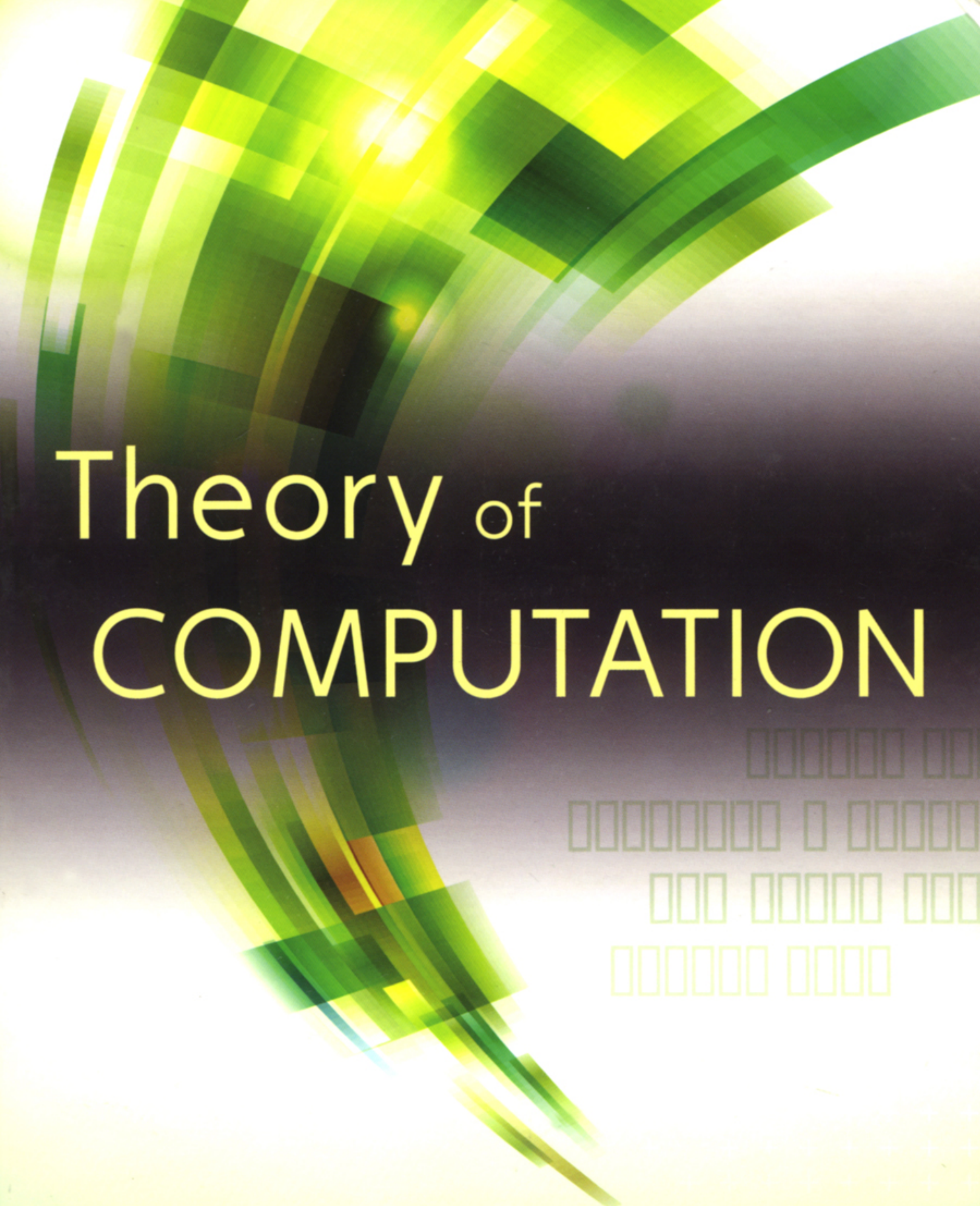

Theory of COMPUTATION

GEORGE TOURLAKIS

WILEY

Theory of Computation

Theory of Computation

George Tourlakis
York University
Toronto, Canada

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representation or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Tourlakis, George J.
Theory of computation / George Tourlakis.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-118-01478-3 (hardback)
1. Computable functions. 2. Functional programming languages. I. Title.
QA9.59.T684 2012
511.3'52—dc23 2011051088

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my parents

CONTENTS

Preface

At the intuitive level, any practicing mathematician or computer scientist —indeed any student of these two fields of study— will have no difficulty at all to recognize a *computation* or an *algorithm*, as soon as they see one, the latter defining, *in a finite manner*, computations for any given input. It is also an expectation that students of computer science (and, increasingly nowadays, of mathematics) will acquire the skill to devise algorithms (normally expressed as computer programs) that solve a variety of problems.

But how does one tackle the questions "is there an algorithm that solves such and such a problem for all possible inputs?" —a question with a potentially "no" answer— and also "is there an algorithm that solves such and such a problem via computations that take no more *steps* than some (fixed) polynomial function of the input length?" —this, too, being a question with a, potentially, "no" answer.

Typical (and tangible, indeed "interesting" and practically important) examples that fit the above questions, respectively, are

- "is there an algorithm which can determine whether or not a given computer program (the latter written in, say, the C-language) is *correct*?"[1]

[1] A "correct" program produces, *for every input*, precisely the output that is expected by an *a priori* specification.

and

- "is there an algorithm that will determine whether or not any given Boolean formula is a tautology, doing so via computations that take no more *steps* than some (fixed) polynomial function of the input length?"

For the first question we have a definitive "no" answer,[2] while for the second one we simply do not know, at the present state of knowledge and understanding of what "computing" means.[3]

But what do we mean when we say that "there is *no algorithm* that solves a given problem" —with or without restrictions on the algorithm's computation lengths? This appears to be a much harder statement to validate than "there *is* an algorithm that solves such and such a problem" —for the latter, all we have to do is *to produce such an algorithm* and a proof that it works as claimed. By contrast, the former statement implies a, mathematically speaking, *provably failed search* over the entire set of *all algorithms*, while we were looking for one that solves our problem.

