labeling the single edge is the desired regular expression.

Example 3.6: The following generalized automaton is given

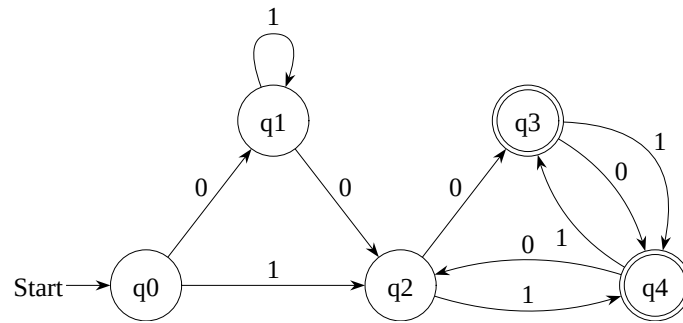


Fig. 3.19: The generalized automaton

We move to an automaton with a single final state:

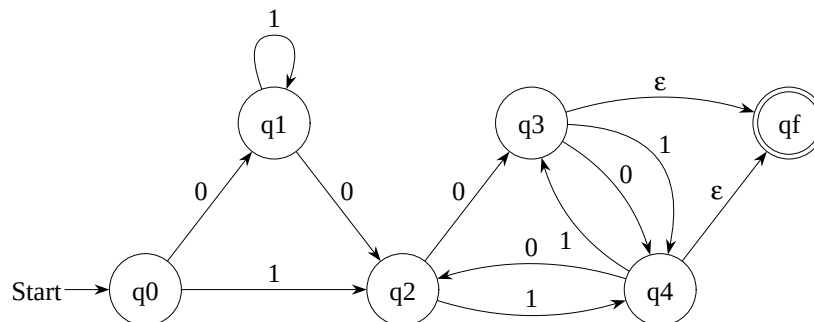
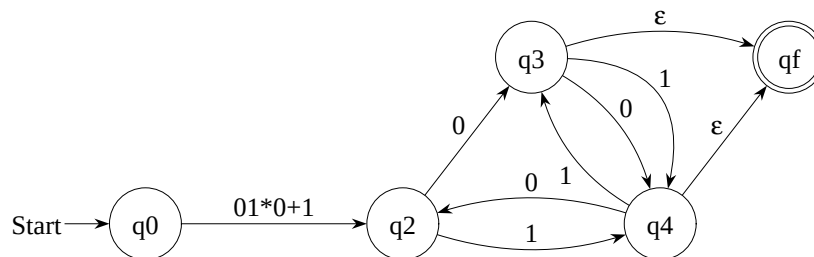
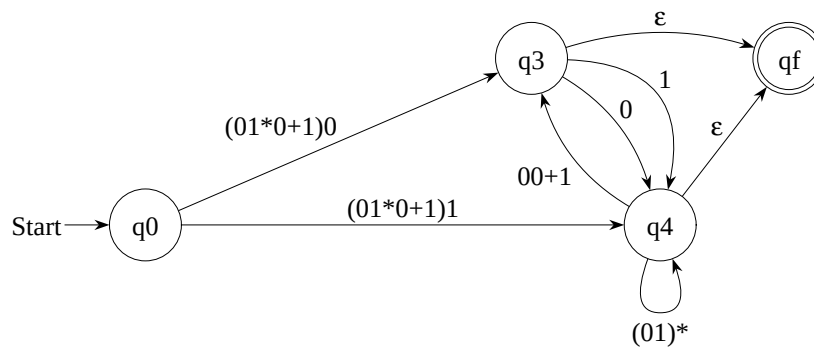


Fig. 3.20: An equivalent generalized automaton with a single final state

After eliminating state q_1 :

Fig. 3.21: An equivalent generalized automaton after eliminating state q_1

After eliminating state q_2 :

Fig. 3.22: An equivalent generalized automaton after eliminating state q_2

After eliminating state q_3 :

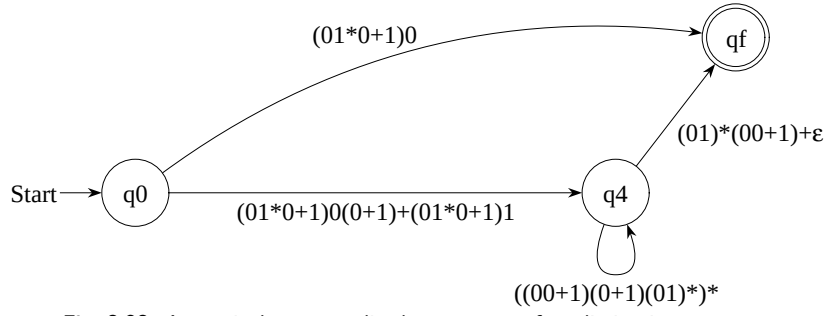


Fig. 3.23: An equivalent generalized automaton after eliminating state q_3

The regular expression on the edge (q_0, q_4) can be simplified by elementary algebra:

$$\begin{aligned}
 (01 * 0 + 1)0(0 + 1) + (01 * 0 + 1)1 &= (01 * 0 + 1)(00 + 01) + (01 * 0 + 1)1 \\
 &= (01 * 0 + 1)[(00 + 01) + 1] \\
 &= (01 * 0 + 1)(00 + 01 + 1)
 \end{aligned}$$

After the simplification, the automaton looks as follows:

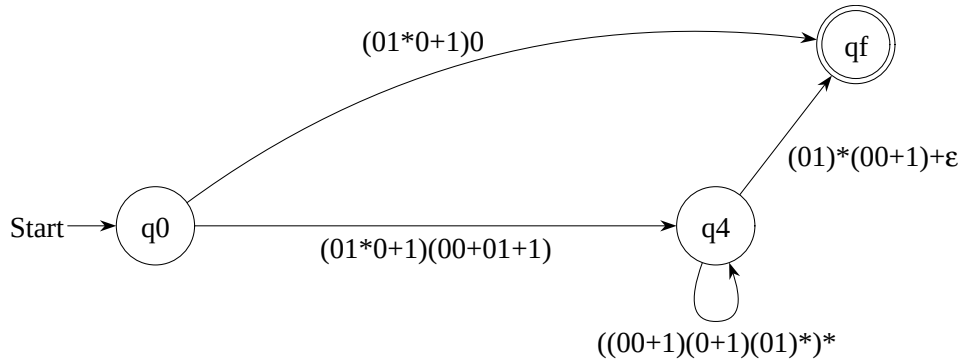


Fig. 3.24: An equivalent generalized automaton after simplifying the edge (q_0, q_4)

Finally, eliminate state q_4 :

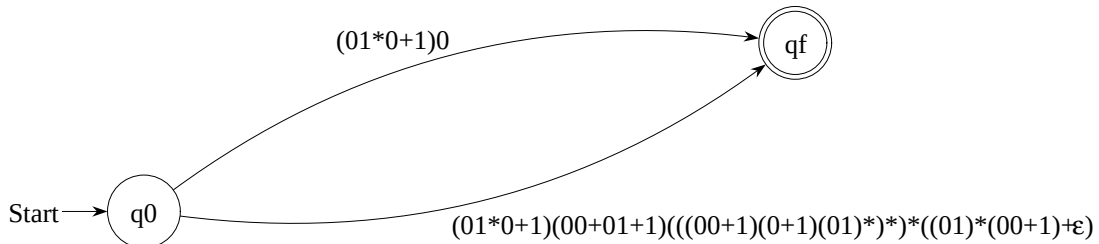


Fig. 3.25: An equivalent generalized automaton after eliminating state q_4

Final result:

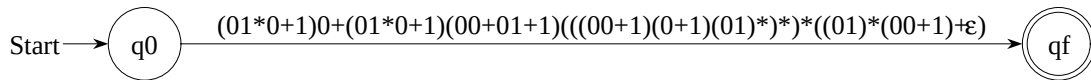


Fig. 3.26: Final equivalent generalized automaton

➤ **Remark:** In the example above, we did not bother to add state q_s to the automaton, because state q_0 already satisfies the conditions for a generalized automaton. In similar cases, this saves us some effort. The same remark applies to the accepting state: if the original automaton has

only one accepting state, with no edges leaving it, there is no point in adding state q_f .

- As can be seen, the regular expression produced by our algorithm is not particularly simple. In many cases, there is reason to suppose that a simpler equivalent regular expression exists. The question of how to find the simplest regular expression for a language L will arise later, together with several other similar questions.

3.3 Properties of Regular Languages

- A quotation from a previous lecture:

“The fact that all the models of computation we have encountered so far have proved equivalent and define the same class of languages – the regular languages – indicates that this class is natural in a certain sense and deserves special attention”.

- At this stage, we examine properties that characterize the regular languages independently of the various models of computation that generate these languages.
- To begin, we can ask several questions about the class of regular languages:
 - a. If L is a given language, is L regular? Do nonregular languages exist? If so, is there a reasonable way to prove that a given language is not regular?
 - b. If L is a regular language, is the complement language $\overline{L}$ also regular?
 - c. If L_1, L_2 are regular languages, is their union $L_1 \cup L_2$ also a regular language? Is the intersection $L_1 \cap L_2$ a regular language? Is the concatenation $L_1 L_2$ a regular language? And so on’
 - d. As we saw in the preceding lessons, there are many ways to define (or generate) regular languages: by means of various kinds of automata, regular expressions, and, later, we will also learn about context-free grammars.
 - e. Is there a method (an algorithm) for determining whether two different definitions define the same language?
- There are other questions that we will encounter later.
- We begin with the first question. The answer to this question, and to several others, follows from an important mathematical theorem in computability theory, commonly called the “**pumping theorem**” or the “**pumping lemma**”.
- The pumping lemma is also useful for questions such as: is the language L defined by an automaton (of any kind) finite? Infinite? Empty? Equal to Σ^* ?
- We will usually use the pumping lemma, however, to prove that certain languages are not regular.



The Pumping Lemma for Regular Languages

i Theorem 3.5: Let L be a regular language. Then there is a natural number n such that for every string w in the language L , if $|w| \geq n$, then w can be decomposed into three strings $w = xyz$ in such a way that $|xy| \leq n$, $|y| \geq 1$, and for every natural number i ,

$$xy^iz \in L$$

That is, the string y can be pumped an unlimited number of times, and the resulting string will always remain in the language L .

- During the proof, we will discover that the number n appearing in the theorem is in fact the number of states in a deterministic automaton that accepts the language L . Since every regular language in fact has infinitely many distinct automata defining it, n can take infinitely many different values.
- If M is the smallest automaton accepting L , then the number of states of M is the smallest possible value of n . This number is called the **critical length** of L .
- As stated above, the pumping lemma is used mainly to prove that certain languages are not regular.
- A language L that does not satisfy the conclusion of the theorem is necessarily a nonregular language.
- To prove that a language L is regular, we must use one of the usual methods: construct an automaton that accepts L , or construct a regular expression that generates L .
- The pumping lemma cannot be used to prove that a language L is regular.
A language L that satisfies the conclusion of the theorem is not necessarily regular!
- The assertion of the pumping lemma is logically complex – it contains three quantifiers.
Only after a sufficient number of examples and exercises will it be possible to grasp the assertion properly.
- We restate the pumping lemma in logical form:
If L is a regular language, then

$$\begin{aligned} \exists n \forall w \in L \exists x, y, z \in \Sigma^* : \\ w = xyz \wedge |xy| \leq n \wedge |y| \geq 1 \wedge \forall i [xy^iz \in L] \end{aligned}$$

- If

$$\begin{aligned} \forall n \exists w \in L \forall x, y, z \in \Sigma^* : \\ (w = xyz \wedge |xy| \leq n \wedge |y| \geq 1) \implies \exists i [xy^iz \notin L] \end{aligned}$$

then L is not a regular language.

- Therefore, a direct way to prove that a language L is not regular is to prove that



- a. for every natural number n
 - b. for every string $w \in L$ of length n
 - c. for every decomposition $w = xyz$ satisfying $|xy| \geq n$ and $|y| \geq 1$
 - d. there exists a natural number i such that $xy^iz \notin L$.
- This is usually done by a proof by contradiction. We assume that L is a regular language, apply the pumping lemma, and obtain a logical contradiction.
- ◆ **Example 3.7:** Let L be the language of all strings over the alphabet $\Sigma = \{a\}$ whose length is 2^m . That is,

$$L = \{a, a^2, a^4, a^8, \dots\} = \{a, aa, aaaa, aaaaaaaaa, \dots\}$$

Then L is not a regular language.

Proof: Suppose, by way of contradiction, that L is a regular language. Let n be the number provided by the pumping lemma, and let $w = a^{2^n}$ be the string whose length is 2^n . By the pumping lemma, there is a decomposition

$$w = xyz, \quad |xy| \leq n, \quad |y| \geq 1$$

such that for every natural number i , $xy^iz \in L$.

Denote $m = |y|$. Then $1 \leq m \leq n < 2^n$.

If, for example, we take $i = 2$, then on the one hand $xy^2z \in L$. But on the other hand,

$$|xy^2z| = |xyyz| = |xyz| + |y| = 2^n + m < 2^n + 2^n = 2^{n+1}$$

We have obtained a string $xy^2z \in L$ satisfying

$$2^n < |xy^2z| < 2^{n+1}$$

contrary to the definition of L . ■

- ◆ **Example 3.8:** The language $L = \{0^k1^k : k = 0, 1, 2, \dots\}$ is not regular.

Proof: Suppose, by way of contradiction, that L is regular. Let n be the pumping length for L . Consider the string $w = 0^n1^n$. This string has length greater than n , and therefore, by the pumping lemma, there is a decomposition

$$0^n1^n = xyz, \quad |xy| \leq n, \quad |y| \geq 1$$

such that for every i , $xy^iz \in L$.

There are three possibilities for y

- a. y is a substring of 0^n .

In this case, it is clear that in the string xy^2z the number of zeros is greater than the number

of ones, and therefore $xy^2z \notin L$.

- b. y is a substring of 1^n .

This possibility is ruled out for a similar reason.

- c. y is a string of the form 0^i1^j , $i, j > 0$.

In this case,

$$xy^2z = 0^{n-i}0^i1^j0^i1^j1^{n-j} = 0^n1^j0^i1^n \notin L$$

Proof of the Pumping Lemma

i Definition 3.6: A **loop** in a deterministic automaton M is any computation sequence that begins and ends in the same state q . All the other states in the computation sequence must be distinct from one another and from q .

$$q \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} p_{k-1} \xrightarrow{a_k} q$$

- Every loop corresponds to a string $y = a_1a_2 \cdots a_k$ labeling the edges of the loop.
- **Auxiliary claim:** Let M be a deterministic automaton containing the loop

$$q \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} p_{k-1} \xrightarrow{a_k} q$$

and let

- a. $y = a_1a_2 \cdots a_k$ be the string labeling the edges of the loop.
- b. x be a string for which there is a computation sequence beginning at q_0 and ending at q .
- c. z be a string for which there is a computation sequence beginning at q and ending at f , where $f \in F$.

Then for every natural number i , the string xy^iz is accepted by the automaton M .

Proof: We summarize all the data concisely as

$$q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{z} f$$

It is immediately apparent that this is a computation sequence proving that $xyz \in L$. Similarly, it is easy to see that $xy^iz \in L$ for every natural number i . For example, a computation sequence for xy^3z is

$$q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{y} q \xrightarrow{y} q \xrightarrow{z} f$$

- **Pumping lemma:** Let L be a regular language. Then there is a natural number n such that for every string w in the language L , if $|w| \geq n$, then w can be decomposed into three strings

$w = xyz$ in such a way that $|xy| \leq n$, $|y| \geq 1$, and for every natural number i ,

$$xy^iz \in L$$

✔ **Proof:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic automaton accepting the language L . Let n be the number of states of M ($n = |Q|$). We prove that the number n is the number guaranteed by the pumping lemma.

Let w be a string in the language L . Suppose that $w = a_1a_2a_3 \cdots a_m$, $m \geq n$, and that the computation sequence of w is

$$q_0 = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots p_{m-1} \xrightarrow{a_m} f = p_m$$

where $f = p_m$ is an accepting state of M . The string w consists of at least n symbols, so the computation sequence of w contains at least $n + 1$ states. The set Q contains exactly n states, so the computation sequence of w must contain at least one state q that occurs twice

$$q_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots p_{j-1} = q \xrightarrow{a_j} p_j \cdots p_{k-1} \xrightarrow{a_k} p_k = q \cdots p_{m-1} \xrightarrow{a_m} f$$

Note that q may be q_0 or f , and therefore this description of the computation sequence is not completely general.

The computation sequence of w may contain several states that repeat. We assume that q is the first one. Therefore, necessarily $j \leq n$.

Moreover, q may occur more than twice. We consider only the first two occurrences of q . Under this assumption, $k \leq n$. Indeed, the sequence of states $(q_0, p_1, \dots, p_n)$ contains $n + 1$ states. Thus, this sequence of states must already contain two identical states $p_j = p_k = q$, $k \leq n$.

We have therefore obtained the following loop in the automaton M :

$$q \xrightarrow{a_j} p_j \cdots p_{k-1} \xrightarrow{a_k} q$$

Denote

$$\begin{aligned} x &= a_1 \cdots a_{j-1} \\ y &= a_j \cdots a_k \\ z &= a_{k+1} \cdots a_m \end{aligned}$$

Then $|xy| = |a_1 \cdots a_k| \leq n$, $|y| \geq 1$. By the auxiliary claim, $xy^iz \in L$ for every natural number i . ■

➤ **Remarks:**

- a. If $q = q_0$, then $x = \varepsilon$.
- b. If $k = m = n$, then $z = \varepsilon$.
- c. An automaton need not contain a loop. If an automaton M contains no loop, then the



language accepted by M is necessarily finite. It can be proved that, in this case, the length of every string accepted by M does not exceed the number of states of M .

Closure Properties of Regular Languages

i Theorem 3.6: If L_1, L_2 are regular languages, then

- $L_1 \cup L_2$ is a regular language.
- $L_1 L_2$ is a regular language.
- L_1^* is a regular language.

Proof: Let r_1, r_2 be regular expressions generating the languages L_1, L_2 , respectively. Then the regular expression $r_1 + r_2$ generates the language $L_1 \cup L_2$. We proved that every language generated by a regular expression is regular, and therefore $L_1 \cup L_2$ is a regular language. Similarly, we prove that $L_1 L_2, L_1^*$ are regular languages. ■

Note that this theorem was in fact already proved during the proof of the equivalence theorem between regular expressions and nondeterministic automata with ϵ -transitions. As part of that proof, we constructed automata accepting the languages $L_1 \cup L_2, L_1 L_2, L_1^*$.

We present the corresponding diagrams once again as a reminder: M_1, M_2 are normal automata generating the languages L_1, L_2

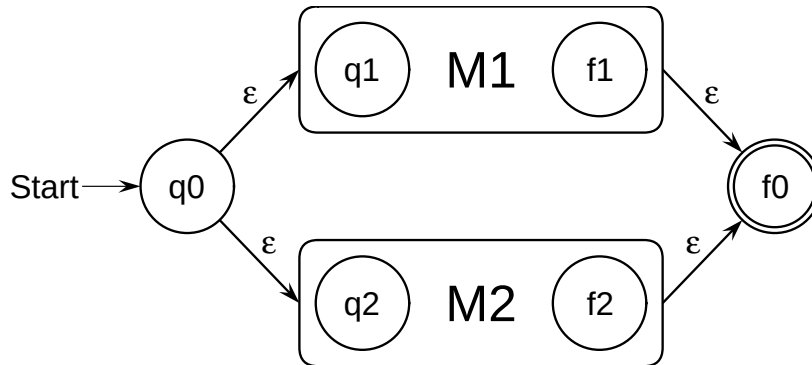


Fig. 3.27: A normal automaton for the language $L_1 \cup L_2$

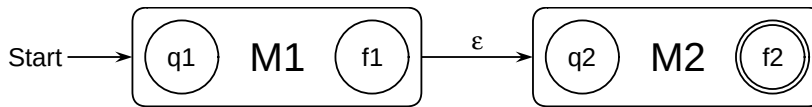


Fig. 3.28: A normal automaton for the language $L_1 L_2$

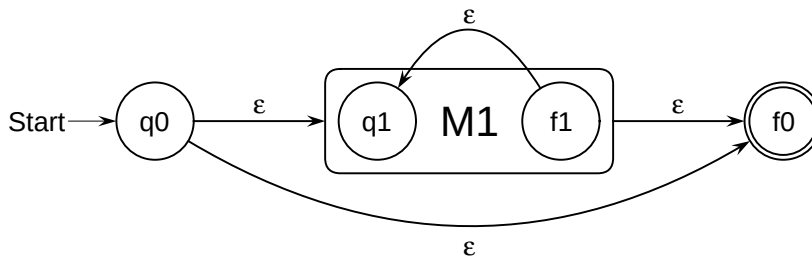


Fig. 3.29: A normal automaton for the language L_1^*

Theorem 3.7: If L is a regular language over the alphabet Σ , then its complement $\bar{L} = \Sigma^* - L$ is also regular.

Proof: Let

$$M = (Q, \Sigma, \delta, q_0, F)$$

be an ordinary deterministic automaton defining the language L . Define the automaton

$$M' = (Q, \Sigma, \delta, q_0, Q - F)$$

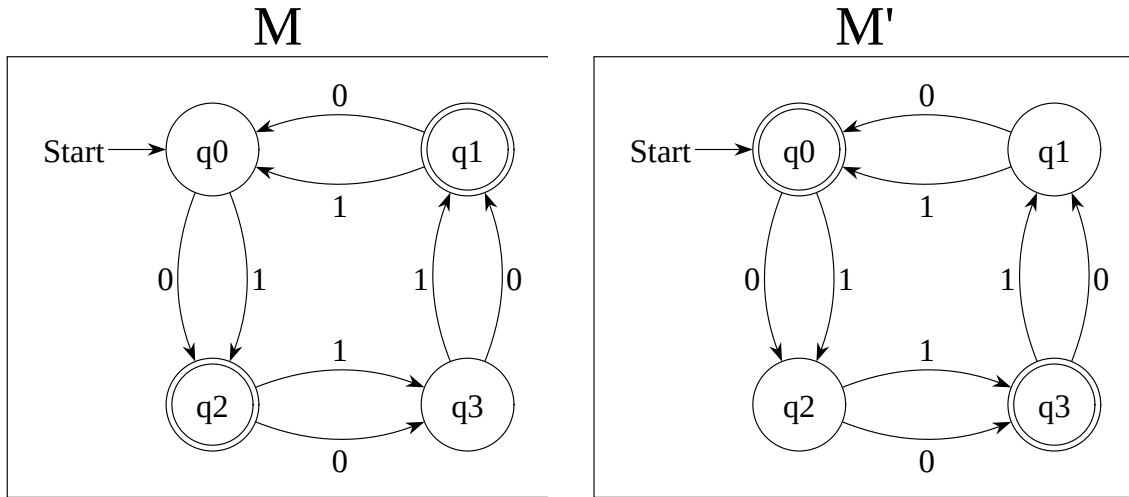
The automaton M' is identical to the automaton M except for its set of accepting states $Q - F$ (the set of states that do not belong to F).

By definition, M' accepts a string w if and only if $\delta(q_0, w) \in Q - F$, if and only if $\delta(q_0, w) \notin F$. We have therefore obtained

$$w \in L(M') \Leftrightarrow w \notin L(M)$$

Thus, the automaton M' accepts the language $\bar{L}$. ■

Example 3.9:



איור 3.30: egaugnal tneelpmoc eht rof notamotua nA

It is not difficult to see that

$$L(M) = \{ w \in \Sigma^* : |w| \text{ is odd} \}$$

$$L(M') = \{ w \in \Sigma^* : |w| \text{ is even} \}$$

i Theorem 3.8: If L_1, L_2 are regular languages, then their intersection $L_1 \cap L_2$ is also regular.

✓ **Proof:**

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

■

➤ Another proof: Let

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$

$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$$

be two deterministic automata accepting L_1, L_2 , respectively. The **Cartesian product** of the two automata M_1, M_2 is defined by

$$M = M_1 \times M_2 = (Q_1 \times Q_2, \Sigma, \delta, [q_1, q_2], F_1 \times F_2)$$

For every $p_1 \in Q_1, p_2 \in Q_2, a \in \Sigma$,

$$\delta([p_1, p_2], a) = [\delta_1(p_1, a), \delta_2(p_2, a)]$$

It is not difficult to prove that $L(M_1 \times M_2) = L_1 \cap L_2$.

■

Substitutions

➤ Let Σ be a given alphabet, and let R be a set of regular expressions over an alphabet Δ . A **substitution** is a function

$$f : \Sigma \rightarrow R$$

Each symbol $a \in \Sigma$ corresponds to a regular expression $r_a = f(a)$. In other words, the function f maps every symbol $a \in \Sigma$ to a regular language described by the regular expression $r_a = f(a)$.

➤ If L is a given language over the alphabet Σ , we can define a new language $f[L]$ by applying the substitution f to every string in L . For every string $w \in L$, if $w = \varepsilon$, then $f(\varepsilon) = \varepsilon$; if $w = a_1 a_2 \cdots a_n$, then

$$f(a_1 a_2 \cdots a_n) = f(a_1) f(a_2) \cdots f(a_n)$$

Since each of the values $f(a_1), f(a_2), \dots, f(a_n)$ is a regular expression, $f(w)$ is a longer regular expression – the concatenation of all the preceding regular expressions.

The language $f[L]$ is defined by

$$f[L] = \bigcup_{w \in L} L(f(w))$$



❖ **Example 3.10:** Suppose that $\Sigma = \{0, 1\}$, $\Delta = \{a, b, c\}$. Define a substitution f by

$$\begin{aligned} f(0) &= a^*b \\ f(1) &= caa \end{aligned}$$

Here are several substitutions of strings from Σ^* :

$$\begin{aligned} f(00) &= a^*ba^*b \\ f(101) &= caaa^*bcaa \\ f(0101) &= a^*bcaaa^*bcaa \end{aligned}$$

We can also (in relatively simple cases) compute $f[L]$ when L is a language over the alphabet Σ . For example, take

$$L = \{ (01)^n : n = 0, 1, 2, \dots \}$$

It is not difficult to see that $f[L]$ is the language generated by the regular expression $(a^*bcaa)^*$.

❗ **Theorem 3.9:** The class of regular languages is closed under substitutions. That is, if L is a regular language and f is a substitution, then $f[L]$ is also regular.

✔ **Proof:** Let r be a regular expression generating the language L . Replace every occurrence of a symbol a by the regular expression $f(a)$. The result is a regular expression generating the language $f[L]$. ■

❖ **Second proof:** Let

$$M = (Q, \Sigma, \delta, q_0, F)$$

be a deterministic automaton accepting the language L . Define a generalized NFA

$$M' = (Q, \Delta, \delta', q_0, F)$$

with the same state set Q , the same start state q_0 , and the same set of final states F . The only difference lies in the symbol set Δ and the transition function

$$\delta'(q, f(a)) = \delta(q, a)$$

δ' operates on regular expressions over the alphabet Δ , since for every $a \in \Sigma$, $f(a)$ is a regular expression over the alphabet Δ .

❖ **Example 3.11:** Suppose that $\Sigma = \{0, 1\}$, $\Delta = \{a, b, c\}$. Define a substitution f by

$$\begin{aligned} f(0) &= a^*b \\ f(1) &= caa \end{aligned}$$

Let $L = \{01^i01^j01^k : i, j, k = 0, 1, 2, \dots\}$. It is easy to find that L is generated by the regular expression $01^*01^*01^*$, and therefore, by the algorithm in the first proof,

$$f[L] = a^*b(caa)^*a^*b(caa)^*a^*b(caa)^*$$

It is easy to construct a deterministic automaton M for L and construct M' for $f[L]$ as in the second proof. See Figure 3.31.

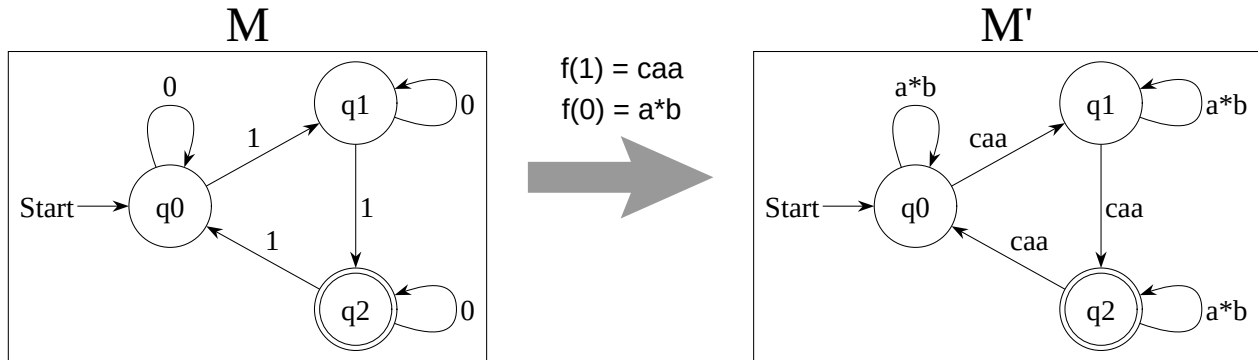


Fig. 3.31: Conversion of the language L into the language $f[L]$

Homomorphism

- A **homomorphism** is a function h that maps every symbol a in some alphabet Σ to a string $h(a)$ over an alphabet Δ (possibly $\Delta = \Sigma$). Unlike an ordinary substitution, $h(a)$ must be an ordinary word and not a regular expression.
- The natural extension of h to every string $w \in \Sigma^*$ is as follows:
if $w = a_1a_2 \cdots a_n \in \Sigma^*$, then

$$h(w) = h(a_1)h(a_2) \cdots h(a_n)$$

and, of course, $h(\varepsilon) = \varepsilon$. Note again that the result $h(w)$ is a string over the alphabet Δ .

- Let L be a language over the alphabet Σ . Applying h to every string in L gives a new language L' over the alphabet Δ :

$$L' = h[L] = \{h(w) : w \in L\}$$

- **Inverse image of a homomorphism:** If L' is a language over the alphabet Δ , then the inverse image of L' under h is a language L over the alphabet Σ defined by

$$L = h^{-1}[L'] = \{w \in \Sigma^* : h(w) \in L'\}$$

❖ **Example 3.12:** Let $\Sigma = \{0, 1\}$, $\Delta = \{a, b, c\}$, and let h be the homomorphism

$$\begin{aligned} h(0) &= aa \\ h(1) &= cab \end{aligned}$$

Then

$$\begin{aligned} h(000) &= aaaaaa \\ h(1001) &= cabaaaaaab \\ h(1101) &= cabcabaaab \end{aligned}$$

a. If L is the language $01^*0 + 10^*$, then

$$h[L] = L(aa(cab)^*aa + cab(aa)^*)$$

b. If L' is the language ba^* , then $h^{-1}[L'] = \emptyset$. This is because for every $w \in \{0, 1\}^*$, $h(w) \notin L'$.

c. If L' is the language $c^*(a + b)^*$, then $h^{-1}[L']$ is the language $10^* + 0^*$ (why?).

i Theorem 3.10: The class of regular languages is closed under homomorphisms and inverse images of homomorphisms. That is:

- a. If L is a regular language over the alphabet Σ , then $h[L]$ is also a regular language (over the alphabet Δ).
- b. If L' is a regular language over the alphabet Δ , then $h^{-1}[L']$ is also a regular language (over the alphabet Σ).

✓ **Proof:** Assertion a' is an immediate consequence of the corresponding theorem on substitutions. We therefore prove part b' only. Let

$$M' = (Q, \Delta, \delta', q_0, F)$$

be a deterministic automaton accepting the language L' . We construct a deterministic automaton

$$M = (Q, \Sigma, \delta, q_0, F)$$

that accepts the language $L = h^{-1}[L']$. The automaton M is identical to the automaton M' except for the alphabet Σ and the transition function. For every $a \in \Sigma$ and every $q \in Q$,

$$\delta(q, a) = \delta'(q, h(a))$$

Note that $h(a)$ may be a long string or ε , but the natural extension of δ' is defined for every

string over the alphabet Δ^* , so this definition is valid.

It is fairly routine (a homework exercise!) to prove by induction on $|w|$ that

$$\delta(q_0, w) = \delta'(q_0, h(w))$$

and therefore

$$w \in L(M) \Leftrightarrow h(w) \in L(M')$$

It follows that

$$L(M) = \{ w \in \Sigma^* : h(w) \in L(M') \} = h^{-1}[L(M')]$$

■

✦ **Example 3.13:** $\Sigma = \{0, 1\}$, $\Delta = \{a, b, c\}$,

$$\begin{aligned} h(0) &= aa \\ h(1) &= cab \end{aligned}$$

Let L' be the language $c^*(a+b)^*$. In a previous part, we asserted that $h^{-1}[L']$ is the language $10^* + 0^*$. We prove this assertion using the method presented in the proof of the last theorem. To do so, we must find a deterministic automaton for the language L' . The task is not difficult.

✦ We computed the automaton M using the formula

$$\delta(q, a) = \delta'(q, h(a))$$

Here are all the results explicitly

$$\begin{aligned} \delta(q_0, 0) &= \delta'(q_0, h(0)) = \delta'(q_0, aa) = q_1 \\ \delta(q_0, 1) &= \delta'(q_0, h(1)) = \delta'(q_0, cab) = q_3 \\ \delta(q_1, 0) &= \delta'(q_1, h(0)) = \delta'(q_1, aa) = q_1 \\ \delta(q_1, 1) &= \delta'(q_1, h(1)) = \delta'(q_1, cab) = q_3 \\ \delta(q_2, 0) &= \delta'(q_2, h(0)) = \delta'(q_2, aa) = q_2 \\ \delta(q_2, 1) &= \delta'(q_2, h(1)) = \delta'(q_2, cab) = q_2 \\ \delta(q_3, 0) &= \delta'(q_3, h(0)) = \delta'(q_3, aa) = q_1 \\ \delta(q_3, 1) &= \delta'(q_3, h(1)) = \delta'(q_3, cab) = q_2 \end{aligned}$$

✦ Homomorphisms can be used to prove that certain languages are not regular.

✦ **Proposition 3.11:** The language $L' = \{ a^{2n}(cab)^n : n = 0, 1, 2, \dots \}$ is not regular.

Proof: Suppose, by way of contradiction, that L' is a regular language. Define a homomorphism



h as before

$$\begin{aligned} h(0) &= aa \\ h(1) &= cab \end{aligned}$$

It is not difficult to see that

$$L = h^{-1}[L'] = \{0^n 1^n : n = 0, 1, 2, \dots\}$$

Therefore, by the theorem above, the language L is also regular, contradicting the fact that it is not regular. ■

3.4 Decision Algorithms for Regular Languages

- Suppose we have a description of a regular language L (by a suitable automaton or by a regular expression). Can we determine from the description of L alone whether L is finite? Infinite? Empty? And other questions of this kind.
- Suppose we have two different descriptions of regular languages L_1, L_2 . Is there an algorithm for determining whether $L_1 = L_2$? $L_1 \subseteq L_2$? $L_1 \cap L_2 = \emptyset$?
- Suppose we have a description of a regular language L . Is there an algorithm for finding an automaton with a minimal number of states that accepts L ?
- Since every regular language can be described by infinitely many models of computation, we must decide which model is most suitable for formulating the algorithms. This decision is important for consistency, but principally because the complexity of the algorithms is closely related to the type of model. Although we generally have algorithms for translating one model of computation into another, it is entirely possible that some models of computation may greatly complicate the algorithms or even make them impossible.
- It turns out that the finite-automaton model (in its various forms) is sufficiently convenient for most purposes, so we choose it as our official model of computation.
- In light of this decision, we must formulate all our questions in terms of automata. For example, the question of whether a regular language L is finite or infinite must be translated into the following question: is there an algorithm that takes as input a description of an automaton M (defining the language L) and tells us whether M accepts only finitely many strings, or whether infinitely many distinct strings are accepted by M ?

An Algorithm for Deciding Whether $L(M) \neq \emptyset$

i Proposition 3.12: An automaton M defines a nonempty language if and only if M accepts at least one string from the set $U = \bigcup_{i=0}^n \Sigma^i$, where n is the number of states of M .

- Since the symbol set Σ is finite, the set $\bigcup_{i=0}^n \Sigma^i$ is also finite. This set has $1 + k + k^2 + \dots + k^n$



elements, where k is the number of symbols.

- Although this number may be very large, the number of required tests is finite, and at the end of the process a decision is obtained: if none of the strings in U is accepted by M , then no string is accepted by M .

- Proof of the claim:** If M accepts one of the strings in U , then certainly $L(M) \neq \emptyset$. It remains to prove that if $L(M) \neq \emptyset$, then M necessarily accepts a string in U .

Let w be a string of minimal length belonging to $L(M)$. We prove that necessarily $|w| \leq n$. If $|w| > n$, then by the pumping lemma $w = xyz$, $|y| \geq 1$. If we substitute $i = 0$ into the conclusion of the pumping lemma, we obtain that xz is a shorter string belonging to $L(M)$, contrary to the minimality of the length of w . ■

- An idea for a second proof: let w be as above. If $|w| > n$, then the computation sequence for w contains more than n states, so there is a state q that occurs twice in the computation sequence for w . It follows that the computation sequence for w must have the form

$$q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{z} f, \quad |y| \geq 1$$

It is then clear that the shorter string xz is also accepted by M .

An Algorithm for Deciding Whether $L(M)$ Is Infinite or Finite

i Proposition 3.13: An automaton M defines an infinite language if and only if M accepts at least one string of length m , $n \leq m < 2n$, where n is the number of states of M .

- In other words, the language $L(M)$ is infinite if and only if M accepts at least one string belonging to the set $V = \bigcup_{i=n}^{2n-1} \Sigma^i$.
- Clearly, the set V is finite, so testing all the strings in this set takes finite time and yields a decision.

- Proof of the claim:** Suppose that w is a string of length m , $m \geq n$, belonging to $L(M)$. By the pumping lemma, there is a decomposition $w = xyz$, $|y| \geq 1$, such that for every i , $xy^iz \in L(M)$. The fact that $|y| \geq 1$ guarantees that for every $i \neq j$, $xy^iz \neq xy^jz$, and hence $L(M)$ contains infinitely many distinct strings.

Conversely, suppose that $L(M)$ is an infinite language. Among all the strings in $L(M)$ whose lengths are greater than n , let w be one of minimal length. If $|w| < 2n$, the proof is complete. If $|w| \geq 2n$ (that is, the language $L(M)$ contains no strings of length between n and $2n-1$), then by the pumping lemma $w = xyz$, $|xy| \leq n$, $|y| \geq 1$. Therefore $1 \leq |y| \leq n$, and hence $n \leq |xz| \leq 2n-1$. We have already seen that the string xz also belongs to $L(M)$. We have obtained a contradiction. ■

- Note that the two algorithms presented are not the most efficient, but for now it is important to establish their existence.

- For example, a faster algorithm for determining whether $L(M) \neq \emptyset$ is: from the transition diagram of M , remove all states that cannot be reached from the start state q_0 . If at least one accepting state remains at the end of the process, then $L(M) \neq \emptyset$.
- A faster algorithm for determining whether $L(M)$ is empty, finite, or infinite is as follows:
 - a. Perform the process in the preceding algorithm.
 - b. If no accepting state remains, then $L(M) = \emptyset$, and the algorithm terminates with the answer “ $L(M)$ is finite”. If at least one accepting state remains, proceed to the next step.
 - c. Check whether M contains a loop. If there is no loop, the algorithm terminates with the answer “ $L(M)$ is finite”. If there is a loop, proceed to the next step.
 - d. Check whether there is a path from one of the states on the loop to an accepting state. If not, the algorithm terminates with the answer “ $L(M)$ is finite”. If so, the algorithm terminates with the answer “ $L(M)$ is infinite”.

An Algorithm for Deciding Whether $L(M_1) = L(M_2)$

- Recall that two automata M_1 and M_2 are called equivalent if they define the same language (that is, $L(M_1) = L(M_2)$). It is not easy to determine whether two automata define the same language merely by examining their transition diagrams. We use several of the preceding results to prove that the following algorithm is correct.
- **The algorithm**
 - a. Construct an automaton M_3 that accepts the language $L(M_1) \cap \overline{L(M_2)}$.
 - b. Construct an automaton M_4 that accepts the language $\overline{L(M_1)} \cap L(M_2)$.
 - c. Construct an automaton M_5 that accepts the language $L(M_3) \cup L(M_4)$.
 - d. Check whether $L(M_5) = \emptyset$. If so, then $L(M_1) = L(M_2)$. If not, then $L(M_1) \neq L(M_2)$.
- **Proof of correctness:** In the preceding lessons, we learned how to construct automata for the languages $\overline{L}$, $L_1 \cup L_2$, and $L_1 \cap L_2$, given automata for L , L_1 , L_2 . Thus, there is no difficulty in carrying out steps a–c. The question of whether $L(M_1) = L(M_2)$ is equivalent to the question of whether

$$L(M_1) - L(M_2) = \emptyset \quad \text{and} \quad L(M_2) - L(M_1) = \emptyset$$

By basic set-theoretic identities,

$$\begin{aligned} L(M_1) - L(M_2) &= L(M_1) \cap \overline{L(M_2)} \\ L(M_2) - L(M_1) &= L(M_2) \cap \overline{L(M_1)} \end{aligned}$$

Therefore, the question of whether $L(M_1) = L(M_2)$ is equivalent to the question of whether

$$(L(M_1) \cap \overline{L(M_2)}) \cup (L(M_2) \cap \overline{L(M_1)}) = \emptyset$$

■

- For now, the more difficult question remains unanswered: suppose we have a description of a regular language L . Is there an algorithm for finding an automaton with a minimal number of states that accepts L ?
- To answer this question, we need one of the deeper theorems in automata theory, the Myhill-Nerode theorem.

3.5 The Myhill-Nerode Theorem

- The discussion in this section is more abstract and difficult than usual because it concerns a more complex result (and a very important one) than those proved so far. The proof of the Myhill-Nerode theorem relies on the theory of equivalence relations and equivalence classes, without which the statement of the theorem itself and its proof would not be clear. We briefly review the essentials of the theory of equivalence relations.

i **Definition 3.7:** A relation R defined on the elements of a set A is called an **equivalence relation** if

- a. R is a **reflexive** relation

$$\forall x \in A : xRx$$

- b. R is a **symmetric** relation

$$\forall x, y \in A : xRy \Rightarrow yRx$$

- c. R is a **transitive** relation

$$\forall x, y, z \in A : (xRy \wedge yRz) \Rightarrow xRz$$

- Every equivalence relation on a set A defines a natural partition of A into disjoint subsets called **equivalence classes**. An equivalence class is any maximal subset of A such that every two of its elements satisfy the relation R .

i **Definition 3.8:** Let R be an equivalence relation on a set A . For every element $x \in A$, the **equivalence class** of x is a subset of A denoted by the expression $[x]$ and defined by

$$[x] = \{ y \in A : xRy \}$$



- It is not difficult to prove the following fundamental facts:
- a. For every $x_1, x_2 \in A$, if $x_1 R x_2$, then $[x_1] = [x_2]$. That is, the equivalence class of x_1 is identical to the equivalence class of x_2 .
 - b. For every $x_1, x_2 \in A$, if $\neg x_1 R x_2$, then $[x_1] \cap [x_2] = \emptyset$. That is, the equivalence class of x_1 is disjoint from the equivalence class of x_2 .
 - c. Every element of A belongs to exactly one equivalence class ($x \in [x]$).
 - d. If $E_1, E_2, \dots, E_n$ are all the equivalence classes, then

$$A = E_1 \cup E_2 \cup \dots \cup E_n$$

i **Definition 3.9:** The **index** of an equivalence relation R is a natural number denoting the number of equivalence classes of R on A , if this number is finite. Otherwise, we say that the relation R has infinite index.

- ◆ **Example 3.14:** If $A = \{0, 1, 2, \dots\}$ is the set of nonnegative integers, and R is the relation defined by

$$xRy \Leftrightarrow x - y \text{ is divisible by } 3$$

then it is very easy to prove that R is an equivalence relation on A with index **3**. The equivalence classes of R are

$$\begin{aligned} E_0 &= \{0, 3, 6, 9, 12, \dots\} \\ E_1 &= \{1, 4, 7, 10, 13, \dots\} \\ E_2 &= \{2, 5, 8, 11, 14, \dots\} \end{aligned}$$

i **Definition 3.10:** An equivalence relation R' is called a **refinement** of an equivalence relation R if every equivalence class of R' is contained in one of the equivalence classes of R .

- ◆ **Example 3.15:** If $A = \{0, 1, 2, \dots\}$, and R' is the relation defined by

$$xRy \Leftrightarrow x - y \text{ is divisible by } 6$$

then R' is an equivalence relation on A with index **6**. The equivalence classes of R are

$$\begin{aligned} E'_0 &= \{0, 6, 12, 18, 24, \dots\} \\ E'_1 &= \{1, 7, 13, 19, 25, \dots\} \\ E'_2 &= \{2, 8, 14, 20, 26, \dots\} \\ E'_3 &= \{3, 9, 15, 21, 27, \dots\} \\ E'_4 &= \{4, 10, 16, 22, 28, \dots\} \\ E'_5 &= \{5, 11, 17, 23, 29, \dots\} \end{aligned}$$



and it is immediately apparent that R' is a refinement of R from the preceding example.

i Proposition 3.14: If R' is a refinement of R , then the index of R' is greater than or equal to the index of R .

i Proposition 3.15: Let R, R' be equivalence relations. Then R' is a refinement of R if and only if

$$\forall x, y \in A [xR'y \Rightarrow xRy]$$

➤ Every finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ has a very natural equivalence relation defined on the set of all strings Σ^* . For every two strings $x, y \in \Sigma^*$,

$$xRy \Leftrightarrow \delta(q_0, x) = \delta(q_0, y)$$

That is, xRy if and only if the computation sequences for x and y beginning in state q_0 end in the same state. Note that this fact does not necessarily imply that $x = y$. It is very easy to prove that R is an equivalence relation on Σ^* .

i Proposition 3.16: The relation R has exactly n equivalence classes, where n is the number of states in the automaton M . Therefore, R is always an equivalence relation of finite index.

➤ That is, the number of equivalence classes of R equals the number of states of M . Every state q in the automaton M corresponds to an equivalence class

$$E_q = \{ x \in \Sigma^* : \delta(q_0, x) = q \}$$

➤ The language $L(M)$ accepted by the automaton M is in fact the union of the equivalence classes corresponding to the accepting states of M .

➤ That is, if $f_1, f_2, \dots, f_k$ are all the accepting states of M , then

$$L(M) = E_{f_1} \cup E_{f_2} \cup \dots \cup E_{f_k}$$

➤ **Remark:** We assume that the automaton is one in which every state can be reached from state q_0 . If there is a state q that cannot be reached from state q_0 , then q (and all the edges connected to it) can be removed without changing $L(M)$.

➤ In addition to being an equivalence relation, R has another important property

$$\forall x, y, z \in \Sigma^* : xRy \Rightarrow xzRyz$$

A relation R with this property is called a **right-invariant relation**.

- This property of R is easy to prove. By a previous exercise,

$$\delta(q_0, xz) = \delta(\delta(q_0, x), z)$$

If we are given that xRy , then we conclude that

$$\delta(q_0, x) = \delta(q_0, y)$$

Therefore,

$$\delta(\delta(q_0, x), z) = \delta(\delta(q_0, y), z)$$

and again, by the same exercise, we obtain

$$\delta(q_0, xz) = \delta(q_0, yz)$$

- In summary, we proved that every automaton M naturally defines a right-invariant equivalence relation R . But equivalence relations (even if they are right-invariant) can also be defined in many other ways. We do not need automata to define such relations.
- The question arises: if we are given a right-invariant equivalence relation R on Σ^* , is there an automaton M that defines R ?
- The answer to this question is negative, because R may have infinite index, whereas every such R defined by an automaton must have finite index!
- We must therefore reformulate the question: if we are given a right-invariant equivalence relation R on Σ^* whose index is finite, is there an automaton M that defines R ?
- This question is interesting because R can be defined in a way entirely unrelated to automata theory. For example, define the relation R on the set $\{0, 1\}^*$ by:

$$xRy \Leftrightarrow xy \text{ is a string with an even number of zeros}$$

For example, if $x = 110100$, $y = 0100111001$, then xRy .

It is not difficult to prove that R is a right-invariant equivalence relation, but it is not clear whether there is an automaton M that generates R as described above.

- The surprising answer to this question is affirmative: every right-invariant equivalence relation of finite index can be defined by a suitable automaton. This assertion is only part of the Myhill–Nerode theorem; before presenting the full theorem, we need one more definition.
- Similarly, we can define a relation R_L for every language L . Let L be a language over the alphabet Σ . For every $x, y \in \Sigma^*$, define

$$xR_Ly \Leftrightarrow \forall z \in \Sigma^* [xz \in L \iff yz \in L]$$



In other words, the relation xR_Ly holds if and only if for every $z \in \Sigma^*$, either $xz, yz \in L$ or $xz, yz \notin L$.

❖ **Example 3.16:** Let L be the language $(10)^*1^*$.

- (a) If $x = 100$, $y = 1101$, then xR_Ly .
- (b) If $x = 101$, $y = 1111$, then $\neg xR_Ly$ (for example, check $z = 010$).
- (c) If $x = 111$, $y = 1111$, then xR_Ly .

❗ **Proposition 3.17:** R_L is a right-invariant equivalence relation.

✔ **Proof:** Reflexivity and symmetry follow immediately from the definition of R_L .

Transitivity: suppose that uR_Lv and vR_Lw . We prove that uR_Lw . By the definition of R_L , the assumptions are

$$\forall z \in \Sigma^* [uz \in L \Leftrightarrow vz \in L]$$

$$\forall z \in \Sigma^* [vz \in L \Leftrightarrow wz \in L]$$

and they immediately imply that

$$\forall z \in \Sigma^* [uz \in L \Leftrightarrow wz \in L]$$

that is, uR_Lw .

Right invariance: We must prove

$$xR_Ly \Rightarrow xzR_Lyz$$

Suppose that xR_Ly . We must prove that for every z , xzR_Lyz . The last assertion means that for every w ,

$$xzw \in L \Leftrightarrow yzw \in L$$

and this follows immediately from the assumption xR_Ly . ■

❗ **Theorem 3.18: (Myhill–Nerode)** For a language L over an alphabet Σ , the following three assertions are equivalent

- a. L is accepted by a finite automaton.
- b. L is a union of some equivalence classes of a right-invariant equivalence relation of finite index.
- c. The relation R_L has finite index.

✔ **Proof:** To prove that the three assertions are equivalent, it suffices to prove the following three implications

$$a \Rightarrow b, \quad b \Rightarrow c, \quad c \Rightarrow a$$

$b \Leftarrow a$: This assertion was already proved above.

$c \Leftarrow b$: Suppose that L is a union of several equivalence classes of a right-invariant equivalence relation R' of finite index.

We must prove that R_L is a right-invariant equivalence relation of finite index.

The fact that R_L is a right-invariant equivalence relation is true for every language L , as proved above. It remains only to prove that R_L has finite index. For this purpose, we prove that R' is a refinement of R_L . By a previous assertion, it suffices to prove that

$$\forall x, y \in \Sigma^* [xR'y \Rightarrow xR_Ly]$$

Suppose that $xR'y$. The fact that R' is right-invariant implies

$$\forall z \in \Sigma^* [xzR'yz]$$

and therefore

$$xz \in L \Leftrightarrow yz \in L$$

and therefore xR_Ly .

a $\Leftarrow$ c: Suppose that the relation R_L has finite index. That is, the relation R_L has a finite number of equivalence relations. Let

$$E_1, E_2, \dots, E_n$$

be all the equivalence classes of R_L .

We must prove that there is an automaton M accepting the language L . We begin by defining the state set of M

$$Q = \{q_1, q_2, \dots, q_n\}$$

That is, each equivalence class E_i , $1 \leq i \leq n$, corresponds to a state q_i .

Now define the transition function δ . Let q_i be a given state and a a given symbol. From the corresponding equivalence class E_i , choose an arbitrary representative $x \in E_i$. The string xa belongs to exactly one equivalence class E_j . Define $\delta(q_i, a) = q_j$. In short, the definition of δ is:

$$\delta(q_i, a) = q_j \Leftrightarrow \exists x \in E_i [xa \in E_j]$$

It is not difficult to see that the definition of δ does not depend on the choice of the representative x .

Definition of the start state: there is exactly one equivalence class E_{i_0} that contains the empty string ε . Define q_{i_0} to be the start state of the automaton M .

Definition of the accepting states: Since R_L is an equivalence relation,

$$\Sigma^* = E_1 \cup E_2 \cup \dots \cup E_n$$

Therefore, not every equivalence class is necessarily contained in the language L (this occurs

only when $L = \Sigma^*$). It is not difficult to prove, using the invariance property of R_L , that

$$\begin{aligned} x \in E_i \wedge x \in L &\Rightarrow E_i \subseteq L \\ x \in E_i \wedge x \notin L &\Rightarrow E_i \cap L = \emptyset \end{aligned}$$

That is, for every equivalence class E_i ,

$$E_i \subseteq L \quad \text{or} \quad E_i \cap L = \emptyset$$

We can now define the set of accepting states of the automaton M :

$$F = \{ q_i : E_i \subseteq L \}$$

This completes the definition of the automaton M . It remains to prove that M accepts the language L . Let $w = a_1 a_2 \cdots a_k$ be any string over the alphabet Σ . We must prove that

$$\delta(q_{i_0}, w) \in F \Leftrightarrow w \in L$$

Let

$$q_{i_0} \xrightarrow{a_1} q_{i_1} \xrightarrow{a_2} q_{i_2} \cdots q_{i_{k-1}} \xrightarrow{a_k} q_{i_k}$$

be the computation sequence for w . By the definition of δ ,

$$a_{i_1} \in E_{i_1}, a_{i_1} a_{i_2} \in E_{i_2}, \dots, a_{i_1} a_{i_2} \cdots a_{i_k} \in E_{i_k}$$

If M accepts the string w , then $q_{i_k} \in F$, and therefore, by the definition of F , $E_{i_k} \subseteq L$, and we obtain $w \in L$. If M does not accept w , then similarly $w \notin L$. ■

◆ **Example 3.17:** Let L be the language accepted by the following automaton M

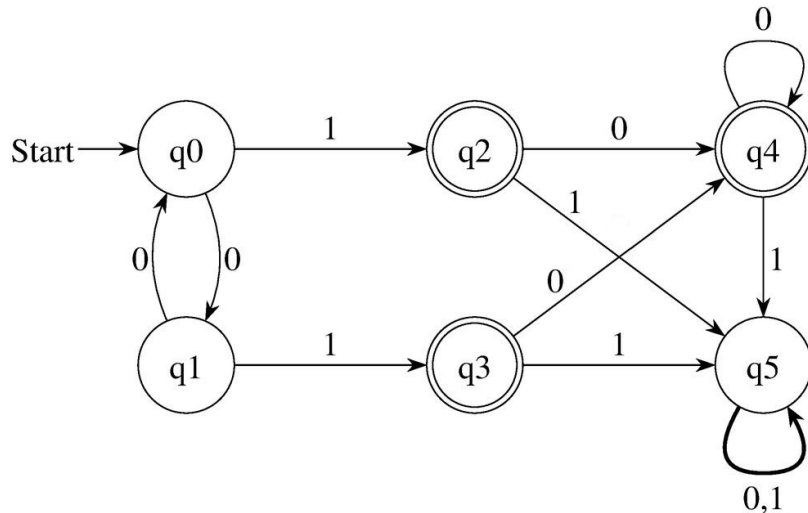


Fig. 3.32: A finite automaton M accepting the language L

Let R be the equivalence relation defined by M . Since all the states of M are reachable from the start state q_0 , the relation R has exactly 6 equivalence classes

$$\begin{aligned} E_0 &= (00)^* & E_1 &= (00)^*0 \\ E_2 &= (00)^*1 & E_3 &= (00)^*01 \\ E_4 &= 0^*100^* & E_5 &= 0^*10^*1(0+1)^* \end{aligned}$$

The accepting states of M are $\{q_2, q_3, q_4\}$, and therefore

$$L = E_2 \cup E_3 \cup E_4 = (00)^*1 + (00)^*01 + 0^*100^*$$

This is a rather lengthy description of L by a regular expression. Clearly, the smaller the number of equivalence classes, the shorter the regular expression.

- In the proof of the Myhill–Nerode theorem, we saw that the equivalence relation with the lowest suitable index is precisely the relation R_L , which we defined by

$$xR_Ly \Leftrightarrow \forall z \in \Sigma^*[xz \in L \iff yz \in L]$$

In the present case, one can find that the relation xR_Ly holds if and only if one of the following conditions holds:

- a. The strings x , y consist only of zeros.
- b. In the strings x , y , the symbol 1 occurs exactly once.
- c. In the strings x , y , the symbol 1 occurs more than once.

It follows that the relation R_L has exactly three equivalence classes

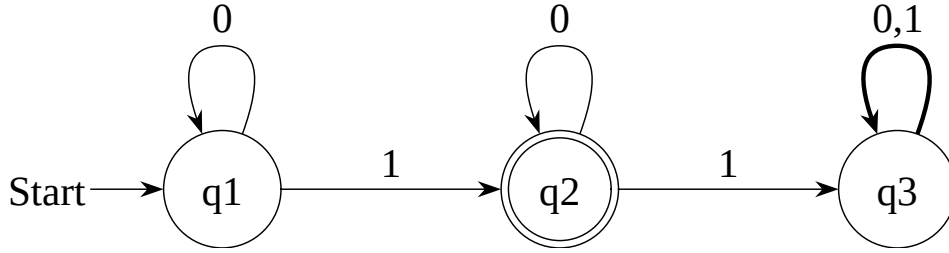
$$H_1 = 0^*, \quad H_2 = 0^*10^*, \quad H_3 = 0^*10^*1(0+1)^*$$

The language L itself consists only of the equivalence class H_2 (why?). That is,

$$L = H_2 = 0^*10^*$$

Now that we have the relation R_L of lowest index and its equivalence classes, we can use the algorithm in the Myhill–Nerode theorem to construct an automaton with a minimal number of states for L :

This automaton contains exactly three states – q_1 , q_2 , q_3 – corresponding to the three equivalence classes H_1 , H_2 , H_3 . The start state is q_1 because the empty string ε belongs to H_1 . The following table gives the transition function δ , with an explanation for each row.

Fig. 3.33: An automaton with a minimal number of states for the language L

$\delta(q_1, 0) = q_1$	$0 \in H_1$	$00 \in H_1$
$\delta(q_1, 1) = q_2$	$0 \in H_1$	$01 \in H_2$
$\delta(q_2, 0) = q_2$	$010 \in H_2$	$0100 \in H_2$
$\delta(q_2, 1) = q_3$	$010 \in H_2$	$0101 \in H_3$
$\delta(q_3, 0) = q_3$	$0101 \in H_3$	$01010 \in H_3$
$\delta(q_3, 1) = q_3$	$0101 \in H_3$	$01011 \in H_3$

3.6 An Algorithm for Finding a Minimal Automaton

Isomorphism Between Two Automata

i **Definition 3.11:** Two deterministic automata over the same alphabet Σ

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$M' = (Q', \Sigma, \delta', q'_0, F')$$

are called **isomorphic** if there is a one-to-one and onto function

$$f : Q \rightarrow Q'$$

such that

- $f(q_0) = q'_0$
- $(\forall q \in Q)[q \in F \Leftrightarrow f(q) \in F']$
- $(\forall q \in Q)(\forall a \in \Sigma)[\delta(q, a) = \delta'(f(q), a)]$

➤ Notation: $M \cong M'$.

➤ The matter is actually simpler. The automata M and M' are isomorphic if and only if their transition diagrams are identical in every respect, except possibly for the names of the states. Of course, the names of the states are of no importance. Thus, two isomorphic automata are really the same automaton in two different representations.

❖ **Example 3.18:**

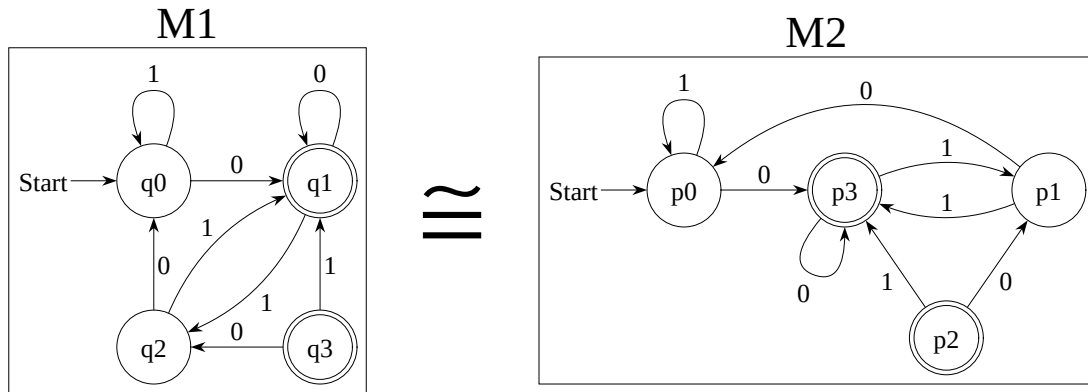


Fig. 3.34: Isomorphism between two automata

❖ The isomorphism function in this example

$$f : \{q_0, q_1, q_2, q_3\} \rightarrow \{p_0, p_1, p_2, p_3\}$$

is defined by

$$f(q_0) = p_0, \quad f(q_1) = p_3, \quad f(q_2) = p_1, \quad f(q_3) = p_2$$

❖ To prove that this function indeed defines an isomorphism, all the conditions in the definition must be checked. For example, in the automaton M , $F = \{q_1, q_3\}$. It must be verified that

$$F' = \{f(q_1), f(q_3)\} = \{p_3, p_2\}$$

It must also be verified that $\delta(q, a) = \delta'(f(q), a)$.

i Proposition 3.19: If $M \cong M'$, then $L(M) = L(M')$.

i Definition 3.12: A deterministic automaton M is called a **minimal automaton** if there is no deterministic automaton with fewer states that is equivalent to it.

❖ **Example 3.19:** The following automaton is minimal.

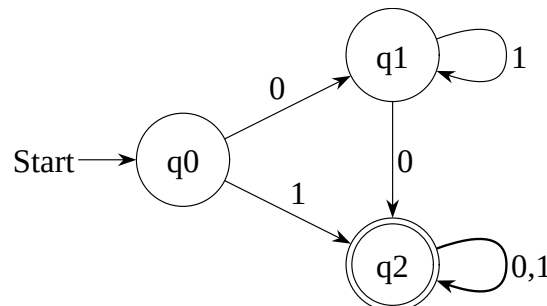


Fig. 3.35: A minimal automaton

❖ Proving that a particular automaton is minimal is usually not a simple matter. It is easy to see

that the language accepted by the automaton above is

$$L = 01^*0(0 + 1)^* + 1(0 + 1)^*$$

It can be found (exercise!) that the index of the relation R_L is **3**. Therefore, by the Myhill–Nerode theorem, every automaton defining L must have at least **3** states.

❖ **Example 3.20:** The following automaton is not minimal.

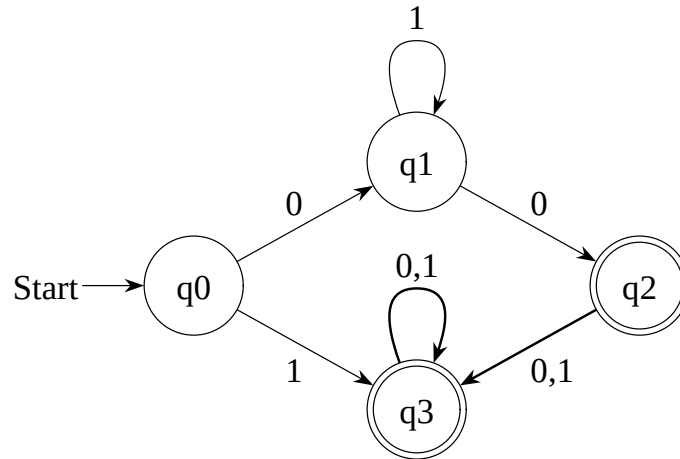


Fig. 3.36: An automaton that is not minimal

It is not difficult to see that this automaton is equivalent to the preceding automaton. In this case, it is easy to see, but in other cases it may be very difficult or impossible.

❖ **Proposition 3.20:** For every automaton M , an equivalent minimal automaton can be found.

- ❖ **Proof:** Let n be the number of states of M . There are finitely many automata with fewer than n states (up to isomorphism). Examine each of these automata and determine whether it is equivalent to M (recall that we have an algorithm for determining whether two automata are equivalent).
- ❖ If none of them is equivalent to M , then M is minimal, and we are done. If several are equivalent to M , choose an automaton with the smallest number of states, and finish.
- ❖ **Question:** Every automaton has an equivalent minimal automaton. Can there be two such nonisomorphic minimal automata?
- ❖ The algorithm above is atrocious in terms of computation time. The number of automata with fewer than n states is of order at least n^n . Moreover, this number of automata must be multiplied by the computation time required to test equivalence to M .
- ❖ Thus, the algorithm above is impractical.
- ❖ Compilers, scanners, and the many kinds of interpreters use automata to perform their operations. Using minimal automata may significantly accelerate their execution.
- ❖ Computer chips, too, are in fact automata. Proper chip design must take into account the

corresponding minimal automaton (or something close to it).

- Later, we present an extremely fast algorithm that is also practical.
- One consequence of the Myhill–Nerode theorem is the following theorem.

Existence and Uniqueness of an Equivalent Minimal Automaton

i Theorem 3.21: For every deterministic automaton M , there is a unique equivalent minimal automaton.

- In other words, if M_1 and M_2 are two minimal automata equivalent to M , then necessarily $M_1 \cong M_2$.
- ✓ **Proof:** Let L be the language accepted by M , and let n be the index of the relation R_L . By the Myhill–Nerode theorem, every automaton defining L must have at least n states. In the proof of the Myhill–Nerode theorem, we constructed such an automaton using the equivalence classes of R_L . Denote this automaton by M' , and denote its state set by

$$Q' = \{q'_0, q'_1, \dots, q'_{n-1}\}$$

It remains to prove that M' is the unique minimal automaton equivalent to M . Suppose that M'' is also a minimal automaton equivalent to M . We prove that $M'' \cong M$. Clearly, the automaton M'' also has exactly n states

$$Q'' = \{q''_0, q''_1, \dots, q''_{n-1}\}$$

Let R'' be the equivalence relation defined by the automaton M'' . Then, by the Myhill–Nerode theorem, R'' is a refinement of R_L . This fact necessarily implies that R'' and R_L have exactly the same equivalence classes, and therefore these two relations are completely identical.

We conclude by defining the isomorphism function $f : Q' \rightarrow Q''$. Let $q' \in Q'$, and let w be a string for which $\delta'(q'_0, w) = q'$. Let q'' be the state of M'' that defines the equivalence class of w . Define $f(q') = q''$. The details of the proof that f is indeed an isomorphism are left as an exercise. ■

- We now turn to the description of the algorithm for finding the minimal automaton. For this purpose, we need the following two concepts.
- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic automaton. Two states p, q are called **equivalent** if

$$\forall w \in \Sigma^* : \delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F$$

- Two states p, q are called **distinguishable** if there is a string w such that $\delta(p, w) \in F$ but $\delta(q, w) \notin F$, or vice versa.
- **The idea:** We can treat any two or more equivalent states as if they were actually one state.



That is, we merge them into a single state. This requires redefining the transition function δ . We therefore find all the classes of equivalent states. Each class of equivalent states becomes a single state in the minimal automaton M' . We define the transition function δ' accordingly, thereby obtaining the minimal automaton equivalent to M .

- The only problem with this idea is: to determine whether two states p and q are equivalent, according to the definition, infinitely many tests must be performed. We must verify that for every string w , $\delta(p, w)$ belongs to F if and only if $\delta(q, w)$ belongs to F .
- It is simpler to find all pairs of states (p, q) that are distinguishable. If we can create a complete list of them, we can easily determine the classes of equivalent states.
- At first glance, this problem also appears not to be so simple. To determine whether states p , q are distinguishable, we must find a string w for which exactly one of the two states $\delta(p, w)$, $\delta(q, w)$ belongs to F . Here too, it is not clear whether the search for w is finite or infinite.
- It turns out that there is a fast, practical algorithm for finding all pairs of distinguishable states, based on two very simple claims. (Running time $O(n^2)$; an algorithm with running time $O(|\Sigma|n \log n)$ also exists).
- **Auxiliary claim 1:** For any two states $p, q \in Q$, if there is a symbol a such that the states $r = \delta(p, a)$, $s = \delta(q, a)$ are distinguishable, then the states p , q are also distinguishable.
- **Proof:** If the states r , s are distinguishable, then by definition there is a string w such that

$$[\delta(r, w) \in F \wedge \delta(s, w) \notin F] \vee [\delta(r, w) \notin F \wedge \delta(s, w) \in F]$$

By the definition of the extension of δ :

$$\begin{aligned} \delta(p, aw) &= \delta(\delta(p, a), w) = \delta(r, w) \\ \delta(q, aw) &= \delta(\delta(q, a), w) = \delta(s, w) \end{aligned}$$

and therefore

$$[\delta(p, aw) \in F \wedge \delta(q, aw) \notin F] \vee [\delta(p, aw) \notin F \wedge \delta(q, aw) \in F]$$

and therefore the states p , q are distinguishable. ■

- **Auxiliary claim 2:** Every pair of states (p, q) such that $p \in F$ and $q \in Q - F$ is distinguishable.
- **Proof:** For example, if we take the empty string $w = \varepsilon$, then

$$\delta(p, \varepsilon) = p \in F, \quad \delta(q, \varepsilon) = q \notin F$$

■



- We now present an algorithm for marking all pairs of distinguishable states (p, q) . Assume that we have a two-dimensional table of the following form:

q1						
q2						
q3						
⋮						
⋮						
⋮						
q _n						
	q0	q1	q2	⋯⋯⋯	q _{n-1}	

Fig. 3.37: Table of distinguishable states

- Note that in this table, exactly one cell corresponds to each pair of distinct states. This is why the table is triangular rather than square.
- At the beginning of the process, this table is empty. Whenever we discover that a pair of states (p, q) is distinguishable, we mark the corresponding cell with an X.
- Each pair of states (p, q) has an associated list of pairs of states, $\text{list}[p, q]$. As we will see, this list contains all pairs whose distinguishability depends on the distinguishability of the pair (p, q) . The first time we discover that the pair (p, q) is distinguishable, all the pairs in the list $\text{list}[p, q]$ are immediately marked as distinguishable.
- At the beginning of the process, all these lists are empty: $\text{list}[p, q] = \emptyset$.
- By auxiliary claim 2, we can immediately mark all pairs (p, q) such that $p \in F$ and $q \in Q - F$.
- Next, process every as-yet unmarked pair of states (p, q) , in any convenient order, and apply the following procedure to the pair (p, q) :
- **Procedure Check(p,q):**
- Iterate over all symbols a in the alphabet Σ .
 - If the pair of states $(\delta(p, a), \delta(q, a))$ is marked, also mark the pair (p, q) and execute the procedure $\text{ListsMark}(p, q)$. Return.
 - If all the pairs $(\delta(p, a), \delta(q, a))$ are unmarked, insert the pair (p, q) into every list $\text{list}[\delta(p, a), \delta(q, a)]$, provided that $\delta(p, a) \neq \delta(q, a)$. Return.
- **Procedure ListsMark(p,q):**
- If the list $\text{list}[p, q]$ is empty – return.
 - Otherwise, for every pair (r, s) in the list $\text{list}[p, q]$, mark (r, s) and execute

ListsMark(r, s).

- This completes the algorithm.
- After finding all pairs of distinguishable states, every pair of states (p, q) that is not marked in the table above is a pair of equivalent states. We denote this by $p \equiv q$. Clearly, this is an equivalence relation. Each equivalence class of this relation corresponds to one state in the minimal automaton M' .
- The transition function δ' of M' is defined naturally as follows:
if $\delta(p, a) = q$, then $\delta'(p', a) = q'$, where p', q' are the equivalence classes of p, q , respectively.

Example

- Before proving the correctness of the algorithm, let us consider a simple example of its use. We find the minimal automaton M' for the automaton M given in Figure 3.38.

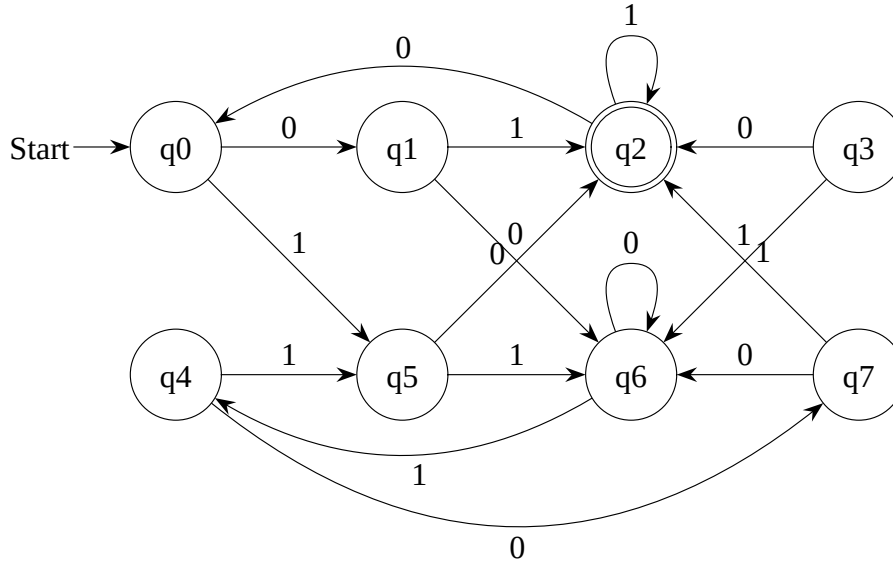


Fig. 3.38: The automaton M

- In this case, $F = \{q_2\}$, so we immediately mark all the pairs

$$(q_0, q_2), (q_1, q_2), (q_2, q_3), (q_2, q_4), (q_2, q_5), (q_2, q_6), (q_2, q_7)$$

- Now mark the pair (q_0, q_1) because

$$(\delta(q_1, 1), \delta(q_0, 1)) = (q_2, q_5)$$

and the pair (q_2, q_5) is already marked.

- Mark the pair (q_0, q_3) because

$$(\delta(q_0, 0), \delta(q_3, 0)) = (q_1, q_2)$$

and the pair (q_1, q_2) is already marked.

- Now consider the pair (q_0, q_4)

$$(\delta(q_0, 0), \delta(q_4, 0)) = (q_1, q_7)$$

$$(\delta(q_0, 1), \delta(q_4, 1)) = (q_5, q_5)$$

The pair (q_1, q_7) is unmarked, and the pair (q_5, q_5) is invalid. We therefore add the pair (q_0, q_4) to the list of (q_1, q_7) :

$$\text{list}[q_1, q_7] = \{(q_0, q_4)\}$$

- Consider the pair (q_0, q_6)

$$(\delta(q_0, 0), \delta(q_6, 0)) = (q_1, q_6)$$

$$(\delta(q_0, 1), \delta(q_6, 1)) = (q_5, q_4)$$

Both pairs (q_1, q_6) , (q_5, q_4) are unmarked, so

$$\text{list}[q_1, q_6] = \{(q_0, q_6)\}$$

$$\text{list}[q_5, q_4] = \{(q_0, q_6)\}$$

- When we reach the pair (q_1, q_6) , it is marked because

$$(\delta(q_1, 1), \delta(q_6, 1)) = (q_2, q_4)$$

and the pair (q_2, q_4) is already marked. The list of (q_1, q_6) is not empty. It contains the pair (q_0, q_6) , so we must also mark the pair (q_0, q_6) . (The string **01** distinguishes q_0 from q_6).

- After processing all possible pairs, we obtain the final table:



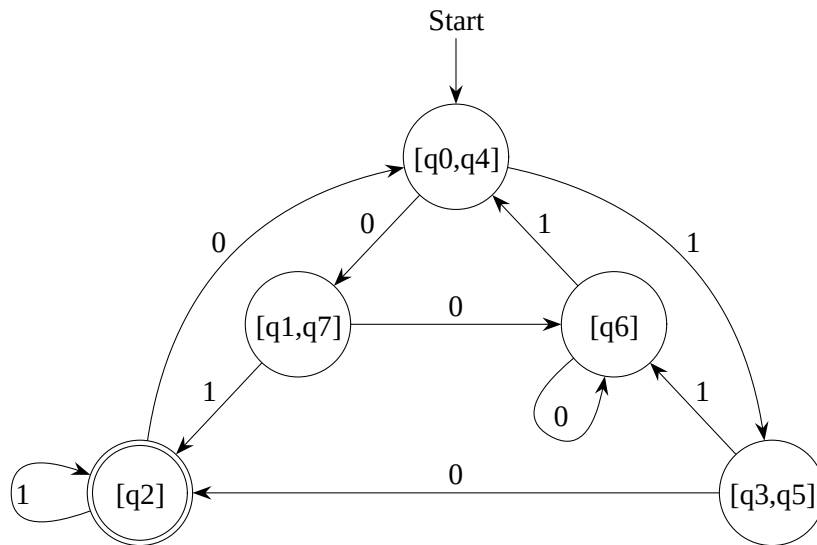
q1	X						
q2	X	X					
q3	X	X	X				
q4		X	X	X			
q5	X	X	X		X		
q6	X	X	X	X	X	X	
q7	X		X	X	X	X	X
	q0	q1	q2	q3	q4	q5	q6

Fig. 3.39: Table of distinguishable states

➤ The table shows that the equivalent states are

$$q_0 \equiv q_4, \quad q_1 \equiv q_7, \quad q_3 \equiv q_5$$

➤ The minimal automaton is shown in Figure 3.40

Fig. 3.40: A minimal automaton for $L(M)$

Proof of the Correctness of the Algorithm

➤ Let

$$M = (Q, \Sigma, \delta, q_0, F)$$

be a given deterministic automaton to which the algorithm is applied, and let

$$M' = (Q', \Sigma, \delta', [q_0], F')$$

be the resulting automaton.

$$Q' = \{ [q] : q \in Q \}$$

where $[q]$ is the equivalence class of all states equivalent to q , and

$$\delta'([q], a) = [\delta(q, a)]$$

We must prove that

a. this definition of M' is well-defined.

b. $L(M') = L(M)$

c. M' is indeed a minimal automaton.

