An Introduction to the Theory of Computation

Eitan Gurari, Ohio State University
Computer Science Press, 1989, ISBN 0-7167-8182-4

To Shaula, Inbal, Itai, Erez, Netta, and Danna

[errata | sketches of solutions | notes on the hypertext version | zipped files]

[next] [tail] [up]

Preface

Computations are designed to solve problems. Programs are descriptions of computations written for execution on computers. The field of computer science is concerned with the development of methodologies for designing programs, and with the development of computers for executing programs. It is therefore of central importance for those involved in the field that the characteristics of programs, computers, problems, and computation be fully understood. Moreover, to clearly and accurately communicate intuitive thoughts about these subjects, a precise and well-defined terminology is required.

This book explores some of the more important terminologies and questions concerning programs, computers, problems, and computation. The exploration reduces in many cases to a study of mathematical theories, such as those of automata and formal languages; theories that are interesting also in their own right. These theories provide abstract models that are easier to explore, because their formalisms avoid irrelevant details.

Organized into seven chapters, the material in this book gradually increases in complexity. In many cases, new topics are treated as refinements of old ones, and their study is motivated through their association to programs.

Chapter 1 is concerned with the definition of some basic concepts. It starts by considering the notion of strings, and the role that strings have in presenting information. Then it relates the concept of languages to the notion of strings, and introduces grammars for characterizing languages. The chapter continues by introducing a class of programs. The choice is made for a class, which on one hand is general enough to model all programs, and on the other hand is primitive enough to simplify the specific investigation of programs. In particular, the notion of nondeterminism is introduced through programs. The chapter concludes by considering the notion of problems, the relationship between problems and programs, and some other related notions.

Chapter 2 studies finite-memory programs. The notion of a state is introduced as an abstraction for a location in a finite-memory program as well as an assignment to the variables of the program. The notion of state is used to show how finite-memory programs can be modeled by abstract computing machines, called finite-state transducers. The transducers are essentially sets of states with rules for transition between the states. The inputs that can be recognized by finite-memory programs are characterized in terms of a class of grammars, called regular grammars. The limitations of finite-memory programs, closure properties for simplifying the job of writing finite-memory programs, and decidable properties of such programs are also studied.

Chapter 3 considers the introduction of recursion to finite-memory programs. The treatment of the new programs, called recursive finite-domain programs, resembles that for finite-memory programs in

Chapter 2. Specifically, the recursive finite-domain programs are modeled by abstract computing machines, called pushdown transducers. Each pushdown transducer is essentially a finite-state transducer that can access an auxiliary memory that behaves like a pushdown storage of unlimited size. The inputs that can be recognized by recursive finite-domain programs are characterized in terms of a generalization of regular grammars, called context-free grammars. Finally, limitations, closure properties, and decidable properties of recursive finite-domain programs are derived using techniques similar to those for finite-memory programs.

Chapter 4 deals with the general class of programs. Abstract computing machines, called Turing transducers, are introduced as generalizations of pushdown transducers that place no restriction on the auxiliary memory. The Turing transducers are proposed for characterizing the programs in general, and computability in particular. It is shown that a function is computable by a Turing transducer if and only if it is computable by a deterministic Turing transducer. In addition, it is shown that there exists a universal Turing transducer that can simulate any given deterministic Turing transducer. The limitations of Turing transducers are studied, and they are used to demonstrate some undecidable problems. A grammatical characterization for the inputs that Turing transducers recognize is also offered.

Chapter 5 considers the role of time and space in computations. It shows that problems can be classified into an infinite hierarchy according to their time requirements. It discusses the feasibility of those computations that can be carried out in "polynomial time" and the infeasibility of those computations that require "exponential time." Then it considers the role of "nondeterministic polynomial time." "Easiest" hard problems are identified, and their usefulness for detecting hard problems is exhibited. Finally, the relationship between time and space is examined.

Chapter 6 introduces instructions that allow random choices in programs. Deterministic programs with such instructions are called probabilistic programs. The usefulness of these programs is considered, and then probabilistic Turing transducers are introduced as abstractions of such programs. Finally, some interesting classes of problems that are solvable probabilistically in polynomial time are studied.

Chapter 7 is devoted to parallelism. It starts by considering parallel programs in which the communication cost is ignored. Then it introduces "high-level" abstractions for parallel programs, called PRAM's, which take into account the cost of communication. It continues by offering a class of "hardware-level" abstractions, called uniform families of circuits, which allow for a rigorous analysis of the complexity of parallel computations. The relationship between the two classes of abstractions is detailed, and the applicability of parallelism in speeding up sequential computations is considered.

The motivation for adding this text to the many already in the field originated from the desire to provide an approach that would be more appealing to readers with a background in programming. A unified treatment of the subject is therefore provided, which links the development of the mathematical theories to the study of programs.

The only cost of this approach occurs in the introduction of transducers, instead of restricting the

attention to abstract computing machines that produce no output. The cost, however, is minimal because there is negligible variation between these corresponding kinds of computing machines.

