



Mathematical Logic

George Tourlakis

York University

Department of Computer Science and Engineering

Toronto, Ontario, Canada



WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

This page intentionally left blank

Mathematical Logic

This page intentionally left blank

Mathematical Logic

George Tourlakis

York University

Department of Computer Science and Engineering

Toronto, Ontario, Canada



WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2008 by John Wiley & Sons, Inc. All rights reserved.

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

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations 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 or for technical support, 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 may not be available in electronic format. For information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Tourlakis, George J.

Mathematical logic / George Tourlakis.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-28074-4 (cloth)

I. Logic, Symbolic and mathematical—Textbooks. I. Title.

QA9.T68 2008

511.3—dc22

2008009433

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To those who taught me

This page intentionally left blank

CONTENTS

Preface	xi
Acknowledgments	xvii

PART I BOOLEAN LOGIC

1	The Beginning	3
1.1	Boolean Formulae	8
1.2	Induction on the Complexity of WFF: Some Easy Properties of WFF	17
1.3	Inductive Definitions on Formulae	22
1.4	Proofs and Theorems	37
1.5	Additional Exercises	48
2	Theorems and Metatheorems	51
2.1	More Hilbert-Style Proofs	51
2.2	Equational-Style Proofs	60
2.3	Equational Proof Layout	63
2.4	More Proofs: Enriching Our Toolbox	66
		vii

2.5	Using Special Axioms in Equational Proofs	76
2.6	The Deduction Theorem	81
2.7	Additional Exercises	86
3	The Interplay between Syntax and Semantics	89
3.1	Soundness	90
3.2	Post's Theorem	93
3.3	Full Circle	99
3.4	Single-Formula Leibniz	100
3.5	Appendix: Resolution in Boolean Logic	104
3.6	Additional Exercises	107
PART II PREDICATE LOGIC		
4	Extending Boolean Logic	113
4.1	The First-Order Language of Predicate Logic	115
4.2	Axioms and Rules of First-Order Logic	138
4.3	Additional Exercises	148
5	Two Equivalent Logics	151
6	Generalization and Additional Leibniz Rules	155
6.1	Inserting and Removing “ $(\forall x)$ ”	155
6.2	Leibniz Rules that Affect Quantifier Scopes	165
6.3	The Leibniz Rules “8.12”	168
6.4	Additional Useful Tools	170
6.5	Inserting and Removing “ $(\exists x)$ ”	178
6.6	Additional Exercises	187
7	Properties of Equality	191
8	First-Order Semantics—Very Naïvely	195
8.1	Interpretations	196
8.2	Soundness in Predicate Logic	201
8.3	Additional Exercises	208
Appendix A: Gödel's Theorems and Computability		211
A.1	Revisiting Tarski Semantics	212

A.2	Completeness	223
A.3	A Brief Theory of Computability	232
A.3.1	A Programming Framework for Computable Functions	233
A.3.2	Primitive Recursive Functions	243
A.3.3	URM Computations	254
A.3.4	Semi-computable Relations; Unsolvability	259
A.4	Gödel's First Incompleteness Theorem	263
A.4.1	Supplement: $\phi_x(x) \uparrow$ Is First-Order Definable in \mathfrak{N}	275
References		281
Index		285

This page intentionally left blank

Preface

This volume is about the foundation of mathematics, the way it was conceptualized by Russell and Whitehead [56], Hilbert (and Bernays) [22], and Bourbaki¹ [2]: *Mathematical Logic*. This is the discipline that, much later, Gries and Schneider [17] called the “glue” that holds mathematics together.

Mathematical logic, on one hand, *builds* the tools for mathematical reasoning with a view of providing a *formal* methodology—i.e., one that relies on the *form* or *syntax* of mathematical statements rather than on their meaning—that is meant to be *applied* for constructing mathematical arguments that are correct, well documented, and therefore understandable.

On the other hand, it studies the interplay between the written structure (syntax) of mathematical statements and their meaning: Are the theorems that we prove by pure syntactic manipulation true under some reasonable definition of *true*? Are there any true mathematical statements that our tools *cannot* prove? The former question will be answered in the affirmative later in this book, while the latter question, interestingly, has both “no” (Gödel’s completeness theorem [15]) and “yes” (Gödel’s

¹“Nicolas Bourbaki” is the pen-name of a team of top mathematicians who are responsible for the monumental work, “Éléments de Mathématique”, which starts with logic as *the foundation*, or “connecting glue” in the words of [17], and then proceeds to extensively cover fields such as set theory, algebra, topology, analysis, measure, and integration.

(first)incompleteness theorem [16]) answers!² Both of these answers are carefully reconstructed in the Appendix to Part II.