We explain why the definition of δ' is valid. If $p \equiv q$, then $\delta(p, a) \equiv \delta(q, a)$. For if $\delta(p, a) \not\equiv \delta(q, a)$, then there is a string w such that

$$\delta(\delta(p, a), w) \in F \not\equiv \delta(\delta(q, a), w) \in F$$

This immediately implies that

$$\delta(p, aw) \in F \not\equiv \delta(q, aw) \in F$$

and therefore $p \not\equiv q$, contrary to the assumption.

It is not difficult to prove by induction on $|w|$ that

$$\forall w \in \Sigma^* : \delta'(q_0, w) = [\delta(q_0, w)]$$

and therefore

$$\forall w \in \Sigma^* : \delta'(q_0, w) \in F' \Leftrightarrow \delta(q_0, w) \in F$$

It follows that

$$L(M) = L(M')$$

It remains to prove that M' is a minimal automaton. For this purpose, we prove that the number of states of M' equals the number of equivalence classes of R_L , where $L = L(M)$. Suppose, by way of contradiction, that the number of states of M' is greater than the index of R_L . Then there are two states p and q such that $[p] \neq [q]$, but the equivalence classes of the two states are contained in a single equivalence class of R_L . That is, there are two strings x, y such that

$$\delta(q_0, x) = p, \quad \delta(q_0, y) = q$$

but xR_Ly .

The assumption $[p] \neq [q]$ implies that the pair (p, q) is distinguishable, and therefore there is

a string w that distinguishes p from q :

$$\delta(p, w) \in F \not\equiv \delta(q, w) \in F$$

This immediately implies that

$$\delta(q_0, xw) \in F \not\equiv \delta(q_0, yw) \in F$$

which immediately implies that

$$xw \in L \not\equiv yw \in L$$

and hence $\neg xwR_L yw$. This contradicts the fact that $xR_L y$, because R_L is right-invariant under concatenation. Therefore, the number of states of M' must equal the index of R_L , and hence M' is a minimal automaton. ■

Exercises for chapter 3

Guidance: Some of the exercises can be solved, or their results verified, by using a simple simulation program that can be run in an Internet browser. We would be pleased to receive code from students for solutions to additional exercises.

<https://colab.research.google.com/github/samyzaf/notebooks/blob/main/fa2.ipynb>

- Find a regular expression for the following nondeterministic automaton

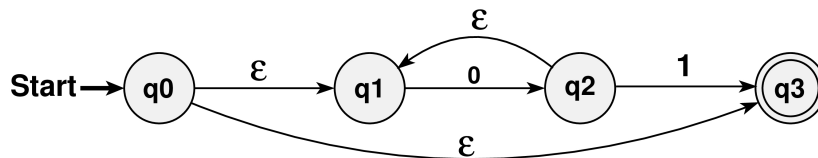


Fig. 3.41: A nondeterministic automaton with ϵ -transitions and 4 states

- Consider the automaton M in Figure 3.42
 - Is there a string matching the regular expression 1^*001^* that is accepted by M ?
 - Prove that all strings matching the regular expression $(11)^*0(111)^*$ are accepted by M .
 - Describe precisely all strings matching the regular expression 0^* that are accepted by M .

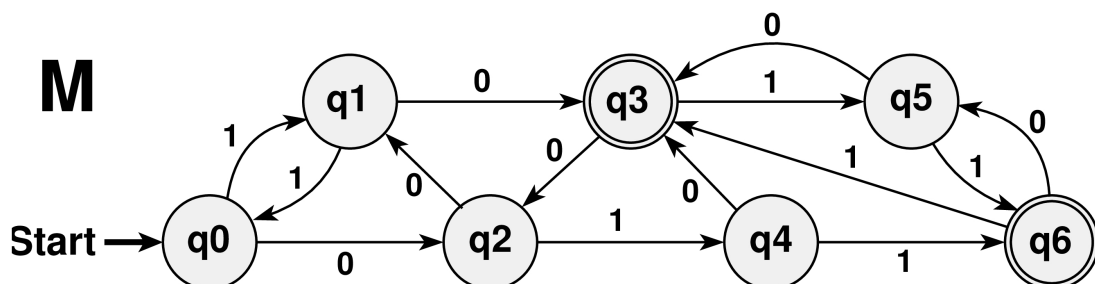


Fig. 3.42: A deterministic automaton M with seven states $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ over the alphabet $\{0, 1\}$

3. Find a regular expression for the language L and its complement $\overline{L}$ (in parts a and b)
- (a) $L = \{0^n 1^{2m} : n \geq 3, m \geq 0\}$
- (b) $L = \{0^n 1^m : n < 4, m < 4\}$
- (c) $L = \{uvu : u, v \in \{0, 1\}^*, |u| \leq 3\}$
- (d) $L = \{w \in \{0, 1\}^* : w[n-1:n] \neq 01\}$
4. Try to give a simple verbal description of the language generated by the regular expression

$$(00)^* 1 (00)^* + 0(00)^* 10(00)^*$$



5. Construct a nondeterministic automaton for each of the following regular expressions
- (a) $(0^* + 1)1^*0^*$ (b) $(01)^* + (10)^*$ (c) $(01)^+ + (10)^+$
6. Let L be the language of all binary strings containing an even number of zeros and an odd number of ones, but not containing the substring 01 .
- a. Construct a five-state deterministic automaton that accepts L .
- b. Find a regular expression that generates L .

7. Find the shortest string in each of the following languages

(a) $10 + (0 + 11)0^*1$ (b) $10((110)^* + 10)^+$ (c) $((00+11)^* + (001+110)^*)^*$



8. Develop an algorithm for finding the shortest string that matches a given regular expression.
9. Find a regular expression over the alphabet $\{0, 1\}$ for the language of all binary numbers whose numerical value is greater than 40 .
10. The regular expression $(0 + 1)^* 0 (0 + 1)^* 1 (0 + 1)^* 0 (0 + 1)^*$ is given. Find two distinct strings that match the expression and two strings that do not match it.



11. Prove: for every language L , $L^+ = L^*$ if and only if $\varepsilon \in L$.
12. Prove that: $(ab)^+(b^*a^* + b^*) = (ab)^*ab^+a^*$.
13. Convert the deterministic automaton depicted in Figure 3.43 into a regular expression

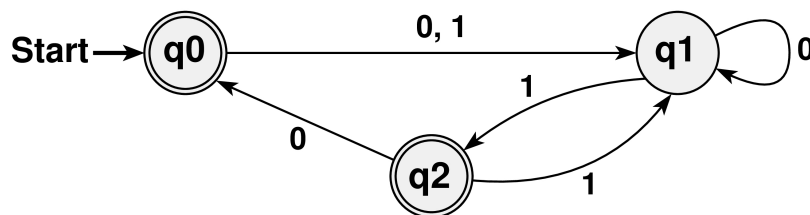


Fig. 3.43: A deterministic automaton with three states $\{q_0, q_1, q_2\}$ over the alphabet $\{0, 1\}$



14. Find a regular expression generating all binary strings that do not contain **001** as a substring.
15. What is the language accepted by the following deterministic automaton? Try to give as simple a description as possible.

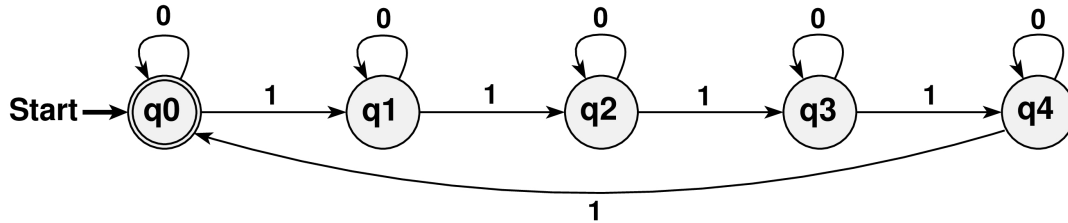


Fig. 3.44: A deterministic automaton with five states $\{q_0, q_1, q_2, q_3, q_4\}$ over the alphabet $\Sigma = \{0, 1\}$

Answer: The automaton accepts binary strings whose number of ones is a multiple of 5. The corresponding regular expression is $0^*((10^*)^4)^*$.

16. What is the language accepted by the deterministic automaton in Figure 3.45? Try to give as simple a description as possible.

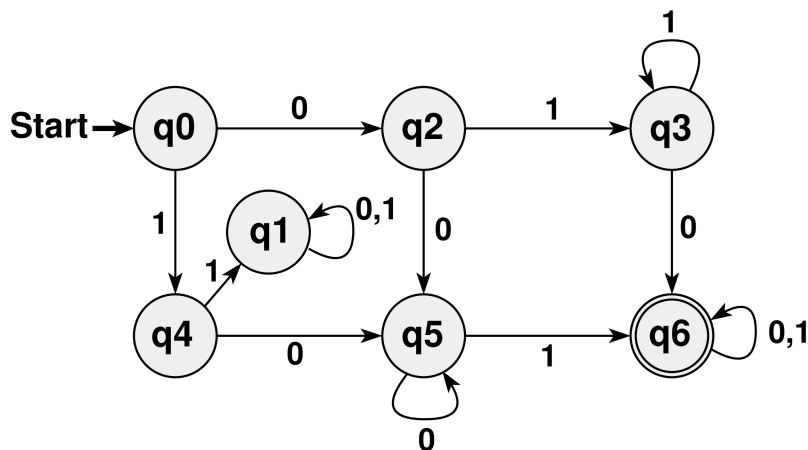


Fig. 3.45: A deterministic automaton with seven states $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ over the alphabet $\Sigma = \{0, 1\}$

17. What is the language accepted by the generalized automaton in Figure 3.46? Try to give as simple a description as possible.

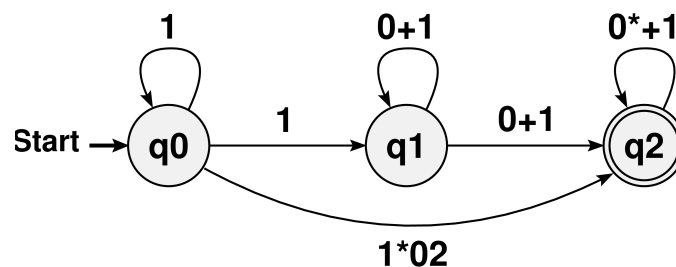


Fig. 3.46: A generalized automaton with three states $\{q_0, q_1, q_2\}$ over the alphabet $\Sigma = \{0, 1, 2\}$

18. Find a regular expression for the language accepted by the generalized automaton in Figure 3.47.

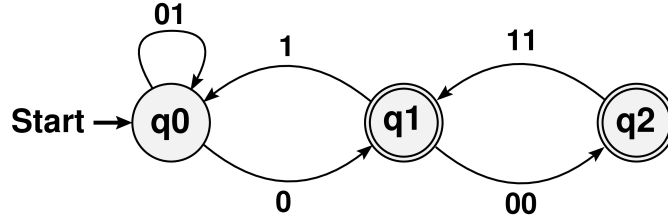


Fig. 3.47: A generalized automaton with three states $\{q_0, q_1, q_2\}$ over the alphabet $\Sigma = \{0, 1\}$



19. Let L be a given language over an alphabet Σ . Define a new language by

$$L_{\frac{1}{2}} = \{x \in \Sigma^* : \exists y \in \Sigma^* [|x| = |y| \wedge xy \in L]\}$$

That is, $L_{\frac{1}{2}}$ consists of all halves of words (of even length) in the language L . Prove: if L is regular, then $L_{\frac{1}{2}}$ is regular.

20. Let L be a given language. Define the language

$$L' = \{w \in \Sigma^* : w \in L \wedge w^R \in L\}$$

Prove or disprove by a counterexample: if L is regular, then L' is also regular.

21. Let L be a given language. Define the language

$$L' = \{w \in \Sigma^* : w \in L \wedge w = w^R\}$$

Prove or disprove by a counterexample: if L is regular, then L' is also regular.

22. Find all equivalence classes of the relation R_L (defined in connection with the Myhill–Nerode theorem) for the following languages

(a) $(10 + 01)(0 + 1)^*$

(b) $110(0 + 1)^*100$

23. Let $L = \{ww : w \in \{0, 1\}^*\}$. Prove that the index of the language L is ∞ .

24. Prove that the language $L = \{a^m b^n c^n : m, n \in \mathbb{N}\}$ is not regular, but nevertheless satisfies the conclusion of the pumping lemma for $n = 1$.



25. Let L be a language over an alphabet Σ . Define the language L' by

$$L' = \{w \in \Sigma^* : \exists x, y, z \in \Sigma^* [xyz \in L \wedge w = xz]\}$$

Prove or disprove by a counterexample: if L is a regular language, then L' is also regular.

Hint: a^*bc^*

26. Prove that the following languages are not regular



- (a) $L = \{a^m b^n c^{m+n} : m, n \in \mathbb{N}\}$
- (b) $L = \{0^m 1^n 0^n : m, n \in \mathbb{N}\}$
- (c) $L = \{0^m 1^n 0^{m^2+n^2} : m, n \in \mathbb{N}\}$
- (d) $L = \{ww : w \in \{0, 1\}^*\}$
- (e) $L = \{0^n 1^{n+i} : n \in \mathbb{N}, 1 \leq i \leq 2\}$
- (f) $L = \{w_1 0 w_2 : w_1, w_2 \in \{0, 1\}^*, w_1 \neq w_2\}$

27. Prove that the following languages are not regular

- (a) $L_1 = \{0^n 1^{2n} 2^n : n \in \mathbb{N}\}$
- (b) $L_2 = \{0^{n^2} 1^{2n} : n \in \mathbb{N}\}$
- (c) $L_3 = \{w 0 1 0 w : w \in \{0, 1\}^*\}$
- (d) $L_4 = \{www : w \in \{0, 1\}^*\}$

28. Figure 3.48 presents two deterministic automata M_1 , M_2 . Using the product automaton (first solve Exercises 8, 9, in the preceding chapter), find a regular expression generating the language $L = L(M_1) \cap L(M_2)$.



Fig. 3.48: Figure for Exercise 28: construct a regular expression for the intersection language $L(M_1) \cap L(M_2)$



29. Let L be a regular language in which all words have length at most n . Prove that every deterministic automaton accepting L must have at least $n + 1$ states.



30. Let L be a regular language in which all words have length at least n . Prove that every deterministic automaton accepting L must have at least $n + 1$ states.

31. Convert the deterministic automaton in Figure 3.49 into an equivalent automaton with a minimal number of states.

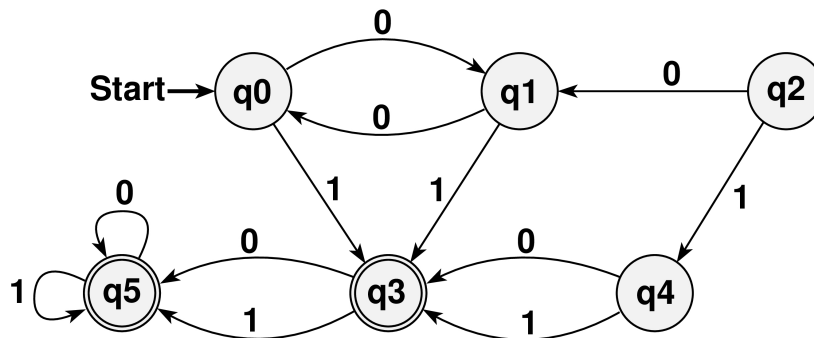


Fig. 3.49: Figure for Exercise 31: convert the deterministic automaton into an equivalent automaton with a minimal number of states





32. Convert the deterministic automaton in Figure 3.50 into an equivalent automaton with a minimal number of states.

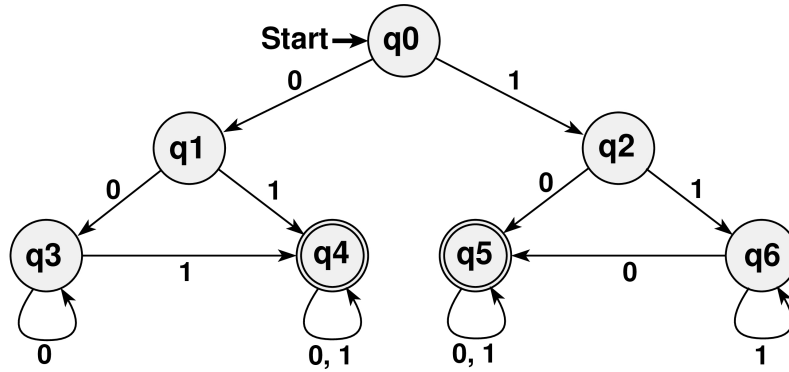


Fig. 3.50: Convert the deterministic automaton into an equivalent automaton with a minimal number of states



33. Convert the deterministic automaton in Figure 3.51 into an equivalent automaton with a minimal number of states.

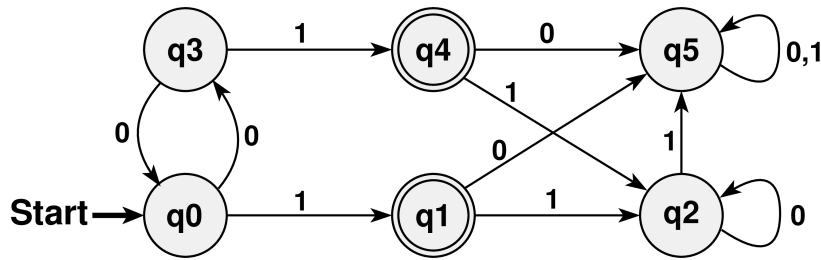


Fig. 3.51: Figure for Exercise 33: convert the deterministic automaton into an equivalent automaton with a minimal number of states



34. Find a minimal deterministic automaton for each of the following languages. Prove your answer.

- (a) $L = \{0^n1 : n \geq 0\} \cup \{1^n0 : n \geq 1\}$
- (b) $L = \{0^m1^n : m \geq 2, n \geq 1\}$
- (c) $L = \{0^{2n} : n \geq 0\} \cup \{1^{3n} : n \geq 0\}$
- (d) $L = \{0^n : n \neq 1, n \neq 3\}$

35. Prove or disprove by a counterexample: if $M = (Q, \Sigma, \delta, q_0, F)$ is a minimal deterministic automaton for the language L , then $\bar{M} = (Q, \Sigma, \delta, q_0, Q - F)$ is a minimal deterministic automaton for the language $\bar{L}$,



Chapter 4

Context-Free Languages

In this chapter, we study a more general and important family of formal languages called **context-free languages**. Most programming languages and content languages are context-free languages, so familiarity with the fundamental properties of these languages is important for advanced courses in programming languages, compilers, and algorithms.

4.1 Context-Free Grammars

- i** **Definition 4.1:** A **context-free grammar** G consists of the following four components
- V A finite set of **syntactic variables** (or simply **variables**).
 - T A finite set of terminal symbols (**Terminals**).
 - $\mathcal{P}$ A finite set of productions of the form $A \rightarrow \alpha$, where A is a variable and α is a string from $(V \cup T)^*$.
 - S A syntactic variable with a special status – the **start variable**.

- Abbreviated notation: $G = (V, T, \mathcal{P}, S)$
- We shall always assume that the sets V and T are disjoint.

➤ **Example 4.1:**

$$\begin{aligned} V &= \{S, A, B\} \\ T &= \{a, b, c\} \end{aligned}$$

The set of productions

$$\begin{aligned} S &\rightarrow Ac \\ A &\rightarrow bAB \\ B &\rightarrow cBB \end{aligned}$$

- Every string over the alphabet $V \cup T$ is called a **sentential form**, and every string in T^* is called a **sentence**.
- The productions are the rules according to which we derive one sentential form from another. We



say that a sentential form γ is **derived** from a sentential form β by the production $A \rightarrow \alpha$, if β contains the variable A , and γ is obtained by replacing the variable A with the sentential form α . In other words, $\beta = \beta_1 A \beta_2$, $\gamma = \beta_1 \alpha \beta_2$.

- For example, the sentential form $bAcBB$ is **immediately derived** from the sentential form bAB by applying the third production. We write this as follows:

$$bAB \Rightarrow bAcBB$$

For readability, we use red and underlining to emphasize the production that was applied:

$$bA\textcolor{red}{B} \Rightarrow bA\underline{cBB}$$

Similarly,

$$bccABa \Rightarrow bccbABBa$$

At first glance, it is not clear which production was used here, so in similar cases we use highlighting when necessary:

$$bcc\textcolor{red}{A}Ba \Rightarrow bcc\underline{bAB}Ba$$

The red color and underlining clearly show that we applied the second production.

- **Emphasis:** every production has the form $A \rightarrow \beta$, where A is a variable belonging to the set V , and β is a sentential form belonging to $(V \cup T)^*$. This is the principal characteristic of a context-free grammar.

i Definition 4.2: Let $G = (V, T, P, S)$ be a given context-free grammar, let α , β_1 , β_2 be three sentential forms in $(V \cup T)^*$, and let $A \rightarrow \alpha$ be a production of G . Then from the sentential form $\beta_1 A \beta_2$ we can **immediately derive** the sentential form $\beta_1 \alpha \beta_2$, and we say that the production $A \rightarrow \alpha$ is applied to the sentential form $\beta_1 A \beta_2$ in order to immediately derive the sentential form $\beta_1 \alpha \beta_2$. In abbreviated notation,

$$\beta_1 A \beta_2 \Rightarrow \beta_1 \alpha \beta_2$$

i Definition 4.3: A finite sequence of sentential forms $(\alpha_1, \alpha_2, \dots, \alpha_n)$ over a context-free grammar G is called a **derivation sequence** if

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$$

i Definition 4.4: We say that a sentential form γ is **derived** from a sentential form β , and write

$$\beta \xRightarrow{*} \gamma$$

if there is a derivation sequence $(\alpha_1, \alpha_2, \dots, \alpha_n)$ such that $\beta = \alpha_1$, $\gamma = \alpha_n$.

The trivial derivation $\beta \xRightarrow{*} \beta$ is also included in the definition.

- In other words, $\beta \xRightarrow{*} \gamma$ if and only if the sentential form γ follows from the sentential form β by applying zero or more productions. It is important to distinguish between an immediate derivation $\beta \Rightarrow \gamma$ and a derivation $\beta \xRightarrow{*} \gamma$. We leave the easy proof of the following assertion to the student.

i Proposition 4.1: (Fundamental properties of the derivation relation)

- a. For every sentential form α , $\alpha \xRightarrow{*} \alpha$ (application of zero productions).
- b. If $\alpha \xRightarrow{*} \beta$ and $\beta \xRightarrow{*} \gamma$, then $\alpha \xRightarrow{*} \gamma$.
- c. If $\beta_1 \xRightarrow{*} \beta_2$, then for any two sentential forms α, γ ,

$$\alpha\beta_1\gamma \xRightarrow{*} \alpha\beta_2\gamma$$

- i Definition 4.5:** Let G be a given context-free grammar. The *language generated* by the grammar G is defined by

$$L(G) = \left\{ w : w \in T^* \text{ and } S \xRightarrow{*} w \right\}$$

This is the set of all sentences that can be derived from the start variable S .

- That is, a string w belongs to the language $L(G)$ if and only if
- a. w consists only of terminal symbols ($w \in T^*$).
 - b. w is derived from the start variable S ($S \xRightarrow{*} w$).
- To prove that a string of terminal symbols w belongs to the language $L(G)$, we must prove that there is a derivation sequence $(S, \alpha_1, \alpha_2, \dots, \alpha_n, w)$

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$$

that begins with the start variable S and ends with the string w .

- Warning: For $w \in L(G)$, there may be more than one derivation sequence!

- ◆ **Example 4.2:** The language of arithmetic expressions can be defined by a context-free grammar similar to the one in the following example

$$\begin{aligned} V &= \{S\} \\ T &= \{a, b, +, *, (,)\} \end{aligned}$$



The set of productions

$$\begin{aligned} S &\rightarrow S + S \\ S &\rightarrow S * S \\ S &\rightarrow (S) \\ S &\rightarrow a \\ S &\rightarrow b \\ S &\rightarrow c \end{aligned}$$

➤ For example, let us derive the expression $a * (b + c)$

$$\begin{aligned} S &\rightarrow S * S \rightarrow S * (S) \rightarrow a * (S) \rightarrow a * (S + S) \\ &\rightarrow a * (b + S) \rightarrow a * (b + c) \end{aligned}$$

This proves that

$$S \xRightarrow{*} a * (b + c)$$

and therefore

$$a * (b + c) \in L(G)$$

➤ A rigorous proof that this grammar indeed generates the language of all valid arithmetic expressions (using only the symbols a , b , c) is not simple at this stage. Instead, we turn to a simpler example.

◆ **Example 4.3:** Let G be the following grammar

$$\begin{aligned} V &= \{S\} \\ T &= \{0, 1\} \end{aligned}$$

The set of productions $\mathcal{P}$:

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow 01 \end{aligned}$$

i Proposition 4.2: For every $n \geq 1$, $0^n 1^n \in L(G)$.

✓ **Proof:** By applying the first production $n - 1$ times and finally applying the second production, we obtain

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \dots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n 1^n$$

Therefore, $S \xRightarrow{*} 0^n 1^n$, and hence $0^n 1^n \in L(G)$. ■

i Proposition 4.3: If $w \in L(G)$, then there is an $n \geq 1$ such that $w = 0^n 1^n$.

✓ **Proof:** Let

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n = w$$

be a derivation sequence for w .

Every application of the first production leads to a sentential form containing exactly one S , whereas applying the second production leaves us with no variables at all, thereby ending the derivation. Therefore, necessarily, all the sentential forms $\alpha_1, \dots, \alpha_{n-1}$ are obtained by applying the first production, and only the sentential form α_n is obtained by applying the second production. Therefore, $\alpha_{n-1} = 0^{n-1}S1^{n-1}$, and finally $\alpha_n = w = 0^n1^n$. ■

i Definition 4.6: A language L is called a **context-free language** if there is a context-free grammar G such that $L = L(G)$.

◇ **Example 4.4:** The language

$$L = \{0^n1^n : n = 1, 2, 3, \dots\}$$

is a context-free language. In one of the preceding lessons, we proved that L is not a regular language. We may therefore conclude the following theorem.

i Theorem 4.4: There are context-free languages that are not regular languages.

➤ **Question:** Is every regular language a context-free language?

➤ Later, we shall prove that the answer to this question is affirmative, and hence the class of context-free languages is larger than the class of regular languages.

i Definition 4.7: Two context-free grammars G_1, G_2 are called **equivalent** if

$$L(G_1) = L(G_2)$$

Abbreviated notation: $G_1 \equiv G_2$.

◇ **Example 4.5:** Let G_1 be the grammar

$$\begin{aligned} V_1 &= \{S\} \\ T_1 &= \{0, 1\} \\ S &\rightarrow 0S1 \\ S &\rightarrow 01 \end{aligned}$$



and let G_2 be the grammar

$$\begin{aligned} V_2 &= \{S, A\} \\ T_2 &= \{0, 1\} \\ S &\rightarrow A \\ A &\rightarrow 0A1 \\ A &\rightarrow 01 \end{aligned}$$

It is easy to see that $G_1 \equiv G_2$.

Notation Conventions

- a. Syntactic variables will usually be denoted by the capital letters $A, B, C, D, E, \dots$
- b. Terminal symbols will be denoted by the lowercase letters $a, b, c, d, e, \dots$, or by the digits $0, 1, 2, 3, 4, \dots$
- c. Sentential forms will usually be denoted by lowercase Greek letters

$$\alpha, \beta, \gamma, \dots$$

- d. Sentences will be denoted by

$$u, v, w, x, y, z$$

- e. If the set $\mathcal{P}$ contains several productions with the same left-hand side

$$A \rightarrow \alpha_1, \quad A \rightarrow \alpha_2, \quad \dots, \quad A \rightarrow \alpha_n$$

it is customary to combine all of them into a single production

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

The symbol " $|$ " means "or".

For example, instead of the three productions

$$\begin{aligned} A &\rightarrow 0BB \\ A &\rightarrow 11 \\ A &\rightarrow A10 \end{aligned}$$

we write

$$A \rightarrow 0BB | 11 | A10$$

As another example, the entire grammar of arithmetic expressions can be written using the following single production

$$S \rightarrow S + S | S * S | (S) | a | b | c$$



- f. In most cases in which we present a context-free grammar G , we shall not bother to list the elements of the sets V and T . The essential part is the list of productions of G , from which one can determine exactly which symbols are variables and which are terminal symbols.

◆ **Example 4.6:** We present a grammar G that generates the language L of all binary strings with an equal number of zeros and ones. For every $w \in \{0, 1\}^*$, define

$$\begin{aligned}\#_0(w) &= \text{the number of zeros in the string } w \\ \#_1(w) &= \text{the number of ones in the string } w\end{aligned}$$

Then L is defined by

$$L = \{w \in \{0, 1\}^* : \#_0(w) = \#_1(w)\}$$

It is easy to prove that L is not a regular language, but we now present a context-free grammar G that generates L :

$$\begin{aligned}V &= \{S, A, B\} \\ T &= \{0, 1\}\end{aligned}$$

The set of productions:

$$\begin{array}{ll} S \rightarrow 0B & A \rightarrow 1AA \\ S \rightarrow 1A & B \rightarrow 1 \\ A \rightarrow 0 & B \rightarrow 1S \\ A \rightarrow 0S & B \rightarrow 0BB \end{array}$$

◆ **Auxiliary claim:** For every $w \in \{0, 1\}^+$,

- $S \xRightarrow{*} w$ if and only if $\#_0(w) = \#_1(w)$.
- $A \xRightarrow{*} w$ if and only if $\#_0(w) = \#_1(w) + 1$.
- $B \xRightarrow{*} w$ if and only if $\#_0(w) + 1 = \#_1(w)$.

Proof: By induction on the length of w .

If $|w| = 1$, then $A \xRightarrow{*} 0$, $B \xRightarrow{*} 1$, and no string of length 1 is derived from S . All the productions, except for $A \xRightarrow{*} 0$ and $B \xRightarrow{*} 1$, increase the length of the string. Therefore, 0 cannot be reached through B , nor can 1 be reached through A . Thus, the induction basis has been proved.

Now assume that the three assertions above are true for every string of length at most n . We prove that they are also true for strings of length $n + 1$. Let w be a string of length $n + 1$.

(a) If $S \xRightarrow{*} w$, then every derivation sequence for w must begin with $S \Rightarrow 0B$ or $S \Rightarrow 1A$.

In the first case, $w = 0u$, where u is a string of length n and $B \xRightarrow{*} u$. By the induction hypothesis,

$$\#_1(u) = \#_0(u) + 1$$

and it follows immediately that $\#_0(w) = \#_1(w)$. The second case is proved similarly.



This proves only one direction of part a'. It remains to prove that if

$$\#_0(w) = \#_1(w)$$

then $S \xRightarrow{*} w$. The first symbol of w must be 0 or 1. If $w = 0u$, then

$$\#_1(u) = \#_0(u) + 1$$

and therefore, by the induction hypothesis, $B \xRightarrow{*} u$, from which it follows that

$$S \Rightarrow 0B \xRightarrow{*} 0u = w$$

The case $w = 1v$ is proved similarly.

Parts b' and c' are proved similarly. ■

4.2 Additional Properties of the Derivation Relation

i Theorem 4.5: If

$$\begin{aligned}\alpha &\xRightarrow{*} \beta \\ \alpha' &\xRightarrow{*} \beta'\end{aligned}$$

then

$$\alpha\alpha' \xRightarrow{*} \beta\beta'$$

✓ Proof: Let

$$\alpha = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_m = \beta$$

$$\alpha' = \alpha'_1 \Rightarrow \alpha'_2 \Rightarrow \alpha'_3 \Rightarrow \cdots \Rightarrow \alpha'_n = \beta'$$

be the corresponding derivation sequences. The lengths of the two sequences are not necessarily equal. We must examine all three cases $m < n$, $m = n$, $m > n$. We consider only the case $m < n$ and leave the other two cases as an exercise. The following is the derivation sequence for $\beta\beta'$ (from $\alpha\alpha'$)

$$\begin{aligned}\alpha_1\alpha'_1 &\Rightarrow \alpha_2\alpha'_1 \Rightarrow \alpha_2\alpha'_2 \Rightarrow \alpha_3\alpha'_2 \Rightarrow \alpha_3\alpha'_3 \Rightarrow \cdots \Rightarrow \alpha_m\alpha'_m \Rightarrow \\ &\Rightarrow \alpha_m\alpha'_{m+1} \Rightarrow \alpha_m\alpha'_{m+2} \Rightarrow \cdots \Rightarrow \alpha_m\alpha'_n\end{aligned}$$

As required by the definition of derivation, we use exactly one production to pass from one sentential form to the next. ■



i **Theorem 4.6:** If

$$\begin{array}{ccc} \alpha_1 & \xRightarrow{*} & \beta_1 \\ \alpha_2 & \xRightarrow{*} & \beta_2 \\ & \vdots & \\ \alpha_k & \xRightarrow{*} & \beta_k \end{array}$$

then

$$\alpha_1 \alpha_2 \cdots \alpha_k \xRightarrow{*} \beta_1 \beta_2 \cdots \beta_k$$

✓ **Proof:** By induction on k . For $k = 1$, the assertion is clear.

Induction hypothesis: the assertion is true for k .

We must prove: if

$$\begin{array}{ccc} \alpha_1 & \xRightarrow{*} & \beta_1 \\ \alpha_2 & \xRightarrow{*} & \beta_2 \\ & \vdots & \\ \alpha_k & \xRightarrow{*} & \beta_k \\ \alpha_{k+1} & \xRightarrow{*} & \beta_{k+1} \end{array}$$

then

$$\alpha_1 \alpha_2 \cdots \alpha_k \alpha_{k+1} \xRightarrow{*} \beta_1 \beta_2 \cdots \beta_k \beta_{k+1}$$

For this purpose, define

$$\begin{aligned} \alpha &= \alpha_1 \alpha_2 \cdots \alpha_k \\ \alpha' &= \alpha_{k+1} \\ \beta &= \beta_1 \beta_2 \cdots \beta_k \\ \beta' &= \beta_{k+1} \end{aligned}$$

By the induction hypothesis,

$$\alpha_1 \alpha_2 \cdots \alpha_k \xRightarrow{*} \beta_1 \beta_2 \cdots \beta_k$$

that is, $\alpha \xRightarrow{*} \beta$. By the given assumptions, $\alpha_{k+1} \xRightarrow{*} \beta_{k+1}$. that is, $\alpha' \xRightarrow{*} \beta'$. Therefore, the preceding theorem implies that $\alpha \alpha' \Rightarrow \beta \beta'$. that is,

$$\alpha_1 \alpha_2 \cdots \alpha_k \alpha_{k+1} \xRightarrow{*} \beta_1 \beta_2 \cdots \beta_k \beta_{k+1} \quad \blacksquare$$

➤ A useful immediate consequence of this theorem is the following.

i Theorem 4.7: Suppose that

$$\begin{aligned} X_1 &\xRightarrow{*} w_1 \\ X_2 &\xRightarrow{*} w_2 \\ &\vdots \\ X_k &\xRightarrow{*} w_k \end{aligned}$$

where, for every i , $X_i \in T \cup V \cup \{\varepsilon\}$ and w_i is a sentence. If

$$A \rightarrow X_1 X_2 \cdots X_k$$

is a given production, then

$$A \xRightarrow{*} w_1 w_2 \cdots w_k$$

Example 4.7: Let G be the following grammar

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow aBC|bc \\ B &\rightarrow AbC|ac \\ C &\rightarrow ABc|ab \end{aligned}$$

The set of variables is $V = \{A, B, C\}$, and the terminal symbols are $T = \{a, b, c\}$.

It is easy to prove all five of the following derivations

$$\begin{aligned} a &\xRightarrow{*} a \\ A &\xRightarrow{*} aacab \\ b &\xRightarrow{*} b \\ C &\xRightarrow{*} bcacc \\ C &\xRightarrow{*} ab \end{aligned}$$

It is also easy to prove that

$$A \xRightarrow{*} aAbCC$$

Therefore, the last theorem implies

$$A \xRightarrow{*} aaacabbbcacccab$$

❖ **Exercise 4.1:** Let G be the following grammar

$$S \rightarrow ABC$$

$$A \rightarrow aBC$$

$$B \rightarrow AbC$$

$$C \rightarrow ABc$$

Prove that $L(G) = \emptyset$. That is, no sentence can be derived from the variable S !

➤ There are also converse theorems for each of the preceding theorems.

❗ **Theorem 4.8:** If $\alpha\alpha' \xRightarrow{*} \gamma$, then γ can be decomposed into two sentential forms $\gamma = \beta\beta'$ such that

$$\begin{aligned} \alpha &\xRightarrow{*} \beta \\ \alpha' &\xRightarrow{*} \beta' \end{aligned}$$

Moreover, if the length of the derivation sequence for $\alpha\alpha' \xRightarrow{*} \gamma$ is n , then the length of each of the derivation sequences for $\alpha \xRightarrow{*} \beta$ and $\alpha' \xRightarrow{*} \beta'$ is at most n .

✔ **Proof:** We must define two sequences of sentential forms

$$\begin{aligned} \alpha &= \alpha_1, \quad \alpha_2, \quad \dots, \quad \alpha_k \\ \alpha' &= \alpha'_1, \quad \alpha'_2, \quad \dots, \quad \alpha'_m \end{aligned}$$

such that

$$\begin{aligned} \alpha &= \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \dots \Rightarrow \alpha_m = \beta \\ \alpha' &= \alpha'_1 \Rightarrow \alpha'_2 \Rightarrow \alpha'_3 \Rightarrow \dots \Rightarrow \alpha'_n = \beta' \end{aligned}$$

and such that

$$\gamma = \beta\beta'$$

Let

$$\alpha\alpha' = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n = \gamma$$

be the derivation sequence corresponding to $\alpha\alpha' \xRightarrow{*} \gamma$. The definition of α_1, α'_1 is trivial

$$\alpha_1 = \alpha, \quad \alpha'_1 = \alpha'$$

The transition from the sentential form $\gamma_1 = \alpha_1\alpha'_1$ to the sentential form γ_2 is made by applying exactly one production whose left-hand side occurs in α_1 or in α'_1 . Therefore, applying this production changes only one of the strings. If the production is applied to α_1 , we call the result

α_2 . If the production is applied to α'_1 , we call the result α'_2 . We continue in this manner until we reach the sentential form γ_n . The remaining details of the proof are clear.

i Theorem 4.9: If

$$\alpha_1 \alpha_2 \cdots \alpha_k \xRightarrow{*} \beta$$

then the sentential form β can be decomposed into k sentential forms

$$\beta = \beta_1 \beta_2 \cdots \beta_k$$

such that

$$\begin{array}{ccc} \alpha_1 & \xRightarrow{*} & \beta_1 \\ \alpha_2 & \xRightarrow{*} & \beta_2 \\ & \vdots & \\ \alpha_k & \xRightarrow{*} & \beta_k \end{array}$$

✓ Proof: By induction on k . This is very similar to the proof of the corresponding theorem above.

i Theorem 4.10: If

$$A \rightarrow X_1 X_2 \cdots X_k \Rightarrow \cdots \Rightarrow w$$

then there is a unique decomposition of the string w into k strings

$$w = w_1 w_2 \cdots w_k$$

such that

$$\begin{array}{ccc} X_1 & \xRightarrow{*} & w_1 \\ X_2 & \xRightarrow{*} & w_2 \\ & \vdots & \\ X_k & \xRightarrow{*} & w_k \end{array}$$

Moreover, the length of every derivation sequence corresponding to $X_i \xRightarrow{*} w_i$ is less than the length of the derivation sequence corresponding to $A \xRightarrow{*} w$.

4.3 Regular Grammars

➤ We now present a type of grammar that generates the regular languages. This will enrich our computational models for regular languages and also help us understand how the regular languages fit within the framework of context-free languages.

i Definition 4.8: A grammar $G = (V, T, \mathcal{P}, S)$ is called a **right-regular grammar** if all its

productions have one of the following forms

$$X \rightarrow wY$$

$$X \rightarrow w$$

where $X, Y \in V$ are variables and $w \in T^*$ is a sentence (terminal symbols only).

➤ Note that the sentence w may also be the empty string ε !

◆ **Example 4.8:** The grammar $G = (\{S, A\}, \{0, 1\}, \mathcal{P}, S)$, whose productions $\mathcal{P}$ are

$$S \rightarrow 01S$$

$$S \rightarrow A$$

$$A \rightarrow 10$$

is a right-regular grammar. It is easy to see that the grammar G generates the regular language $(01)^+10$.

➤ Similarly, we define a **left-regular grammar**:

❗ **Definition 4.9:** A grammar $G = (V, T, \mathcal{P}, S)$ is called a **left-regular grammar** if all its productions have one of the following forms

$$X \rightarrow Yw$$

$$X \rightarrow w$$

where $X, Y \in V$ are variables and $w \in T^*$ is a sentence (terminal symbols only).

❗ **Definition 4.10:** A grammar $G = (V, T, \mathcal{P}, S)$ is called a **regular grammar** if it is a **right-regular grammar** or a **left-regular grammar**

❗ **Theorem 4.11:** Let G be a regular grammar. A language L is generated by G if and only if L is a regular language.

♥ **Proof:** Two directions must be proved.

Direction 1: Without loss of generality, suppose that G is a right-regular grammar. We must prove that $L(G)$ is a regular language. For this purpose, it suffices to construct a generalized automaton M that accepts $L(G)$. Denote the set of variables of G by $V = \{q_0, q_1, q_2, \dots, q_n\}$, where $S = q_0$ is the start variable. Define a transition function δ as follows: for every production of the form

$$q_i \rightarrow wq_j$$

define

$$\delta(q_i, w) = q_j$$

and for every production of the form

$$q_i \rightarrow w$$

define

$$\delta(q_i, w) = q_f$$

where q_f is the sole accepting state of the automaton M . We prove that $L(M) = L(G)$. Let w be a sentence accepted by the automaton M . Then there is a computation sequence

$$q_0 \xrightarrow{w_0} q_{i_1} \xrightarrow{w_1} q_{i_2} \xrightarrow{w_2} q_{i_3} \cdots q_{i_{k-1}} \xrightarrow{w_{k-1}} q_{i_k} \xrightarrow{w_k} q_f$$

such that $w = w_0 w_1 w_2 \cdots w_k$, $0 \leq i_0, i_2, \dots, i_k \leq n$.

By the definition of δ , we can write

$$q_0 \Rightarrow w_0 \mathbf{q}_{i_1} \Rightarrow w_0 \underline{w_1 \mathbf{q}_{i_2}} \Rightarrow w_0 w_1 \underline{w_2 q_{i_3}} \Rightarrow \cdots \Rightarrow w_0 w_1 w_2 \cdots w_k = w$$

We have thus obtained a derivation sequence for w . That is, $w \in L(G)$.

Now suppose that $w \in L(G)$ and prove that $w \in L(M)$. The assumption $w \in L(G)$ implies that there is a derivation sequence

$$q_0 \Rightarrow \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_k = w$$

Since G is a right-regular grammar, α_0 must have the form $w_0 q_{i_1}$ for $w_0 \in T^*$, $q_{i_1} \in V$. The other possibility allowed by a right-regular grammar is that $\alpha_0 = w$, in which case we are done. Similarly, the transition from α_0 to α_1 must have the form $w_0 q_{i_1} \Rightarrow w_0 w_1 q_{i_2}$, because, by definition, the only way to advance in the derivation sequence is by a production of the form $q_{i_1} \Rightarrow w_1 q_{i_2}$. Continuing in this way to the end, we find that our derivation sequence must have the following form

$$q_0 \Rightarrow w_0 \mathbf{q}_{i_1} \Rightarrow w_0 \underline{w_1 \mathbf{q}_{i_2}} \Rightarrow w_0 w_1 \underline{w_2 q_{i_3}} \Rightarrow \cdots \Rightarrow w_0 w_1 w_2 \cdots w_k = w$$

As we saw above, this sequence is equivalent to a computation sequence in the generalized automaton M

$$q_0 \xrightarrow{w_0} q_{i_1} \xrightarrow{w_1} q_{i_2} \xrightarrow{w_2} q_{i_3} \cdots q_{i_{k-1}} \xrightarrow{w_{k-1}} q_{i_k} \xrightarrow{w_k} q_f$$

Therefore, we have obtained $w \in L(M)$.

Direction 2: Suppose that L is a regular language over an alphabet T . We must prove that there is a regular grammar generating L . By the preceding chapters, we may assume that there is a nondeterministic automaton M with a start state q_0 having no incoming edges and a single accepting state q_f having no outgoing edges, such that $L = L(M)$. Define the grammar G by the following components

- a. Our set of terminal symbols is, of course, the alphabet T .



- b. Our set of variables is the set of states of M

$$V = \{q_0, q_1, q_2, \dots, q_n, q_f\}$$

- c. For every assignment $\delta(w, q_i) = q_j$ of the transition function δ , define a production $q_i \Rightarrow wq_j$.
- d. For every assignment $\delta(w, q_i) = q_f$ of the transition function δ , define a production $q_i \Rightarrow w$.

By construction, it is clear that G is a right-regular grammar. We leave it to the reader to verify that the language generated by G is the language L . ■

❖ **Example 4.9:** The following right-regular grammar is given

$$\left\{ \begin{array}{l} V = \{S, A, B\} \\ T = \{a, b, c\} \end{array} \right\} \left\{ \begin{array}{l} S \rightarrow bcA \\ A \rightarrow aacA \\ A \rightarrow cbbbB \\ B \rightarrow aaccA \\ B \rightarrow bbc \end{array} \right.$$

Construct a generalized automaton M equivalent to the grammar G , and find a corresponding regular expression for the language $L(G)$.

🔴 **Solution:** Using the construction presented in the proof, it is easy to obtain the following result

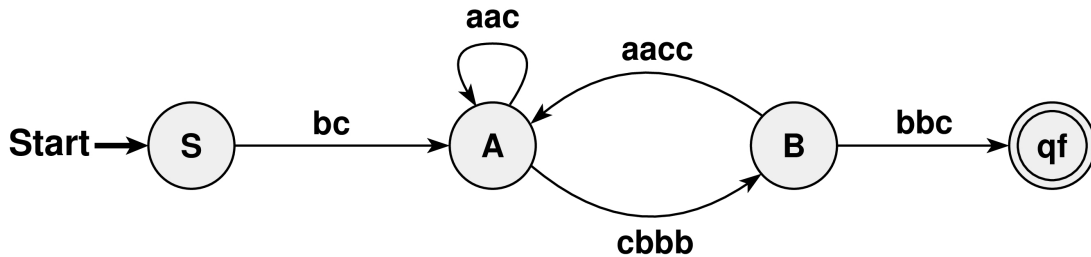


Fig. 4.1: A generalized automaton M equivalent to the right-regular grammar G

It is also easy to obtain the following regular expression for the language $L(G)$

$$bc(aac)^*cbbb(aaccbbb)^*bbc$$

❖ **Example 4.10:** Construct a right-regular grammar for the language accepted by the following automaton

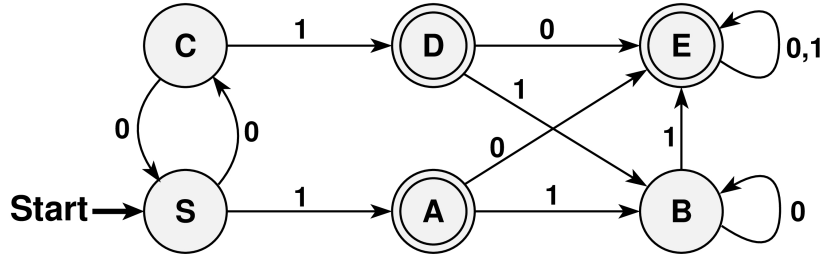


Fig. 4.2: Convert the following deterministic automaton into an equivalent automaton with a minimal number of states

Using the algorithm for converting an automaton to a grammar presented in the proof of the last theorem, it is easy to obtain the following result

$$\begin{aligned}
 S &\rightarrow 1|1A \\
 A &\rightarrow 0|0E|1B \\
 B &\rightarrow 1|1E|0B \\
 C &\rightarrow 1|1D|0S \\
 D &\rightarrow 0|0E|1B \\
 E &\rightarrow 0|1|0E|1E
 \end{aligned}$$

4.4 Derivation Trees and Derivation Sequences

- Let G be a context-free grammar. Every sentence in the language $L(G)$ has a corresponding **derivation tree** that reflects the syntactic structure of the sentence more clearly and compactly than a derivation sequence.
- The additional information contained in derivation trees is especially important for applications such as programming-language compilers and the many kinds of scanners.
- Before turning to the formal definition, let us consider a simple example. Let G be the following grammar

$$\begin{aligned}
 V &= \{S\} \\
 T &= \{a, b, +, *, (,)\} \\
 S &\rightarrow S + S \\
 S &\rightarrow S * S \\
 S &\rightarrow (S) \\
 S &\rightarrow a|b|c
 \end{aligned}$$

We saw earlier that the sentence $a * (b + c)$ belongs to the language $L(G)$. The derivation tree of this sentence is shown in Figure 4.3.

- Every node in the tree is a variable, a terminal symbol, or sometimes ϵ .

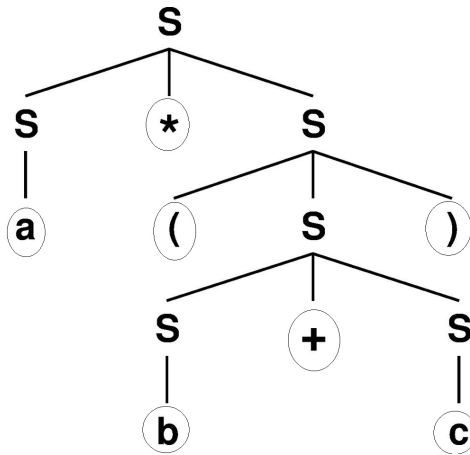
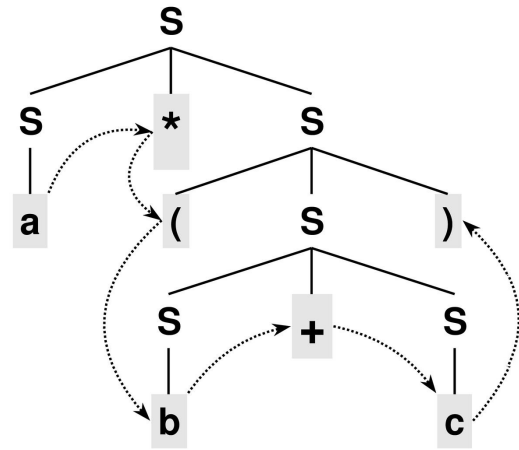
Fig. 4.3: A derivation tree for the sentence $a * (b + c)$ 

Fig. 4.4: Reconstructing a sentence from its derivation tree

- Every node that has children is called an **internal node**.
Every node without children is called a **leaf**.
Every tree has exactly one node that is not the child of any other node. This node is called the **root of the tree**. In a derivation tree, the root is always S .
- Every internal node in the tree must be a variable.
Every leaf in the tree must be a terminal symbol (or ε).
- If A is an internal node in the tree, and $X_1, X_2, \dots, X_k$ are the children of A in order from left to right, then

$$A \rightarrow X_1 X_2 \cdots X_n$$

must be a production.

- If we concatenate all the leaves in the tree in their order from left to right, we obtain the sentence $a * (b + c)$:

i **Definition 4.11:** Let $G = (V, T, P, S)$ be a context-free grammar. A tree is a **derivation tree (parse tree)** of G if

- a. Every node in the tree belongs to the set $V \cup T \cup \{\varepsilon\}$.
- b. The root of the tree is S .
- c. Every internal node in the tree is a variable.
- d. If A is an internal node in the tree, and $X_1, X_2, \dots, X_k$ are the children of A in order from left to right, then

$$A \rightarrow X_1 X_2 \cdots X_n$$

must be a production of G .

- e. If one of the nodes in the tree is ε , then this node is a leaf and is its parent's only child.
The sentence obtained by concatenating all the leaves of a derivation tree (from left to right) is called the **yield of the derivation tree**.

➤ Another example: Let G be the following grammar

$$\begin{array}{ll}
 S \rightarrow 0B & A \rightarrow 1BB \\
 S \rightarrow 1A & B \rightarrow 1 \\
 A \rightarrow 0 & B \rightarrow 1S \\
 A \rightarrow 0S & B \rightarrow 0BB
 \end{array}$$

The derivation tree of the sentence **001101** is shown in Figure 4.5.

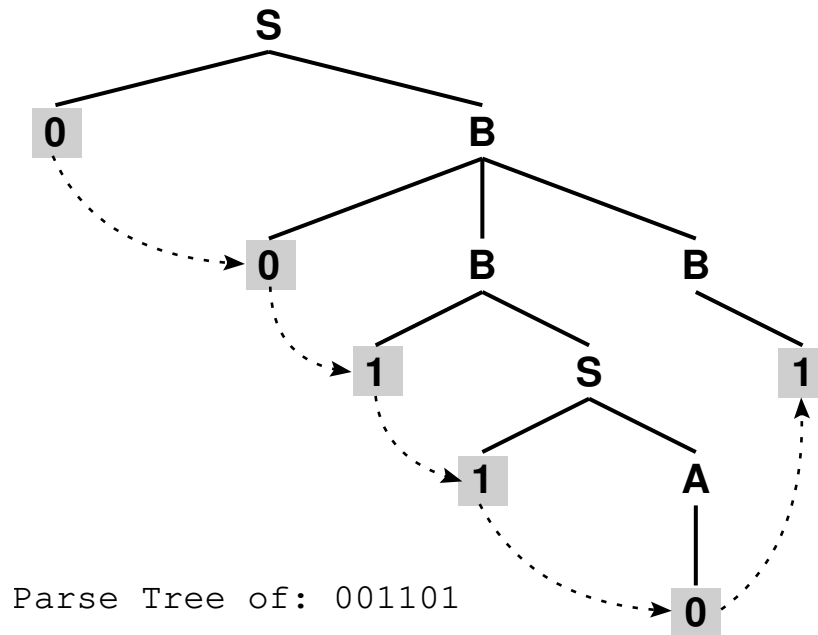


Fig. 4.5: The derivation tree of the sentence 001101

➤ There are two possible derivation sequences for the sentence 001101:

$$\begin{array}{l}
 S \Rightarrow 0\textcolor{red}{B} \Rightarrow 00\textcolor{red}{B}B \Rightarrow 001\textcolor{red}{S}B \Rightarrow 001\textcolor{red}{S}1 \Rightarrow 0011\textcolor{red}{A}1 \Rightarrow 001101 \\
 S \Rightarrow 0\textcolor{red}{B} \Rightarrow 00\textcolor{red}{B}B \Rightarrow 00\textcolor{red}{B}1 \Rightarrow 001\textcolor{red}{S}1 \Rightarrow 0011\textcolor{red}{A}1 \Rightarrow 001101
 \end{array}$$

In a sense, these two sequences are folded into the derivation tree. Therefore, a derivation tree is generally more informative than a derivation sequence.

4.5 The Relationship Between Derivation Trees and Derivation Sequences

➤ Two immediate questions arise concerning derivation trees:

- For every sentence $w \in L(G)$, is there a derivation tree whose yield is w ?
- If w is the yield of a certain derivation tree, is it necessarily true that $w \in L(G)$?

The answer to both questions is, of course, affirmative. To prove this, we need a few more concepts.

i Definition 4.12: The **height** of a tree is a natural number indicating the length of the longest path in the tree.

i Definition 4.13: A **subtree** of a derivation tree is any part of the derivation tree consisting of a particular node together with all its descendants.

➤ A subtree looks exactly like a derivation tree, except that its root need not be S .

➤ **Example 4.11:**

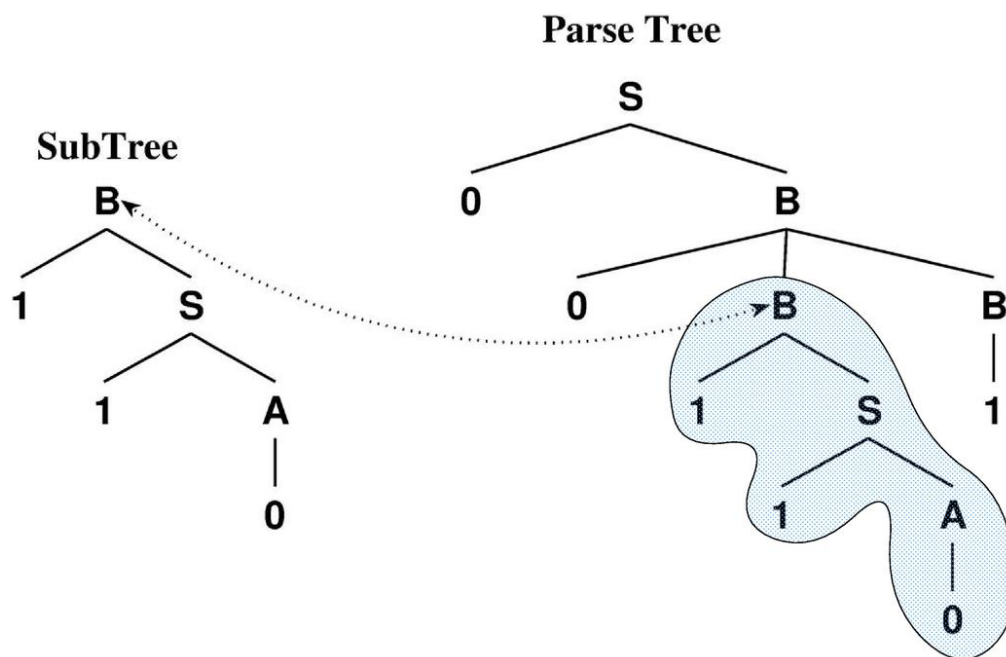


Fig. 4.6: A derivation tree and a derivation subtree

The height of the derivation tree is 6, and the height of the subtree is 4.

- Every node A in a derivation tree corresponds to a subtree whose root is A .
- **Terminology:** Every tree whose root is A is called an **A -tree**.
- All the concepts that apply to an ordinary derivation tree also apply to an A -tree (where A may be any symbol in the set $V \cup T$, or even ϵ).
- For example, the yield of an A -tree is the sentence obtained by concatenating all the leaves of the tree from left to right.

i Theorem 4.12: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar. Then $S \xRightarrow{*} w$ if and only if there is a derivation tree whose yield is w .

➤ This theorem is an immediate consequence of an even stronger theorem.

i Theorem 4.13: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar. Then $A \xRightarrow{*} w$ if and only if there is an A -tree whose yield is w .

✓ Proof: First, denote the two assertions appearing in the theorem as follows

a. There is an A -tree whose yield is w .

b. $A \xRightarrow{*} w$

We must prove two things

$$a \Leftarrow b, \quad b \Leftarrow a$$

$b \Leftarrow a$:

By induction on the height of the tree. Suppose that there is an A -tree whose yield is w . We prove that $A \xRightarrow{*} w$. If the A -tree has height 1, this is immediate.

Induction hypothesis: for every A -tree of height at most n , if u is the yield of the tree, then $A \xRightarrow{*} u$.

Let τ be an A -tree of height $n + 1$, and suppose that w is the yield of τ . We must prove that $A \xRightarrow{*} w$.

Let $B_1, B_2, \dots, B_k$ be all the children of A in the tree τ , and let $\tau_1, \tau_2, \dots, \tau_k$ be the subtrees corresponding to these children. Clearly, the height of each of these trees is at most n .

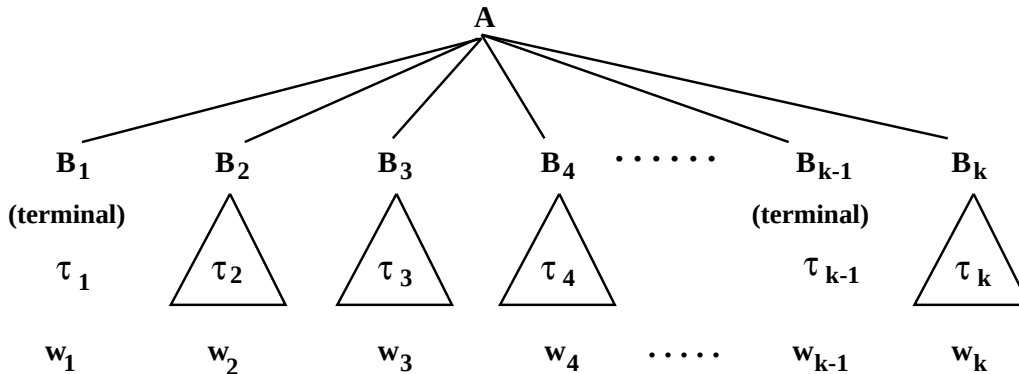


Fig. 4.7: Decomposing a derivation tree into k subtrees

For every $1 \leq i \leq k$, let w_i be the yield of τ_i . By the induction hypothesis, $B_i \xRightarrow{*} w_i$. It is also clear that

$$w = w_1 w_2 w_3 \cdots w_k$$

and therefore

$$B_1 B_2 B_3 \cdots B_k \xRightarrow{*} w_1 w_2 w_3 \cdots w_k = w$$

By the definition of a derivation tree,

$$A \xRightarrow{*} B_1 B_2 B_3 \cdots B_k$$

and therefore we have obtained $A \xRightarrow{*} w$.

$b \Leftarrow a$:

By induction on the length of the derivation sequence. Suppose that $A \xRightarrow{*} w$. We prove that there is an A -tree whose yield is w .

Induction basis: if $n = 2$, then $A \rightarrow w$, so this is in fact a production. Suppose that $w = a_1 a_2 \cdots a_k$ (every a_i is a terminal symbol).

Induction hypothesis: if the length of the derivation sequence for $A \xRightarrow{*} u$ is at most n , then there is an A -tree whose yield is u .

Now suppose that $A \xRightarrow{*} w$ is a derivation of length $n + 1$. Let

$$A \rightarrow X_1 X_2 \cdots X_n$$

be the first production in the sequence. By a preceding theorem, there is a decomposition

$$w = w_1 w_2 \cdots w_k$$

such that

$$\begin{array}{ccc} X_1 & \xRightarrow{*} & w_1 \\ X_2 & \xRightarrow{*} & w_2 \\ & \vdots & \\ X_k & \xRightarrow{*} & w_k \end{array}$$

where each of the derivation sequences has length at most n . Therefore, by the induction hypothesis, for every $1 \leq i \leq k$, there is an X_i -tree whose yield is w_i . Denote this tree by τ_i . Define an A -tree by "gluing" all these trees beneath the root A , as shown in the following figure

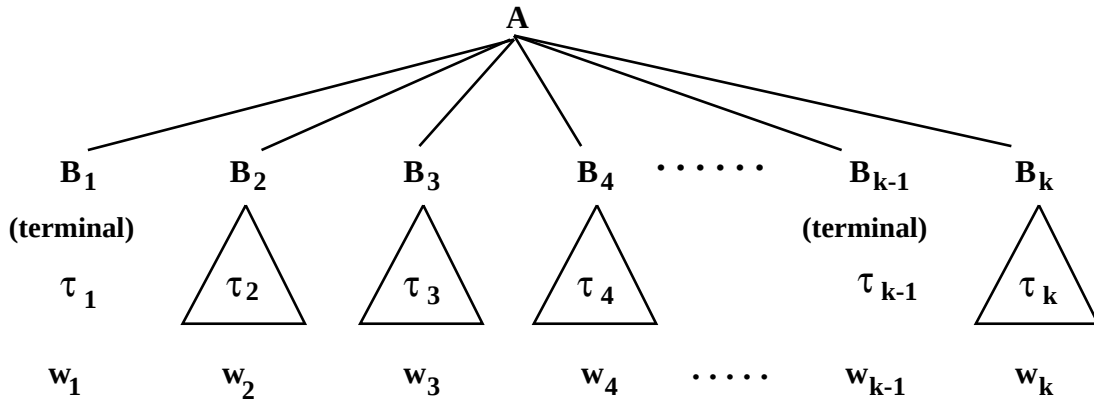


Fig. 4.8: Gluing k trees to a common root

Clearly, we have obtained a valid A -tree whose yield is w . ■

- **Conclusion:** $w \in L(G)$ if and only if there is a derivation tree whose yield is w .
- The conclusion follows by substituting $A = S$ in the last theorem.

4.6 Ambiguity

i Definition 4.14: Let

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_n$$

be a given derivation. If the transition from one sentential form to the next is made by applying the corresponding production to the leftmost variable, then the derivation is called a **leftmost derivation**.

➤ A **rightmost derivation** is defined similarly.

◆ **Example 4.12:** Let G be the following grammar

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow aBC|bc \\ B &\rightarrow AbC|ac \\ C &\rightarrow ABc|ab \end{aligned}$$

A leftmost derivation:

$$\begin{aligned} S &\Rightarrow \textcolor{red}{A}BC \Rightarrow \textcolor{red}{a}\textcolor{blue}{B}\textcolor{blue}{C}BC \Rightarrow \textcolor{red}{aa}\textcolor{red}{c}\textcolor{red}{C}BC \Rightarrow \textcolor{red}{aacab}\textcolor{red}{B}C \Rightarrow \\ &\Rightarrow \textcolor{red}{aacabac}\textcolor{red}{C} \Rightarrow \textcolor{red}{aacabacab} \end{aligned}$$

A rightmost derivation:

$$S \Rightarrow AB\textcolor{red}{C} \Rightarrow A\textcolor{red}{B}\textcolor{blue}{ab} \Rightarrow \textcolor{red}{A}\textcolor{blue}{acab} \Rightarrow \textcolor{blue}{bcacab}$$

- There is a relationship between these two concepts and the concept of a derivation tree.
- A sentence $w \in L(G)$ may have more than one derivation tree yielding it. However, every derivation tree τ of w corresponds to exactly one leftmost derivation and exactly one rightmost derivation.
- To justify the assertion made in the preceding item, we need a more general concept of a derivation tree.

i Definition 4.15: Let $G = (V, T, P, S)$ be a context-free grammar. A **partial derivation tree** of G is a tree satisfying all the conditions of an ordinary derivation tree except the condition that the leaves must be terminal symbols.

- In other words, a partial derivation tree is a derivation tree “in formation” or “growing”. In a partial derivation tree, a leaf may also be a variable (which may later produce children and ultimately terminal symbols). In this case, the yield of the tree is a sentential form and not



necessarily a sentence.

- If τ is a derivation tree for a sentence w , then in general we can extract from τ a large number of different derivation sequences for w . The following assertion shows that among all these sequences, there is exactly one leftmost derivation.

i Proposition 4.14: Every derivation tree for a sentence w corresponds to exactly one leftmost derivation and exactly one rightmost derivation.

- Instead of giving a dry formal proof, we present a short algorithm with a suitable example.

The algorithm:

- ◆ Let τ be a derivation tree whose yield is the sentence w .
- ◆ **Step 1:** Let τ_1 be the partial derivation tree consisting only of the root S of τ . This is a tree with only one node.
- ◆ **Step 2:** Let τ_2 be the partial derivation tree obtained by adding all the children of S to the tree τ_1 .
- ◆ **Step $k + 1$:** In general, if at stage k we have a partial derivation tree τ_k , let α_k be the sentential form generated by τ_k . Scan the sentential form α_k from left to right until reaching the first variable X that occurs in it. Add all the children of X to the tree τ_k to obtain the next partial tree τ_{k+1} .
- ◆ The algorithm terminates at the first point at which $\tau_n = \tau$.
- The required leftmost derivation is

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_n$$

where α_i is the yield of the partial tree τ_i .

- There is an entirely analogous algorithm for the rightmost derivation.

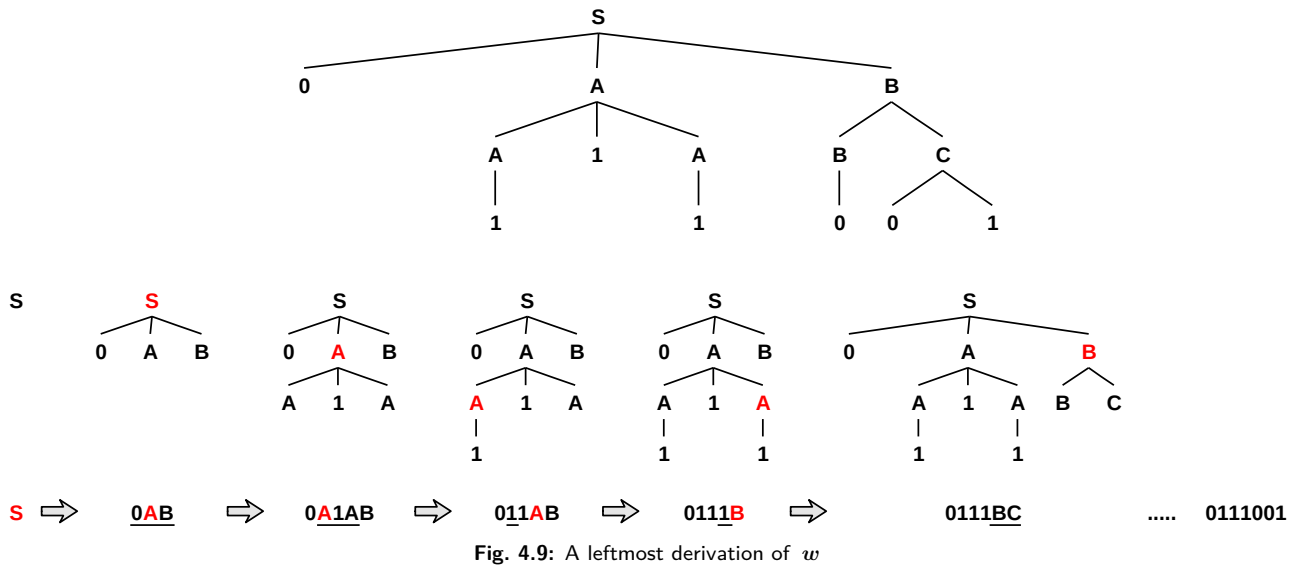
◆ **Example 4.13:** Let G be the following grammar

$$\begin{aligned} S &\rightarrow 0AB \\ A &\rightarrow A1A|1 \\ B &\rightarrow BC|0 \\ C &\rightarrow 01 \end{aligned}$$

The following is a derivation tree for $w = 0111001$, together with some of the partial trees corresponding to the leftmost derivation of w

- Since, by a preceding theorem, every sentence w in the language $L(G)$ has at least one derivation tree, we obtain the following immediate conclusion.





Theorem 4.15: Every sentence w has at least one leftmost derivation and at least one rightmost derivation.

- There is a simple way to construct a derivation tree from a given leftmost (or rightmost) derivation. We leave the details as an exercise.
- Therefore, the number of derivation trees for a given sentence w equals the number of leftmost derivations of w (or the number of rightmost derivations).
- However, since a sentence w may have more than one derivation tree (an example follows), it is possible for the same sentence to have more than one leftmost derivation. A grammar in which this occurs is an "undesirable" grammar.

Definition 4.16: A context-free grammar G is called **ambiguous** if there is a sentence $w \in L(G)$ for which there are two different derivation trees.

- Every grammar that is not ambiguous is called an **unambiguous grammar**.
- By the discussion above, a grammar G is ambiguous if and only if there is a sentence $w \in L(G)$ for which there are two different leftmost derivations.
- **Remark:** The existence of two different derivation sequences for w is not sufficient to conclude that G is ambiguous!

Definition 4.17: A context-free language L is called **inherently ambiguous** if every context-free grammar generating L is ambiguous.

- As we have seen, a context-free language may be generated by more than one grammar. It is easy to find an example of a context-free language generated by an unambiguous grammar and also by an ambiguous grammar.

❖ **Example 4.14:**

$$S \rightarrow 0S1$$

$$S \rightarrow 01$$

We saw that this grammar generates the language

$$L = \{0^n 1^n : n = 1, 2, 3, \dots\}$$

We also saw that every sentential form derived from S contains at most one variable. Therefore, every derivation in this grammar is both a leftmost and a rightmost derivation. Thus, L is an unambiguous language. The following is an ambiguous grammar generating L

$$S \rightarrow ASB$$

$$S \rightarrow 0S1|01$$

$$A \rightarrow 0$$

$$B \rightarrow 1$$

This grammar is ambiguous because, for example, the sentence **0011** has two different leftmost derivations

$$S \Rightarrow ASB \Rightarrow 0SB \Rightarrow 001B \Rightarrow 0011$$

$$S \Rightarrow 0S1 \Rightarrow 0011$$

- ❖ Equivalently, one can prove that a grammar is ambiguous by finding two different derivation trees for the same sentence. Figure 4.10 shows two different derivation trees for the sentence **0011**.

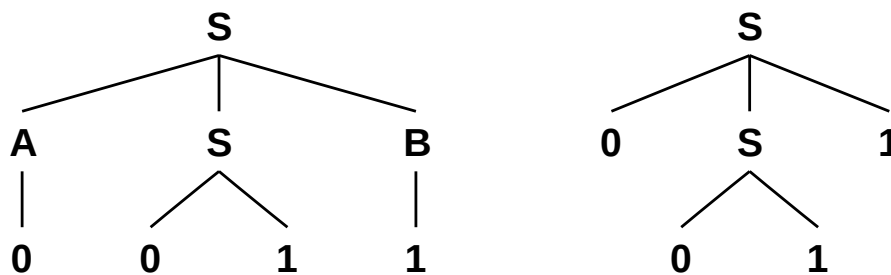


Fig. 4.10: Two different derivation trees for the sentence **0011**

- ❖ There are inherently ambiguous languages. That is, every grammar generating them is necessarily ambiguous. However, proving such a fact is not simple.
- ❖ Unambiguous grammars are very important in the field of scanners and compilers that use derivation trees. It is essential that every sentence (a program or a block of a program) have exactly one derivation tree for the compiler to operate correctly.



4.7 Useful and Useless Variables

i Definition 4.18: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar. A variable X is called a *useful variable* of G if there is a derivation

$$S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w$$

where α, β are sentential forms and w is a sentence.

If no such derivation exists, then X is called a *useless variable*.

➤ In other words, a useful variable is a variable that occurs in some derivation beginning with the variable S and ending with a sentence w .

➤ **Example 4.15:** Let G_1 be the grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow AA|1 \\ B &\rightarrow BB|0 \\ C &\rightarrow AC|01 \end{aligned}$$

It is easy to see that C is a useless variable, whereas A and B are useful variables.

➤ If we remove the last production, we obtain a smaller grammar G_2

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow AA|1 \\ B &\rightarrow BB|0 \end{aligned}$$

that is equivalent to the grammar G_1 . That is, $L(G_1) = L(G_2)$.

- In this example, identifying the useless variable was not difficult. There are examples in which such a task may be very difficult.
- Is there an algorithm for identifying all the useless variables in a given grammar?
- If so, we can remove all the unnecessary productions and obtain an equivalent grammar that is simpler than the original grammar!
- This fact is very important for writing smaller and faster compilers or scanners.



◆ **Example 4.16:** Let G be the grammar

$$\begin{aligned} S &\rightarrow AB|AC \\ A &\rightarrow AA|111 \\ B &\rightarrow BB|000 \\ C &\rightarrow A01C \end{aligned}$$

Here too, C is a useless variable. No sentence can be derived from a sentential form containing C . Therefore, this grammar can be simplified to the following equivalent grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow AA|111 \\ B &\rightarrow BB|000 \end{aligned}$$

◆ **Example 4.17:** Let G be the grammar

$$\begin{aligned} S &\rightarrow AB|C|D \\ A &\rightarrow AA|11 \\ B &\rightarrow BB|00 \\ C &\rightarrow A0D \\ D &\rightarrow B1C \end{aligned}$$

Useful variables: A, B .

Useless variables: C, D .

Simplification:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow AA|11 \\ B &\rightarrow BB|00 \end{aligned}$$

◆ **Example 4.18:**

$$\begin{aligned} S &\rightarrow AB|CD \\ A &\rightarrow AA|11 \\ B &\rightarrow BB|00 \\ C &\rightarrow A|10 \\ D &\rightarrow DD \end{aligned}$$

Here too, the useless variables are C, D . However, in this example, note that there is a derivation beginning with a sentential form containing C and ending with a sentence

$$ABC \xRightarrow{*} 110010$$



and there is a derivation beginning with the variable S and ending with a sentential form containing C

$$S \xRightarrow{*} CD$$

Nevertheless, C is a useless variable! Why does this not contradict the definition?

- We now turn to finding an algorithm that converts a given grammar into an equivalent grammar with no useless variables. For convenience, we divide the task into two easier subtasks.

i Proposition 4.16: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar such that

$$L(G) \neq \emptyset$$

There is an algorithm for finding an equivalent grammar $G' = (V', T, \mathcal{P}', S)$ in which, for every variable $A \in V'$, there is a sentence $w \in T^*$ such that $A \xRightarrow{*} w$.

The Algorithm

- We describe one of the simpler algorithms (although it is not especially efficient). We construct the set V' in a finite number of stages by constructing an increasing sequence of sets of variables

$$V_1 \subset V_2 \subset V_3 \subset \cdots \subset V_n = V'$$

At each stage, we add more variables having the desired property. The process stops as soon as no new variables are added.

1. V_1 is the set of all variables A for which there is a production $A \rightarrow w$, where w is a sentence.
2. Suppose that we have already constructed the set V_k . The set V_{k+1} consists of all variables A for which there is a production

$$A \rightarrow \alpha$$

where α is a sentential form satisfying

$$\alpha \in (T \cup V_k)^*$$

3. The process ends at the first point at which $V_n = V_{n+1}$.
When we reach such a stage, define $V' = V_n$ and proceed to the final stage.
 4. Finally, we must define the new set of productions $\mathcal{P}'$. The set $\mathcal{P}'$ consists of all productions in $\mathcal{P}$ in which every symbol belongs to the set $T \cup V'$.
- The new grammar is therefore $G' = (V', T, \mathcal{P}', S)$. The set of terminal symbols T and the start variable S remain unchanged.