On the other hand, the benefit is extensive. This approach helps considerably in illustrating the importance of the field, and it allows for a new treatment of some topics that is more attractive to those readers whose main background is in programming. For instance, the notions of nondeterminism, acceptance, and abstract computing machines are introduced here through programs in a natural way. Similarly, the characterization of pushdown automata in terms of context-free languages is shown here indirectly through recursive finite-domain programs, by a proof that is less tedious than the direct one.

The choice of topics for the text and their organization are generally in line with what is the standard in the field. The exposition, however, is not always standard. For instance, transition diagrams are offered as representations of pushdown transducers and Turing transducers. These representations enable a significant simplification in the design and analysis of such abstract machines, and consequently provide the opportunity to illustrate many more ideas using meaningful examples and exercises.

As a natural outcome, the text also treats the topics of probabilistic and parallel computations. These important topics have matured quite recently, and so far have not been treated in other texts.

The level of the material is intended to provide the reader with introductory tools for understanding and using formal specifications in computer science. As a result, in many cases ideas are stressed more than detailed argumentation, with the objective of developing the reader's intuition toward the subject as much as possible.

This book is intended for undergraduate students at advanced stages of their studies, and for graduate students. The reader is assumed to have some experience as a programmer, as well as in handling mathematical concepts. Otherwise no specific prerequisite is necessary.

The entire text represents a one-year course load. For a lighter load some of the material may be just sketched, or even skipped, without loss of continuity. For instance, most of the proofs in Section 2.6, the end of Section 3.5, and Section 3.6, may be so treated.

Theorems, Figures, Exercises, and other items in the text are labeled with triple numbers. An item that is labeled with a triple *i.j.k* is assumed to be the *k*th item of its type in Section *j* of Chapter i.

Finally, I am indebted to Elizabeth Zwicky for helping me with the computer facilities at Ohio State University, and to Linda Davoli and Sonia DiVittorio for their editing work. I would like to thank my colleagues Ming Li , Tim Long , and Yaacov Yesha for helping me with the difficulties I had with some of the topics, for their useful comments, and for allowing me the opportunities to teach the material. I am also very grateful to an anonymous referee and to many students whose feedback guided me to the current exposition of the subject.

[next] [front] [up]

Chapter 1 GENERAL CONCEPTS

Computations are designed for processing information. They can be as simple as an estimation for driving time between cities, and as complex as a weather prediction.

The study of computation aims at providing an insight into the characteristics of computations. Such an insight can be used for predicting the complexity of desired computations, for choosing the approaches they should take, and for developing tools that facilitate their design.

The study of computation reveals that there are problems that cannot be solved. And of the problems that can be solved, there are some that require infeasible amount of resources (e.g., millions of years of computation time). These revelations might seem discouraging, but they have the benefit of warning against trying to solve such problems. Approaches for identifying such problems are also provided by the study of computation.

On an encouraging note, the study of computation provides tools for identifying problems that can feasibly be solved, as well as tools for designing such solutions. In addition, the study develops precise and well-defined terminology for communicating intuitive thoughts about computations.

The study of computation is conducted in this book through the medium of programs. Such an approach can be adopted because programs are descriptions of computations.

Any formal discussion about computation and programs requires a clear understanding of these notions, as well as of related notions. The purpose of this chapter is to define some of the basic concepts used in this book. The first section of this chapter considers the notion of strings, and the role that strings have in representing information. The second section relates the concept of languages to the notion of strings, and introduces grammars for characterizing languages. The third section deals with the notion of programs, and the concept of nondeterminism in programs. The fourth section formalizes the notion of problems, and discusses the relationship between problems and programs. The fifth section defines the notion of reducibility among problems.

[next] [tail] [up]

1.1 Alphabets, Strings, and Representations

The ability to represent information is crucial to communicating and processing information. Human societies created spoken languages to communicate on a basic level, and developed writing to reach a more sophisticated level.

The English language, for instance, in its spoken form relies on some finite set of basic sounds as a set of primitives. The words are defined in term of finite sequences of such sounds. Sentences are derived from finite sequences of words. Conversations are achieved from finite sequences of sentences, and so forth.

Written English uses some finite set of symbols as a set of primitives. The words are defined by finite sequences of symbols. Sentences are derived from finite sequences of words. Paragraphs are obtained from finite sequences of sentences, and so forth.

Similar approaches have been developed also for representing elements of other sets. For instance, the natural number can be represented by finite sequences of decimal digits.

Computations, like natural languages, are expected to deal with information in its most general form. Consequently, computations function as manipulators of integers, graphs, programs, and many other kinds of entities. However, in reality computations only manipulate strings of symbols that represent the objects. The previous discussion necessitates the following definitions.

Alphabets and Strings

A finite, nonempty ordered set will be called an *alphabet* if its elements are *symbols* , or *characters* (i.e., elements with "primitive" graphical representations). A finite sequence of symbols from a given alphabet will be called a *string* over the alphabet. A string that consists of a sequence $a_1, a_2, \ldots, a_n$ of symbols will be denoted by the juxtaposition $a_1a_2 \ldots a_n$. Strings that have zero symbols, called *empty strings*, will be denoted by ϵ.