One evidently needs a precise definition of the concept of algorithm that is neither experiential, nor technology-dependent in order to assert that we encountered such a failed "search". This directly calls for a *mathematical theory* whose objects of study include *algorithms* (and, correspondingly, *computations*) in order to construct such sets of (all) algorithms within the theory and be able to reason about the membership problem of such sets. This theory we call the *theory of computation*. It contains tools which, *in principle*, can "search"[4] the set of all algorithms to see whether a problem is solvable by one; or, more ambitiously, to see if it can be solved by an algorithm whose computations are "efficient" —under some suitable definition of efficiency.

The theory of computation is the *metatheory* of computing. In the field of computing one computes: that is, develops programs and large scale software that are well-

[2] There is some interesting "small print" here! As long as the concept of *algorithm* is identified with that of, say, the Shepherdson-Sturgis "machines" of this volume —or for that matter with Turing machines— then the answer is definitely a "no": There is a simple mathematical proof that we will see later on, that no Shepherdson-Sturgis machine (nor a Turing machine) exists that solves the problem. Now, such an identification has been advocated by Alonzo Church as part of his famous belief known as "Church's Thesis". If one accepts this identification, then the result about the non-existence of a Shepherdson-Sturgis machine that solves the problem is tantamount to the non-existence of an *algorithm* that does so. However, Church's "thesis" is empirical, rather than provable, and is not without detractors; cf. Kalmár (1957). Suffice it to say that *this* statement is mathematically valid: *No program, written in any programming language, which is equivalent in expressive power to that of our Shepherdson-Sturgis machines, exists that solves the problem.*

[3] There is substantial evidence that the answer, if discovered, will likely be "no".

[4] The quotes are necessary since it is not precisely a *search* that one performs. For example, the unsolvability —by any algorithm— of the *program correctness problem* is based on a so-called *reduction* technique that we will learn in this volume. A reduction basically establishes that a problem A is solvable by algorithmic means if we *assume* that we have a "black-box" algorithmic solution —that we may "call" just as we call a built-in function— of another problem, B. We say that "A is reduced (or reducible) to B". If we now *know* (say via a previous mathematical proof of the fact) that A cannot be algorithmically solved, then nor can B! We will, as a starting point, show the unsolvability by algorithmic means, certainly not by any Shepherdson-Sturgis machine, of a certain "prototype" problem, known as *the halting problem*, "$x \in K$?". This will be done by a technique akin to Cantor's *diagonalization*. After this, many reduction arguments are effected by showing that K is reducible to a problem A. This renders A unsolvable!

documented, correct, efficient, reliable and easily maintainable. In the (meta)theory of computing one tackles the fundamental questions of the *limitations of computing*, limitations that are intrinsic rather than technology-dependent.[5] These limitations may rule out outright the existence of algorithmic solutions for some problems, while for others they rule out efficient solutions.

Our approach is anchored on the concrete (and assumed) practical knowledge about general computer programming attained by the reader in a first year programming course, as well as the knowledge of discrete mathematics at the same level. The next natural step then is to develop the metatheory of *general computing*, building on the computing experience that we have assumed the reader attained. This will be our chapter on computability, that is, the most general *meta*theory of computing. We develop this metatheory via the programming formalism known as Shepherdson-Sturgis *Unbounded Register Machines* (URM) —which is a straightforward abstraction of modern high level programming languages. Within that chapter we will also explore a restriction of the URM programming language, that of the *loop programs* of A. Meyer and D. Ritchie. We will learn that while these loop programs can only compute a very small subset of "all the computable functions", nevertheless are *significantly more than adequate* for programming solutions of any "practical", computationally solvable, problem. For example, even restricting the nesting of loop instructions to *as low as two*, we can compute —in principle— enormously large functions, which with input x can produce outputs such as

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{x}}}}}\right\}10^{350000}\text{ 2's} \tag{1}$$

The qualification above, "in principle", stems from the enormity of the output displayed in (1) —even for the input $x = 0$— that renders the above function way beyond "practical".

The chapter —after spending considerable care in developing the technique of *reductions*— concludes by demonstrating the intimate connection between the *unsolvability phenomenon* of computing on one hand, and the *unprovability phenomenon* of proving within first-order logic (cf. Gödel (1931)) on the other, when the latter is called upon to reason about "rich" theories such as (Peano's) arithmetic —that is, the theory of natural numbers, equipped with: the standard operations (plus, times); relations (less than); as well as with the principle of mathematical induction.

What to include and what not to include in an introductory book on the theory of computation is a challenge that, to some extend, is resolved by the preferences of the author. But I should like to think that the choices of topics made in this volume are more rational than simply being manifestations of "preference".

The overarching goal is to develop for the reader a "first-order" grounding in the fundamentals, that is, the theoretical limitations of computing in its various models of computation, from the most general model —the URM— down to the finite automaton.

[5] However this metatheory is called by most people "theory". Hence the title of this volume.

We view the technique of *reductions* as fundamental in the analysis of limitations of computing, and we spend a good deal of space on this topic, a variant of which (polynomial-time reductions) the student of computer science will encounter in Subsection 5.1.2 and will re-encounter in later studies as well, for example, in a course on algorithms and complexity. On the other hand, we do not hesitate to omit combinatorial topics such as "Post's correspondence problem", which only leads to specialized results (e.g., the algorithmic unsolvability of detecting ambiguity in context free languages) that we feel embody a less fundamental technical interest. Our emphasis is on laying the foundational tools and concepts that allow us to carry out a mathematical analysis of, and acquire a thorough understanding of, theoretical limitations of computing in both their absolute manifestation (uncomputability) and also in their relative manifestation (complexity and "intractability").

Consistent with our stated goal and emphasis, we purposely give short shrift to the area of so-called "positive" results, apart from a few familiarization examples of "programming" with URMs, loop programs, FA, NFA, and PDA. This is not a course about writing algorithms, but mostly about what algorithms *cannot do at all* and about what *they have a lot of trouble doing*. For example, results of Chapter 5 immediately imply that, in general, FORTRAN-like programs that allow nesting of the loop instruction equal to just *three* have highly *impractical* run times; certainly as high as[6]

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}x\ 2\text{'s}$$

Thus, we leave out "applications" such as lexical scanners via finite automata; automata-minimization; parsing of context free languages using LL, LR, recursive-descend, and other parsers; and defer them to a later course on *compiler writing tools* —these topics do not belong here. We would rather concentrate on what is *foundationally* important and omit what is not.