◆ **Example 4.19:** Let G be the following grammar

$$\begin{aligned} S &\rightarrow AB11 \\ A &\rightarrow E1F \\ B &\rightarrow A0EE \\ C &\rightarrow EF10C \\ D &\rightarrow CEF|00 \\ E &\rightarrow 00F \\ F &\rightarrow 1 \end{aligned}$$

Apply the algorithm to G

$$\begin{aligned} V_1 &= \{D, F\} \\ V_2 &= \{D, F, E\} \\ V_3 &= \{D, F, E, A, B\} \\ V_4 &= \{D, F, E, A, B, S\} \end{aligned}$$

The grammar G' is therefore

$$\begin{aligned} T &= \{0, 1\} \\ V &= \{A, B, D, E, F, S\} \\ S &\rightarrow AB11 \\ A &\rightarrow E1F \\ B &\rightarrow A0EE \\ D &\rightarrow 00 \\ E &\rightarrow 00F \\ F &\rightarrow 1 \end{aligned}$$

The variable C and the productions

$$C \rightarrow EF10C, \quad D \rightarrow CEF$$

Nevertheless, the grammar G' is not entirely free of useless variables. It is easy to see that D is a useless variable of G' !

Proof of the Correctness of the Algorithm

- The running time of stage 1 is finite because the set of productions $\mathcal{P}$ is finite, and all we must do is identify productions of the form $A \rightarrow w$ and insert A into the set V_1 .
- The running time of stage 2 is also finite because the sets $\mathcal{P}$ and $(T \cup V_k)^*$ are finite, and we must identify within them productions of the form $A \rightarrow \alpha$, where α is a sentential form in



$$(T \cup V_k)^*.$$

- c. Similarly, it is clear that the running time of stage 4 is finite.
 - d. It remains to prove that the recursive stage 2 is performed a finite number of times. Clearly, at every stage k , $V_k \subseteq V_{k+1}$. Since at every stage k we also have $V_k \subseteq V$, the sequence must necessarily "get stuck" at some stage. That is, there is a stage n at which $V_n = V_{n+1}$. Therefore, the algorithm above always terminates after a finite computation time.
- ☛ All four items above prove only that our algorithm always terminates after a finite time. We must still prove the following two facts

- a. The grammar $G' = (V', T, P', S)$ has the desired property.
- b. The grammar G' is equivalent to the grammar G .

If A is a variable belonging to V' , then $A \in V_k$ for some stage k . If $k = 1$, then there is a production $A \rightarrow w$, and we are done.

If $k = 2$, then there is a production

$$A \rightarrow X_1 X_2 \cdots X_m$$

where every X_i is a terminal symbol or a variable belonging to V_1 . For every X_i belonging to V_1 , there is a production

$$X_i \rightarrow w_i$$

where w_i is a sentence. Therefore, if every occurrence of X_i (that is a variable) is replaced by the sentence w_i , we obtain $A \xRightarrow{*} w$, for a sentence w . The cases in which $A \in V_3$, and so on', are handled similarly.

We now prove that if, in the grammar G , $A \xRightarrow{*} w$, then necessarily $A \in V'$. We do so by induction on the length n of the derivation $A \xRightarrow{*} w$. If $n = 2$, then $A \rightarrow w$ is a production, so $A \in V_1$, and therefore $A \in V'$.

If $n > 2$, then

$$A \rightarrow X_1 X_2 \cdots X_m \Rightarrow \cdots \Rightarrow w$$

By a preceding theorem, there is a decomposition $w = w_1 w_2 \cdots w_m$ such that

$$\begin{array}{ccc} X_1 & \xRightarrow{*} & w_1 \\ X_2 & \xRightarrow{*} & w_2 \\ & \vdots & \\ X_m & \xRightarrow{*} & w_m \end{array}$$

and each of these derivations has length less than n . Therefore, by the induction hypothesis, if X_i is a variable, then it belongs to V' . Since $A \rightarrow X_1 \cdots X_m$, it follows that $A \in V'$, and we are done.



It remains to prove that $G' \equiv G$. First, it is clear that

$$L(G') \subseteq L(G)$$

because $\mathcal{P}' \subseteq \mathcal{P}$ and also $V' \subseteq V$. Suppose, by way of contradiction, that there is a sentence $w \in L(G)$ such that $w \notin L(G')$. Then every derivation $S \xRightarrow{*} w$ must involve at least one variable from the set $V - V'$, or a production from the set $\mathcal{P} - \mathcal{P}'$ (and hence also a variable from the set $V - V'$). Denote this variable by X .

$$S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w$$

It is clear from this that there is a substring u of w such that $X \xRightarrow{*} u$. Therefore, $X \in V'$. A contradiction. ■

- So far, we have completed only half the task, because the new grammar G' may still contain useless variables (as we saw in the example above). However, only a short step remains to complete the full task.

i Proposition 4.17: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar. There is an algorithm for finding an equivalent grammar $G' = (V', T, \mathcal{P}', S)$ in which, for every symbol $X \in V' \cup T'$, there are sentential forms

$$\alpha, \beta \in (V' \cup T')^*$$

such that $S \xRightarrow{*} \alpha X \beta$.

- ✓ **Proof:** We construct the sets V' and T' in stages, as in the preceding assertion. This time, we construct two sequences: one for V' and the other for T' .

1. $V_1 = \{S\}$. T_1 is the set of all terminal symbols occurring in productions of the form $S \rightarrow \alpha$.
2. Suppose that we have already constructed the sets V_k and T_k . For every $A \in V_k$ and every production $A \rightarrow \alpha$, insert all the variables occurring in the sentential form α into the set V_{k+1} , and all the terminal symbols occurring in α into the set T_{k+1} . Include the set V_k in V_{k+1} and the set T_k in T_{k+1} .
3. The process ends at the first stage n at which $V_n = V_{n+1}$ and $T_n = T_{n+1}$.
When we reach such a stage, define

$$\begin{aligned} V' &= V_n \\ T' &= T_n \end{aligned}$$

and proceed to the final stage.

4. Finally, we must define the new set of productions $\mathcal{P}'$. The set $\mathcal{P}'$ consists of all productions



in $\mathcal{P}$ in which every symbol belongs to the set $T' \cup V'$.

We leave the details of the proof of correctness of the algorithm as an exercise.

❖ **Example 4.20:** Apply the algorithm to the grammar obtained in the preceding example

$$\begin{aligned}
 T &= \{0, 1\} \\
 V &= \{A, B, D, E, F, S\} \\
 S &\rightarrow AB11 \\
 A &\rightarrow E1F \\
 B &\rightarrow A0EE \\
 D &\rightarrow 00 \\
 E &\rightarrow 00F \\
 F &\rightarrow 1
 \end{aligned}$$

and obtain

$$\begin{aligned}
 V_1 &= \{S\} & T_1 &= \{1\} \\
 V_2 &= \{S, A, B\} & T_2 &= \{1, 0\} \\
 V_3 &= \{S, A, B, E, F\} & T_3 &= \{1, 0\}
 \end{aligned}$$

Therefore, the new grammar equivalent to the preceding one is

$$\begin{aligned}
 T' &= \{0, 1\} \\
 V' &= \{A, B, E, F, S\} \\
 S &\rightarrow AB11 \\
 A &\rightarrow E1F \\
 B &\rightarrow A0EE \\
 E &\rightarrow 00F \\
 F &\rightarrow 1
 \end{aligned}$$

The variable D and the production $D \rightarrow 00$ have disappeared. It is easy to see that this grammar has no remaining useless variables.

❗ **Theorem 4.18:** Every context-free language is generated by a context-free grammar with no useless variables.

✔ **Proof:** Let L be a context-free language. Then there is a context-free grammar G that generates L . Let G_1 be the grammar obtained by applying Assertion 4.16 to G , and let G_2 be the grammar obtained by applying Assertion 4.17 to G_1 . Then $G_2 \equiv G$. We prove that the grammar G_2 has no useless variables.

Suppose, by way of contradiction, that X is a useless variable of G_2 . By Auxiliary Claim 2, the

grammar G_2 has a derivation

$$S \xRightarrow{*} \alpha X \beta$$

Since all the symbols of G_2 are among the symbols of G_1 , Auxiliary Claim 1 implies that there is a sentence w such that

$$S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w \quad (\text{in } G_1)$$

Clearly, all the symbols occurring in this derivation belong to the grammar G_2 , and therefore all the productions involved in this derivation also belong to the grammar G_2 . Thus, this derivation is valid in the grammar G_2 as well, and therefore X is a useful variable of G_2 . A contradiction. ■

- In summary: to convert a given grammar G into a grammar with no useless variables, apply the first algorithm to G , and then apply the second algorithm to the resulting grammar.
- Note that the order in which the algorithms are applied is important! If we apply the second algorithm first and then the first algorithm, we do not necessarily obtain a grammar with no useless variables.
- ◆ **Example 4.21:** Let G be the following grammar

$$\begin{aligned} S &\rightarrow AB|a \\ A &\rightarrow a \end{aligned}$$

After applying the first algorithm, we obtain the equivalent grammar

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow a \end{aligned}$$

After applying the second algorithm, we obtain the grammar

$$S \rightarrow a$$

If we apply the second algorithm first, we again obtain G itself as the result, and applying the first algorithm yields

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow a \end{aligned}$$

which is not a desired result, since A is a useless variable.



4.8 ϵ -Productions

- Our goal now is to continue the process of simplifying a context-free grammar by removing productions of the form

$$A \rightarrow \epsilon$$

Productions of this kind are called **ϵ -productions**.

- ϵ -productions are undesirable because they present a serious obstacle to the existence of various algorithms. They cause unnecessary proliferation of derivations and derivation trees, thereby making search processes longer and more complicated.

◆ **Example 4.22:**

$$\begin{aligned} S &\rightarrow ABa \\ A &\rightarrow BBC|bb \\ B &\rightarrow \epsilon \\ C &\rightarrow AAb \end{aligned}$$

The sentence ϵ does not belong to the language $L(G)$ (why?). Therefore, the production $B \rightarrow \epsilon$ is unnecessary.

Simplification

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow C|bb \\ C &\rightarrow AAb \end{aligned}$$

◆ **Example 4.23:**

$$\begin{aligned} S &\rightarrow ABa \\ A &\rightarrow BBC|bb \\ B &\rightarrow \epsilon \\ B &\rightarrow ab \\ C &\rightarrow AAb \end{aligned}$$

In this example too, we can see that $\epsilon \notin L(G)$. However, care must be taken in the simplification process! The production $B \rightarrow ab$ cannot be removed! The treatment of the productions $A \rightarrow BBC$ and $S \rightarrow ABa$ is also delicate.

The equivalent grammar is

$$\begin{aligned} S &\rightarrow ABa|Aa \\ A &\rightarrow BBC|BC|C|bb \\ B &\rightarrow ab \\ C &\rightarrow AAb \end{aligned}$$

- Clearly, if $\epsilon \in L(G)$, then we cannot remove all productions of this kind. However, if $\epsilon \notin L(G)$,

it turns out that there is an algorithm for converting every grammar with ϵ -productions into an equivalent grammar with no ϵ -productions.

- The following is an example in which an ϵ -production is necessary

$$S \rightarrow aSb | SS | \epsilon$$

In this case, the production $S \rightarrow \epsilon$ cannot be removed because $\epsilon \in L(G)$. This grammar defines the language of strings of balanced parentheses (a represents a left parenthesis and b a right parenthesis). The production $S \rightarrow \epsilon$ has an important role: to end a derivation elegantly. For example,

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaSSbb \Rightarrow aaaSbaSbbb \xRightarrow{*} aaababbb$$

- In such cases, we prefer to discard the sentence ϵ from the language $L(G)$ and find an equivalent grammar with no ϵ -productions for the language $L(G) - \epsilon$.
- The idea is to "capture" all variables for which there is a derivation $A \xRightarrow{*} \epsilon$, and then apply a simple procedure that we describe below.

i Definition 4.19: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar. A variable A is called a **nullable variable** of G if there is a derivation

$$A \xRightarrow{*} \epsilon$$

i Theorem 4.19: Let $G = (V, T, \mathcal{P}, S)$ be a context-free grammar, and let $L = L(G)$. Then there is a context-free grammar G' with neither nullable variables nor useless variables that generates the language $L - \{\epsilon\}$.

- ✓ **Proof:** We can find all the nullable variables using the following recursive process

1. If $A \rightarrow \epsilon$ is a production, then A is a nullable variable.
2. If $B \rightarrow X_1 X_2 \cdots X_k$ is a production, and every X_i is a nullable variable, then B is also a nullable variable.
3. Repeat the preceding stage recursively until all the nullable variables have been found.

Now, after identifying all the nullable variables, we must define the set of productions in the new grammar. The price of removing the ϵ -productions is a substantial increase in the number of productions in the new grammar.

For every production $A \rightarrow X_1 X_2 \cdots X_k$ in $\mathcal{P}$, we must add to $\mathcal{P}'$ all productions of the form

$$A \rightarrow Y_1 Y_2 \cdots Y_n$$



where

- a. If X_i is not a nullable variable, then $Y_i = X_i$.
- b. If X_i is a nullable variable, then Y_i may be X_i or ε .
- c. Do not include the case in which every X_i is ε .

Clearly, in this way no ε -production enters $\mathcal{P}'$.

For example, if $A \rightarrow BCD$ is a production in $\mathcal{P}$, where B, C, D are nullable variables, then all the following productions must be inserted into $\mathcal{P}'$

$$A \rightarrow BCD|BC|BD|B|C|D$$

If $A \rightarrow BaCD$ is a production in $\mathcal{P}$, where B, C, D are nullable variables and $a \in T$, then all the following productions must be inserted into $\mathcal{P}'$

$$A \rightarrow BaCD|BaC|BaD|Ba|aC|aD|a$$

Define the resulting grammar

$$G'' = (V, T, \mathcal{P}', S)$$

The only thing that has changed so far is the set of productions $\mathcal{P}'$! By induction on the length of the derivation, one can prove the following assertion:

for every variable A and every sentence w , the grammar G'' has a derivation $A \xRightarrow{*} w$ if and only if $w \neq \varepsilon$ and the grammar G has the derivation $A \xRightarrow{*} w$.

In other words:

$$L(G'') = L(G) - \{\varepsilon\}$$

We leave the details of the proof as an exercise.

The final step in the proof of the theorem is to apply to G'' the algorithm for eliminating all useless variables. There is no danger that this will reintroduce ε -productions into the grammar, because the algorithm for removing useless variables does not add any productions! (On the contrary, it removes productions). ■

4.9 Unit Productions

- Every production of the form $A \rightarrow B$ is called a **unit production**.
- A production $A \rightarrow a$, where a is a symbol, or $A \rightarrow \varepsilon$, is not a unit production!
- Our next goal is to simplify a grammar by removing all unit productions.
- In a unit production $A \rightarrow B$, the variable B is, in a certain sense, an "unnecessary intermediary" that slows the derivation.

◆ **Example 4.24:**

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow B \\ B &\rightarrow CD \\ C &\rightarrow b \\ D &\rightarrow aa \end{aligned}$$

Simplification:

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow CD \\ C &\rightarrow b \\ D &\rightarrow aa \end{aligned}$$

i Theorem 4.20: Every context-free language not containing ε is generated by a context-free grammar with no useless variables, no ε -productions, and no unit productions.

♥ **Proof:** Let L be a context-free language not containing ε , and let $G = (V, T, P, S)$ be a context-free grammar generating L . By the preceding theorem, we may assume that G contains no ε -production. Define a new set of productions P'

1. Every production in P that is not a unit production is inserted into P' .
2. For every derivation $A \xRightarrow{*} B$ and every production $B \rightarrow \alpha$ that is not a unit production, insert the production $A \rightarrow \alpha$ into P' .

The only problem that may arise with this algorithm is that the search in stage 2 for derivations of the form $A \xRightarrow{*} B$ may be infinite. However, the problem does not arise, because the grammar G contains no ε -productions! This fact implies that every derivation $A \xRightarrow{*} B$ must have the following form

$$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_m \Rightarrow B$$

where every B_i is a variable. We may assume that no variable occurs twice in the sequence above. Since the number of variables in the grammar G is a fixed finite number k , the number of derivations of this kind is at most k^k . Therefore, the running time of stage 2 is finite.

Define the new grammar $G' = (V, T, P', S)$. We must prove that $G' \equiv G$. That is, $L(G') = L(G)$.

Clearly, if $A \Rightarrow \alpha$ is a production in P' , then $A \xRightarrow{*} \alpha$ in the grammar G . It follows immediately that if there is a derivation of a string w in the grammar G' , then we can easily convert it into a derivation in the grammar G . Therefore, $L(G') \subseteq L(G)$.

It remains to prove that $L(G) \subseteq L(G')$: let $w \in L(G)$, and let

$$S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n = w$$



i Theorem 4.21: (Chomsky Normal Form – CNF)

Every context-free language not containing ε is generated by a grammar in which all productions have the form $A \rightarrow BC$ or $A \rightarrow a$. Here A, B, C are variables, and a is a terminal symbol.

Proof

- Let L be a context-free language not containing ε , and let G be a context-free grammar generating it. We present an algorithm for converting the grammar G into a grammar in CNF.
- By the preceding theorems, we can construct an equivalent grammar $G_1 = (V, T, \mathcal{P}, S)$ with no unit productions or ε -productions. Thus, if a production has only one symbol on its right-hand side, that symbol must be a terminal symbol, and the production is therefore already in the desired form.
- Now let $A \rightarrow X_1 X_2 \cdots X_m$ be a production in $\mathcal{P}$, where $m \geq 2$. Such a production is not in the desired form, so we convert it in stages into several productions having the desired form. For every $0 \leq i \leq m$, if X_i is a terminal symbol a , add a new variable named C_a to the grammar, together with a new production $C_a \rightarrow a$ (which is, of course, in the desired form). Retain the production $A \rightarrow X_1 X_2 \cdots X_m$ for the moment, but replace X_i by the variable C_a (for each corresponding i).
- After performing all the operations above, we obtain a new set of variables V' and a new set of productions $\mathcal{P}'$. Call the new grammar G_2 ($G_2 = (V', T, \mathcal{P}', S)$).

i Proposition 4.22: The grammar G_2 is equivalent to the grammar G_1 .

- One direction of the proof of the assertion is clear: $L(G_1) \subseteq L(G_2)$. The other direction is easily proved by induction on the length of the derivation.
- At this stage, we have obtained an intermediate result: every context-free language is generated by a grammar in which all productions have the form $A \rightarrow a$ or $A \rightarrow B_1 B_2 \cdots B_m$, $m \geq 2$ (a is a terminal symbol and the B_i are variables).
- In the final stage, we convert the grammar G_2 into a grammar in Chomsky normal form. For this purpose, add new "intermediate variables" and replace some of the productions in $\mathcal{P}'$.
- For every production $A \rightarrow B_1 B_2 \cdots B_m$, $m \geq 3$, add $m - 2$ new variables $D_1, D_2, \dots, D_{m-2}$, and replace the production $A \rightarrow B_1 B_2 \cdots B_m$ with the following set of productions:

$$\begin{aligned}
 A &\rightarrow B_1 D_1 \\
 D_1 &\rightarrow B_2 D_2 \\
 D_2 &\rightarrow B_3 D_3 \\
 &\vdots \\
 D_{m-2} &\rightarrow B_{m-1} B_m
 \end{aligned}$$



- Let V'' be the new set of variables, $\mathcal{P}''$ the new set of productions, and let $G_3 = (V'', T, \mathcal{P}'', S)$ be the new grammar. Clearly, the grammar G_3 is now in Chomsky normal form. It remains to prove that G_3 is equivalent to the grammar G_2 . One direction of the proof is clear: if $A \xRightarrow{*} \beta$ in the grammar G_2 , then $A \xRightarrow{*} \beta$ in the grammar G_3 . The other direction is proved by routine induction on the length of the derivation.

Example

- Convert the following grammar into a grammar in CNF:

$$\begin{aligned} S &\rightarrow bA|aB \\ A &\rightarrow bAA|aS|a \\ B &\rightarrow aBB|bS|b \end{aligned}$$

The only productions in the desired form are $A \rightarrow a$ and $B \rightarrow b$. All the other productions are not in the desired form. This grammar has no unit productions or ε -productions, so we can immediately begin replacing terminal symbols on the right-hand side with variables (except, of course, in the productions $A \rightarrow a$ and $B \rightarrow b$).

Replace the production $S \rightarrow bA$ with the two productions $S \rightarrow C_bA$ and $C_b \rightarrow b$.

Similarly, replace the production $S \rightarrow aB$ with the productions $S \rightarrow C_aB$ and $C_a \rightarrow a$.

Replace the production $A \rightarrow bAA$ with $A \rightarrow C_bAA$.

Replace the production $A \rightarrow aS$ with $A \rightarrow C_aS$.

Replace the production $B \rightarrow aBB$ with $B \rightarrow C_aBB$.

Replace the production $B \rightarrow bS$ with $B \rightarrow C_bS$.

The Final Stage

Replace the production $A \rightarrow C_bAA$ with the two productions $A \rightarrow C_bD_1$ and $D_1 \rightarrow AA$.

Replace the production $B \rightarrow C_aBB$ with the two productions $B \rightarrow C_aD_2$ and $D_2 \rightarrow BB$.

The final result is:

$$\begin{aligned} S &\rightarrow C_bA|C_aB & D_1 &\rightarrow AA \\ A &\rightarrow C_aS|C_bD_1|a & D_2 &\rightarrow BB \\ B &\rightarrow C_bS|C_aD_2|b & C_a &\rightarrow a \\ & & C_b &\rightarrow b \end{aligned}$$



4.11 An Algorithm for Deciding Membership

- **The membership problem:** Is there an algorithm that receives a context-free grammar $G = (V, T, P, S)$ and a sentence w , and determines whether $w \in L(G)$ or $w \notin L(G)$?
- In other words, is there an automatic way to determine whether there is a derivation

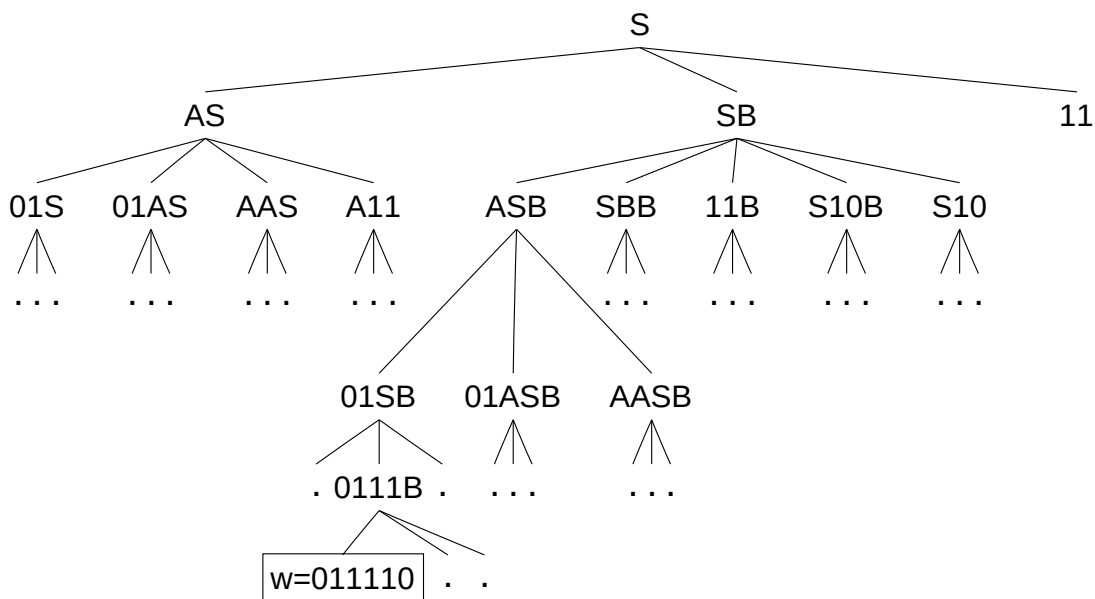
$$S \xRightarrow{*} w$$

(for every context-free grammar G and every sentence w).

- Related problems: if such an algorithm exists, can we find at least one derivation of w from S ? Can we find the list of all derivations of w from S ?
- The answer to the membership problem is affirmative. It is easy to produce at least two simple algorithms (though they are not especially efficient). However, certain restrictions must be imposed on the grammar G for these algorithms to be possible.
- The most direct algorithm (Top-down) is illustrated by the following example. Suppose, for example, that the grammar G is

$$\begin{aligned} S &\rightarrow AS|SB|11 \\ A &\rightarrow 01A|01 \\ B &\rightarrow 10B|10 \end{aligned}$$

and we wish to know whether the sentence $w = 011110$ is generated by G . We can perform an exhaustive search, as in Figure 4.11.



The tree stops at each node w , where $|w|=6$

Fig. 4.11: A search tree for finding a suitable derivation of the sentence 011110

- The tree represents an exhaustive search program in which we begin with the start variable S as the root and then branch to all the sentential forms immediately derived from it. At each stage, we repeat this operation for every leaf node that is not a sentence. The process stops after all the leaves of the tree are sentences.
- Clearly, such a tree may grow out of control after only a few stages, so an exhaustive graphical representation is usually impractical. Moreover, there is no guarantee that this process must terminate. It is very easy to construct grammars in which derivation sequences can continue indefinitely (exercise).
- Grammars in Chomsky normal form, on the other hand, guarantee that the tree is bounded by the length of the sentence we are trying to derive, and therefore guarantee that the search terminates within a predetermined time (which may be very long, so the question of efficient search is postponed until later).
- Formally, define a sequence of sets of sentential forms recursively:

$$\begin{aligned} F_0 &= \{S\} \\ F_{n+1} &= \{\beta \in (V \cup T)^* : \exists \alpha \in F_n, \alpha \Rightarrow \beta\} \end{aligned}$$

In short, F_n is the set of all sentential forms that can be derived from the variable S in n steps.

- Clearly, if $w \in L(G)$, then there is an n such that $w \in F_n$, and therefore a finite search will discover this (though perhaps only after too long). But if $w \notin L(G)$, the search may be infinite!? Unless we have a good reason to determine that the search should be stopped at some stage.
- If the grammar G is in Chomsky normal form, for example, then clearly all the sentential forms in the set F_n have length at least n . Therefore, the search stops at stage n , where n is the length of the word w .
- The algorithm can be improved by refining the sets F_n . For example, reject every sentential form that begins (or ends) with a sequence of terminal symbols that does not match the word w .
- Nevertheless, the running time of this algorithm is too long, and it is impractical.
- Another direct algorithm (Bottom-up) is based on the following idea:

$$\begin{aligned} H_0 &= \{w\} \\ H_{n+1} &= \{\alpha \in (V \cup T)^* : \exists \beta \in H_n, \alpha \Rightarrow \beta\} \end{aligned}$$

$w \in L(G)$ if and only if there is an n such that $S \in H_n$. We leave it to the reader to determine the details and the conditions under which this algorithm works.

- However, this algorithm also requires too much time and is impractical. We now turn to the description of a faster, practical algorithm.



- It turns out that if the grammar G is in Chomsky normal form, there is a much simpler way to determine whether $w \in L(G)$.
- By the theorem of the preceding section, there is an algorithm for converting almost every grammar G into a grammar in Chomsky normal form. The exceptions are grammars in which $\varepsilon \in G$.
- It is very easy to identify a grammar that generates ε :
 1. Check whether the grammar G contains a production of the form $A \rightarrow \varepsilon$. If not, then $\varepsilon \notin L(G)$, and we are done. If so, proceed to the next stage.
 2. Convert the grammar G into a grammar with no useless variables. Proceed to the next stage.
 3. Find the list of all nullable variables of G (in one of the preceding sections, we presented a simple algorithm for finding all the nullable variables). If S is in the list, then $\varepsilon \in L(G)$. Otherwise, $\varepsilon \notin L(G)$.
- In any event, by the Chomsky normal form theorem, there is an algorithm for converting the grammar G into a grammar G' in CNF such that $L(G') = L(G) - \{\varepsilon\}$.
- Therefore, assuming that we have a membership algorithm for grammars in CNF, we can use it to solve the more general problem as follows. After receiving a grammar G and a sentence w , perform the following steps:
 1. If $\varepsilon \in G$ and $w = \varepsilon$, then the algorithm terminates with the answer **YES**. If $\varepsilon \notin G$ but $w = \varepsilon$, then the algorithm terminates with the answer **NO**. Otherwise, proceed to the next stage.
 2. Convert the grammar G into a grammar G' in Chomsky normal form. Since at this stage it is known that $w \neq \varepsilon$, it follows that

$$w \in L(G) \iff w \in L(G')$$

Proceed to the next stage.

- 3. Use the membership algorithm for grammars in CNF to determine whether $w \in L(G')$.
- Thus, it remains only to present the membership algorithm for grammars in CNF in order to complete the solution of the problem.
- Let $G = (V, T, P, S)$ be a grammar in CNF, and let $w = c_1 c_2 \cdots c_n$ be a sentence in the language T^* . The length of w is n .
- For every i , $1 \leq i \leq n$, and every j , $1 \leq j \leq n - i + 1$, denote by $w_{i,j}$ the substring of w that begins with the symbol c_i and has length j . In other words:

$$w_{i,j} = c_i c_{i+1} \cdots c_{i+j-1}$$

- To determine whether $S \xRightarrow{*} w$, check, for all possible i , j and for every syntactic variable



$A \in V$, whether $A \xRightarrow{*} w_{i,j}$. If we can construct the sets

$$V_{i,j} = \left\{ A \in V : A \xRightarrow{*} w_{i,j} \right\}$$

then our problem is solved simply by checking whether $S \in V_{1,n}$. (Since $w = w_{1,n}$!).

- To understand the algorithm, note one more fact: if $j > 1$ and $A \xRightarrow{*} w_{i,j}$, then the derivation must have the form

$$A \xRightarrow{*} BC \xRightarrow{*} \cdots \xRightarrow{*} w_{i,j}$$

since every production in a grammar in CNF must have the form $A \rightarrow a$ or $A \rightarrow BC$.

- There is a k , $0 < k < j$, and a decomposition $w_{i,j} = w_{i,k}w_{i+k,j-k}$ such that

$$B \xRightarrow{*} w_{i,k} \quad \text{and} \quad C \xRightarrow{*} w_{i+k,j-k}$$

The fact that $k \neq 0$ and $k \neq j$ follows from the fact that in a grammar in CNF, an empty string cannot be derived from any syntactic variable!

- In other words, to determine whether $A \in V_{i,j}$, we must check, for every k , $0 < k < j$, and for every production $A \rightarrow BC$, whether $B \in V_{i,k}$ and $C \in V_{i+k,j-k}$. Assuming that we have the sets $V_{i,k}$, $V_{i+k,j-k}$, the testing time does not exceed the number of productions multiplied by the square of the number of variables.
- We construct the sets $V_{i,j}$ recursively on j (independently of i). First, construct the sets $V_{i,1}$ for every $1 \leq i \leq n$. The construction of the set $V_{i,j}$ is based on the previously constructed sets $V_{i,j'}$, $j' < j$.

➤ The algorithm

Input: A grammar G in CNF, a sentence w , and n (the length of w).

Goal: Determine whether $w \in L(G)$.

1. For every i , $i = 1, 2, \dots, n$, construct the set

$$V_{i,1} = \{ A \in V : A \rightarrow w_{i,1} \}$$

$w_{i,1}$ is simply the single symbol c_i , so we must examine all the productions of G and insert into the set V_i every variable A that occurs in a production of the form $A \rightarrow c_i$. The computation time is very short.

2. Now let j , $2 \leq j \leq n$.

We assume, of course, that all the sets $V_{i,j'}$, $j' < j$ have already been constructed.

$$A \in V_{i,j} \Leftrightarrow A \rightarrow BC, B \in V_{i,k}, C \in V_{i+k,j-k}, 0 < k < j$$



3. If $S \in V_{1,n}$, output YES; otherwise, output NO.

➤ **Example:** Suppose, for example, that the grammar G is given by

$$\begin{aligned} S &\rightarrow AB|BC \\ A &\rightarrow BA|a \\ B &\rightarrow CC|b \\ C &\rightarrow AB|a \end{aligned}$$

and the word w is given by $w = bbab$. Run the algorithm:

$$\begin{aligned} V_{1,1} &= \{B\} & V_{1,2} &= \emptyset \\ V_{2,1} &= \{B\} & V_{2,2} &= \{A, S\} \\ V_{3,1} &= \{A, C\} & V_{3,2} &= \{S, C\} \\ V_{4,1} &= \{B\} \\ \\ V_{1,3} &= \{A\} & V_{1,4} &= \{S, C\} \\ V_{2,3} &= \{S, C\} \end{aligned}$$

The test $S \in V_{1,4}$ succeeds, and therefore w is generated by G .

➤ It is customary to represent the sets $V_{i,j}$ by the following table:

j	4	S,C			
	3	A	S,C		
	2	$\emptyset$	A,S	S,C	
	1	B	B	A,C	B
		1	2	3	4
		i			

Fig. 4.12: Execution table

➤ Discussion of the related problems: the algorithm answers only the question whether $w \in L(G)$. Can we find at least one derivation of w ? All the derivations?

4.12 Greibach Normal Form

- We conclude the chapter with another important theorem, which guarantees that every context-free grammar G that does not generate ε has an equivalent grammar in the following simple form, which is useful in important applications.

i Definition 4.20: A context-free grammar G is said to be **in Greibach normal form** if all its productions have the form $X \rightarrow a\alpha$, where α is a string of syntactic variables ($\alpha \in V^*$) and a is a terminal symbol ($a \in T$).

◆ **Example 4.25:**

$$\begin{aligned} V &= \{S, A, B, C\} \\ T &= \{a, b, c, d\} \\ S &\rightarrow aSS|bA|dBC \\ A &\rightarrow cBB|d \\ B &\rightarrow a|b|c \\ C &\rightarrow aBC|a \end{aligned}$$

i Theorem 4.23: (Greibach Normal Form) Let G be a context-free grammar such that $\varepsilon \notin L(G)$. Then there is an equivalent grammar G' in Greibach normal form.

- The highly complex proof of the theorem is beyond the scope of the course and will not be presented here. The interested student can find a relatively short proof at the following link:

 [Greibach Normal Form](#)

or a more detailed proof of several subforms of Greibach normal form:

 [Proof of Greibach Normal Form \(page 16\)](#)

4.13 The Pumping Lemma for Context-Free Languages

i Theorem 4.24: For every infinite context-free language L , there is a natural number p (the **pumping length**) such that for every sentence $w \in L$, if $|w| \geq p$, then w can be decomposed into five parts

$$w = xuyvz$$

such that

- (a) For every $n \geq 0$, $xu^n yv^n z \in L$,
- (b) $|uv| > 0$,

(c) $|uyv| \leq p$.

✓ **Proof:** Without loss of generality, we may assume that L does not contain the empty sentence ε , and therefore, by Theorem 4.20, we may assume that the grammar G for L contains neither useless variables nor unit productions. Examine all the productions $X \rightarrow \alpha$ of the grammar G and calculate the maximum length of α

$$N = \max \{ |\alpha| : X \rightarrow \alpha \text{ is a production of } G \}$$

Let w be a sentence generated by the grammar G , and let τ be the derivation tree of w . Recall that at level 0 of the tree there is one node S (the start variable); at level 1, there are at most N children of the node S ; at level 2, there are at most N^2 children of the node S ; and so on. Therefore, if τ is a derivation tree of height k , then the derived sentence w contains at most N^k symbols.

Let $m = |V|$ be the number of syntactic variables in the grammar G . Since our language is infinite, we can choose a sentence w of length at least N^{m+2} . By the preceding argument, the derivation tree τ of w must have height at least $m+2$, and therefore there is a branch of length $m+2$ in the tree τ . The branch contains a single terminal symbol at its end, but all the other $m+1$ symbols must be syntactic variables. Since the number of syntactic variables in the grammar G is m , the pigeonhole principle implies that there is a syntactic variable A that occurs at least twice on the branch. Let i be the level at which A occurs for the first time, and let τ_i be the subtree of τ containing all the nodes from level 0 through level i . Similarly, let j be the level at which A occurs for the second time, and let τ_j be the subtree of τ containing all the nodes from level 0 through level j .

The yield of the tree τ_i must have the form $\alpha_1 A \beta_1$, where α_1, β_1 are sentential forms of G . The sentential forms α_1, β_1 are obtained by concatenating the leaves of the tree τ_i (to the left and right of the variable A). Contrary to the impression created by Figure 4.13, not all the leaves of τ_i must be at level i . In general, some of the leaves occur at lower levels. A similar remark applies to the tree τ_j . Nevertheless, Figure 4.13 reflects the logical structure of our argument.



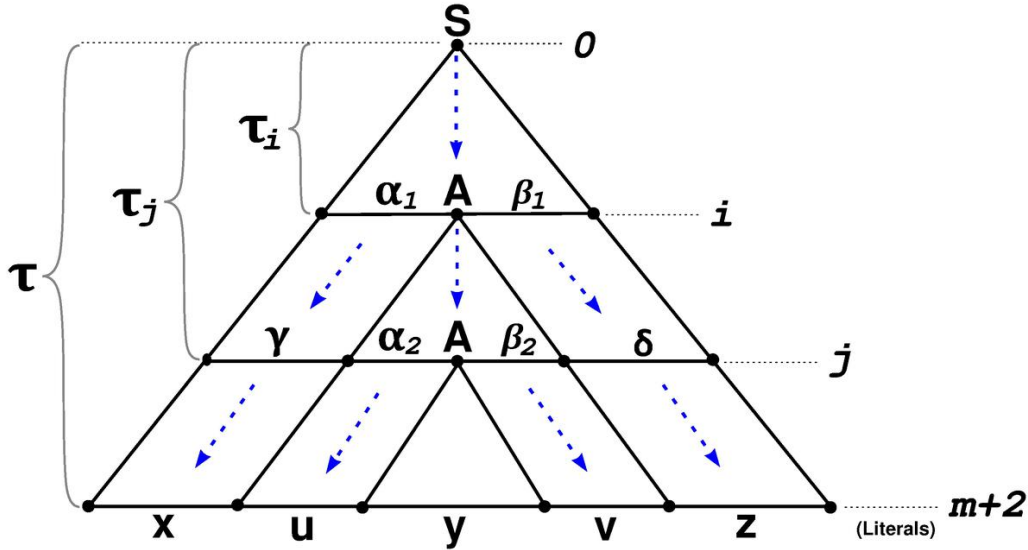


Fig. 4.13: The derivation tree of a sentence w of length $m + 2$. The variable A occurs twice along one of the branches.

Denote by γ the concatenation of all the leaves in the tree τ_j that are derived from the sentential form α_1 . That is, $\alpha_1 \xRightarrow{*} \gamma$. Similarly, construct the concatenation of the leaves of the tree τ_j that are derived from β_1 : $\beta_1 \xRightarrow{*} \delta$. The occurrence of the variable A at level i yields the sentential form $\alpha_2 A \beta_2$ at level j .

$$A \xRightarrow{*} \alpha_2 A \beta_2 \quad (4.1)$$

and therefore

$$S \xRightarrow{*} \alpha_1 A \beta_1 \xRightarrow{*} \gamma \alpha_2 A \beta_2 \delta \quad (4.2)$$

Now pass to the leaves of the tree τ (terminal symbols!). Figure 4.13 shows that

$$\gamma \xRightarrow{*} x, \quad \alpha_2 \xRightarrow{*} u, \quad A \xRightarrow{*} y, \quad \beta_2 \xRightarrow{*} v, \quad \delta \xRightarrow{*} z \quad (4.3)$$

where x, u, y, v, z are strings of terminal symbols only! From the two derivations (4.1), (4.3), we obtain

$$A \xRightarrow{*} \alpha_2 A \beta_2 \xRightarrow{*} u A v \quad (4.4)$$

From the two derivations (4.2), (4.3), we obtain

$$S \xRightarrow{*} \gamma \alpha_2 A \beta_2 \delta \xRightarrow{*} x u A v z \quad (4.5)$$

From the two derivations (4.4), (4.5), we obtain

$$S \xRightarrow{*} x u A v z \xRightarrow{*} x u u A v v z \xRightarrow{*} x u^2 y v^2 z$$

By repeatedly using the derivations (4.4), (4.5), we obtain

$$S \xRightarrow{*} x u^2 A v^2 z \xRightarrow{*} x u^2 u A v v^2 z \xRightarrow{*} x u^3 y v^3 z$$

By induction, we obtain, for every natural number n ,

$$S \xRightarrow{*} xu^n Av^n z$$

In summary, for every sentence w in the language L of length at least $p = N^{m+2}$ (the pumping length p), there is a decomposition $w = xuyvz$ such that, for every natural number n , $xu^n yv^n z \in L$.

It remains to prove parts b and c of the theorem.

(b) Suppose, by way of contradiction, that $|uv| = 0$. That is, $\alpha_2 = \beta_2 = \varepsilon$. In this case, the derivation $A \xRightarrow{*} \alpha_2 A \beta_2$ becomes $A \xRightarrow{*} A$, which can occur only if the grammar G contains unit productions, contrary to the assumption that G contains no unit productions.

(c) As stated above, our pumping length is $p = N^{m+2}$, where $m+2$ is the height of the derivation tree τ of the sentence $w = xuyvz$. As we saw at the beginning of the proof, the maximum length of the yield of a derivation tree of height $m+2$ is N^{m+2} , and therefore

$$|uyv| \leq |w| \leq N^{m+2} = p \quad \blacksquare$$

◆ **Example 4.26:** Using the pumping lemma for context-free languages, prove that the language

$$L = \{a^n b^n c^n : n \in \mathbb{N}\}$$

is not a context-free language.

🔴 **Solution:** Suppose, by way of contradiction, that L is a context-free language. Let p be the pumping length of L guaranteed by the pumping lemma for context-free languages. Consider the sentence $a^p b^p c^p$. By the theorem, it has a decomposition

$$a^p b^p c^p = xuyvz$$

such that $|uv| > 0$, $|uyv| \leq p$, and for every natural number n , $xu^n yv^n z \in L$. The constraint $|uyv| \leq p$ implies that $uyv \subseteq a^p b^p$ or $uyv \subseteq b^p c^p$. The size of uyv does not allow it to begin with one of the symbols of a^p and end with one of the symbols of c^p , because its length would then be greater than p , contrary to the assumption. In the first case, uyv contains no occurrence of the symbol c , and the expression $|uv|$ must contain at least one occurrence of a or b (but not of c). Without loss of generality, suppose that uv contains an occurrence of a . Then $xu^n yv^n z$ increases the number of occurrences of a without bound, whereas the number of occurrences of c remains fixed. That is, for sufficiently large n , we obtain a sentence in which the number of occurrences of a is greater than the number of occurrences of c and which therefore does not belong to the language L , contradicting the pumping lemma. Therefore, L is not a context-free language.

4.14 Additional Properties of Context-Free Languages

- As with the questions asked about regular languages, we can ask whether the union of context-free languages is context-free. Does the intersection of context-free languages yield a context-free language? Similar questions can be asked about concatenation and the complement of a context-free language.
- Unlike the case of regular languages, the answers to these questions are not all affirmative.

i Theorem 4.25: If L_1, L_2 are two context-free languages, then their union $L_1 \cup L_2$ is also a context-free language.

- ♥ **Proof:** Let $G_1 = (V_1, T_1, P_1, S_1)$, $G_2 = (V_2, T_2, P_2, S_2)$ be context-free grammars for the languages L_1, L_2 , respectively. Without loss of generality, we may assume that the sets of variables V_1, V_2 are disjoint (a variable can always be replaced by another without affecting the language). Define a grammar $G_3 = (V_3, T_3, S_3, P_3)$ that generates the language $L_1 \cup L_2$. The set of variables of G_3 is

$$V_3 = \{S_3\} \cup V_1 \cup V_2$$

That is, the variables of G_3 are the union of the variables of G_1, G_2 , together with a start variable S_3 . We may assume that the start variable S_3 is not contained in the set $V_1 \cup V_2$. The set of terminal symbols of G_3 is $T_3 = T_1 \cup T_2$. The set of productions P_3 of the grammar G_3 is defined by

$$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 | S_2\}$$

That is, the productions of the grammar G_3 are all the productions of G_1, G_2 , together with one additional production $S_3 \rightarrow S_1 | S_2$.

We prove that $L(G_3) = L_1 \cup L_2$.

Direction 1: Suppose that $w \in L_1 \cup L_2$. Then $w \in L_1$ or $w \in L_2$. If $w \in L_1$, then there is a derivation sequence $S_1 \xRightarrow{*} w$. Therefore, $S_3 \rightarrow S_1 \xRightarrow{*} w$ is a derivation sequence for w in the grammar G_3 , and hence $w \in L(G_3)$. If $w \in L_2$, then $S_3 \rightarrow S_2 \xRightarrow{*} w$ is a derivation sequence for w in G_3 , and again we obtain $w \in L(G_3)$.

Direction 2: Suppose that $w \in L(G_3)$. Then there is a derivation sequence $S_3 \xRightarrow{*} w$. The variable S_3 occurs in only one production: $S_3 \rightarrow S_1 | S_2$. Therefore, the derivation sequence $S_3 \xRightarrow{*} w$ must begin with the production $S_3 \rightarrow S_1$ or $S_3 \rightarrow S_2$. In the first case, $S_3 \rightarrow S_1 \xRightarrow{*} w$, and hence $S_1 \xRightarrow{*} w$, so $w \in L_1$. In the second case, $S_3 \rightarrow S_2 \xRightarrow{*} w$, and hence $S_2 \xRightarrow{*} w$, so $w \in L_2$. ■

- **Remark:** In Direction 2, the derivation $S_1 \xRightarrow{*} w$ involves only variables and terminal symbols in the grammar G_1 , because S_1 and the other variables do not belong to the grammar G_1 . Therefore, the entire derivation takes place in the grammar G_1 . A similar remark applies to the derivation $S_2 \xRightarrow{*} w$.



i Theorem 4.26: If L_1, L_2 are two context-free languages, then their concatenation L_1L_2 is also a context-free language.

✓ Proof: Let $G_1 = (V_1, T_1, P_1, S_1)$, $G_2 = (V_2, T_2, P_2, S_2)$ be context-free grammars for the languages L_1, L_2 , respectively. Without loss of generality, we may assume that the sets of variables V_1, V_2 are disjoint. Define a grammar $G_3 = (V_3, T_3, S_3, P_3)$ that generates the language $L_1 \cup L_2$. The definition of G_3 is identical to that in the preceding theorem, except for the additional production

$$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1S_2\}$$

The start variable S_3 of G_3 generates the concatenation S_1S_2 of the start variables S_1, S_2 of G_1, G_2 , thereby making it possible to generate all combinations uv , $u \in L_1, v \in L_2$. We leave the details to the student. ■

An immediate consequence of the last theorem is the following theorem.

i Theorem 4.27: If L is a context-free language, then the language L^* is also context-free.

✓ Proof: Recall that

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

Using the preceding theorem and induction, we find that for every positive natural number n , the language L^n is context-free. For $n = 0$, $L^0 = \{\varepsilon\}$ is a context-free language (because it is a finite language), and therefore its union with the union of the languages L^n yields a context-free language. ■

➤ Remark: as in the two preceding theorems, the last theorem can also be proved by constructing a suitable grammar for the language L^* : if $G_1 = (V, T, S, P)$ is a context-free grammar for L , then by adding a new start variable S' and a production $S' \rightarrow SS' | \varepsilon$, we obtain a context-free grammar for the language L^* . Verifying the details is simple.

➤ The intersection of two context-free languages does not necessarily yield a context-free language, as the following example shows.

◆ **Example 4.27:** Consider the two languages

$$\begin{aligned} L_1 &= \{a^i b^i c^j : i, j \in \mathbb{N}\} \\ L_2 &= \{a^j b^i c^i : i, j \in \mathbb{N}\} \end{aligned}$$

It is easy to construct a context-free grammar for each of them (a simple exercise!), or to see that each is a concatenation of two context-free languages (for example, L_1 is the concatenation of

the language $(ab)^i$ with the language c^j). The intersection $L_1 \cap L_2$ is the language

$$L = \{ a^i b^i c^i : i \in \mathbb{N} \}$$

which we proved in Example 4.26 is not a context-free language.

- ❖ **Example 4.28:** Return once more to the languages L_1, L_2 from the preceding example. We saw that both are context-free languages. We now prove that at least one of the three complement languages $\overline{L_1}, \overline{L_2}, \overline{L_1 \cup L_2}$ is not context-free. Suppose, by way of contradiction, that all of them are context-free. Then

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

is a context-free language, contrary to what we obtained in the preceding example.

- ❖ The conclusion from the last theorems is that the class of context-free languages is closed under union and concatenation, but is not closed under intersection and complementation.
- ❖ At the end of the next chapter, we prove that the intersection of a context-free language with a regular language is context-free. For this purpose, we need another type of automaton: a pushdown automaton.

4.15 Applications to Programming Languages and Natural Languages

- ❖ The subject of context-free grammar arose from the need to understand and characterize the grammar of programming languages and natural languages in everyday life.
- ❖ Up to this point, context-free grammar has been studied from a theoretical viewpoint, and its connection to programming languages and natural languages has not yet been clarified.
- ❖ When we turn to practice, things take a somewhat different form. For example, every terminal symbol we have encountered so far was represented as a single symbol from the English alphabet, but in practice, more than 26 terminal symbols are required, and therefore, in many cases, terminal symbols are represented by strings over the language's alphabet. In such cases.
- ❖ To construct a sentence, these strings must be concatenated using a separator, usually whitespace, although it may be any suitable symbol that does not create ambiguity. The separator may be omitted when there is no danger of ambiguity.
- ❖ For example, consider the following sentence

"Be bold, be courageous, be your best."

In the Python programming language, the sentence above is represented by a list of ten “terminal symbols”, which are in fact words and punctuation marks in the language.



```

sent = ['Be', 'bold', ',', 'be', 'courageous', ',', 'be', 'your', 'best', '.']
sent[0] = 'Be'
sent[1] = 'bold'
sent[2] = ','
sent[3] = 'be'
sent[4] = 'courageous'
sent[5] = ','
sent[6] = 'be'
sent[7] = 'your'
sent[8] = 'best'
sent[9] = '.'

```

Note that in addition to words such as 'Be', 'bold', the punctuation marks ',', '.' are also words in the full grammatical sense! That is, they are terminal symbols in the grammar we wish to construct. As before, a sentence is a concatenation of words. This time, however, the concatenation is performed using spaces between the words. Since there is no danger of ambiguity, there is no need to insert a space between ordinary words and punctuation marks (which are also words in the syntactic sense!).

❖ **Example 4.29:** For example, we can define a simple grammar G that generates sentences of the kind above.

$$\left\{ \begin{array}{l} \text{Sentence} \rightarrow \text{Be Trait} \mid \text{Sentence , Command .} \\ \text{Command} \rightarrow \text{Be Trait} \mid \text{be Trait} \mid \text{sentence} \\ \text{condition} \rightarrow \text{c1} \mid \text{c2} \mid \text{c3} \\ \text{Trait} \rightarrow \text{bold} \mid \text{courageous} \mid \text{your best} \end{array} \right.$$

The start variable is: `statement`.

The set of syntactic variables is

$$V = \{\text{statement}, \text{condition}, \text{sentence}, \text{result}\}$$

The set of terminal symbols is

$$T = \{\text{if}, \text{then}, \text{else}, \text{c1}, \text{c2}, \text{c3}, \text{r1}, \text{r2}, \text{r3}\}$$

Exercises for chapter 4

Guidance: Some of the exercises can be solved, or their results verified, by using a simple simulation program that can be run in an Internet browser. We would be pleased to receive code from students for solutions to exercises.

<https://colab.research.google.com/github/samyzaf/notebooks/blob/main/cfg.ipynb>



1. Let G be the following grammar

$$\begin{aligned} S &\rightarrow SAS|A|\varepsilon \\ A &\rightarrow 00|\varepsilon \end{aligned}$$

Convert the grammar G into a grammar in Chomsky normal form.

2. Let G be the following grammar

$$\begin{aligned} S &\rightarrow 01S|B0S|A0S|0 \\ A &\rightarrow 0BA|SB \\ B &\rightarrow AB1|SA \end{aligned}$$

Convert the grammar G into a grammar in Chomsky normal form.

3. Prove that if G is a context-free grammar in Chomsky normal form, then for every sentence $w \in L(G)$, if $|w| = n \geq 1$, the length of every derivation sequence for w is $2n + 1$.
4. Construct a deterministic automaton accepting the language generated by the following grammar

$$\begin{aligned} S &\rightarrow 001A \\ A &\rightarrow 10B \\ B &\rightarrow 11|0A \end{aligned}$$

5. Construct a grammar generating the language $L = \{0^m 1^n : m \geq 3, n \geq 2\}$.
6. Construct a context-free grammar for the language accepted by the following nondeterministic automaton

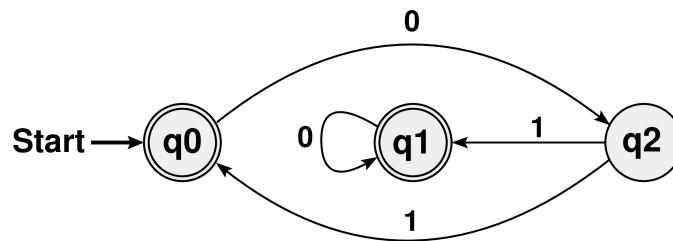


Fig. 4.14: A nondeterministic automaton with three states $\{q_0, q_1, q_2\}$ over the alphabet $\Sigma = \{0, 1\}$

7. Construct a context-free grammar for the language of all words over the alphabet $\Sigma = \{a, b\}$ having at least two occurrences of the symbol a , and at least three occurrences of the symbol b .
8. Construct a context-free grammar for the language of all words over the alphabet $\Sigma = \{0, 1\}$

of the form $w = 0^m 1^n$, where $m + n$ is even.

9. Construct a context-free grammar for the language of all words over the alphabet $\Sigma = \{a, b\}$ in which the number of occurrences of the symbol a is even, and the number of occurrences of the symbol b is odd.
10. A derivation tree is given for the sentence $w = abaaacac$ according to an unknown context-free grammar G (see Figure 4.15)

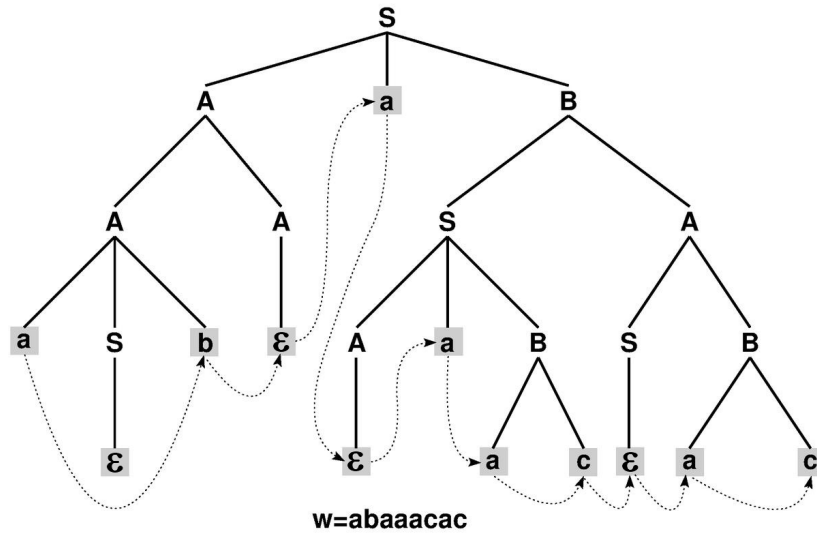


Fig. 4.15: The derivation tree of the sentence $w = abaaacac$ according to an unknown grammar G

- a. Write the leftmost derivation of the sentence w according to the grammar G .
 - b. Write the rightmost derivation of the sentence w according to the grammar G .
 - c. Find a decomposition $w = w_1 w_2 w_3$ such that $w_2 \neq \varepsilon$ and $w_1 w_2 w_2 w_3 \in L(G)$.
 - d. What can be said about the ambiguity of G ?
11. At the following Microsoft link, you can learn about the syntax rules for real numbers in the C programming language

C Floating-Point Constants

Construct a context-free grammar for the language of C real numbers.

12. Prove that if L_1 is a regular language, then the language $L_2 = \{wv : w \in L_1, |v| = 5\}$ is also regular.
13. Let L_1, L_2 be two languages over a common alphabet Σ . Define the left quotient operation between languages by

$$L_1/L_2 = \{w \in \Sigma^* : \exists u [u \in L_2 \wedge uw \in L_1]\} \quad (4.6)$$

- a. Give a simple description of L_1/L_2 for the case $L_1 = L(0^*100^*)$, $L_2 = L(010^*)$.
 - b. Prove that if L_1, L_2 are regular languages, then the language L_1/L_2 is also regular.
14. Let L_1, L_2 be two languages over a common alphabet Σ . Prove or disprove by a counterexample: if L_1 and $L_1 \cup L_2$ are regular, then L_2 is regular.
 15. Prove or disprove by a counterexample: if L_2 is nonregular and $L_1 \subseteq L_2$, then L_1 is also nonregular.
 16. Construct an algorithm for testing whether $L_3 = L_1L_2$ for any three regular languages L_1, L_2, L_3 .
 17. Construct an algorithm for testing whether $L_3 = L_1/L_2$ for any three regular languages L_1, L_2, L_3 (see Equation (4.6)).
 18. Let L be a given language over an alphabet Σ . The language L_{tail} is defined as the set of all suffixes of words in the language L

$$L_{tail} = \{ w \in \Sigma^* : \exists u [u \in \Sigma^* \wedge uw \in L] \}$$

Prove that if L is a regular language, then L_{tail} is also regular.

19. A string u is called a **proper prefix** of a string w if u is a prefix of w but $u \neq w$. In other words, u is a proper prefix of w if there is an $i < |w|$ such that $u = w[1 : i]$. Let L be a given language over an alphabet Σ . The language L_{min} is defined as the set of all words $w \in L$ such that every proper prefix of w does not belong to the language L .

$$L_{min} = \{ w \in L : \forall i < |w| [w[1 : i] \notin L] \}$$

Prove that if L is a regular language, then L_{min} is also regular.

20. Construct a context-free grammar for the language L accepted by the following automaton

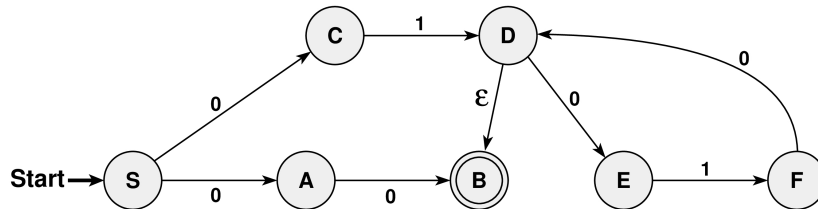


Fig. 4.16: A nondeterministic automaton with ϵ -transitions over the alphabet $\Sigma = \{0, 1\}$

21. Let G be the following regular grammar

$$\begin{aligned} S &\rightarrow 01A|00B|11 \\ A &\rightarrow 0B|1C|00 \\ B &\rightarrow 0A|1B \\ C &\rightarrow 01S \end{aligned}$$

Construct a nondeterministic automaton accepting the language $L(G)$.

22. Let G be the following regular grammar (with ε -transitions)

$$\begin{aligned} S &\rightarrow abS|bbaA|\varepsilon \\ A &\rightarrow abA|bB \\ B &\rightarrow acS|bC|\varepsilon \\ C &\rightarrow aC|bA \end{aligned}$$

Construct a nondeterministic automaton accepting the language $L(G)$.

23. Let G be the following grammar over the alphabet $T = \{0, 1\}$

$$\begin{aligned} S &\rightarrow 0|0S|1A \\ A &\rightarrow 1|1A \end{aligned}$$

Prove that no sentence in the language $L(G)$ contains the substring 10.

24. Prove that every sentence generated by the following grammar has an equal number of zeros and ones

$$S \rightarrow \varepsilon|1S0|0S1|SS$$

25. Let G be the following grammar over the alphabet $T = \{0, 1\}$

$$S \rightarrow 0|S1S|SS1|1SS|S0$$

Prove that the sentence $w = 10011011$ is not generated by the grammar G .

Hint: Using recursion, prove that every sentence generated by G has more zeros than ones.

26. Construct a right-regular grammar for the language $((1 + 00)^*01)^*$.

27. Find a right-regular grammar equivalent to the following grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A0|1A1|0|1 \\ B &\rightarrow \varepsilon|1B|0B \end{aligned}$$



28. Construct a nondeterministic automaton for the language generated by the following grammar

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow 10A|1A|\varepsilon \end{aligned}$$

29. A context-free grammar $G = (V, T, P, S)$ is called a **linear grammar** if every production has the form $A \rightarrow wB$, $A \rightarrow Bw$, or $A \rightarrow w$. Prove that there is a nonregular language generated by a linear grammar.

30. A context-free grammar $G = (V, T, P, S)$ is called a **strong linear grammar** if every production has the form $A \rightarrow aB$ or $A \rightarrow \varepsilon$. Prove that a language L is generated by such a grammar if and only if it is a regular language.

Hint: Find a way to convert every such grammar into a finite automaton, and a way to convert every finite automaton into a strong linear grammar.



31. Find a grammar for the language of all sentences over the alphabet $\{0, 1\}$ in which the number of zeros is twice the number of ones.



32. Find a context-free grammar for the following language

$$L = \{a^i b^j c^n : i + j = n\}$$



33. Find a context-free grammar for the following language

$$L = \{a^i b^j c^m d^n : i + j = m + n\}$$

34. Define a context-free grammar G by the productions

$$\begin{cases} \text{statement} \rightarrow \text{if condition then sentence} \mid \text{sentence} \\ \text{sentence} \rightarrow \text{if condition then sentence else statement} \mid \text{result} \\ \text{condition} \rightarrow \text{c1} \mid \text{c2} \mid \text{c3} \\ \text{result} \rightarrow \text{r1} \mid \text{r2} \mid \text{r3} \end{cases}$$

The start variable is: `statement`.

The set of syntactic variables is

$$V = \{\text{statement}, \text{condition}, \text{sentence}, \text{result}\}$$

The set of terminal symbols is

$$T = \{\text{if}, \text{then}, \text{else}, \text{c1}, \text{c2}, \text{c3}, \text{r1}, \text{r2}, \text{r3}\}$$



Let w be the following sentence

if c1 then if c2 then r1 else if c3 then r2 else r3

Prove that G is an ambiguous grammar by finding two different derivation trees for the sentence w .

Hint: Note that, unlike the schematic grammars encountered at the beginning of the chapter, in which the symbols were joined with no separator, in the present grammar, concatenation is performed by inserting a space between every two consecutive symbols. In fact, this is the situation in all grammars for programming languages and natural languages.

35. The following is a context-free grammar for arithmetic expressions

$$\begin{cases} \text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{expr} * \text{expr} \mid \text{expr} / \text{expr} \\ \text{expr} \rightarrow a \mid b \mid c \\ \text{expr} \rightarrow (\text{expr}) \end{cases}$$

where the set of syntactic variables is $V = \{\text{expr}\}$, and the set of terminal symbols is

$$T = \{*, +, -, /, a, b, c, (,)\}$$

The start variable is expr .

- a. Show that this grammar is ambiguous by finding an arithmetic expression with two different derivation trees.
- b. Find an equivalent unambiguous grammar for the same language.

36. Let $L = \{a^i b^j c^k : i = j \vee j = k, i, j, k \geq 0\}$.

- a. Construct a context-free grammar generating L .
- b. Determine whether the grammar found in the preceding part is ambiguous. Justify your answer.

37. Let L be the language

$$L = \{0^i 1^j : 0 \leq i \leq j \leq 2i\}$$

- a. Prove that L is a context-free language by constructing a context-free grammar for it.
- b. Is the grammar found in the preceding part ambiguous? If so, can you find an equivalent grammar that is not ambiguous?

38. Prove that the language $L = \{a^i b^j c^k : i \leq j \leq k\}$ is not context-free.

Hint: Let p be the pumping length of L , and let $w = a^p b^p c^p$. Use the pumping lemma for context-free languages.

39. Prove that the language $L = \{a^i b^j : j = i^2\}$ is not context-free.



40. Prove that the language $L = \{0^n : n \text{ is a prime number}\}$ is not context-free.

41. Prove that the language $L = \{a^i b^j c^k : k = i \cdot j\}$ is not context-free.

Hint: $w = a^p b^p c^{p^2}$

42. Prove that the language $L = \{0^i 1^j : j \leq i^2\}$ is not context-free.

43. Determine whether the following languages are context-free.

(a) $L = \{0^i 1^j 0^i 1^j : i, j \in \mathbb{N}\}$

(b) $L = \{0^i 1^j 0^k 1^\ell : i, j, k, \ell \in \mathbb{N}, i + j \leq k + \ell\}$

(c) $L = \{a^i b^i c^j : i, j \in \mathbb{N}, i \leq j\}$

(d) $L = \{u0v : u, v \in \{0, 1\}^*, u \neq v\}$

(e) $L = \{u0v : u, v \in \{0, 1\}^*, u = 1v1\}$

(f) $L = \{u0v : u, v \in \{0, 1\}^*, u \subseteq v\}$ ($u \subseteq v$: u is a substring of v)



44. Let $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$.

Prove that L is not a context-free language.

Hint: Try to use Theorem 5.7.



45. Let L_1, L_2 be two languages over a common alphabet Σ . Define the left quotient operation between languages by

$$L_1/L_2 = \{w \in \Sigma^* : \exists u [u \in L_2 \wedge uw \in L_1]\}$$

Prove that if L_1 is a context-free language and L_2 is a regular language, then L_1/L_2 is a context-free language.



46. Let $L = \{w \in \{0, 1\}^* : \#_0(w) = 2\#_1(w)\}$.

Prove that L is a context-free language by finding a suitable grammar that generates it. Prove that the grammar you found is correct.



47. Let L be a given language over an alphabet Σ . The language L_{tail} is defined as the set of all suffixes of words in the language L

$$L_{tail} = \{w \in \Sigma^* : \exists u [u \in \Sigma^* \wedge uw \in L]\}$$

Prove that if L is a context-free language, then L_{tail} is also context-free.

48. Let L be a context-free language. Prove that there is a natural number n such that for every string $u \in L$, if $|w| \geq n$, then there is a string $v \in L$ such that $|u| < |v| \leq |u| + n$.





49. Prove that the language $L = \{a^i b^j c^k : i = j \vee j = k, i, j, k \geq 0\}$ which we showed in Exercise 36 is context-free, is also inherently ambiguous.

50. Find a grammar generating all regular expressions over the alphabet $\{0, 1\}$.

Hint: The set of terminal symbols is $T = \{0, 1, *, (,)\}$.

51. Give an example of a language L that satisfies the pumping property but is not context-free.



52. Prove the following stronger version of the pumping lemma for context-free languages, in which $u \neq \varepsilon$ and also $v \neq \varepsilon$:

For every infinite context-free language L , there is a natural number p (the **pumping length**) such that for every sentence $w \in L$, if $|w| \geq p$, then w can be decomposed into five parts

$$w = xuyvz$$

such that

- (a) For every $n \geq 0$, $xu^n y v^n z \in L$,
- (b) $|u| > 0$,
- (c) $|v| > 0$,
- (d) $|uyv| \leq p$.



Chapter 5

Nondeterministic Pushdown Automata

5.1 Definition

i **Definition 5.1:** A *nondeterministic pushdown automaton* is a theoretical computing machine M characterized by the following six components:

Q **Set of states**

These are the states in which the automaton may be at any stage.

Σ **Input alphabet (Input Symbols)**

This is the alphabet over which the strings supplied to the automaton are defined.

Denote by Σ_ϵ the alphabet together with the empty string ϵ : $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

Γ **Stack alphabet (Stack Symbols)**

This is the set of symbols that may be pushed onto or popped from the automaton's stack.

In some cases, it may also include the input symbols. Denote by Γ_ϵ the set $\Gamma \cup \{\epsilon\}$, and by Γ^* all the words that can be formed using Γ .

δ **Transition function** $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma^*)$

F **Set of accepting states**, $F \subseteq Q$.

q_0 **Start state**

This is the state in which the automaton is at the beginning of every computation.

Abbreviated notation:

$$M = (Q, \Sigma, \Gamma, \delta, F, q_0)$$

⦿ Whenever a pushdown automaton is discussed, all six components above must be stated explicitly (unless they are clear from the context): $(Q, \Sigma, \Gamma, \delta, F, q_0)$.

⦿ A pushdown automaton is essentially an ordinary nondeterministic automaton augmented with a memory device. The stack attached to it is a memory device that enables the automaton to

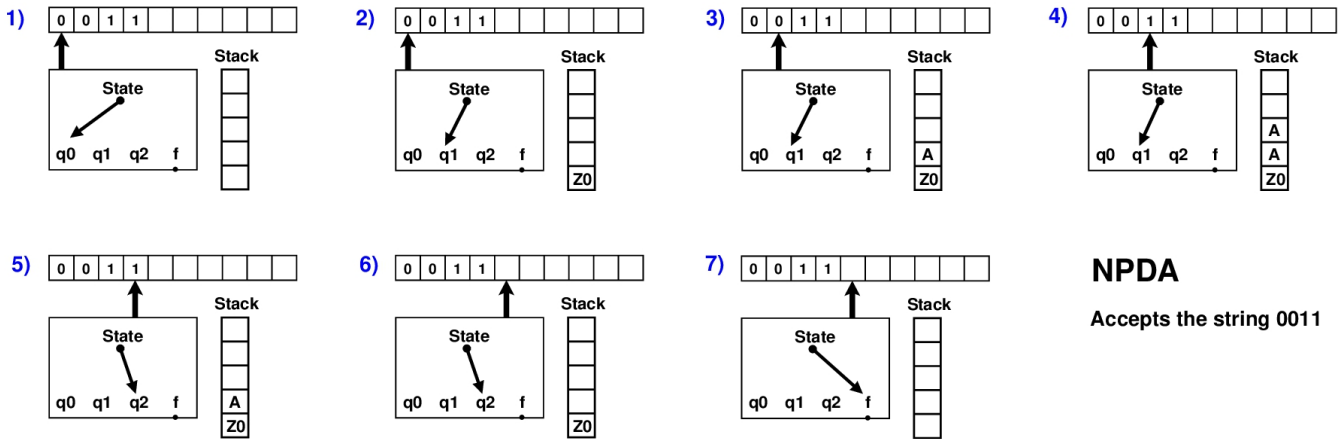


Fig. 5.1: A computation sequence of a nondeterministic pushdown automaton for the language $0^n 1^n$ (Example 1 below)

store information about earlier operations and use it at later stages of the computation.

- Readers familiar with programming languages will recognize the stack as a familiar basic data structure, sometimes called **LIFO**, which stands for **Last In First Out**.
- Nevertheless, this is a very limited memory device, because at every stage of the computation the automaton has access only to the **top symbol** of the stack. That is, the automaton can read only the symbol at the **top of the stack**, and has no information about the other symbols in the stack.
- Figure 5.1 shows an example of a computation sequence of a pushdown automaton M that we encounter later in Example 5.1. We use the figure to gain an initial impression of how a pushdown automaton operates.
 - ◆ **Stage 1:** The string **0011** is placed on the input tape, and the automaton is then activated (**Start**).
 - ◆ **Stage 2:** According to the transition diagram (see Figure 5.5), when the automaton is in the start state q_0 , there is only one option: read the empty string ϵ (that is, the input head does not move); the automaton pushes the symbol Z_0 onto the top of the stack (Z_0 usually marks the bottom of the stack); the automaton moves to state q_1 ; the input head does not move. No symbol is popped from the stack.
 - ◆ **Stage 3:** According to the transition diagram, when the automaton is in state q_1 and the input head reads **0**, the automaton pushes the symbol **A** onto the top of the stack, moves to state q_1 , and advances to the next symbol on the tape. No symbol is popped from the stack.
 - ◆ **Stage 4:** According to the transition diagram, when the automaton is in state q_1 and the input head reads **1**, the automaton pops the symbol **A** from the top of the stack, moves to state q_2 , and advances to the next symbol on the tape. No symbol is pushed onto the stack.
 - ◆ **Stage 5:** When the automaton is in state q_2 and the input head reads **1**, the automaton pops the symbol **A** from the top of the stack, remains in state q_2 , and advances to the next symbol on the tape. No symbol is pushed onto the stack.
 - ◆ **Stage 6:** The operation is identical to Stage 5.

- ◆ **Stage 7:** The automaton has finished scanning the input and has also reached the bottom of the stack (it sees the symbol Z_0). According to the transition diagram (see Figure 5.5), the automaton pops the symbol Z_0 from the stack (thus emptying the stack) and moves to an accepting state f .
- At every stage of the computation, the automaton's next step is based only on the following three items:
 1. The current state q ($q \in Q$)
 2. The current symbol a read from the input ($a \in \Sigma_\epsilon$)
 3. The symbol z at the top of the stack ($z \in \Gamma_\epsilon$)
- For every **configuration** $(q, a, z) \in Q \times \Sigma_\epsilon \times \Gamma_\epsilon$, the transition function $\delta(q, a, z)$ returns a set of zero or more pairs (r, γ) (a subset of $Q \times \Gamma^*$), where r is a new state and γ is a word over the stack alphabet.

$$\delta(q, a, z) = \{(r_1, \gamma_1), (r_2, \gamma_2), \dots, (r_n, \gamma_n)\}, \quad (r_i, \gamma_i) \in Q \times \Gamma^*$$

- Every pair $(r, \gamma) \in \delta(q, a, z)$ corresponds to a possible step in the computation, according to the following scheme:

if the automaton M

- a. is in state q ($q \in Q$),
- b. reads the symbol a ($a \in \Sigma_\epsilon$),
- c. has z at the top of its stack ($z \in \Gamma_\epsilon$),

then the automaton M performs the following operations:

1. moves to state r ($r \in Q$)
2. pops z from the stack
3. pushes the word γ onto the top of the stack ($\gamma \in \Gamma^*$)

Graphical Representation

- As with an ordinary automaton, a pushdown automaton is represented graphically by a directed labeled graph. Every pair (r, γ) in the set $\delta(q, a, z)$ corresponds to a labeled edge $q \xrightarrow{a, z/\gamma} r$ in the graph:

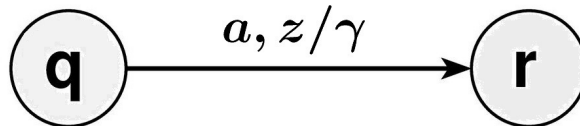


Fig. 5.2: The structure of an edge in a nondeterministic pushdown automaton: in state q ; on reading input symbol a and stack symbol z , pop z , push γ , and move to state r

Clarifications

- a. The symbol a in part b above may be the empty string ε , and therefore stack operations can be performed without reading an input symbol!
- b. The stack symbol z in part c above may be the empty string ε , and therefore a word $\gamma \in \Gamma^*$ can be pushed onto the stack without popping its top symbol!
- c. The word γ may be empty. In that case, if $z \in \Gamma$, a pop operation is performed without any push operation, and if $z = \varepsilon$, no stack operation is performed.
- d. We emphasize: the pop operation always precedes the push operation!
- e. In the extreme case in which $a = \varepsilon$, $z = \varepsilon$, $\gamma = \varepsilon$, the automaton moves from state q to state r without reading an input symbol and without performing any stack operation. Although this may appear useless, there may be situations in which such a transition simplifies the algorithm for the transition function.
- f. A given triple (q, a, z) may correspond to many edges of this kind, and therefore, at each stage of the computation, several possible actions are available (a nondeterministic automaton).

Accepting (accept) and Rejecting (reject) Input

- As with an ordinary automaton, a pushdown automaton is supplied with a word

$$w = a_1 a_2 \cdots a_n \in \Sigma^*$$

When choosing a computation path for w , the following cases may arise

- a. After all the symbols $a_1, a_2, \dots, a_n$ have been read, the computation path stops in a state $q \in F$. In this case, we say that the word w is **accepted** by the automaton M .
 - b. After all the symbols $a_1, a_2, \dots, a_n$ have been read, the computation path stops in a state $q \notin F$. In this case, we say that the word w is **rejected** or **not accepted** by the automaton M .
 - c. After the symbols $a_1, a_2, \dots, a_i, i < n$, have been read, the machine is in a situation in which the transition function does not permit it to advance to the next symbol a_{i+1} . In this case too, we say that the word w is **rejected** or **not accepted** by the automaton M (even if the automaton is in an accepting state at this stage!). We also say that the automaton has **stalled (stalled)** because the word could not be read in full.
- The set of all words w accepted by the automaton M is denoted by $L(M)$ and is called **the language accepted by M** . Two automata M, M' are called **equivalent** if $L(M) = L(M')$.
- If, for every $(q, a, z) \in Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$, the set $\delta(q, a, z)$ is empty or has a single element, then the automaton M is called a **deterministic pushdown automaton**.
- Different textbooks give many different definitions of pushdown automata (deterministic or non-



deterministic), which at first glance may seem confusing or troubling. In fact, however, all these definitions are equivalent in the sense that they define exactly the same family of languages, and the equivalence proofs are usually simple and clear.

- Recall that having several different models of computation is usually beneficial, because it gives us a wide range of possible solutions to a given problem.
- There are also books in which acceptance of an input w is defined by the existence of a computation path that reads all the symbols of w and ends with an empty stack. After gaining more knowledge, the student will be able to prove that an automaton of this kind is equivalent to our automaton (that is, it defines the same class of languages).