Example 1.1.1 $\Sigma_1 = \{a, \ldots, z\}$ and $\Sigma_2 = \{0, \ldots, 9\}$ are alphabets. abb is a string over Σ_1, and 123 is a string over Σ_2. ba12 is not a string over Σ_1, because it contains symbols that are not in Σ_1. Similarly, 314 . . . is not a string over Σ_2, because it is not a finite sequence. On the other hand, ϵ is a string over any alphabet.

The empty set Ø is not an alphabet because it contains no element. The set of natural numbers is not an alphabet, because it is not finite. The union $\Sigma_1 \cup \Sigma_2$ is an alphabet only if an ordering is placed on its symbols. □

An alphabet of cardinality 2 is called a *binary alphabet*, and strings over a binary alphabet are called *binary strings*. Similarly, an alphabet of cardinality 1 is called a *unary alphabet*, and strings over a unary alphabet are called *unary strings*.

The *length* of a string α is denoted $|\alpha|$ and assumed to equal the number of symbols in the string.

Example 1.1.2 {0, 1} is a binary alphabet, and {1} is a unary alphabet. 11 is a binary string over the alphabet {0, 1}, and a unary string over the alphabet {1}.

11 is a string of length 2, $|\epsilon| = 0$, and $|01| + |1| = 3$. □

The string consisting of a sequence α followed by a sequence β is denoted $\alpha\beta$. The string $\alpha\beta$ is called the *concatenation* of α and β. The notation α^i is used for the string obtained by concatenating i copies of the string α.

Example 1.1.3 The concatenation of the string 01 with the string 100 gives the string 01100. The concatenation $\epsilon\alpha$ of ϵ with any string α, and the concatenation $\alpha\epsilon$ of any string α with ϵ give the string α. In particular, $\epsilon\epsilon = \epsilon$.

If $\alpha = 01$, then $\alpha^0 = \epsilon$, $\alpha^1 = 01$, $\alpha^2 = 0101$, and $\alpha^3 = 010101$. □

A string α is said to be a *substring* of a string β if $\beta = \gamma\alpha\rho$ for some γ and ρ. A substring α of a string β is said to be a *prefix* of β if $\beta = \alpha\rho$ for some ρ. The prefix is said to be a *proper prefix* of β if $\rho \neq \epsilon$. A substring α of a string β is said to be a *suffix* of β if $\beta = \gamma\alpha$ for some γ. The suffix is said to be a *proper suffix* of β if $\gamma \neq \epsilon$.

Example 1.1.4 ϵ, 0, 1, 01, 11, and 011 are the substrings of 011. ϵ, 0, and 01 are the proper prefixes of 011. ϵ, 1, and 11 are the proper suffixes of 011. 011 is a prefix and a suffix of 011. □

If $\alpha = a_1 \ldots a_n$ for some symbols $a_1, \ldots, a_n$ then $a_n \ldots a_1$ is called the *reverse* of α, denoted α^{rev}. β is said to be a *permutation* of α if β can be obtained from α by reordering the symbols in α.

Example 1.1.5 Let α be the string 001. $\alpha^{rev} = 100$. The strings 001, 010, and 100 are the permutations of α. □

The set of all the strings over an alphabet Σ will be denoted by Σ^* . Σ^+ will denote the set $\Sigma^* - \{\epsilon\}$.

Ordering of Strings

Searching is probably the most commonly applied operation on information. Due to the importance of this operation, approaches for searching information and for organizing information to facilitate searching, receive special attention. Sequential search, binary search, insertion sort, quick sort, and merge sort are some examples of such approaches. These approaches rely in most cases on the existence of a relationship that defines an ordering of the entities in question.

A frequently used relationship for strings is the one that compares them alphabetically, as reflected by the ordering of names in telephone books. The relationship and ordering can be defined in the following manner.

Consider any alphabet Σ. A string α is said to be *alphabetically smaller* in Σ^* than a string β, or equivalently, β is said to be *alphabetically bigger* in Σ^* than α if α and β are in Σ^* and either of the following two cases holds.

a. α is a proper prefix of β.
b. For some γ in Σ^* and some a and b in Σ such that a precedes b in Σ, the string γa is a prefix of α and the string γb is a prefix of β.

An ordered subset of Σ^* is said to be *alphabetically ordered*, if β is not alphabetically smaller in Σ^* than α whenever α precedes β in the subset.

Example 1.1.6 Let Σ be the binary alphabet {0, 1}. The string 01 is alphabetically smaller in Σ^* than the string 01100, because 01 is a proper prefix of 01100. On the other hand, 01100 is alphabetically smaller than 0111, because both strings agree in their first three symbols and the fourth symbol in 01100 is smaller than the fourth symbol in 0111.