Much has been written on logic, which is nowadays a mature mathematical body of knowledge and research. The majority of books written with upper-level undergraduate audiences (and beyond) in mind deal mostly with the *metamathematics* or *metatheory* of mathematical logic; that is, they view logic as a mathematical object and study its abilities and limitations (such as incompleteness), and the theory of models, giving short shrift to the issue of *using* logic as a tool.

There are currently only two books that the author is aware of that chronologically precede this volume and address almost exclusively the interests and needs of the *user* of logic. Both present the subject as a set of tools with which one can do mathematics (or computer science, or philosophy, or anything else that requires reasoning) rigorously, correctly, formally, and with adequate documentation: [2] and [17].

The former tersely introduces logic in its first chapter with a view of applying it as a rigorous tool for *theorem generation* in the numerous (and very advanced) chapter-volumes that follow (from set theory and algebra to topology, and measure and integration).

The latter, a much more recent entry in the literature, is an elementary text (aimed at undergraduate university curricula in computer science) in the same spirit as Bourbaki's, which proposes to use logic, once again, as a tool to prove theorems of interest to computer scientists. Indeed, the second part of [17] is on *discrete mathematics* in the sense that this term, more or less, is understood by most computer science departments today.

Similarly, the volume in your hands aims to thoroughly teach the *use* of logic as a tool for reasoning appropriate for upper-level undergraduate university students in fields of study such as computer science, mathematics, and philosophy. For the first group, this is an introduction to *formal methods*—a subject that is often included in computer science curricula—providing the student with the *tools*, the *methodology*, and a solid grounding on *technique*. As the student advances along the computer science curriculum, this volume's toolbox on formal methods will find serious applications in courses such as design and analysis of algorithms, theory of computation, computational complexity, software specification and design, artificial intelligence, and program verification.

The second group's curriculum, at the targeted level, in addition to a solid course on the use of logic, will normally also require a more ambitious inquiry into the

²It is not that Gödel was of two minds on the issue. Rather, the question can be made precise in two different ways, and, correspondingly, one gets two different answers. One way is to think of "universal" truth, such as the truth of " $x = x$ ". Universal truth is completely certifiable by the syntactic tools. The other is to think of truth in the "standard models" of some "rich" theories—rich in what one can formulate and prove in them, that is. Formal (Peano) arithmetic—that is, the axiomatic system that attempts to explain the *set of natural numbers and the arithmetic operations and relations on it*, the standard model—is such a rich theory. Gödel showed the existence of true arithmetical statements in the model that cannot be syntactically proved in the axiom system of Peano arithmetic. One such true statement says, "I am not a theorem."

capabilities and limitations of logic viewed as a mathematical tool. This further trip into the metatheory of logic will traditionally want to delve into two foundational gems beyond those of *soundness* and propositional *completeness*. Both are due to Gödel, namely, his completeness and incompleteness theorems. The Appendix to Part II settles the former in full detail, and also offers a proof of the latter (actually, only of the *first* incompleteness theorem³), basing it on an inherent limitation of “general” models of computation, in the process illuminating the connection between the phenomena of *uncomputability* and *unprovability*.

As a side-effect of constructing a self-contained proof of the first incompleteness theorem, we had to develop a fair amount of *computability* theory that will be of direct interest to all the readers, in particular, those in computer science.

The third group of readers, philosophy majors, traditionally require less coverage in a course in logic than what I have presented here; however, philosophy curricula often include a course in symbolic logic at an advanced undergraduate level, and this volume will be an apt companion for such studies.

The book’s aim to teach the practice of logic dictates that it must look and feel much like a serious text on programming. In fact, I argue at the very beginning of the first chapter, that learning and practicing logic is a process like that of learning and practicing programming. As a result, the emphasis is on presenting a multitude of tools, and on *using* these tools in many fully written and annotated proofs, an approach that is intended to enhance the reader’s effectiveness as a “prover”, giving him⁴ many examples for emulation.

There are some important differences—despite the superficial similarities that the common end-aims impose—between the approach and content in this volume and that in its similarly aimed predecessors [2] and [17].

Bourbaki provides tools for use by the “practicing mathematician” and does not bother with any semantic issues, presumably on the assumption that the mathematician knows full well how the syntactic and semantic notions interact and relate, and has an already well developed experience and ability to use *semantic methods* toward finding *counterexamples* when needed. He merely introduces and uses the so-called Hilbert style of proofs (cf. 1.4.12) that is most commonly used by mathematicians.

The text of [17] is equally silent about the interplay between syntax and semantics, and about any aspect of the metatheory, and refers to Hilbert-style proofs only tangentially. The authors prefer to exclusively propound the *equational* (or *calculational*) proof style (cf. Section 2.2), originally proposed in [11]. Moreover, unlike [2], they take liberties with their formalism.⁵ For example, even though they argue in their introduction in favor of using *formal methods* in practical reasoning, they distance themselves from a true syntactic approach, especially in their Chapter 8, where facts

³The second incompleteness theorem, that the freedom of contradiction of “rich” axiomatic systems such as Peano arithmetic *cannot* be proved “from within”, is beyond the scope of this volume. Indeed, the only complete proofs in print for this result are found in [22], Vol. II, and in [53].