Another challenge is where to *start* building this metatheory. What should be our abstraction of a computer program? It should be a straightforward observation that since this metatheory, or "theory" as we nickname it, *abstracts* computing practices —in order to analyze and study said abstractions mathematically— the student must have encountered in the first instance *the concrete counterparts* of these *abstractions* for the latter to make any sense.

It is hardly the case that, prior to the second year of study, students have "programmed" scanners or parsers. Rather, students have programmed solutions for less specialized problems, using a *high level general purpose* language such as C/C++, Java, possibly Pascal, etc. They never programmed an automaton, a push-down automaton, or anything like a Turing machine (unless they have taken up machine language in the first year).

Yet the overwhelming majority of the literature develops the "theory of computation", in a manner of speaking, backwards —invariably starting with the theory of

[6] See 5.2.0.47 and 5.2.0.49. L_3 programs have run times bounded by Ackermann's $A_3^k(x)$, for some $k > 0$.

finite automata, as if automata is precisely what the reader was programming in his[7] first university course on programming. We apply this principle: Before the student studies the (meta)theory, he must have attained a good grasp of the *practice* that this theory attempts to dissect and discuss. Thus, it is natural to start our story with the (meta)theory of *general purpose computer programs*.

Because of these considerations, our first chapter is on URMs and computability. The choice of URMs as an abstraction of general-purpose computing —a relative latecomer (cf. Shepherdson and Sturgis (1963)) in the search for a good answer to "what would be a good technology-independent model of computation?"— also connects well with the experience of the student who will come to this course to learn what makes things tick in programming, and why some things do not tick at all. He most likely learned his programming via a high level language like C or Java rather than through *machine language*. The ubiquitous Turing machine (Turing (1936, 1937)) is more like machine language, indeed, is rather even less user-friendly.[8] It offers no advantage at this level of exposition, and rather presents an obscure and hard-to-use (and hard to "arithmetize"[9]) model of computation that one *need not* use as the *basis* of computability. On the other hand it lends itself well to certain studies in complexity theory and is an eminently usable tool in the proof of Cook's theorem (cf. Subsection 5.1.3). So we will not totally avoid the Turing machine!

We turn to the formulaic topics of a book on *Automata and Languages* — Chapter 3— only after we become familiar, to some extent, with the (general) computability theory, including the special computability theory of more "practical" functions, the primitive recursive functions. *Automata* are introduced as a very restricted *programming formalism*, and their limitations (in expressivity) and their associated languages are studied.

It is often said, with justification, that a course in theory of computation has as side-effect the firming up of the student's grasp of (discrete) mathematical techniques and mathematical reasoning, as well as the ability to apply such techniques in computer science and beyond. Of course, it cannot be emphasized enough that the student of a theory of computation course must be equipped already with the knowledge expected to be acquired by the successful completion of a one-semester course on discrete mathematics. This required background knowledge is often encapsulated, retold, and aspects of it are emphasized, in the space of a few pages at the front-end of a book like this. This is the ubiquitous "Chapter 0" of many books on the subject. In the case of the present book I would like, most of all, to retell two stories, *logic* and *induction*, that I often found being insufficiently developed in the student's "toolbox", notwithstanding earlier courses he may have taken. Thus, in Subsection 1.1.1 we develop the notational and how-to parts of elementary predicate logic in the space of some 20 pages, paying special attention to correctness of exposition. Section 1.4 presents the induction principle on the natural numbers in two steps: One, how

[7] Pronouns such as "he", "his", "him" are, *by definition*, gender-neutral in this volume and are used solely for textual convenience.

[8] Machine language can manipulate *numbers*, whereas a Turing machine can only manipulate *digits*!

[9] This verb will make sense later.

to use its various forms, and a proof of their equivalence to the least (positive) integer principle. Two, we argue, at the intuitive level, why induction *must* be a *valid principle* after all![10] We also go over concepts about sets and related notation, as well as relations and functions, very quickly since they do not need much retelling. We will also introduce quickly and in an elementary fashion a topic likely not encountered by the reader in the typical "discrete math" course: the distinction between two infinities —*countable* and *uncountable*— so that we can have an excuse to introduce the reader to Cantor's ingenious (and simple) *diagonalization* argument, that recurs in one or another shape and form, over and over, in the computability and complexity part of the theory of computation.

On intuitive arguments; "formalization" and why a course in theory cannot be taught exclusively *by hand-waving*: The main reason that compels us to teach (meta)theory in a computer science curriculum is not so much to prevent the innocent from trying to program a solution for the halting problem (cf. 2.5.0.16), just as we do not teach courses in geometry just to prevent circle-squaring "research". Rather, formal mathematical methods used in a course in the theory of computation, more so than the results themselves, are transferable skills that the student becomes endowed with, which equip him to model and mathematically analyze concrete phenomena that occur in computation, and through a mathematical process of reasoning to be able to recognize, understand, and correlate such phenomena. These formal methods, skills and results, put the "science" keyword into *computer science*.

Intuition, obtained through experience, is invaluable, of course, and we often argue intuitively *before* we offer a proof of a fact. But: one *cannot* have "proof-by-intuition".

We have included in this volume a good amount of *complexity theory* that will likely be mostly skipped whenever the book is called upon to serve a second year course on the theory of computation. There are a few "high level complexity" results already in Section 2.7 using diagonalization (cf. 2.7.1.9 and 2.7.1.11). Later, quite a bit is developed in Chapter 5, including the concept of $\mathcal{NP}$-completeness and Cook's theorem; an account of Cobham's class of *feasibly computable functions* (mostly delegated to the Exercises section, 5.3); and some elements of the hierarchy theory of the primitive recursive functions culminating in the rather startling fact that we cannot algorithmically solve the *correctness problem* of FORTRAN-like programs even if we restrict the *nesting of loops* to just *two levels*. FORTRAN-like languages have as abstract counterpart the *loop programs* of Meyer and Ritchie (1967) that we study in the chapters on computability (2nd) and complexity (5th).