5.2 Additional Clarifications

- We describe in greater detail how computation operates in a pushdown automaton that accepts by an accepting state. Let w be a given word in Σ^* . At every given stage of the computation, the automaton reads one symbol from the word w (from left to right), is in a state q , and has z at the top of its stack.
- As stated above, for the configuration (q, a, z) , the transition function $\delta(q, a, z)$ returns a set of zero or more pairs (r, γ) (a subset of $Q \times \Gamma^*$), where r is a new state and γ is a word over the stack alphabet, that is, an element of Γ^* .
- According to the choice of a pair (r, γ) from the set $\delta(q, a, z)$,
 1. the automaton M moves to state r ;
 2. pop (pop) the top stack symbol z , and in its place push (push) the word γ onto the top of the stack;
 3. It may happen that $\delta(q, a, z) = \emptyset$ (the empty set). In that case, the computation terminates. If the automaton has not read all the symbols of the word or is not in an accepting state, the result of the computation is negative: the input is not accepted.
- Theoretically, the stack may be empty during the computation when a symbol must be read from it, and in that case the automaton “stalls”. In such a case, the computation terminates, because the transition function cannot be applied without a top stack symbol! This situation is regarded as termination of the computation without acceptance of the input.

i Proposition 5.1: Every regular language can be defined by a pushdown automaton.

- ✓ **Proof:** The assertion follows immediately from the simple fact that every ordinary automaton

$$M = (Q, \Sigma, \delta, F, q_0)$$



can easily be converted into an **equivalent** pushdown automaton:

$$M' = (Q, \Sigma, \Gamma, \delta', F, q_0)$$

The stack contains only one symbol: $\Gamma = \{Z_0\}$. The transition function δ' of the pushdown automaton is simply:

$$\delta'(q, a, z) = (r, Z_0), \quad r = \delta(q, a)$$

We leave the details to the student as an exercise. ■

- The question arises whether there are nonregular languages accepted by a pushdown automaton.
- In fact, we shall see that the languages accepted by pushdown automata are exactly all the context-free languages! Thus, the pushdown automaton is the model of computation for context-free languages, just as the ordinary automaton is the model of computation for regular languages.
- It is useful to view the stack as an ordinary word γ over the stack alphabet Γ

$$\gamma = \text{stack} = z_0 z_1 z_2 \dots z_n$$

The **top of the stack** z_0 is therefore the first letter in this word (which is read from left to right).

- At any given moment, the automaton **sees** only the leftmost letter in this word (z_0) and knows nothing about the other letters in the stack.
- The transition operation δ is actually an operation consisting of two parts
 - a. Popping the top of the stack (pop z_0)
 - b. Pushing (push y) a new word $y = y_0 y_1 y_2 \dots y_m$ onto the top of the stack
- After the transition, the stack has the following form:

$$\gamma = \text{stack} = y_0 y_1 \dots y_m z_1 z_2 \dots z_n$$

- After the transition operation, the new top of the stack is therefore y_0 .
- A pure pop operation occurs only when $y = \varepsilon$.
- A pure push operation occurs only when $z = \varepsilon$.
- If the stack is empty, a pop operation cannot be performed. Therefore, if the transition function calls for a pop operation, the machine stalls, and this is regarded as a state in which the input is rejected (rejected).



5.3 First Example

✦ **Example 5.1:** We construct a pushdown automaton that accepts the following language:

$$L = \{0^n 1^n : n = 1, 2, 3 \dots\}$$

In Figure 5.1, we saw an example of a computation sequence in this automaton. Despite the simple definition, in Example 3.8, we used the pumping lemma to prove that the language L is not regular, and therefore there is no ordinary automaton that accepts L . This example therefore proves that a pushdown automaton is more powerful than an ordinary automaton.

✦ **Input alphabet:** $\Sigma = \{0, 1\}$

✦ **Set of states:** $Q = \{q_0, q_1, q_2, f\}$

- q_0 is the start state. The only possible transition is to push a symbol Z_0 onto the stack and move to state q_1 .
- q_1 is the state that permits the symbol 0 to be read in a loop. Every time the symbol 0 is read, a symbol A is pushed onto the stack. In this way, the automaton “remembers” how many 0 symbols have been read. When the automaton first encounters the input 1 , it pops a symbol A from the stack and moves to state q_2 .
- q_2 permits the symbols 1 to be read. Every time the symbol 1 is read, the automaton pops a symbol A from the stack. When only Z_0 remains in the stack, the automaton pops it and moves to an accepting state f with an empty stack.
- f is the sole accepting state of the automaton.

✦ **Accepting states:** $F = \{f\}$

✦ **Stack alphabet:** $\Gamma = \{Z_0, A\}$

- Z_0 is a stack symbol used to mark the bottom of the stack.
- We use A to count the number of zeros read: every time the automaton reads a symbol 0 , we push A onto the top of the stack, and every time a symbol 1 is read, we pop A from the top of the stack.

✦ **Transition Diagram (Transition Diagram)**

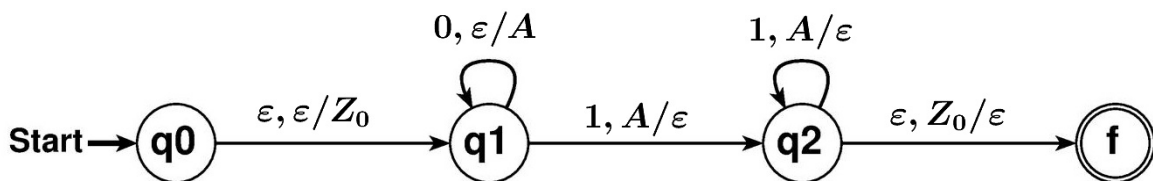


Fig. 5.3: Transition diagram for a nondeterministic pushdown automaton accepting the language $0^n 1^n$

- Note that the transition diagram does not permit the symbol 1 as input at the beginning of a computation!

- b. Also note that the transition diagram does not permit the symbol $\mathbf{0}$ to be read after entering state q_2 !
- c. The impossibility of these situations is expressed explicitly by the transition function, which returns an empty result $\emptyset$, but in the transition diagram it is implied by the absence of a suitable edge for such a transition.

➤ **Transition function:**

$\delta(q_0, \varepsilon, \varepsilon) = \{(q_0, Z_0)\}$	$\delta(q_1, \varepsilon, \varepsilon) = \emptyset$	$\delta(q_2, \varepsilon, \varepsilon) = \emptyset$
$\delta(q_0, \varepsilon, Z_0) = \emptyset$	$\delta(q_1, \varepsilon, Z_0) = \emptyset$	$\delta(q_2, \varepsilon, Z_0) = \{(f, \varepsilon)\}$
$\delta(q_0, \varepsilon, A) = \emptyset$	$\delta(q_1, \varepsilon, A) = \emptyset$	$\delta(q_2, \varepsilon, A) = \emptyset$
$\delta(q_0, 0, \varepsilon) = \emptyset$	$\delta(q_1, 0, \varepsilon) = \{(q_1, A)\}$	$\delta(q_2, 0, \varepsilon) = \emptyset$
$\delta(q_0, 0, Z_0) = \emptyset$	$\delta(q_1, 0, Z_0) = \emptyset$	$\delta(q_2, 0, Z_0) = \emptyset$
$\delta(q_0, 0, A) = \emptyset$	$\delta(q_1, 0, A) = \emptyset$	$\delta(q_2, 0, A) = \emptyset$
$\delta(q_0, 1, \varepsilon) = \emptyset$	$\delta(q_1, 1, \varepsilon) = \emptyset$	$\delta(q_2, 1, \varepsilon) = \emptyset$
$\delta(q_0, 1, Z_0) = \emptyset$	$\delta(q_1, 1, Z_0) = \emptyset$	$\delta(q_2, 1, Z_0) = \emptyset$
$\delta(q_0, 1, A) = \emptyset$	$\delta(q_1, 1, A) = \{(q_2, \varepsilon)\}$	$\delta(q_2, 1, A) = \{(q_2, \varepsilon)\}$

- As can be seen, in most configurations the transition function returns an empty result, and therefore, in most of the following examples, we do not bother to write them at all. The transition diagram is shorter and more informative, and in most cases is preferable to the transition function.
- Note that different textbooks give different definitions of a pushdown automaton that do not require the function δ to be defined at all for such configurations. In such cases, δ is defined by a **partial function**. That is, δ is not defined over the entire space of possible configurations $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$, but only over some of them (those that might be reached in some computation).
- Because the concept of a partial function is less familiar and intuitive, we chose the option of an ordinary total function. It should therefore be remembered that, for every configuration (q, a, z) for which $\delta(q, a, z) = \emptyset$, the practical meaning is one of the following two possibilities:
 1. The automaton M never reaches this configuration.
 2. If the automaton reaches this configuration, it is "stuck," and therefore the computation terminates immediately. In this situation, if the automaton M is in an accepting state, then the word w is accepted. In every other case, the computation terminates without acceptance.

➤ **Transition Table (Transition Table)**

Another way to present the transition function is by means of a transition table. For a pushdown automaton, another dimension is added to the table: the stack. Therefore, a complete description of all the automaton's states requires a three-dimensional table. To overcome this problem, the first column of the table contains all combinations $(q, z) \in Q \times \Gamma_\varepsilon$, and the first row contains the alphabet Σ_ε . For every entry (q, z) in column 1 and every input a in row 1, the corresponding table cell contains the value of $\delta(q, a, z)$. Table (5.4) gives a more focused description of the transition function for our automaton M .

δ	0	1	ϵ
q_0, Z_0	$\emptyset$	$\emptyset$	$\emptyset$
q_0, A	$\emptyset$	$\emptyset$	$\emptyset$
q_0, ϵ	$\emptyset$	$\emptyset$	$\{(q_1, Z_0)\}$
q_1, Z_0	$\emptyset$	$\emptyset$	$\emptyset$
q_1, A	$\emptyset$	$\{(q_2, \epsilon)\}$	$\emptyset$
q_1, ϵ	$\{(q_1, A)\}$	$\emptyset$	$\emptyset$
q_2, Z_0	$\emptyset$	$\emptyset$	$\{(f, \epsilon)\}$
q_2, A	$\emptyset$	$\{(q_2, \epsilon)\}$	$\emptyset$

Fig. 5.4: Transition table for the pushdown automaton M for the language $0^n 1^n$, $n = 1, 2, 3 \dots$

- This is, in fact, a **deterministic** pushdown automaton. This is clear from the table (and also from the transition diagram): in none of the cases do we have more than one possible way to proceed to the next stage of the computation (all the table cells contain singletons or empty sets). In many other cases, of course, the result of δ may well include many possible configurations, as we shall see in the following examples.
- In many cases, we prefer the transition diagram to the table, because the transition table also includes impractical configurations, and the abundance of details interferes with understanding the automaton's operation. The transition diagram, by contrast, highlights only the important details and presents a clear picture of how the automaton operates.
- **Computation Sequence**

Another important concept associated with the many kinds of automata is the **computation sequence**, which we have already encountered. A **computation sequence** for a pushdown automaton is a derivation chain of configurations. Every **configuration** consists of a triple containing the current state, the symbol to be read, and the contents of the stack.

We give only an example of a computation sequence for the word $w = 0011$

$$q_0 \xrightarrow{\epsilon, \epsilon/Z_0} q_1 \xrightarrow{0, \epsilon/A} q_1 \xrightarrow{0, \epsilon/A} q_1 \xrightarrow{1, A/\epsilon} q_2 \xrightarrow{1, A/\epsilon} q_2 \xrightarrow{\epsilon, Z_0/\epsilon} f$$

This can also be written as follows:

$$\begin{aligned} (q_0, \downarrow 0011, \epsilon) \vdash (q_1, \downarrow 0011, Z_0) \vdash (q_1, \downarrow 0011, AZ_0) \vdash (q_1, 00\downarrow 1, AAZ_0) \\ \vdash (q_2, 001\downarrow 1, AZ_0) \vdash (q_2, 0011\downarrow \epsilon, Z_0) \vdash (f, 0011, \epsilon) \end{aligned}$$

In formal expressions, the arrow must be replaced by the index number of the current input symbol. For example, instead of $(q_1, 00\downarrow 1, AAZ_0)$, we should write $(q_1, 0011, 3, AAZ_0)$. A more pictorial illustration of this computation sequence appears in Figure 5.1, which we encountered at the beginning of the chapter.

➤ General notation:

- a. To indicate briefly the existence of a computation sequence from configuration (p, w, i, α) to configuration (q, w, j, β) , write

$$(p, w, i, \alpha) \vdash^* (q, w, j, \beta)$$

where $p, q \in Q$, $u, v \in \Sigma^*$, $\alpha, \beta \in \Gamma^*$, and the indices i, j point to the current input symbol in w . The symbol $\vdash^*$ is the counterpart of the symbol $\xRightarrow{*}$ encountered in the study of context-free grammars.

- b. If there is a computation sequence for a word w that begins in state p with stack α and ends in state q with stack β , then write this as follows:

$$(p, \alpha) \xRightarrow{w} (q, \beta)$$

This is abbreviated notation for the existence of a computation sequence

$$(p, w, 0, \alpha) \vdash^* (q, w, n, \beta)$$

where $n = |w|$ is the final stage of the computation. That is, either q is an accepting state ($q \in F$), or there is no transition from the configuration (q, w, n, β) (the automaton is “stuck”).

5.4 Second Example

◆ **Example 5.2:** Define the language

$$L = \{ w \in \{0, 1\}^* : \#_0(w) = \#_1(w) \}$$

Recall that the expression $\#_0(w)$ denotes the number of zeros in the word w , and the expression $\#_1(w)$ denotes the number of ones in the word w . Thus, the language L contains all words w having equal numbers of zeros and ones.

- It is easy to prove (using the pumping lemma) that the language L is not regular. In this example, we must construct a pushdown automaton M that accepts L .

➤ **Input alphabet:** $\Sigma = \{0, 1\}$

➤ **Set of states:** $Q = \{q_0, f\}$

- a. q_0 is the start state, from which Z_0 can be pushed onto the stack, followed immediately by a transition to state q_1 .
- b. q_1 is the state in which most cases are handled (see Figure 5.5).



- c. f denotes the accepting state of the automaton and can be reached through only one edge, in a balanced situation in which the number of zeros read equals the number of ones read.

➤ **Accepting states:** $F = \{f\}$

➤ **Stack alphabet:** $\Gamma = \{Z_0, A, B\}$

- Z_0 is the initial stack symbol and also indicates a situation in which the number of zeros read so far equals the number of ones.
- A indicates one excess zero (that is, the number of zeros read is one greater than the number of ones read).
- B indicates one excess symbol 1 (that is, the number of ones read is one greater than the number of zeros read).
- Every time the symbol 0 is read, inspect the top of the stack. If the top symbol is B , pop it and proceed to the next symbol (“cancellation”). If the top of the stack is Z_0 or A , push A onto the stack and proceed to the next symbol.
- Every time the symbol 1 is read, inspect the top of the stack. If the top symbol is A , pop it and proceed to the next symbol (“cancellation”). If the top of the stack is Z_0 or B , push B onto the stack and proceed to the next symbol.

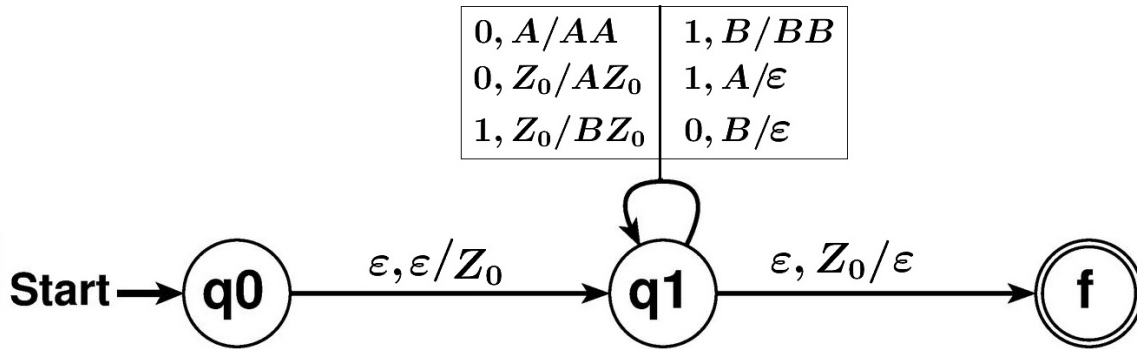


Fig. 5.5: Transition diagram for the pushdown automaton M

➤ **Transition function:**

$$\begin{aligned}
 \delta(q_0, \epsilon, \epsilon) &= \{(q_1, Z_0)\} \\
 \delta(q_1, 0, Z_0) &= \{(q_1, AZ_0)\} \\
 \delta(q_1, 0, A) &= \{(q_1, AA)\} \\
 \delta(q_1, 0, B) &= \{(q_1, \epsilon)\} \\
 \delta(q_1, 1, Z_0) &= \{(q_1, BZ_0)\} \\
 \delta(q_1, 1, A) &= \{(q_1, \epsilon)\} \\
 \delta(q_1, 1, B) &= \{(q_1, BB)\} \\
 \delta(q_1, \epsilon, Z_0) &= \{(f, \epsilon)\}
 \end{aligned}$$

In every other configuration, the transition function returns the empty set.

➤ A computation sequence for the word $w = 0110$ is

$$\begin{aligned} (q_0, \overset{\downarrow}{\varepsilon}0110, \varepsilon) \vdash (q_1, \overset{\downarrow}{0}110, Z_0) \vdash (q_1, 0\overset{\downarrow}{1}10, AZ_0) \vdash (q_1, 01\overset{\downarrow}{1}0, Z_0) \\ \vdash (q_1, 011\overset{\downarrow}{0}, BZ_0) \vdash (q_1, 0110\overset{\downarrow}{\varepsilon}, Z_0) \vdash (f, 0110, Z_0) \end{aligned}$$

➤ In Example 4.6, we encountered the same language L and constructed a context-free grammar G for it

$$\begin{aligned} V &= \{S, A, B\} \\ T &= \{0, 1\} \end{aligned} \quad \left\{ \begin{array}{l} S \rightarrow 0B|1A \\ A \rightarrow 0|0S|1AA \\ B \rightarrow 1|1S|0BB \end{array} \right.$$

A quick glance reveals a certain similarity between the grammar G and the automaton M . Later in the chapter, we shall see that a pushdown automaton is in fact a computational model equivalent to a context-free grammar, and we shall also present an algorithm showing how to pass from a grammar to an automaton and vice versa.

5.5 Elementary Pushdown Automata

➤ **Definition 5.2:** A pushdown automaton M is called **elementary** if it satisfies the following three conditions

- a. There is a single accepting state f .
- b. Whenever a computation ends in an accepting state, the stack is emptied.
- c. Every transition rule may push only one symbol onto the stack or pop one symbol (but not perform both operations together!).

➤ In some books, as well as in commonly used software systems, a pushdown automaton is defined in this manner. This greatly simplifies code management (pushing a string of symbols onto the stack complicates management of the automaton). From a theoretical perspective, however, the general definition is preferable because it permits shorter proofs and shorter computation sequences, and allows a pushdown automaton to be specified more concisely.

➤ In this section, we prove that the two definitions are in fact equivalent. In the next section, we use this fact to prove the equivalence of pushdown automata and context-free grammars.

i Theorem 5.2: For every pushdown automaton M , there is an equivalent elementary pushdown automaton M' . That is, $L(M) = L(M')$.

✓ **Proof:** Let $M = (Q, \Sigma, \Gamma, \delta, F, q_0)$ be a given pushdown automaton. We construct an elementary pushdown automaton $M' = (Q', \Sigma, \Gamma, \delta', \{f\}, q_0)$ by means of the following operations. For every edge $q \xrightarrow{a, z/\gamma} r$ in the automaton M , define several new edges in the automaton M' that perform the same operation.

- a. If $\gamma = \varepsilon$, this is an operation that pops the top of the stack and is suitable for an elementary automaton, so such an edge remains in the automaton M' .
- b. If $\gamma = z_1 z_2 \cdots z_k$ is a string of length $k \geq 1$, then add to Q' $k + 1$ new states: $r_0, r_1, r_2, \dots, r_k$, and replace the edge $q \xrightarrow{a, z/\gamma} r$ with the following $k + 1$ edges
 - ♦ $q \xrightarrow{a, z/\varepsilon} r_0$ (pop the symbol z)
 - ♦ $r_0 \xrightarrow{\varepsilon, \varepsilon/z_1} r_1$ (push the symbol z_1)
 - ♦ $r_1 \xrightarrow{\varepsilon, \varepsilon/z_2} r_2$ (push the symbol z_2)
 - ♦ $\dots$
 - ♦ $r_k \xrightarrow{\varepsilon, \varepsilon/z_k} r$ (push the symbol z_k)
- c. Note that the two preceding parts are also valid when $a = \varepsilon$!
- d. If $z = \varepsilon$, but $\gamma = z_1 z_2 \cdots z_k$ is a string of length $k \geq 1$, then proceed exactly as in part b, except for the pop operation, which is not needed here.
- e. If $z = \varepsilon$ and also $\gamma = \varepsilon$, then add to Γ an additional symbol z_0 , and add to the set of states Q a new state r_0 , and decompose the edge $q \xrightarrow{a, \varepsilon/\varepsilon} r$ into two edges:

$$q \xrightarrow{a, \varepsilon/z_0} r_0, \quad r_0 \xrightarrow{\varepsilon, z_0/\varepsilon} r$$

It is easy to verify that the new automaton M' is an elementary pushdown automaton equivalent to M ; that is, $L(M') = L(M)$. ■

5.6 Equivalence of Context-Free Grammars and Pushdown Automata

- Let L be the language generated by a context-free grammar G . In this section, we construct a pushdown automaton M that accepts L . Later, we prove that the converse is also true: if L is a language accepted by a pushdown automaton M , then there is a context-free grammar G that generates L .
- The proof of the first direction is easier to understand with the aid of a preliminary example. Return to the grammar G from Example 4.6 (page 114)

$$\begin{array}{lcl} V & = & \{S, A, B\} \\ T & = & \{0, 1\} \end{array} \quad \left\{ \begin{array}{l} S \rightarrow 0B|1A \\ A \rightarrow 0|0S|1AA \\ B \rightarrow 1|1S|0BB \end{array} \right.$$

We proved that the grammar G generates the language of "balanced" words (equal numbers of zeros and ones)

$$L = \{w \in \{0, 1\}^* : \#_0(w) = \#_1(w)\}$$



For example, the word $w = 01110010$ belongs to the language L because it has four zeros and four ones. Consider its tsomtfel derivation sequence

$$\begin{aligned} S &\Rightarrow 0\textcolor{red}{B} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{S} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{A} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{A}\textcolor{red}{A} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}0\textcolor{red}{S}\textcolor{red}{A} \\ &\Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}00\textcolor{red}{B}\textcolor{red}{A} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}00\textcolor{red}{1}\textcolor{red}{A} \Rightarrow 0\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}00\textcolor{red}{1}0 \end{aligned}$$

We now show the computation path of w in the pushdown automaton M defined by

$$\Sigma = \{0, 1\}, \quad \Gamma = \{0, 1, S, A, B, Z_0\}, \quad Q = \{q_0, q_1, f\}$$

- a. In state q_0 , the automaton pushes SZ_0 onto the stack and moves to state q_1 .
- b. In state q_1 , the automaton simulates the derivation sequence of the input w by applying a grammar rule $X \Rightarrow \alpha$ (corresponding to the derivation sequence of w) to the top of the stack. Whenever the top of the stack is a syntactic variable X , it pops that variable, pushes the sentential form α , and remains in state q_1 . If the top of the stack is a terminal symbol a identical to the current input symbol, the automaton reads a and pops a from the stack. The state of the automaton remains q_1 .
- c. When the top of the stack is Z_0 , the automaton pops it and moves to an accepting state f . A complete specification of the transition function of M is given later in the proof of Assertion 5.3.

► We illustrate this process by means of the complete computation sequence for $w = 01110010$

$$\begin{aligned} (q_0, \textcolor{red}{0}1110010, \varepsilon) &\vdash (q_1, \textcolor{red}{0}1110010, SZ_0) && \vdash (q_1, \textcolor{red}{0}1110010, \textcolor{red}{0}BZ_0) \\ &\vdash (q_1, 0\textcolor{red}{1}110010, BZ_0) && \vdash (q_1, 0\textcolor{red}{1}\textcolor{red}{1}10010, \textcolor{red}{1}SZ_0) \\ &\vdash (q_1, 01\textcolor{red}{1}10010, SZ_0) && \vdash (q_1, 01\textcolor{red}{1}\textcolor{red}{1}0010, \textcolor{red}{1}AZ_0) \\ &\vdash (q_1, 011\textcolor{red}{1}0010, AZ_0) && \vdash (q_1, 011\textcolor{red}{1}\textcolor{red}{1}0010, \textcolor{red}{1}AAZ_0) \\ &\vdash (q_1, 0111\textcolor{red}{0}010, AAZ_0) && \vdash (q_1, 0111\textcolor{red}{0}010, \textcolor{red}{0}SAZ_0) \\ &\vdash (q_1, 01110\textcolor{red}{0}10, SAZ_0) && \vdash (q_1, 01110\textcolor{red}{0}10, \textcolor{red}{0}BAZ_0) \\ &\vdash (q_1, 011100\textcolor{red}{1}0, BAZ_0) && \vdash (q_1, 011100\textcolor{red}{1}0, \textcolor{red}{1}AZ_0) \\ &\vdash (q_1, 0111001\textcolor{red}{0}, AZ_0) && \vdash (q_1, 0111001\textcolor{red}{0}, \textcolor{red}{0}Z_0) \\ &\vdash (q_1, 01110010\textcolor{red}{\varepsilon}, Z_0) && \vdash (f, 01110010, \varepsilon) \end{aligned}$$

The first ten configurations of the computation sequence are shown in Figures 5.6, 5.7.

i Proposition 5.3: Let L be the language generated by a context-free grammar G . There is a pushdown automaton M that accepts L .

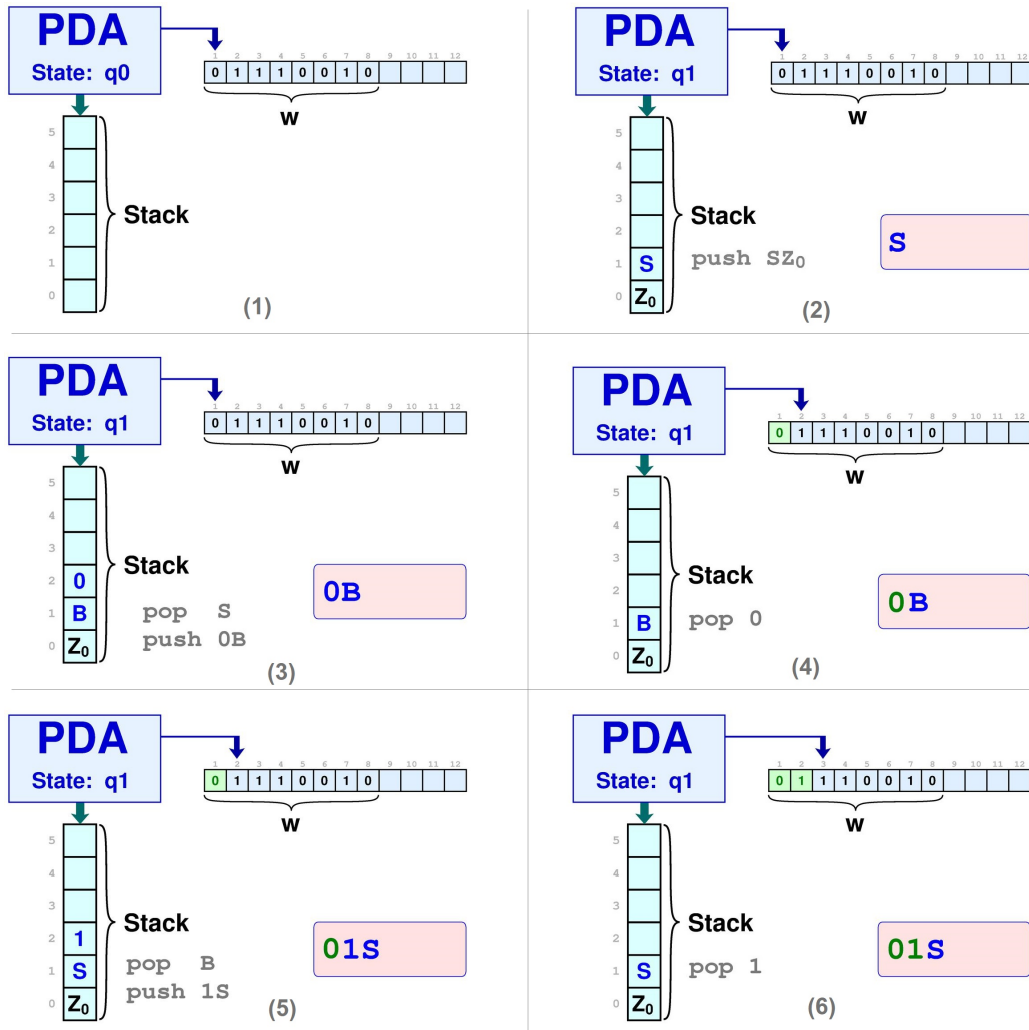


Fig. 5.6: Simulation of a derivation sequence in the grammar G by a pushdown automaton, stages 1–6

✓ **Proof:** Construct a pushdown automaton M that accepts M as follows:

◆ **Input alphabet:**

$$\Sigma = \{ a : G \text{ rammarg eht fo lobmys lanimret a si } a \}$$

◆ **Stack alphabet:**

$$\Gamma = \{ z : G \text{ rammarg eht fo elbairav citcatnys ro lobmys lanimret a si } z \} \cup \{ Z_0 \}$$

The stack alphabet includes all the variables of the grammar, the terminal symbols T , and Z_0 , which marks the bottom of the stack.

◆ The idea is to construct a pushdown automaton M in which every computation sequence is identical to a derivation sequence in the grammar G . At every stage of the computation, the combination of the symbols already read and the stack symbols is a sentential form in the derivation sequence. Thus, a word w is accepted by M if and only if there is a derivation sequence in G ending with the word w .

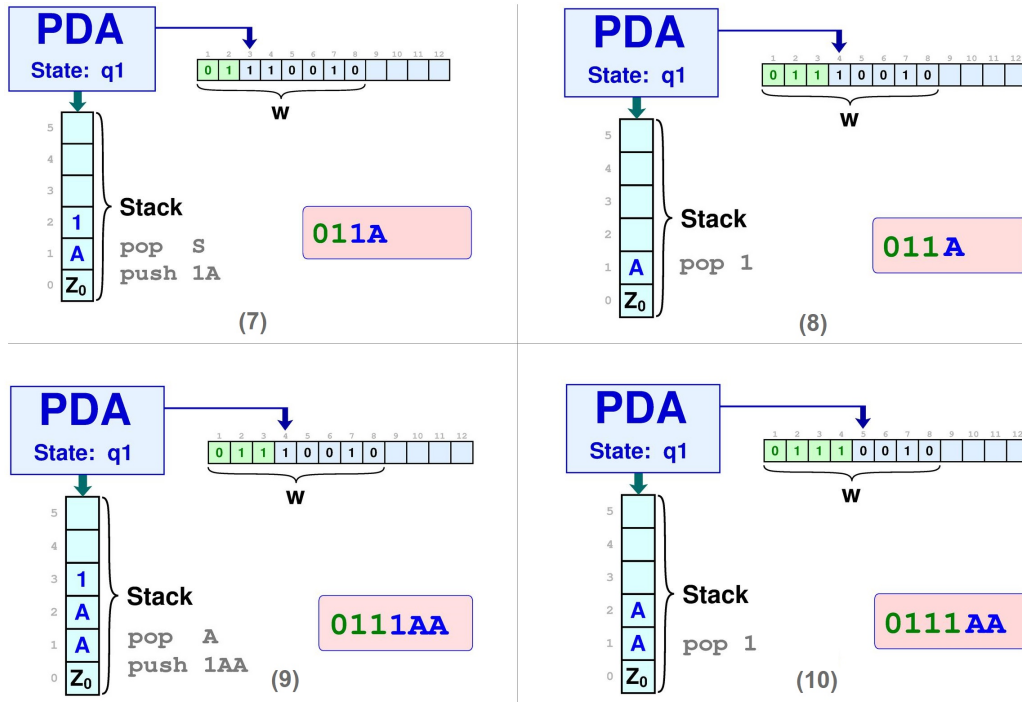


Fig. 5.7: Simulation of a derivation sequence in the grammar G by a pushdown automaton, stages 7–10

◆ Set of states: $Q = \{q_0, q_1, f\}$

q_0 is the start state, in which the start variable S is pushed onto the stack and the automaton moves to state q_1 . Most of the work is performed in state q_1 by means of the following loop:

- a. If the top of the stack is a syntactic variable X , define the transition function by

$$\delta(q_1, \varepsilon, X) = \{(q_1, \alpha) : \alpha \in \Gamma^*, X \rightarrow \alpha\}$$

That is, in state q_1 , if the top of the stack is the variable X , then, without reading any input, we can pop X and push a sentential form α onto the stack, where $X \rightarrow \alpha$ is a production in the grammar G . After the operation, we remain in state q_1 . There may, of course, be several productions of this kind, so our automaton is nondeterministic.

- b. If the current input symbol is a and the top of the stack is also a , then the automaton reads a from the input and pops a from the stack. The automaton remains in state q_1 . In terms of the transition function, this is written as

$$\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$$

Recall that the empty word ε indicates that after popping a , nothing is pushed onto the stack.

- c. If the top of the stack is a terminal symbol b different from the current input symbol a , the automaton stalls (that is, the input is rejected). In other words, for every terminal symbol $b \neq a$,

$$\delta(q_1, a, b) = \emptyset$$

- d. If the top of the stack is Z_0 , move to an accepting state f by an ε -transition

$$\delta(q_1, \varepsilon, Z_0) = \{(f, \varepsilon)\}$$

Figure 5.8 summarizes the four items in a single picture.

◆ **Accepting states:** $F = \{f\}$

f denotes the sole accepting state of the automaton and can be reached only after all input symbols have been read and the stack contains only the symbol Z_0 .

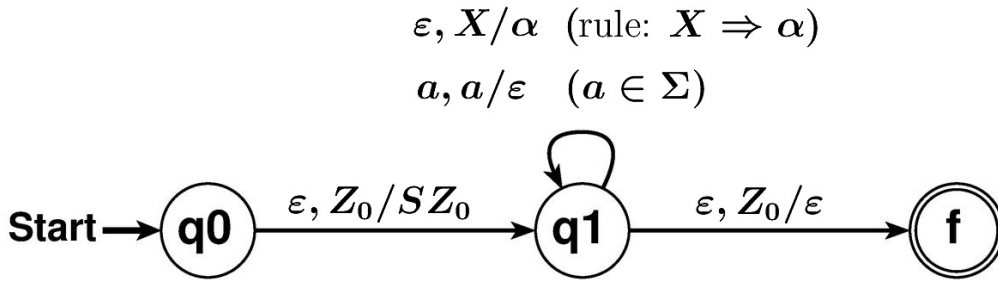


Fig. 5.8: Transition diagram for the pushdown automaton M that accepts the language $L(G)$ for a context-free grammar G

- ◆ We must prove that $L(M) = L(G)$. There are two directions here:

Direction 1: If $w \in L(G)$, then $w \in L(M)$

Direction 2: If $w \in L(M)$, then $w \in L(G)$

- ◆ We begin by proving Direction 1: Let w be a word in the language $L(G)$. We must prove that w is accepted by the automaton M . By Theorem 4.15, there exists a leftmost derivation sequence that yields the word w

$$S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_m = w \quad (5.1)$$

We convert this derivation sequence into a computation sequence of w in the automaton M , according to items a–d in the definition of its transition function listed above. It is important to remember that whenever the top of the stack is a syntactic variable X , the production corresponding to the derivation in (5.1) must be chosen. We prove by induction on the length of the computation sequence that the concatenation of the symbols read by the automaton with the contents of the automaton's stack (without Z_0) is identical to the sentential form α_k or α_{k+1} in the derivation sequence (5.1), where k denotes the index of the current symbol being read by the automaton.

Induction basis: The automaton M starts with an empty stack and looks at the first symbol of w . According to the first member of the derivation, $\alpha_1 = S$, we must push SZ_0 onto the stack, where S is the start variable of the grammar G , and Z_0 denotes the bottom of the stack.

Step n of the computation sequence: Let $u = a_1a_2 \cdots a_i$ be the string of symbols read by the automaton up to step n of the computation, let a be the current symbol seen by the automaton, and let $\gamma = x_1x_2 \cdots x_j$ be the stack contents (without Z_0). The induction hypothesis is that

there exists k such that $\alpha_k = u\gamma$. We examine two possibilities:

- a. If $x_1 = X$ is a syntactic variable of the grammar G , then it is the leftmost variable in the sentential form $\alpha_k = u\gamma$, and since the derivation sequence (5.1) is leftmost, it follows that there exists a production $X \rightarrow \beta$ in the grammar G that justifies the derivation $\alpha_k \Rightarrow \alpha_{k+1}$ in (5.1). According to the definition of our transition function, the automaton performs an ε -transition by popping the symbol X from the stack and pushing β in its place. Therefore, at step $n+1$ of the computation, the string of symbols read is still u (because this was an ε -transition), and its concatenation with the stack contents is $u\beta x_2 x_3 \cdots x_j = \alpha_{k+1}$.
- b. If $x_1 = b$ is a terminal symbol, then it must be identical to the current symbol a . This follows from $\alpha_k = u\gamma$, and hence the string ub must be a prefix of every sentential form α_ℓ for every $k \leq \ell \leq m$. Since $\alpha_m = w$, and ua is also a prefix of w , we must have $a = b$. According to the definition of the automaton's transition function, the symbol a must be read and the symbol a popped from the stack. Therefore, at step $n+1$ of the computation, the concatenation of the symbols read with the stack contents is $uax_2 \cdots x_j = \alpha_k$.

It is clear from the preceding analysis that the automaton M must finish the computation by reading the word w , and thus we have proved that $L(G) \subseteq L(M)$.

- ◆ We now turn to the proof of Direction 2: Let $w \in L(M)$. We must prove that $w \in L(G)$. Let $\{(p_i, w, k_i, \gamma_i)\}_{i=1}^n$ be a computation sequence of w . We convert it into a derivation sequence of w in the grammar G . At step $i = 1$, the automaton is in the start state $p_1 = q_0$, with an empty stack $\gamma = \varepsilon$. According to the definition of the transition function, it must move to state q_1 and push $\gamma = SZ_0$ onto the stack. Therefore, the first sentential form in our derivation sequence is $\alpha_1 = S$. At every step $i > 1$, let $u = w[1 : k_i - 1]$ be the string of all symbols in w that have already been read during the computation, and let $\gamma_i = x_1 \cdots x_j$ be the stack contents. If $x_1 = X$ is a syntactic variable of the grammar G , then the next sentential form in the derivation sequence is $\alpha = u\gamma_i$. According to the definition of the transition function, we obtain a derivation sequence that ends in w , and therefore $w \in L(G)$. ■

i Proposition 5.4: Let L be the language accepted by a pushdown automaton M . Then L is a context-free language.

- ♥ **Proof:** Let $M = (Q, \Sigma, \Gamma, \delta, F, q_0)$ be a given pushdown automaton. By Theorem 5.2, we may assume that M is an elementary pushdown automaton. That is:

- a. There is a single accepting state f .
- b. Whenever a computation ends in an accepting state, the stack is emptied.
- c. In each transition rule, only one symbol may be pushed onto the stack, or one symbol may be popped (but not both operations together!).

We construct a context-free grammar G that generates $L(M)$ as follows:

- a. For every pair of states $p, q \in Q$, there is a corresponding syntactic variable X_{pq} .
- b. The start variable is $X_{q_0 f}$.



- c. For every state $q \in Q$, there is a corresponding production $X_{pp} \rightarrow \varepsilon$.
- d. For every three states $p, q, r \in Q$, there is a corresponding production

$$X_{pr} \rightarrow X_{pq}X_{qr}$$

- e. For every four states $p_1, q_1, q_2, p_2 \in Q$, for every stack symbol $z \in \Gamma$, and for every $a, b \in \Sigma_\varepsilon$, if the transition rules of M include the transitions

$$\begin{array}{ccc} p_1 & \xrightarrow{a, \varepsilon/z} & q_1 \\ q_2 & \xrightarrow{b, z/\varepsilon} & p_2 \end{array}$$

then we add to the grammar G the production

$$X_{p_1 p_2} \rightarrow aX_{q_1 q_2}b$$

The meaning of the production is: in state p_1 , the automaton M encounters a in the input and pushes z onto the stack. Several intermediate moves are performed, and before it moves to state p_2 , the automaton reads b and pops z .



Fig. 5.9: The scenario required for adding the production $X_{p_1 p_2} \rightarrow aX_{q_1 q_2}b$ to the grammar G

We prove that $L(G) = L(M)$. For this purpose, we prove both directions:

1. If the word $w \in \Sigma^*$ is generated by the grammar G , then it is also accepted by the automaton M .
2. If the word $w \in \Sigma^*$ is accepted by the automaton M , then $X_{q_0 f} \xRightarrow{*} w$ (that is, w is generated by G).

To this end, we prove two stronger claims:

◆ **Claim 1:** If the word $w \in \Sigma^*$ is generated by the variable X_{pq} , that is, $X_{pq} \xRightarrow{*} w$, then there exists a computation sequence of w that starts in state q with an empty stack and ends in state q with an empty stack.

◆ **Claim 2:** If, in the automaton M , there exists a computation sequence of w that starts in state q with an empty stack and ends in state q with an empty stack, then $X_{pq} \xRightarrow{*} w$.

The first direction follows immediately from Claim 1: If $w \in \Sigma^*$ is generated by the grammar G , that is, $X_{q_0 f} \xRightarrow{*} w$, then, by Claim 1, there exists a computation sequence for w that starts in state q_0 and ends in an accepting state f . But this is precisely the definition of a word accepted by the automaton M ! Similarly, the first direction follows from Claim 2. It therefore

remains to prove both claims.

- ◆ The proof of Claim 1 is by induction on the length of the derivation sequence of w . At the induction basis, we have a derivation of length 2: $X_{pq} \rightarrow w$. The only derivation of this kind permitted by G is of the form $X_{pp} \rightarrow \varepsilon$. Therefore, $w = \varepsilon$. Clearly, for the pushdown automaton M in state p with an empty stack, a computation sequence of length 1 yields the empty word $w = \varepsilon$. Thus, the induction basis is trivially true.
- ◆ Now assume that we have a derivation sequence $X_{pq} \Rightarrow w$ of length $n + 1$, for $n \geq 2$. The first step in the sequence must be of the form $X_{pq} \rightarrow aX_{rs}b$ or $X_{pq} \rightarrow X_{pr}X_{rq}$. In the first case, we can extract from our derivation a derivation of length n : $X_{rs} \Rightarrow u$, where $w = aub$. By the induction hypothesis, there exists in the automaton M a computation sequence of the word u that starts in state r with an empty stack, and ends in state s with an empty stack. As we have learned, we denote this computation sequence by $(r, \varepsilon) \xRightarrow{u} (s, \varepsilon)$. It follows from this statement that if the stack γ is not empty, then it does not change as a result of the computation. That is: $(r, \gamma) \xRightarrow{u} (s, \gamma)$. In addition, there exists a stack symbol $z \in \Gamma$ such that $p \xrightarrow{a, \varepsilon/z} r$, $s \xrightarrow{b, z/\varepsilon} q$. Concatenating the last three sequences yields a computation sequence $(p, \varepsilon) \xRightarrow{w} (q, \varepsilon)$:

$$(p, \varepsilon) \xRightarrow{w} (q, \varepsilon) = p \xrightarrow{a, \varepsilon/z} r + (r, \{z\}) \xRightarrow{u} (s, \{z\}) + s \xrightarrow{b, z/\varepsilon} q$$

In the second case, $X_{pq} \rightarrow X_{pr}X_{rq}$, there exist two derivation sequences $X_{pr} \Rightarrow u$, $X_{rq} \Rightarrow v$, of length at most n , such that $w = uv$. By the induction hypothesis, there exist two computation sequences $(p, \varepsilon) \xRightarrow{u} (r, \varepsilon)$, $(r, \varepsilon) \xRightarrow{v} (q, \varepsilon)$. Concatenating them yields the computation sequence $(p, \varepsilon) \xRightarrow{w} (q, \varepsilon)$ and completes the proof of Claim 1.

- ◆ We now turn to the proof of Claim 2: Let $w \in \Sigma^*$ such that $(p, \varepsilon) \xRightarrow{w} (q, \varepsilon)$. We must show that the grammar G contains a derivation sequence $X_{pq} \xRightarrow{*} w$. We prove this by induction on the length of the derivation. At the induction basis, the computation sequence of length $(p, \varepsilon) \xRightarrow{\varepsilon} (p, \varepsilon)$ yields the empty word $w = \varepsilon$. By definition, the grammar G contains a production $X_{pp} \rightarrow \varepsilon$.

Let $(p, \varepsilon) \xRightarrow{w} (q, \varepsilon)$ be a computation sequence of length $n + 1$. Since the stack is empty at the start of the computation, the first step must perform a push operation onto the stack, $p \xrightarrow{a, \varepsilon/z} r$. Since the stack is empty at the end of the computation, the last step of the computation sequence must perform a pop operation $s \xrightarrow{b, y/\varepsilon} q$.

Possibility 1: $y = z$, and throughout the computation the symbol z is not popped until the end of the computation. Then we can describe our computation sequence as follows:

$$p \xrightarrow{a, \varepsilon/z} r, \quad (r, \{z\}) \xRightarrow{u} (s, \{z\}), \quad s \xrightarrow{b, z/\varepsilon} q$$

where $w = aub$. Since the symbol z remains on the stack until the end of the computation,



we may also write $(r, \varepsilon) \xRightarrow{u} (s, \varepsilon)$. The length of the computation sequence $(r, \varepsilon) \xRightarrow{u} (s, \varepsilon)$ is $n - 1$, and by definition the grammar G contains a production $X_{pq} \rightarrow aX_{rs}b$. Therefore, by the induction hypothesis, $X_{rs} \xRightarrow{*} u$, and hence $X_{pq} \xRightarrow{*} w$.

Possibility 2: the stack symbol z is popped in the middle of the computation. In this case, we can decompose our computation sequence into two computation subsequences

$$(p, \varepsilon) \xRightarrow{u} (r, \varepsilon) \xRightarrow{v} (q, \varepsilon)$$

where u is the shortest subword of w for which the stack is not emptied in the middle of the computation $(p, \varepsilon) \xRightarrow{u} (r, \varepsilon)$. By its definition, the grammar G contains the production $X_{pq} \rightarrow X_{pr}X_{rq}$. By the induction hypothesis, $X_{pr} \xRightarrow{*} u$, $X_{rq} \xRightarrow{*} v$, and therefore

$$X_{pq} \rightarrow X_{pr}X_{rq} \xRightarrow{*} uv = w$$

■

An immediate consequence of Claims 5.3, 5.4, is the following theorem.

i Theorem 5.5: A language L is context-free if and only if there exists a pushdown automaton M such that $L = L(M)$.

➤ An immediate consequence of this theorem is

i Theorem 5.6: Every regular language is context-free.

✓ **Proof:** Let L be a regular language. By Definition 2.6, there exists a finite automaton M that accepts L . By Claim 5.1, there exists a pushdown automaton M' that accepts L . Therefore, by the preceding theorem, L is also a context-free language. The grammar that generates L is called a **regular grammar**, and it is described at the beginning of the chapter. ■

➤ Of course, the converse is not true: there exist context-free languages that are not regular. One of the simplest examples is the language $L = \{0^n 1^n : n \in \mathbb{N}\}$ that we encountered in Example 1. It is easy to verify that the language is not regular by using the pumping lemma (see Example 3.8).

➤ In the preceding chapter, we saw that the intersection of two context-free languages does not necessarily yield a context-free language. If one of the languages is regular, then it can be shown that the intersection is necessarily context-free.

i Theorem 5.7: Let L_1 be a context-free language, and let L_2 be a regular language. Then the language $L_1 \cap L_2$ is context-free.

✓ **Proof:** Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, F_1, p_0)$ be a pushdown automaton for L_1 , and let $M_2 = (Q_2, \Sigma, \delta_2, F_2, q_0)$ be a deterministic automaton for the language L_2 . We construct a pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, F, r_0)$ for the language $L_1 \cap L_2$. The state set is $Q_1 \times Q_2$, the start state is $r_0 = (p_0, q_0)$, and the set of accepting states is $F = F_1 \times F_2$. The transition



function δ is defined as follows: For every $((p, q), a, z) \in Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$,

$$\delta((p_1, p_2), a, z) = \{ ((q_1, q_2), \gamma) \in Q \times \Gamma^* : (q_1, \gamma) \in \delta_1(p_1, a, z) \wedge \delta_2(p_2, a) = q_2 \}$$

In addition, we require that if $a = \varepsilon$, then $p_2 = q_2$. The automaton M simulates (simulate) the operation of both automata M_1, M_2 , in parallel, thereby ensuring that the input is accepted by both automata M_1, M_2 . It is not difficult to prove by induction on the length of the input w that

$$((p_0, q_0), w, z) \vdash^* ((p, q), \varepsilon, y)$$

if and only if

$$(p_0, w, z) \vdash^* (p, \varepsilon, y) \quad \text{and} \quad \delta_2^*(q_0, w) = q$$

Therefore, the input w is accepted by M if and only if it is accepted by both M_1 and M_2 . Therefore, $L(M) = L(M_1) \cap L(M_2)$. ■

Exercises for chapter 5

1. Let L be the language of all strings w over the binary alphabet that have odd length and whose middle symbol is 1.
 - (a) Find a context-free grammar for L .
 - (b) Try to give an informal description of a pushdown automaton for L .

2. Let L be the language

$$L = \{ 0^i 1^j : 0 \leq i \leq j \leq 2i \}$$

Prove that L is context-free by constructing a nondeterministic pushdown automaton for it.

3. Construct a pushdown automaton for the language of arithmetic expressions over the alphabet $\Sigma = \{a, b, +, *, (,)\}$.

Hint: The automaton should accept expressions such as $a * (a + b)$, $a * a$, and so on.

4. A string w is called a **palindrome** if $w = w^R$. That is, w is identical to its reversal. Examples: $w = abba$, $w = abacaba$. Let L be the language of all palindromes over the alphabet $\{0, 1\}$. Prove that L is context-free by constructing a pushdown automaton for L .
5. Continuing Exercise 36 from the preceding chapter, let

$$L = \{ a^i b^j c^k : i = j \vee j = k, i, j, k \geq 0 \}$$

Give an informal description of a pushdown automaton for the language L .



6. Let G be the following context-free grammar

$$\begin{aligned} S &\rightarrow 0A1|1A0 \\ A &\rightarrow BAB|B|\varepsilon \\ B &\rightarrow 0|1 \\ C &\rightarrow BCB|S \end{aligned}$$

- Find three distinct words generated by G .
 - Find three distinct words that are not generated by G .
 - Give an informal verbal description of the language $L(G)$.
 - Convert the grammar G into an equivalent pushdown automaton according to the algorithm presented in this chapter.
7. Define a context-free grammar G by the productions

$$\left\{ \begin{array}{l} \text{statement} \rightarrow \text{assign} \mid \text{ifthen} \mid \text{ifthenelse} \\ \text{ifthen} \rightarrow \text{if condition then statement} \\ \text{ifthenelse} \rightarrow \text{if condition then statement else statement} \\ \text{assign} \rightarrow a := 1 \mid a := 2 \end{array} \right.$$

The start variable is `statement`. The set of terminal symbols is

$$T = \{\text{if, condition, then, else, } a := 1, a := 2\}$$

and the set of syntactic variables is

$$V = \{\text{statement, assign, ifthen, ifthenelse}\}$$

- Prove that G is an ambiguous grammar.
- Find an unambiguous grammar for the language it generates.
- Give an informal description of a pushdown automaton for the language $L(G)$.



8. In Exercise 32 of the preceding chapter, you were asked to find a context-free grammar G for the following language

$$L = \{a^i b^j c^n : i + j = n\}$$

Convert the grammar G into an equivalent pushdown automaton according to the algorithm presented in this chapter.



9. Prove that every context-free language over a one-symbol alphabet $\Sigma = \{a\}$ is regular.



10. Examine the transition diagram shown in Figure 5.10, for a pushdown automaton M , and try to determine the language accepted by M . Give an informal description of the language, and find a context-free grammar for it.

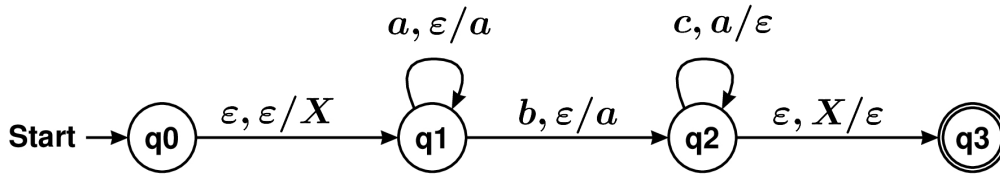


Fig. 5.10: Pushdown automaton diagram for Exercise 10



11. Examine the transition diagram shown in Figure 5.11, for a pushdown automaton M , and try to determine the language accepted by M .

Hint: Note that the stack alphabet is identical to the input alphabet.

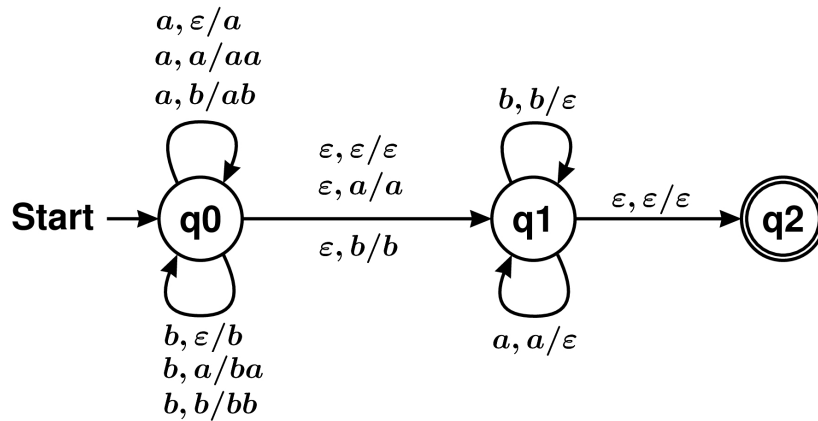


Fig. 5.11: Pushdown automaton diagram for Exercise 11



12. Let L_1 , L_2 be two languages over a common alphabet Σ , accepted by pushdown automata M_1 , M_2 . Construct a pushdown automaton that accepts the language $L_1 \cup L_2$.

Hint: You should use the machines $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, F_1, s_1)$, $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, F_2, s_2)$. For example, the state set Q of the new automaton should be based on the state sets Q_1 , Q_2 , and so on.



13. Let L_1 , L_2 be two languages over a common alphabet Σ , accepted by pushdown automata M_1 , M_2 . Construct a pushdown automaton that accepts the language $L_1 L_2$.



14. Let L_1 , L_2 be two languages over a common alphabet Σ . Let M_1 be a finite automaton that accepts the language L_1 , and let M_2 be a pushdown automaton that accepts M_2 . Construct a pushdown automaton that accepts the language $L_1 \cap L_2$.

Hint: That is, L_1 is a regular language, and L_2 is a context-free language. At the end of the preceding chapter, we learned that the intersection of a regular language and a context-free language yields a context-free language. In this exercise, the student is asked to construct an automaton for $L_1 \cap L_2$ by using automata for L_1 , L_2 .



15. There is another model of a pushdown automaton called a **pushdown automaton with acceptance by empty stack**. Unlike the previous model, this automaton has no accepting states (only ordinary states). At the start of the run, the stack contains only the symbol Z_0 . The in-



put is accepted if and only if, after slobmys tupni lla gninnacs, the stack becomes empty. Prove that this model of a pushdown automaton is equivalent to our model.



Chapter 6

Turing Machines

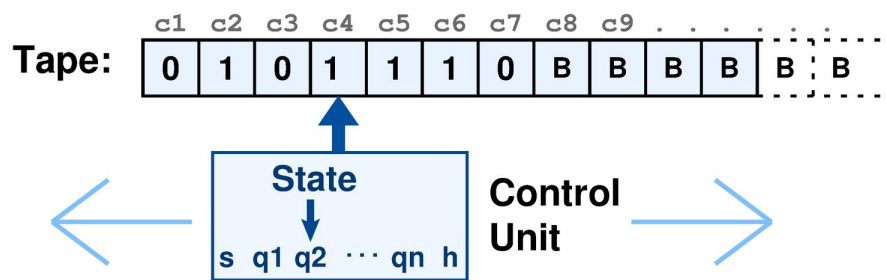


Fig. 6.1: A Turing machine with start state s , halt state h , intermediate states $\{q_1, q_2, \dots, q_n\}$, $\Gamma = \{0, 1, B\}$

6.1 Definition

- The Turing machine is the first theoretical computational model that most closely resembles the modern concept of a computer in its structure and computational capability. It was developed by the mathematician and computer scientist Alan Turing (1912-1954), who is regarded as a founder of theoretical computer science and artificial intelligence. Turing developed this model in 1936 as part of an attempt to give a mathematical definition of an algorithm and of a general computational process.
- Unlike the preceding models, which are based on an input acceptance/rejection architecture, the minimal architecture of a Turing machine is based on input/output.

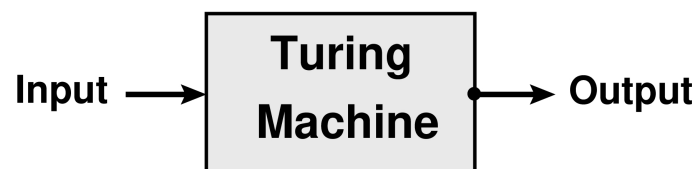


Fig. 6.2: The minimal architecture of a Turing machine

The input is a string over an alphabet Σ , and the output is a string over an alphabet Δ (although in most cases $\Sigma = \Delta$). When the possible output values are limited to two values, such as 0/1 or yes/no, the preceding architecture is obtained as a special case of the input/output architecture.

- The preceding computational models (various kinds of automata and context-free grammars)

allowed us to define only a very restricted class of languages. We saw, for example, that even a simple language such as $\{a^n b^n c^n : n \in \mathbb{N}\}$ is not context-free (and therefore is not regular), yet it is very easy to write a short program in any modern programming language to generate it.

- ❶ A Turing machine allows us to define more complex languages and perform computations that are not possible with the preceding models, and it is, in fact, the most powerful computational model in the following sense: every possible algorithm or computational process can be converted into a computation on a Turing machine. The preceding assertion is called the Church-Turing thesis, and it will be discussed in detail later in the chapter. It is, in fact, a philosophical conjecture that has withstood the test of time to this day.

❶ **Definition 6.1:** A Turing machine $M = (Q, \Gamma, s, h, \delta)$ consists of the following five components:

1. **An infinite work tape** with infinitely many cells, denoted by

$$\text{Tape} = \{c_1, c_2, c_3, c_4, \dots, c_i, \dots\}$$

In most texts, it is represented by an infinite sequence of square cells arranged from left to right, as shown in Figure 6.1.

2. **A control unit** with a read/write head (Tape head) that can perform read and write operations on any cell of the tape (when positioned over it), and move to the neighboring cell on the right or left, depending on the current state q and the current symbol. When the control unit is positioned over cell c_1 , it can, of course, move only to the right.

3. **A set of states Q**

$$Q = \{s, q_1, q_2, \dots, q_n, h\}$$

s - **start state**

h - **halt state**

$q_1, q_2, \dots, q_n$ - **intermediate states** (or **internal states**)

4. **A set of symbols Γ** consisting of three types of symbols:

- ◆ **Input symbols**

An alphabet $\Sigma \subseteq \Gamma$ of symbols whose symbols are used to express strings in the input language.

- ◆ **Output symbols**

An alphabet $\Delta \subseteq \Gamma$ of symbols whose symbols are used to express strings in the output language. In most examples we encounter, $\Delta = \Sigma$.

- ◆ **Auxiliary symbols**

All the remaining symbols in $\Gamma - (\Sigma \cup \Delta)$ are **auxiliary symbols**, which the machine uses to make the computation more efficient. These symbols must be different from the input and output symbols and are supposed to be erased from the tape after the computation ends (“halts”). This set always contains at least one symbol, B , the **blank-cell symbol**, which denotes an empty cell on the tape.



5. A transition function δ

$$\delta : (Q - \{h\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

which describes how the machine should operate when it is in a given state $q \in Q - \{h\}$ and the machine head reads a given symbol $a \in \Gamma$ (in the current cell). The symbol $\rightarrow$ indicates that this is a partial function, and therefore it need not be defined for every element of $(Q - \{h\}) \times \Gamma$. The equality

$$\delta(q, a) = (q', b, D)$$

means the following: If the symbol written in the current cell is a , then erase a , write b , and move to the neighboring cell on the left (if $D = L$) or the neighboring cell on the right (if $D = R$). If $b = a$, then the only operation performed is a move to a neighboring cell, without changing the cell contents. If the machine head is over the first cell and the transition function instructs it to move left, then no movement is actually performed.

▣ Operation of a Turing Machine

1. At the first stage, the input w over the input alphabet Σ is placed at the beginning of the tape. If $|w| = n$, then the input occupies the first n cells of the tape, $c_1, c_2, \dots, c_n$ (from left to right). All the remaining cells on the tape contain the symbol B (blank), indicating that the cell is empty. The machine head begins over cell c_1 , and the machine starts in state s (Start).
2. The machine reads the current state q and the current symbol a , and checks whether the transition function δ is defined for the pair (q, a) .
3. If $\delta(q, a) = (q', b, D)$, and if $b \neq a$, the machine head erases the symbol a and writes the symbol b in its place. If $D = L$, the machine head moves to the neighboring cell on the left, and if $D = R$, it moves to the neighboring cell on the right. The machine moves to state q' .
4. If the function δ is not defined for the pair (q, a) , then the computation terminates without halting, and we say that the machine is **stalled**. We also say that the input supplied to the machine is **rejected**.
5. If $q' \neq h$, the machine returns to stage 2. If $q' = h$, the machine terminates the computation. We say that the machine M has reached the halt state (halt), and that the input w has been **accepted** by the machine. The machine's **output** is the string of symbols at the beginning of the tape, up to the first symbol B (not including it). This string is denoted by $M(w)$.

◆ **Example 6.1:** Consider the following Turing machine M , which has five states:

$$Q = \{s, q_1, q_2, q_3, h\}$$



The set of symbols is

$$\Gamma = \{0, 1, B\}$$

and the transition function is specified by the **transition table (transition diagram)** shown in Table 6.1.

δ	0	1	B
s	$q_2, 1, R$	$q_1, 0, L$	$q_3, 1, R$
q_1	$q_1, 1, R$	$h, 0, R$	q_3, B, L
q_2	$q_1, 1, R$	$q_3, 0, R$	$q_1, 0, L$
q_3	$q_3, 1, R$	$q_1, 1, L$	q_2, B, L

Table 6.1: Transition table for the Turing machine M in Example 6.1

Supply the machine M with the input $w = 1010010$. The length of the input w is 7, so it occupies the seven tape cells $c_1 — c_7$, as shown in the first configuration in Figure 6.3.

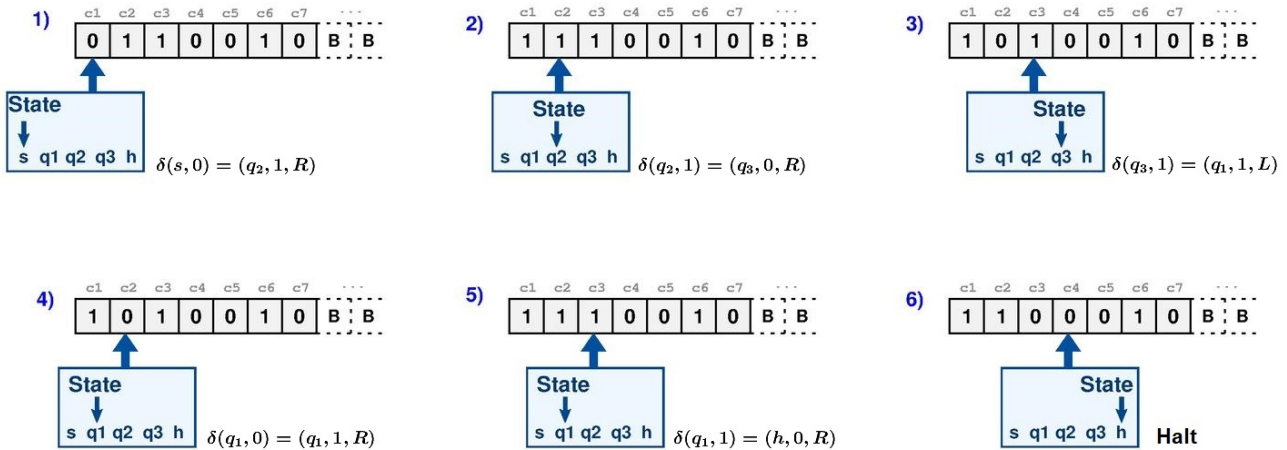


Fig. 6.3: Computation sequence for the input $w = 1010010$ on the Turing machine $M = (Q, \Gamma, s, h, \delta)$

Each snapshot of the machine in Figure 6.3 is called a **configuration**. It is not difficult to verify that the transition function deterministically specifies exactly six configurations of the machine before it halts in state h .

As with automata, a Turing machine can also be represented by a labeled graph, which can be described by a transition diagram. Figure 6.4 shows the transition diagram of the Turing machine presented in Example 6.1. Every edge $q \xrightarrow{a/b,D} r$ in the labeled graph represents what must be done when the machine is in state q and the machine head reads a :

1. Erase the symbol a ,
2. write the symbol b ,
3. move to the neighboring cell in direction D ($D = L$ or $D = R$),
4. and move the machine to state r .

Note that a complete and valid transition table does not guarantee that the machine halts for

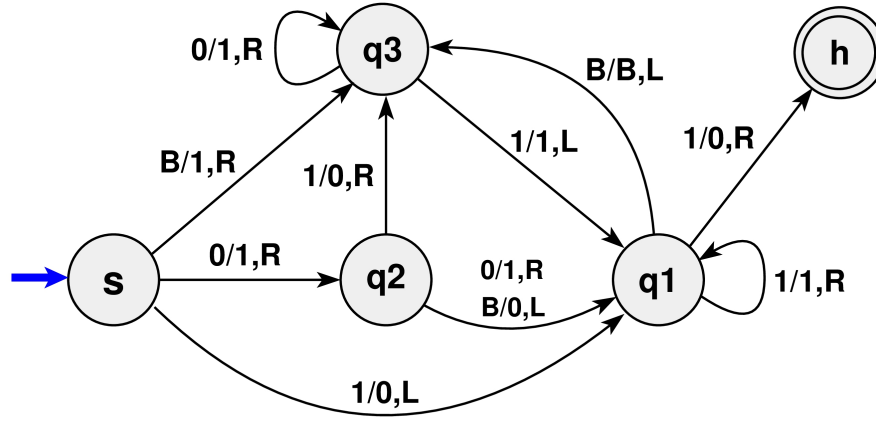


Fig. 6.4: Transition diagram for the Turing machine in Example 6.1

every possible input. It is easy to construct examples in which a Turing machine enters an infinite loop or stalls.

Definition 6.2: Let M be a given Turing machine. The collection of all strings w accepted by M is called **the language accepted by M** and is denoted by $L(M)$.

Definition 6.3: Let Σ be a set of input symbols, and let Δ be a set of output symbols. A partial function $f : \Sigma^* \rightarrow \Delta^*$ is called a **computable function**, or simply a **computational function**, if there exists a Turing machine M with the same input/output symbols such that, for every string $w \in \Sigma^*$, $f(w)$ is defined if and only if the machine M halts on input w , and moreover $f(w) = M(w)$.

Remarks

1. What we have actually defined is a **deterministic Turing machine**. A **nondeterministic Turing machine** is defined in the same way, except that δ is a relation rather than a function. That is,

$$\delta \subseteq ((Q - \{h\}) \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

2. Different books present different ways of defining a Turing machine. We have chosen a relatively simple and convenient definition in order to provide a basic understanding of the concept, but it is worth noting that other books contain definitions that differ in various details. Later, we present several extensions of the Turing machine, such as a special state for rejecting input (reject), the option to remain on a cell without moving right or left in certain states, the option to return to the beginning of the tape (RESET), and so on. In some books, a Turing machine may contain more than one tape. We discuss generalized Turing machines at the end of the chapter. Students interested in studying this subject more deeply are invited to consult the list of books at the beginning of the booklet and the extensive material available on the Internet.

3. Definition 6.3 is, in fact, a formulation of the Church-Turing thesis. This is another way of saying that every algorithm in the most general sense can be carried out by a suitable Turing machine. It is important to note that the concept of an algorithm is not entirely clear or universally agreed upon, since we cannot encompass every possible computational model.

6.2 Computation Sequences

- A simpler and more economical method for describing computational processes in a Turing machine is by means of **computation sequences of machine configurations**.

i **Definition 6.4:** A **configuration** of a Turing machine is a triple (q, w, i) representing a snapshot of the machine at a given instant:

- (a) the state q in which the machine is located,
 - (b) the string of symbols w at the beginning of the tape, in its nonblank cells,
 - (c) the number i of the cell over which the machine head is positioned.
- If the machine head is positioned over an empty cell, the string w includes that cell as well.

i **Definition 6.5:** *The derivation relation $\vdash$ between two configurations:*

Let (q, u, i) and (r, v, j) be two configurations of a given Turing machine. We say that the configuration (q, u, i) **yields** the configuration (r, v, j) , or, equivalently, that the configuration (r, v, j) is **derived** from the configuration (q, u, i) , and write

$$(q, u, i) \vdash (r, v, j)$$

if the configuration (r, v, j) is obtained from the configuration (q, u, i) by applying the transition function δ . That is,

- (a) $\delta(q, u[i]) = (r, v[i], D)$
- (b) if $D = L$, then $j = i - 1$ (the machine head moves left)
- (c) if $D = R$, then $j = i + 1$ (the machine head moves right)

- A **computation sequence** in a Turing machine is any sequence of configurations in which each configuration is derived from the preceding configuration in the sequence:

$$(p_1, w_1, i_1) \vdash (p_2, w_2, i_2) \vdash (p_3, w_3, i_3) \vdash \cdots \vdash (p_k, w_k, i_k)$$

where $p_1, p_2, \dots, p_k \in Q$, $w_1, w_2, \dots, w_k \in \Gamma^*$. For convenience, we omit the cell number from a configuration and instead place an arrow above the corresponding symbol to represent the machine head.

- For example, the computation sequence in Example 6.1 can be described concisely by the following

sequence:

$$\begin{aligned}
 (s, \overset{\downarrow}{0}110010) &\vdash (q_2, \overset{\downarrow}{1}\overset{\downarrow}{1}0010) \vdash (q_3, 1\overset{\downarrow}{0}\overset{\downarrow}{1}0010) \vdash (q_1, 1\overset{\downarrow}{0}10010) \\
 &\vdash (q_1, 1\overset{\downarrow}{1}\overset{\downarrow}{1}0010) \vdash (h, 11\overset{\downarrow}{0}\overset{\downarrow}{0}010)
 \end{aligned}$$

The symbols colored green are those changed by the machine head in the preceding step.

❖ **Example 6.2:** Let M be the Turing machine defined by the transition diagram in Figure 6.5. Write a transition table for M . Write a computation sequence for the input $w = 0010110$. Give a general description of the function computed by the machine on the input.

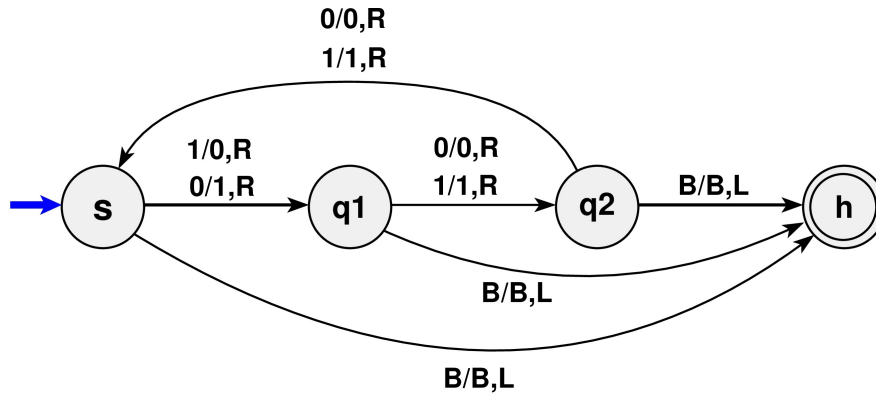


Fig. 6.5: Transition diagram for a Turing machine with two intermediate states $\{q_1, q_2\}$ over the alphabet $\Sigma = \{0, 1\}$

🎓 **Solution:** It is not difficult to construct a transition table by inspecting the diagram (Table 6.2).

δ	0	1	B
s	$q_1, 1, R$	$q_1, 0, R$	h, B, L
q_1	$q_2, 0, R$	$q_2, 1, R$	h, B, L
q_2	$s, 0, R$	$s, 1, R$	h, B, L

Table 6.2: Transition table for the Turing machine M in Example 6.2

Computation sequence for the input $w = 0010110$

$$\begin{aligned}
 (s, \overset{\downarrow}{0}010110) &\vdash (q_1, \overset{\downarrow}{1}\overset{\downarrow}{0}10110) \vdash (q_2, 1\overset{\downarrow}{0}\overset{\downarrow}{1}0110) \vdash (s, 101\overset{\downarrow}{0}110) \\
 &\vdash (q_1, 101\overset{\downarrow}{1}\overset{\downarrow}{1}10) \vdash (q_2, 10111\overset{\downarrow}{1}0) \vdash (s, 101111\overset{\downarrow}{0}) \\
 &\vdash (s, 1011111\overset{\downarrow}{B}) \vdash (h, 101111\overset{\downarrow}{1})
 \end{aligned}$$

The machine's operation is clear: every input symbol whose index is $3k + 1$ is inverted (0 becomes 1, and 1 becomes 0). All other symbols remain unchanged. The machine halts for every possible input.

6.3 Additional Examples

- ❖ **Example 6.3:** Let M be the Turing machine defined by the transition diagram in Figure 6.6. Write a transition table for M . Write a computation sequence for the input $w = 0010110$. Give a general description of the function computed by the machine on the input.

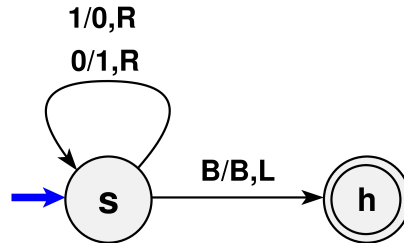


Fig. 6.6: “Inverter”: transition diagram for a Turing machine with no intermediate states over the alphabet $\Sigma = \{0, 1\}$

- 🔗 **Solution:** This machine has no intermediate states, so the transition table contains only one row for the start state s (see Table 6.3).

δ	0	1	B
s	$s, 1, R$	$s, 0, R$	h, B, L

Table 6.3: Transition table for the Turing machine M in Example 6.3

Computation sequence for the input $w = 0010110$

$$\begin{aligned}
 (s, \overset{\downarrow}{0}010110) &\vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{0}10110) \vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}10110) \vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{0}110) \\
 &\vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1}\overset{\downarrow}{1}0) \vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1}\overset{\downarrow}{0}) \vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{0}) \\
 &\vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1}\overset{\downarrow}{B}) \vdash (s, \overset{\downarrow}{1}\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1})
 \end{aligned}$$

Here too, the operation of the machine is completely clear: The machine changes every zero to a one and every one to a zero. The input $w = 0010110$ is transformed into the output $M(w) = 1101001$. Such a machine is sometimes called an inverter.

- ❖ **Example 6.4:** Let M be the Turing machine defined by the transition diagram in Figure 6.7. The input/output symbols are $\Sigma = \{a, b, c\}$. Write a transition table for M and prove that M accepts no input.

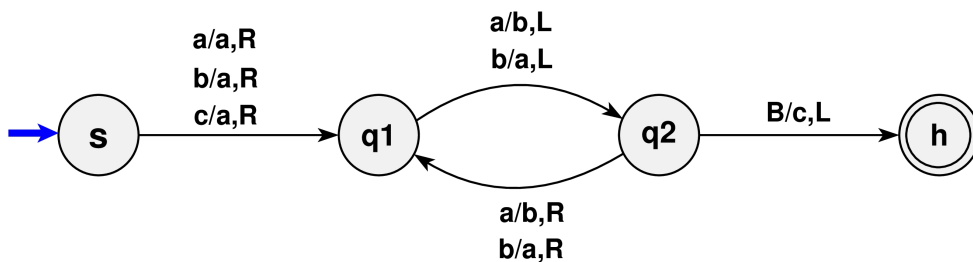


Fig. 6.7: A Turing machine over the input/output alphabet $\{a, b, c\}$ that does not halt on any input

Solution: Again, it is easy to write a transition table from the diagram. In this case, it is important to note that the transition function is partial, and the table contains four empty cells. For example, if the machine is in state q_1 and the machine head reads the symbol c , then the machine is stalled, and therefore the input is rejected.