⁴*His, him, he* and related terms that grammatically indicate gender are, by definition, gender neutral in this volume.

⁵A *formalism* in the context of mathematical logic is any particular way logicians structure their formal methods.

outside logic taken from algebra and number and set theory are presented as axioms of predicate logic.

While the approach in this volume is truly formal, just like Bourbaki's, it is not as terse; we are guilty of the opposite tendency! We also believe that, unlike the seasoned practitioner, the undergraduate mathematics, computer science, and philosophy students need some *reassurance* that the form-manipulation proof-writing tools presented here indeed prove (mathematical) "truths", "all truths", and "nothing but truths". This means that we cannot run away from the most basic and fundamental metatheoretical results. After all, every practitioner needs to know a few things about the properties of his tools; this will make him more effective in their use.

Thus I include proofs of the *soundness* (meta)theorems for both propositional and predicate logics (this addresses the "truths", and "nothing but truths" part) and also the two "completeness" results, of propositional and predicate logics (this is the "all truths" part). However, to maintain both the emphasis on the use of logic and an elementary but rigorous flow of exposition I have delegated the much-harder-to-prove completeness metatheorem of predicate logic ([15]) to a sizable appendix at the end of the book.

Why are soundness and completeness relevant to the needs of the user? Completeness of propositional logic, along with its soundness, give us the much-needed—in the interest of user-friendliness—license to mix semantic and syntactic tools in formal proofs *without sacrificing mathematical rigor*. Indeed, this license (to use *propositional* semantic tools) is extended even in predicate logic, and is made possible by the trick of adding and removing quantifiers ("for all" and "for some"). On the other hand, soundness of the two logics allows the user to *disprove* statements by constructing so-called *countermodels*.




There are also quite a few simpler metatheoretical results, beyond soundness and completeness, that we routinely introduce and prove as needed about formulae (e.g., about their syntax) and about proofs (e.g., the validity of principles of proof such as *hypothesis strengthening*, *deduction theorem*, and *generalization*), using the basic tool of induction (essentially on formula and proof lengths).

The Hilbert style of proving theorems is prevalent in the mathematical literature and is prominently displayed and practiced in this volume. On the other hand, the equational-style of displaying proofs has been gaining in popularity especially in computer science curricula. It is a style of proof that seems well adapted to areas in computer science such as software engineering (in particular, in the field of software engineering *requirements*) and program verification.

For the above reason, equational-style proofs receive a thorough exposition in this volume. It is my intention to endow the reader with enough machinery that will make him proficient in *both* styles of proof, but more importantly, will enable him to choose the style that is best suited to writing a proof for any particular theorem.

In terms of prior knowledge (tools) needed to cope with this volume the reader should at least have high school mathematics (but I expect that this includes mathematical induction and some basic algebra). A degree of mathematical maturity, but no specific additional knowledge, of the kind an upper-level undergraduate will normally have will also be handy.

A word on pedagogical approach. I repeatedly taught the material included here to undergraduate computer science students at York University in Toronto, Canada. I think of this book as the record of my lectures. I have endeavored to make these lectures user-friendly, and therefore accessible to readers who do not have the benefit of an instructor's guidance. Devices to that end include anticipation of questions, promptings for the reader to rethink an issue that might be misunderstood if glossed over ("pauses"), numerous remarks and examples that reflect on a preceding definition or theorem.

Using the symbols , and  , I am marking those passages that are very important, and those that can be skipped at first reading, respectively.

My fondness for footnotes is surely evident (a taste acquired long ago, when I was studying Wilder's excellent *Introduction to the Foundations of Mathematics* ([57]).

I give (mostly) *very* detailed proofs, as I know from experience that omitting details normally annoys students. Moreover, I have expectations that students will achieve a certain style, and effectiveness, in proofs. The best way to teach them to do so is by repeatedly giving examples how. In turn, students will have the opportunity to test and further their understanding by doing several exercises, some of which are embedded in the text while others appear at chapters' end (a total of more than 190 exercises).

Book structure. The book is in two approximately equal-length parts, one on Boolean (or propositional) logic and one on predicate logic. A thorough exposition of Boolean logic pedagogically prepares the reader for the much more difficult predicate logic, at the same time endowing him with several tools that are transferable such as the ubiquitous *Post's theorem* (propositional completeness) and *deduction theorem*.