Were I to use this book in a second year course in the theory of computation I would skim quickly over the mathematical "prerequisites" chapter, and then cover 2.1–2.7, parts of 2.10, certainly Gödel's incompleteness theorem and its relation to *uncomputability*: 2.11 —but not 2.11.1. I would then cover only as much as time permits from Chapter 3 on finite automata; certainly the pumping lemma, consistent

[10] In so doing I will be sure to let the student know that I am not squaring the circle: Induction is not a provable principle of the arithmetic of Peano, it is an axiom. However, this will not stop us from arguing its *plausibility*, i.e., why it is a *reasonable*, "natural" axiom.

with my view that this is a "course" about what *cannot* be done, or cannot be done "easily", rather than a toolbox for how to *do* things. The latter is deferred to a course and book on algorithms.

In a more advanced course where one can proceed faster, I would want also to cover the sections on creative sets and the recursion theorem, and also as much complexity theory as possible from Chapter 5, starting with the material leading to Cook's theorem.

The reader will forgive the many footnotes, which some will assess as bad style! There is always a story within a story, the "... and another thing ... ", that is best delegated to footnotes.

The style of exposition that I prefer is informal and conversational and is expected to serve well not only the readers who who have the guidance of an instructor, but also those readers who wish to learn the elements of the theory of computation on their own. I use several devices to promote understanding, such as frequent "pauses" that anticipate questions and encourage the reader to rethink an issue that might be misunderstood if read but not studied and reflected upon. Additionally, I have included numerous remarks, examples and embedded exercises (the latter in addition to the end-of-chapter exercises) that reflect on a preceding definition or theorem. All pauses are delimited by "**Pause.**" and ◀

The stylized "winding road ahead" warning, ☡, that I first saw in Bourbaki's books (Bourbaki (1966)) and have used in my other books, delimits a passage that is too *important* to skim over.

On the other hand, I am using ☡☡ to delimit passages that I could not resist including, but, frankly, can be skipped (unless you are curious).

There are over 200 end-of-chapter exercises and 41 embedded ones. Many have hints and thus I refrained from (subjectively) flagging them for level of difficulty. After all, as one of my mentors, Alan Borodin, used to say to us (when I was a graduate student at the University of Toronto), "attempt all exercises; but definitely do the ones you cannot do".

GEORGE TOURLAKIS

Toronto
November 2011

CHAPTER 1

MATHEMATICAL FOUNDATIONS

In this chapter we will briefly review tools, methods and *notation* from mathematics and logic, which we will directly apply throughout the remaining of this volume.

1.1 SETS AND LOGIC; NAÏVELY

The most elementary elements from "set theory" and logic are a good starting point for our review. The quotes are necessary since the term *set theory* as it is understood today applies to the *axiomatic* version, which is a vast field of knowledge, methods, tools and research [cf. Shoenfield (1967); Tourlakis (2003b)]—and this is not what we outline here. Rather, we present the standard notation and the elementary operations on sets, on one hand, and take a brief look at infinity and the *diagonal method* of Cantor's, on the other. Diagonalization is a tool of significant importance in computability. The tiny fragment of concepts from set theory that you will find in this section (and then see them applied throughout this volume) are framed within Cantor's original "naïve set theory", good expositions of which (but far exceeding our needs) can be found in Halmos (1960) and Kamke (1950).

We will be forced to interweave our exposition of concepts from set theory with concepts—and notation—from elementary logic, since all mathematics is based on

Theory of Computation. By George Tourlakis

logical deductions, and the vast majority of the literature, from the most elementary to the most advanced, employs logical notation; e.g., symbols such as "$\forall$" and "$\exists$".

The term "set" is *not defined,*[11] in either the modern or in the naïve Cantorian version of the theory. Expositions of the latter, however, often ask the reader to think of a *set* as just a synonym for the words "class",[12] "collection", or "aggregate". Intuitively, a set is a "container" along with its contents—its *elements* or *members*. Taken together, contents and container, are viewed as a *single mathematical object*. In mathematics one deals only with sets that contain mathematical objects (so we are not interested in sets of mice or fish).

Since a set is itself an *object*, a set may contain sets as elements.

All the reasoning that one does in order to develop set theory—even that of the naïve variety—or any part of mathematics, including all our reasoning in this book, utilizes mathematical logic. Logic is the mathematics of reasoning and its "objects" of study are predominantly mathematical "statements" or "assertions"—technically known as *formulae*[13]—and mathematical proofs. Logic can be applied to mathematics either experientially and informally—learned via practice as it were—or formally. The predominance of mathematical writings apply logic informally as a vehicle toward reaching their objectives.[14] Examples of writings where logic is formally applied to mathematics are the volumes that Bourbaki wrote, starting here [Bourbaki (1966)]. More recent examples at the undergraduate and graduate levels are Gries and Schneider (1994) and Tourlakis (2003b) respectively.

In this volume we apply logic informally. An overview is provided in the next subsection.

1.1.1 A Detour via Logic

As is customary in mathematics, we utilize *letters*, upper or lower case, usually from near the end of the alphabet (u, v, y, x, z, S, T, V) to *denote*, that is, to *name* mathematical objects—in particular, *sets*.

By abuse of language we say that u, v, y, x, z, S, T, V *are* (rather than *denote* or *name*) objects. These letters function just like the variables in algebra do; they are *object-variables*.

[11] The reader who has taken Euclidean geometry in high school will be familiar with this parallel: The terms "point", "line", and "plane" are not defined either, but we get to know them intimately through their properties that we develop through mathematical proofs, starting from Euclid's axioms.

[12] In axiomatic set theory a "class" is a kind of collection that may be so "large" that it technically fails to be a set. The axioms force sets to be "small" classes.

[13] More accurately, a "statement" and a formula are two different things. However, the latter mathematically "encodes" the former.