The set {ϵ, 0, 00, 000, 001, 01, 010, 011, 1, 10, 100, 101, 11, 110, 111}, of those strings that have length not greater than 3, is given in alphabetical ordering. □

Alphabetical ordering is satisfactory for finite sets, because each string in such an ordered set can eventually be reached. For similar reasons, alphabetical ordering is also satisfactory for infinite sets of unary strings. However, in some other cases alphabetical ordering is not satisfactory because it can result in some strings being preceded by an unbounded number of strings. For instance, such is the case for the string 1 in the alphabetically ordered set {0, 1}*, that is, 1 is preceded by the strings 0, 00, 000, . . . This deficiency motivates the following definition of canonical ordering for strings. In canonical ordering each string is preceded by a finite number of strings.

A string α is said to be *canonically smaller* or *lexicographically smaller* in Σ^* than a string β, or equivalently, β is said to be *canonically bigger* or *lexicographically bigger* in Σ^* than α if either of the

following two cases holds.

a. α is shorter than β.
b. α and β are of identical length but α is alphabetically smaller than β.

An ordered subset of Σ^* is said to be *canonically ordered* or *lexicographically ordered*, if β is not canonically smaller in Σ^* than α whenever α precedes β in the subset.

Example 1.1.7 Consider the alphabet $\Sigma = \{0, 1\}$. The string 11 is canonically smaller in Σ^* than the string 000, because 11 is a shorter string than 000. On the other hand, 00 is canonically smaller than 11, because the strings are of equal length and 00 is alphabetically smaller than 11.

The set $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$ is given in its canonical ordering. □

Representations

Given the preceding definitions of alphabets and strings, representations of information can be viewed as the mapping of objects into strings in accordance with some rules. That is, formally speaking, a *representation* or *encoding* over an alphabet Σ of a set D is a function f from D to 2^{Σ^*} that satisfies the following condition: $f(e_1)$ and $f(e_2)$ are disjoint nonempty sets for each pair of distinct elements e_1 and e_2 in D.

If Σ is a unary alphabet, then the representation is said to be a *unary representation*. If Σ is a binary alphabet, then the representation is said to be a *binary representation*.

In what follows each element in f(e) will be referred to as a representation, or encoding, of e.

Example 1.1.8 f_1 is a binary representation over {0, 1} of the natural numbers if $f_1(0) = \{0, 00, 000, 0000, \ldots\}$, $f_1(1) = \{1, 01, 001, 0001, \ldots\}$, $f_1(2) = \{10, 010, 0010, 00010, \ldots\}$, $f_1(3) = \{11, 011, 0011, 00011, \ldots\}$, and $f_1(4) = \{100, 0100, 00100, 000100, \ldots\}$, etc.

Similarly, f_2 is a binary representation over {0, 1} of the natural numbers if it assigns to the *i*th natural number the set consisting of the *i*th canonically smallest binary string. In such a case, $f_2(0) = \{\epsilon\}$, $f_2(1) = \{0\}$, $f_2(2) = \{1\}$, $f_2(3) = \{00\}$, $f_2(4) = \{01\}$, $f_2(5) = \{10\}$, $f_2(6) = \{11\}$, $f_2(7) = \{000\}$, $f_2(8) = \{1000\}$, $f_2(9) = \{1001\}$, . . .

On the other hand, f_3 is a unary representation over {1} of the natural numbers if it assigns to the *i*th natural number the set consisting of the *i*th alphabetically (= canonically) smallest unary string. In such a case, $f_3(0) = \{\epsilon\}$, $f_3(1) = \{1\}$, $f_3(2) = \{11\}$, $f_3(3) = \{111\}$, $f_3(4) = \{1111\}$, . . . , $f_3(i) = \{1^i\}$, . . .