Part I is in three chapters. Chapter 1 starts with the basic formation rules of propositional (Boolean) formulae—the syntax—and introduces “induction on formulae” as a tool via which we can prove facts about syntax. It proceeds with Boolean semantics (truth tables) and then continues with the concept of formal proofs—those effected via purely syntactic manipulation—from axioms and rules of inference. Chapter 2 is a veritable database of proofs and theorems, presenting several proofs and proof techniques, including the *deduction theorem*. Both the equational and Hilbert style of proof layouts are used extensively. Chapter 3 revisits semantics, and proves both the soundness and completeness (Post) theorems, thus demonstrating the full equivalence and interchangeability of the semantic and syntactic proof techniques in Boolean logic. It concludes with an exposition of the technique of *resolution* in Boolean logic.

Part II on predicate logic (or calculus) contains five chapters and a lengthy Appendix. Predicate calculus is introduced as an extension of the logic of Part I, so that every tool that we obtained in Part I is still usable in Part II. This part's first chapter, Chapter 4, is about the syntax of formulae, and introduces the axioms, the rules of inference, and the concept of proof, extending without discarding anything of the corresponding concepts of Part I. Chapter 5 simplifies the metatheoretical arguments by introducing a *simpler-to-talk-about* logic, equivalent to ours; that is, a logic with a simpler metatheory. Chapter 6 proves and extensively uses powerful rules of inference that were not postulated up front: techniques for adding and removing the

universal quantifier, powerful Leibniz rules, and techniques for adding and removing the existential quantifier. Our version of predicate calculus, as is common in the literature nowadays, includes equality (=). Chapter 7 advances some basic properties of equality as these flow from the axioms and the rules of inference.

Chapter 8 is a “working” first approximation to Tarski-like semantics and proves (in detailed outline) the soundness theorem for predicate calculus. This is an important tool toward constructing counterexamples, or *countermodels* as we prefer to call them, aiming to show that certain predicate logic formulae are *not* provable.

The Appendix at the very end does several things: It revisits Tarski semantics that were naïvely presented in Chapter 8, proves soundness again, this time totally rigorously, and also proves Gödel’s completeness theorem. It then introduces *computability*, that is, the part of logic that makes the concepts of algorithm, computation, and computable function mathematically precise. In this particular approach to computability, I am using the programming language known in the literature as the Shepherdson-Sturgis ([44]) *unbounded register machines* (URMs). The topics included constitute the very foundation of the theory of computation and they will be of interest not only to mathematics readers but also to those in philosophy and, especially, in computer science, who will find ample supplemental material for their theory of computation courses. These include *partial computable functions*, *primitive recursive functions*, a complete characterization in number-theoretic terms of the partial functions computable by URMs, the *normal form theorems*, the “Kleene predicate” and a “universal” URM, computable and *semi-computable* relations and their behavior in connection with Boolean operations and quantification, *computably enumerable relations*, *unsolvability*, *verifiers* and *deciders*, *first-order definability*, and the *arithmetical relations*. This machinery will next allow us to tackle Gödel’s first incompleteness theorem. This we prove by basing the proof on the nonexistence of a URM program that solves the following problem (*halting problem*) for any choice of x and y : “Will program x ever terminate if its input is y ?”

Suggested coverage. A computer science curriculum in formal logic will probably cover everything but the Appendix. The course MATH1090 at York University, especially designed for computer science majors, does exactly that. However, a hybrid course in logic and computability, often included in computer science curricula, will adjust its pace (e.g., going faster through Part I) to include the computability and Gödel incompleteness topics of the Appendix. A mathematics major will typically see his first course in logic in an upper-undergraduate year. His syllabus will likely require that the book be studied from cover to cover (again, going fast through Part I). A philosophy major’s needs in a course in logic are harder to fit to a prescribed template. Advanced students will likely find all of Part I relevant along with chapters 4–6 of Part II. They will also find a high degree of relevance in the computability and Gödel incompleteness topics of the Appendix.

GEORGE TOURLAKIS

Acknowledgments

The writing of this book would have not been successful without the support and warmth of an understanding family. Thank you.

I also wish to thank my York colleagues, Rick Ganong, Ilijas Farah, Don Pelletier and Stefan Mykytiuk, who have used drafts of this book in their MATH1090 sections and offered me helpful suggestions for improvements. I want to credit all my students in logic courses that I taught over the years, as they have provided me not only with the pleasure of teaching them, but also with the motivation to create a friendly yet mathematically rigorous, thorough, up-to-date, and correct text for their use.

I acknowledge the ever supporting environments at York University and at the University of Crete—where I spent many long-term visits, most recently in the spring term of 2006–07 while on Sabbatical leave from York—which are eminently conducive to projects like this one.

Steve Quigley and Melissa Yanuzzi have been supportive and knowledgeable editors who made all the steps of the project smooth and effective. It has been a pleasure to work with both of them.

I finally wish to record my appreciation of Donald Knuth, Leslie Lamport, and the TUG community (<http://www.tug.org>) for the typesetting tools \TeX and \LaTeX and the UNIX-based distribution of comprehensive typesetting packages that they have made available to the technical writing community, making the writing of books such as this one fun.