[14] Despite the dangers this entails, as Gödel's incompleteness theorems exposed [Gödel (1931)], modern mathematicians are confident that their subject and tools have matured enough, to the point that one can safely apply logic, once again, post-Gödel, informally. For example, Kunen states in his article on set-theoretic *combinatorics*, Kunen (1978), "A knowledge of [formal] logic is neither necessary, nor even desirable".

As is the case in algebra, the variables x, y, z are not the only objects set theory studies. It also studies numbers such as $0, 1, -7$ and π, matrices such as $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ and objects that are the results of function applications such as 7^{23000}, x^{y^z} and 2^x.

Unlike axiomatic set theory, which introduces its objects via formal constructions, naïve set theory allows us to use, "off the shelf", all the mathematical objects such as the above, as well as, of course, objects that are sets such as $\{2, 3, \{1\}\}$ and $A \cup B$.[15]

Logicians like to call mathematical objects *terms*. We utilize in this book the generic names t and s (with primes or subscripts, whenever we need more than two such names) to refer to arbitrary terms that we do not want to be specific about.

1.1.1.1 Definition. The *simplest* possible relations of set theory are of just *two forms*: $t \in s$—read "t is a member of s" or "t belongs to s"—and $t = s$, read "t is equal to s", where, as we indicated above, t and s are any terms whatsoever.

These relations are the *atomic formulae* (of set theory). The qualifier "atomic" refers to two facts:

- These two types cannot be expressed (simulated) in terms of simpler relations by using the notation and tools of logic.
- Using these two relations as building blocks we can *construct* every possible formula of set theory as we will explain shortly. □

1.1.1.2 Example. $x \in y$, $u = v$, $z \in S$ and $3 \in z$ and $2^z = y^3$ are atomic formulae.

$\mathbb{N}$, the set of all *natural numbers* (i.e., all the numbers that we obtain by starting at 0 and repeatedly adding 1: $0, 1, 2, 3, 4, \ldots$), is an important constant in naïve set theory.

By "$\mathbb{N}$ ... *is* an important constant" we mean, of course, via the habitual abuse of language exercised by mathematicians, the accurate "$\mathbb{N}$ ... *denotes* (or *names*) an important constant".

Here is an example that uses $\mathbb{N}$ in an atomic formula: $-7 \in \mathbb{N}$. Incidentally, this formula makes (i.e., encodes) a false statement; we say the *formula is false*.

One may form this basic formula as well, $\mathbb{N} = \bigcup_{i=0}^{\infty}\{i\}$, where the meaning of the symbols "$\{\ldots\}$" and "$\bigcup_{i=0}^{\infty}$" will be introduced later in this section.

Yet another example is $\{1\} \in \{2, 1\}$—a false statement (formula) as we will be able to determine soon. □

Logic (and mathematics) contain much more complex formulae than those of the atomic variety. The added complexity is achieved by repeatedly "gluing" atomic formulae together employing as glue the logical, or *Boolean*, *connectives*

$$\neg, \wedge, \vee, \rightarrow, \equiv$$

[15] Notation for objects such as $\{\ldots\}$ and $x \cup y$ will be reviewed shortly.

and the *quantifiers*

$$\forall, \exists$$

As we have noted already, unlike the case of naïve set theory—where we take for granted the *a priori* presence of *all* objects of mathematics, such as 3, -7, $\mathbb{N}$ and $x^{y^{z^u}}$ — axiomatic set theory needs no *a priori* existence of any objects. Starting just with the relations $x \in y$ and $x = y$ it uses powerful rules, which can be used to build not only all formulae of set theory, but also all the objects of mathematics that we are familiar with, such as the above-mentioned and many others.

What about arithmetic? The arithmetical objects of "pure" (Peano) arithmetic are the variables, constants, and outputs of functions applied on objects that we have already built. What are its formulae? If we are thinking of pure arithmetic, which is studied outside set theory, then we may choose as atomic formulae all those that can be built from the three start-up relations $z = x + y$, $z = x \times y$ and $z = x^y$: new atomic formulae result by substituting arbitrary (arithmetical) objects for variables. Note that the equality relation is obtained from $z = x + y$ by substituting 0 for y.

All formulae of arithmetic can be built, starting from the atomic ones, as explained in the general Definition 1.1.1.3 below. This assertion is revisited in Subsection 2.11.1.

Gödel showed in Gödel (1931) that the atomic formula $z = x^y$ is, well, *not atomic*: It can be simulated (built) within pure arithmetic starting just with $z = x + y$ and $z = x \times y$.

The "practicing mathematician" prefers to work within an "impure" arithmetic, where he has access to sets and their notations, operations, and properties. In particular, this impure arithmetic employs set variables and, more generally, set objects in addition to number variables and number objects.

Throughout this volume a formula (whether specific to set theory or to any other area in mathematics, such as arithmetic—pure or impure) will be *denoted* by an upper case calligraphic letter, such as $\mathscr{A}, \mathscr{B}, \mathscr{F}, \mathscr{G}$.

We now indicate how formulae are put together using brackets, connectives, and quantifiers, employing atomic formulae as basic building blocks. The definition below is generic, thus unified: it applies to the structure of all formulae of mathematics. The choice of atomic formulae (which presupposes an *a priori* choice of mathematical symbols, such as $0, +, \in$) and of types of variables is what determines whether we build set theory formulae, pure or impure arithmetic formulae, or "other".

1.1.1.3 Definition. A set theory formula is one of:

(1) An atomic formula (1.1.1.1).

(2) $(\neg\mathscr{A})$, where $\mathscr{A}$ is known to be[16] a formula.

[16] I.e., to stand for one. Thus, the expression "$(\neg\mathscr{A})$" is constructed by writing "(", followed by writing "$\neg$", followed by *writing in full* whatever $\mathscr{A}$ names, and finally writing ")".

(3) $(\mathscr{A} \wedge \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(4) $(\mathscr{A} \vee \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(5) $(\mathscr{A} \rightarrow \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(6) $(\mathscr{A} \equiv \mathscr{B})$, where $\mathscr{A}$ and $\mathscr{B}$ are known to be formulae.

(7) $((\forall x)\mathscr{A})$, where $\mathscr{A}$ is known to be a formula and x is any variable.