δ	a	b	c	B
s	q_1, a, R	q_1, a, R	q_1, a, R	
q_1	q_2, b, L	q_2, a, L		
q_2	q_1, b, R	q_1, a, R		h, c, L

Table 6.4: Transition table for the Turing machine M in Example 6.4

- ◆ We attempt to understand how the machine operates on short words. Input of length 1: if $w = a$, the machine moves to state q_1 , and the machine head moves right and reads B . There is no outgoing transition from state q_1 for the symbol B , and therefore the machine stalls. Similarly, if $w = b$ or $w = c$, the machine moves right and stalls in state s (there is no outgoing transition for the symbol B in state s).
- ◆ Another way in which an input can be rejected is an *infinite loop*, which, in our machine, occurs for example on the input $w = bbb$:

$$\begin{aligned}
 (s, \downarrow bbb) &\vdash (q_1, \downarrow abb) \vdash (q_2, \downarrow ab) \vdash (q_1, \downarrow ab) \vdash (q_2, \downarrow bbb) \\
 &\vdash (q_1, \downarrow abb) \vdash (q_2, \downarrow ab) \vdash (q_1, \downarrow ab) \vdash (q_2, \downarrow bbb) \\
 &\vdash (q_1, \downarrow abb) \vdash (q_2, \downarrow ab) \vdash (q_1, \downarrow ab) \vdash (q_2, \downarrow bbb) \\
 &\vdash (q_1, \downarrow abb) \vdash (q_2, \downarrow ab) \vdash \dots
 \end{aligned}$$

A quick look at the first three rows reveals that the computation sequence

$$(q_1, \downarrow abb) \vdash (q_2, \downarrow ab) \vdash (q_1, \downarrow ab) \vdash (q_2, \downarrow bbb) \vdash (q_1, \downarrow abb)$$

which begins and ends in exactly the same configuration, repeats itself infinitely many times. Thus, in this case the machine never halts or stalls, but continues to run forever. Formally, this situation is also regarded as rejection of the input w .

- ◆ Even without these observations, however, we can give a complete argument that the machine does not halt on any input, as follows.
 - The assertion is clear for the empty input $w = \varepsilon$.
 - The following assertion is easily proved by induction on the input length: if the machine is in state s or q_1 , then the machine head is positioned over an ordinary symbol.
 - The machine can reach the halt state only if it is in state q_2 and the machine head is positioned

over a blank cell B . State q_2 can be reached only from state q_1 , which can move only to the left, onto an ordinary input cell.

Definition 6.6: A computation sequence

$$(p_1, w_1, i_1) \vdash (p_2, w_2, i_2) \vdash \cdots \vdash (p_k, w_k, i_k)$$

is called an *infinite loop* if its length is at least 2 and its first configuration is identical to its last configuration. That is: $k \geq 2$, $p_1 = p_k$, $w_1 = w_k$, $i_1 = i_k$.

It follows immediately from the definition that if two identical configurations occur in a computation sequence, then the sequence contains an infinite loop.

Exercise 6.1: Construct a Turing machine M that accepts the language of all binary strings containing at least two zeros and at least two ones.

Solution: The idea behind the construction is to choose suitable names for the intermediate states:

- q_0 a state in which one 0 has been read
- q_1 a state in which one 1 has been read
- q_{01} a state in which one 0 and one 1 have been read
- q_{00} a state in which at least two zeros have been read
- q_{11} a state in which at least two ones have been read
- q_{001} a state in which at least two zeros and one 1 have been read
- q_{011} a state in which at least two ones and one 0 have been read

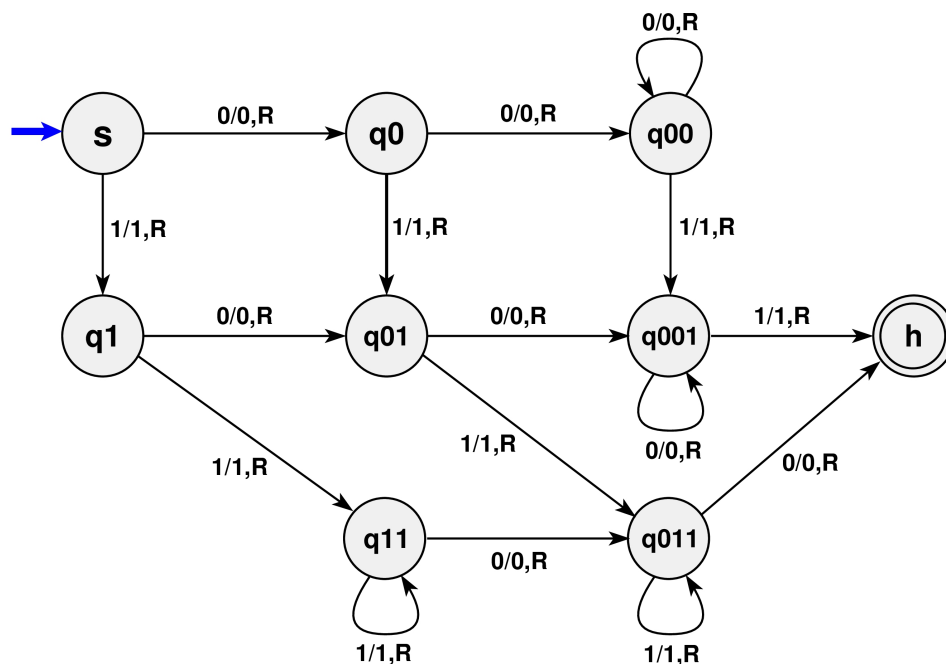


Fig. 6.8: A Turing machine that accepts all binary strings containing at least two 0 symbols and at least two 1 symbols

As can be seen from the transition diagram, we did not bother to handle transitions involving the symbol B . The only situation in which the machine encounters the symbol B is after a complete scan of an input that does not belong to the language, and therefore the machine stalls. Recall that if the machine stalls, the input has not been accepted. Thus, this time our transition function δ is a partial function, as can be seen from the transition table 6.4.

δ	0	1	B
s	$q_0, 0, R$	$q_1, 1, R$	
q_0	$q_{00}, 0, R$	$q_{01}, 1, R$	
q_1	$q_{01}, 0, R$	$q_{11}, 1, R$	
q_{00}	$q_{00}, 0, R$	$q_{001}, 1, R$	
q_{01}	$q_{001}, 0, R$	$q_{011}, 1, R$	
q_{001}	$q_{001}, 0, R$	$h, 1, R$	
q_{011}	$h, 0, R$	$q_{011}, 1, R$	

Table 6.5: Transition table for the Turing machine M in Exercise 6.1

As can be seen from the diagram or the table, no change is made to the input (a read-only operation), so the output of the machine M is identical to the input. The sole function of M is to accept or reject the input. Consider, for example, the computation sequences for $w = 0100100$ and $w = 111011$.

$$\begin{aligned}
 (s, \overset{\downarrow}{0}100100) &\vdash (q_0, \overset{\downarrow}{0}\overset{\downarrow}{1}00100) \vdash (q_{01}, 01\overset{\downarrow}{0}0100) \vdash (q_{001}, 010\overset{\downarrow}{0}100) \\
 &\vdash (q_{001}, 0100\overset{\downarrow}{1}00) \vdash (h, 01001\overset{\downarrow}{0}0)
 \end{aligned}$$

The input is accepted before its scan is complete (two symbols remain unscanned). The input $w = 111011$ is, of course, not accepted by the machine:

$$\begin{aligned}
 (s, \overset{\downarrow}{1}11011) &\vdash (q_1, \overset{\downarrow}{1}\overset{\downarrow}{1}1011) \vdash (q_{11}, 11\overset{\downarrow}{1}011) \vdash (q_{11}, 111\overset{\downarrow}{0}11) \\
 &\vdash (q_{011}, 1110\overset{\downarrow}{1}1) \vdash (q_{011}, 11101\overset{\downarrow}{1}) \vdash (q_{011}, 111011\overset{\downarrow}{B}) \rightarrow \text{stall}
 \end{aligned}$$

◈ **Exercise 6.2:** Construct a Turing machine M over the binary alphabet $\Sigma = \{0, 1\}$ such that, for every $w \in \Sigma^*$, $M(w) = w0w$.

🔴 **Solution:** This time, we must use the additional auxiliary symbols $\{a, b, Y\}$, and therefore the tape alphabet is

$$\Gamma = \{0, 1, a, b, Y\}$$

noitarepo fo nalP We use the auxiliary symbols a , b to temporarily represent the symbols 0 , 1 , and use the symbol Y to separate the two copies of the input w . At the first stage, we scan

the input and place the symbol Y at its end. At the second stage, we copy the input to the left of Y to the right of Y . For every symbol in $\{0, 1\}$ to the left of the separator Y , we temporarily replace it by an auxiliary symbol in $\{a, b\}$ and copy it to the right of Y in the original order (see Figure 6.9).

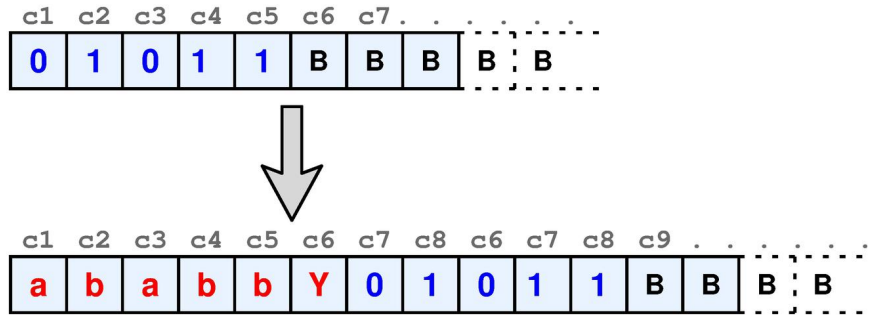


Fig. 6.9: The preparation stage in the computation of the Turing machine M that computes $w0w$

As in the preceding example, it is useful to choose suitable names for the intermediate states:

- s In the start state, replace the symbol 0 by a , move right, and enter state q_{y0} . Replace the symbol 1 by b , move right, and enter state q_{y1} . If the input is empty, write 0 and enter the halt state.
- q_{y0} In this state, move to the end of the input until the first blank cell is encountered, write Y , and enter state q_{d0} .
- q_{y1} In this state, move to the end of the input until the first blank cell is encountered, write Y , and enter state q_{d1} .
- q_{d0} This is the state from which the symbol 0 is duplicated (dup 0). Move right to the first blank cell, write 0 , and enter state q_2 .
- q_{d1} This is the state from which the symbol 1 is duplicated (dup 1). Move right to the first blank cell, write 1 , and enter state q_2 .
- q_2 In this state, the machine head is in the region containing the second copy (to the right of the separator Y), and its purpose is to return us to the other side, to the first cell to the left of the separator Y , and enter state q_c (cross).
- q_c This is the state entered after crossing the separator Y from right to left. The machine head is positioned over the cell to the left of the separator. If we encounter an auxiliary symbol a or b , then we have finished duplicating the entire input. Move right (back to the separator) and enter state q_e (end). If we encounter a symbol 0 or 1 , then enter state q_1 .
- q_1 In this state, the machine head is to the left of the separator and must find the first symbol in $\{0, 1\}$ that has not yet been duplicated. Move left until an auxiliary symbol in $\{a, b\}$ is encountered, then move right and enter state q_d (duplicate).
- q_d In this state, the machine must duplicate the symbol in $\{0, 1\}$ under the machine head. If the current symbol is 0 , the machine writes a , moves right, and enters state q_{d0} (dup 0).

If the current symbol is **1**, the machine writes **b**, moves right, and enters state q_{d1} (dup 1).

- q_e This state is reached only from state q_c when the machine head is positioned over the separator. Replace the separator **Y** by the symbol **0**, and replace all auxiliary symbols in $\{a, b\}$ by output symbols in $\{0, 1\}$. When the left end of the tape is reached, it is impossible to move left, so after replacing the auxiliary symbols, the machine knows that it has finished upon reading an output symbol and enters the halt state.

The transition diagram of the machine is shown in Figure 6.10, and the transition table is shown in Table 6.4.

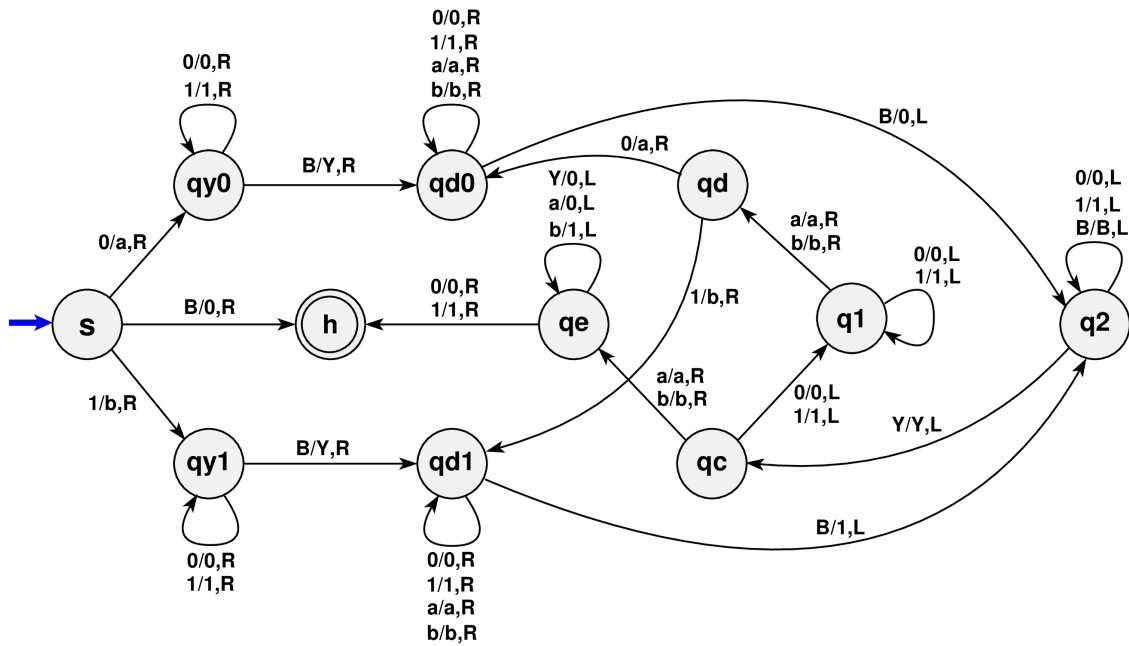


Fig. 6.10: A Turing machine that computes the function $f(w) = w0w$ over the binary alphabet $\{0, 1\}$

δ	0	1	B	a	b	Y
s	q_{y0}, a, R	q_{y1}, b, R	$h, 0, R$			
q_{y0}	$q_{y0}, 0, R$	$q_{y0}, 1, R$	q_{d0}, Y, R			
q_{y1}	$q_{y1}, 0, R$	$q_{y1}, 1, R$	q_{d1}, Y, R			
q_2	$q_2, 0, L$	$q_2, 1, L$	q_2, B, L			q_c, Y, L
q_1	$q_1, 0, L$	$q_1, 1, L$		q_d, a, R	q_d, b, R	
q_c	$q_1, 0, L$	$q_1, 1, L$		q_e, a, R	q_e, b, R	
q_d	q_{d0}, a, R	q_{d1}, b, R				
q_{d0}	$q_{d0}, 0, R$	$q_{d0}, 1, R$	$q_2, 0, L$	q_{d0}, a, R	q_{d0}, b, R	q_{d0}, Y, R
q_{d1}	$q_{d1}, 0, R$	$q_{d1}, 1, R$	$q_2, 1, L$	q_{d1}, a, R	q_{d1}, b, R	q_{d1}, Y, R
q_e	$h, 0, R$	$h, 1, R$	h, B, R	$q_e, 0, L$	q_e, b, L	$q_e, 0, L$

Table 6.6: Transition table for the Turing machine M in Exercise 6.2

❖ **Exercise 6.3:** Construct a Turing machine that computes the function

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*, \quad f(w) = ww$$

Hint: In the preceding exercise, we succeeded in constructing a Turing machine for the function $f(w) = w0w$. At one stage of the machine's operation, the machine head is positioned over the separator symbol Y . Perform a sequence of swaps between Y and the next symbol on the tape until the symbol Y is pushed to the end of the tape and erased. Then scan the entire tape backward and replace the auxiliary symbols $\{a, b\}$ by the output symbols $\{0, 1\}$. All that remains is to determine how to swap Yx into xY .

❖ **Exercise 6.4:** Construct a Turing machine over the alphabet $\{0\}$ that accepts a string $w = 0^n$ if and only if n is a power of 2 (that is, there exists k such that $n = 2^k$).

🍷 **Solution:** The familiar algorithm works here. We need the auxiliary symbols a , x .

1. Write a in the first cell and continue scanning the input from left to right, erasing every second zero and writing x in its place.
2. While returning to the beginning of the tape, check whether the tape contains a single zero or more than one. If the tape contains a single zero, the machine enters the halt state and terminates. Otherwise, proceed to stage 3.
3. Return to the beginning of the tape and check whether the number of zeros remaining on the tape is even or odd. If it is even, return to the beginning of the tape and repeat stage 1 (if it is odd, the input is not accepted by the machine).

We briefly summarize the roles of the machine's various states and leave the remaining details for the student to determine. For example, testing whether the number of zeros on the tape is even or odd is performed using two states, q_e , q_o , between which the machine moves whenever the machine head encounters the symbol 0 .

- s In the start state, mark the first cell of the tape with the auxiliary symbol a , move right, and enter state q_1 .
- q_1 In this state, the machine head moves right until it encounters the first symbol 0 , which must be skipped. The first time the machine encounters 0 , the machine head moves right, and the machine enters state q_2 .
- q_2 In this state, the machine searches for the second symbol 0 , which must be replaced by the symbol x . The first time 0 is encountered, replace it by x , move right, and return to state q_1 .
- q_c In this state, a check (c=check) is performed to determine whether the tape contains a single 0 or more than one. If 0 is encountered, enter state q_z , which checks whether there is another 0 .
- q_z This state is reached only from state q_c , in which one 0 has already been identified; in state q_z , the machine checks whether another 0 exists on the tape. If another 0 is encountered, the machine enters state q_r (returning to the beginning of the tape); otherwise, the machine



enters the halt state h and terminates.

- q_r In this state, the machine returns (r=return) to the beginning of the tape. The machine moves left until it reads the symbol a , which marks the beginning of the tape, and then moves right and enters state q_e to test the parity of the number of zeros on the tape.
- q_e This state indicates that an even number of zeros has been counted up to the current stage. The machine head moves right until it encounters a new symbol 0 , at which point the machine enters state q_o . The scan ends upon encountering a blank cell B , at which point the machine enters state q_s .
- q_o This state indicates that an odd number of zeros has been counted up to the current stage. The machine head moves right until it encounters a new symbol 0 , at which point the machine enters state q_e ; otherwise, the machine stalls over a blank cell B .
- q_s In this state, the machine returns to the beginning of the tape and restarts the algorithm in state s .

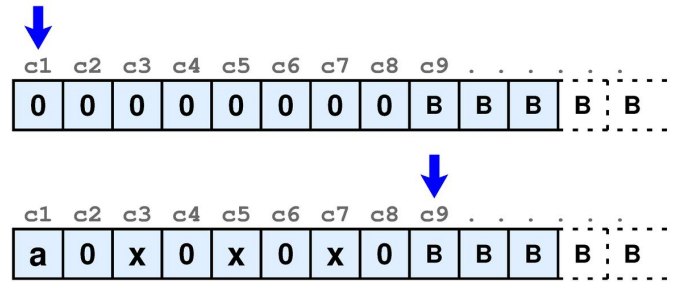


Fig. 6.11: Marking the first cell by a and erasing every second zero in the algorithm for M

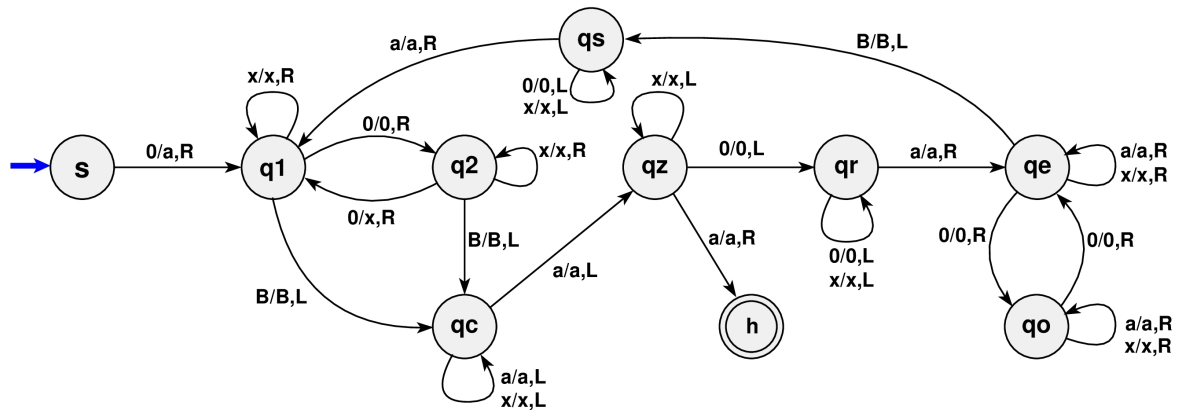


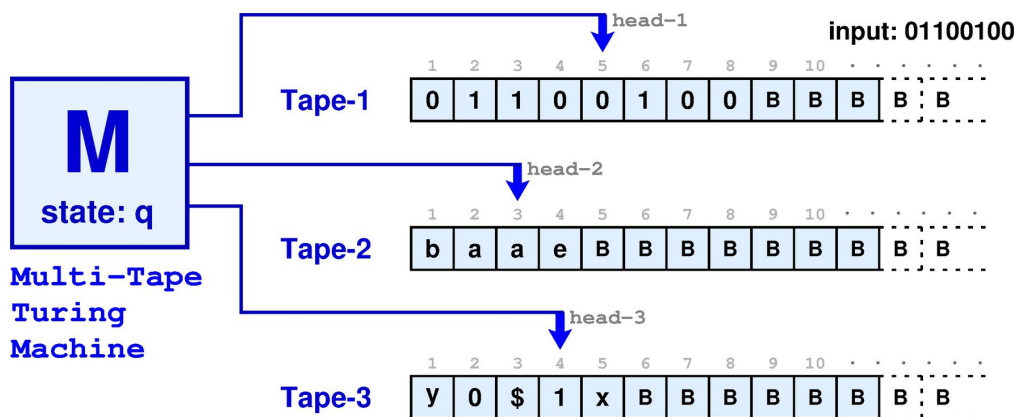
Fig. 6.12: Transition diagram for the Turing machine in Exercise 6.4: recognizing a string of the form 0^{2^k}

δ	0	B	a	x
s	q_1, a, R			
q_1	$q_2, 0, R$	q_c, B, L		q_1, x, R
q_2	q_1, x, R	q_c, B, L		q_2, x, R
q_c	$q_z, 0, L$		q_c, a, L	q_c, x, L
q_z	$q_r, 0, L$		h, a, R	q_z, x, L
q_r	$q_r, 0, L$		q_e, a, R	q_r, x, L
q_s	$q_s, 0, L$		q_1, a, R	q_s, x, L
q_e	$q_o, 0, R$	q_s, B, L	q_e, a, R	q_e, x, R
q_o	$q_e, 0, R$		q_o, a, R	q_o, x, R

Table 6.7: Transition table for the Turing machine M in Exercise 6.4

6.4 Multitape Turing Machines

- In the preceding chapter, we became acquainted with the basic model of a Turing machine, and throughout the section we experienced how difficult it is to work with a basic computational model whose operational capabilities are severely limited for simple operations such as returning the machine head to the beginning of the tape, shifting the input forward, or copying it to another region of the tape. Even a simple operation such as exchanging two neighboring symbols requires nontrivial logistics, which interfere with and distract from the algorithms we wish to implement. These difficulties led to the addition of various extensions to the basic machine, giving us greater convenience and agility in writing algorithms for it.

Fig. 6.13: A Turing machine with three tapes, $\Sigma = \Delta = \{0, 1\}$, $\Gamma = \{0, 1, a, b, c, d, e, x, y, \$\}$

- The most significant extension of the Turing machine is obtained by adding several tapes: tape-1, tape-2, ..., tape- k . Each tape tape- i has its own independent machine head head- i , which can move along the tape and read and write independently of the machine's other tapes.
- In the initial configuration, the machine's input is placed on the first tape (tape-1), all the other tapes are empty, and the heads of all the tapes point to the first cell.
- The transition function

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

is a partial function that receives a vector $(p, a_1, a_2, \dots, a_k)$ consisting of a state p and k symbols $a_1, a_2, \dots, a_k$, located under the heads of the k tapes tape-1, tape-2, ..., tape- k , respectively. The function δ returns a vector of length $2k + 1$:

$$\delta(p, a_1, a_2, \dots, a_k) = (q, b_1, b_2, \dots, b_k, D_1, D_2, \dots, D_k)$$

whose practical meaning is explained below.

- As in an ordinary Turing machine, a multitape machine has the following states:

s – start state

h – halt state

$q_1, q_2, \dots, q_n$ – intermediate states (or internal states)

In some cases, we also wish to include a **reject state r** , which allows us to obtain a rejection answer in finite time. Turing machines with a reject state allow us to define an important class of decidable languages, as we learn later.

Operation of a Multitape Turing Machine

1. At the initial stage, place at the beginning of the first tape (tape-1) the input w over the input alphabet Σ . All the remaining cells on the first tape contain the blank-cell symbol B (blank).
2. The first machine head (head-1) begins over cell c_1 .
3. The machine starts in the start state s (start).
4. All the other tapes must be empty, and their heads point to the first cell c_1 .
5. The machine reads the current state p and the current symbols $a_1, a_2, \dots, a_k$, located under the heads of the k tapes tape-1, tape-2, ..., tape- k , respectively, and checks whether the transition function δ is defined for the vector $(p, a_1, a_2, \dots, a_k)$.
6. If the transition function is defined for the current vector,

$$\delta(p, a_1, a_2, \dots, a_k) = (q, b_1, b_2, \dots, b_k, D_1, D_2, \dots, D_k)$$

then:

- a. For every i , $1 \leq i \leq k$, if $b_i \neq a_i$, the machine head head- i erases the symbol a_i and



writes the symbol b_i in its place.

- b. For every i , $1 \leq i \leq k$, each machine head head- i moves to the neighboring cell on the left if $D_i = L$, to the neighboring cell on the right if $D_i = R$, or remains in place if $D_i = S$ (Stay).
 - c. The machine moves to state q .
6. If the function δ is not defined for the vector $(p, a_1, a_2, \dots, a_k)$, then the computation terminates without halting, and we say that the machine is **stalled**. We also say that the input supplied to the machine is **rejected**.
 7. If $q \neq h$ and there is no reject state r , the machine returns to stage 5.
 8. If $q = h$, the machine terminates the computation. We say that the machine M has reached the halt state (halt), and that the input w has been **accepted** by the machine M . The machine's **output** is the string of output symbols at the beginning of the first tape (tape-1). This string is denoted by $M(w)$.
 9. If the machine M also includes a reject state r (reject) and if $q = r$, then the machine halts in this case as well, and we say that the input w has been **rejected**.
- The **tape contents** are defined as the sequence of symbols beginning in the first cell and ending in the last nonblank cell (B). For example, in the initial configuration, the contents of the first tape are the machine's input, and the contents of the other tapes are empty.

Example 1: Multiplication of Positive Integers

- **Problem description:** We must design a Turing machine M that multiplies two positive integers m , n . The questions before us are:
1. What are the input and output alphabets, and how do we use them to represent integers?
 2. How do we supply the machine M with an input consisting of two numbers?
 3. How many tapes does our machine have?
 4. Which auxiliary symbols do we need?
- Since this is our first example on the subject, we choose the simplest and most convenient design:
1. We use the binary alphabet $\Sigma = \{0, 1\}$, but represent a positive integer n by a string 1^n consisting of n copies of the digit 1. For example, the number 4 is represented by the string 1111.
 2. An input consisting of two positive integers m , n is represented by a string of the form $1^m 0 1^n$. The digit 0 serves as a separator between the two numbers.
 3. We use three tapes: tape-1, tape-2, tape-3. The input is placed on the first tape, and we use the other two tapes to perform the computations.
 4. We use the auxiliary symbol % to mark the beginning of the second and third tapes, and, of



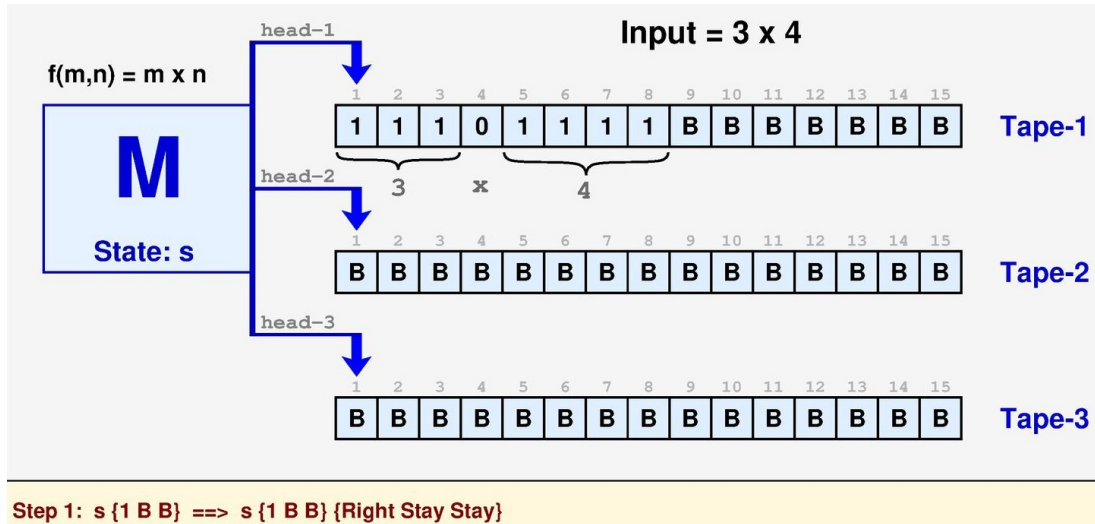


Fig. 6.14: A three-tape Turing machine for computing the function $m \times n$. The input for 3×4 is $w = 11101111$.

course, the auxiliary symbol B to denote a blank cell.

- Figure 6.14 shows the initial configuration of the machine M .
- We begin with a schematic description of the algorithm we wish to implement:
 1. At the first stage, check that the input is valid. The input must be of the form 1^*01^* . That is, it must be a string containing exactly one occurrence of 0 .
 2. At the second stage, copy the number n (that is, the string 1^n) to the second tape. While copying, erase the 01^n portion of the input, so that only the 1^m portion remains on the first tape. See Figure 6.15.
 3. Write the output on the machine's third tape. The output is obtained by the following loop: Copy the contents of tape-2 to the end of tape-3, and immediately afterward erase one symbol 1 from tape-1. Repeat stage 3. The loop terminates after the contents of tape-1 become empty.
 4. Transfer the contents of tape 3 to tape 1 (this is, of course, our output) and enter the halt state h . See Figure 6.16.
- We now present a transition table and a transition diagram for the machine M , which performs the required work precisely. For brevity and simplicity, we omit the input-validation stage (invalid input stalls the machine and is therefore not accepted). We use the following list of states:
 - s Start state. Write the symbol $\%$ at the beginning of tape-2 and tape-3, and enter state q_1 .
Head-1 moves to the separator symbol 0 , and the machine enters state q_1 .
 - q_1 In this state, the number n is transferred from tape-1 to tape-2.
After the transfer, only the number m remains on tape-1 (Figure 6.15).
The machine enters state $qr1$.
 - $qr1$ Return head-1 to the end of the number m .
At the same time, return head-2 to the beginning of tape-2.

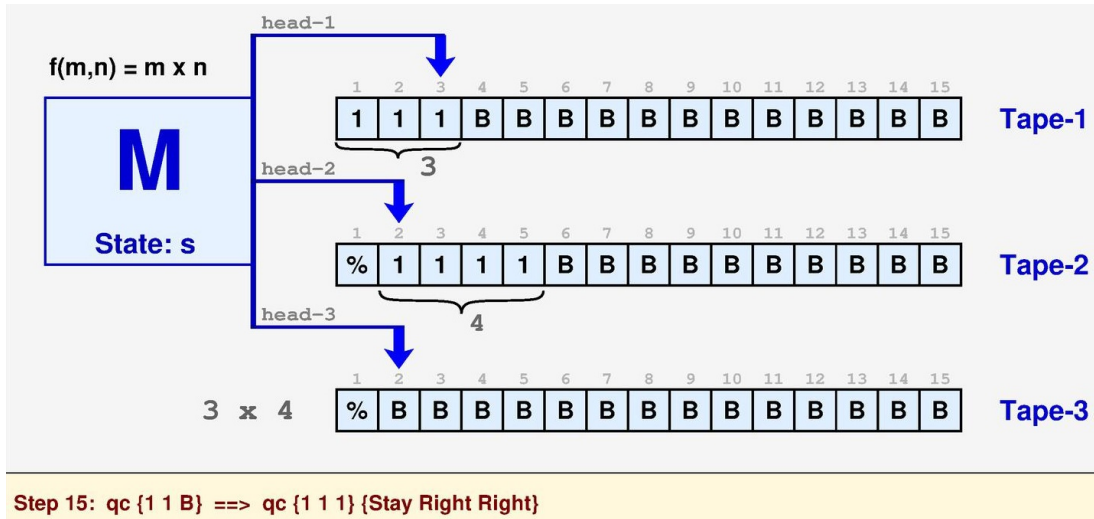


Fig. 6.15: A three-tape Turing machine for computing the function $m \times n$. The second stage of the algorithm.

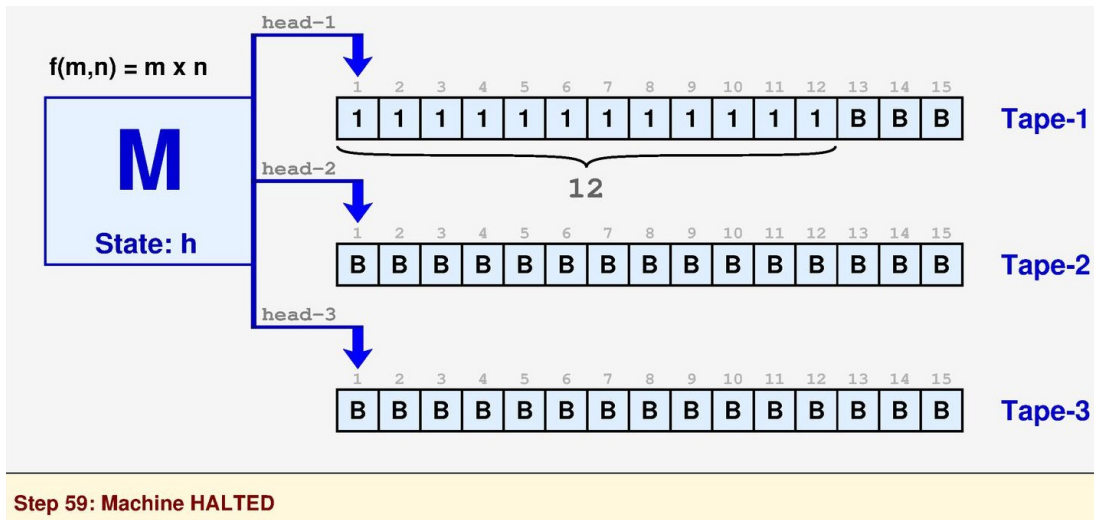


Fig. 6.16: A three-tape Turing machine for computing the function $m \times n$. The final stage of the algorithm.

qc Copy tape-2 (the number n) to the end of tape-3.

$qr2$ Return head-2 to the beginning of tape-2. Erase a symbol 1 from tape-1.

qf Transfer the product from tape-3 to tape-1.

When the transfer is complete, the machine enters the halt state h .

h Halt state

- In a Turing machine with m states, n symbols, and k tapes, the domain of the transition function δ may have size as large as $m \cdot n^k$. This may be a very large number of configurations, but in most cases encountered in the professional literature, the transition function is defined only on a small subset of configurations that can be collected in a state graph. We also note that an efficient and economical algorithm may prevent most combinations from arising (“idle configurations”), and therefore δ need not be defined for them.

- Figure (6.17) shows the state graph of our multiplication machine M . Every edge between two

states represents a table row such as

$$p \xrightarrow{1/0, R|0/B, L|-} q$$

which means that in state p ,

- (a) if head-1 reads the symbol 1 , it erases it, writes 0 , and moves right;
- (b) if head-2 reads 0 , it erases it, writes B , and moves left;
- (c) no operation is performed on the third tape.
- (d) Finally, the machine enters state q .

The symbol $|$ serves to separate the tapes. Thus, every edge in the graph actually represents the operation performed on all the machine's tapes. The symbol “ $-$ ” allows us to combine many different configurations into a single edge, thereby simplifying the description of the transition function.

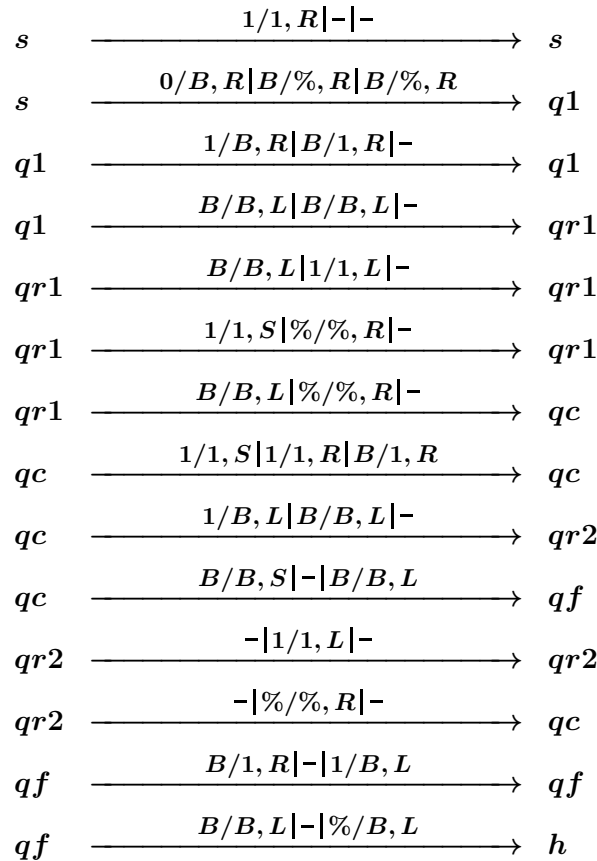


Fig. 6.17: Labeled state graph for a Turing machine M that computes the product of two numbers

- It is important to note that a definition of the transition function of this kind is valid only if there is no conflict between two different edges. No particular combination of head readings may be covered by two different edges and yield different results.
- A similar transition table for a multitape machine would have to be a multidimensional table, which is impractical. Therefore, for a multitape Turing machine, we must use a more compact kind of table based on the machine's state graph (see Figure (6.17)). In every table row, the

first column indicates the current state and the last column the new state. Each tape has a corresponding column specifying what must be done. The expression $a/b, D$ indicates that the symbol a is to be erased, b written, and the head moved in direction D . The expression $-$ in the column for tape- i indicates that no operation is performed on tape i .

Table 6.8: Transition table for a Turing machine M that computes a product

p	tape-1	tape-2	tape-3	q
s	$1/1, R$	$-$	$-$	s
s	$0/B, R$	$B/\%, R$	$B/\%, R$	$q1$
$q1$	$1/B, R$	$B/1, R$	$-$	$q1$
$q1$	$B/B, L$	$B/B, L$	$-$	$qr1$
$qr1$	$B/B, L$	$1/1, L$	$-$	$qr1$
$qr1$	$1/1, S$	$\%/\%, R$	$-$	$qr1$
$qr1$	$B/B, L$	$\%/\%, R$	$-$	qc
qc	$1/1, S$	$1/1, R$	$B/1, R$	qc
qc	$1/B, L$	$B/B, L$	$-$	$qr2$
qc	$B/B, S$	$-$	$B/B, L$	qf
$qr2$	$-$	$1/1, L$	$-$	$qr2$
$qr2$	$-$	$\%/\%, R$	$-$	qc
qf	$B/1, R$	$-$	$1/B, L$	qf
qf	$B/B, L$	$-$	$\%/B, L$	h

For simplicity, the table does not handle invalid input. Invalid input causes the machine to stall or enter an infinite loop. We leave the completion of the task, with a little more effort, as an exercise for the student. Turing-machine simulators available on the Internet may be useful.

- A transition diagram for a multitape Turing machine contains a planar layout of the various states and all the edges between them. Each edge is labeled by an expression describing the transitions covered by that edge. The transition diagram of our machine, corresponding to the table, appears below.
- A graphical simulation of the Turing machine M can be viewed in the following video:

 [Multi-tape Turing Machine for unary integer multiplication](#)

- Programs for drawing labeled graphs from a specification in a simple language can be found on the Internet. Although the result is not always sufficiently attractive, it can provide a quick impression. One well-known program is the free **Graphviz** software from **IBM**, which can be downloaded from the following link:

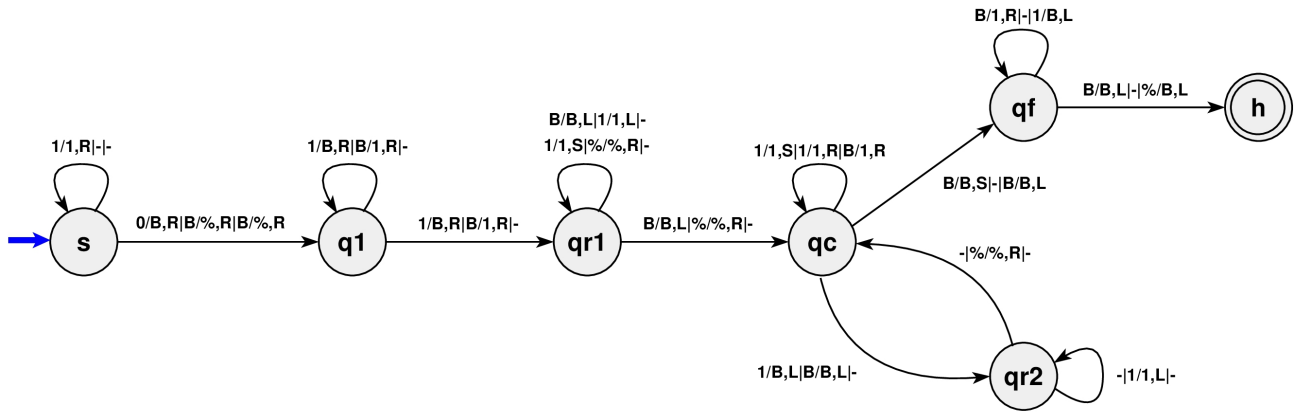


Fig. 6.18: Transition diagram for a three-tape Turing machine that computes the function $m \times n$



Graphviz - Graph Visualization Software

Example 2: Binary Addition

➤ **Problem description:** We must design a Turing machine M that adds two positive integers m , n given in binary form.

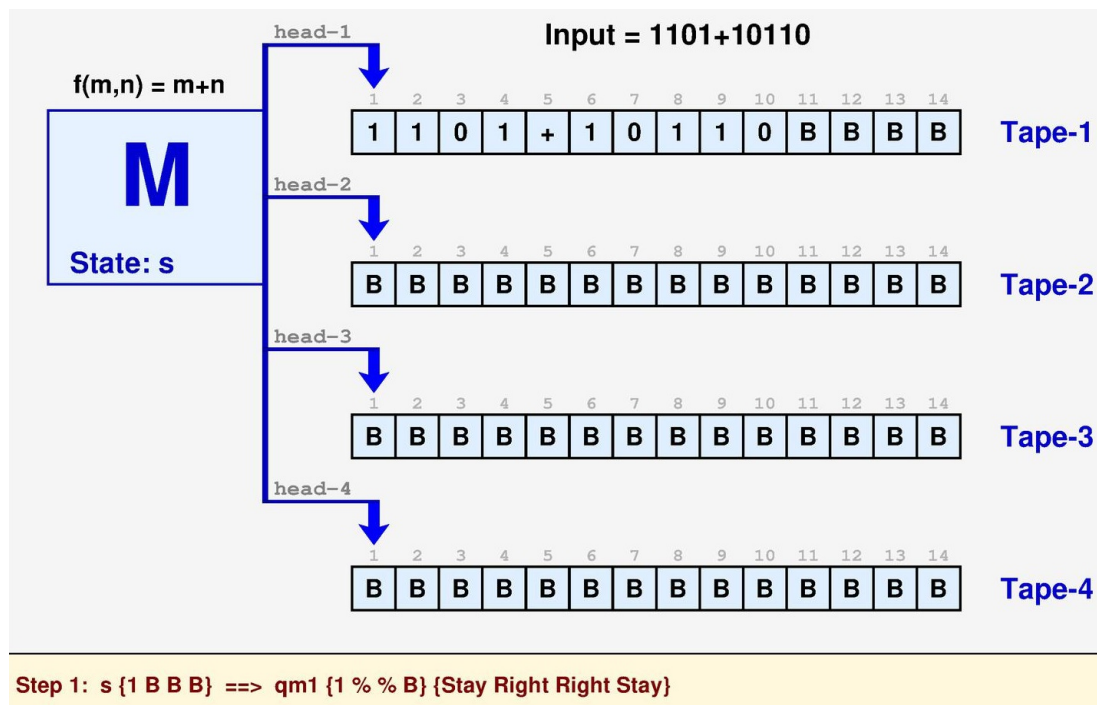


Fig. 6.19: A four-tape Turing machine for performing binary addition $m + n$, where the numbers m , n are represented in binary.

1. We use the binary alphabet $\Sigma = \{0, 1\}$ and represent a positive integer n by its ordinary binary representation. For example, the number 9 is represented by the string 1001.
2. An input consisting of two positive integers m , n is represented by a string of the form $\alpha + \beta$, where α is the binary representation of m and β is the binary representation of n . For example, to perform the addition $6 + 5$, we supply the machine with the input 110+101.
3. We use four tapes: tape-1, tape-2, tape-3, tape-4. The input is placed on the first tape, and



the other three tapes are used to perform computations.

4. We use the auxiliary symbol $+$ for the addition operation, the symbol $\%$ to mark the beginning of the second and third tapes, and, of course, the auxiliary symbol B to denote a blank cell.
- Figure 6.19 shows the initial configuration of the machine M .
- We begin with a schematic description of the algorithm we wish to implement. For simplicity, we omit the input-validation stage and assume that the input is valid.
1. Transfer the number m to tape-2 and the number n to tape-3. At the end of this stage, tape-1 is empty except for its first cell, which stores the carry and initially contains 0 .
 2. Move head-1 to the beginning of the tape, and head-2 and head-3 to the end of the input.
 3. Add the pair of digits under head-2 and head-3 together with the carry value in the first cell of tape-1. Write the result on tape-4, update the carry if necessary, move head-2 and head-3 left, and move head-4 right. Repeat this stage as long as at least one of the symbols under head-2 or head-3 is a binary digit.
 4. Transfer the contents of tape-4 to tape-1 from end to beginning (that is, reverse them!), and then enter the halt state h .
- The following is a list of the states of the machine M , with a brief explanation of their roles:
- s** The machine writes the symbol $\%$ at the beginning of tape-2 and tape-3 and enters state **qm1**.
 - qm1** The machine transfers the first number from tape-1 to tape-2.
Every scanned cell is marked by $\%$.
When head-1 reaches the symbol $+$, the machine enters state **qm2**.
 - qm2** The machine transfers the second number from tape-1 to tape-3.
Every scanned cell is marked by $\%$.
When head-1 reaches the symbol B , it moves left, and the machine enters state **qa**.
 - qa** The machine adds the two bits under head-2 and head-3, using the carry value under the head of tape-1. The result is written on tape-4 (in reverse order). After the addition operation, the tape heads move to the next position.
The operation repeats until tape-2 and tape-3 both read the symbol $\%$, at which point the machine enters state **qf**.
 - qf** The machine transfers the result (which is written in reverse order) from tape-4 to tape-1, and enters the halt state.
 - h** Halt state
- The complete list of transition rules is very long, and therefore we present it more freely by means of the following list of transition rules:



```

1.      s -> qm1    1/1,S|B/%,R|B/%,R|-
2.      s -> qm1    0/0,S|B/%,R|B/%,R|-
3.      qm1 -> qm1    0/%,R|B/0,R|-|-
4.      qm1 -> qm1    1/%,R|B/1,R|-|-
5.      qm1 -> qm2    +/%,R|B/B,L|-|-
6.      qm2 -> qm2    0/%,R|-|B/0,R|-
7.      qm2 -> qm2    1/%,R|-|B/1,R|-
8.      qm2 -> qr1    B/B,L|-|B/B,L|-
9.      qr1 -> qr1    %/B,L|-|-|-
10.     qr1 -> qa     B/0,S|-|-|-

```

```

-----
11.     qa -> qa     0/0,S|0/B,L|0/B,L|B/0,R
12.     qa -> qa     0/0,S|0/B,L|1/B,L|B/1,R
13.     qa -> qa     0/0,S|1/B,L|0/B,L|B/1,R
14.     qa -> qa     0/1,S|1/B,L|1/B,L|B/0,R
15.     qa -> qa     1/0,S|0/B,L|0/B,L|B/1,R
16.     qa -> qa     1/1,S|0/B,L|1/B,L|B/0,R
17.     qa -> qa     1/1,S|1/B,L|0/B,L|B/0,R
18.     qa -> qa     1/1,S|1/B,L|1/B,L|B/1,R
19.     qa -> qa     0/0,S|0/B,L|%/%,S|B/0,R
20.     qa -> qa     0/0,S|1/B,L|%/%,L|B/1,R
21.     qa -> qa     0/0,S|%/%,L|0/B,L|B/0,R
22.     qa -> qa     0/0,S|%/%,L|1/B,L|B/1,R
23.     qa -> qa     1/0,S|0/B,L|%/%,S|B/1,R
24.     qa -> qa     1/1,S|1/B,L|%/%,L|B/0,R
25.     qa -> qa     1/0,S|%/%,L|0/B,L|B/1,R
26.     qa -> qa     1/1,S|%/%,L|1/B,L|B/0,R

```

```

-----
17.     qa -> qf     1/B,S|%/B,L|%/B,L|B/1,S
18.     qa -> qf     0/B,S|%/B,L|%/B,L|B/B,L
19.     qf -> qf     B/1,R|-|-|1/B,L
20.     qf -> qf     B/0,R|-|-|0/B,L
21.     qf -> h       B/B,L|-|-|B/B,S

```

- The correctness of the code was tested using suitable software that simulates a multitape Turing machine. To avoid complicating matters, we did not check whether the input is valid. As stated above, invalid input stalls the machine and is therefore not accepted.



- We divided the code into three parts:
 - (a) The first part transfers the two numbers to tape-2 and tape-3;
 - (b) The middle part, which is the largest, performs the binary-addition loop;
 - (c) The third part transfers the input from tape-4 to tape-1 and performs cleanup.
- We obtained the transition diagram using the Graphviz software mentioned above. The result was far from satisfactory, but it was very helpful in drawing a more aesthetically pleasing diagram (see Figure 6.20). Note that most of the computation is performed in a loop from state **qa** back to itself (16 transitions!). This is, of course, an addition table for two bits with a carry.

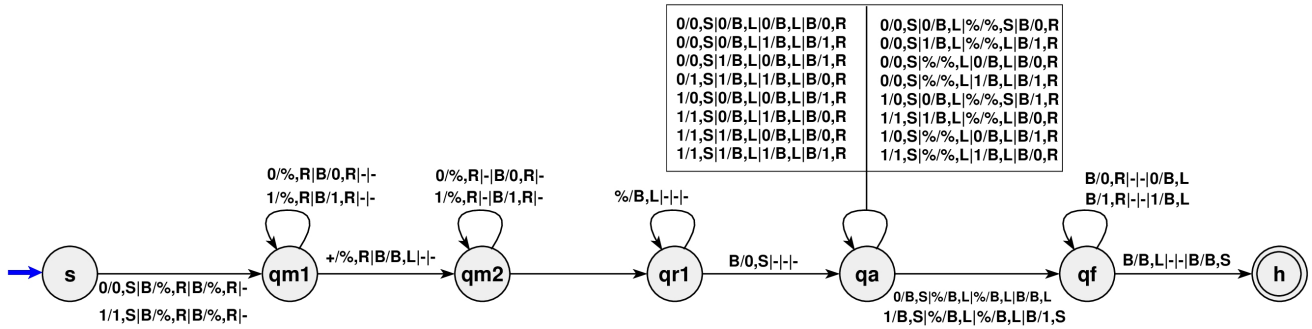


Fig. 6.20: Transition diagram for a four-tape Turing machine that performs binary addition $m + n$.

- The following table may explain the role of state **qa** more clearly. As stated above, the numbers m , n that we wish to add are stored (in binary form) on tape-2 and tape-3. At every stage, head-2 reads one bit of m and head-3 reads one bit of n , and the machine adds the bits by storing the result on tape-4 and the carry on tape-1.

Tape-1 (carry)	Tape-2 (m)	Tape-3 (n)	Tape-4 (m+n)
0/0,S	0/B,L	0/B,L	B/0,R
0/0,S	0/B,L	1/B,L	B/1,R
0/0,S	1/B,L	0/B,L	B/1,R
0/1,S	1/B,L	1/B,L	B/0,R
1/0,S	0/B,L	0/B,L	B/1,R
1/1,S	0/B,L	1/B,L	B/0,R
1/1,S	1/B,L	0/B,L	B/0,R
1/1,S	1/B,L	1/B,L	B/1,R
0/0,S	0/B,L	%/B,L	B/0,R
0/0,S	1/B,L	%/B,L	B/1,R
0/0,S	%/B,L	0/B,L	B/0,R
0/0,S	%/B,L	1/B,L	B/1,R
1/0,S	0/B,L	%/B,L	B/1,R
1/1,S	1/B,L	%/B,L	B/0,R
1/0,S	%/B,L	0/B,L	B/1,R
1/1,S	%/B,L	1/B,L	B/0,R

- A graphical simulation of the Turing machine M can be viewed in the following video:

6.5 Equivalence of Multitape and Ordinary Turing Machines

- At first glance, it appears that a multitape Turing machine should make it possible to define more languages and solve more difficult problems than a single-tape Turing machine, but in fact the two are equivalent computational models. Recall that two computational models are called **equivalent** if they accept the same languages. We prove below that every multitape Turing machine can be simulated by an ordinary single-tape Turing machine.
- Nevertheless, there are clear advantages to using a multitape Turing machine rather than an ordinary machine. Multiple tapes provide greater freedom of operation, convenience, and flexibility than a single tape.
- In the following theorem, we explain how a multitape Turing machine can be simulated by a simple single-tape Turing machine.

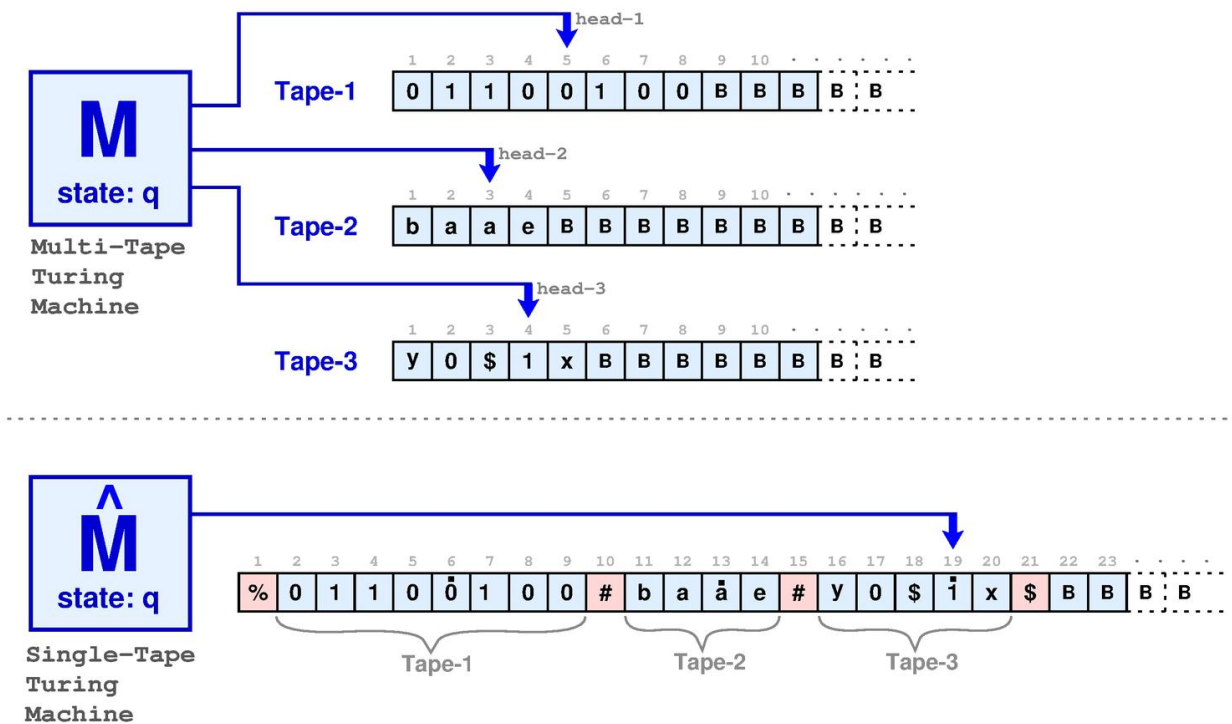


Fig. 6.21: Simulation of a three-tape Turing machine M by a simple Turing machine $\hat{M}$

Theorem 6.1: For every multitape Turing machine, there exists an equivalent ordinary Turing machine.

- ✓ **Proof:** We present the idea of the proof and leave the details as a series of exercises for the student appearing throughout this chapter. Let M be a Turing machine with k tapes, whose alphabet is Γ , whose input alphabet is Σ , and whose set of states is Q . We construct a single-tape Turing

machine $\hat{M}$ that simulates the machine M as follows:

- a. The alphabet $\Gamma_{\hat{M}}$ of the machine $\hat{M}$ contains all the symbols of Γ and, in addition, contains three separator symbols: %, \$, #. We use the symbol % to mark the first cell of the tape, the symbol \$ to mark the end of the word, and the symbol # to separate the segments that simulate the different tapes of the machine M , as shown in Figure 6.21.
- b. In addition to the separator symbols, we must also mark the positions of the k heads. We do this by placing a dot over the corresponding symbol. For this purpose, an additional auxiliary symbol $\dot{a}$ must be added for every symbol $a \in \Gamma$. This, of course, doubles the number of symbols in the alphabet of the machine $\hat{M}$. In summary, the alphabet $\hat{\Gamma}$ of the machine $\hat{M}$ is

$$\hat{\Gamma} = \Gamma \cup \{ \dot{a} : a \in \Gamma \} \cup \{ \#, \%, \$ \}$$

- c. The first procedure that $\hat{M}$ must perform is to add the symbol % to mark the beginning of the input, and to use a state q_{reset} that returns the machine head to the beginning of the input. See Exercise 6.5.
 - d. The second procedure that $\hat{M}$ must perform is to add k separator symbols # to the end of the input. See Exercise 6.6.
 - e. Every dotted symbol is, in effect, a virtual tape head. Each segment contains one dotted symbol that marks the position of the virtual head of the corresponding tape.
- **Dotting an Alphabet:** Adding a dot above a symbol in an alphabet is called a **dotting** operation. When this operation is applied to every symbol in the alphabet, we effectively double the number of symbols. For every symbol a in the machine's alphabet, a new symbol $\dot{a}$ is added. This technique, however, allows us to enrich the machine with more complex operations that are very difficult to achieve using its original alphabet.
- Every transition in the machine M

$$\delta(p, a_1, a_2, \dots, a_k) = (q, b_1, b_2, \dots, b_k, D_1, D_2, \dots, D_k)$$

is translated into a sequence of k operations on the k segments in the machine $\hat{M}$. For every $1 \leq i \leq k$, the machine head moves to the dotted symbol $\dot{a}_i$ in segment i , replaces it by the symbol b_i (without a dot), moves in direction D_i , and dots the new symbol. Clearly, between state p and state q , many intermediate states may be required to perform these k complex operations. Implementing such a program is not simple, but it is recommended for the diligent and curious student.

➤ **Exercise 6.5:** Let M be a single-tape Turing machine. Our goal is to augment M with an additional auxiliary symbol % that marks the beginning of the tape at the start of every computation. The new symbol % allows us to perform a **RESET** operation (returning the machine head to the beginning of the input) relatively easily at any stage of the computation. Write a procedure by which the symbol

% can be added at the beginning of the computation. At the end of the procedure, the machine head must be positioned over the first symbol of the input. Explain which new states and auxiliary symbols must be added to the machine and how the procedure works.

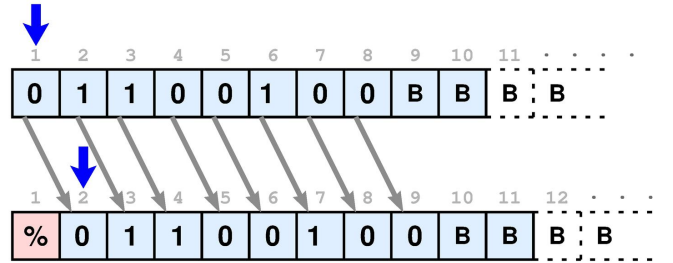


Fig. 6.22: Adding a new auxiliary symbol % to mark the beginning of the tape and shifting the input to the right

Hint: A special state p_x must be added for every symbol $x \in \Gamma$ (to remember the preceding symbol). A state q_{reset} that returns the machine head to the beginning of the input must be added.

1. In the start state s , the machine reads the first input symbol a_1 , writes $\#$ in its place, moves right, and enters state p_{a_1} .
2. If, in state p_x , the machine reads a symbol $y \in \Gamma$, $y \neq B$, then it writes x in its place, moves right, and enters state p_y . If, in state p_x , the machine reads a symbol B , then it writes x in its place, moves left, and enters state q_{reset} .
3. If, in state q_{reset} , the machine reads a symbol $x \in \Gamma$, $x \neq \%$, it moves left and remains in state q_{reset} .
4. If, in state q_{reset} , the machine reads the symbol $\%$, it moves right and the procedure terminates (returning control to the machine M).

❖ **Exercise 6.6:** Let M be a single-tape Turing machine. In this exercise, we wish to design a procedure that divides the tape of M into k segments separated by the symbol $\#$, in order to simulate a multitape Turing machine. We mark the beginning of the input by the symbol $\%$ and the end of the input by the symbol $\$$. The first segment must contain the input, and the remaining empty tapes must contain the symbol $\dot{B}$. The machine alphabet must be extended to represent the positions of the machine heads over each tape, which indicates that the machine head is positioned and we wish to add to M a procedure that appends k copies of the symbol $\#$ to the end of the current word on the tape, and then returns to its original position. Explain which new symbols and states must be added to the machine M to implement such a procedure, and describe how it works.

Hint: As we saw above, to mark the positions of the heads we use the dotting technique; therefore, in addition to the symbols $\{\#, \$, \%\}$, we must add a symbol $\dot{a}$ for every $a \in \Gamma$. A state q_{end} is needed to move the machine head to the end of the input. We must add k new states $p_1, p_2, \dots, p_k$, whose purpose is to create the segments.

1. At the first stage of the procedure, move the machine head to the end of the input. This can be done using a special state q_{end} that moves the head right whenever the current symbol $a \in \Gamma$ differs from B . When the current symbol is B , the head remains in place and the machine enters state p_1 .
2. In state p_1 , the machine head must move to the end of the word, until the first blank cell, write $\#$, move right to the next blank cell, and enter state p_2 .
3. In state p_2 , write the symbol $\dot{B}$, move right, and enter state p_3 .
4. Continue in the same manner until state p_k is entered.
5. In state p_k , write the final symbol $\#$, return to the dotted symbol $\dot{a}$, and remove the dot.

❗ **Theorem 6.2:** Every context-free language can be accepted by a Turing machine.

✔ **Proof:** Let L be a context-free language. By Theorem 5.5, there exists a pushdown automaton P

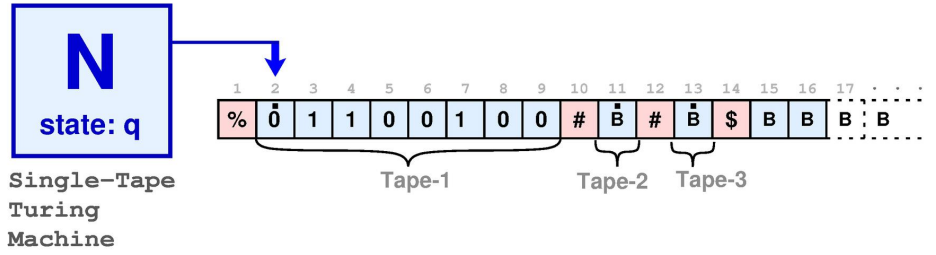


Fig. 6.23: Adding segments to simulate additional tapes

such that $L = L(P)$. By Theorem 5.2, we may assume that the automaton P is an elementary pushdown automaton. We simulate the automaton P by a two-tape Turing machine, according to the following idea.

- The first tape contains the input w , and the second tape simulates the stack of the automaton P (the top of the stack is on the right side of the tape). The transition function of M simulates the automaton's operation.
- The state set Q of M is identical to the state set of P , with the addition of a new start state s . The first operation of M is to write the start state q_0 (of the pushdown automaton) at the beginning of tape-2.
- Let $q \in Q$ be the current state, and let $a \in \Sigma$ be the current input symbol (under the head of tape-1). The transition function of the pushdown automaton may instruct it to read the symbol a , in which case tape-1 of the Turing machine moves right, or to read ϵ , in which case tape-1 remains in place.
- The second operation of the pushdown automaton is to push a stack symbol z_0 or pop the top of the stack. In the first case, head-2 moves right and writes z ; in the second case, it erases the current symbol on tape-2 and moves left.
- The pushdown automaton then moves to state q . The Turing machine also moves to state q .

Clearly, the input w is accepted by P if and only if it is accepted by M . We leave the student to fill in the missing details. ■

- The converse, of course, is not true: if L is a language accepted by a Turing machine, then L is not necessarily context-free. In Example 4.26, we saw that the language

$$L = \{a^n b^n c^n : n \in \mathbb{N}\}$$

is not context-free, but it is very easy to construct a Turing machine that accepts it.

- Thus, the class of languages accepted by Turing machines is larger than the class of context-free languages (which is itself larger than the class of regular languages). In the next section, we present the Church-Turing thesis, which asserts that the Turing machine is, in fact, the most powerful computational model available to us.

6.6 The Church-Turing Thesis

- In 1900, the renowned German mathematician David Hilbert presented to the International Congress of Mathematicians in Paris a list of 23 unsolved mathematical problems, which he posed as the most important challenges for mathematicians of the 20th century. To date, only ten of the 23 problems have been fully and conclusively solved. Seven problems have received partial and disputed solutions. Six problems remain unsolved to this day.

 **WIKIPEDIA** [Wikipedia: Hilbert's problems](#)

- Hilbert's tenth problem concerned the concept of an algorithm, or a computational protocol (the concept of an algorithm had not yet acquired a clear meaning in the mathematics of the late 19th century). The tenth challenge that Hilbert posed to mathematicians of the 20th century was to find an algorithm that receives as input a multivariate polynomial with integer coefficients and determines, by a finite sequence of clear computational steps, whether it has an integer root. For example, the polynomial

$$x^5 - 5x^2y^3 + y^4 - 13 = 0$$

has the integer root $(x, y) = (2, 1)$. This can be found by a sequence of guesses. Hilbert's demand was for a general algorithm that would work for all polynomials of this kind (for most of which, according to algebraic theorems, we have no solution methods at all).

- Hilbert's formulation suggests that he assumed such an algorithm existed but had not yet been discovered. In 1970, the Russian mathematician Yuri Matiyasevich succeeded in giving a precise mathematical proof that no such algorithm exists! The idea that an algorithm for solving a mathematical problem might be impossible was new and could not have been understood at the end of the 19th century.
- A formal mathematical definition of the concept of an algorithm was first given in 1936 by the logician Alonzo Church and independently by Alan Turing. At the same time, many diverse models of computability were developed, and over time they were found to be mutually equivalent. Nevertheless, it is impossible to encompass all the possible ways in which a computation can be performed, and therefore, in a philosophical sense, the concept of an algorithm remains open. The Church-Turing thesis asserts that every algorithm in every computational model that exists or may be discovered in the future has an equivalent algorithm on a Turing machine. In other words, everything that can be computed by any means can be computed by a Turing machine.

❖ **Exercise 6.7:** Construct a Turing machine M_1 that accepts a representation of an undirected graph G and checks its syntactic validity.

* **Hint:** The machine must verify that the parentheses are balanced and that there are two parenthesized groups, one for the vertices and the other for the edges. In the first parenthesized group, the machine must check that the decimal numbers are valid (for example, a multidigit number cannot begin with zero). In the second parenthesized group, the machine must check that the edges are valid, and so on.

❖ **Exercise 6.8:** Construct a Turing machine M_2 that accepts a syntactically valid representation of an undirected graph G and checks the following:

- The vertex list contains no duplicates.
- Every vertex appearing in the edge list must also appear in the vertex list in the first parenthesized group.
- Every edge (i, j) in the second list must satisfy $i \leq j$ (which guarantees a unique representation for every edge).
- The edge list contains no duplicates.

If the representation of G passes all these checks, it is called a **semantically valid representation**.

➤ **Exercise 6.9:** After ensuring that our graph has a semantically valid representation, we can now present a program for a Turing machine M_3 that checks whether the graph is connected. The standard algorithm for testing connectivity uses a vertex-marking operation. We use two forms of marking: **dotting** and **underlining**. In the dotting operation, place a dot over the first digit of the vertex, and in the underlining operation, draw a line under the first digit of the vertex. For example, if the vertex number is **548**, dotting it gives **5̇48**, and underlining it gives **548**. The same vertex can be dotted and underlined simultaneously: **5̇48**. For this purpose, 30 new auxiliary symbols must be added to the tape alphabet:

$$\{\dot{0}, \dot{1}, \dot{2}, \dots, \dot{9}, \underline{0}, \underline{1}, \underline{2}, \dots, \underline{9}, \underline{\dot{0}}, \underline{\dot{1}}, \underline{\dot{2}}, \dots, \underline{\dot{9}}\}$$

The algorithm that the machine M_3 must implement is as follows:

- Dot the first vertex in the vertex list on the tape.
- Scan the vertex list on the tape, find the first undotted vertex v_1 , and underline it.
- Scan the vertex list on the tape, find the first dotted vertex v_2 , and underline it.
- Scan the edge list and check whether it contains the edge whose vertices are v_1, v_2 . If so, remove the underlining from both vertices, dot v_1 , and return to stage 2. If (v_1, v_2) is not an edge of the graph, remove the underlining from vertex v_2 , proceed to the next dotted vertex in the vertex list, and repeat the current stage. After all dotted vertices have been scanned, proceed to stage 5.
- At this stage, the machine scans the vertex list. If all the vertices are dotted, then the graph is connected, and the machine enters the halt state h . Otherwise, the graph is not connected,



and the machine enters the reject state r (reject).

Recursive and Decidable Languages

i Definition 6.7: A language L is called **recursive** if there exists a Turing machine M such that, for every word $w \in L$, the machine M reaches the halt state h , and for every $w \notin L$, the machine does not reach the halt state h or never halts (an infinite loop).

i Definition 6.8: A language L is called **decidable** if there exists a Turing machine M that includes a reject state r such that, for every word w , if $w \in L$, then the machine M halts in the halt state h , and if $w \notin L$, then M halts in the reject state r .

- If M is a Turing machine that accepts a recursive language L , then, for every $w \in L$, we are guaranteed that the machine M reaches the halt state h . But if $w \notin L$, there is no guarantee that the machine halts at all (it may enter an infinite loop), or that it halts in a state other than h . That is, if $w \in L$, we can determine this in finite time, but if $w \notin L$, we cannot always determine it in finite time. If L is a decidable language, then it is possible to determine in finite time both whether $w \in L$ and whether $w \notin L$.

6.8 Encoding a Computational Model

- In the preceding chapters of this booklet, we encountered several classes of languages computable by different computational models. Regular languages can be computed by the various kinds of automata, context-free languages can be accepted by pushdown automata, and decidable and recursive languages can be computed by Turing machines. We also saw that the class of regular languages is contained in the class of context-free languages, and that the class of context-free languages is contained in the class of decidable languages. The Church-Turing thesis states that every computational model ever discovered will yield only decidable languages, and therefore we cannot extend the class of decidable languages. The question arises: is there a language that is not decidable? And how can we prove that a given language is not decidable?
- In fact, one can prove that most languages are not decidable by means of cardinality arguments from set theory. Consider, for example, the binary alphabet $\Sigma = \{0, 1\}$. The collection of all possible words over this alphabet is the set $A = \{0, 1\}^*$. In set theory, it is proved that the cardinality (size) of the set A is $\aleph_0$, because it is a countable set. The set of languages over the binary alphabet is the collection of all subsets of A , namely $\mathcal{P}(A)$. Set theory proves that $|\mathcal{P}(A)| = 2^{\aleph_0}$.
- We prove below that the set of all possible Turing machines over the alphabet $\{0, 1\}$ is countable. That is, it has cardinality $\aleph_0$. It follows that most languages in $\mathcal{P}(A)$ are not decidable!
- Alternatively, we can actually construct an example of a natural language that is not decidable. Later in this section, we construct several examples of concrete languages over the binary alphabet



and prove that they are not decidable. For this purpose, we define the normalization and encoding of a Turing machine (or any computational model encountered thus far).

- For simplicity, we consider the model of a single-tape Turing machine

$$M = (Q, \Gamma, s, h, \delta)$$

but this makes no real difference. All the ideas presented below apply to every computational model familiar to us.

- First, we must define an **encoding** of a Turing machine by means of a string over a given alphabet. The literature presents many encoding methods. We present two natural methods: the first uses a natural alphabet, whereas most standard methods use a binary alphabet.
- The first encoding method relies on the fact that every Turing machine

$$M = (Q, \Gamma, s, h, \delta)$$

can be **normalized** so that all the symbols in its definition are built from the following alphabet:

$$\Sigma_{\text{txt}} = \{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ a\ b\ c\ q\ s\ h\ B\ L\ R\ (\)\ ,\}$$

For simplicity, assume that the state names are always of the form $q_1, q_2, q_3, \dots$, the start state is s , and the halt state is h . The input symbols are of the form $a_1, a_2, a_3, \dots$, the output symbols are of the form $b_1, b_2, b_3, \dots$, and the auxiliary symbols are of the form $c_1, c_2, c_3, \dots$. We also use the symbol B to denote a blank cell on the machine's tape. We emphasize that the alphabet Σ_{txt} also includes a pair of parentheses and the comma symbol, which serves as a separator.

- ◆ **Example 6.5:** Consider the following Turing machine M , which has five states, $Q = \{s, q_1, q_2, q_3, h\}$, a set of symbols $\Gamma = \{0, 1, B\}$, and a transition function specified by the **transition table (transition diagram)** shown in Table 6.9.

δ	0	1	B
s	$q_2, 1, R$	$q_1, 0, L$	$q_3, 1, R$
q_1	$q_1, 1, R$	$h, 0, R$	q_3, B, L
q_2	$q_1, 1, R$	$q_3, 0, R$	$q_1, 0, L$
q_3	$q_3, 1, R$	$q_1, 1, L$	q_2, B, L

Table 6.9: Transition table for the Turing machine M in Example 6.5

To normalize M , replace the symbols $0, 1$ by the symbols a_1, a_2 . The state names q_1, q_2, q_3 already have the normal form and need not be replaced. The same is true of the symbols B, L, R . Thus, after normalization, the transition table has the form shown in Table 6.10.



δ	$a1$	$a2$	B
s	$q2, a2, R$	$q1, a1, L$	$q3, a2, R$
$q1$	$q1, a2, R$	$h, a1, R$	$q3, B, L$
$q2$	$q1, a2, R$	$q3, a1, R$	$q1, a1, L$
$q3$	$q3, a2, R$	$q1, a2, L$	$q2, B, L$

Table 6.10: Transition table for the Turing machine M in Example 6.5

In the string representing the machine M , the transition function δ is encoded by the following string of symbols:

$$\begin{aligned} \langle \delta \rangle = & ((s, a1, q2, a2, R), (s, a2, q1, a1, L), (s, B, q3, a2, R), \\ & (q1, a1, q1, a2, R), (q1, a2, h, a1, R), (q1, B, q3, B, L), \\ & (q2, a1, q1, a2, R), (q2, a2, q3, a1, R), (q2, B, q1, a1, L), \\ & (q3, a1, q3, a2, R), (q3, a2, q1, a2, L), (q3, B, q2, B, L)) \end{aligned}$$

The expression $\langle x \rangle$ denotes *the encoding of x* . The encoding of the entire machine M has the following form:

$$\begin{aligned} \langle M \rangle = & (q1, q2, q3, s, h), (a1, a2, B), ((s, a1, q2, a2, R), (s, a2, q1, a1, L), \\ & (s, B, q3, a2, R), (q1, a1, q1, a2, R), (q1, a2, h, a1, R), (q1, B, q3, B, L), \\ & (q2, a1, q1, a2, R), (q2, a2, q3, a1, R), (q2, B, q1, a1, L), \\ & (q3, a1, q3, a2, R), (q3, a2, q1, a2, L), (q3, B, q2, B, L)) \end{aligned}$$

The encoding of M consists of the following three parts:

- (a) list of states: $(q1, q2, q3, s, h)$,
- (b) tape alphabet (input/output/auxiliary symbols): $(a1, a2, B)$,
- (c) transition function.

- As stated above, most encoding methods in textbooks are binary and less intuitive. We use our intuitive encoding to convert it into a binary encoding method that is widely used in industry in similar contexts. For this purpose, we represent every character in the alphabet Σ_{txt} by five bits, as described in the conversion table 6.11. Our alphabet has 22 characters, so five bits suffice to represent all of them (32 distinct characters can be encoded by 5 bits).
- Using Table 6.11, we can encode every string w over the alphabet Σ_{txt} by a binary string obtained by replacing every character a with its corresponding five bits in Table 6.11. For example, the binary representation of the state $q3$ is 0110100011 (ten bits: $q \mapsto 01101$, $3 \mapsto 00011$). The textual encoding of the machine M contains 208 characters from the

Symbol	Binary	Symbol	Binary
0	00000	<i>a</i>	01010
1	00001	<i>b</i>	01011
2	00010	<i>c</i>	01100
3	00011	<i>q</i>	01101
4	00100	<i>s</i>	01110
5	00101	<i>h</i>	01111
6	00110	<i>B</i>	10000
7	00111	<i>L</i>	10001
8	01000	<i>R</i>	10010
9	01001	(	10011
,	10101	)	10100

Table 6.11: Representation of characters by binary strings of length 5

alphabet Σ_{txt} , and therefore its binary encoding contains 1040 bits ($5 \times 208 = 1040$).