This page intentionally left blank

PART I

BOOLEAN LOGIC


This page intentionally left blank


CHAPTER 1



THE BEGINNING

Mathematical logic, or as we will simply say, “logic”, is the science of mathematical reasoning. Its core consists of the study of the *form, meaning, use, and limitations* of logical deductions, the so-called *proofs*.

This volume, which is aimed at upper-level undergraduate university students who follow a course of study in computer science, mathematics, or philosophy, will emphasize mainly *the use* of proofs—it is written with the interests of the user in mind.

1.0.1 Remark. (Before we Begin) The symbol “”⁶ goes at least as far back as the writings of Bourbaki. It has been made widely accessible to authors—who like to typeset their writings themselves—through the typesetting system of Donald Knuth (known as “ \TeX ”).

I use these “road signs” as follows: A passage enclosed between two single “” symbols is purported to be *very noteworthy*, so please heed!

On the other hand, a passage enclosed between two *double* signs (“”) means two things.

⁶This symbol is a stylized typographical version of the “(dangerous) winding-road” road sign.

The *bad news* is that it is rather difficult, or esoteric, or *both*. The *good news* is that you do not *need* to understand (or even read) its contents in order to understand all that follows. It is only there in the interest of the “demanding” reader. Such “doubly dangerous” passages allow me to *digress* without injuring continuity—you can ignore these digressions! □

Learning to *use* logic, which is what this book is about, is like learning to use a *programming language*.

In the latter case, probably familiar to you from introductory programming courses, one learns the correct syntax of programs, and also learns what the various syntactic constructs do—that is, their semantics. After that, one embarks—for the balance of the programming course—on a set of increasingly challenging programming exercises, so that the student becomes proficient in programming in said language.

We will do an exactly analogous thing in this volume: We will learn to write *proofs*, which are nothing else but *annotated* sequences of formulae and are similar to computer programs in terms of syntactic structure—the annotations playing a role closely similar to that of *comments* in computer programs.

But to do that, we need to know, to begin with, what *are* the *rules* of *correctly writing down* a formula and a proof! We have to start with the *syntax* of these objects—formulae and proofs—precisely as it is done in the case of programming and its related objects, the programs.

Thus, we will begin with learning the syntax of the logical language, that is, what syntactically correct formulae and proofs *look like*. We will also learn what various syntactic constructs “say” (semantics). For example, we will learn that a formula makes a “statement”. A proof also makes a statement, that *every formula in it is true in some very intuitively acceptable sense*.

We will learn that correctly written proofs are finite and “checkable” means toward discovering mathematical “truths”. We will also learn via a lot of practice how to write a large variety of proofs that certify all sorts of useful truths of mathematics.

The above task, writing proofs—or “programming in logic” if you will—is our main aim. This will equip you with a *toolbox* that you can use to discover or certify truths. It will be handy in your studies in computer science, and in whatever area of study or research you embark upon and where reasoning is required.

However, we will also look at this toolbox, the logic, as *an object of study* and study some of its properties. After all, if you want to take up, say, carpentry, then you need to know *about* tools such as hammers—their properties (e.g., hard and heavy) and limitations (e.g., unfriendly to fingers).



When *using* the toolbox to prove theorems, you work *within* logic. On the other hand, when *studying* the toolbox, you work in logic’s *metatheory* (in metalogic) to talk and reason *about* logic.

People often do this kind of study with programming languages, looking at them as objects of study rather than as instruments to write programs with. For example, in an advanced course on the comparative study of programming languages one looks at several programming languages and compares them for features, suitability

for certain programming tasks—for any specific task some are more suitable than others—limitations, etc.

Here is another analogy: In the “real world” that we live in, one builds flight simulators, which we use to simulate flying an airplane, and in the process we learn how to do so. The real world where the simulator is *built* is the simulator’s metatheory, where we can, among other things, study the properties and limitations of simulators and compare several simulators for features such as relative “power” (i.e., how effective or realistic they are), etc. Similarly, formal logic is built within “real mathematics”, as we will see in the next section. It, too, is a “simulator” employed to write formal proofs that certify the truth of mathematical statements. These proofs imitate the kind of informal proofs one typically employs in informal mathematics but do so within a precisely specified system of notation (called *language*), rules, and assumptions. Thus, *using* formal logic is a means to *learn* how to write proofs—and not only *formal* proofs!—just as using a flight simulator is a means of learning how to fly a real plane. The metatheory of logic—the “real mathematics”—addresses questions among the deepest of which is the question of how far formal logic can go in discovering mathematical truths.



Let us next look more closely at the similarity between *programming languages and programming* on one hand and *logical languages and proving* on the other, and argue that, similar as the two activities may be, the second one is a bit easier!