The three representations f_1, f_2, and f_3 are illustrated in Figure 1.1.1. □

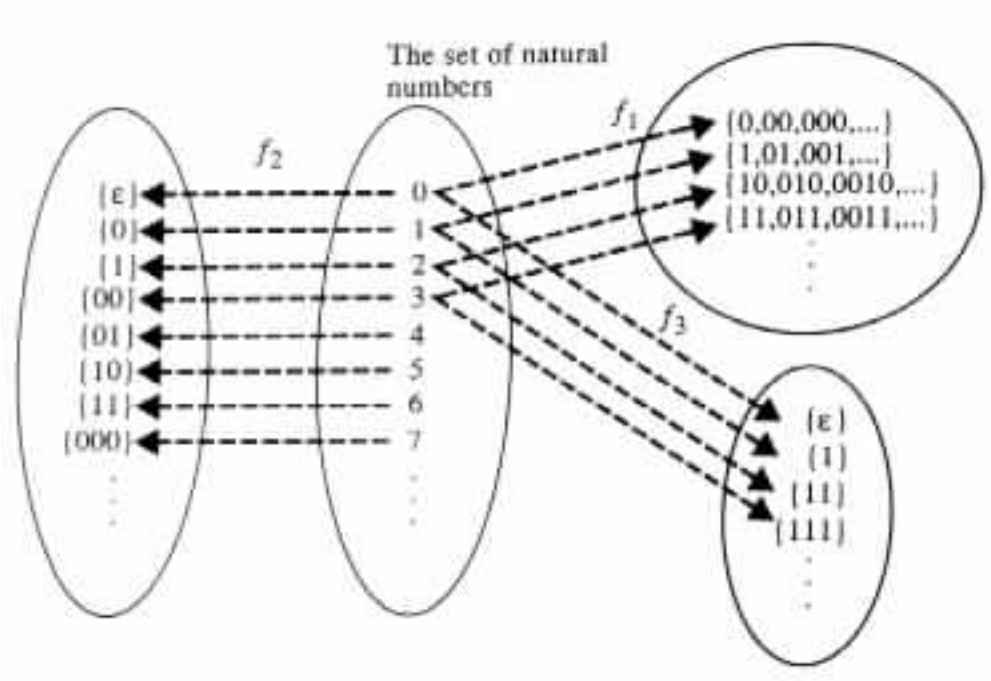

Figure 1.1.1 Representations for the natural numbers.

In the rest of the book, unless otherwise is stated, the function f_1 of Example 1.1.8 is assumed to be the binary representation of the natural numbers.

[next] [front] [up]

[next] [prev] [prev-tail] [tail] [up]

1.2 Formal Languages and Grammars

The universe of strings is a useful medium for the representation of information as long as there exists a function that provides the interpretation for the information carried by the strings. An interpretation is just the inverse of the mapping that a representation provides, that is, an *interpretation* is a function g from Σ^* to D for some alphabet Σ and some set D. The string 111, for instance, can be interpreted as the number one hundred and eleven represented by a decimal string, as the number seven represented by a binary string, and as the number three represented by a unary string.

The parties communicating a piece of information do the representing and interpreting. The representation is provided by the sender, and the interpretation is provided by the receiver. The process is the same no matter whether the parties are human beings or programs. Consequently, from the point of view of the parties involved, a language can be just a collection of strings because the parties embed the representation and interpretation functions in themselves.

Languages

In general, if Σ is an alphabet and L is a subset of Σ^*, then L is said to be a *language* over Σ, or simply a language if Σ is understood. Each element of L is said to be a *sentence* or a *word* or a *string* of the language.

Example 1.2.1 {0, 11, 001}, {ϵ, 10}, and {0, 1}* are subsets of {0, 1}*, and so they are languages over the alphabet {0, 1}.

The empty set Ø and the set {ϵ} are languages over every alphabet. Ø is a language that contains no string. {ϵ} is a language that contains just the empty string. □

The *union* of two languages L_1 and L_2, denoted $L_1 \cup L_2$, refers to the language that consists of all the strings that are either in L_1 or in L_2, that is, to { x | x is in L_1 or x is in L_2 }. The *intersection* of L_1 and L_2, denoted $L_1 \cap L_2$, refers to the language that consists of all the strings that are both in L_1 and L_2, that is, to { x | x is in L_1 and in L_2 }. The *complementation* of a language L over Σ, or just the

complementation of L when Σ is understood, denoted $\overline{L}$, refers to the language that consists of all the strings over Σ that are not in L, that is, to { x | x is in Σ^* but not in L }.

Example 1.2.2 Consider the languages $L_1 = \{\epsilon, 0, 1\}$ and $L_2 = \{\epsilon, 01, 11\}$. The union of these languages is $L_1 \cup L_2 = \{\epsilon, 0, 1, 01, 11\}$, their intersection is $L_1 \cap L_2 = \{\epsilon\}$, and the complementation of L_1 is $\overline{L_1} = \{00, 01, 10, 11, 000, 001, \ldots\}$.

$\emptyset \cup L = L$ for each language L. Similarly, $\emptyset \cap L = \emptyset$ for each language L. On the other hand, $\overline{\emptyset} = \Sigma^*$ and $\overline{\Sigma^+} = \emptyset$ for each alphabet Σ. □

The *difference* of L_1 and L_2, denoted $L_1 - L_2$, refers to the language that consists of all the strings that are in L_1 but not in L_2, that is, to { x | x is in L_1 but not in L_2 }. The *cross product* of L_1 and L_2, denoted $L_1 \times L_2$, refers to the set of all the pairs (x, y) of strings such that x is in L_1 and y is in L_2, that is, to the relation { (x, y) | x is in L_1 and y is in L_2 }. The *composition* of L_1 with L_2, denoted L_1L_2, refers to the language { xy | x is in L_1 and y is in L_2 }.

Example 1.2.3 If $L_1 = \{\epsilon, 1, 01, 11\}$ and $L_2 = \{1, 01, 101\}$ then $L_1 - L_2 = \{\epsilon, 11\}$ and $L_2 - L_1 = \{101\}$.

On the other hand, if $L_1 = \{\epsilon, 0, 1\}$ and $L_2 = \{01, 11\}$, then the cross product of these languages is $L_1 \times L_2 = \{(\epsilon, 01), (\epsilon, 11), (0, 01), (0, 11), (1, 01), (1, 11)\}$, and their composition is $L_1L_2 = \{01, 11, 001, 011, 101, 111\}$.