$\langle M \rangle =$ 10011011010000110101011010001010101101000111010101110101010111101001010110011
01010000011010101010000101010110000101001010110011100110111010101010100000110101
01101000101010101010000101010110010101001010110011011101010101010000101010101101
00001101010101000001101011000110100101011001101110101011000010101011010001110101
01010000101010110010101001010110011011010000110101010100000110101011010000110101
010100001010101100101010010101100110110100001101010101000010101011111010101010
00001101011001010100101011001101101000011010110000101010101000011101011000010101
10001101001010110011011010001010101010100000110101011010000110101010100001010101
10010101001010110011011010001010101010100001010101011010001110101010100000110101
100101010010101100110110100010101100001010101101000011010101010000011010110001
10100101011001101101000111010101010000011010101101000111010101010000101010110010
10100101011001101101000111010101010000101010101101000011010101010000101010110001
10100101011001101101000111010101010000101010101101000011010101010000101010110001
101001010110011011010001110101100001010101011000010101011000010101100011010010100

- Figure 6.26 shows the binary encoding divided into groups of five bits, alongside the textual encoding: each group of five bits in the binary encoding corresponds to one character in the text on the right.

Binary Encoding															Text	
10011	01101	00001	10101	01101	00010	10101	01101	00011	10101	01110	10101	01111	10100	10101	10011	(<i>q</i> ₁ , <i>q</i> ₂ , <i>q</i> ₃ , <i>s</i> , <i>h</i>), (
01010	00001	10101	01010	00010	10101	10000	10100	10101	10011	10011	01110	10101	01010	00001	10101	<i>a</i> ₁ , <i>a</i> ₂ , <i>B</i>), ((<i>s</i> , <i>a</i> ₁ ,
01101	00010	10101	01010	00010	10101	10010	10100	10101	10011	01110	10101	01010	00010	10101	01101	<i>q</i> ₂ , <i>a</i> ₂ , <i>R</i>), (<i>s</i> , <i>a</i> ₂ , <i>q</i>
00001	10101	01010	00001	10101	10001	10100	10101	10011	01110	10101	10000	10101	01101	00011	10101	<i>l</i> , <i>a</i> ₁ , <i>L</i>), (<i>s</i> , <i>B</i> , <i>q</i> ₃ ,
01010	00010	10101	10010	10100	10101	10011	01101	00001	10101	01010	00001	10101	01101	00001	10101	<i>a</i> ₂ , <i>R</i>), (<i>q</i> ₁ , <i>a</i> ₁ , <i>q</i> ₁ ,
01010	00010	10101	10010	10100	10101	10011	01101	00001	10101	01010	00010	10101	01111	10101	01010	<i>a</i> ₂ , <i>R</i>), (<i>q</i> ₁ , <i>a</i> ₂ , <i>h</i> , <i>a</i>
00001	10101	10010	10100	10101	10011	01101	00001	10101	10000	10101	01101	00011	10101	10000	10101	<i>l</i> , <i>R</i>), (<i>q</i> ₁ , <i>B</i> , <i>q</i> ₃ , <i>B</i> ,
10001	10100	10101	10011	01101	00010	10101	01010	00001	10101	01101	00001	10101	01010	00010	10101	<i>L</i>), (<i>q</i> ₂ , <i>a</i> ₁ , <i>q</i> ₁ , <i>a</i> ₂ ,
10010	10100	10101	10011	01101	00010	10101	01010	00010	10101	01101	00011	10101	01010	00001	10101	<i>R</i>), (<i>q</i> ₂ , <i>a</i> ₂ , <i>q</i> ₃ , <i>a</i> ₁ ,
10010	10100	10101	10011	01101	00010	10101	10000	10101	01101	00001	10101	01010	00001	10101	10001	<i>R</i>), (<i>q</i> ₂ , <i>B</i> , <i>q</i> ₁ , <i>a</i> ₁ , <i>L</i>
10100	10101	10011	01101	00011	10101	01010	00001	10101	01101	00011	10101	01010	00010	10101	10010	), (<i>q</i> ₃ , <i>a</i> ₁ , <i>q</i> ₃ , <i>a</i> ₂ , <i>R</i>
10100	10101	10011	01101	00011	10101	01010	00010	10101	01101	00001	10101	01010	00010	10101	10001	), (<i>q</i> ₃ , <i>a</i> ₂ , <i>q</i> ₁ , <i>a</i> ₂ , <i>L</i>
10100	10101	10011	01101	00011	10101	10000	10101	01101	00010	10101	10000	10101	10001	10100	10100	), (<i>q</i> ₃ , <i>B</i> , <i>q</i> ₂ , <i>B</i> , <i>L</i>))

Fig. 6.26: Binary and textual encodings of the Turing machine M from Example 6.5

- There are many diverse methods for encoding a computational model by a binary string or by a string over a larger alphabet. We briefly present another idea for binary encoding: Use the digit 0 to represent states and symbols, and the digit 1 as punctuation and parentheses. States and symbols can be represented by strings of zeros of the form 0^n . For example, the state q_3 or the symbol a_3 is encoded by 000. A transition rule of the form $\delta(q_3, b) = (q_4, a, R)$ can be encoded by the string 0001001000010100. Note that the symbol a is encoded by 0, and the symbol b by 00. Similarly, the movement directions L , R are encoded by 0, 00. The

digit **1** serves as a separator (or punctuation mark). There is no danger of ambiguity because the meaning of 0^n is determined by its position in the string. The strings **11** and **111** can be used as punctuation marks or parentheses. For example, the encoding of a machine M may begin and end with the string **111**. We leave the development of this encoding idea into a precise formal definition as an exercise for the student.

- It is also clear that all the encodings presented above can be constructed similarly for the various kinds of automata, pushdown automata, and context-free grammars (after appropriately normalizing the syntactic variables and terminal symbols). If M is any computational model, we denote by $\langle M \rangle$ the encoding of M . We note that the particular encoding chosen does not matter, because the translation from one encoding to another can always be computed by a suitable Turing machine. In general, it is advisable to choose the encoding most convenient and appropriate for the context under discussion.
- Define the language

$$L_{\text{DFA}} = \{ \langle M \rangle : \text{notamotua etinif citsinimreted a si } M \}$$

Assuming that we use the binary encoding, L_{DFA} is a language over the binary alphabet $\Sigma = \{0, 1\}$, and therefore we can ask whether it is regular, context-free, or at least decidable. Similarly, we can define the language of all encodings of all possible Turing machines by

$$L_{\text{TM}} = \{ \langle M \rangle : \text{enihcam gniruT epat-elnis a si } M \}$$

and ask the same questions about the nature of the language L_{TM} .

i Proposition 6.3: The languages L_{DFA} , L_{TM} are regular.

- ✓ **Proof:** We briefly prove the assertion for the language L_{TM} . The proof for the language L_{DFA} is identical. To prove that the language L_{TM} is regular, it suffices to show that it is generated by a suitable regular expression. As stated above, an encoding $\langle M \rangle$ consists of three parts:

- (a) list of states: $(q_1, q_2, q_3, \dots, s, h)$,
- (b) tape alphabet (input/output/auxiliary symbols):

$$(a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots, B)$$

- (c) transition function: a list of quintuples of the form (p, a, q, b, D) , where p, q are states, a, b are symbols, and D is a direction symbol L or R .

We begin with a regular expression for a machine state:

$$\alpha = q_1(0 + 1 + 2 + \dots + 9)^*$$

Thus, the corresponding regular expression for part a is $((\alpha)^+ s, h)$.

A regular expression for an input symbol is $\alpha_1 = a1(0 + \dots + 9)^*$, and for an output symbol: An output symbol: $\alpha_2 = b1(0 + \dots + 9)^*$, an auxiliary symbol: $\alpha_3 = b1(0 + \dots + 9)^*$. Thus, a regular expression for part b is $((\alpha_1,)^+(\alpha_2,)^*(\alpha_3,)^*)$.

A regular expression for a general state in a transition rule is

$$\beta = s + h + q1(0 + 1 + 2 + \dots + 9)^*$$

A regular expression for a general symbol in a transition rule is

$$\gamma = (a + b + c)1(0 + 1 + 2 + \dots + 9)^*$$

Thus, a regular expression for a transition rule is

$$\rho = (\beta, \gamma, \beta, \gamma, L + R)$$

and therefore a regular expression for part c (a list of transition rules) is $(\rho,)^+$. ■

- Another immediate consequence of the encoding property is: the sets L_{DFA} , L_{TM} are countable languages.

Explanation: Recall that a set is called **countable** if its cardinality is equal to that of the natural numbers $\mathbb{N}$. A nontrivial example of a countable set is the set of rational numbers $\mathbb{Q}$. The set of real numbers is uncountable. The first impression one gets when considering the collection of all Turing machines is that it is larger than $\mathbb{N}$, because the description of a Turing machine may be complex and the number of combinations in such a description appears very large. But the mapping from a Turing machine M to its encoding $e = \langle M \rangle$, $e \in \{0, 1\}^*$, is one-to-one. The mapping from e to $1e$ is therefore one-to-one. We can regard the expression $1e$ as the binary representation of a natural number, and therefore obtain a one-to-one mapping from L_{TM} to the natural numbers. Thus, L_{TM} is countable.

6.9 The Halting Problem

- Now that we have defined the normalization and encoding of a computational model, we finally have the tools needed to define languages that are more complex and interesting than those encountered thus far.
- We use the term **binary Turing machine** to denote a Turing machine whose input and output alphabet is the binary alphabet $\{0, 1\}$ and that also has a reject state. For every binary word $w \in \{0, 1\}^*$, we use $\langle M, w \rangle$ to denote a binary or textual encoding (the choice is immaterial) of the machine M together with the input w . Define the language

$$L_{\text{TMH}} = \{ \langle M, w \rangle : w \text{ tupni no stlah taht enihcam gniruT yranib a si } M \}$$



We prove that the language L_{TMH} is not decidable, thereby providing a natural example of an easily defined undecidable language. Before that, we prove that L_{TMH} is recursive. That is, there exists a Turing machine that accepts L_{TMH} . This example therefore proves that the class of recursive languages contains the class of decidable languages but is not equal to it.

i Proposition 6.4: The language L_{TMH} is recursive.

✓ Proof: A completely formal proof is, of course, impractical, so we give one conceptual proof among the many that can be found in the extensive literature on the subject. For example, we can construct a multitape Turing machine U with the following properties:

- (a) The input to U is an encoding $\langle M \rangle$ of a binary Turing machine M on tape-1, and a binary string w that serves as input to M on tape-2.
- (b) The machine U must simulate the machine M on tape-3.
- (c) All the transition rules of M are stored by the machine U on tape-4.
- (d) It writes the current state of M on tape-5.
- (e) At the first stage, the machine U copies the input w from tape-2 to tape-3 and positions head-3 over the first symbol a of w on tape-3.
- (f) At the second stage, the machine U writes the start state s of the machine M on tape-5.
- (g) At the third stage, the machine U scans the transition rules stored on tape-4 and isolates the rule corresponding to state s and current symbol a . If no such rule exists, the machine U enters a reject state and terminates. If there is a corresponding rule $s \xrightarrow{a/b,D} q$, it proceeds to the next stage.
- (h) The machine U erases the symbol a on tape-3, writes b in its place, moves head-3 in direction D , erases state s from tape-5, and writes state q in its place.
- (i) If q is an accepting state of M , the machine U enters its accepting state and terminates. Otherwise, the machine U returns to stage 7 of the algorithm.

This concise description of the proof idea will suffice. ■

i Proposition 6.5: The language L_{TMH} is not decidable.

✓ Proof: The proof is based on the diagonalization method of the mathematician Georg Cantor in set theory. Define the following language:

$$L_{TMD} = \{ \langle M \rangle : \langle M, \langle M \rangle \rangle \notin L_{TMH} \}$$

In words: L_{TMD} is the language of all encodings $\langle M \rangle$ of binary Turing machines M that do not accept the input $w = \langle M \rangle$. That is, the machine M does not accept the binary string $w = \langle M \rangle$ that encodes the machine itself. We prove by contradiction that the language L_{TMD} is not decidable.

Suppose, for contradiction, that L_{TMD} is decidable. Then there exists a binary Turing machine



D that decides L_{TMD} . That is, for every binary Turing machine M , if M does not accept the input $\langle M \rangle$, then D accepts $\langle M \rangle$, and if the machine M accepts the input $\langle M \rangle$, then D does not accept $\langle M \rangle$.

Since the machine D is itself a binary Turing machine, we ask whether D accepts $\langle D \rangle$. There are, of course, only two possibilities:

- a. The machine D accepts $\langle D \rangle$. Then, by the definition of D , D does not accept $\langle D \rangle$!
 - b. The machine D does not accept $\langle D \rangle$. Then, by the definition of D , D does accept $\langle D \rangle$!
- In both cases, we obtain a logical contradiction! Therefore, the assumption that the language L_{TMD} is decidable is false! ■

6.10 Not the End ...

The present chapter requires several further additions, which will be included in the near future if time permits.

Exercises for chapter 6

1. What is the language L accepted by the Turing machine shown in Figure 6.27? Try to give the simplest possible description of the language L .

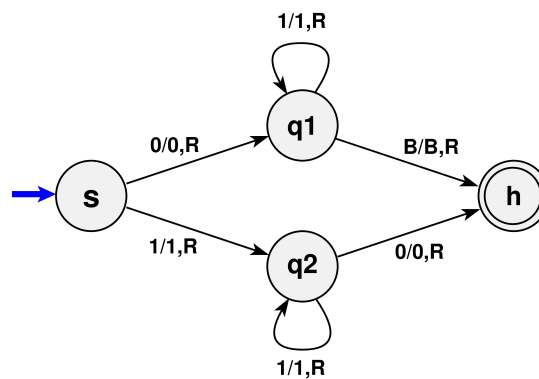


Fig. 6.27: Transition diagram for a Turing machine over the alphabet $\Sigma = \{0, 1\}$

2. Let M be a Turing machine whose input/output alphabet is $\{0, 1\}$, whose state set is $Q = \{s, q_1, q_2, h\}$, whose sole auxiliary symbol is B , and whose transition rules are given in Table 6.12.
 - (a) Find the computation sequences of M for the input $w = 0000$.
 - (b) Find the computation sequences of M for the input $w = 0100$.
 - (c) Give a description of the language $L(M)$.

δ	0	1	B
s			q_1, B, L
q_1	$q_1, 0, L$	$q_2, 1, R$	h, B, R
q_2	$h, 0, L$		

Table 6.12: Transition table for the Turing machine M in Exercise 4

3. Construct a Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that accepts the language

$$L = \{0^n 1^{2n} : n = 1, 2, 3, \dots\}$$

Write a transition table and draw a transition diagram for the machine M . Find a computation sequence for the input $w = 001111$.

4. Let M be a Turing machine whose transition table is given in Table 4.

δ	0	1	B
s			q_1, B, L
q_1	$q_1, 0, L$	$q_1, 1, L$	q_4, B, R
q_2	$q_3, 0, R$	$q_2, 1, R$	q_5, B, L
q_3	$q_3, 0, R$	$q_2, 0, L$	q_5, B, L
q_4	$q_3, 0, R$	$q_2, 1, R$	q_5, B, L
q_5	$h, 0, R$	h, B, R	

Table 6.13: Transition table for the Turing machine M in Exercise 4

- (a) Write a computation sequence for the input $w = 111010$.
 (b) Write a computation sequence for the input $w = 000110$.
 (c) Draw a transition diagram for the machine M .
 (d) What function does the machine M compute?
5. Let M be a Turing machine whose transition table is given in Table 5. What is the language $L(M)$?
6. Design a Turing machine M that divides an even-length string w into two equal halves. The input alphabet is $\{0, 1\}$. Use an auxiliary symbol $\#$ to separate the two halves. For example, if $w = 00101110$, then $M(w) = 0010\#1110$. If the input length w is not even, the machine does not halt.



δ	0	1	B
s			q_1, B, L
q_1	$q_1, 0, L$	$q_1, 1, L$	q_4, B, R
q_2	$q_3, 0, R$	$q_2, 1, R$	q_5, B, L
q_3	$q_3, 0, R$	$q_2, 0, L$	q_5, B, L
q_4	$q_3, 0, R$	$q_2, 1, R$	q_5, B, L
q_5	q_5, a, L		h, B, R

Table 6.14: Transition table for the Turing machine M in Exercise 5

7. Design a Turing machine M over the unary alphabet $\{0\}$ that computes the function

$$f(0^n) = \begin{cases} 0^k, & n = 2k \\ 0^{k+1}, & n = 2k + 1 \end{cases}$$

8. Design a Turing machine M over the unary alphabet $\{0\}$ that computes the function

$$f(0^n) = 0^{(n\%5)}$$

where $n\%5$ denotes the remainder when n is divided by 5 using integer division.

9. Construct a Turing machine M over the binary alphabet $\Sigma = \{0, 1\}$ such that, for every $w \in \Sigma^*$, $M(w) = ww$.

Hint: First study Exercise 6.2 carefully and continue from there. The only difficulty is shifting the second copy of w to the left.

10. Construct a Turing machine M over the alphabet $\Sigma = \{0, 1, \#\}$ that accepts the language

$$L = \{w\#w : w \in \{0, 1\}^*\}$$

11. Construct a Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that accepts the language

$$L = \{ww^R : w \in \{0, 1\}^*\}$$

where w^R denotes the reversal of the word w . Write a transition table and draw a suitable transition diagram.

12. Construct a Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that computes the partial function

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$$



defined by

$$f(w) = \begin{cases} u, & w = uu^R \\ \text{denifednu} & w \neq uu^R \end{cases}$$

Write a transition table and draw a suitable transition diagram.

Hint: Use the solution of the preceding exercise. Make sure that if the input is not of the form uu^R , then the machine does not halt (it either stalls or enters an infinite loop).

- 13.** Construct a Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that computes the partial function

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$$

defined by

$$f(1^n 0 1^m) = 1^{n-m}, \quad n > m \geq 0$$

For every word w that is not of the form $1^n 0 1^m$, the machine does not halt (that is, it either stalls or enters an infinite loop).

- 14.** Construct a Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that computes the partial function

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$$

defined by

$$f(1^n 0 1^m) = 1^{\text{div}(n,m)}, \quad n \geq 0, \quad m > 0$$

where the operator **div** denotes integer division. For example, $f(1111111011) = 111$. For every word w that is not of the form $1^n 0 1^m$, the machine does not halt (that is, it either stalls or enters an infinite loop).

- 15.** Construct a Turing machine M over the alphabet $\Sigma = \{a, b, c\}$ that accepts the language

$$L = \{a^n b^n c^n : n \in \mathbb{N}, n > 0\}$$

Hint: Use three auxiliary symbols $\{x, y, z\}$. Scan the input, repeatedly replacing three symbols a, b, c by the three symbols x, y, z .

- 16.** Construct a Turing machine M over the alphabet $\Sigma = \{a, b, c\}$ that accepts the language

$$L = \{a^i b^j c^k : i, j, k \in \mathbb{N}, k = i + j\}$$

Hint: At the first stage, check that the input has the form $a^* b^* c^*$. If so, return the machine head to the beginning of the tape, and then repeatedly erase a symbol a or b followed by a symbol c . Make sure that all the symbols in $\{a, b, c\}$ have been erased by the end of the process.

- 17.** Let L_1, L_2 be two languages over a common alphabet Σ that can be computed by a Turing machine. Prove that each of the following languages is computable by a Turing machine:



- (a) $L_1 \cup L_2$ (b) $L_1 \cap L_2$ (c) $L_1 L_2$
 (d) $L_1 - L_2$ (d) L_1^n (e) L_1^*

18. A **Turing machine with a reset operation RESET** is a Turing machine as defined in Definition 6.1, augmented with the ability to perform a reset operation in which the machine head can move immediately to the beginning of the tape and enter a desired state, but without the ability to move left. If $\delta(q, a) = (r, b, \text{RESET})$, then the machine head writes the symbol b in place of the symbol a , moves to the first cell of the tape, and the machine enters state r . Prove that every language or function computable by this machine is also computable by an ordinary Turing machine.

Hint: In the ordinary machine, the entire input can be shifted to the right, and an auxiliary symbol can be placed in the first cell to indicate that it is the first cell. The **RESET** operation can be performed by entering a special state that returns the machine head to the beginning of the tape. For every rule $\delta(q, a) = (r, b, \text{RESET})$, such a special state q_r must be added! Thus, simulating a Turing machine with a reset operation by an ordinary Turing machine may double the number of states!



19. Construct a Turing machine M over the alphabet $\Sigma = \{0, 1, +\}$ that computes the binary addition function $x + y = z$. The input to the machine has the form $x + y$, where $x, y \in \{0, 1\}^*$, and the machine's output must be the binary sum z of the two numbers. For example, $M(110 + 11) = 1001$.

20. Construct a four-tape Turing machine M over the alphabet $\Sigma = \{a, b, c\}$ that accepts the language

$$L = \{a^n b^n c^n : n \in \mathbb{N}, n > 0\}$$

Hint: The input w must be scanned three times. On the first pass, whenever head-1 reads a , head-2 writes a on tape-2 and advances. In the same manner, copy all occurrences of the symbol b to tape-2, and all occurrences of the symbol c to tape-3. At the next stage, return all the heads of tape-2, tape-3, and tape-4 to the beginning and enter the checking state q_{check} . In state q_{check} , the heads of tape-2, tape-3, and tape-4 move right together and read the three symbols beneath them. If none of the three is B , continue moving. If all three symbols are B , the input is accepted and the machine halts. If one is B and another is not B , the machine halts and the input is rejected.

21. Construct a four-tape Turing machine M over the alphabet $\Sigma = \{0, 1\}$ that accepts the language

$$L = \{uu^R : u \in \{0, 1\}^*\}$$

Hint: At the first stage, move head-1 to the end of tape-1. At the second stage, copy the input w from tape-1, from end to beginning. After stage 2 is complete, tape-2 contains the reversal of the input. At the third stage, compare the contents of tape-1 and tape-2: return both heads to the beginning and compare each pair of corresponding symbols. The only problem is that this procedure also accepts an input such as $w = 11011$, which does not satisfy the definition of L . To address this, use the third tape to check the parity of the input length. At the first stage, write 0 on tape-3, and on each move alternate it to 1 and then back to 0.

22. Return to Exercises 6, 7, 8, 9, 10, and solve them again using a multitape Turing machine. Make sure that the number of tapes chosen is appropriate for each problem.



23. Read carefully the definition of a pushdown automaton at the beginning of Chapter 5. Based on this definition, write a detailed formal definition of a **two-stack automaton**, or, more briefly, a



2-stack automaton. We note only that you may assume that the two stacks share a common alphabet, and that the transition function has the form

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma^* \times \Gamma^*)$$

- a. Prove that the computational model of a 2-stack automaton is more powerful than that of a one-stack automaton.

Hint: Prove that every Turing machine can be converted into an equivalent 2-stack automaton.

- b. Similarly, define a **3-stack automaton**.
- c. Is the computational model of a 3-stack automaton more powerful than that of a 2-stack automaton?



Chapter 7

Logic Circuits

- The purpose of this chapter is to present the computational model of the **logic circuit**, which is in fact the most common computational model in today's modern computing industry, which is based overwhelmingly on electronic chips (hardware implementations of logic circuits). In fact, each of the computational models encountered in the preceding chapters is implemented on top of a logic circuit by means of simulation programs. This, of course, does not diminish the importance of the preceding models. State machines, for example, are critical to VLSI circuit design processes, and in fact a substantial portion of the electronic components found in processors are implementations of state machines.
- We do not intend to cover the subject thoroughly, but rather to present several central concepts and ideas related to the theory of computational models. We assume that most students have encountered, or will encounter, the subject in greater depth in other courses (such as logic design, digital systems, and VLSI circuit design). The mathematical theory on which the field is based is, of course, Boolean algebra, which was developed in the middle of the preceding century.

7.1 Boolean Algebra – A Brief Review

- Boolean algebra was developed by the logician George Boole (1815-1864), who is regarded as the first herald of the modern information age. It is considered the first attempt to construct a mathematical model for the systematic treatment of the laws of thought, and it had a decisive influence on the development of modern logic and the logical foundations of all branches of mathematics.
- The main distinction between Boolean algebra and ordinary algebra is that variables in Boolean algebra take only two possible values: **0**, **1** (a common semantics: true or false). The algebraic operations are defined by means of **binary logical operations** such as: $\vee$, $\wedge$, $\oplus$, $\downarrow$, $\uparrow$, and the negation connective $\neg$. Since our space of combinations is small, we can define these



operations by means of **truth tables**

OR			AND			XOR			NOR			NAND			NOT	
x	y	$x \vee y$	x	y	$x \wedge y$	x	y	$x \oplus y$	x	y	$x \downarrow y$	x	y	$x \uparrow y$	x	$\neg x$
1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	0
1	0	1	1	0	0	1	0	1	1	0	0	1	0	1	0	1
0	1	1	0	1	0	0	1	1	0	1	0	0	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1

➤ Algebraic combinations can be used to define compound expressions such as

$$(1) \quad x_1 \wedge (x_2 \vee x_3)$$

$$(2) \quad (x_1 \oplus x_2) \uparrow (\neg x_3 \downarrow x_4)$$

$$(3) \quad (x_5 \vee \neg x_2 \vee x_3) \wedge \neg(x_3 \oplus \neg x_4)$$

Several additional logical connectives not appearing in the above list can be named, such as **XNOR**, but the purpose of the discussion is only to review the material (on the assumption that the student has already encountered it).

✦ **Exercise 7.1:** Study Exercise 32 at the end of Chapter 4, and then construct a context-free grammar for the language of Boolean expressions over a minimal alphabet. A simple proposed solution appears in Exercise 1 at the end of the chapter.

➤ Using the truth tables of the basic connectives, we can construct truth tables for more complex **Boolean expressions**. For example, the truth table of the expression $x_1 \wedge (x_2 \vee x_3)$ is shown in Table 7.1.

x_1	x_2	x_3	$x_1 \wedge (x_2 \vee x_3)$
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

Table 7.1: The truth table of the Boolean expression $x_1 \wedge (x_2 \vee x_3)$

- Each row¹ in a truth table is called an **assignment**. Formally, an assignment $\mathbf{a}$ is a function

$$\mathbf{a}: \{x_1, x_2, \dots, x_n\} \rightarrow \{0, 1\}$$

It is customary to write $\mathbf{a} = (a_1, a_2, \dots, a_n)$, where $a_i = \mathbf{a}(x_i)$. An assignment for which the expression has the value **1** is called a **satisfying assignment**. An assignment for which the expression has the value **0** is called a **falsifying assignment**. In our example there are three satisfying assignments and five falsifying assignments.

- We shall use the letters $\mathbf{x}, \mathbf{y}, \mathbf{z}$ for Boolean variables, and Greek letters ϕ, ψ, φ , to denote Boolean expressions. An expression such as

$$\phi(x_1, x_2, \dots, x_n)$$

will denote a Boolean expression in which all variables belong to the set $\{x_1, x_2, \dots, x_n\}$.

- If $(a_1, a_2, \dots, a_n)$ is any assignment of the variables $(x_1, x_2, \dots, x_n)$, then the expression

$$\phi(a_1, a_2, \dots, a_n)$$

will denote the Boolean value (**0** or **1**) obtained by the substitution

$$x_1 = a_1, \quad x_2 = a_2, \quad \dots, \quad x_n = a_n$$

in the expression ϕ . Substituting an assignment into a Boolean expression allows us to define a Boolean function, but we must begin with a more general definition.

i **Definition 7.1:** A **Boolean function** is a function of the form

$$f: \{0, 1\}^n \rightarrow \{0, 1\}^m$$

where $\{0, 1\}^n$ is the space of all binary sequences of length n

$$\{0, 1\}^n = \{ (a_1, a_2, \dots, a_n) : a_i = 0 \vee a_i = 1 \}$$

When $m = 1$, $f: \{0, 1\}^n \rightarrow \{0, 1\}$ is also called a **truth function**.

i **Definition 7.2:** Every Boolean expression $\phi(x_1, x_2, \dots, x_n)$ corresponds to a truth function

$$\hat{\phi}: \{0, 1\}^n \rightarrow \{0, 1\}$$

as follows: for every assignment $(a_1, a_2, \dots, a_n) \in \{0, 1\}^n$

$$\hat{\phi}(a_1, a_2, \dots, a_n) = \phi(a_1, a_2, \dots, a_n)$$

¹ The row includes only the values of the Boolean variables

The function $\hat{\phi}$ is called the **truth function of ϕ** (or the truth table of ϕ).

❖ **Example 7.1:** For example, we may denote the expression $x_1 \wedge (\neg x_2 \vee x_1)$ by $\phi(x_1, x_2)$. This notation allows us to perform a substitution such as $x_1 = 1, x_2 = 0$, in the abbreviated form $\phi(1, 0)$. It is not difficult to compute the truth function $\hat{\phi} : \{0, 1\}^2 \rightarrow \{0, 1\}$ of the expression ϕ :

$$\begin{aligned}\hat{\phi}(1, 1) &= 1 \\ \hat{\phi}(1, 0) &= 1 \\ \hat{\phi}(0, 1) &= 0 \\ \hat{\phi}(0, 0) &= 0\end{aligned}$$

This is, of course, another way to write the truth table of $x_1 \wedge (\neg x_2 \vee x_1)$. In fact, the formal definition of a truth table is the function $\hat{\phi}$!

❗ **Definition 7.3:** Two Boolean expressions ϕ, ψ are called **equivalent** if their truth tables are identical (that is, $\hat{\phi} = \hat{\psi}$). We denote this by the assertion $\phi \iff \psi$.

❖ **Exercise 7.2:** Prove that $x \vee y \iff \neg(\neg x \wedge \neg y)$.

* **Hint:** Show that the truth table of $\neg(\neg x \wedge \neg y)$ is identical to the truth table of $x \vee y$.

❖ The result of the preceding exercise is called **De Morgan's law**, named after the British logician Augustus De Morgan (1806-1871), who discovered it. There are two **Boolean identities** of this kind

$$\begin{aligned}x \vee y &\iff \neg(\neg x \wedge \neg y) \\ x \wedge y &\iff \neg(\neg x \vee \neg y)\end{aligned}\tag{7.1}$$

Another customary form of these identities is

$$\begin{aligned}\neg(x \vee y) &\iff \neg x \wedge \neg y \\ \neg(x \wedge y) &\iff \neg x \vee \neg y\end{aligned}\tag{7.2}$$

There are many other identities of this kind that we shall not bother to enumerate. As review exercises, we leave it to the student to prove the following identities (distributive and associative

laws)

$$\begin{array}{llll}
 x \wedge (y \vee z) & \iff & (x \wedge y) \vee (x \wedge z) & \text{(Distributivity)} \\
 x \vee (y \wedge z) & \iff & (x \vee y) \wedge (x \vee z) & \text{(Distributivity)} \\
 x \oplus (y \oplus z) & \iff & (x \oplus y) \oplus z & \text{(Associativity)} \\
 x \downarrow (y \downarrow z) & \iff & (x \downarrow y) \downarrow z & \text{(Associativity)}
 \end{array} \tag{7.3}$$

Again, this is merely a comparison of truth tables.

- The associative laws provide us with “syntactic convenience” by freeing us from the awkwardness involved in using parentheses. For example, it is far more convenient to write (and also to read) an expression such as $x_1 \vee x_2 \vee x_3 \vee x_4$ than $x_1 \vee (x_2 \vee (x_3 \vee x_4))$.

i Definition 7.4: A Boolean expression ϕ is called a **tautology** if all the values of its truth function are **1**.

- For example, the expression $x_1 \vee \neg x_1$ is a tautology. Using De Morgan’s law, it is easy to prove that $\neg(x_1 \wedge \neg x_1)$ is a tautology.
- If ϕ is any Boolean expression and τ is any tautology, then the expression $\phi \vee \tau$ is a tautology. In addition, it is easy to prove the following identity

$$\phi \wedge \tau \iff \phi \tag{7.4}$$

- ◆ **Example 7.2:** Identifying tautologies within Boolean expressions allows them to be simplified. For example, the expression

$$\phi = (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3)$$

By the associative law,

$$\phi \iff x_1 \wedge x_2 \wedge (x_3 \vee \neg x_3)$$

Since the expression $x_3 \vee \neg x_3$ is a tautology, by identity (7.4), we obtain

$$\phi \iff x_1 \wedge x_2$$

- As we shall soon see, Boolean algebra is of great importance in the fields of logic-circuit and electronic-chip design. For reasons of optimization, it is customary in these fields to restrict oneself to a minimal set of logical connectives (to save manufacturing costs and simplify design procedures). Let $\mathcal{C}$ be a set of logical connectives. We denote by $\mathcal{B}(\mathcal{C})$ the language of all Boolean expressions that can be constructed using only Boolean variables and connectives from



the set $\mathcal{C}$.

- For example, the expression $\neg x_1 \wedge x_2$ belongs to the language $\mathcal{B}(\neg, \wedge)$, but does not belong to the language $\mathcal{B}(\neg, \oplus, \uparrow)$.
- To restrict ourselves to a minimal set of logical connectives, we must ensure that it has maximal expressive power. That is, all possible Boolean functions can be expressed using it. Here is the precise definition.

i Definition 7.5: A set $\mathcal{C}$ of logical connectives is called **complete** if, for every truth function $f: \{0, 1\}^n \rightarrow \{0, 1\}$, there is a Boolean expression $\phi(x_1, x_2, \dots, x_n)$ in the language $\mathcal{B}(\mathcal{C})$ such that $f = \hat{\phi}$. The set $\mathcal{C}$ is called **minimally complete** if it is complete, but each of its proper subsets is not complete.

- In other words, a set of connectives is complete if it can be used to construct a Boolean expression for every possible truth table.

i Proposition 7.1: The set of connectives $\{\neg, \wedge, \vee\}$ is complete.

- ✓ **Proof:** Let $f: \{0, 1\}^n \rightarrow \{0, 1\}$ be any truth function. We must prove the existence of a Boolean expression ϕ in the language $\mathcal{B}(\neg, \wedge, \vee)$ such that $\hat{\phi} = f$. Since the subject of Boolean algebra lies beyond the scope of this course, we shall present only the idea of the proof for the case $n = 3$. Suppose, for example, that the function $f: \{0, 1\}^3 \rightarrow \{0, 1\}$ is given by

$$\begin{array}{ll} f(1, 1, 1) = 1 & f(0, 1, 1) = 0 \\ f(1, 1, 0) = 1 & f(0, 1, 0) = 1 \\ f(1, 0, 1) = 0 & f(0, 0, 1) = 0 \\ f(1, 0, 0) = 1 & f(0, 0, 0) = 0 \end{array}$$

Define the expression $\phi(x_1, x_2, x_3)$ by

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \quad (7.5)$$

The algorithm for constructing ϕ is simple: for each assignment $(a_1, a_2, a_3) \in \{0, 1\}^3$ satisfying $f(a_1, a_2, a_3) = 1$, construct an expression $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ such that $\varphi_i = x_i$ if $a_i = 1$; otherwise, $\varphi_i = \neg x_i$. For example, the assignment $(0, 1, 0)$ corresponds to the expression $\neg x_1 \wedge x_2 \wedge \neg x_3$. Then join all these expressions with the connective $\vee$ to obtain expression (7.5). It is easy to see that the truth table of $\phi(x_1, x_2, x_3)$ is precisely the function f . ■

i Definition 7.6: A Boolean expression of the form

$$\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n$$

is called a **disjunction**.



A Boolean expression of the form

$$\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$$

is called a **conjunction**.

A Boolean expression of the form

$$(\varphi_1^1 \wedge \varphi_2^1 \wedge \cdots \wedge \varphi_{n_1}^1) \vee (\varphi_1^2 \wedge \varphi_2^2 \wedge \cdots \wedge \varphi_{n_2}^2) \vee \cdots \vee (\varphi_1^m \wedge \varphi_2^m \wedge \cdots \wedge \varphi_{n_m}^m) \quad (7.6)$$

in which every expression φ_j^i is a Boolean variable x_k or its negation $\neg x_k$, is said to be in **disjunctive normal form**, or, briefly, a **DNF expression**.

- In brief: the general form of a Boolean expression in DNF is a disjunction all of whose components are conjunctions of expressions of the form x_k or $\neg x_k$. Proposition 7.1 can also be stated as follows.

i Corollary 7.2: For every truth function $f : \{0, 1\}^n \rightarrow \{0, 1\}$, there exists a DNF expression ϕ such that

$$f = \hat{\phi}$$

i Corollary 7.3: For every Boolean expression there is an equivalent DNF expression.

- ✓ **Proof:** Let ϕ be any Boolean expression. Consider the Boolean function $\hat{\phi}$. By the preceding corollary, there is a DNF expression ψ such that $\hat{\phi} = \hat{\psi}$. That is, $\phi \iff \psi$. ■
- Similarly, we define an expression in **Conjunctive Normal Form**, or, briefly, CNF, as a Boolean expression of the form

$$(\varphi_1^1 \vee \cdots \vee \varphi_{n_1}^1) \wedge (\varphi_1^2 \vee \cdots \vee \varphi_{n_2}^2) \wedge \cdots \wedge (\varphi_1^m \vee \cdots \vee \varphi_{n_m}^m)$$

where φ_j^i is a Boolean variable x or its negation $\neg x$.

i Proposition 7.4: For every Boolean expression there is an equivalent CNF expression.

- ✓ **Proof:** We present only the idea of the proof. Let ϕ be a given Boolean expression. By Corollary 7.3, there is a DNF expression ψ equivalent to the negation $\neg\phi$. Using De Morgan's law, it is easy to transform $\neg\psi$ into a CNF expression equivalent to ϕ . ■
- ◆ **Example 7.3:** From the truth table of the logical connective **NOR**, it is easy to find its DNF and CNF forms

$$\begin{aligned} x \downarrow y &\iff \neg x \wedge \neg y && \text{(DNF)} \\ x \downarrow y &\iff (\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) && \text{(CNF)} \end{aligned} \quad (7.7)$$



The CNF form is not intuitive in many cases, but may be important in practice.

i Proposition 7.5: The set of connectives $\{\neg, \wedge\}$ is minimally complete.

✓ Proof: By De Morgan's law, $\phi \vee \psi \iff \neg(\neg\phi \wedge \neg\psi)$. That is, the connective $\vee$ can be defined using $\{\neg, \wedge\}$. Therefore, by Proposition 7.1, the set of connectives $\{\neg, \wedge\}$ is complete. The set $\{\neg\}$ is not complete because it cannot be used to construct expressions containing more than one Boolean variable. The set $\{\wedge\}$ is not complete, as we shall see in Corollary 7.10. ■

i Proposition 7.6: The set of connectives $\{\neg, \vee\}$ is minimally complete.

✓ Proof: By De Morgan's law, $\phi \wedge \psi \iff \neg(\neg\phi \vee \neg\psi)$. That is, the connective $\wedge$ can be defined using the connectives $\{\neg, \vee\}$. Therefore, by the preceding proposition, the set of connectives $\{\neg, \vee\}$ is complete. The set $\{\vee\}$ is not complete, as we shall see in Corollary 7.10. ■

i Proposition 7.7: The connective **NAND** constitutes a complete set of connectives.

✓ Proof: By Proposition 7.5, it is enough to prove that the connectives $\{\neg, \wedge\}$ can be defined using the connective $\uparrow$:

$$\begin{aligned}\neg\phi &\iff \phi \uparrow \phi \\ \phi \wedge \psi &\iff \neg(\phi \uparrow \psi) \\ &\iff (\phi \uparrow \psi) \uparrow (\phi \uparrow \psi)\end{aligned}$$

We leave it to the reader to verify the details. ■

i Proposition 7.8: The connective **NOR** constitutes a complete set of connectives.

✓ Proof: By Proposition 7.6, it is enough to prove that the connectives $\{\neg, \vee\}$ can be defined using the connective $\downarrow$.

$$\begin{aligned}\neg\phi &\iff \phi \downarrow \phi \\ \phi \vee \psi &\iff \neg(\neg\phi \wedge \neg\psi) \\ &\iff \neg(\phi \downarrow \psi) \\ &\iff (\phi \downarrow \psi) \downarrow (\phi \downarrow \psi)\end{aligned}$$

i Proposition 7.9: Every expression $\psi(x_1, x_2)$ in the language $\mathcal{B}(\vee, \wedge)$ satisfies $\psi(1, 1) = 1$.

🔗 Solution: We use induction on the length of ψ . The shortest expressions in the language are the Boolean variables x_i , for which the assertion is trivial. Let ψ be an expression of length $n > 1$ in the language $\mathcal{B}(\vee, \wedge)$. Then $\psi = \psi_1 \vee \psi_2$ or $\psi = \psi_1 \wedge \psi_2$, where $\psi_1, \psi_2 \in \mathcal{B}(\vee, \wedge)$ are expressions of length less than n . By the induction hypothesis, $\psi_1(1, 1) = 1$, $\psi_2(1, 1) = 1$,



and therefore, in both cases,

$$\psi(1, 1) = \psi_1(1, 1) \vee \psi_2(1, 1) = 1 \vee 1 = 1$$

$$\psi(1, 1) = \psi_1(1, 1) \wedge \psi_2(1, 1) = 1 \wedge 1 = 1$$

■

- The following proposition is a simple example of proving that a computational possibility is infeasible.

i **Corollary 7.10:** The set of connectives $\{\vee, \wedge\}$ is not complete.

- ✓ **Proof:** It is enough to prove the existence of a single expression $\phi(x_1, x_2)$ for which there is no equivalent expression in the language $\mathcal{B}(\vee, \wedge)$. For example, it is not difficult to conjecture that an expression such as $\phi = x_1 \uparrow x_2$ is not logically equivalent to any expression in the language $\mathcal{B}(\vee, \wedge)$. The basic technique for proving assertions of this kind is based on the following idea: find a property common to all expressions in the language $\mathcal{B}(\vee, \wedge)$ and prove that ϕ does not have it. In the preceding proposition, we proved that every expression $\psi(x_1, x_2)$ in the language $\mathcal{B}(\vee, \wedge)$ satisfies $\psi(1, 1) = 1$. The expression $\phi = x_1 \uparrow x_2$ does not have this property, and hence it has no equivalent expression in the language $\mathcal{B}(\vee, \wedge)$. ■

- Boolean functions can also be defined by arithmetic operations, provided that they produce a Boolean value. For example,

$$f(x_1, x_2, x_3, x_4) = |\max(x_1, x_2) - \min(x_3, x_4)|$$

defines an interesting Boolean function that is challenging to express using the standard logical connectives (in the shortest possible form; see Exercise 2 at the end of the chapter).

7.2 Logic Circuits

- A logic circuit is the simplest computational model based on Boolean algebra. In its simplest form, the computational model is based on the structure of a directed graph, although in practice a directed multigraph model is required. We begin with the simple model.

i **Definition 7.7:** A **logic circuit** is a finite, directed, acyclic graph with no isolated vertices (that is, every vertex in the graph has at least one incoming or outgoing edge).

(a) A vertex with no incoming edges is called an **input gate**.

(b) A vertex with one incoming edge but no outgoing edges is called an **output gate**.

(c) Every other vertex is called an **internal gate** or a **logic gate**.

(d) Every logic gate must have an outgoing edge to another logic gate or to an output gate (or



to both).

- (e) Every logic gate represents a Boolean function that is to be applied to the values emerging from the gates entering it. The result of the function is passed to the gates to which it has outgoing edges.
- (f) The number of inputs of every logic gate must be compatible with the type of Boolean function associated with it. For example, a logic gate of type **NOT** cannot have more than one input!

❖ **Example 7.4:** In Figure 7.1, we see a logic circuit with

- (a) input gates: x_1, x_2, x_3, x_4, x_5
- (b) output gates: y_1, y_2, y_3, y_4
- (c) logic gates: g_1, g_2, g_3, g_4, g_5

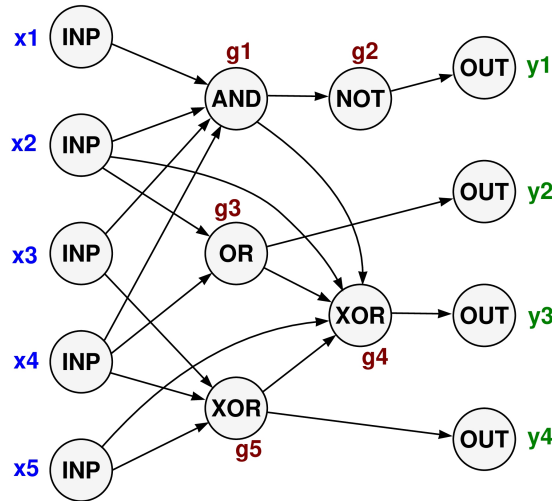


Fig. 7.1: A logic circuit

Each of the gates represents a logical operator (**AND**, **OR**, **NOT**, and twice **XOR**).

- Formally, the definition of the logic circuit in Figure 7.1 must include three things: a set of vertices V , a set of edges E , and a set of labels L

$$\begin{aligned}
 V &= \{x_1, x_2, x_3, x_4, x_5, g_1, g_2, g_3, g_4, g_5, y_1, y_2, y_3, y_4\} \\
 E &= \{(x_1, g_1), (x_2, g_1), (x_2, g_3), (x_2, g_4), (x_3, g_1), (x_3, g_5), (x_4, g_1), \\
 &\quad (x_4, g_3), (x_4, g_5), (x_5, g_4), (x_5, g_5), (g_1, g_2), (g_1, g_4), (g_2, y_1), \\
 &\quad (g_3, y_2), (g_3, g_4), (g_4, y_3), (g_5, g_4), (g_5, y_4)\} \\
 L &= \{(g_1, \text{AND}), (g_2, \text{NOT}), (g_3, \text{OR}), (g_4, \text{XOR}), (g_5, \text{XOR})\}
 \end{aligned}$$

In most cases, however, we shall be satisfied with a clear diagram that provides all this information visually.

- The diagram provides further insights beyond the specification. For example, Figure 7.1 makes it clear why the definition requires the graph to be acyclic. What would happen, for example, if an

edge from gate **g2** led back to gate **g1**? An infinite loop would be created, and the computation would never terminate. We also observe that logic gate **g3** has an outgoing edge to gate **g4** and also to output vertex **y2**.

- The reason for the term **logical operator** is that this is not an ordinary Boolean function, but rather a general name for an infinite family of Boolean functions having the same definition (apart from the number of variables). For example, the logical operator **AND** names the family of all Boolean functions $f_n: \{0, 1\}^n \rightarrow \{0, 1\}$ defined as follows:

$$f_n(x_1, x_2, \dots, x_n) = \prod_{i=1}^n x_i$$

The expression $\prod_{i=1}^n x_i$ denotes the arithmetic product of all input elements $(x_1, x_2, \dots, x_n)$. We use the operator notation $\text{AND}(x_1, x_2, \dots, x_n)$ for all these functions. We note that in various contexts it is customary to write $x_1 \wedge x_2 \wedge \dots \wedge x_n$. Similarly, the logical operator **OR** is defined by

$$\text{OR}(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \exists i: x_i = 1 \\ 0, & \forall i: x_i = 0 \end{cases}$$

The operator **NOR** is customarily defined by

$$\text{NOR}(x_1, x_2, \dots, x_n) = \text{NOT}(\text{OR}(x_1, x_2, \dots, x_n))$$

It is not difficult to verify that

$$\text{NOR}(x_1, x_2, \dots, x_n) = 1 \iff [x_1 = 0 \wedge x_2 = 0 \wedge \dots \wedge x_n = 0]$$

The operator **XOR** is defined by

$$\text{XOR}(x_1, x_2, \dots, x_n) = \begin{cases} 1 & \sum_{i=1}^n x_i = 1 \\ 0 & \sum_{i=1}^n x_i \neq 1 \end{cases}$$

Equivalently:

$$\text{XOR}(x_1, x_2, \dots, x_n) = 1 \iff \sum_{i=1}^n x_i = 1$$

That is, the operator **XOR** has the value **1** if and only if exactly one element of the input sequence is **1**.

- A logic circuit is in fact a generalization of a **parse tree** in the language of Boolean expressions (although it is not a tree in the strict sense). As noted above, the language of Boolean expressions is, of course, context-free.



❖ **Example 7.5:** The logic circuit **COUNT3** in Figure 7.2 counts the number of ones in its input string. The result is, of course, obtained in binary notation. If, for example, we feed the values $(x_1, x_2, x_3) = (1, 0, 1)$ into the circuit, we obtain the output $(y_1, y_2) = (1, 0)$, which in decimal notation is, of course, 2.

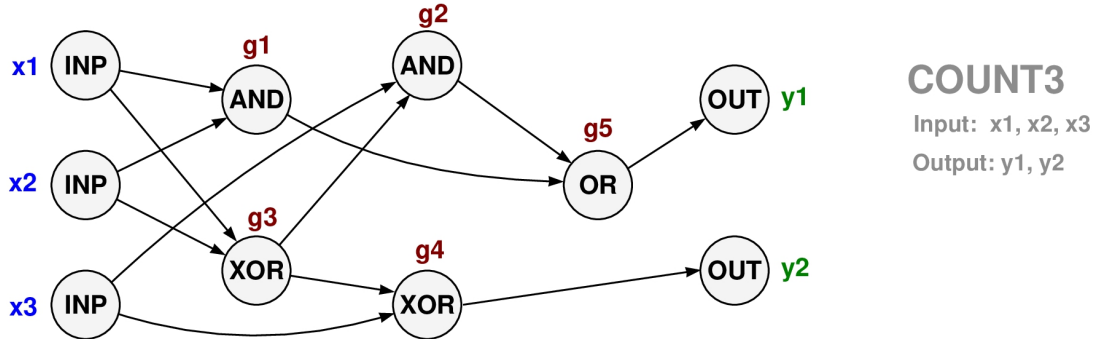


Fig. 7.2: The logic circuit **COUNT3** counts the number of input bits whose value is 1 in a 3-bit input

There can be at most three ones in a 3-bit input, and hence two bits suffice to count the number of ones in the input. At first glance, it is not clear how the circuit performs this task; it is advisable to check several examples by hand to gain an understanding of how it works. Various simulation programs available on the Internet can also be used for this purpose. Later in the chapter, we present a simulation program written especially for this chapter, in which logic circuits of this kind can be modeled and run (click the following link for further details).

PyCirc - Python Logic Circuit Simulation Package

- Methods for constructing and designing logic circuits to perform tasks of this kind are, of course, beyond the scope of this course. The interested student can find them in more advanced courses on logic design, VLSI circuit design, digital systems, and the like.
- To define how a **computation** is performed by a logic circuit, we need the concept of the **depth** of a logic gate and of a logic circuit.

i Definition 7.8: Let $G = (V, E, L)$ be a given logic circuit. For every gate g in the circuit, define its depth $\text{depth}(g)$ recursively

- ◆ If g is an input gate (INP), then its depth is zero: $\text{depth}(g) = 0$
- ◆ If g is not an input gate, then its depth is determined by the gates entering it

$$\text{depth}(g) = 1 + \text{Max} \{ \text{depth}(v) : (v, g) \in E \}$$

Recall that E is the set of edges in the graph G .

The **depth of the circuit** G is the maximum depth of its gates and is denoted by $\text{depth}(G)$.

❖ **Example 7.6:** We return to the logic circuit in Figure 7.2 and try to map each gate to its depth by a simple manual procedure. We begin with the **INP** input gates, whose depth is zero by definition, and proceed to the following gates in traversal order. See Figure 7.3.

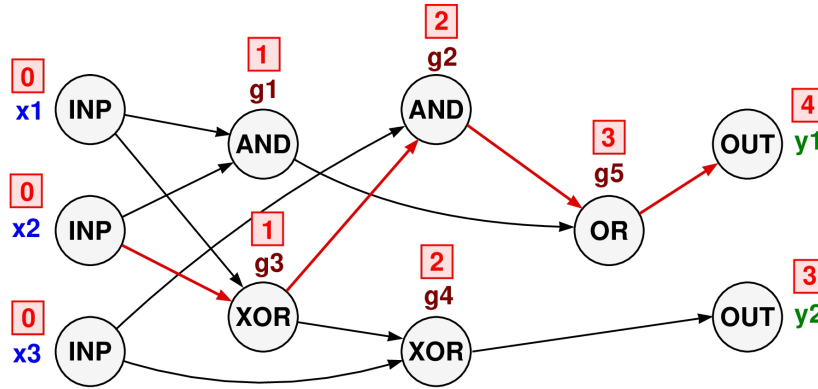


Fig. 7.3: A logic circuit of depth 4. The red label indicates the depth of the gate.

By definition, the depth of the circuit in Figure 7.3 is 4. The fact that a logic circuit contains no cycles guarantees that the recursion in Definition 7.8 always terminates at **INP** gates. The following exercise presents an equivalent definition of the depth of a gate or the depth of a circuit, which is sometimes found in other texts.

✦ **Exercise 7.3:** Prove that $\text{depth}(g) = n$ if and only if the maximum length of a path in the graph G that ends at gate g is n (the path length is measured by the number of edges). Deduce that $\text{depth}(G) = n$ if and only if the length of a longest path in the graph G is n .

* **Hint:** Induction on the depth of the gate. In the graph in Figure 7.3, there are exactly two paths of maximum length 4. One of them is marked in red (what is the other path?).

➤ The length of a longest path in a logic circuit is critically important in the design of electronic chips, since these paths generally determine the response time of the circuit (the time elapsed between receiving the input and obtaining the output). The assumption is that the transition from each stage to the next takes one unit of computation time, and therefore a circuit of depth n requires at least n time units to complete the computation.

✦ **Example 7.7:** In Figure 7.4, we converted the logic circuit from Figure 7.1 into a **syntactic construction graph**, in which every logic gate (g_1, g_2, g_3, g_4, g_5) is replaced by its corresponding Boolean expression. There is a strong resemblance between a syntactic construction graph and the **parse tree** of a sentence in a context-free language, which we encountered in Chapter 4. It is advisable to review Definition 4.11 in Chapter 4 (page 124). See also Exercise 1 at the end of the chapter.

Unlike an ordinary parse tree, which shows how a sentence decomposes into its simple components, in a construction graph the direction of growth is reversed: we begin with the elementary components (the Boolean variables x_1, x_2, x_3, x_4, x_5), and proceed to increasingly complex expressions. We note that, contrary to common practice, we have added output gates to the graph, y_1, y_2, y_3, y_4 , as is customary in most industrial models of logic circuits (this has a small cost in terms of additional physical area and logical complexity, but the gain is enormous in terms of simplifying design procedures and modular representation).

➤ We also observe that Figure 7.4 includes a **depth-level scale**. At level 0 (Level 0), we see the

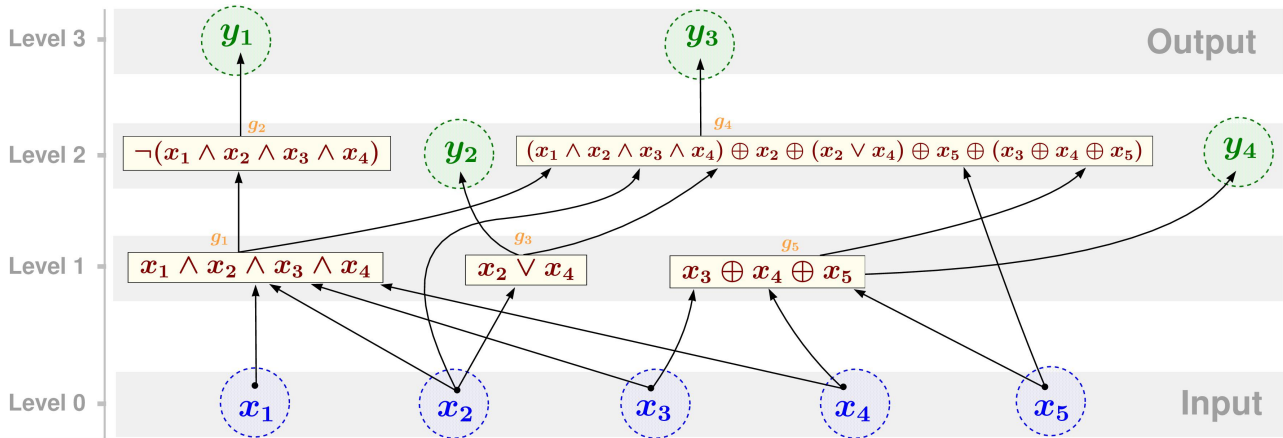


Fig. 7.4: Representation of the logic circuit in Figure 7.1 by a syntactic construction graph

simplest expressions: Boolean variables. At every higher level appear the expressions that can be constructed from expressions at lower levels. As we shall see below (Definition 7.8), the highest level number is also called the **depth of the graph** and serves as a kind of complexity measure for logic circuits.

❖ **Example 7.8:** In Figure 7.5, we converted the logic circuit **COUNT3** from Figure 7.2 into a **syntactic construction graph**, in which every logic gate (g_1, g_2, g_3, g_4, g_5) is replaced by its corresponding Boolean expression. Unlike the preceding example, this time the depth of the graph increases from left to right. Again, note the depth scale, which classifies the gates into depth levels. Each depth degree k corresponds to a depth level containing all vertices of depth degree k .

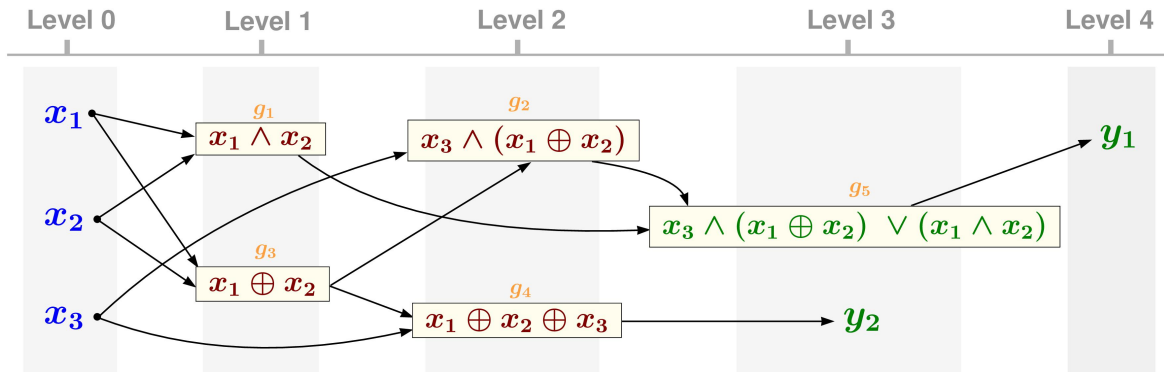


Fig. 7.5: Representation of the logic circuit in Figure 7.2 by a syntactic construction graph

As can be learned from the last two examples, the notion of depth allows us to classify all gates into depth levels.

❗ **Definition 7.9:** Let $G = (V, E, L)$ be a logic circuit of depth n . For every $k < n$, the set of all gates of depth degree k

$$L_k(G) = \{g \in V : \text{depth}(g) = k\}$$

is called **level k of the circuit**. The **width of the circuit** is the maximum size of the levels $L_k(G)$.

In Example 1, the width of the circuit is **3**; in Example 2, the width of the circuit is **2**

i Definition 7.10: Let $\mathbf{G} = (\text{Gates}, \text{Edges})$ be a logic circuit with input gates $(x_1, x_2, \dots, x_n)$, logic gates $(g_1, g_2, \dots, g_k)$, and output gates $(y_1, y_2, \dots, y_m)$.

♦ A **frame** of $\mathbf{G}$ is a mapping

$$F: \text{Gates} \rightarrow \{0, 1, \text{NA}\}$$

between the set of gates of $\mathbf{G}$ and their values at a given time. Each gate may take one of three possible values: **0**, **1**, **NA**. The expression **NA** is the **null value**; it indicates that the gate has not yet been initialized, that is, that it has no value.

♦ The frame F is called the **null frame** of $\mathbf{G}$ if $F(g) = \text{NA}$ for every g .

♦ Let $\vec{a} = (a_1, a_2, \dots, a_n) \in \{0, 1\}^n$ be an assignment for $\mathbf{G}$. The frame f is called the **initial frame** of the assignment $\vec{a}$ if, for every input gate x_i , $F(x_i) = a_i$, and for every other gate g , $F(g) = \text{NA}$.

➤ The initial frame for an assignment $\vec{a} = (a_1, a_2, \dots, a_n)$ is in fact the initialization of the input gates of the logic circuit $\mathbf{G}$; all the other gates of the circuit are still set to the null value **NA**.

➤ Our challenge is to characterize the details of the computation process in a logic circuit formally. Most textbooks skip this stage and settle for a loose explanation, but in the computing age in which we now find ourselves, without a rigorous formal specification of this process, it is impossible to write software and construct simulators for logic circuits, a necessary stage preceding the design and manufacture of electronic chips. Since this booklet is industry-oriented, we shall spend some time on the formal specification. We use the logic circuit in Figure 7.5 to illustrate the stages of the computation.

➤ **Null frame:**

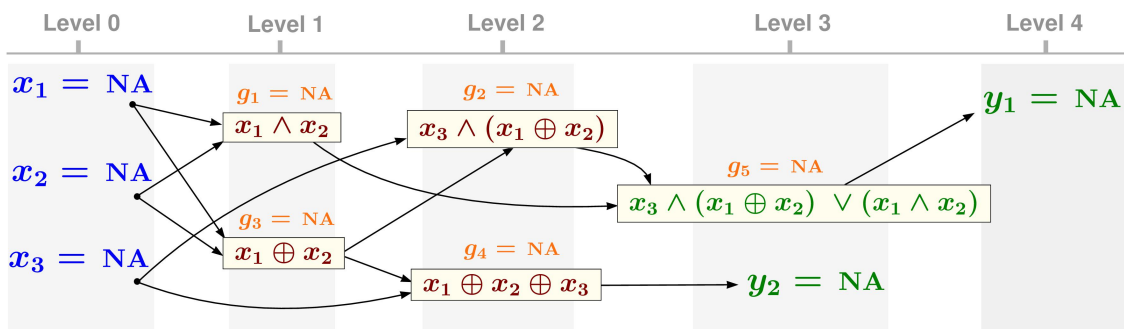


Fig. 7.6: The null frame of the logic circuit **COUNT3**

➤ Depth-0 frame:

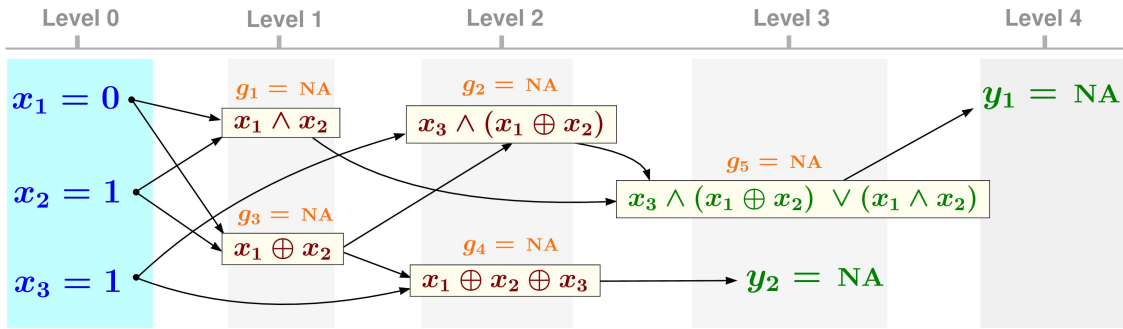


Fig. 7.7: Initial frame of the logic circuit **COUNT3** for the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

➤ Depth-1 frame:

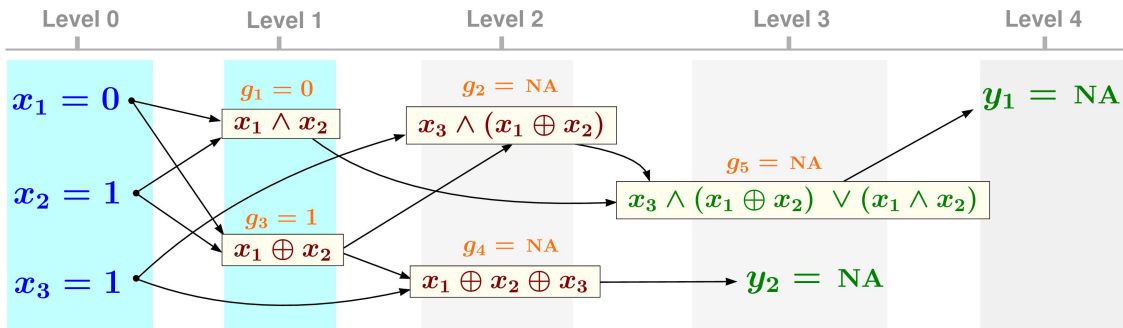


Fig. 7.8: Depth-1 frame of the logic circuit **COUNT3** for the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

➤ Depth-2 frame:

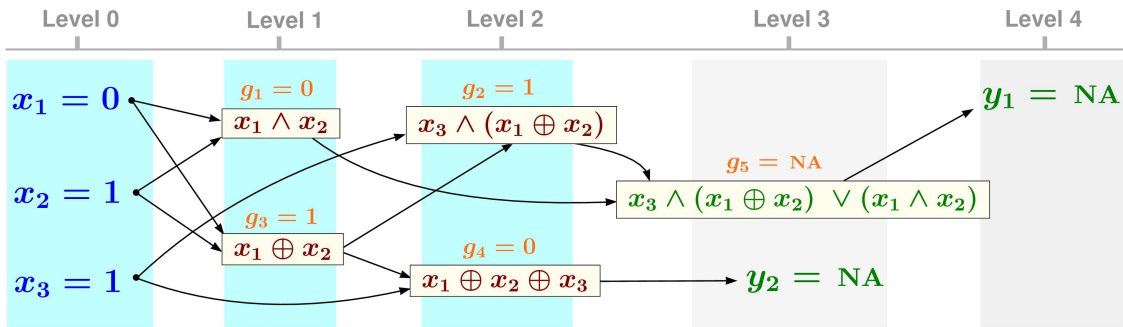


Fig. 7.9: Depth-2 frame of the logic circuit **COUNT3** for the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

➤ Depth-3 frame:

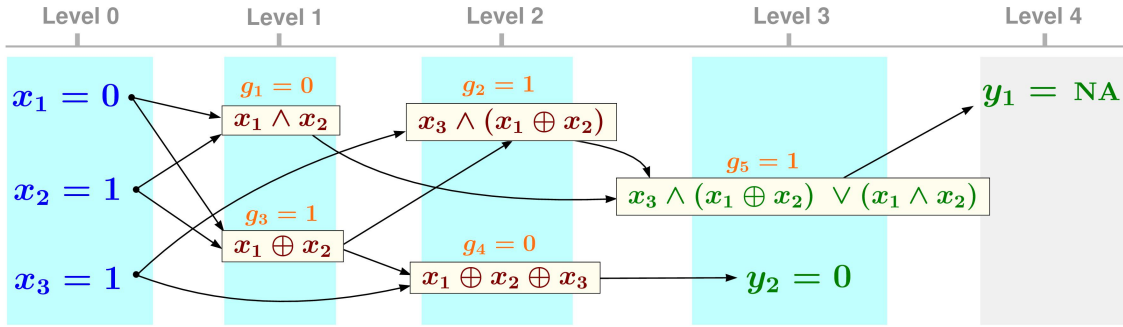


Fig. 7.10: Depth-3 frame of the logic circuit **COUNT3** for the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

Depth-4 frame:

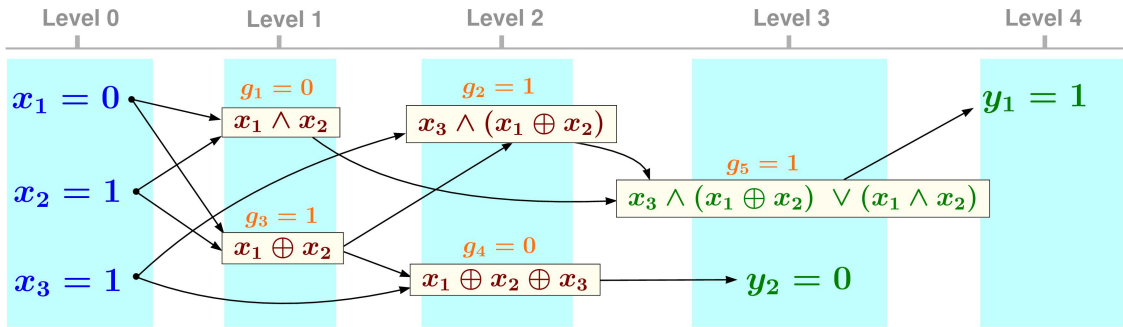


Fig. 7.11: Depth-4 frame of the logic circuit **COUNT3** for the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

- Definition 7.11: (Derivation)** A frame F_2 is said to be derived from a frame F_1 if
- (a) The input values in both frames are initialized to the same values.
 - (b) There exists k such that all depth levels $i \leq k$ of F_1 are initialized (that is, the gate values are 0 or 1). For every $i > k$, the depth level of F_1 is not initialized.
 - (c) All depth levels $i \leq k + 1$ of F_2 are initialized, and all levels $i > k + 1$ are not initialized.
 - (d) The values of the gates at level $k + 1$ in frame F_2 are obtained by applying their logical function to the values from the levels below them.

- Definition 7.12:** Let $\vec{a} = (a_1, a_2, \dots, a_n) \in \{0, 1\}^n$ be an assignment for a logic circuit G . A **computation sequence** of G for the input $\vec{a}$ is a sequence of frames $\{F_i\}_{i=0}^m$ in which every frame f_{i+1} is derived from its predecessor f_i .
- Let $\vec{a} = (a_1, a_2, \dots, a_n) \in \{0, 1\}^n$ be an assignment for a logic circuit G . The computation is a sequence of frames $\{F_i\}_{i=1}^m$, F_1 is the initial frame of G

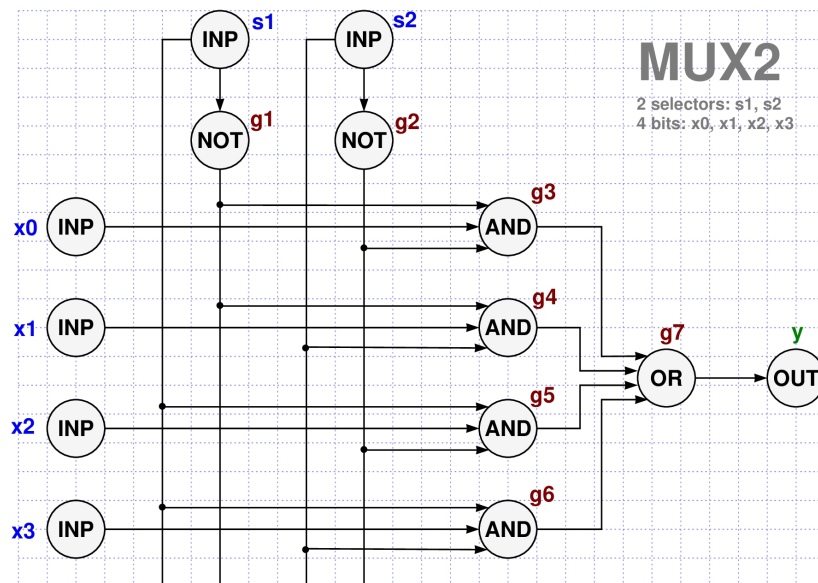
Example 7.9: We now show how the computation is performed in the logic circuit **COUNT3** from Example 7.5

Figure 7.2 presents the **computation table** of the logic circuit as a list of frames. Row 0 of the table presents the frame in which the input **101** has been fed into the circuit, but no operation has yet been performed. Row 1 of the table presents the circuit frame after the logic gates at depth 1 have been activated (these are gates g_1 , g_3).

Frame	x_1	x_2	x_3	g_1	g_2	g_3	g_4	g_5	y_1	y_2
0	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
1	1	0	1	NA	NA	NA	NA	NA	NA	NA
2	1	0	1	0	NA	1	NA	NA	NA	NA
3	1	0	1	0	1	1	0	1	NA	NA
4	1	0	1	0	1	1	0	0	1	0

Table 7.2: Computation table for the logic circuit COUNT3 with input 101

- The frame diagrams above and the last computation table were produced using a hardware-description language for logic circuits called PyCircHDL, which was developed especially for this course and will be presented in greater detail in the next section.
- ◆ **Example 7.10:** A multiplexer of order k is a logic circuit MUX_k that receives two input sequences: a sequence of $n = 2^k$ binary values $(x_0, x_1, \dots, x_{n-1})$, and a sequence of k selectors $(s_1, s_2, \dots, s_k)$.

Fig. 7.12: A multiplexer of type MUX2. Input: a pair of selectors (s_1, s_2) , four bits (x_0, x_1, x_2, x_3) . Output: one bit y .

- ◆ The output of MUX_k is a single bit y that takes the value x_i where i is the natural number represented by $s_1 s_2 \dots s_k$ in binary notation. There are exactly 2^k natural numbers that can be represented by k bits, and therefore k selectors can select any one of the bits x_i , $0 \leq i \leq n - 1 = 2^k - 1$. Figure 7.12 shows an implementation of a MUX2 multiplexer by a logic circuit with 4 input bits (x_0, x_1, x_2, x_3) and a pair of selectors (s_1, s_2) .

- ◆ The following table summarizes the operation of the **MUX2** multiplexer shown in Figure [7.12](#)

s ₁	s ₂	y
0	0	x ₀
0	1	x ₁
1	0	x ₂
1	1	x ₃

The multiplexer is an important elementary electronic component used extensively in more complex logic circuits. In most cases, it is used to select the appropriate input (from among the bits entering it) by using the selectors as switches.

- ◆ This example also allows us to present the conventional drawing method used in electronics and logic-design textbooks for logic circuits. All lines are vertical or horizontal, as they are on a physical silicon chip. Two intersecting lines are connected only when a conspicuous black dot appears at their intersection; otherwise, they may be regarded as two lines on different layers of the chip, with no connection between them (the physical chip is actually a three-dimensional body with several layers, with alternating vertical and horizontal wires running on each layer).

Constructing Complex Circuits Using Library Cells

- Every logic circuit that we have constructed with considerable effort can serve as a **component** in the design of more complex circuits, thereby saving us much effort and also simplifying the diagrams substantially. Figure [7.13](#) shows 12 typical examples of basic library cells. A library cell may be highly complex and contain dozens or hundreds of components (including library cells!), but from the perspective of the design process, all that is required is knowledge of its functional specification: which gates are its inputs and outputs, and what function it performs; everything between the inputs and outputs appears as a black box.
- From now on, we shall use a more rigorous notation in which the names of the inputs and outputs of every cell that we use (pins) must also be specified. There are infinitely many **AND** units, characterized by their number of inputs: **AND2**, **AND3**, **AND4**, and so on. In addition, every input and every output has a fixed name! Below, for every edge, we shall have to state clearly from which output it emerges and into which input it enters, using the names of the inputs and outputs.

Figure [7.13](#) shows 12 typical examples of library cells. Note the cells with several outputs (**COUNT3**, **NOT3**, **ZERO3**), and the cells with no inputs at all! (**ZERO**, **ONE**, **ZERO3**). We shall later see how cells with no inputs, such as **ZERO** or **ONE**, serve as **constants**. The cell **ZERO**, for example, receives no input but constantly emits zero. It can be used to set inputs of other cells to zero and thereby define new cells.