- (1) In programming, you use the syntactic rules to write a program that solves a problem.
- (2) In logic, you use the syntactic rules to write a proof that establishes a theorem.

In the latter task you are done as soon as the proof ends. At the end of the proof you have your theorem, exactly as stated.

In the former task, programming, it is not enough to just write a program! You next have to convince your boss, or your instructor, that the program indeed solves the problem; that it is “semantically correct” with respect to the problem’s specification.

Note that in proving a theorem you have a purely syntactic task. Once your correctly written proof ends with the theorem you were trying to prove, you are done. There is no messing about with semantics.

There is another reason why programming is harder than proving theorems: Programming has to be painstakingly precise because it involves your writing instructions for a dumb machine to “understand” and follow. You must be absolutely and pedantically clear in your instructions.

On the other hand, you address a proof to a human who knows as much as you do, or more, about the subject. This human will in general accommodate a few shortcuts that you may want to take in your presentation.

In short, proofs are read by “intelligent” humans, while programs are read by “dumb” computers. We need to work really hard to speak at the level of the latter.

Will you ever need to deal with semantics in logic? Yes! Semantics is useful when you want to *disprove* (or *refute*) something, that is, to prove that it is a false

statement, a fallacy. We will talk about semantics later—three times: once under Boolean logic, once under predicate logic, and one last time in the Appendix.

There are many methodologies or paradigms (and corresponding programming languages suitable for the task) for writing programs. For example (add the word *programming* after each italicized keyword), *procedural* (Algol, Pascal, Turing), *functional* (LISP), *logic* (Prolog), and *object-oriented* (C++, Java). Most computer science departments will expose their students to many of the above.

Similarly there are several methodologies for writing proofs. For example (add the word *style* after each italicized keyword), *equational* (the one favored by [17]), *Hilbert* (favored by the majority of the mathematics, computer science, and logic literature), *Gentzen's natural deduction*, etc.

My aim is to assist the reader to become an able user of the first two styles: the equational and the Hilbert style of proof.

In both methodologies, an important required component is the systematic annotation of the proof steps. Such annotation explains why we do what we do, and has a function similar to that of comments in a program.

Okay; one can grant that a computer science student needs to learn programming. But logic? You see, the proper understanding of propositional logic is fundamental to the most basic levels of computer programming, while the ability to correctly use variables, scope, and quantifiers is crucial in the use of loops, and subroutines, and in software design. Logic is used in many diverse areas of computer science, including digital design, program verification, databases, artificial intelligence, algorithm analysis, computability, complexity, and software specification. Besides, any science that requires you to reason correctly to reach conclusions uses logic.

When one is learning a programming language, one often starts by learning a small *subset* of the language, just to smooth the learning curve. Analogously, we will first learn—and practice—a subset of the *logical language*. This we will do not due to some theoretical necessity, but due to pedagogical prudence. This particular, “easy” subset of (the “full”) logic that we will embark upon learning goes by many names: *Boolean logic*, *propositional logic*, *sentential logic*, *sentential calculus*, and *propositional calculus*.

The “full logic” we will call by any of the names *predicate calculus*, *predicate logic*, or *first-order logic*.

I like the *calculus* qualifier. It connotes that there is a *precise way* to “calculate” within logic. It emphasizes that building proofs is an algorithmic and precise process, just like programming.



Indeed, it turns out that you can write a program, say, in Pascal, that will accept no input, but if it is allowed to run forever it will print all the theorems of logic⁷ (and not just those of the Boolean variety)—and never print a non-theorem!—in some order, possibly with some repetitions (cf. A.4.7 on p. 270).

⁷We will soon appreciate that there are infinitely many theorems in logic.

Equivalently,⁸ we can write a program that is a *theorem verifier*. That is, given as input a theorem, the program will verify that it is so, in a finite number of steps. If the input is a non-theorem, our verifier will decline an answer—it will run forever.

Thus, proving theorems is a *mechanical process*!



Digression: The above assertion is an example of a true assertion *about* the logic, not one that we can prove *using exclusively the tools of* logic as a tool. It is a *metatheorem* of logic as we say, *not* a theorem.

The proof of this metatheorem requires techniques much more powerful than—indeed external to—those that the logic provides. We will prove this metatheorem in the Appendix to Part II (A.4.6).

So metatheorems are truths *about* the logic that we prove with tools external to the logic, while theorems are truths that the logic itself is *capable of proving*.

There is some danger that the above statement, “proving theorems is a mechanical process”, may be misinterpreted by some as one advocating that we build proofs by mindlessly shuffling symbols. *Nothing is further from reality*.

The statement must be understood precisely as written. It says that there is a “mindless” way, a programmable way, to generate and print *all possible theorems* of logic, and, equivalently, also a programmable way to *verify* all theorems, which, however, refuses to verify any non-theorem by “looping” forever when presented with any such as input.