$L - \emptyset = L$, $\emptyset - L = \emptyset$, $\emptyset L = \emptyset$, and $\{\epsilon\}L = L$ for each language L. □

L^i will also be used to denote the composing of i copies of a language L, where L^0 is defined as $\{\epsilon\}$. The set $L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$, called the *Kleene closure* or just the *closure* of L, will be denoted by L^*. The set $L^1 \cup L^2 \cup L^3 \ldots$, called the *positive closure* of L, will be denoted by L^+.

L^i consists of those strings that can be obtained by concatenating i strings from L. L^* consists of those strings that can be obtained by concatenating an arbitrary number of strings from L.

Example 1.2.4 Consider the pair of languages $L_1 = \{\epsilon, 0, 1\}$ and $L_2 = \{01, 11\}$. For these languages $L_1{}^2 = \{\epsilon, 0, 1, 00, 01, 10, 11\}$, and $L_2{}^3 = \{010101, 010111, 011101, 011111, 110101, 110111, 111101, 111111\}$. In addition, ϵ is in $L_1{}^*$, in $L_1{}^+$, and in $L_2{}^*$ but not in $L_2{}^+$. □

The operations above apply in a similar way to relations in $\Sigma^* \times \Delta^*$, when Σ and Δ are alphabets. Specifically, the *union* of the relations R_1 and R_2, denoted $R_1 \cup R_2$, is the relation { (x, y) | (x, y) is in R_1 or in R_2 }. The *intersection* of R_1 and R_2, denoted $R_1 \cap R_2$, is the relation { (x, y) | (x, y) is in R_1 and in

R_2 }. The *composition* of R_1 with R_2, denoted R_1R_2, is the relation { (x_1x_2, y_1y_2) | (x_1, y_1) is in R_1 and (x_2, y_2) is in R_2 }.

Example 1.2.5 Consider the relations $R_1 = \{(\epsilon, 0), (10, 1)\}$ and $R_2 = \{(1, \epsilon), (0, 01)\}$. For these relations $R_1 \cup R_2 = \{(\epsilon, 0), (10, 1), (1, \epsilon), (0, 01)\}$, $R_1 \cap R_2 = \emptyset$, $R_1R_2 = \{(1, 0), (0, 001), (101, 1), (100, 101)\}$, and $R_2R_1 = \{(1, 0), (110, 1), (0, 010), (010, 011)\}$. □

The *complementation* of a relation R in $\Sigma^* \times \Delta^*$, or just the complementation of R when Σ and Δ are understood, denoted $\overline{R}$, is the relation { (x, y) | (x, y) is in $\Sigma^* \times \Delta^*$ but not in R }. The *inverse* of R, denoted R^{-1}, is the relation { (y, x) | (x, y) is in R }. $R^0 = \{(\epsilon, \epsilon)\}$. $R^i = R^{i-1}R$ for $i \geq 1$.

Example 1.2.6 If R is the relation $\{(\epsilon, \epsilon), (\epsilon, 01)\}$, then $R^{-1} = \{(\epsilon, \epsilon), (01, \epsilon)\}$, $R^0 = \{(\epsilon, \epsilon)\}$, and $R^2 = \{(\epsilon, \epsilon), (\epsilon, 01), (\epsilon, 0101)\}$. □

A language that can be defined by a formal system, that is, by a system that has a finite number of axioms and a finite number of inference rules, is said to be a *formal language*.

Grammars

It is often convenient to specify languages in terms of grammars. The advantage in doing so arises mainly from the usage of a small number of rules for describing a language with a large number of sentences. For instance, the possibility that an English sentence consists of a subject phrase followed by a predicate phrase can be expressed by a grammatical rule of the form <sentence> → <subject><predicate>. (The names in angular brackets are assumed to belong to the grammar metalanguage.) Similarly, the possibility that the subject phrase consists of a noun phrase can be expressed by a grammatical rule of the form <subject> → <noun>. In a similar manner it can also be deduced that "Mary sang a song" is a possible sentence in the language described by the following grammatical rules.

$$
\begin{aligned}
\langle sentence \rangle &\rightarrow \langle subject \rangle \langle predicate \rangle \\
\langle subject \rangle &\rightarrow \langle noun \rangle \\
\langle predicate \rangle &\rightarrow \langle verb \rangle \langle article \rangle \langle noun \rangle \\
\langle noun \rangle &\rightarrow \langle name \rangle \\
\langle noun \rangle &\rightarrow \langle string \rangle \\
\langle name \rangle &\rightarrow \langle u\_character \rangle \langle string \rangle \\
\langle string \rangle &\rightarrow \langle string \rangle \langle character \rangle \\
\langle string \rangle &\rightarrow \langle character \rangle \\
\langle character \rangle &\rightarrow a \\
&\vdots \\
\langle character \rangle &\rightarrow z \\
\langle u\_character \rangle &\rightarrow A \\
&\vdots \\
\langle u\_character \rangle &\rightarrow Z \\
\langle verb \rangle &\rightarrow sang \\
\langle article \rangle &\rightarrow a
\end{aligned}
$$

The grammatical rules above also allow English sentences of the form "Mary sang a song" for other names besides Mary. On the other hand, the rules imply non-English sentences like "Mary sang a Mary," and do not allow English sentences like "Mary read a song." Therefore, the set of grammatical rules above consists of an incomplete grammatical system for specifying the English language.