(8) $((\exists x)\mathscr{A})$, where $\mathscr{A}$ is known to be a formula and x is any variable. We say in the last two cases that "$\mathscr{A}$ is the *scope* of Qx, where Q is $\forall$ or $\exists$".

We call $\forall$ the *universal* and $\exists$ the *existential* quantifiers. We will extend the terminology "quantifier" to apply to the compound symbols $(\forall x)$ or $(\exists)$. □

1.1.1.4 Definition. (Immediate Predecessors) Let $\mathscr{F}$ be a formula. By 1.1.1.3 it has one of the forms (1)–(8). If it is of type (1), then it has no *immediate predecessors*—i.e., it was not built using connectives or quantifiers from simper formulae. If it has the forms (2)–(8), then in each case its immediate predecessors are the formulae $\mathscr{A}$ and $\mathscr{B}$ [the latter enters in cases (3)–(6)] that were used to build it. We use the acronym *ip* for *immediate predecessors*. □

The presence of brackets guarantees that the decomposition or deconstruction of a formula into its immediate predecessors is unique. This fact can be proved, but it is beyond our aims so we will not do so here [see Bourbaki (1966); Enderton (1972); Tourlakis (2008, 2003a)]. Logicians refer to it as the *unique readability* of a formula.

1.1.1.5 Example. Here are some formulae:

$x \in y$, $3 = z$, $z = x^w$—by (1),

$(\neg x = y)$—by (1), followed by an application of (2); we usually write this more simply as "$x \neq y$",

$(x \in y \vee z = x^w)$—by (1), followed by an application of (4),

$((\forall x)z = x^w)$—by (1), followed by an application of (7),

$(x = 0 \rightarrow x = 0)$—by (1), followed by an application of (5), and

$(x = 0 \rightarrow ((\forall x)x = 0))$—by (1), followed by an application of (7) to obtain $((\forall x)x = 0)$, and then by an application of (5).

The reader should check that we inserted brackets precisely as prescribed by Definition 1.1.1.3. □

1.1.1.6 Remark. (Building a formula) If $\mathscr{F}$ is (stands for, that is) a formula we can deconstruct it according to Definition 1.1.1.3 using a natural process.

Initialize: Write down $\mathscr{F}$. Flag it *pending*.
Repeat this process *until it cannot be carried further*:
{

Write down, *above* whatever you have written so far, the *ip* of all *pending* formulae (if they have ip); and *remove* the flag "pending" from the latter. *Add* the flag to the ones you have just written.

}

The process is terminating since we write *shorter and shorter formulae* at every step (*and* remove the flags); we cannot do this forever!

Clearly, if we now review *from top to bottom* the sequence that we wrote, we realize that it traces forward the process of constructing $\mathscr{F}$ by repeated application of Definition 1.1.1.3. This top-down view of our "deconstruction" is a *formula-construction sequence* for $\mathscr{F}$.

For example, applying the process to the last formula of the preceding example we get:

$x = 0$
$x = 0$
$((\forall x)x = 0)$
$(x = 0 \to ((\forall x)x = 0))$

where one copy of $x = 0$ was contributed by the bottom formula and the other (at the top) by $((\forall x)x = 0)$.

Going forward we can discard copies that we do not need. Thus a valid formula construction is also this one:

$x = 0$
$((\forall x)x = 0)$
$(x = 0 \to ((\forall x)x = 0))$

Indeed, we validate the first formula in the sequence via (1) of 1.1.1.3; the second using the first and (7); and the last one using the first two and (5). □

A term such as x^2 has x as its only input variable. An atomic formula such as $z \in \mathbb{N}$ has z as its only input variable, while the (atomic) formula $x + y = y^w$ has x, y and w as input variables. Whenever we want to draw attention to the input variables—say, x, u, S and z—of a term t or a formula $\mathscr{A}$ we will write $t(x, u, S, z)$ or $\mathscr{A}(x, u, S, z)$, respectively. This is entirely analogous to writing "$f(x, z) = x^2 + \sin z$" in order to name the expression (term) $x^2 + \sin z$ as a function $f(x, z)$ of the two listed variables.

1.1.1.7 Definition. (Input Variables—in Terms) All the variables that occur in a term—other than an x that occurs in a term of the form $\{x : \ldots\}$ (which is a set object that will be introduced shortly)—are input variables. □

1.1.1.8 Example. Thus, the term x has x as its only input variable; while the term 3 has no input variables. $x^{2^{z^y}}$ has x, z, y as its input variables. We will soon introduce terms (set objects) such as $\{x : x = 0\}$. This object, which the reader may recognize as a fancy way to simply write $\{0\}$, has no input variables. □

1.1.1.9 Definition. (Input Variables—in Formulae) A variable occurrence[17] in an atomic formula $t \in s$ or $t = s$ is an *input occurrence* precisely if it is an input occurrence in one of the terms t and s. Thus, "$0 \in \{x : x = 0\}$" has no input variables while "$x = 0$" has one.

Formation rules (2)–(6) in Definition 1.1.1.3 "*preserve*" the input occurrences of variables in the constituent formulae $\mathscr{A}$ and $\mathscr{B}$ that we join together using one of $\neg, \wedge, \vee, \rightarrow, \equiv$ as glue. On the other hand, each quantifier $(\forall z)$ or $(\exists z)$ *forces* each occurrence of a variable as described below to become non-input:

- The occurrence z in the quantifier
- Any occurrence of z in the scope of said $(\forall z)$ or $(\exists z)$

Thus, if we start with $\mathscr{A}(x, y, z)$, of inputs x, y, z, the new formula $((Qy)\mathscr{A}(x, y, z))$, where Q stands here for one of $\forall, \exists$, has only x and z as input variables. □

We have carefully referred to *occurrences*, rather than *variables*, in the above definition. A *variable can* be both input and non-input. An *occurrence cannot* be both. For example, in $(x = 0 \rightarrow (\forall x)x = 0)$ the first x-occurrence is input; the last two are non-input. The *variable* x is both.

Thus "x is an input/non-input variable" (of a formula) means that there are occurrences of x that are input/non-input.