But it is not a recipe for how *we* ought to behave when we write proofs. This is *not* the way a mathematician, or you or I, go about proving things—mindlessly. In fact, if we do not understand what is going on, we cannot go too far.

Moreover, interesting, even important, as this result (about the existence of theorem verifiers) may be *theoretically*, it is *useless practically*, as we further discuss below.

Our task is different. In general, we are more inquisitive. Given an arbitrary (mathematical) statement, we do not know ahead of time *if it is a theorem or not*. This italicized statement, the so-called *decision problem* of logic, is what we normally are interested in. Thus, our “verifier” is not very helpful, for if the statement that we present it as input is *not* a theorem, then the verifier will run forever, not giving an answer.

Hmm. Can we not write a *decider* for logic? The answer to this is interesting, but also reassuring to mathematicians (and all theorists): Their jobs are secure!

- (1) For Boolean logic, we can, since the question “Is this statement a theorem?” translates to “Is this statement a tautology?” (cf. 3.2.1). The latter can be settled algorithmically via truth tables. But there is a catch: Checking a formula (the formal counterpart of a “statement”) for tautology status is an *unfeasible* problem.⁹ So we can do it *in principle*, but this fact is devoid of any practical value.

⁸That this formulation of the claim is equivalent to the preceding one is a standard result of computability. Cf. Appendix to Part II, Remark A.3.91 on p. 262.

⁹The term *unfeasible*—also *intractable*—has a technical connotation in complexity theory: It means a problem for which we know of no algorithm that runs in polynomial time as a function of the input



(2) For predicate logic, the answer is more pleasing to mathematicians.

First, there exists *no decider* for this logic if we expand it minimally so that it can reason about the theory of natural numbers (this is Alonzo Church's theorem, [3, 4]).

Second, even if one were to be satisfied simply with a *verifier* for theorems, then we still would have *no general solution of any practical value* in hand. Indeed, again considering the logic augmented so that it can “do number theory”, *any chosen verifier V for this logic* would be extremely slow in providing answers in the following precise sense: For *any choice* of a step-counting function $f(n)$, there is an *infinite subset, S* , of the set of theorems of number theory, such that *each theorem-member, T* , of S that is composed of n symbols requires for its verification more than $f(n)$ steps to be performed by V .¹⁰ This is a result of Hartmanis ([19]).



Let us stop digressing for now. In the next section we begin the study of the sublogic known as *propositional calculus*.

1.1 BOOLEAN FORMULAE

We will continue stressing the algorithmic nature of the discipline of proving, just as it is the case in the discipline of programming.

In particular, just as in serious programming courses the *programming language* is introduced via precise *formation rules* that allow us to write syntactically correct programs, we will be every bit as serious by introducing very precisely the rules for writing syntactically correct (1) formulae and (2) proofs.

Once again, the syntax of the logical language is much simpler to describe than that of any commercially available programming language.

So, how does one build—i.e., what are the rules for writing down correctly—formulae?

Continuing with the programming analogy, you will recall that to define a programming language, i.e., the syntax of its programs, one starts with the list of admissible symbols, the so-called *alphabet*. In some languages, the alphabet includes symbols such as “3, 4, 0, [, A, B, c, d, E, +, ×, -” and “keywords”—that is, multiple-character symbols—such as *if, then, else, do, begin*.

Similarly, in Boolean logic, we start with the basic building blocks, which *collectively* form what is called the *alphabet* (for formulae). Namely,

length—or worse, we know that such an algorithm does not exist. In this case it is the former. However, there is a connection with the so-called “P vs. NP” open question (see [5]). If a polynomial algorithm that recognizes tautologies does exist, then the open problem is settled as “P = NP”, something that the experts in the field consider highly unlikely. The truth table method runs in exponential time.

¹⁰For example, consider $f(n) = 2^{2^{2^n}}$. If we think of $f(n)$, for each n , as representing picoseconds of run time of the verifier V (1 picosecond is 10^{-12} seconds), then every member of S of length more than 4 symbols will require the verifier V to run for more than 5.70045×10^{288} years!

A1. *Symbols for variables*, called the *Boolean* or *propositional* or *sentential variables*. These are p, q, r , with or without primes or subscripts (i.e., p', q_{13}, r'''_{51} are also symbols for variables).



We often need to write down expressions such as “ $A[p := B]$ ”, to be defined later (1.3.15), but do not wish to restrict them to the specific variable p . Nor can we say things such as “for any Boolean variable p we consider $A[p := B]$...” as there is only one specific p !