For the investigation conducted here it is sufficient to consider only grammars that consist of finite sets of grammatical rules of the previous form. Such grammars are called Type 0 grammars , or phrase structure grammars , and the formal languages that they generate are called Type 0 languages.

Strictly speaking, each *Type 0 grammar* G is defined as a mathematical system consisting of a quadruple <N, Σ, P, S>, where

N
: is an alphabet, whose elements are called *nonterminal* symbols.

Σ
: is an alphabet disjoint from N, whose elements are called *terminal* symbols.

P
: is a relation of finite cardinality on $(N \cup \Sigma)^*$, whose elements are called *production rules*. Moreover, each production rule (α, β) in P, denoted $\alpha \rightarrow \beta$, must have at least one nonterminal symbol in α. In each such production rule, α is said to be the *left-hand side* of the production rule, and β is said to be the *right-hand side* of the production rule.

S
: is a symbol in N called the *start* , or *sentence* , symbol.

Example 1.2.7 <N, Σ, P, S> is a Type 0 grammar if N = {S}, Σ = {a, b}, and P = {$S \rightarrow aSb$, $S \rightarrow \epsilon$}. By

definition, the grammar has a single nonterminal symbol S, two terminal symbols a and b, and two production rules $S \rightarrow aSb$ and $S \rightarrow \epsilon$. Both production rules have a left-hand side that consists only of the nonterminal symbol S. The right-hand side of the first production rule is aSb, and the right-hand side of the second production rule is ϵ.

$<N_1, \Sigma_1, P_1, S>$ is not a grammar if N_1 is the set of natural numbers, or Σ_1 is empty, because N_1 and Σ_1 have to be alphabets.

If $N_2 = \{S\}$, $\Sigma_2 = \{a, b\}$, and $P_2 = \{S \rightarrow aSb, S \rightarrow \epsilon, ab \rightarrow S\}$ then $<N_2, \Sigma_2, P_2, S>$ is not a grammar, because $ab \rightarrow S$ does not satisfy the requirement that each production rule must contain at least one nonterminal symbol on the left-hand side. □

In general, the nonterminal symbols of a Type 0 grammar are denoted by S and by the first uppercase letters in the English alphabet A, B, C, D, and E. The start symbol is denoted by S. The terminal symbols are denoted by digits and by the first lowercase letters in the English alphabet a, b, c, d, and e. Symbols of insignificant nature are denoted by X, Y, and Z. Strings of terminal symbols are denoted by the last lowercase English characters u, v, w, x, y, and z. Strings that may consist of both terminal and nonterminal symbols are denoted by the first lowercase Greek symbols α, β, and γ. In addition, for convenience, sequences of production rules of the form

$$\begin{array}{lcl} \alpha & \rightarrow & \beta_1 \\ \alpha & \rightarrow & \beta_2 \\ & \vdots & \\ \alpha & \rightarrow & \beta_n \end{array}$$

are denoted as

$$\begin{array}{lcl} \alpha & \rightarrow & \beta_1 \\ & \rightarrow & \beta_2 \\ & \vdots & \\ & \rightarrow & \beta_n \end{array}$$

Example 1.2.8 $<N, \Sigma, P, S>$ is a Type 0 grammar if $N = \{S, B\}$, $\Sigma = \{a, b, c\}$, and P consists of the following production rules.

$$\begin{array}{lcl} S & \rightarrow & aBSc \\ & \rightarrow & abc \\ & \rightarrow & \epsilon \\ Ba & \rightarrow & aB \\ Bb & \rightarrow & bb \end{array}$$

The nonterminal symbol S is the left-hand side of the first three production rules. Ba is the left-hand side of the fourth production rule. Bb is the left-hand side of the fifth production rule.

The right-hand side aBSc of the first production rule contains both terminal and nonterminal symbols. The right-hand side abc of the second production rule contains only terminal symbols. Except for the trivial case of the right-hand side ϵ of the third production rule, none of the right-hand sides of the production rules consists only of nonterminal symbols, even though they are allowed to be of such a form. □

Derivations

Grammars generate languages by repeatedly modifying given strings. Each modification of a string is in accordance with some production rule of the grammar in question G = <N, Σ, P, S>. A modification to a string γ in accordance with production rule $\alpha \rightarrow \beta$ is derived by replacing a substring α in γ by β.

In general, a string γ is said to directly derive a string γ' if γ' can be obtained from γ by a single modification. Similarly, a string γ is said to derive γ' if γ' can be obtained from γ by a sequence of an arbitrary number of direct derivations.