The standard name utilized in the literature for input variables is *free variables*. Non-input variable occurrences are technically called *bound occurrences*, but are also called *apparent occurrences*, since even though they are visible, they are not allowed—indeed it makes no sense—to receive arguments (input). This is analogous to the "Σ-notation" for sums: $\sum_{i=1}^{3} i$ means $1 + 2 + 3$. While we can "see" the variable i, it is not really there![18] It cannot accept inputs. For example, "$\sum_{2=1}^{3} 2$" is total nonsense.

The jargon input/non-input *is deliberately chosen: We may substitute terms* only *in those variable occurrences that are free (input).*

If $\mathscr{F}$ is some formula and $x, y, z, \ldots$ is the *complete* list of variables that occur in it, we can draw attention to this fact by writing $\mathscr{F}(x, y, z, \ldots)$. If $x, y, z, \ldots$ is a list of variables such that *some*[19] among them occur in $\mathscr{F}$, then we indicate this by $\mathscr{F}[x, y, z, \ldots]$.

In the context of $\mathscr{F}[x, y, z, \ldots]$ [or $\mathscr{F}(x, y, z, \ldots)$], $\mathscr{F}[t_1, t_2, t_3, \ldots]$ [correspondingly $\mathscr{F}(t_1, t_2, t_3, \ldots)$] stands for the formula obtained from $\mathscr{F}$ by replacing each *original* occurrence of $x, y, z, \ldots$ in $\mathscr{F}$ by the terms $t_1, t_2, t_3, \ldots$ respectively.

Some people call this operation *simultaneous* or *parallel* substitution. Thus, if $\mathscr{F}[x, y]$ names "$x = y$", whereas t_1 is $y + 1$, and t_2 is 5, then $\mathscr{F}[t_1, t_2]$ is "$y + 1 = 5$" and *not* "$5 + 1 = 5$". The latter result would have been obtained if we first substituted

[17] For example, in $x = x$ the variable x has two occurrences.

[18] A fact demonstrated strongly by the explicit form of the sum, $1 + 2 + 3$.

[19] "Some" includes "none" and "all" as special cases.

t_1 in x to obtain $y + 1 = y$, and *then* substituted t_2 in y to obtain $5 + 1 = 5$. If we are to do this "simultaneous substitution" right, then we must not substitute t_2 into the y to the left of "$=$"; this y is *not* "original".

Observe also that if x does *not* occur in $\mathscr{F}[x]$, then $\mathscr{F}[t]$ is just the *original* $\mathscr{F}$.

Before we turn to the *meaning* (and naming) of the connectives and quantifiers, let us agree that we can get away with much fewer brackets than Definition 1.1.1.3 prescribes. The procedure to do so is to *agree* on connective and quantifier "priorities" so that we know, in the absence of brackets, which of the two connectives/quantifiers is supposed to "win" if they both compete to apply on the same part in a formula.

By analogy, a high school student learns the convention that "$\times$ has a higher priority than $+$", thus $2 + 3 \times 4$ means $2 + (3 \times 4)$—that is, $\times$ rather than $+$ claims the part "3".

Our convention is this: The connective $\neg$ as well as the quantifiers $\forall$ and $\exists$ have the highest priority, equal among the three. In order of decreasing priority, the remaining *binary* connectives[20] are listed as $\wedge, \vee, \to, \equiv$. If two binary connectives compete to glue with a subformula, then the higher-priority one wins. For example, assuming that $\mathscr{A}$ has already in place all the brackets that are prescribed by Definition 1.1.1.3, then $\ldots \to \mathscr{A} \vee \cdots$ means $\ldots \to \Big(\mathscr{A} \vee \cdots$, while $\ldots \neg\mathscr{A} \wedge \cdots$ means $\ldots \Big(\neg\mathscr{A}\Big) \wedge \cdots$.

If *two instances of the same binary connective* compete to glue with a subformula, then *the one to the right* wins. For example, assuming that $\mathscr{A}$ has all the brackets prescribed by Definition 1.1.1.3 in place, then $\ldots \to \mathscr{A} \to \cdots$ means $\ldots \to \Big(\mathscr{A} \to \cdots$.

Similarly, if any of $\neg, \forall, \exists$ compete for a part of a formula, again *the one to the right* wins. E.g., $\ldots\neg(\forall x)(\exists y)\mathscr{A} \cdots$ means $\ldots \Bigg(\neg\bigg((\forall x)\Big((\exists y)\mathscr{A}\Big)\bigg)\Bigg) \cdots$, where once again we assumed that $\mathscr{A}$ has all the brackets prescribed by Definition 1.1.1.3 already in place.

How do we "compute" the truth or falsehood of a formula? To begin with, to succeed in this we must realize that just as a function gives, in general, different outputs for different inputs, in the same way the "output" of a formula, its *truth-value*, can only be computed, in general, if we "freeze" the input variables. For each such frozen instance of the input side, we can compute the output side: true or false.

But where do the inputs come from? For areas of study like calculus or arithmetic the answers are easy: From the set of real numbers—denoted by $\mathbb{R}$—and the set of natural numbers respectively.

For set theory it sounds easy too: From the set of all sets!

If it were not for the unfortunate fact that "the set of all sets" does not exist, or, to put it differently, it is a *non-set class* due to its enormity, we could have left it

[20] "Binary" since they each glue *two* subformulae.

at that. To avoid paradoxes such as "a set that is *not* a set"—cf. Section 1.3 on diagonalization for an insight into why some collections cannot be sets—we will want to take our inputs from a (comfortably large) *set* in any given set-theoretic discussion: the so-called *reference set* or *domain*.

1.1.1.10 Remark. The mathematician's intuitive understanding of the statement "$\mathscr{F}$ is *true* (resp. *false*)" is that "$\mathscr{F}$ is true (resp. false) for *all* the possible values of the free (input) variables of $\mathscr{F}$".