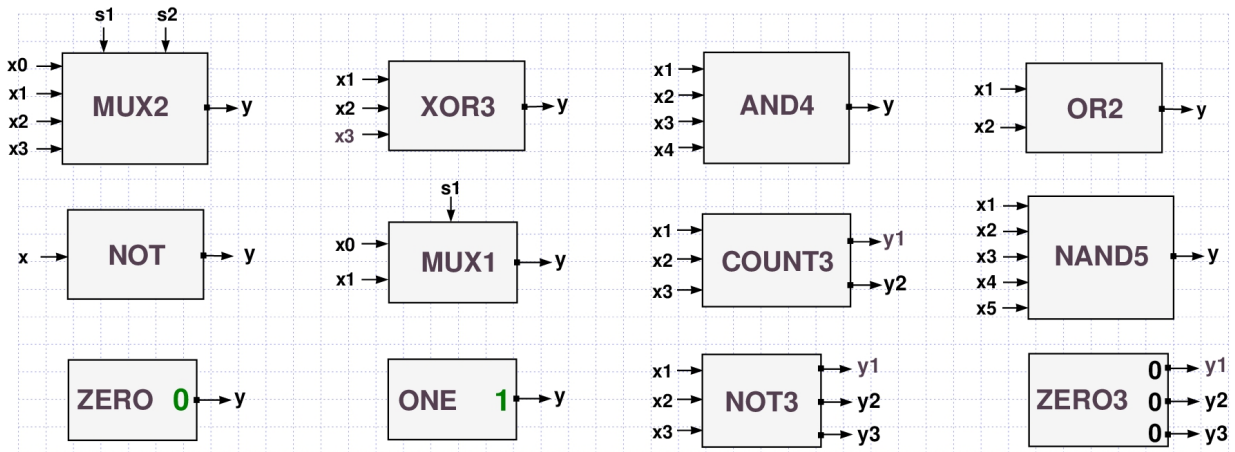


Fig. 7.13: Minimal architecture of typical library cells

❖ **Example 7.11:** It turns out that the models of a logic circuit encountered in Examples 7.4, 7.5, 7.10 are too simple and are unsuitable for describing industrial logic circuits, although most textbooks use them for an initial introduction to the subject. Figure 7.14 presents an example of a logic circuit that is very close to a real-world industrial model. In this model, the names of the inputs and outputs play a critical role. Note all the subtleties in the diagram:

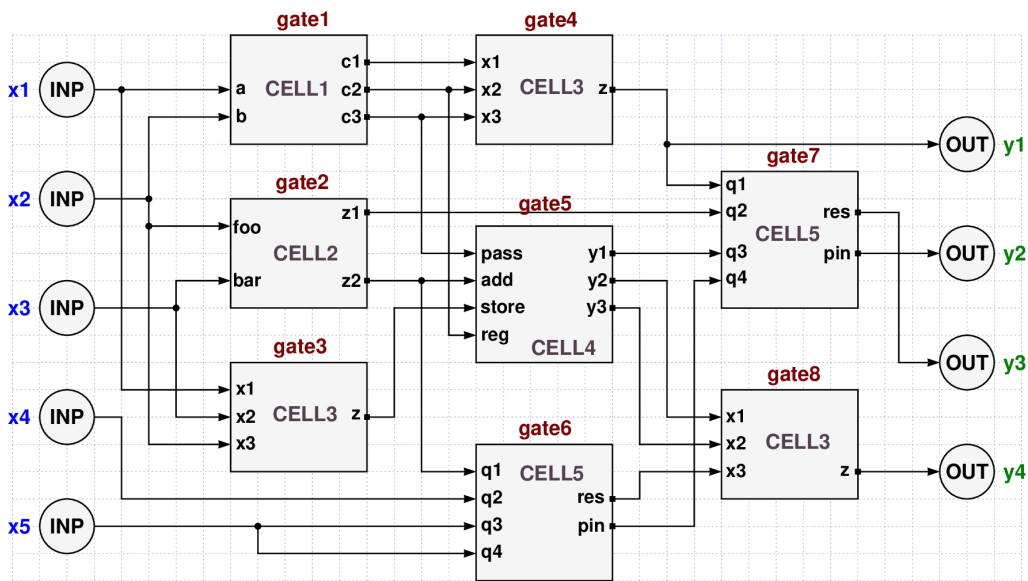


Fig. 7.14: A more realistic architecture of a real-world logic circuit

- ◆ Our circuit includes 7 gates **gate1...gate7** belonging to 5 cell types **CELL1...CELL5**. Gates **gate3**, **gate4**, **gate8** have the same cell type **CELL3**. This can also be seen from the number and names of their inputs x_1 , x_2 , x_3 , and one output z (but this is, of course, insufficient; the logical function must be identical). The type of gates **gate6**, **gate7** is **CELL5**.
- ◆ In most cases encountered so far (apart from the addition units), a logic gate had only one output, but formally it may have several distinct outputs, as can be seen in gates **gate1**, **gate2**, **gate5**, **gate6**, **gate7**.
- ◆ A more fundamental phenomenon that we encounter for the first time is the possibility of multiple edges between two gates. In the preceding directed-graph model (DiGraph), there can be at most

one edge connecting any two distinct gates. In our circuit, by contrast, there are three distinct edges from gate **gate1** to gate **gate4**! Likewise, there are two distinct edges from gate **gate5** to gate **gate8**. The first edge emerges from the (pin) **y₂** of **gate5** and enters pin **x₁** of **gate8**. The second edge emerges from pin **y₃** of **gate5** and enters pin **x₂** of **gate8**. Thus, the edges are distinguished by the output pin and input pin corresponding to each edge. This means that the preceding directed-graph model DiGraph is not sufficiently general, and we must move to the **directed multigraph (MultiDiGraph)** model, which permits multiple edges between a pair of vertices.

- ◆ The outputs from a given gate may split between two different gates: for example, Figure 7.14 shows that three edges emerge from gate **gate5**, one to gate **gate7** and two to gate **gate8**.
- ◆ Nevertheless, we shall continue to use the simple directed-graph model for circuits built from the elementary logical connectives, in which the names of inputs and outputs are unimportant.
- ◈ **Example 7.12:** For comparison, Figure 7.15 shows the MultiDiGraph model of the **MUX2** multiplexer encountered in Example 7.10

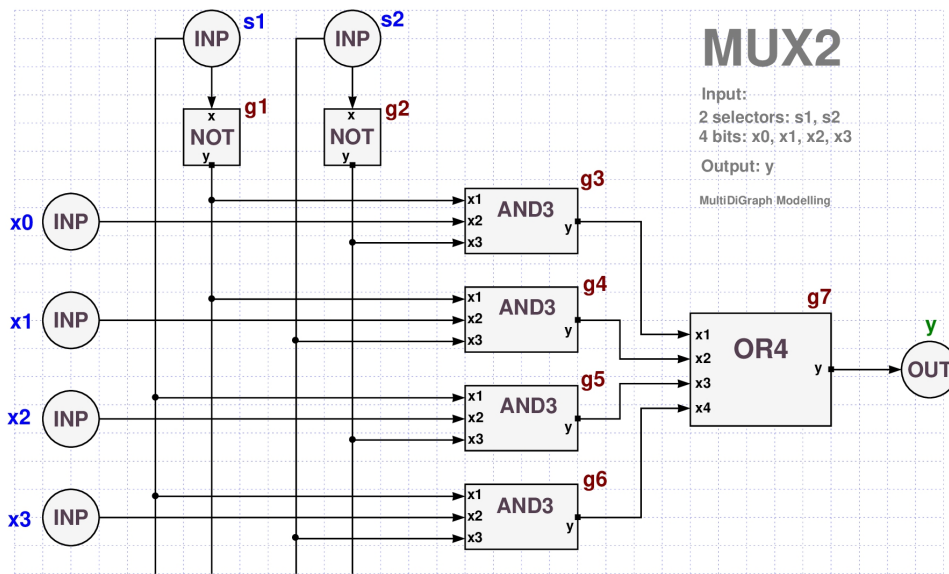


Fig. 7.15: The MultiDiGraph model of the **MUX2** multiplexer from Example 7.10

- It contains four instances of the cell **AND3**: **g3**, **g4**, **g5**, **g6**, one instance of the cell **OR4**: **g7**, and two instances of **NOT**: **g1**, **g2**. Of course, in this example the directed-graph model (DiGraph) is sufficient and preferable, but in many other cases there is no alternative to using the more general model.

7.3 Simulation and Hardware-Description Languages for Logic Circuits

- In the context of mathematical computational models, there are two main types of programming languages for investigating and analyzing them by means of computational tools. The first type consists of simulation languages, whose purpose is to convert the mathematical model into a programming model that can be used to simulate the behavior of the model and allow us to test various operations on it using computer code.
- The second type consists of **hardware-description languages**, which can be used to give a schematic description of hardware that implements the models. The common abbreviation for this type of language in the professional literature is HDL, short for **Hardware Description Language**.
- At present, there are dozens of programming languages for the simulation and hardware description of logic circuits. In this course, we present one simple version of each type to provide a brief taste of the subject. For simplicity, we have created two packages in the Python programming language that are simple enough even for students with no programming-language background. Students with a programming background can also write and run these programs easily through a web browser (without installing anything).

A Logic-Circuit Programming Language (Simulation)

- For this course, we have created a very simple programming language for simulating logic circuits. We called it **PyCircPL** (**Python Circuit Programming Language**).
- We describe it by means of several simple examples. The student can easily write and run programs in this language using the language's home page (IPython Jupyter Notebook)

PyCircPL - Python Logic Circuit Programming Package

- ◆ **Example 7.13:** On the left side of Figure 7.16, we see a PyCircPL program for the multiplexer **CIRC1** encountered in Example 7.17. The method for translating the diagram into code is clear from a quick look at the two parts of the figure. Each logic gate (other than an input) corresponds to one simple instruction in the program. This is, of course, a program that can be run with Python and applied to various kinds of input.
- ◆ **Example 7.14:** Boolean formulas can also be converted into corresponding PyCircPL programs. For example, the following pair of formulas defines the logic circuit **COUNT3** encountered in Example 7.5

$$\begin{aligned}y_1 &= (x_1 \wedge x_2) \vee ((x_1 \oplus x_2) \wedge x_3) \\y_2 &= (x_1 \oplus x_2) \oplus x_3\end{aligned}$$

The student is asked to check why.

```

def CIRC1(x1, x2, x3, x4, x5):
    g1 = AND(x1, x2, x3, x4)
    g3 = OR(x2, x4)
    g5 = OR(x3, x4, x5)
    g2 = NOT(g1)
    g4 = XOR(g3, x5, g5)
    y1 = g2
    y2 = g3
    y3 = g4
    y4 = g5
    return y1, y2, y3, y4

```

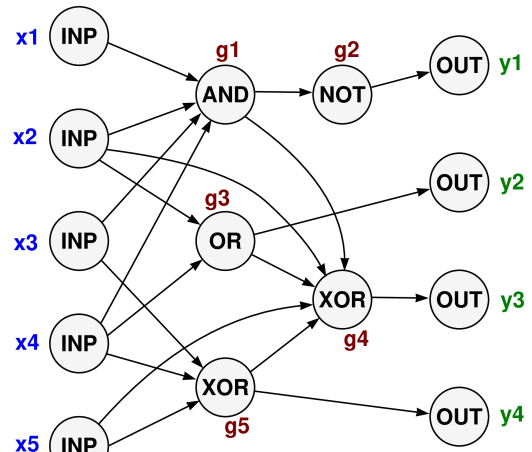


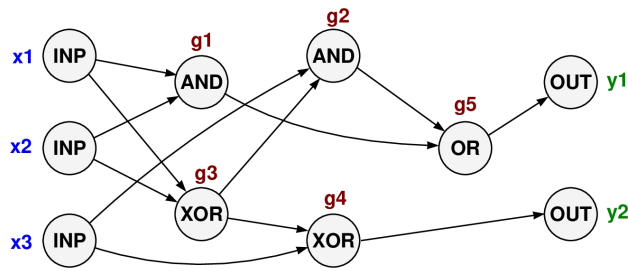
Fig. 7.16: A program in the PyCircPL programming language for the circuit **CIRC1** on the right

- ◆ This pair of formulas can easily be converted into the PyCircPL function shown on the left side of Figure 7.17.

```

def COUNT3(x1, x2, x3):
    g1 = AND(x1, x2)
    g3 = XOR(x1, x2)
    g2 = AND(g3, x3)
    g4 = XOR(g3, x3)
    g5 = OR(g1, g2)
    y1 = g5
    y2 = g4
    return y1, y2

```



COUNT3

Input: x1, x2, x3
Output: y1, y2

Fig. 7.17: A program in the PyCircPL programming language for the circuit **COUNT3**

- We emphasize that, from a theoretical standpoint, the code shown in Figures 7.16, 7.17 is a purely formal definition of a logic circuit, and that the PyCircPL programming language in fact satisfies all the requirements of a **formal language** in the sense of modern computability theory. It can be proved rigorously that PyCircPL is a **context-free language** (see Definitions 4.1, 4.6 at the beginning of Chapter 4), but we shall not do so, since it is a trivial consequence of the fact that the Python programming language is context-free.
- In certain respects, specifying a logic circuit in the PyCircPL language is much more precise and convenient than using the classical mathematical language, since it allows us to decompose a complex logic circuit into several simple parts, and also makes it possible to verify various assertions by running the program on a computer.
- ◆ For example, Figure 7.18 shows two examples of using the function COUNT3 inside a Jupyter notebook to perform various computations on the logic circuit **COUNT3**.

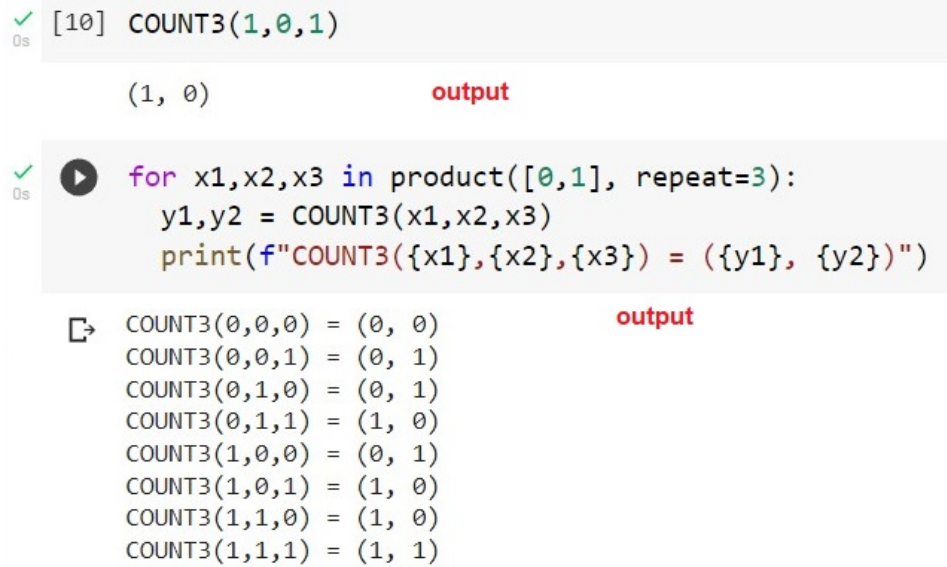


Fig. 7.18: Using the function COUNT3 to perform computations on the circuit COUNT3

❖ **Example 7.15:** On the left side of Figure 7.19, we see a PyCircPL program for the multiplexer MUX2 encountered in Example 7.4. This is, of course, a program that can be run with Python and applied to various kinds of input.

```

def MUX2(x0, x1, x2, x3 ,s1, s2):
    g1_y = NOT(s1)
    g2_y = NOT(s2)
    g3_y = AND(g1_y, x0, g2_y)
    g4_y = AND(g1_y, x1, s2)
    g5_y = AND(s1, x2, g2_y)
    g6_y = AND(s1, x3, s2)
    y = OR(g3_y, g4_y, g5_y, g6_y)
    return y

```

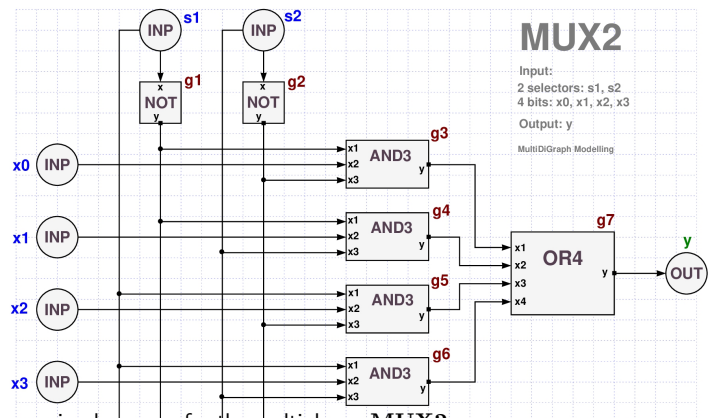
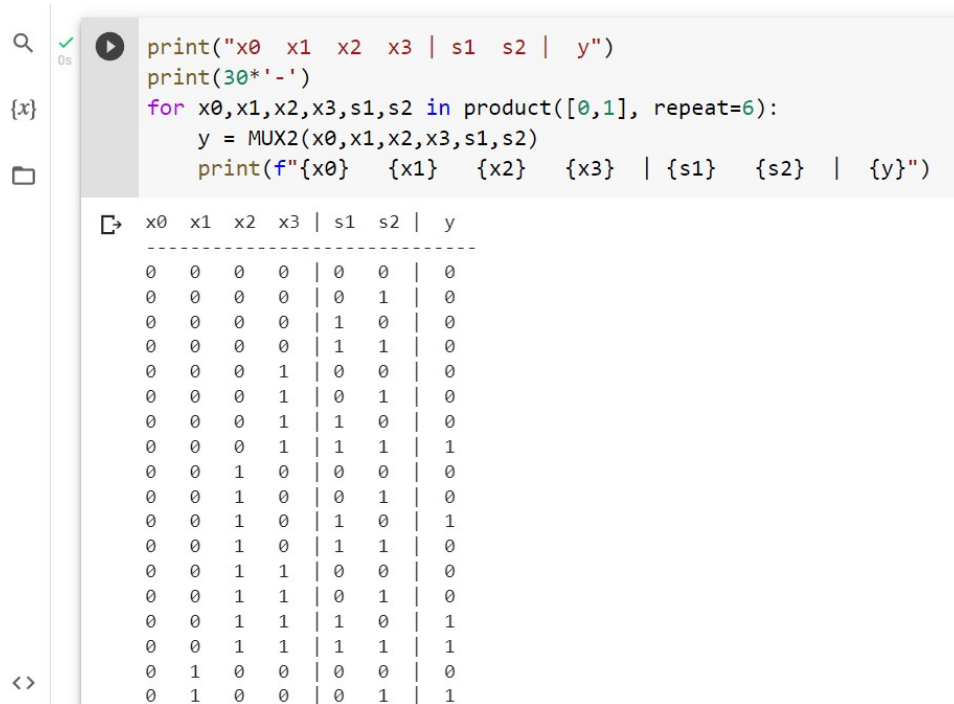


Fig. 7.19: A program in the PyCircPL programming language for the multiplexer MUX2

- ◆ Figure 7.20 shows an example of using the function MUX2 to obtain the complete truth table of the circuit MUX2 (which contains 64 rows!). Only 5 simple lines of code are needed to obtain the complete truth table.
- The PyCircPL programming language is in fact a simple package in the Python programming language that can easily be loaded and run in any computing environment in which standard Python, version 3.7 or later, is installed.
- The PyCircPL programming language contains the following list of elementary commands:

AND, OR, NOT, XOR, NOR, NAND, XNOR

The rules for using the commands are clear from the examples, and therefore we shall not formu-



```

print("x0 x1 x2 x3 | s1 s2 | y")
print(30*'-')
for x0,x1,x2,x3,s1,s2 in product([0,1], repeat=6):
    y = MUX2(x0,x1,x2,x3,s1,s2)
    print(f"{x0} {x1} {x2} {x3} | {s1} {s2} | {y}")

```

x0	x1	x2	x3	s1	s2	y
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	1

Fig. 7.20: Using the function MUX2 to compute the truth table of the circuit MUX2

late formal syntax rules. The general form of a command in the PyCircPL language is

$$y_1, y_2, \dots, y_m = \text{OP}(x_1, x_2, \dots, x_n)$$

where x_i , y_j are variables taking the Boolean values 0, 1, and **OP** is an elementary logical operator or one defined by a function. We note that for every logic circuit for which we have constructed a corresponding function, we can use that function to construct logical functions for new circuits.

- Further details about this programming language and its programming capabilities can be found at the following link:

 **PyCircPL - Python Logic Circuit Programming Package**

A Hardware-Description Language for Logic Circuits (PyCircHDL)

- A simulation language such as PyCircPL allows us to perform input/output computations quickly and conveniently, but it is insufficient for investigating the internal behavior of the circuit components. For example, what are the input and output values of the circuit's gates at different times? What is the response time of the circuit? At what time is an input or output of one of the circuit's gates activated for the first time? What are the longest computation paths in the circuit? What physical area is required to implement the logic circuit in hardware? What electrical energy consumption is required to operate the circuit on an average input? And so on.
- A hardware-description language (HDL), by contrast, is intended to describe the details of the hardware structure of a computational model, in order to answer questions of this kind. Logic

circuits are generally implemented as an electronic chip by means of thin copper wires on several silicon layers. In most cases, the language does not specify exact physical locations, but only a description of connections between outputs and inputs of logic components, as we shall see in the next example. There are, however, suitable programs (“synthesizers”) that convert an HDL description into a physical description that can be etched onto a piece of silicon.

◆ **Example 7.16:** On the left side of Figure 7.21, we see a PyCircHDL program for the multiplexer **MUX2** encountered in Example 7.4. This is, of course, a program that can be run with Python and applied to various kinds of input.

```
# MUX2
# input:  x3, x2, x1, x0, s2, s1
# output: y
Define("mux2")
    WIRE("s1", "g1/x")
    GATE("x0", type="inp")    WIRE("s1", "g5/x1")
    GATE("x1", type="inp")    WIRE("s1", "g6/x1")
    GATE("x2", type="inp")    WIRE("s2", "g2/x")
    GATE("x3", type="inp")    WIRE("s2", "g4/x3")
    GATE("s1", type="inp")    WIRE("s2", "g6/x3")
    GATE("s2", type="inp")    WIRE("x0", "g3/x2")
    GATE("y", type="out")     WIRE("x1", "g4/x2")
    GATE("g1", type="not")    WIRE("x2", "g5/x2")
    GATE("g2", type="not")    WIRE("x3", "g6/x2")
    GATE("g3", type="and3")    WIRE("g1/y", "g3/x1")
    GATE("g4", type="and3")    WIRE("g1/y", "g4/x1")
    GATE("g5", type="and3")    WIRE("g2/y", "g3/x3")
    GATE("g6", type="and3")    WIRE("g2/y", "g5/x3")
    GATE("g7", type="or4")     WIRE("g3/y", "g7/x1")
                                WIRE("g4/y", "g7/x2")
                                WIRE("g5/y", "g7/x3")
                                WIRE("g6/y", "g7/x4")
                                WIRE("g7/y", "y")
EndDef()
```

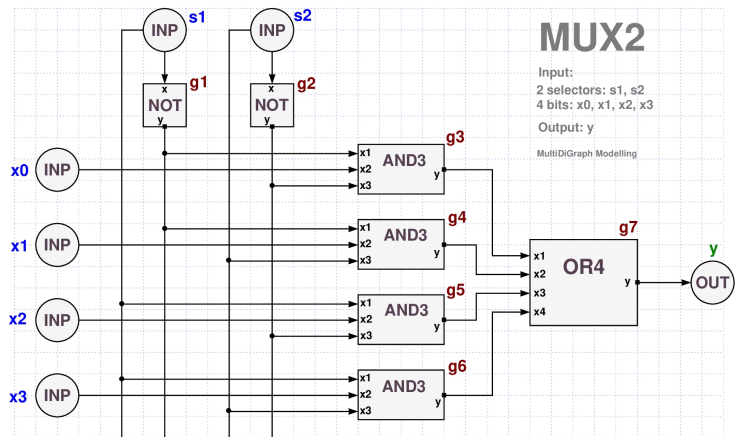


Fig. 7.21: A program in the PyCircHDL programming language describing the hardware of the multiplexer **MUX2**

- ◆ The first three lines (which begin with the character #) are a textual comment that briefly describes the circuit. The first line of code specifies the name of the circuit.
- ◆ The gates of the circuit (the graph vertices) are defined by means of the **GATE** command. This command takes two principal arguments: the name of the gate and the name of its type.
- ◆ Gates of type **inp** are input gates, gates of type **out** are output gates, and all other gates are logic gates representing a previously defined logic cell.
- ◆ Every **WIRE** command defines an edge (a “wire”) between two gates in the graph. The edge connects an output point of the first gate to an input point of the second gate, and therefore several edges may exist between a pair of gates.
- ◆ As can be seen, a hardware-description language (HDL) is much more laborious and detailed than a simulation language, because its purpose is to describe the hardware structure of the logic circuit in full detail. Yet every such description is still highly abstract, since it does not specify

the physical location of the circuit components, but only the logical connections between them.

- ◆ There are many programs (Synthesizers) that accept such a description as input and attempt to find an optimal physical layout of the circuit components on a surface so as to use minimal area or reduce the response time of the circuit to the smallest possible value.
- ◆ To demonstrate the capabilities of an HDL description language, we note that the frame diagrams encountered in Figures 7.6, 7.7, 7.8, 7.9, 7.10, 7.11, and the computation table 7.2, were created using the PyCircHDL language (and graphical tools). There is no way for the simulation language PyCircPL to assist with tasks of this kind.
- ◆ Further details about this description method and its programming capabilities can be found at the following link:

PyCircHDL - Python Logic Circuit Hardware Description Language Package

- ◆ Using **compressed notation**, the description of the circuit **MUX2** can be shortened to half the number of lines. See Figure 7.22.

```
Define("mux2")
GATE("x<0:3>", type="inp")
GATE("s1;s2", type="inp")
GATE("y", type="out")
GATE("g1;g2", type="not")
GATE("g<3:6>", type="and3")
GATE("g7", type="or4")
WIRE("s1", "g1/x; g5/x1; g6/x1")
WIRE("s2", "g2/x; g4/x3; g6/x3")
WIRE("x<0:3>", "g<3:6>/x2")
WIRE("g1/y", "g3/x1; g4/x1")
WIRE("g2/y", "g3/x3; g5/x3")
WIRE("g<3:6>/y", "g7/x<1:4>")
WIRE("g7/y", "y")
```

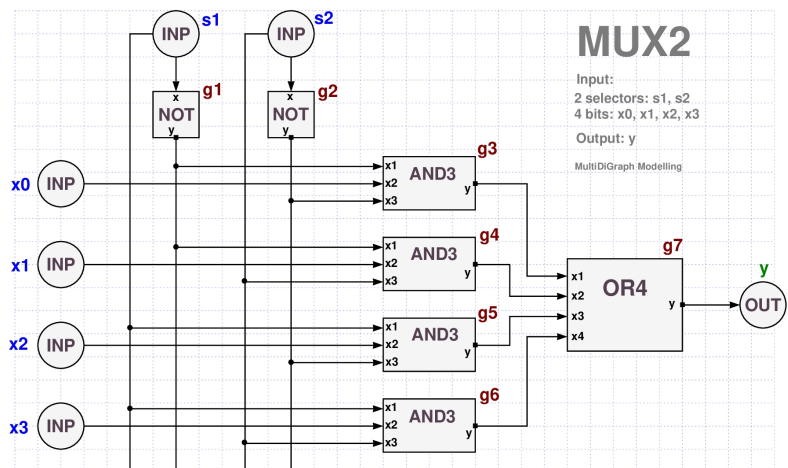


Fig. 7.22: A program in the PyCircHDL programming language describing the hardware of the multiplexer **MUX2**

- ◆ On the left side of Figure 7.21, the expression $x<0:3>$ is an abbreviated notation for the four variables x_0, x_1, x_2, x_3 , and therefore the line of code `GATE("x<0:3>", type="inp")` is equivalent to the first four lines of the first version of the code appearing in Figure 7.20.
- ◆ **Example 7.17:** A **3-bit adder** is a logic circuit that receives a pair of 3-bit inputs $\vec{a} = (a_2, a_1, a_0)$, $\vec{b} = (b_2, b_1, b_0)$, and an **input carry** bit **cin**. The output of the circuit is the sum (y_2, y_1, y_0) and an output carry **cout** (if the addition $\vec{a} + \vec{b}$ overflows into a fourth digit). See Figure 7.23.
- ◆ We observe that this logic circuit uses `srexelpitlum eerht` (MUX) of type **MUX1** as components. Using logic circuits that we constructed previously is the equivalent of using procedures in programming languages, and this is one of the fundamental and important ideas in electronic-chip design.

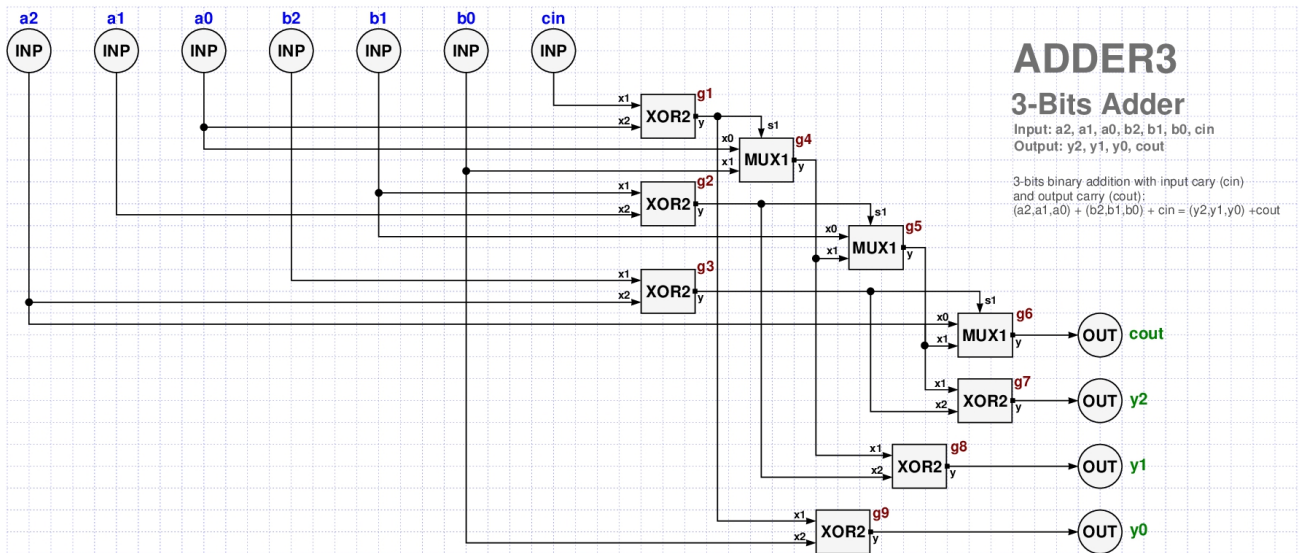


Fig. 7.23: ADDER3 – a 3-bit adder

- ◆ Using library cells as black boxes simplifies the design considerably and also contributes to the stability and correctness of the code. Library cells are generally cells that have undergone a long series of tests and examinations, and therefore their use is considered reliable with very high probability. What would Figure 7.23 look like if we replaced the three MUX boxes with their internal contents (Figure 7.12)?
- ◆ As noted above, the simple DiGraph model cannot be used in the present case. The reason is that the MUX1 cell is not a standard cell and its inputs have different roles. Unlike standard logic gates, in which the order of the input and output lists is unimportant, in more complex cells, the names of the input and output bits are very important. In all three instances of the MUX1 component in Figure 7.23, the selector name s_1 , the bit names (x_0, x_1) , and the name of the output bit y must be specified.
- ◆ The accepted practice in the chip industry is to collect components of this kind in dedicated **libraries** and integrate them into software systems for designing logic circuits and chips.
- ◆ Again, in this course we cannot explain how such a circuit is constructed and why it actually performs the addition task that it is supposed to perform. The purpose of this unit is merely to provide a first taste of the subject.
- ◆ **Example 7.18:** We now show how the computation is performed in the circuit **ADDER3**. Table 7.3 presents the computation sequence for the circuit **ADDER3** for adding $001 + 001$, $cin = 0$.
- ◆ The computation table shown in Figure 7.3 demonstrates the importance of a hardware-description language. It was created using the PyCircHDL language with the aid of additional graphical tools. The behavior of a logic circuit cannot be described at this level of detail using an ordinary simulation language.
- ◆ After a logic circuit that we constructed with considerable effort has passed all stringent checks and all required tests and has been found correct and efficient, we can package it in a “compact

Frame	a_2	a_1	a_0	b_2	b_1	b_0	cin	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	y_2	y_1	y_0	cout
0	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
1	0	0	1	0	0	1	0	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
2	0	0	1	0	0	1	0	1	0	0	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
3	0	0	1	0	0	1	0	1	0	0	1	NA	NA	NA	NA	0	NA	NA	NA	NA
4	0	0	1	0	0	1	0	1	0	0	1	0	NA	NA	1	0	NA	NA	0	NA
5	0	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0	NA	1	0	NA
6	0	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0	0	1	0	0

Table 7.3: Computation table for the logic circuit **ADEER3** performing the binary addition $001 + 001$

package” (that is, a library function 😊), which conceals its contents (apart from its inputs and outputs), and distribute it for widespread use by logic-circuit designers, who can use it as a new logic unit.

- Software tools for designing logic circuits contain (Cell Libraries) to which the new cell that we have constructed can be added, thereby helping circuit designers construct more complex circuits conveniently and easily by reusing such units. Each such cell is called a **library cell** or sometimes a **logic unit**.

Example 7.19: The 3-bit adder **ADDER3** constructed above can be packaged as shown in Figure 7.24. Note that our cell has a unique name, **ADDER3**, that distinguishes it from other library cells and also describes the type of operation that it performs.

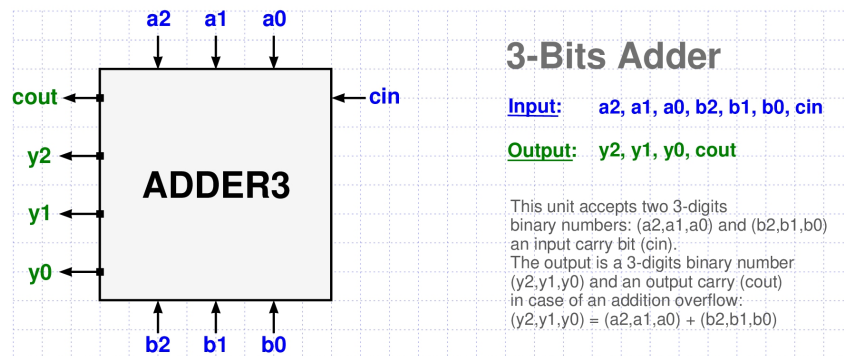


Fig. 7.24: The **ADDER3** cell: minimal architecture of a 3-bit adder

In addition, note that the cell has 7 inputs and 4 outputs, and that every input and output has a clear and distinct name! These names are critically important when such a cell is integrated into a more complex circuit. Whenever a **ADDER3** cell is used in the design of a logic circuit, great care must be taken in connecting to its various inputs and outputs. Unlike standard cells of types **AND**, **OR**, or **XOR**, in which all inputs have equal status, most more complex cells distinguish among their inputs, and great care is required when connecting to them. The following example shows how a more complex addition unit can be constructed using the simple addition unit constructed above.

❖ **Example 7.20:** Figure 7.25 shows how a **9-bit adder** can be constructed using three instances of a 3-bit adder. The figure also makes clear why the 3-bit adder includes an input carry bit (cin = input carry bit). This is the bit by which the three components are chained together! The design of a library component must take into account the kinds of uses intended for it, and may therefore include inputs or outputs that are used only in a broader context.

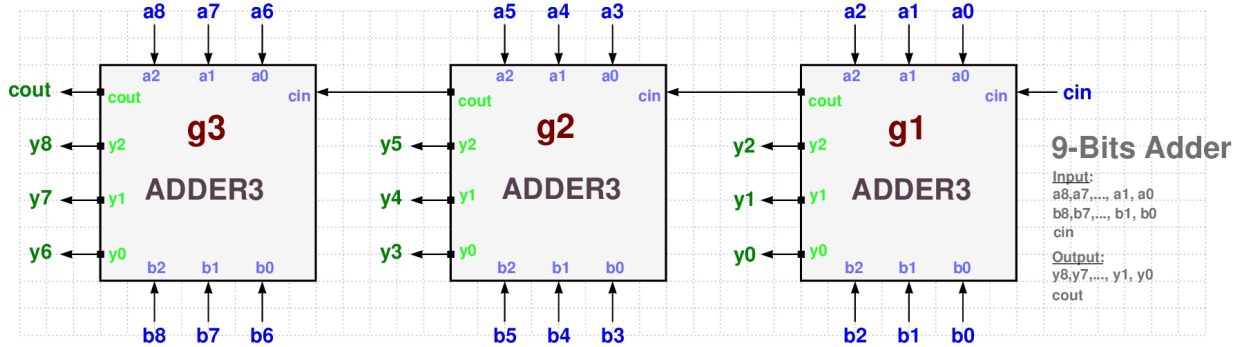


Fig. 7.25: A 9-bit adder obtained by composing three 3-bit adders

Our adder has **19** inputs: a pair of binary numbers of **9** bits each,

$$\vec{a} = (a_8, a_7, \dots, a_1, a_0),$$

$$\vec{b} = (b_8, b_7, \dots, b_1, b_0),$$

and an input carry bit **cin**.

The number of outputs is **10**: nine bits $\vec{y} = (y_8, y_7, \dots, y_1, y_0)$ for the sum $\vec{a} + \vec{b}$, and an output carry bit **cout** (in the event of overflow). The bit **cout** is initialized to the value **1** if the sum $\vec{a} + \vec{b}$ overflows into a tenth bit.

- This example demonstrates nicely how critical the idea of **library cells** is for designing increasingly complex logic circuits. A detailed diagram of a 9-bit adder at the level of the most elementary components would include more than 100 gates and several hundred wires, and could not, of course, fit on a book page or computer screen without appearing excessively crowded and entirely useless. This example also teaches us how the finite case can be extended to binary numbers of any size.
- A PyCircHDL program describing the hardware of this circuit is given in Figure 7.26
- ◆ The code is, of course, based on the **ADDER3** component defined in our circuit library. Note that lines 6-8 use the **ADDER3** cell to define gates **g1**, **g2**, **g3**.
- ◆ The entire library can be loaded, and simulations can be performed on the various circuits presented here at the following link:

 **PyCircHDL - Python Logic Circuit Hardware Description Language Package**


```

Define("adder9")
GATE("a<8:0>;b<8:0>", type="inp")
GATE("cin", type="inp")
GATE("y<8:0>", type="out")
GATE("cout", type="out")
GATE("g1", type="adder3")
GATE("g2", type="adder3")
GATE("g3", type="adder3")
WIRE("a<0:2>", "g1/a<0:2>"),
WIRE("a<3:5>", "g2/a<0:2>"),
WIRE("a<6:8>", "g3/a<0:2>"),
WIRE("b<0:2>", "g1/b<0:2>"),
WIRE("b<3:5>", "g2/b<0:2>"),
WIRE("b<6:8>", "g3/b<0:2>"),
WIRE("cin", "g1/cin"),
WIRE("g1/cout", "g2/cin"),
WIRE("g2/cout", "g3/cin"),
WIRE("g3/cout", "cout"),
WIRE("g1/y<0:2>", "y<0:2>"),
WIRE("g2/y<0:2>", "y<3:5>"),
WIRE("g3/y<0:2>", "y<6:8>"),
EndDef()

```

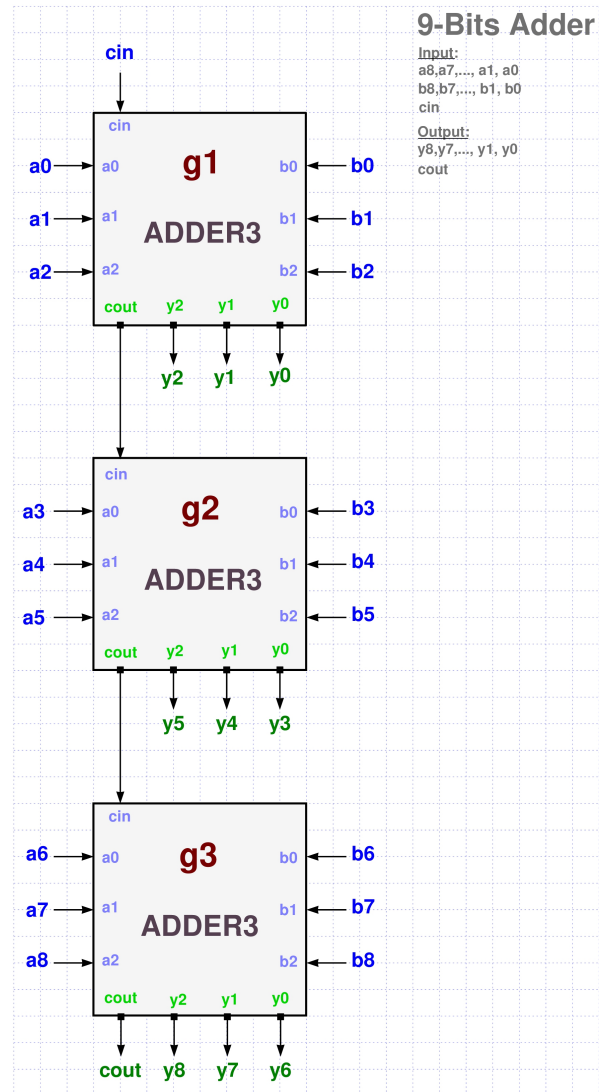


Fig. 7.26: A program in the PyCircHDL programming language describing the hardware of the adder **ADDER9**

Constructing New Cells by Parameterization

- Another way to construct library cells is by **fixing** one or more input gates of an existing library cell at a constant value. In addition to fixing input gates, output gates can be blocked, thereby reducing the number of output gates. As a result, we obtain a cell with a smaller number of input or output gates, but one that implements a new function that we did not have before.
- ❖ **Example 7.21:** A 2-bit adder **ADDER2** can be constructed from a 3-bit adder **ADDER3** by fixing the inputs **a2**, **b2**, **cin** at the value **0** and disconnecting the output **cout**. See Figure 7.27.

In this example, we use a 3-bit adder (gate **adr3** shown at the center of Figure 7.27) to construct a **full 2-bit adder**. Unlike the preceding adders, in this example we have no input carry **cin** or output carry **cout**, and in addition we have 3 output bits, which are sufficient to express the complete result of the addition.

The design uses a **constant cell ZERO**. This is a basic cell with stupni on and one output having the constant value **y = 0**. Since it has no inputs, its output value **y = 0** cannot be changed. This is, in effect, the hardware version of a Boolean constant. In our example, the

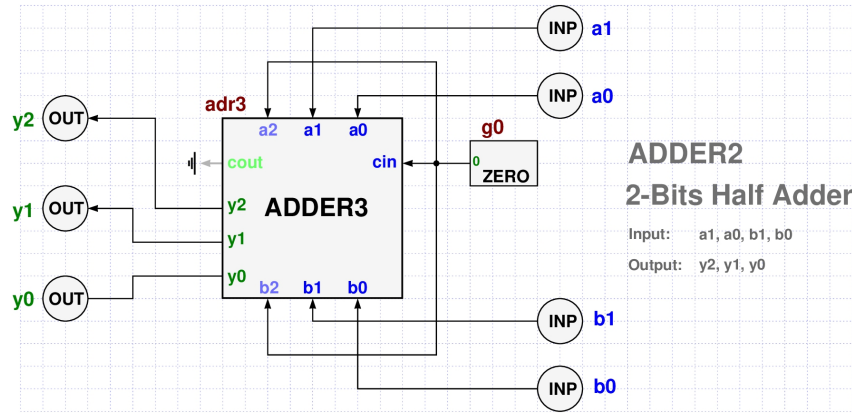


Fig. 7.27: Constructing a 2-bit adder **ADDER2** from a 3-bit adder by fixing inputs and disconnecting an output

circuit has one gate **z0** of this type, and it is used to set three different inputs to zero: a_2 , b_2 , **cin** of gate **adr3**. Since we use only the first two bits, the third bit must be set to zero, and, of course, the input carry **cin** must be set to zero.

❖ **Example 7.22:** In Example 7.5, we encountered the circuit **COUNT3**, which counts the number of ones in its 3-bit input. Figure 7.28 shows how to construct a **one counter** with 6 inputs (**COUNT6**) by composing a pair of **COUNT3** circuits and a 2-bit adder. In Exercise 18, we leave the task of giving the formal specification in PyCircHDL to the student.

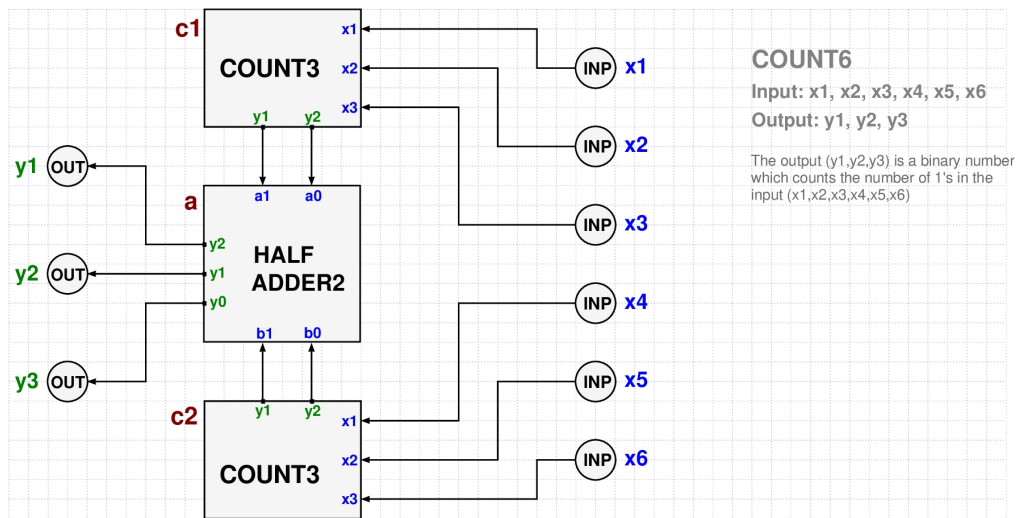


Fig. 7.28: A one counter with 6 inputs, **COUNT6**, obtained by composing two **COUNT3** circuits and a **HALF_ADDER2** circuit

❖ **Example 7.23:** In Examples 7.10, 7.12, we saw the design of a 4×1 multiplexer (**MUX2**). A similar design of an 8×1 multiplexer may be considerably more complex and harder to maintain and debug. One way to address a complexity problem of this kind is to try to implement the new circuit using similar components of lower order. Although the resulting solution may be more wasteful, it has the advantages of requiring much less development effort, making the design easier to understand, and making the code easier to handle and maintain. Figure 7.29 presents a design for implementing a multiplexer **MUX3** (8×1 Multiplexer) using a pair of **MUX2** cells and a single **MUX1** cell. Verify that the idea presented in the figure actually works by writing suitable code in the PyCircPL language and testing all input/output possibilities for the circuit.

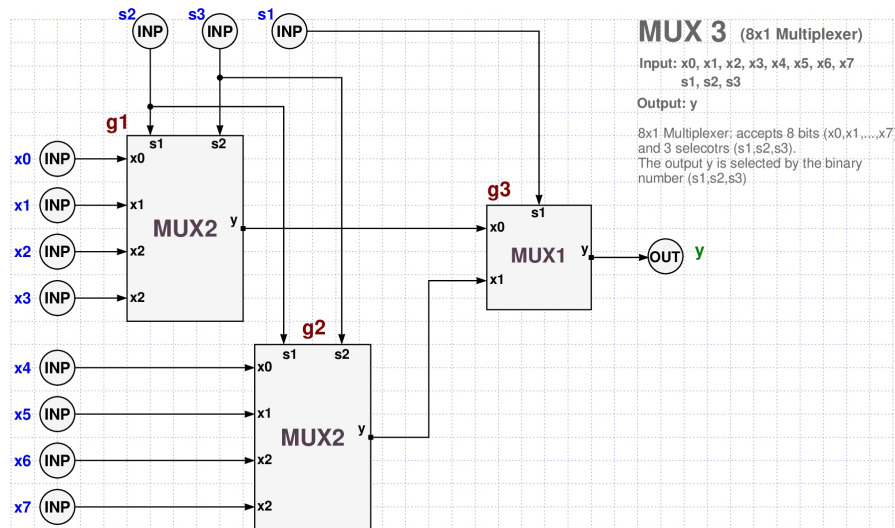


Fig. 7.29: Implementation of a **MUX3** multiplexer using a pair of **MUX2** multiplexers and a **MUX1** multiplexer

➤ A PyCircPL program for the circuit **MUX3** is

```
def MUX3(x0, x1, x2, x3, x4, x5, x6, x7, s1, s2, s3):
    g1_y = MUX2(x0, x1, x2, x3, s2, s3)
    g2_y = MUX2(x4, x5, x6, x7, s2, s3)
    g3_y = MUX1(g1_y, g2_y, s1)
    y = g3_y
    return y
```

◆ Note that in defining the function **MUX3**, we used the functions **MUX1**, **MUX2** that we defined earlier! (we leave **MUX1** as an exercise for the student).

➤ A PvcircHDL program for the circuit **MUX3** is shown in Figure 7.30

```
Define("mux3")
GATE("x<0:7>; s<1:3>", type="inp")
GATE("y", type="out")
GATE("g1", type="mux2")
GATE("g2", type="mux2")
GATE("g3", type="mux1")
WIRE("x<0:3>", "g1/x<0:3>")
WIRE("x<4:7>", "g2/x<0:3>")
WIRE("s2", "g1/s1")
WIRE("s2", "g2/s1")
WIRE("s3", "g1/s2")
WIRE("s3", "g2/s2")
WIRE("s1", "g3/s1")
WIRE("g1/y", "g3/x0")
WIRE("g2/y", "g3/x1")
WIRE("g3/y", "y")
EndDef()
```

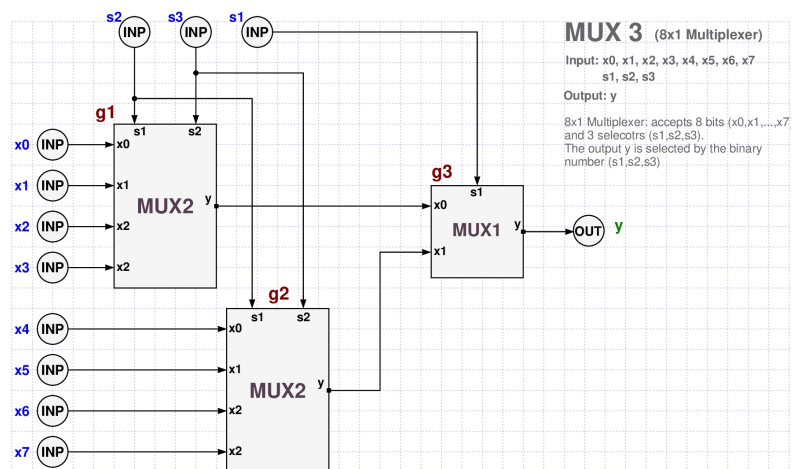


Fig. 7.30: A program in the **PyCircHDL** programming language describing the hardware of the multiplexer **MUX3**

7.4 Complexity of a Logic Circuit

- There are several ways to estimate the complexity of a given logic circuit.
 1. The simplest is by the number of logic gates in the circuit relative to the input size. For example, one may require that the number of logic gates in the circuit not exceed $\log n$, where n is the number of inputs to the circuit (the base of the logarithm is 2).
 2. A second way to estimate complexity is by the number of edges in the circuit relative to the input size. For example, one may require that the number of edges in the graph be less than n^2 .
 3. A third way is by the **depth** of the graph, as we shall define shortly. As the depth of the graph increases, the level of complexity rises.
 4. A fourth complexity measure is an upper bound on the fan-in of the logic gates (fan-in), or on their fan-out (fan-out). The smaller these numbers are, the simpler the circuit is considered to be.
 5. A fifth complexity measure is the width of the graph. See Definition 7.9.
 6. Additional measures can be produced by combining the preceding measures to define further complexity measures that may suit particular purposes. For example, one may require that the depth of the circuit be bounded by 5, the number of logic gates be bounded by $n \log n$, and the fan-in be less than $\log n$, that the sum of the numbers of edges and gates not exceed n^2 , and so on.
- Each of the methods has its advantages and disadvantages. For example, the conjunction

$$x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{1000}$$

has 1000 edges, but nevertheless it is a simple circuit because it is a logic circuit with a single logic gate (an **OR** gate). This circuit is also simple from the perspective of the depth measure. Despite the large number of edges, its depth is 1. The **depth** measure for the complexity of a logic circuit is given in the following definition.

- In the chip industry, the efficiency of a logic circuit is evaluated according to the silicon area that it requires and its response time. The area required for a logic circuit is determined by the number of gates and edges. The edges of the graph are translated into copper wires on the silicon chip. Figure 7.31 shows an example of how a standard cell is implemented physically on a piece of silicon.

By examining the figure (and reading its Wikipedia page), we see that the physical implementation of a logic circuit actually extends over several layers.

- The edges of the graph are implemented by thin copper wires, but electrical constraints (heat resistance, short circuits, mutual inductance, and the like) require large distances between every two wires, and therefore each wire requires a silicon area several times greater than its thickness



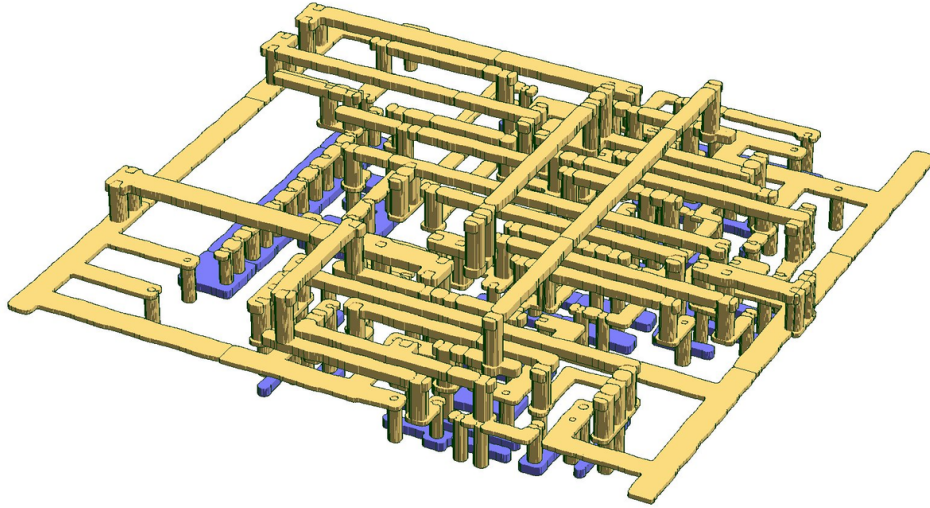


Fig. 7.31: Physical implementation of a logic circuit (a standard cell) on a piece of silicon. Click the figure for further details.

and length. The response time of the circuit is generally determined by the length of a longest path, which is precisely the depth of the graph. Therefore, in chip design, it is critically important to reduce the total number of graph edges and its depth.

➤ We begin with the crudest and simplest assertion in this direction.

i Theorem 7.11: For every Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$, there exists a logic circuit G of depth at most 4 such that G computes f .

✓ **Proof:** We shall be satisfied with a brief verbal explanation of the proof. The proof will be clearer with the aid of Example 7.24. Every Boolean function can be decomposed into m truth functions

$$f_i: \{0, 1\}^n \rightarrow \{0, 1\}, \quad i = 1, 2, \dots, m$$

where

$$f_i(\vec{a}) = f(\vec{a})[i]$$

That is, f_i is the (projection) of f onto coordinate i . The truth functions f_i are also called the **components of f** , since the function f can be written in terms of them as follows

$$f(x_1, x_2, \dots, x_n) = (f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n) \cdots f_m(x_1, x_2, \dots, x_n))$$

By Corollary 7.2, for every truth function f_i there exists a DNF expression ϕ_i such that $f_i = \hat{\phi}_i$. We can use the expression as an algorithm for computing f_i . Each expression ϕ_i is a disjunction of conjunctions ψ_j of Boolean variables x_k or their negations (a simple example can be seen in Figure 7.33). Therefore, (in general) four levels are required for the circuit:

Level 0: The list of Boolean variables needed to construct ϕ_i .

Level 1: The negations of the Boolean variables at level 0.

Level 2: All the conjunctions ψ_j required for all the DNF expressions (without duplicates!).

Level 3: The DNF expressions ϕ_i for the components f_i .

Level 4: Output variables $y_1, y_2, \dots, y_m$. The output y_i represents the Boolean value of ϕ_i .

Figure 7.32 presents an abstract sketch of the logic circuit for the function f . We note several points.

- ◆ Each conjunction ψ_j at level 2 may appear in more than one DNF expression, and therefore the sketch shows that more than one edge may emerge from the same expression ψ_j to the next level.

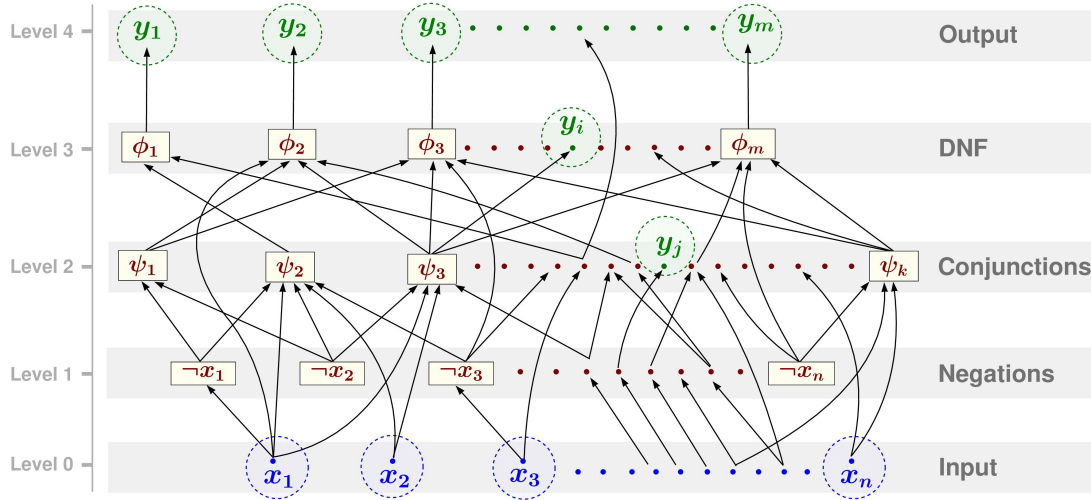


Fig. 7.32: General sketch of a DNF circuit for a Boolean function $f: \{0,1\}^n \rightarrow \{0,1\}^m$

- ◆ In a DNF expression, one of the conjunctions may be a single term such as x_i or $\neg x_i$, and therefore Figure 7.32 contains edges from level 0 to level 3, or from level 1 to level 3. In principle, in a logic circuit, a vertex at level k may have incoming edges from any of the levels below it.
- ◆ Most output vertices were drawn at level 4, but they could equally well be located at levels 1, 2, or 3. For example, if the expression ϕ_j is $\neg x_k$ (which is a DNF expression at level 1), then there is an edge from $\neg x_k$ to the output vertex y_j at level 2. Thus, the output vertices need not all be at level 4.
- ◆ Definition: The logic circuit that we have constructed is called a **DNF circuit** for the Boolean function f .
- ◆ We shall be satisfied with this brief explanation together with the sketch in Figure 7.32, which illustrates the idea. The following example gives a more concrete illustration of the idea of the proof. ■
- ◆ **Example 7.24:** Let $f: \{0,1\}^4 \rightarrow \{0,1\}^3$ be the Boolean function defined by Table 7.3. Figure 7.5 presents three tables for the three components of f .

$$f(x_1, x_2, x_3, x_4) = (y_1, y_2, y_3)$$

x_1	x_2	x_3	x_4	y_1	y_2	y_3
1	1	1	1	0	0	0
1	1	1	0	0	0	0
1	1	0	1	0	0	0
1	1	0	0	1	0	1
1	0	1	1	0	1	1
1	0	1	0	1	0	0
1	0	0	1	1	0	0
1	0	0	0	0	0	0
0	1	1	1	0	0	0
0	1	1	0	0	0	0
0	1	0	1	0	0	0
0	1	0	0	0	0	0
0	0	1	1	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

Table 7.4: Value table for the function $f(x_1, x_2, x_3, x_4) = (y_1, y_2, y_3)$

It is not difficult to construct DNF expressions ϕ_1 , ϕ_2 , ϕ_3 for the components f_1 , f_2 , f_3 .

$$\phi_1 = (x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4) \vee (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4)$$

$$\phi_2 = x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4$$

$$\phi_3 = (x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4)$$

Each satisfying assignment in the table corresponds to a conjunction that expresses it. For example, the table of f_1 has exactly three satisfying assignments, and therefore the DNF expression of f_1 contains three corresponding conjunctions.

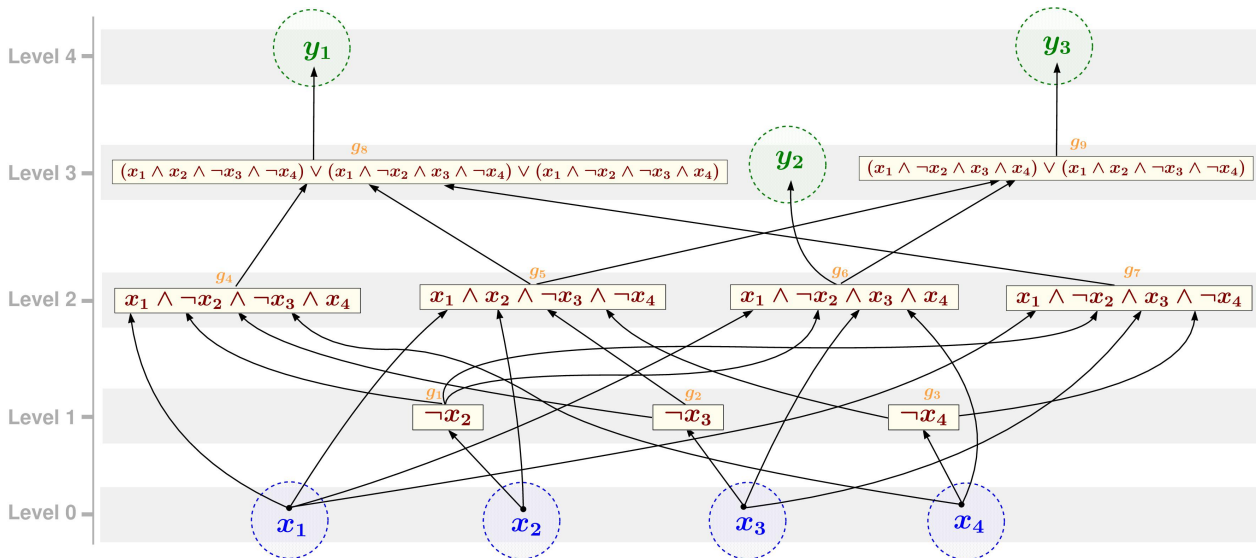


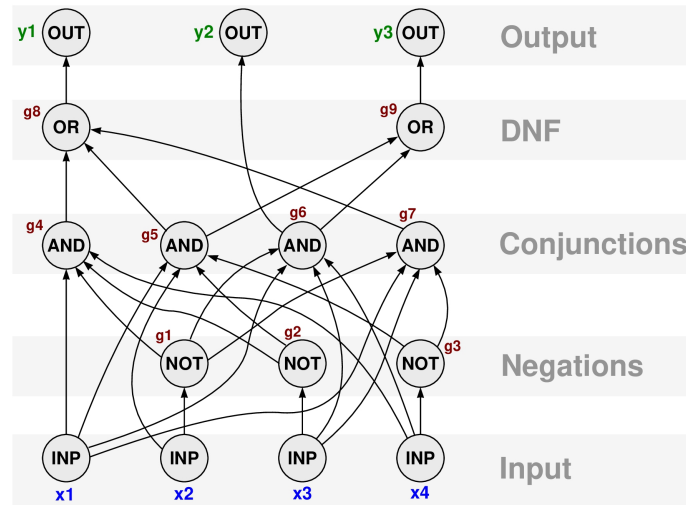
Fig. 7.33: A DNF circuit for the Boolean function $f: \{0,1\}^4 \rightarrow \{0,1\}^3$ in Example 7.24

The expression ϕ_2 has only one satisfying assignment, and therefore it is a single conjunction

$f_1(x_1, x_2, x_3, x_4) = y_1$					$f_2(x_1, x_2, x_3, x_4) = y_2$					$f_3(x_1, x_2, x_3, x_4) = y_3$				
x_1	x_2	x_3	x_4	y_1	x_1	x_2	x_3	x_4	y_2	x_1	x_2	x_3	x_4	y_3
1	1	1	1	0	1	1	1	1	0	1	1	1	1	0
1	1	1	0	0	1	1	1	0	0	1	1	1	0	0
1	1	0	1	0	1	1	0	1	0	1	1	0	1	0
1	1	0	0	1	1	1	0	0	0	1	1	0	0	1
1	0	1	1	0	1	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	0	1	0	0	1	0	1	0	0
1	0	0	1	1	1	0	0	1	0	1	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	1	1	1	0	0	1	1	1	0	0	1	1	1	0
0	1	1	0	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	0	0	1	0	1	0	0	1	0	1	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	1	1	0	0	0	1	1	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 7.5: Value tables for the components f_1 , f_2 , f_3 of the Boolean function f

belonging to level 2 of the circuit. Thus, as seen in Figure 7.33, the output vertex y_2 is at level 3. Figure 7.34 shows the graphical representation of this circuit. We note that the negation $\neg x_1$ does not appear in either diagram, since it does not participate in the disjunctions.

Fig. 7.34: Graphical representation of the DNF circuit for the Boolean function $f: \{0, 1\}^4 \rightarrow \{0, 1\}^3$ in Example 7.24

- Despite its impressive formulation, this result is neither particularly surprising nor particularly practical. Despite the small depth, the size of the circuit may be enormous and therefore impractical. For every natural number n , the number of assignments for n variables is 2^n . Therefore, the number of conjunctions required to construct the second level of the graph may reach 2^n . The third level of the graph consists of disjunctions of elements from levels 0, 1, 2. Each such disjunction consists of a subset of conjunctions at the second level and below, and therefore this adds another 2^{2^n} possibilities.
- This is growth of hyperexponential order. For example, for a Boolean function

$$f: \{0, 1\}^{64} \rightarrow \{0, 1\}^{64}$$

more than $2^{2^{64}}$ vertices may be required. This number is 2^{63} times greater than the number of atoms in the universe!² There is no chance that we can construct or describe suitable hardware for every circuit of this kind. Shannon's estimate theorem (Theorem 7.12), which we shall encounter below, proves mathematically that for most Boolean functions, it is impossible to find a logic circuit whose size is less than order $O(2^n)$.

- Since the construction method proposed in the proof of Theorem 7.11 is inefficient, the question arises whether there are alternative methods for constructing simpler circuits for the same function.

i **Definition 7.13:** Two logic circuits G_1, G_2 are called **equivalent** if they compute the same Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$.

i **Definition 7.14:** A set of logical connectives $\mathcal{C}$ is called a **basis** for a logic circuit G if all logic gates of G belong to the basis $\mathcal{C}$. The set of connectives $\mathcal{C} = \{\neg, \vee, \wedge\}$ is called the **De Morgan basis**, and every logic circuit constructed over this basis is called a **De Morgan circuit**.

- As we shall learn in Exercise 14 at the end of the chapter, ternary logical connectives acting on three Boolean variables can also be defined, and so on. Most of the literature on Boolean logic is based on **binary logical connectives** that act on a pair of variables, and **the unary connective** $\neg$, which acts on a single Boolean variable.
- There are exactly two unary connectives, and exactly **16** binary connectives (why?). In most cases, only connectives from the set $\{\neg, \vee, \wedge, \downarrow, \oplus, \uparrow\}$. Exercise 3 at the end of the chapter defines two additional logical connectives, $\rightarrow, \leftrightarrow$. Apart from these, there are 9 additional binary logical connectives that we can define. The set of 16 logical connectives is called the **complete binary basis**.
- It is clear from the definition that every DNF circuit is a De Morgan circuit, but in many cases DNF circuits can be simplified into much simpler De Morgan circuits (at the cost of increasing the depth by order $O(n)$) that are not necessarily DNF circuits.
- We restate one of the most fundamental problems in circuit design: given a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$ and a set of logical connectives $\mathcal{C}$, find a smallest logic circuit G over the basis $\mathcal{C}$ for G . For clarity, we first define the notions of the size and depth of a circuit.

i **Definition 7.15:** Let $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$ be a Boolean function. The **size** $\text{Size}(f)$ of f is defined as the minimum number of gates among all logic circuits that compute f . The **depth** $\text{Depth}(f)$ of f is defined as the minimum depth of a logic circuit for f . More precisely, we note that the logic gates that may be used in the logic circuits come from the basis $\{\text{AND}, \text{OR}, \text{NOT}, \text{XOR}, \text{NOR}, \text{NAND}\}$.

² By a very rough estimate

- Without an agreed basis of logic gates, this definition has no meaning, and therefore it is important to specify the set of logic gates from which the circuits in question are constructed. The quantities **Size**(f) or **Depth**(f) may be smaller if we enlarge the basis of logic gates.
- Recall that the logic gates **{AND, OR, XOR, NOR, NAND}** are multivariable. They act on two or more Boolean variables, unlike binary gates such as $\wedge$, $\vee$, or $\oplus$, which act on only a pair of Boolean variables.
- It turns out that this is a very difficult problem (an NP-complete problem, as we shall see below). This means that there are on practical algorithms for finding the simplest circuits for a given Boolean function, and to this very day an intensive search is under way for methods of simplifying logic circuits. A large portion of the research and development effort in the chip industry is devoted to problems of this kind, and this effort will probably never cease.
- In many practical cases, increasing the depth of the graph allows a significant reduction in the size of the graph (the number of vertices and edges). For this reason, logic-circuit design is one of the most difficult and complex fields in the chip industry.
- **Example 7.25:** Define the Boolean function $f: \{0, 1\}^3 \rightarrow \{0, 1\}$ by Table 7.6.

$$f(x_1, x_2, x_3) = y_1$$

x_1	x_2	x_3	y_1
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

Table 7.6: Value table for the function $f(x_1, x_2, x_3) = y_1$

By the construction method presented in the proof of Theorem 7.11, we can construct a DNF circuit for f , as shown in Figure 7.35

It is easy to convert Table 7.6 (or the DNF circuit in Figure 7.35) into a corresponding DNF expression ϕ

$$\phi = (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)$$

The expression ϕ can be simplified using the Karnaugh-map technique, which we shall not cover in this course, or by using Boolean algebra. We use the idea presented in Example 7.2

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \iff x_1 \wedge x_2 \wedge (x_3 \vee \neg x_3) \iff x_1 \wedge x_2$$

The expression $x_3 \vee \neg x_3$ is a tautology and can therefore be omitted. Replace the first two



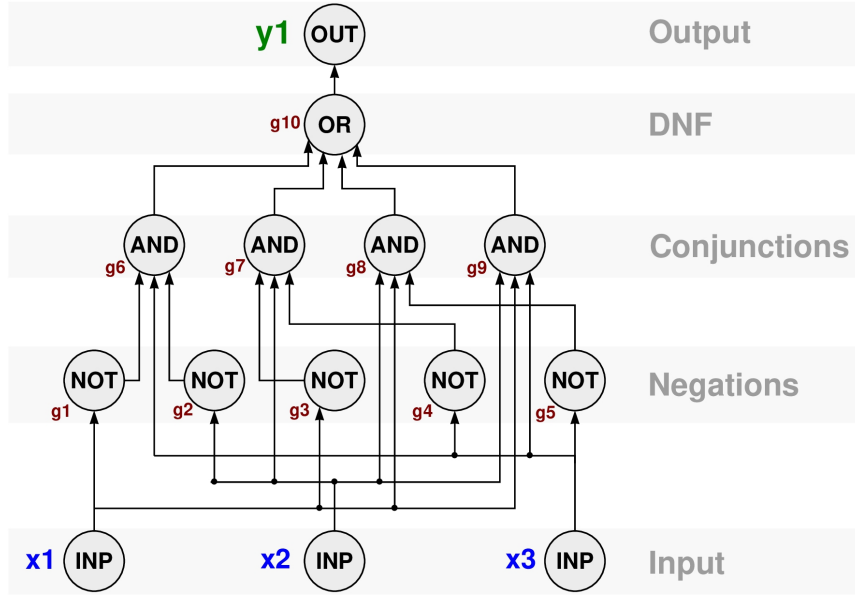


Fig. 7.35: A DNF circuit for the Boolean function $f: \{0, 1\}^3 \rightarrow \{0, 1\}$ in Example 7.25

conjunctions in the expression ϕ by $x_1 \wedge x_2$. Using the distributive law and eliminating a tautology, we can continue to simplify ϕ

$$\begin{aligned}
 \phi &\iff x_1 \wedge x_2 \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \\
 &\iff [(x_1 \wedge x_2 \vee \neg x_1 \wedge x_2) \wedge (x_1 \wedge x_2 \vee \neg x_3)] \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \\
 &\iff [x_2 \wedge (x_1 \vee \neg x_1) \wedge (x_1 \wedge x_2 \vee \neg x_3)] \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \\
 &\iff [x_2 \wedge (x_1 \wedge x_2 \vee \neg x_3)] \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \\
 &\iff (x_1 \wedge x_2) \vee (x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) \\
 &\iff x_2 \wedge (x_1 \vee \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)
 \end{aligned}$$

We have therefore obtained

$$\phi \iff x_2 \wedge (x_1 \vee \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)$$

It is not difficult to construct a logic circuit for the simplified expression, as shown in Figure 7.36. This is, of course, a De Morgan circuit (the basis is $\{\neg, \vee, \wedge\}$) and not a DNF circuit.

We succeeded in reducing the number of logic gates from 10 to 7. The number of edges (or inputs; it is enough to count arrows), on the other hand, decreased from 25 to 12! This is, of course, a significant reduction in the circuit's complexity.

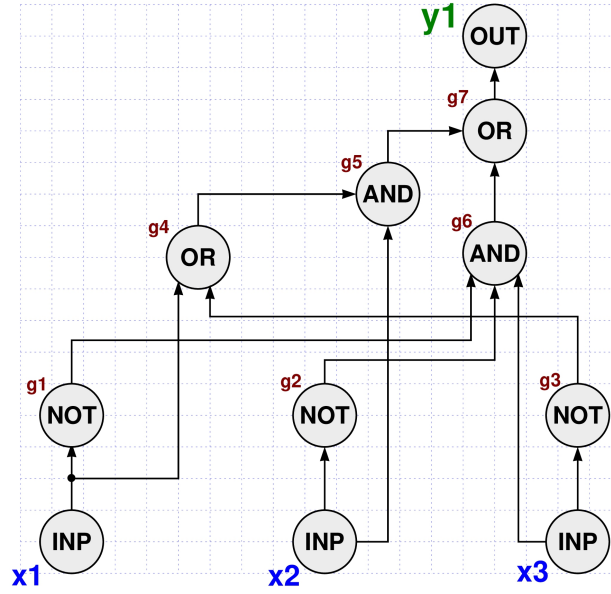


Fig. 7.36: A simpler De Morgan circuit for the Boolean function $f: \{0,1\}^3 \rightarrow \{0,1\}$ in Example 7.25

7.5 Shannon's Estimate Theorem

- Recall the notation $\{0,1\}^*$, which denotes the collection of all binary sequences of arbitrary length

$$\{0,1\}^* = \bigcup_{n=0}^{\infty} \{0,1\}^n$$

This is, in fact, the collection of all finite binary sequences.

- To connect with the computability topics discussed in the preceding chapters, we shall also have to consider infinite Boolean functions

$$f: \{0,1\}^* \rightarrow \{0,1\}^*$$

We restrict ourselves to a particular type in which the length of the output $f(\vec{x})$ is fixed for every $\vec{x} \in \{0,1\}^n$.

Definition 7.16: A Boolean function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is called **regular** if, for every natural number n , there is a natural number m such that $f(\{0,1\}^n) \subseteq \{0,1\}^m$; that is, the length of $f(\vec{x})$ is m for every binary sequence $\vec{x} \in \{0,1\}^n$.

- Example 7.26:** In fact, we encountered such functions at the beginning of the chapter under the

term **operator**

$$\text{AND: } \{0, 1\}^* \rightarrow \{0, 1\}$$

$$\text{OR: } \{0, 1\}^* \rightarrow \{0, 1\}$$

$$\text{XOR: } \{0, 1\}^* \rightarrow \{0, 1\}$$

$$\text{NOR: } \{0, 1\}^* \rightarrow \{0, 1\}$$

$$\text{NAND: } \{0, 1\}^* \rightarrow \{0, 1\}$$

The operators **AND**, **OR**, **XOR**, **NAND**, are extensions of the binary logical connectives $\wedge$, $\vee$, $\oplus$, $\downarrow$, $\uparrow$, to Boolean sequences of arbitrary length. As we saw at the beginning of the chapter, such an extension is possible only for binary logical connectives that satisfy the associative and commutative laws (as the following exercise proves).

✦ **Exercise 7.4:** Let $\diamond$ be a binary logical connective satisfying the commutative law $x \diamond y = y \diamond x$, and the associative law $(x \diamond y) \diamond z = x \diamond (y \diamond z)$. Let $\phi(x_1, x_2, \dots, x_n)$ be a Boolean expression in the language $\mathcal{B}(\diamond)$. Prove that for every satisfying assignment $(a_1, a_2, \dots, a_n)$ and every permutation $\vec{b}$ of $\vec{a}$, we have $\phi(\vec{a}) \iff \phi(\vec{b})$.

Hint: Try induction on the length of ϕ .

✦ **Example 7.27:** Using the above operators, we can define more complex Boolean functions $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ such as

$$f(x_1, x_2, \dots, x_n) = (\text{OR}(x_1, x_2, \dots, x_n), \text{NAND}(x_1, x_2, \dots, x_n), x_1 \oplus x_n)$$

or, in abbreviated form,

$$f(\vec{x}) = (\text{OR}(\vec{x}), \text{NAND}(\vec{x}), \vec{x}[1] \oplus \vec{x}[n])$$

The form of the function is $f: \{0, 1\}^* \rightarrow \{0, 1\}^3$.

- ✦ Computability theory distinguishes between uniform computational models and nonuniform computational models. A logic circuit is one of the well-known examples of a nonuniform computational model, in the sense that infinitely many different logic circuits $\{G_n\}_{n=1}^{\infty}$ are required to compute a complete Boolean function $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$.
- ✦ For every natural number, define the **restriction** of a regular Boolean function f to the domain $\{0, 1\}^n$ to be the Boolean function $f_n: \{0, 1\}^n \rightarrow \{0, 1\}^m$ satisfying $f_n(\vec{x}) = f(\vec{x})$ for every $\vec{x} \in \{0, 1\}^n$. A regular function f is said to be computable if there exists an infinite sequence of logic circuits $\{G_n\}_{n=1}^{\infty}$ having a common basis, such that for every natural number n , the circuit G_n computes the function f_n .

❗ **Theorem 7.12: Shannon's Estimate Theorem.**

For every $\varepsilon > 0$, there exists a natural number N such that, for every $n \geq N$, an over-



whelming majority of the Boolean functions $f: \{0,1\}^n \rightarrow \{0,1\}$ (with probability $1 - \epsilon$) satisfy

$$\text{Size}(f) \geq (1 - \epsilon) \frac{2^n}{n}$$

That is, for an overwhelming majority of the functions $f: \{0,1\}^n \rightarrow \{0,1\}$, there is no circuit of size less than $(1 - \epsilon) \frac{2^n}{n}$ that computes f .

Proof: The proof by Claude Elwood Shannon (1916-2001) is longer and more detailed. We shall be satisfied with rough estimates that obtain the same result. The guiding idea is: there are enormously many Boolean functions from $\{0,1\}^n$ to $\{0,1\}$, but far too few small logic circuits. That is, there are not enough small logic circuits for all possible Boolean functions.