Formally, a string γ is said to *directly derive* in G a string γ', denoted $\gamma \Rightarrow_G \gamma'$, if γ' can be obtained from γ by replacing a substring α with β, where $\alpha \rightarrow \beta$ is a production rule in G. That is, if $\gamma = \rho\alpha\delta$ and $\gamma' = \rho\beta\delta$ for some strings α, β, ρ, and δ such that $\alpha \rightarrow \beta$ is a production rule in G.

Example 1.2.9 If G is the grammar <N, Σ, P, S> in Example 1.2.7, then both ϵ and aSb are directly derivable from S. Similarly, both ab and a^2Sb^2 are directly derivable from aSb. ϵ is directly derivable from S, and ab is directly derivable from aSb, in accordance with the production rule $S \rightarrow \epsilon$. aSb is directly derivable from S, and a^2Sb^2 is directly derivable from aSb, in accordance with the production rule $S \rightarrow aSb$.

On the other hand, if G is the grammar <N, Σ, P, S> of Example 1.2.8, then $aBaBabccc \Rightarrow_G aaBBabccc$ and $aBaBabccc \Rightarrow_G aBaaBbccc$ in accordance with the production rule $Ba \rightarrow aB$. Moreover, no other string is directly derivable from aBaBabccc in G. □

γ is said to *derive* γ' in G, denoted $\gamma \Rightarrow_G^* \gamma'$, if $\gamma_0 \Rightarrow_G \ldots \Rightarrow_G \gamma'_n$ for some $\gamma_0, \ldots, \gamma_n$ such that $\gamma_0 = \gamma$ and $\gamma_n = \gamma'$. In such a case, the sequence $\gamma_0 \Rightarrow_G \ldots \Rightarrow_G \gamma_n$ is said to be a *derivation* of γ from γ' whose *length* is equal to n. $\gamma_0, \ldots, \gamma_n$ are said to be *sentential forms*, if $\gamma_0 = S$. A sentential form that contains no terminal symbols is said to be a *sentence* .

Example 1.2.10 If G is the grammar of Example 1.2.7, then a^4Sb^4 has a derivation from S. The derivation $S \Rightarrow_G^* a^4Sb^4$ has length 4, and it has the form $S \Rightarrow_G aSb \Rightarrow_G a^2Sb^2 \Rightarrow_G a^3Sb^3 \Rightarrow_G a^4Sb^4$. □

A string is assumed to be in the language that the grammar G generates if and only if it is a string of terminal symbols that is derivable from the starting symbol. The language that is *generated* by G, denoted L(G), is the set of all the strings of terminal symbols that can be derived from the start symbol, that is, the set { w | w is in Σ^*, and $S \Rightarrow_G^* w$ }. Each string in the language L(G) is said to be generated by G.

Example 1.2.11 Consider the grammar G of Example 1.2.7. ϵ is in the language that G generates because of the existence of the derivation $S \Rightarrow_G \epsilon$. ab is in the language that G generates, because of the existence of the derivation $S \Rightarrow_G aSb \Rightarrow_G ab$. a^2b^2 is in the language that G generates, because of the existence of the derivation $S \Rightarrow_G aSb \Rightarrow_G a^2Sb^2 \Rightarrow_G a^2b^2$.

The language L(G) that G generates consists of all the strings of the form a ... ab ... b in which the number of a's is equal to the number of b's, that is, L(G) = { a^ib^i | i is a natural number }.

aSb is not in L(G) because it contains a nonterminal symbol. a^2b is not in L(G) because it cannot be derived from S in G. □

In what follows, the notations $\gamma \Rightarrow \gamma'$ and $\gamma \Rightarrow^* \gamma'$ are used instead of $\gamma \Rightarrow_G \gamma'$ and $\gamma \Rightarrow_G^* \gamma'$, respectively, when G is understood. In addition, Type 0 grammars are referred to simply as grammars, and Type 0 languages are referred to simply as languages , when no confusion arises.

Example 1.2.12 If G is the grammar of Example 1.2.8, then the following is a derivation for $a^3b^3c^3$. The underlined and the overlined substrings are the left- and the right-hand sides, respectively, of those production rules used in the derivation.

$$
\begin{aligned}
\underline{S} &\Rightarrow \overline{aBSc} \\
&\Rightarrow aB\overline{aBSc}c \\
&\Rightarrow aBa\underline{B}\overline{ab}cc \\
&\Rightarrow aBa\overline{a\underline{B}b}ccc \\
&\Rightarrow a\underline{B}a\overline{abb}ccc \\
&\Rightarrow a\overline{a\underline{B}a}bbccc \\
&\Rightarrow aa\overline{a\underline{B}b}bccc \\
&\Rightarrow aaa\overline{abb}bccc
\end{aligned}
$$

The language generated by the grammar G consists of all the strings of the form a ... ab ... bc ... c in which there are equal numbers of a's, b's, and c's, that is, L(G) = { $a^ib^ic^i$ | i is a natural number }.