We get around this difficulty by employing so-called *metavariables* or *syntactic variables*—i.e., *symbols outside the alphabet that we can use to refer to or point, generally, to any variable*. We adopt the names for those to be the boldface $\mathbf{p}, \mathbf{q}, \mathbf{r}$ with or without primes or subscripts. Thus \mathbf{p}'_{01} names *any* variable p, q, r''' , q'''_{987} , etc. Rarely if ever in this volume will we need to use more Boolean metavariables than these two: \mathbf{p}, \mathbf{q} .

We can now use the expression “for every Boolean variable \mathbf{p} we consider $A[\mathbf{p} := B]$...” referring to what \mathbf{p} *names* rather than to \mathbf{p} itself. Two analogous examples are, from algebra, “for every natural number n ” (n is not a natural number!) and, from programming, where we might say about Algol, “for each variable \mathbf{x} , the instruction $\mathbf{x} := \mathbf{x} + 1$ means to increase the value of \mathbf{x} by one.” Again, \mathbf{x} is not a variable of Algol; $X13, YXZ99$, though, are. But it would be meaningless to offer the general statement “for each variable $X13$, the instruction $X13 := X13 + 1$, etc.” since $X13$ is a *specific* variable of the Algol syntax. The programming language metavariable \mathbf{x} allows us to speak of all of Algol’s variables collectively!

On the other hand, the expression “for every Boolean *metavariable*” refers to the set of metavariables themselves, $\{\mathbf{p}, \mathbf{q}, \mathbf{r}'_{00}, \dots\}$ and will be rarely, if ever, used. The expression “for every Boolean *metavariable* \mathbf{p} ” is as nonsensical as “for every Boolean variable p ”.



A2. Two *symbols for Boolean constants*, namely \top and \perp . These are pronounced variously in the literature: *verum* (also *top*, or *symbol “true”*) and *falsum* (also *bottom*, or *symbol “false”*¹¹).

A3. Brackets, namely, (and).

A4. “Boolean connectives”, namely, the symbols listed below, separated by commas

$$\neg, \wedge, \vee, \rightarrow, \equiv \tag{i}$$

Let us denote by \mathcal{V} the alphabet consisting of the symbols described in **A1–A4**.

¹¹Usually, the qualifier *symbol* is dropped and then the context is called upon to distinguish between “true/false” the *symbols* vs. “true/false” the *Boolean values* of the metatheory (introduced in Section 1.3). In particular, cf. Definition 1.3.2 and Remark 1.3.3.

1.1.1 Remark. (1) Even though I say very emphatically that p, q, r , etc., and also \top and \perp , are just *symbols*,¹²—the former standing for variables, the latter for constants—yet, I will stop using the qualification *symbols*, and just say *variables* and *constants*. This entails an agreement: I always *mean* to say *symbols*, I just don't say it.

(2) Most variable symbols are formed through the use of “subsymbols”—such as $0, 1, 2, '—$ that are not members of the alphabet \mathcal{V} themselves; e.g., p''''_{110034} . This does not detract from the fact that each variable (name) is a *single* symbol of \mathcal{V} , entirely analogously with, say, the keywords of Algol *if, then, begin, for*, etc.

(3) Readers who have done some elementary course in logic, or in the context of a programming course, may have learned that \neg, \vee are the only connectives one really *needs* since the rest can be expressed in terms of these two. Thus we have deliberately introduced redundancy in the adopted set of connectives (*i*) above. This choice in the end will prove to be user-friendly and will serve our aim to give a prominent role to the connective \equiv , in the axioms and in rules of inference (Section 1.4). \square

1.1.2 Definition. (Strings or Expressions; Substrings) We call a *string* (also *word* or *expression*), over a given alphabet, any *ordered* sequence of the alphabet's symbols, written adjacent to each other *without* any visible separators (such as spaces, commas, or the like).

For example, *aabba* is a string of symbols over the alphabet $\{a, b, c, 0, 1, 2, 3\}$ (note that you don't have to use all the alphabet symbols in any given string, and, moreover, repetitions are allowed). *Ordered* means that *the position of symbols in the string matters*; e.g., $aab \neq aba$.

We denote arbitrary strings over the alphabet **A1–A4** by *string variables*, i.e., names that stand for arbitrary¹³ or specific¹⁴ strings. Specific strings, or string constants, are sometimes enclosed in double quotes to avoid ambiguity. For example, if we say

Let A be the string aab.

we need to know whether the period is part of the string or not. If it's *not* we symbolically indicate so by writing

Let A be the string “aab”.

If it were part of the string, then we would have written instead

Let A be the string “aab.”.

String variables—by agreement—will be denoted by uppercase letters $A, B, C, D, E, P, Q, R, S, W$ etc., with or without primes or subscripts. In particular, since Boolean expressions (and theorems) are strings, this naming is valid for this special case, too.

¹²Some logicians put it more emphatically: “*meaningless symbols*”.

¹³E.g., “let A be any string”.

¹⁴E.g., “let A stand for $(\neg(p \wedge q))$ ”.