In set theory, we learned that the number of functions $f: A \rightarrow B$ is $|B|^{|A|}$, where $|A|$ denotes the number of elements in the set A . Therefore, there are 2^{2^n} Boolean functions $f: \{0,1\}^n \rightarrow \{0,1\}$.

Let s be a given natural number. We try to estimate (roughly) how many logic circuits of size s exist. Every gate may have at most s inputs (including the possibility of one input for a **NOT** gate). Therefore, the number of possible input sets for a given gate is

$$\binom{s}{1} + \binom{s}{2} + \binom{s}{3} + \cdots + \binom{s}{s} = 2^s - 1$$

Since our basis contains six types of gates, **{AND, OR, NOT, XOR, NOR, NAND}**, the number of possibilities for mapping logical functions to the circuit vertices is 6^s . We can conclude that there are at most $6^s \cdot 2^s = 12^s$ ways to construct a logic circuit with s gates. This is, of course, a very rough upper estimate.

Given a natural number n , take $s = (1 - \epsilon) \frac{2^n}{n}$. We wish to compare the number of logic circuits of size s that can be constructed for Boolean functions $f: \{0,1\}^n \rightarrow \{0,1\}$, with the number of possible functions (2^{2^n}).

$$\begin{aligned} 12^s &< 24^s \cdot s^s = 24^{(1-\epsilon)\frac{2^n}{n}} \cdot \left[(1 - \epsilon) \frac{2^n}{n} \right]^{(1-\epsilon)\frac{2^n}{n}} \\ &= \left[\frac{24(1 - \epsilon)}{n} \right]^{(1-\epsilon)\frac{2^n}{n}} \cdot 2^{(1-\epsilon)2^n} < 2^{2^n} \end{aligned}$$

A rough estimate for the number of circuits with at most s gates is

$$12^1 + 12^2 + 12^3 + \cdots + 12^s < s \cdot 2^s$$



We compute the following limit

$$\begin{aligned}
 \lim_{n \rightarrow \infty} \frac{s \cdot 12^s}{2^{2^n}} &= \lim_{n \rightarrow \infty} \frac{(1 - \varepsilon)^{\frac{2^n}{n}} \cdot 12^{(1 - \varepsilon)\frac{2^n}{n}}}{2^{2^n}} = (1 - \varepsilon) \lim_{n \rightarrow \infty} \frac{12^{(1 - \varepsilon)\frac{2^n}{n}}}{n \cdot 2^{2^n - n}} \\
 &\leq \lim_{n \rightarrow \infty} \frac{16^{\frac{2^n}{n}}}{2^{2^n - n}} = \lim_{n \rightarrow \infty} \frac{2^{\frac{4}{n} \cdot 2^n}}{2^{2^n - n}} = \lim_{n \rightarrow \infty} \frac{1}{2^{2^n - \frac{4}{n} \cdot 2^n - n}} \\
 &= \lim_{n \rightarrow \infty} \frac{1}{2^{\frac{n-4}{n} \cdot 2^n - n}} = 0
 \end{aligned}$$

Clearly, the denominator $2^{\frac{n-4}{n} \cdot 2^n - n}$ tends to infinity as $n \rightarrow \infty$, and therefore the above limit is zero. By the definition of the limit of a sequence, there exists a natural number N such that, for every $n \geq N$, $\frac{s \cdot 12^s}{2^{2^n}} < \varepsilon$. As noted above, $s \cdot 12^s$ is an upper bound on the number of logic circuits of size s for Boolean functions $f: \{0, 1\}^n \rightarrow \{0, 1\}$. Therefore, for every $n \geq N$, with probability $1 - \varepsilon$, a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$ has a logic circuit of size $(1 - \varepsilon)\frac{2^n}{n}$ or less that computes it. In other words: for every $n \geq N$, an overwhelming majority of the Boolean functions $f: \{0, 1\}^n \rightarrow \{0, 1\}$ have no logic circuit of size less than $(1 - \varepsilon)\frac{2^n}{n}$ that computes them. ■

- The practical significance of Shannon's estimate theorem is that for an overwhelming majority of Boolean functions, there is no logic circuit of practical size that computes them. That is: for sufficiently large natural numbers n , with probability nearly 1, the smallest logic circuit that exists for a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$ has size at least $(1 - \varepsilon)\frac{2^n}{n}$. This is a growth rate of exponential order, which is the number-one enemy of every algorithm.

i Corollary 7.13: (Shannon) An overwhelming majority of the Boolean functions $f: \{0, 1\}^n \rightarrow \{0, 1\}$ cannot be expressed by a Boolean formula of length less than $(1 - \varepsilon)\frac{2^n}{n}$, for every $0 < \varepsilon < 1$.

- ✓ **Proof:** We shall be satisfied with the idea of the proof. As we saw on the preceding pages, there is a close correlation between the size of a formula and the size of its corresponding logic circuit, and therefore this is an immediate consequence of Shannon's estimate theorem. ■

Binary Encoding of a Logic Circuit

- To define a complete computational model based on a logic circuit, we need a binary encoding of a logic circuit. Recall that a logic circuit is defined as a directed graph $G = (V, E, L)$, as described in Definition 7.7. We use the same encoding idea as in the section 6.8 in Chapter 6.
- Without loss of generality, we may assume that in every such graph input vertices are labeled x_i , output vertices are labeled y_i , and logic gates are labeled g_i , where i is a natural number in decimal notation. We also need parentheses and a single punctuation mark:

Symbols = {x y g 0 1 2 3 4 5 6 7 8 9 () , NOT AND OR XOR NOR NAND}



Altogether, using 22 symbols, we can describe every finite logic-circuit graph. The following table shows the mapping of the 22 symbols to a 5-bit binary code for each symbol.

Symbol	binary	Symbol	binary
0	00000	x	01011
1	00001	y	01100
2	00010	g	01101
3	00011	NOT	01110
4	00100	OR	01111
5	00101	AND	10000
6	00110	XOR	10001
7	00111	NOR	10010
8	01000	NAND	10011
9	01001	(	10100
,	01010	)	10101

Table 7.7: Representation of the elementary symbols by binary strings of length 5

- Since the list of vertices of the graph can be reconstructed from the list of edges, there is no need to encode the set V . We use parentheses to group all ordered pairs in the edge set E and in the gate-labeling set L . The following example provides a complete explanation of the encoding process.
- ❖ **Example 7.28:** We compute the binary code of the logic circuit presented in Example 7.4. The string that we must encode is

$$s = \text{"(x1, g1), (x2, g1), (x2, g3), (x2, g4), (x3, g1), (x3, g5), (x4, g1), (x4, g3), (x4, g5), (x5, g4), (x5, g5), (g1, g2), (g1, g4), (g2, y1), (g3, y2), (g3, g4), (g4, y3), (g5, g4), (g5, y4), ((g1, AND), (g2, NOT), (g3, OR), (g4, XOR), (g5, XOR))"}$$

The string s contains 189 elementary symbols, and therefore the binary code contains 945 binary digits ($189 \times 5 = 945$)

```

⟨G⟩ = 101001010001011100001010100110100001101010101010100010110001001010011010000110101
01010101000101100010010100110100011101010101010100010110001001010011010010010101
01010101000101100011010100110100001101010101010100010110001101010011010010110101
01010101000101100100010100110100001101010101010100010110010001010011010001110101
010101010001011001000101001101001011010101010101000101100101010011010010010101
010101010001011001010100110100101101010101010100011010000101010011010001010101
0101010100010110100010101001100000101010101010100011010001101010011010010010101
01010101000101001000101001100000111010101010101000110100101010011010010010101
01010101000101001010100110000100101011010110100101000110100001010101000010101
0101010100010100010010100111010101010101000110100011010100111101010101010100
01101001000101010001101010101010100011010010101010100011010110101

```

Figure 7.37 shows how the binary string s is divided into 5-bit bytes, with the symbol represented by each byte written above it.

(	(	x	1	,	g	1	)	,	(	x	2	,	g	1	)	
10100	10100	01011	00001	01010	01101	00001	10101	01010	10100	01011	00010	01010	01101	00001	10101	
,	(	x	2	,	g	3	)	,	(	x	2	,	g	4	)	
01010	10100	01011	00010	01010	01101	00011	10101	01010	10100	01011	00010	01010	01101	00100	10101	
,	(	x	3	,	g	1	)	,	(	x	3	,	g	5	)	
01010	10100	01011	00011	01010	01101	00001	10101	01010	10100	01011	00011	01010	01101	00101	10101	
,	(	x	4	,	g	1	)	,	(	x	4	,	g	3	)	
01010	10100	01011	00100	01010	01101	00001	10101	01010	10100	01011	00100	01010	01101	00011	10101	
,	(	x	4	,	g	5	)	,	(	x	5	,	g	4	)	
01010	10100	01011	00100	01010	01101	00101	10101	01010	10100	01011	00101	01010	01101	00100	10101	
,	(	x	5	,	g	5	)	,	(	g	1	,	g	2	)	
01010	10100	01011	00101	01010	01101	00101	10101	01010	10100	01101	00001	01010	01101	00010	10101	
,	(	g	1	,	g	4	)	,	(	g	2	,	y	1	)	
01010	10100	01101	00001	01010	01101	00100	10101	01010	10100	01101	00010	01010	01100	00001	10101	
,	(	g	3	,	y	2	)	,	(	g	3	,	g	4	)	
01010	10100	01101	00011	01010	01100	00010	10101	01010	10100	01101	00011	01010	01101	00100	10101	
,	(	g	4	,	y	3	)	,	(	g	5	,	g	4	)	
01010	10100	01101	00100	01010	01100	00011	10101	01010	10100	01101	00101	01010	01101	00100	10101	
,	(	g	5	,	y	4	)	)	(	(	g	1	,	AND	)	
01010	10100	01101	00101	01010	01100	00100	10101	10101	10100	01101	00001	01010	10000	10101		
,	(	g	2	,	NOT	)	,	(	g	3	,	0	)	,	(	
01010	10100	01101	00010	01010	01110	10101	01010	10100	01101	00011	01010	01111	10101	01010	10100	
g	4	,	XOR	)	,	(	g	5	,	XOR	)	)				
01101	00100	01010	10001	10101	01010	10100	01101	00101	01010	10001	10101	10101				

Fig. 7.37: The binary code of the logic circuit G , divided into five-bit bytes

i Definition 7.17: A binary function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is said to be **computable** by a sequence of logic circuits $\{G_n\}_{n=1}^\infty$ if, for every n and every $\vec{x} \in \{0,1\}^n$, $f(\vec{x}) = G_n(\vec{x})$.

i Proposition 7.14: Every function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is **computable** by a sequence of logic circuits $\{G_n\}_{n=1}^\infty$

✓ Proof: This is, of course, an immediate consequence of Theorem 7.11. By that proposition, for every n there exists a DNF circuit G_n that computes the component $f_n: \{0,1\}^n \rightarrow \{0,1\}^*$. ■

✗ Proposition 7.14 has no practical significance, since in most cases we do not have the computational ability to construct the logic circuits G_n . As we learned from Shannon's estimate theorem (Theorem 7.12), there is probability nearly 1 that every circuit G_n is of exponential order (and cannot be simplified), and therefore the assertion that such a circuit exists has no practical significance. We must therefore restrict ourselves to circuit sequences that can be computed.

i Definition 7.18: A sequence of logic circuits $\{G_n\}_{n=1}^\infty$ is called **recursive** if there exists a Turing machine M that accepts the language 1^n , and, for every natural number n , the output on 1^n is the binary code $\langle G_n \rangle$ of the logic circuit G_n .

7.6 Finished, but Not Complete

The present chapter is still under development and will require several further changes and additions, which will be made in the future if time permits. Suggestions for improvements and corrections will be gratefully received. Complete solutions (including handwritten ones) to the exercises will be welcome. An e-mail address appears at the bottom of the page.

Exercises for chapter 7

1. Study Exercise 32 at the end of Chapter 4, and then construct a context-free grammar for the language of Boolean expressions over the alphabet

$$T = \{x, y, z, \wedge, \vee, \neg, (,)\}$$

2. Let $f: \{0, 1\}^4 \rightarrow \{0, 1\}$ be the following function

$$f(x_1, x_2, x_3, x_4) = |\max(x_1, x_2) - \min(x_3, x_4)|$$

Explain why f is a Boolean function. You must prove that all the values it takes are 0 or 1. Try to find a Boolean expression ϕ in the language $\mathcal{B}(\neg, \vee, \wedge)$, as short as possible, whose truth table is f .

3. The logical connectives $\rightarrow, \leftrightarrow$ are defined by the following truth tables³

x	y	$x \rightarrow y$
1	1	1
1	0	0
0	1	1
0	0	1

x	y	$x \leftrightarrow y$
1	1	1
1	0	0
0	1	0
0	0	1

- (a) Prove that the set of connectives $\{\neg, \leftrightarrow\}$ is not complete.
- (b) Prove that the set of connectives $\{\neg, \rightarrow\}$ is minimally complete.
- (c) Is the set of connectives $\{\rightarrow, \leftrightarrow\}$ complete?
- (d) Is the set of connectives $\{\uparrow, \leftrightarrow\}$ complete?

³ The logical connective $\rightarrow$ is the **material-implication connective**.

The logical connective $\leftrightarrow$ is the logical-equivalence connective, **logical equivalence**.



Hint: Part (a): Prove that the number of satisfying assignments of every expression in the language $\mathcal{B}(\neg, \leftrightarrow)$ is even. Use induction on the length of the expression. Part (b): after proving that the set $\{\neg, \leftrightarrow\}$ is complete, you must also prove that the set $\{\rightarrow\}$ is not complete! Try to find a simple property shared by all expressions in the language $\mathcal{B}(\rightarrow)$.

4. The logical connective **XNOR** is defined by the following truth table (the negation of **XOR**).

XNOR

x	y	$x \odot y$
1	1	1
1	0	0
0	1	0
0	0	1

Study the logic circuit in Figure 7.38, and find all the satisfying assignments for this circuit.

Hint: Find all binary sequences (a_1, a_2, a_3) which, when fed into the circuit as input, give $y = 1$. Note: observe that the truth tables of the two connectives $\leftrightarrow$, $\odot$ are identical! They are in fact the same logical connective with two different names.

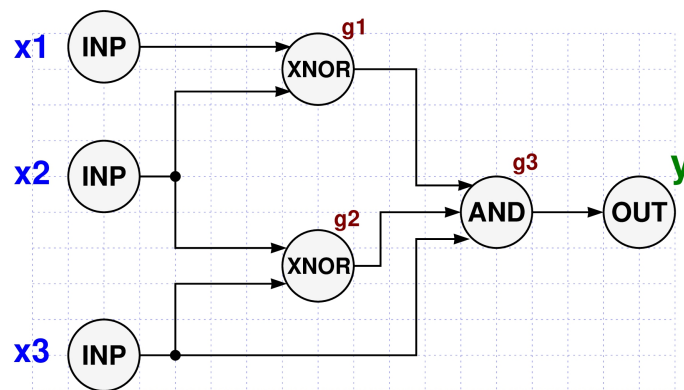


Fig. 7.38: A logic circuit for Exercise 4

- Find a Boolean expression in the language $\mathcal{B}(\uparrow)$ equivalent to the expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.
- Find a Boolean expression in the language $\mathcal{B}(\oplus)$ equivalent to the expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.
- Study the logic circuit in Figure 7.39 carefully and convert it into an equivalent Boolean expression $\phi(x_1, x_2, x_3)$. Using Boolean algebra, simplify the expression ϕ to a shorter Boolean expression ψ . Convert the resulting expression ψ into a corresponding logic circuit. Verify that the new logic circuit has fewer logic gates than the original circuit. Final task: convert the circuit obtained into an equivalent logic circuit constructed solely from **NAND** gates.
- Study the logic circuit in Figure 7.40. This circuit has 7 logic gates. Try to simplify it to a logic circuit having only 3 logic gates. Try to prove that it cannot be simplified to a logic circuit having only two logic gates. All circuits in this exercise are restricted to the set of gates **OR**,

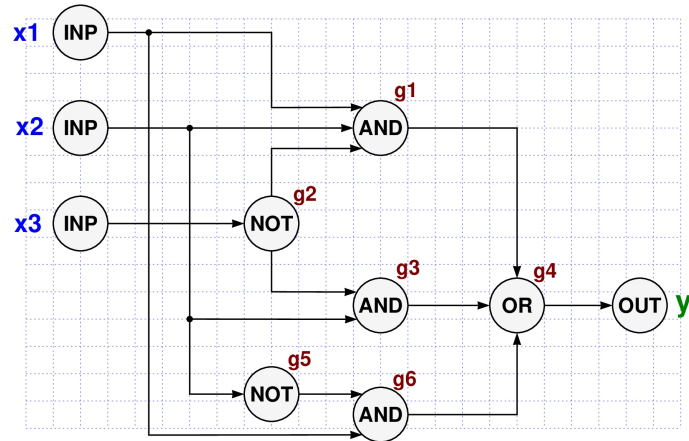


Fig. 7.39: A logic circuit for Exercise 7

AND, NOT, NAND.

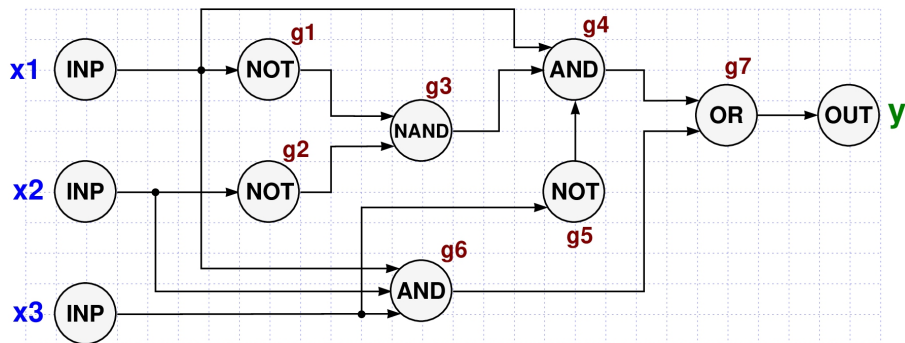


Fig. 7.40: A logic circuit for Exercise 8

9. Study the logic circuit in Figure 7.41. This circuit is constructed entirely from **NAND** components. Find an equivalent Boolean expression $\phi(x_1, x_2)$ corresponding to this circuit. Using Boolean algebra, simplify the expression ϕ to a shorter Boolean expression ψ in the language $\mathcal{B}(\neg, \vee, \wedge)$.

Hint: The truth function of the circuit may help.

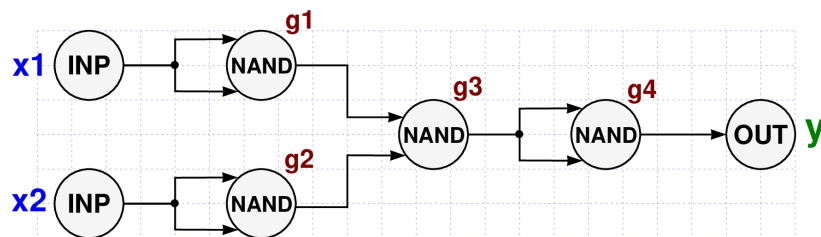


Fig. 7.41: A logic circuit for Exercise 9

10. Design a logic circuit that receives four input bits x_0, x_1, x_2, x_3 and produces an output bit y , such that $y = 1$ if and only if the input $x_3x_2x_1x_0$ is a binary number divisible by 2.
11. Design a logic circuit that receives four input bits x_1, x_2, x_3, x_4 and produces an output bit y , such that $y = 1$ if and only if the number of ones in the sequence (x_1, x_2, x_3, x_4) is even.
12. Design a logic circuit $\text{MULT}_{2 \times 3}$ that receives 5 input bits: s_0, s_1, x_0, x_1, x_2 , and produces

5 output bits: y_0, y_1, y_2, y_3, y_4 , such that the product of the two binary numbers s_1s_0 and $x_2x_1x_0$ is $y_4y_3y_2y_1y_0$.

Hint: For this purpose, you may use the 3-bit adder defined in Example 7.17, or construct a full 3-bit adder for this purpose.

13. Design a logic circuit G that receives 8 input bits: $x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4$, and produces an output bit y , such that

$$f(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4) = 1 \iff x_1 + y_1 = x_2 + y_2 = x_3 + y_3 = x_4 + y_4 = 1$$

The circuit must be as small and shallow as possible.

Hint: In other words, the sequence (y_1, y_2, y_3, y_4) is the complement of the sequence (x_1, x_2, x_3, x_4) .

14. The C programming language includes a **ternary** logical connective called **SEL**, defined by

$$x = 1 \implies x?y:z = y, \quad x = 0 \implies x?y:z = z$$

More precisely, we write the truth table of $x?y:z$ (see Table 7.8)

x	y	z	$x?y:z$
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

Table 7.8: The truth table of the logical connective $x?y:z$

Note that, unlike the preceding connectives, this is a logical connective acting on three Boolean values! It is therefore called **ternary**. Determine whether this connective constitutes a complete system of connectives.

Hint: In other words, the truth value of $x?y:z$ is identical to the value y if $x = 1$, and to the value z if $x = 0$. This is in fact a multiplexer of order 1, denoted by **MUX1**. Try to determine whether the expressions $x \vee y$, $x \wedge y$, $\neg x$ can be expressed using the connective **SEL**.

15. Construct a logic circuit for the expression

$$\phi = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$$

in which all logic gates are **NAND** gates and the number of inputs to every logic gate is exactly 2. What is the circuit with the minimum number of such gates that can be constructed for ϕ ?



16. Construct a logic circuit for the expression

$$\phi = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$$

in which all logic gates are **NAND** gates and the number of inputs to every logic gate is exactly 2. What is the circuit with the minimum number of such gates that can be constructed for ϕ ?

17. Design a logic circuit G , as simple as possible, over the standard basis **{AND, OR, NOT}**, that receives an input of four bits x_3, x_2, x_1, x_0 and produces an output of 7 bits $y_1, y_2, y_3, y_4, y_5, y_6, y_7$. The four input bits $x_3x_2x_1x_0$ are to represent a decimal digit between 0 and 9. The output represents 7 control bits for a digital display with 7 illuminated segments (see Figure 7.42). The output includes 7 control bits for turning the display segments on or off. The following table shows an example of three runs of the circuit G for the digits 3, 4, 5.

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \longrightarrow \{0, 1\}^4 \longrightarrow \{0, 1\}^7$$

d	x_3	x_2	x_1	x_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1

(7.8)

Hint: Complete Table (7.8). There are 16 binary sequences of length 4, and therefore the complete table contains 16 rows. Note that only 10 rows in the table represent decimal digits (the remaining rows are irrelevant, and arbitrary values of y_i may be chosen for them). The practical assumption is that an input that does not represent a decimal digit will never be fed into the circuit.

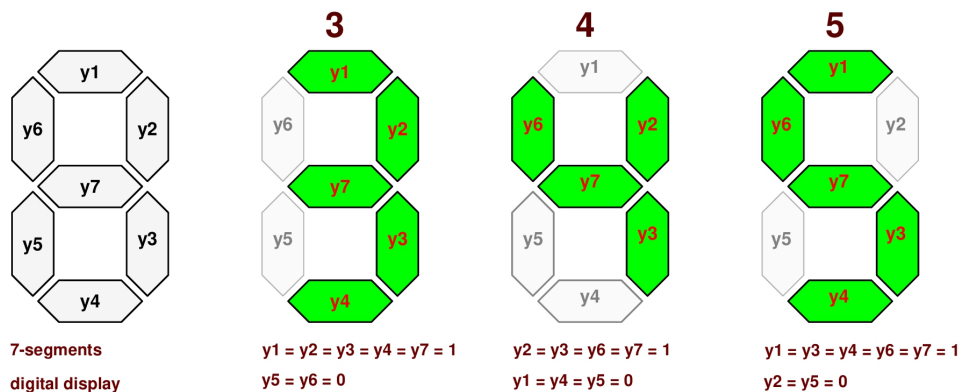


Fig. 7.42: A 7-segment digital display. Diagram for Exercise 17.

18. Figure 7.28 presents a design for implementing the circuit **COUNT6**, which counts the number of ones in a 6-bit input using a pair of **COUNT3** circuits and a full 2-bit adder. Verify that the idea presented in the figure actually works by writing code corresponding to the figure in the PyCirc language and testing all possible inputs for the circuit.
19. Let $\vec{a} = (a_1, a_2, \dots, a_n)$ be a sequence of Boolean values. We say that $a_i = 1$ is an **isolated one** if its neighboring values are 0. That is: when $1 < i < n$, $a_{i-1} = a_{i+1} = 0$; when $i = 1$,

$a_2 = 0$; and when $i = n$, $a_{n-1} = 0$. An example of a Boolean sequence with four isolated ones is shown in Figure 7.43. In this example, the isolated ones are a_1 , a_4 , a_{10} , a_{16} .

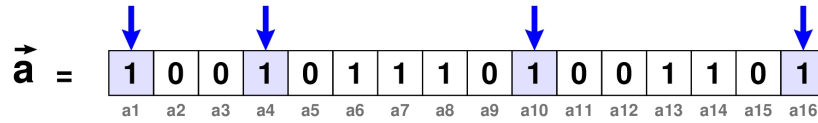


Fig. 7.43: Diagram for Exercise 19

- (a) Prove that every sequence of length 8 has at most four isolated ones.
- (b) Define the Boolean function $f: \{0,1\}^8 \rightarrow \{0,1\}^3$ by $f(\vec{x}) = (y_1, y_2, y_3)$, where (y_1, y_2, y_3) is the number of isolated ones in the sequence $\vec{x}$. By the preceding part, three bits suffice to express this number in binary notation. Construct a DNF circuit similar to that in Figure 7.33 for the function f .
- (c) Write a program in the PyCirc specification language for the circuit constructed in the preceding part, and verify that it operates correctly by testing all possible sequences. Use the code examples in the chapter.
20. Study Figure 7.44 and try to conjecture which Boolean function is implemented by the circuit in the figure. Write a corresponding PyCirc program for this circuit and verify your conjecture by running the program on all possible inputs. Give as precise a verbal specification as possible of the operation of a circuit of this kind for every possible size (input size n and output size 2^n). Construct a similar circuit **DECODER3** for an input of size $n = 3$, and verify it by writing a corresponding PyCirc program.

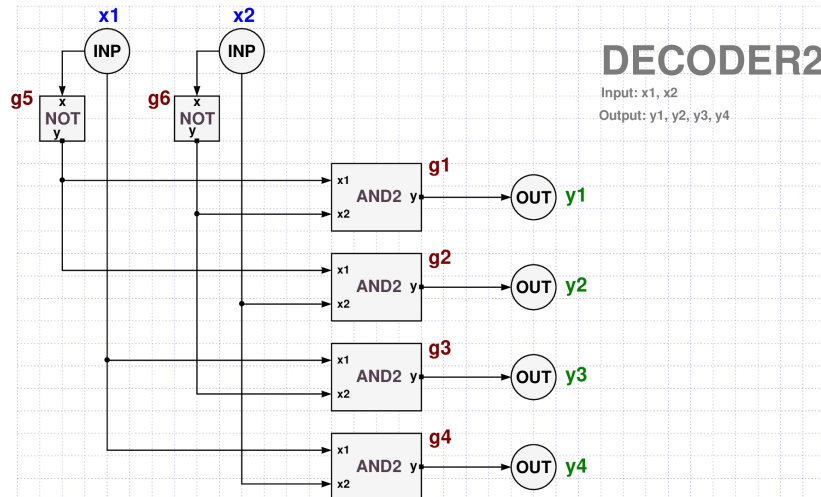


Fig. 7.44: Diagram for Exercise 20

21. (a) Construct a Boolean function $f: \{0,1\}^8 \rightarrow \{0,1\}^3$ that compares two binary numbers

of 4 bits each, $a = (a_3, a_2, a_1, a_0)$, $b = (b_3, b_2, b_1, b_0)$,

$$f(a_3, a_2, a_1, a_0, b_3, b_2, b_1, b_0) = \begin{cases} (1, 0, 0), & a < b \\ (0, 1, 0), & a = b \\ (0, 0, 1), & a > b \end{cases}$$

(b) Construct a logic circuit for the cell **MC4** (4-bits Magnitude Comparator) that implements this Boolean function. You may use any of the cells constructed above. See Figure 7.45.

(c) Write a corresponding PyCirc program for the circuit that you constructed.

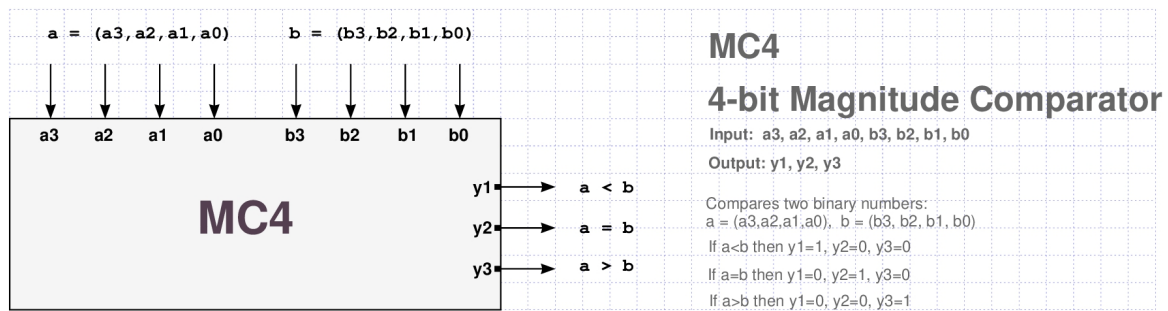


Fig. 7.45: Diagram for Exercise 21

22. Draw a logic circuit for the Boolean function

$$f(x_1, x_2, x_3) = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3)$$

based solely on **DECODER** and **OR** cells. The **DECODER** cell was defined in Exercise 20. Can you construct a logic circuit for f based solely on an **8x1** multiplexer (**MUX3**)?

23. **Project 1:** Based on your understanding of the language PyCircPL, design a similar programming language for a nondeterministic finite automaton.

Hint: Review the definitions of the various automata at the beginning of the booklet. Of course, you are not asked to implement the programming language, but only to provide its syntactic specification. For example, there should be a command such as $T(q_3, a) = q_1, q_2, q_5$, meaning that in state q_3 , on input a , the machine must move to one of the states q_1 , q_2 , or q_5 .

24. **Project 2:** Based on your understanding of the language PyCircHDL, design a similar programming language for the hardware description of a multitape Turing machine.

Hint: Review the definition of a Turing machine from Chapter 6. Of course, you are not asked to implement the programming language, but only to provide its syntactic specification.

Chapter 9

Random-Access Machines (RAM)

9.1 General Description

- A **random-access machine**, or, briefly, a **RAM** machine, is an enhancement of a register machine. Among all the computational models encountered in this course, this is the one closest to the structure of the computer used in everyday life. We shall take this opportunity to use this machine to illustrate several additional concepts and properties of computability (somewhat more than is usually customary in courses of this kind). Here is the formal definition of a register machine.

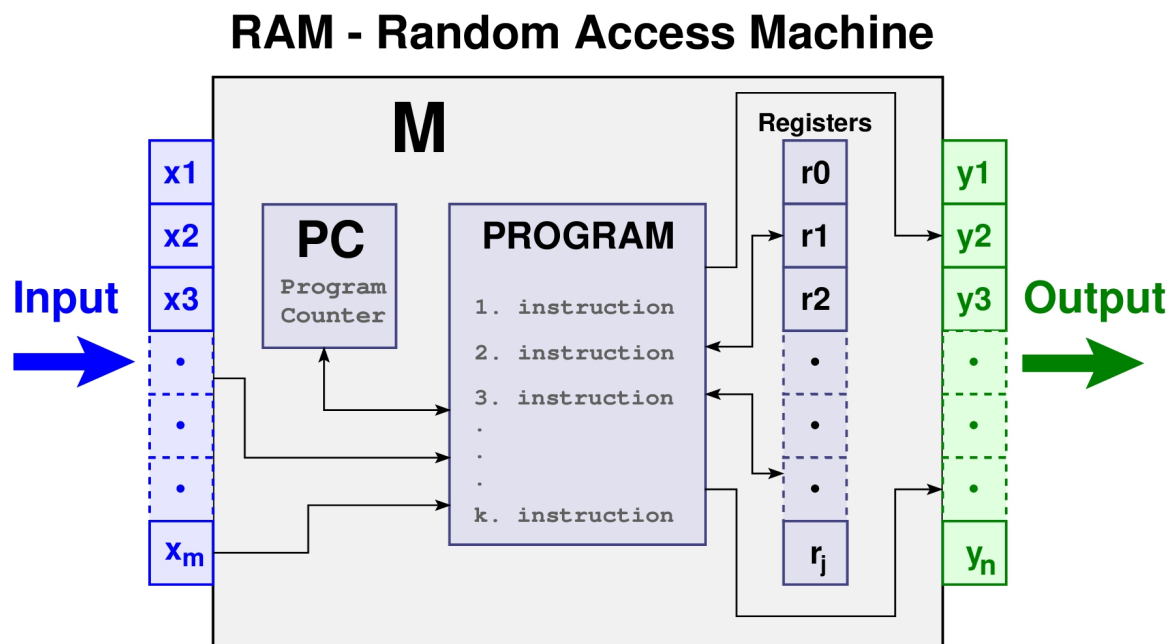


Fig. 9.1: Architecture of a random-access machine **M**

- ◆ One of the principal limitations of a Turing machine is its inability to access cells directly. If the machine head is positioned on cell number 100, then 900 computation steps are required to

access cell 1000 in order to read its contents or write to it. In a RAM machine, this limitation does not exist—any input cell can be read or any output cell written in a single step!

i Definition 9.1: A *random-access machine* M is a theoretical machine consisting of the following components

(a) A sequence of *input cells*

$$x_1, x_2, x_3, \dots, x_m$$

Each input cell can contain an integer. At initialization, all cells contain zero.

(b) A sequence of *output cells*

$$y_1, y_2, y_3, \dots, y_n$$

Each output cell can contain an integer. At initialization, all cells contain zero.

(c) A sequence of *internal memory cells (registers)*

$$r_0, r_1, r_2, \dots, r_n$$

These are internal memory cells by means of which the machine performs the computations required by its program P .

(d) A program P consisting of a sequence of operating **instructions**. The instructions are numbered from 1 to some natural number k . At each stage, the machine executes the instruction whose number is stored in the program counter PC .

(e) The **program counter** PC : a natural-number memory cell indicating the number of the instruction to be executed at the next stage. In the initial state, $PC = 1$.

- Unlike the register machine seen in the preceding chapter, a RAM machine has a clear separation among input cells, output cells, and internal memory cells (registers). The machine cannot write to input cells! It can only read them! The machine can, of course, write to the output cells.
- The customary practice is to perform all computations using the internal memory cells (registers), and to write the result to the output cells only near the end. The principal component responsible for performing the work is, of course, the **program** at the heart of the machine.
- Unlike a modern computer, which can switch among different programs, a RAM machine has one fixed program that it executes in every computation cycle.

i Definition 9.2: A **RAM program** is a numbered sequence of instructions. Every instruction in the program must be of one of the following types:

- $a = b$ Load the value of cell b into cell a .
- $a = i$ Load cell a with the integer i
- $a = b \pm c$ An addition or subtraction operation
- $a = b * c$ A multiplication operation
- JUMP**(i) Set the value of the program counter PC to i ($PC = i$)



JZERO (r, i)	If $r = 0$, then $PC = i$ (Jump If Zero)
JPOS (r, i)	If $r > 0$, then $PC = i$ (Jump If Positive)
JE (r_1, r_2, i)	If $r_1 = r_2$, then $PC = i$ (Jump If Equal)
HALT	Halt instruction

- ◆ In instructions of type $a = b$, cell a must be a register or an output cell. Cell b may be any of the three possibilities: an input cell, an output cell, or a register.
- ◆ In every instruction that does not refer to the program counter PC , the machine automatically increments the program counter PC by 1 immediately after executing the instruction. The machine halts at a **HALT** instruction, after which the user can read its output from the output cells y_i .

- A RAM machine has a finite set of registers $r_0, r_1, r_2, \dots, r_j$. These are in fact **internal memory cells** by means of which the machine's program performs the required computations without altering the input or output cells (as happens in a simple register machine).
- In assembly-language programs, register r_0 has a traditional role in the program's instruction sequence (accumulator). In the examples below, we shall not make significant use of it.
- The customary practice is to copy the input cells x_i to appropriate registers and only then perform the required computations (in any case, the input cells cannot be written to!).
- In a theoretical RAM machine, every cell may contain an integer (positive or negative) of arbitrary size. In real computers, of course, the contents of a cell are limited in size (in practice, they are a binary string of 32 or 64 bits).
- The **program counter PC (Program Counter)** is a special memory component in the machine whose purpose is to hold the number of the instruction to be executed next. At initialization, of course, $PC = 1$. Every instruction in the program may change the value of PC directly if it is an instruction of the form **JUMP**, **JZERO**, **JPOS**, or **JE**. In every other case, the value of PC is incremented by 1 after the instruction terminates.
- Unlike a Turing machine, whose head must travel a long distance from one cell to another, in a RAM machine every memory cell can be accessed immediately. For example, if the machine head is positioned over cell r_{27} , and the program needs to read r_{527} , an ordinary Turing machine must visit every cell between cell r_{27} and cell r_{527} ! That is, 500 computation steps are required! In a RAM machine, this operation is performed immediately—only a single step is required to move from cell r_{27} to cell r_{527} !
- In various texts, an instruction of type $a = b$ is sometimes called a **load instruction (Load)**. For example, the instruction $r_3 = x_7$ **loads** the input value x_7 into register r_3 .
- An instruction of the form $a = i$, where a is an input/output/register cell and i is an integer, is sometimes called a **store instruction (Store)**. For example, the instruction $r_5 = -18$ **stores**



the integer -18 in register r_5 .

- ◆ **Example 9.1:** We shall construct a RAM machine called **EXP** for the exponentiation function $f(x_1, x_2) = x_1^{x_2}$. This must be designed carefully: the machine must handle boundary cases such as 0^0 , x^0 , 0^y . The expression 0^0 is undefined, and this must be indicated by an error bit. Machine **M** has two input cells, x_1 , x_2 , and two output cells, y_1 , y_2 . Cell y_1 contains the result of exponentiation for a valid input. Cell y_2 contains 1 if the input is invalid ($x_1 = x_2 = 0$).
- ◆ Before writing the program for **EXP**, we present a flowchart in Figure 9.2.

EXP(x1,x2 : y1,y2):

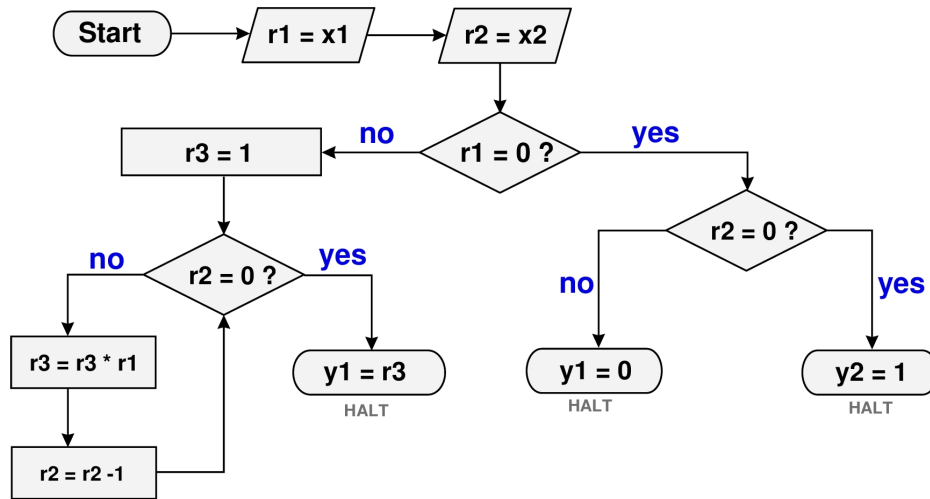


Fig. 9.2: Flowchart for the exponentiation function **EXP**

- ◆ The program for the machine **EXP** is presented in Figure 9.3.

PROGRAM EXP(x1, x2 : y1, y2)

```

1. r1 = x1
2. r2 = x2
3. JZERO(r1, 9)
4. r3 = 1
5. JZERO(r2, 14)
6. r3 = r3 * r1
7. r2 = r2 - 1
8. JUMP(5)
9. JZERO(r2, 12)
10. y1 = 0
11. JUMP(15)
12. y2 = 1
13. JUMP(15)
14. y1 = r3
15. HALT
  
```

EXP(x1,x2 : y1,y2)

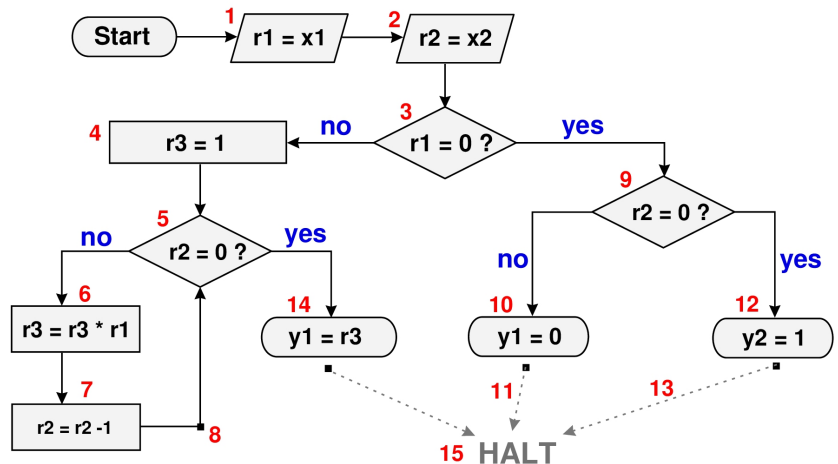


Fig. 9.3: Mapping instruction numbers to the flowchart for the exponentiation function **EXP**

- ◆ **Example 9.2:** We shall construct a RAM machine called **DIVMOD** for the division-with-remainder function $f(x_1, x_2) = \text{divmod}(x_1, x_2)$. The function **divmod** (as it is called in various programming languages) gives the integer quotient $y_1 = x_1/x_2$ and the remainder $y_2 = x_1 \% x_2$. Therefore, our machine contains two input cells, x_1, x_2 , and two output cells, y_1, y_2 . As before, the machine uses three memory cells, r_1, r_2, r_3 .
- ◆ Figure 9.4 shows the flowchart and program for **DIVMOD**. When studying the program, one must bear in mind that every time the machine starts operating, all cells are initialized to zero! Therefore, we did not bother to add instructions that clear the memory cells, as we did, for example, in an ordinary register machine.
- ◆ For simplicity, our algorithm does not handle invalid input for which $x_2 = 0$. Constructing a complete algorithm that handles every type of input requires additional effort, and we leave this as an exercise for the student. See Exercise 2 at the end of the chapter.

PROGRAM DIVMOD($x_1, x_2 : y_1, y_2$)

```

1.  $r_1 = x_1$ 
2.  $r_2 = x_2$ 
3.  $r_0 = r_2 - r_1$ 
4. JPOS( $r_0, 8$ )
5.  $r_3 = r_3 + 1$ 
6.  $r_1 = r_1 - r_2$ 
7. JUMP(3)
8.  $y_1 = r_3$ 
9.  $y_2 = r_1$ 
10. HALT

```

DIVMOD($x_1, x_2 : y_1, y_2$):

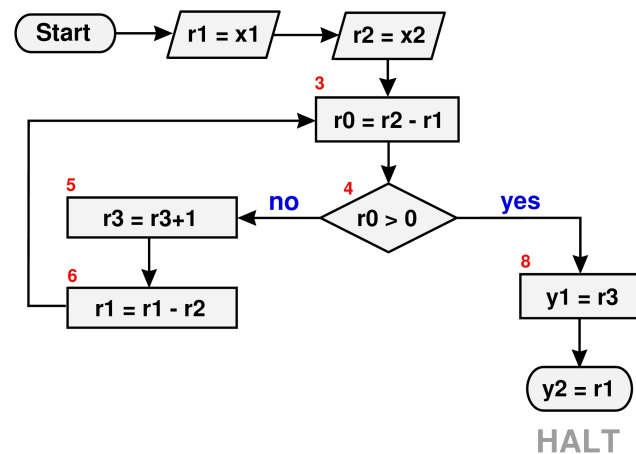


Fig. 9.4: A RAM program and flowchart for division with remainder, **DIVMOD**

- ◆ **Example 9.3:** We shall construct a RAM machine called **MAX4** for the maximum function of a sequence of 4 integers, $\text{MAX4}(x_1, x_2, x_3, x_4)$. Machine **MAX4** has four input cells, x_1, x_2, x_3, x_4 , and a single output cell, y_1 . At the end of the computation, the output value y_1 must be the maximum value of the vector (x_1, x_2, x_3, x_4) . The machine uses a single memory cell! r_0 .
- ◆ Figure 9.5 shows the flowchart and program for **MAX4**.
- For convenience, RAM programs can be written and run in the following Jupyter notebook:

 **PyRamPl - Python Random Access Machine Simulation Package**

This environment may be very helpful in solving the exercises at the end of the chapter.

PROGRAM MAX4($x_1, x_2, x_3, x_4 : y_1$)

1. $y_1 = x_1$
2. $r_0 = y_1 - x_2$
3. JPOS($r_0, 5$)
4. $y_1 = x_2$
5. $r_0 = y_1 - x_3$
6. JPOS($r_0, 8$)
7. $y_1 = x_3$
8. $r_0 = y_1 - x_4$
9. JPOS($r_0, 11$)
10. $y_1 = x_4$
11. HALT

MAX4($x_1, x_2, x_3, x_4 : y_1$)

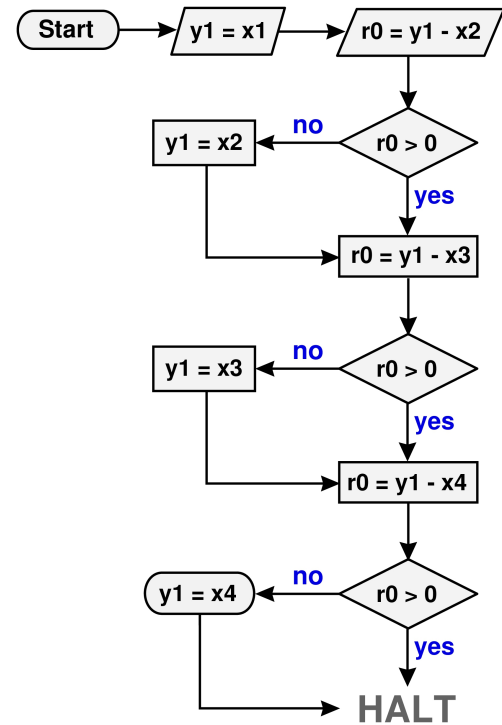


Fig. 9.5: A RAM program and flowchart for the machine MAX4

9.2 Run Tables

- As with a register machine, the operation of a RAM machine can also be understood by studying run tables. A run table of a RAM machine consists of the list of the machine's configurations, beginning with the initialization stage (Init), in which the input is supplied, and ending with the termination stage (HALT), at which the computation ends (if it does indeed end).
- Each row in the run table includes the configuration number, the current instruction in the program, and the contents of all input, memory, and output cells (after the current instruction has been executed). The transition from one configuration to the next is performed by executing exactly one instruction.
- Executing an instruction may change one of the memory or output cells (but not an input cell!), or change the program counter **PC**.
- ❖ **Example 9.4:** Figure 9.6 shows the run table of the machine **EXP** (the exponentiation function) on input $x_1 = 3$, $x_2 = 4$. Unlike the run table of an ordinary register machine, here we see distinct columns (colored yellow, pink, and light blue) for input cells, memory cells, and output cells. The contents of input cells x_1 , x_2 remain fixed throughout the run, since input cells cannot be written to (except during initialization).

PROGRAM EXP($x_1, x_2 : y_1, y_2$)

```

1. r1 = x1
2. r2 = x2
3. JZERO(r1, 9)
4. r3 = 1
5. JZERO(r2, 14)
6. r3 = r3 * r1
7. r2 = r2 - 1
8. JUMP(5)
9. JZERO(r2, 12)
10. y1 = 0
11. JUMP(15)
12. y2 = 1
13. JUMP(15)
14. y1 = r3
15. HALT

```

	INPUT
	OUTPUT
	MEMORY

Frame	Instruction	x1	x2	r1	r2	r3	y1	y2
0	Init	3	4	0	0	0	0	0
1	1. r1 = x1	3	4	3	0	0	0	0
2	2. r2 = x2	3	4	3	4	0	0	0
3	3. JZERO(r1, 9)	3	4	3	4	0	0	0
4	4. r3 = 1	3	4	3	4	1	0	0
5	5. JZERO(r2, 14)	3	4	3	4	1	0	0
6	6. r3 = r3 * r1	3	4	3	4	3	0	0
7	7. r2 = r2 - 1	3	4	3	3	3	0	0
8	8. JUMP(5)	3	4	3	3	3	0	0
9	5. JZERO(r2, 14)	3	4	3	3	3	0	0
10	6. r3 = r3 * r1	3	4	3	3	9	0	0
11	7. r2 = r2 - 1	3	4	3	2	9	0	0
12	8. JUMP(5)	3	4	3	2	9	0	0
13	5. JZERO(r2, 14)	3	4	3	2	9	0	0
14	6. r3 = r3 * r1	3	4	3	2	27	0	0
15	7. r2 = r2 - 1	3	4	3	1	27	0	0
16	8. JUMP(5)	3	4	3	1	27	0	0
17	5. JZERO(r2, 14)	3	4	3	1	27	0	0
18	6. r3 = r3 * r1	3	4	3	1	81	0	0
19	7. r2 = r2 - 1	3	4	3	0	81	0	0
20	8. JUMP(5)	3	4	3	0	81	0	0
21	5. JZERO(r2, 14)	3	4	3	0	81	0	0
22	14. y1 = r3	3	4	3	0	81	81	0
23	15. HALT	3	4	3	0	81	81	0

Fig. 9.6: Run table for the machine EXP: computing the exponentiation function $\exp(x_1, x_2) = x_1^{x_2}$. Input: $x_1 = 3$, $x_2 = 4$.

❖ **Example 9.5:** Figure 9.6 shows the run table of the machine DIVMOD (the integer-division-with-remainder function) on input $x_1 = 18$, $x_2 = 5$.

9.3 Generalized RAM Machine

- Because of the limited number of available instructions, constructing a RAM machine is generally a complex and tedious task, even for a simple function. Every addition or removal of a program line requires all lines of code to be updated, and even if we have constructed a clear flowchart, it does not always make writing a linear program for the machine easier.
- The way to address this problem is to construct simple building blocks and use them as components to construct increasingly complex machines, as we did for logic circuits in Chapter 7. This idea is presented in the following definition.

i Definition 9.3: A generalized RAM machine M is a RAM machine that contains RAM machines as **components (Components)**. Unlike an ordinary RAM machine, the program P of M may use component names as additional instructions. For this purpose, it has an extended instruction set: for every internal machine M_i , there is an instruction of the form

$$b_1, b_2, \dots, b_m = M_i(a_1, a_2, \dots, a_n) \quad (9.1)$$

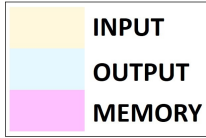
Whenever one of the internal machines M_i is called, machine M initializes machine M_i with

PROGRAM DIVMOD($x_1, x_2 : y_1, y_2$)

```

1.  $r_1 = x_1$ 
2.  $r_2 = x_2$ 
3.  $r_0 = r_2 - r_1$ 
4. JPOS( $r_0, 8$ )
5.  $r_3 = r_3 + 1$ 
6.  $r_1 = r_1 - r_2$ 
7. JUMP(3)
8.  $y_1 = r_3$ 
9.  $y_2 = r_1$ 
10. HALT

```



Frame	Instruction	x_1	x_2	r_0	r_1	r_2	r_3	y_1	y_2
0	Init	18	5	0	0	0	0	0	0
1	1. $r_1 = x_1$	18	5	0	18	0	0	0	0
2	2. $r_2 = x_2$	18	5	0	18	5	0	0	0
3	3. $r_0 = r_2 - r_1$	18	5	-13	18	5	0	0	0
4	4. JPOS($r_0, 8$)	18	5	-13	18	5	0	0	0
5	5. $r_3 = r_3 + 1$	18	5	-13	18	5	1	0	0
6	6. $r_1 = r_1 - r_2$	18	5	-13	13	5	1	0	0
7	7. JUMP(3)	18	5	-13	13	5	1	0	0
8	3. $r_0 = r_2 - r_1$	18	5	-8	13	5	1	0	0
9	4. JPOS($r_0, 8$)	18	5	-8	13	5	1	0	0
10	5. $r_3 = r_3 + 1$	18	5	-8	13	5	2	0	0
11	6. $r_1 = r_1 - r_2$	18	5	-8	8	5	2	0	0
12	7. JUMP(3)	18	5	-8	8	5	2	0	0
13	3. $r_0 = r_2 - r_1$	18	5	-3	8	5	2	0	0
14	4. JPOS($r_0, 8$)	18	5	-3	8	5	2	0	0
15	5. $r_3 = r_3 + 1$	18	5	-3	8	5	3	0	0
16	6. $r_1 = r_1 - r_2$	18	5	-3	3	5	3	0	0
17	7. JUMP(3)	18	5	-3	3	5	3	0	0
18	3. $r_0 = r_2 - r_1$	18	5	2	3	5	3	0	0
19	4. JPOS($r_0, 8$)	18	5	2	3	5	3	0	0
20	8. $y_1 = r_3$	18	5	2	3	5	3	3	0
21	9. $y_2 = r_1$	18	5	2	3	5	3	3	3
22	10. HALT	18	5	2	3	5	3	3	3

Fig. 9.7: Run table for the machine **DIVMOD**: computing integer division with remainder. Input: $x_1 = 3$, $x_2 = 4$.

the corresponding values of cells $a_1, \dots, a_n$, and after machine M_i completes its computation, machine M copies the output values to cells $b_1, \dots, b_m$.

- Figure 9.8 presents a schematic description of a generalized RAM machine. At the bottom of the diagram, we see a sequence of components $M_1, M_2, \dots, M_k$ consisting of internal RAM machines contained within the generalized machine M . The arrowed lines indicate two-way communication between the program P of machine M and the components $M_1 \dots M_k$. Whenever an extended instruction of type (9.1) is executed, machine M sends a computation request to component M_i in one direction, and receives the output values in the other direction.
- To avoid confusion, we did not state in the definition that each of the machines $M_1, \dots, M_k$ may itself be a generalized RAM machine! Thus, this is in fact a recursive definition, but we shall not elaborate further, since we have already considered a similar hierarchical model for logic circuits.
- As with the hierarchical structure of logic circuits encountered in Chapter 7, in the present case we can also develop a similar (in fact, almost identical) hardware-description language for describing generalized RAM machines. To avoid unnecessary duplication, we shall not discuss this subject further and leave it to the student as a learning and research challenge.
- ◆ **Example 9.6:** We shall construct a generalized RAM machine called **GCD** for Euclid's algorithm for finding the greatest common divisor of two natural numbers x_1, x_2 . Machine **GCD** has a single RAM component, which is a copy of the machine **DIVMOD** constructed in the preceding

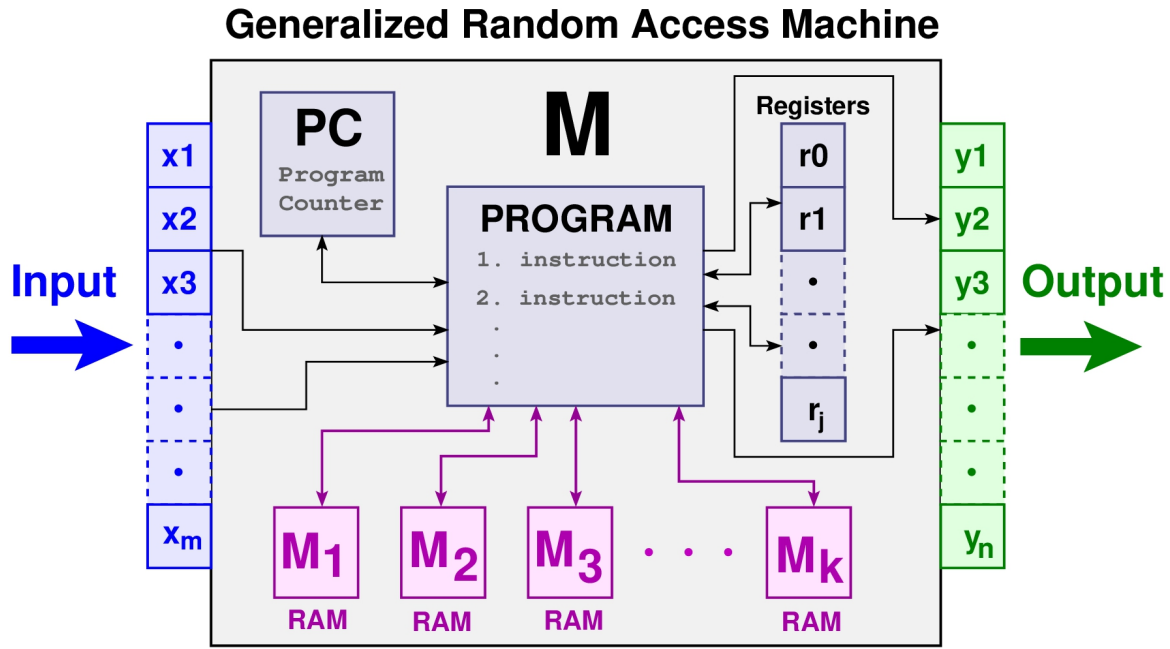


Fig. 9.8: Architecture of a generalized RAM machine M

example. Figure 9.9 shows the flowchart and program for **GCD**.

- ◆ Instructions 15, 17, shown in red, are extended instructions of the form (9.1). All other instructions in the **GCD** program are standard instructions of an ordinary RAM machine. It is important to note that in instructions 15, 17, the machine **DIVMOD** receives only the values of memory cells r_1 , r_2 , and has no way to change the values of its inputs! Therefore, there is no danger that the operation of an internal component of a generalized RAM machine will contaminate its memory space.
- ◆ As with the idea presented in the preceding chapter (see Section ??), using machine **DIVMOD** in the program of **GCD** is a procedure in the GCD algorithm! We can, of course, justify this use in the same way that we justified the use of procedures for register machines, but we shall be satisfied with the general idea sketched in Figure ??, without entering into the fine details.

i Theorem 9.1: Every generalized RAM machine can be converted into an ordinary RAM machine.

- The conversion process is called **compiling** or **compilation**, and it was presented in detail in Section ?? of the preceding chapter in the context of register machines, which are a kind of early version of RAM machines; therefore, we shall not repeat it here.
- The immediate conclusion from Theorem 9.1 is that generalized RAM machines do not extend the computational power of ordinary RAM machines. That is, every function $f: \mathbb{Z}^m \rightarrow \mathbb{Z}^n$ computable by generalized RAM machines can also be computed by ordinary RAM machines. The only, and very important, difference is that the ordinary machine may be extremely large and complex, whereas the generalized machine is generally much simpler, easier to understand, and easier to control.

```

PROGRAM GCD( $x_1$ ,  $x_2$  :  $y_1$ )
1.  $r_1 = x_1$ 
2.  $r_2 = x_2$ 
3. JZERO( $r_1$ , 11)
4. JZERO( $r_2$ , 13)
5.  $r_0 = r_1 - r_2$ 
6. JPOS( $r_0$ , 15)
7.  $r_0 = r_2 - r_1$ 
8. JPOS( $r_0$ , 17)
9.  $y_1 = r_1$ 
10. HALT
11.  $y_1 = r_2$ 
12. HALT
13.  $y_1 = r_1$ 
14. HALT
15.  $r_3, r_1 = \text{DIVMOD}(r_1, r_2)$ 
16. JUMP(3)
17.  $r_3, r_2 = \text{DIVMOD}(r_2, r_1)$ 
18. JUMP(4)
19. HALT

```

GCD(x_1, x_2 : y_1)

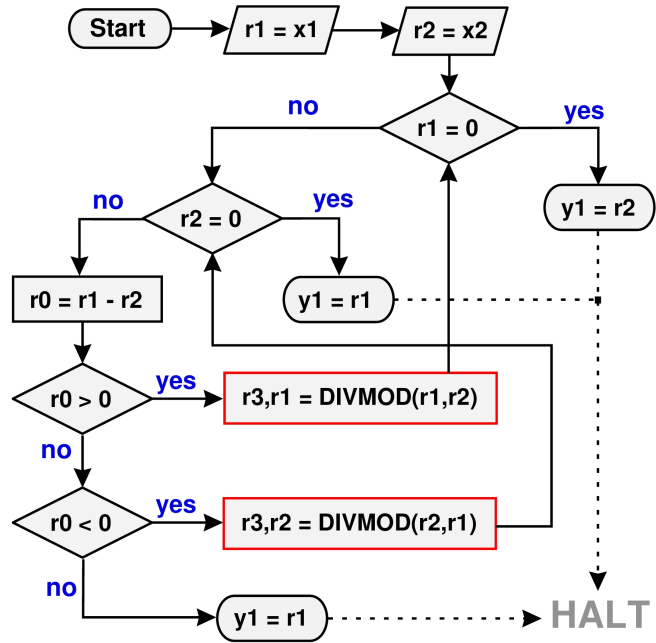


Fig. 9.9: A RAM program and flowchart for the GCD algorithm (Euclid)

❖ **Example 9.7:** We shall construct a generalized RAM machine called **PRIME** for an algorithm that tests whether a natural number x_1 is prime. This is the simplest algorithm (and also the least efficient). For simplicity, only positive integers are handled. Machine **PRIME** has a single RAM component, which is a copy of the machine **DIVMOD** constructed in Example 9.6. Figure 9.10 shows the flowchart and program for **GCD**.

- ◆ The idea is very simple: if $x_1 \leq 3$, then x_1 is prime, and therefore the machine halts with $y_1 = 1$. If $x_1 > 3$, the machine enters a loop that begins with $r_0 = 2$ and ends with $r_0 = x_1 - 1$, testing whether x_1 is divisible by r_0 . If it is, then $y_1 = 0$ and the machine halts.
- The following concept and examples will be important for proving the computational equivalence between a RAM machine and a Turing machine.

❗ **Definition 9.4:** Bit i of a natural number n is binary digit i in the binary representation of n (where counting proceeds from right to left).

❖ **Example 9.8:** The third bit of the number $n = 25$ is 1. This can be seen immediately from the binary representation $25 = 11001_2$. Bit 2 is 0, and bit 0 is 1! Thus, counting begins with $i = 0$ (from right to left). In the following example, we implement a RAM machine called

PROGRAM PRIME($x_1 : y_1$)

```

1.  $y_1 = 1$ 
2.  $r_1 = x_1 - 3$ 
3. JPOS( $r_1, 5$ )
4. HALT
5.  $r_0 = 2$ 
6.  $r_1, r_2 = \text{DIVMOD}(x_1, r_0)$ 
7. JZERO( $r_2, 11$ )
8.  $r_0 = r_0 + 1$ 
9. JE( $r_0, x_1, 12$ )
10. JUMP(6)
11.  $y_1 = 0$ 
12. HALT

```

PRIME($x_1 : y_1$)

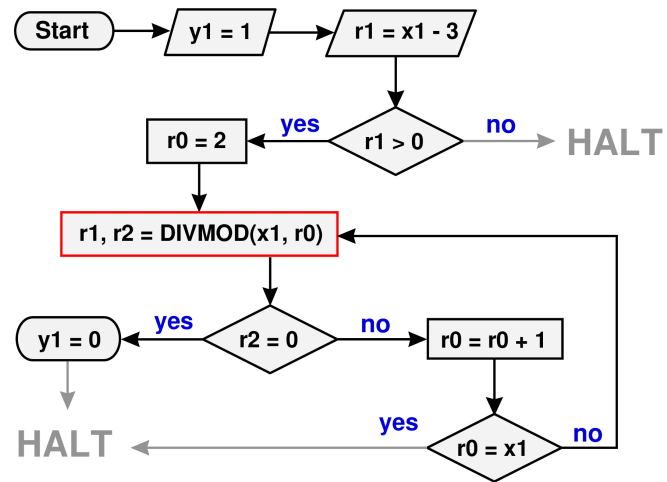


Fig. 9.10: A RAM program and flowchart for the algorithm **PRIME**

GETBIT that performs these computations for us

```

GETBIT(25, 0) = 1
GETBIT(25, 1) = 0
GETBIT(25, 2) = 0
GETBIT(25, 3) = 1
GETBIT(25, 4) = 1

```

This example contains the first germ of the idea: the natural number **25** represents the tape configuration **10011 ...** of a Turing machine. Further details follow.

◆ **Example 9.9:** We shall construct a generalized RAM machine called **GETBIT** for an algorithm that obtains bit x_2 of a natural number x_1 . The algorithm applies only to natural numbers. Figure 9.11 shows the flowchart and program for **GETBIT**.

◆ **Example 9.10:** The second operation needed to simulate a Turing machine by a RAM machine is changing a bit. We shall construct a generalized RAM machine called **SETBIT** to set bit x_2 of a natural number x_1 to the binary value x_3 . The algorithm applies only to natural numbers. Figure 9.12 shows the flowchart and program for **SETBIT**.

◆ We leave it to the student to verify the following examples

```

SETBIT(41, 0, 0) = 40
SETBIT(41, 0, 1) = 41
SETBIT(41, 3, 0) = 33
SETBIT(41, 3, 1) = 41
SETBIT(41, 4, 0) = 41
SETBIT(41, 4, 1) = 57

```



```

PROGRAM GETBIT(x1, x2 : y1)
  1. r1 = x1
  2. r2 = x2
  3. r1, r0 = DIVMOD(r1, 2)
  4. JZERO(r2, 7)
  5. r2 = r2 - 1
  6. JUMP(3)
  7. y1 = r0
  8. HALT

```

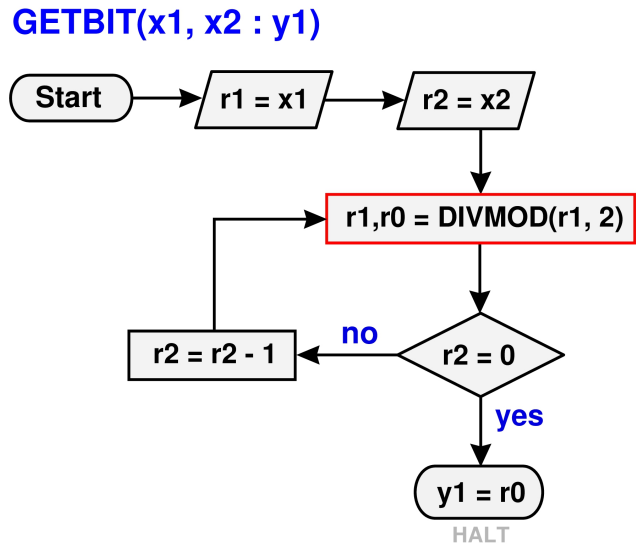


Fig. 9.11: A RAM program and flowchart for the algorithm **GETBIT**

We note only that $41 = 101001_2$.

- ◆ This is the first example in which we encounter a generalized machine one of whose components is itself a generalized machine (**GETBIT**). Machine **SETBIT** uses machine **GETBIT** internally, and machine **GETBIT** uses machine **DIVMOD**. This example reveals the hierarchical structure characteristic of many computational models.

9.4 Equivalence Between a RAM Machine and a Turing Machine

- In this section, we prove that a RAM machine has no advantage over a Turing machine in terms of computational power, and conversely. Of course, writing programs for a RAM machine is much more convenient than writing them for a Turing machine.
- We shall be satisfied with describing the general idea of the proofs of the two theorems.

i **Definition 9.5:** A partial function $f: \mathbb{Z}^m \rightarrow \mathbb{Z}^n$ is called **RAM-computable** if there exists a RAM machine that computes f .

i **Theorem 9.2:** Every RAM-computable function can also be computed by a Turing machine.

- ✓ **Proof:** This is the easy direction of the equivalence. In Section 6.4 (page 209), we learned that addition and multiplication operations on integers can be performed by Turing machines. This accounts for most of the power of RAM machines. The various JUMP instructions can be simulated using dedicated states. Further details will be given in the next version of the booklet.

■

PROGRAM SETBIT($x_1, x_2, x_3 : y_1$)

```

1.  $r_0 = \text{GETBIT}(x_1, x_2)$ 
2.  $\text{JE}(r_0, x_3, 9)$ 
3.  $r_1, r_2 = \text{EXP}(2, x_2)$ 
4.  $\text{JPOS}(x_3, 7)$ 
5.  $y_1 = x_1 - r_1$ 
6. HALT
7.  $y_1 = x_1 + r_1$ 
8. HALT
9.  $y_1 = x_1$ 
10. HALT

```

SETBIT($x_1, x_2, x_3 : y_1$)

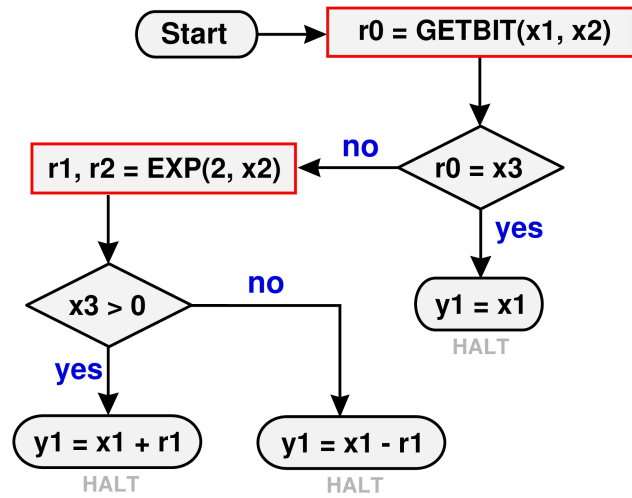


Fig. 9.12: A RAM program and flowchart for the algorithm SETBIT

Theorem 9.3: Every recursive function $f: \mathbb{Z}^m \rightarrow \mathbb{Z}^n$ is also RAM-computable.

Proof: The complete general proof is too long, too laden with details, and too tedious. We shall be satisfied with demonstrating its principal ideas on the simple Turing machine from Example 6.1 (page 196). The transition table of M is given in Table .

δ	0	1	B
s	$q_2, 1, R$	$q_1, 0, L$	$q_3, 1, R$
q_1	$q_1, 1, R$	$h, 0, R$	q_3, B, L
q_2	$q_1, 1, R$	$q_3, 0, R$	$q_1, 0, L$
q_3	$q_3, 1, R$	$q_1, 1, L$	q_2, B, L

Table 9.1: Transition table for the Turing machine M from Example 6.1

◆ We shall construct a RAM machine called **TSIM**($x_1 : y_1$) that simulates the Turing machine M . For this purpose, all its components must be represented by integers. The sequence of states

$$Q = \{s, q_1, q_2, q_3, h\}$$

is represented by

$$Q = \{0, 1, 2, 3, 4\}$$

That is, the initial state is 0 , state q_1 is represented by the number 1 , and so on. The input to machine M over a binary alphabet is represented by an integer x_1 , as we learned in the examples in the preceding section. For simplicity, we use more readable names for the registers of **TSIM**.

For example, **tape** is the register that stores the current configuration of the tape, **head** is an integer indicating the current position of the machine head, and **state** is a register that stores the current state of the machine.

```

PROGRAM TSIM(x1 : y1)
  1. tape = x1
  2. state = 0
  3. head = 0
  4. JE(state, 0, A)
  5. JE(state, 1, B)
  6. JE(state, 2, C)
  7. JE(state, 3, D)
  8. JE(state, 4, Z)
  :
  A. bit = GETBIT(tape, head)
  A+1. JE(bit, 0, X)
  A+2. state = 1
  A+3. SETBIT(tape, head, 0)
  A+4. head = head - 1
  A+5. JUMP(4)
  :
  X. state = 2
  X+1. SETBIT(tape, head, 1)
  X+2. head = head + 1
  X+3. JUMP(4)
  :
  Z. y1 = tape
  Z+1. HALT

```

- ◆ In the first instruction of the program, input x_1 is copied to a register called **tape**, which stores the tape of the Turing machine in decimal form. Register **state** stores the current state of the Turing machine, and register **head** stores the position of the machine head over the tape.
- ◆ In the second instruction of the program, register **state** is initialized to the value **0**, which is the initial state, and register **head** is initialized to the value **0**, which is the first cell of the tape. $x_1 = 41 = 101001_2$ for our RAM machine **TSIM**.
- ◆ Instructions 4-8 test the value of the state **state** and jump to the corresponding region of the program. For simplicity, we used the capital letters A, X, Z to represent instruction numbers in the program. For example, when **state** = **0**, the program jumps to instruction number A.
- ◆ The instructions numbered between A and A+5 cover only the first row of the Turing machine's transition table

$$\delta(s, 0) = (q_2, 1, R), \quad \delta(s, 1) = (q_1, 0, L)$$

Instruction A retrieves the current symbol on the tape and places it in register **bit**. If **bit** = **0**, the program jumps to instruction X. Otherwise, if **bit** = **1**, the assignment **state** = **1** is performed, the current bit is changed to **0** (**SETBIT(tape, head, 0)**), and the machine head is moved to the left (**head** = **head** - **1**).



- ◆ After each such loop terminates, the program jumps back to instruction 4, and the value of register **state** is tested again. If **state** = 4 (the halt state), the program jumps to state Z, where register **tape** is copied to output cell **y₁**, and a halt instruction is then executed.
- ◆ What we see is only a small part of the program. The complete program must contain approximately 50 instructions. We have presented only the part that handles the first row of the transition table. Each additional row requires about ten additional instructions in the program, and further instructions will probably be required to handle exceptions such as the tape head overflowing past the end or beginning of the tape.
- ◆ For example, if the input to the Turing machine **M** is **101001**, then it is translated into the input

$$x_1 = 41 = 101001_2$$

for our RAM machine, and instruction A then appears as follows

$$\text{bit} = \text{GETBIT}(\text{tape}, \text{head}) = \text{GETBIT}(41, 0) = 1$$

We shall be satisfied with these remarks for now, and leave it to the student to complete the general proof if this is required as a project in a more advanced course. ■

- For convenience, RAM programs can be written and run in the following Jupyter notebook:

 **PyRamPl - Python Random Access Machine Simulation Package**

This environment may be very helpful in solving the exercises at the end of the chapter.

9.5 Finished, but Not Complete

The present chapter is still under development and will require several further changes and additions, which will be made in the future if time permits. Suggestions for improvements and corrections will be gratefully received. Complete solutions (including handwritten ones) to the exercises will be welcome. An e-mail address appears at the bottom of the page.

➤ **To be continued ...**



Exercises for chapter 9

1. Construct a RAM machine for the **integer square-root** function **INTSQR**. The integer square root of a positive integer n is the largest integer m satisfying $m^2 \leq n$. For simplicity, if $n \leq 0$, the output of the machine is 0 .
2. Review Example 9.2 and design an improved **DIVMOD** algorithm that also handles invalid input ($x_2 = 0$). Verify that your algorithm operates correctly for every pair of integers (including negative integers and the number $0!$).

Hint: This time, the machine must contain three output cells, y_1, y_2, y_3 . The pair of cells y_1, y_2 functions as before in the case of valid input. In the case of invalid input, this must be indicated in cell y_3 by an appropriate value.

3. The Fibonacci function is defined recursively by

$$F(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ F(n-2) + F(n-1), & n \geq 2 \end{cases}$$

Construct a RAM machine called **FIBO** for the Fibonacci function.

4. Construct a generalized RAM machine for the maximum function **MAX8** on a sequence of **8** integers. Machine **MAX8** has eight input cells, $x_1, x_2, \dots, x_8$, and a single output cell, y_1 . At the end of the computation, the output value y_1 must be the maximum value of the vector $(x_1, x_2, \dots, x_8)$.

Hint: Use the machine **MAX4** constructed in Example 9.3. Apply **MAX4** once to the sequence (x_1, x_2, x_3, x_4) , and a second time to the sequence (x_5, x_6, x_7, x_8) .

5. Construct a generalized RAM machine called **NPRIMES** that receives two positive integers x_1, x_2 and counts how many prime numbers lie between x_1 and x_2 (inclusive).
6. A positive natural number n is called a **perfect number** if it is equal to the sum of its divisors. For example, $n = 6$ is a perfect number because $6 = 1 + 2 + 3$. The number $n = 28$ is also a perfect number because $28 = 1 + 2 + 4 + 7 + 14$.

(a) Construct a RAM machine called **ISPERFECT** that receives a positive natural number x_1 and tests whether it is perfect. The output is $y_1 = 1$ if x_1 is perfect; otherwise, $y_1 = 0$.

(b) Construct a RAM machine called **NPERFECT** that receives a positive natural number x_1 and counts how many perfect numbers there are between 1 and x_1 (including x_1).



7. A RAM-Turing machine is defined exactly like a Turing machine (see Definition 6.1 at the beginning of Chapter 6), but the machine head can reach any cell on the tape immediately. Every cell on the machine's tape has an address in the form of a natural number. The **transition**



function δ differs from the usual one:

$$\delta : (Q - \{h\}) \times \Gamma \times \mathbb{N} \rightarrow Q \times \Gamma \times \mathbb{N}$$

The meaning of $\delta(q, a, m) = (r, b, n)$ is: in state q , when the machine head reads symbol a in the cell whose address is m , the machine writes b , the machine head moves to cell n , and the machine enters state r .

(a) Give a complete formal definition of a RAM-Turing machine.

(b) Construct a RAM-Turing machine for the function

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*, \quad f(w) = ww$$

and estimate its computation speed relative to an ordinary Turing machine (with a single tape).

(c) Prove the equivalence of a RAM-Turing machine and an ordinary Turing machine.