Thus, if we are working in arithmetic, "$2n + 1$ is odd" means the same thing as "it is true that, for all $n \in \mathbb{N}$, $2n + 1$ is odd". "$2^n > n$" again omits an implied prefix "it is true that, for all $n \in \mathbb{N}$". An example of a false statement *with* input variables is "$2n$ is odd". □

1.1.1.11 Definition. An *instance* of a formula $\mathscr{F}$, in symbols $\mathscr{F}'$, is a formula obtained from $\mathscr{F}$ by replacing *each* of its variables by some value from the relevant reference set.

Clearly, $\mathscr{F}'$ is variable-free—a so-called *closed formula* or *sentence*—and therefore it has a well-defined truth-value: exactly one of *true* or *false*.

Sometimes we use more explicit notation: An instance of $\mathscr{G}(x, y, z, \ldots)$ or of $\mathscr{G}[x, y, z, \ldots]$ is $\mathscr{G}(i, j, k, \ldots)$ or $\mathscr{G}[i, j, k, \ldots]$, respectively, where $i, j, k, \ldots$ are objects (constants) from the reference set.

$\mathscr{F}'$ and $\mathscr{G}'$ are *consistent* or *common* instances of $\mathscr{F}$ and $\mathscr{G}$ if *every* free variable that appears in both of the latter receives the same value in both instances. □

1.1.1.12 Example. Let $\mathscr{A}$ stand for "$x(x + 1)$ is even", $\mathscr{B}$ stand for "$2x + 1$ is even" and $\mathscr{C}$ stand for "x is even", where x is a variable over $\mathbb{N}$. Then,

$$
\begin{array}{ll}
\mathscr{A} & \text{is true,} \\
\mathscr{B} & \text{is false, and} \\
\mathscr{C} & \text{is neither true, nor false.}
\end{array}
$$

The lesson from this is that if the truth-value of a formula depends on variables, then *not true* is *not* necessarily the same as *false*. □

We will not be concerned with how the truth-value of atomic formulae is "computed"; one can think of them as entities analogous to "built-in functions" of computer programming: *Somehow, the way to compute their true/false output is a matter that a hidden procedure (alternatively, our math knowledge and sophistication) can do for us.*

Our purpose here rather is to describe how the *connectives and quantifiers* behave toward determining the truth-value of a complex formula.

In view of Remark 1.1.1.10, the road toward the semantics of $\mathscr{A} \vee \mathscr{B}$, $((\forall x)\mathscr{A})$, etc., passes through the semantics of arbitrary instances of these; namely, we need to only define the meaning of $\mathscr{A}' \vee \mathscr{B}'$, $((\forall x)\mathscr{A})'$, etc., respectively.

1.1.1.13 Definition. (Computing with Connectives and Quantifiers) Let $\mathscr{A}$ and $\mathscr{B}$ be any formulae, and $\mathscr{A}'$ and $\mathscr{B}'$ be arbitrary *common* instances (1.1.1.11).

(1) $\neg\mathscr{A}'$—pronounced "not $\mathscr{A}'$"—is true *iff*[21] $\mathscr{A}'$ is false.

(2) $\mathscr{A}' \vee \mathscr{B}'$—pronounced "$\mathscr{A}'$ or $\mathscr{B}'$"—is true iff either $\mathscr{A}'$ is true or $\mathscr{B}'$ is true, or *both* (so-called *inclusive or*).

(3) $\mathscr{A}' \wedge \mathscr{B}'$—pronounced "$\mathscr{A}'$ and $\mathscr{B}'$"—is true iff $\mathscr{A}'$ is true *and* $\mathscr{B}'$ is true.

(4) $\mathscr{A}' \rightarrow \mathscr{B}'$—pronounced "if $\mathscr{A}'$, then $\mathscr{B}'$"—is true iff either $\mathscr{A}'$ is false *or* $\mathscr{B}'$ is true, or *both.*[22]

(5) $\mathscr{A}' \equiv \mathscr{B}'$—pronounced "$\mathscr{A}'$ iff $\mathscr{B}'$"—is true just in case[23] $\mathscr{A}'$ and $\mathscr{B}'$ are both true or both false.

(6) The instance $(\forall x)\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$—which is pronounced "for all x, $\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ (holds)"[24]— is true iff, for *all* possible values k of x from the domain, $\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n)$ is true.

(7) The instance $(\exists x)\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ —which is pronounced "for some x, $\mathscr{A}(i_1, \ldots, i_m, x, j_1, \ldots, j_n)$ (holds)"—is true iff, for *some* value k of x from the domain, $\mathscr{A}(i_1, \ldots, i_m, k, j_1, \ldots, j_n)$ is true. □

1.1.1.14 Remark. (Truth Tables) The content of the preceding definition—cases (1)–(5)—is normally captured more visually in table form (we have removed the primes for readability):

$\mathscr{A}$	$\mathscr{B}$	$\neg \quad \mathscr{A}$	$\mathscr{A} \vee \mathscr{B}$	$\mathscr{A} \wedge \mathscr{B}$	$\mathscr{A} \rightarrow \mathscr{B}$	$\mathscr{A} \equiv \mathscr{A}$
f	**f**	**t**	**f**	**f**	**t**	**t**
f	**t**	**t**	**t**	**f**	**t**	**f**
t	**f**	**f**	**t**	**f**	**f**	**f**
t	**t**	**f**	**t**	**t**	**t**	**t**

We read the table as follows: First, the symbols **t** and **f** stand for the values "true" and "false" respectively. Second, the two columns to the left of the vertical line || give all possible pairs of values (outputs) of $\mathscr{A}$ and $\mathscr{B}$. Third, below $\neg\mathscr{A}$, $\mathscr{A} \vee \mathscr{B}$, etc., we list the computed truth-values (of the formulae of the first row) that correspond to the assumed $\mathscr{A}$ and $\mathscr{B}$ values.

The odd alignment under $\neg\mathscr{A}$ is consistent with all the others: It emphasizes the placement of the "result" under the "operator"—here $\neg$—that causes it. □

[21] *if and only if*

[22] Other approaches to "implication" are possible. For example, the *Intuitionists* have a different understanding for $\rightarrow$ than that of the majority of mathematicians, who adopt the classical definition above.

[23] A synonym of "iff".

[24] The verb "holds" means "is true".